The Design of a Task Parallel Library

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt
Microsoft Research
{daan,schulte,sburckha}@microsoft.com

Abstract

The Task Parallel Library (TPL) is a library for .NET that makes it easy to take advantage of potential parallelism in a program. The library relies heavily on generics and delegate expressions to provide custom control structures expressing structured parallelism such as map-reduce in user programs. The library implementation is built around the notion of a *task* as a finite CPU-bound computation. To capture the ubiquitous apply-to-all pattern the library also introduces the novel concept of a replicable task. Tasks and replicable tasks are assigned to threads using work stealing techniques, but unlike traditional implementations based on the THE protocol, the library uses a novel data structure called a 'duplicating queue'. A surprising feature of duplicating queues is that they have sequentially inconsistent behavior on architectures with weak memory models, but capture this non-determinism in a benign way by sometimes duplicating elements. TPL ships as part of the Microsoft Parallel Extensions for the .NET framework 4.0, and forms the foundation of Parallel LINQ queries (however, note that the productized TPL library may differ in significant ways from the basic design described in this article).

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Concurrent Programming Structures

General Terms Languages, Design

Keywords Domain specific languages, parallelism, work stealing

1. Introduction

Many existing applications can be naturally decomposed so that they can be executed in parallel. Take for example the following (naïve) method for multiplying two matrices:

```
void MatrixMult(int size, double[,] m1
                ,double[,] m2, double[,] result){
  for(int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
      result[i, j] = 0;
      for(int k = 0; k < size; k++){
        result[i, j] += m1[i, k] * m2[k, j];
      }
    }
  }
}
```

In this example, the outer iterations are independent of each other and can potentially be done in parallel. A straightforward way to parallelize this algorithm would be to use `size` threads, where each thread would execute the two inner loops with its corresponding iteration index — but wait: that would be prohibitively expensive, since each thread needs a stack and other book keeping information. It is better to use a small pool of threads, and assign to them a set of tasks to execute, where a *task* is a finite CPU bound computation, like the body of a for loop. So instead of `size` threads we should create `size` tasks, each of them executing the two inner loops for its iteration index. The tasks would be executed by n threads, where n is typically the number of processors. Using tasks instead of threads has many benefits – not only are they more efficient, they also abstract away from the the underlying hardware and the OS specific thread scheduler.

The Task Parallel Library (TPL) enables programmers to easily introduce this potential task parallelism in a program. Using TPL we can replace the outer for loop of the matrix multiplication with a call to the static `Parallel.For` method:

```
void ParMatrixMult(int size, double[,] m1
                   ,double[,] m2, double[,] result){
  Parallel.For(0, size, delegate(int i){
    for(int j = 0; j < size; j++){
      result[i, j] = 0;
      for(int k = 0; k < size; k++){
        result[i, j] += m1[i, k] * m2[k, j];
      }
    });
  });
}
```


OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

The `Parallel.For` construct is a static method of the `Parallel` class that takes three arguments: the first two arguments specify the iteration domain (between `0` and `size`), and the last argument is a delegate expression that is called for each index in the domain. The delegate represents the task to execute. A delegate expression is the C# equivalent of an anonymous function definition or lambda expression. In this case, the delegate takes the iteration index as its first argument and executes the *unchanged* loop body of the original algorithm. Note that in C# the delegate passed to the `Parallel.For` loop automatically captures `m1`, `m2`, `result`, and `size`. This automatic capture of free variables makes it particularly easy to experiment with introducing concurrency in existing programs. In Section 6.1 we show to write a parallel matrix computation by hand without using the TPL and compare its performance.

TPL can be viewed as a small *embedded domain specific language* and its methods behave like custom control structures in the language (Hudak 1996). We argue that the necessary ingredients for such library approach in strongly typed languages are parametric polymorphism (generics) and first-class anonymous functions (delegates).

Moreover, through the abstraction provided by these two ingredients, we can implement all the high level control structures in terms of just two primitive concepts – *tasks and replicable tasks*. This guarantees that the library abstractions have consistent semantics and behave regularly.

The parallel abstractions offered by the library only *express* potential parallelism, but they do *not guarantee* it. For example, on a single-processor machine, `Parallel.For` loops are executed sequentially, closely matching the performance of strictly sequential code. On a dual-core machine, however, the library might use two worker threads to execute the loop in parallel, depending on the workload and configuration. Since no concurrency is guaranteed, the library is specifically intended for speeding up CPU bound computations, and not for asynchronous programming. TPL also does not help to correctly synchronize parallel code that uses shared memory. Other mechanisms, such as locks, are needed to protect concurrent modifications to shared memory, and it is the programmer's responsibility to ensure that the code can be safely executed in parallel.

Under the hood, tasks and replicable tasks are assigned by the runtime to special worker threads. TPL uses standard *work-stealing* for this work distribution (Frigo et al. 1998) where tasks are held in a thread local task queue. When the task queue of a thread becomes empty, the thread will try to steal tasks from the task queue of another thread. The performance of work stealing algorithms is in a large part determined by the efficiency of their task queue implementations.

We provide a novel implementation of these task queues, called *duplicating queues*. A surprising feature of the duplicating queue is that it behaves in a sequentially inconsistent way on weak memory models. However, the non-determinism of the queue is carefully captured in a benign way by sometimes duplicating elements in the queue (but never losing or inventing elements, and still ensuring that tasks are only executed once). Of course, exploiting weak memory models is playing with fire. That's why we verified the correctness of the duplicating queue formally using the Checkfence tool (Burckhardt et al. 2007).

Initially developed at Microsoft Research, TPL is shipping as a major part of the Microsoft Parallel Extensions to the .NET framework and currently available for download as a Community Technology Preview (Microsoft 2008). TPL forms the foundation of Parallel LINQ queries and has already been used for parallelizing various applications (we will report on some of them). Note though that the productized library has changed in many ways including the integration with the new thread pool implementation of .NET 4.0. Consequently the design of TPL has evolved, and the productized library may differ in significant ways from the library as described in this paper (as an example, the productized TPL also contains support for asynchronous programming and I/O bound computations).

To summarize, we make the following main contributions:

- We show how we can embed a realistic parallel framework as a library in a strongly typed language, where the combination of generics and delegates proved vital to define custom control structures.
- We show that the library needs just two primitives: tasks and replicable tasks, where the latter forms a convenient abstraction to capture parallel iteration and aggregation.
- A duplicating queue is a novel data structure for implementing non-blocking task queues that can work efficiently on architectures with weak memory models. These queues are sequentially inconsistent but capture the resulting non-determinism in a benign way by sometimes duplicating elements.
- We verified the correctness of the duplicating queue formally using Checkfence.
- We provide a detailed assessment of TPLs performance and discuss its use in a Microsoft product.

The rest of the paper is structured as follows: § 2 introduces example abstractions for parallel programming provided by the TPL; § 3 defines these abstraction in terms of the underlying task primitives; § 4 describes TPL's strategy for work distribution; § 5 introduces the duplicating queue and provides correctness arguments; § 6 reports on experience by early adopters and provides performance numbers; § 7 discusses related work; § 8 concludes.

2. Using the TPL

This section introduces some basic abstractions for parallelism in the TPL. One of the most basic patterns is standard

fork-join parallelism. As an example, consider the following sequential quicksort implementation:

```
static void SeqQuickSort<T>(T[] dom, int lo, int hi)
  where T : IComparable<T>
{
  if(hi - lo <= Threshold)
    InsertionSort(dom, lo, hi);

  int pivot = Partition(dom, lo, hi);
  SeqQuickSort(dom, lo, pivot - 1);
  SeqQuickSort(dom, pivot + 1, hi);
}
```

The algorithm is generic in the element type `T` and only requires that they can be compared. Under a certain threshold, the algorithm falls back on insertion sort which performs better for a small number of elements. Otherwise, we partition the input array in two parts and quick sort both parts separately. These two sorts can be performed in parallel since each sort works on a distinct part of the array. We can express this conveniently using the `Parallel.Do` method:

```
static void ParQuickSort<T>(T[] dom, int lo, int hi)
     where T : IComparable<T>
{
  if(hi - lo <= Threshold)
    InsertionSort(dom, lo, hi);

  int pivot = Partition(dom, lo, hi);
  Parallel.Do(
    delegate{ ParQuickSort(dom, lo, pivot - 1); },
    delegate{ ParQuickSort(dom, pivot + 1, hi); }
 );
}
```

The `Parallel.Do` method is a static method that takes two or more delegates as arguments and potentially executes them in parallel. Since quick sort is recursive, a lot of parallelism is exposed since every invocation introduces more parallel tasks. Depending on the available number of threads however, the library might actually execute only some of the tasks in parallel and the remaining ones sequentially. We will later investigate why that is a good strategy.

The above code also shows how TPL acts as an embedded domain specific language: the `Parallel.Do` method is very close to extending the language with a parallel **do** statement. Having first-class anonymous functions is essential to write a parallel do conveniently. C# automatically captures the free variables `dom`, `lo`, `pivot`, and `hi`. In Java programers need to simulate anonymous functions using inner classes which is less convenient and only variables marked as **final** can be captured. In C++, the situation is very cumbersome as we need to introduce a new class with the free variables as its fields.

2.1 Map-reduce parallelism

Often, a **for** loop is used to iterate over a domain and aggregate the values into a single result. Take for example the following iteration that sums the prime numbers less than 10000:

```
int sum = 0;
for(int i = 0; i < 10000; i++){
  if(isPrime(i))
     sum += i;
}
```

Unfortunately we cannot parallelize this loop as it stands since the iterations are not independent due to the modification of the shared `sum` variable. A correct parallel version would protect the addition with a lock, as in:

```
int sum = 0;
Parallel.For(0, 10000, delegate(int i){
  if(isPrime(i)){
    lock (this){ sum += i; }
  }
});
```

This program now suffers from a performance problem since all parallel iterations contend for both the same lock and the same memory location. We can avoid this if each worker thread maintains a thread local sum and only adds to the global sum at the end of the loop. In TPL this particular pattern is captured by the `Parallel.Aggregate` operation, and we can rewrite the example as:

```
int sum = Parallel.Aggregate(
  0, 10000,   // domain
  0,          // initial value
  delegate(int i){ return (isPrime(i) ? i : 0) },
  delegate(int x, int y){ return x+y; }
);
```

The aggregate operation takes five arguments. The first two specify the input domain, which can also be an enumerator. The next argument is the initial value for the result. The next two arguments are two delegate functions. The first function is applied to each element, and the other is used to combine the element results. The library will automatically use a thread local variable to compute the thread local results without any locking, only using a lock to combine the final thread local results. If the aggregation is done in parallel, it is possible that elements are combined in a different order than a sequential aggregation. Therefore, it is required that the combining delegate function is associative and commutative, and where the initial value is its unit element.

2.2 Non-structured parallelism

For cases where there is no standard control abstraction, TPL also allows the immediate use of tasks or futures. A future is a task where the associated action returns a result. We show the use of futures with a naıv̈e Fibonacci example. Assume

we have a sequential `Fib` function. Its parallel version, called `ParFib`, looks as follows:

```
static int ParFib(int n){
  if(n <= 8) return Fib(n);
  var f2 = new Future<int>(() => ParFib(n-2));
  int f1 = ParFib(n-1);
  return (f1 + f2.Value);
}
```

The `Future<int>` call takes the **int** returning delegate `() => ParFib(n-2)` as a parameter. The result of the future is retrieved through `Value`. Since `ParFib` is recursive, an exponential amount of parallelism is introduced: every `ParFib(n-2)` branch of the call tree creates a subtask which can be done in parallel. The call to `f2.Value` waits until the result is available. Futures are a nice example why embedded domain specific languages need both first-class functions but also parametric polymorphism (generics) in order to achieve enough abstraction to define custom control structures.

The future abstraction works well with symbolic code that is less structured than loops. Since futures are true first-class values, one can use futures to introduce parallelism between logically distinct parts of a program. For example, one can store future values in data structures where another distinct phase will actually request the values of these futures. An interesting application domain is game development. One phase calculates the new health of all actors as a future, while the following phases use the values of those health futures. Another example might be a just-in-time compiler that starts off by generating code for the main procedure, and storing the code for all other procedures in a future value. When calling other procedures, the code may now have been generated already by a parallel task.

3. Task primitives

When building a parallel library, it is important to have just few primitive concepts in order to ensure that operations have regular and consistent semantics. Surprisingly, we can build the entire library on just two primitive concepts, namely tasks and replicable tasks. All other abstractions, like futures and parallel **for** loops are expressed in terms of these two primitives.

3.1 Tasks

A *task* represents a lightweight finite CPU bound computation. It is defined as:

```
delegate void Action();
class Task{
  Task(Action action);
  void Wait();
  bool IsCompleted{ get; }
  ...
}
```

A task is created by supplying an associated action that can potentially be executed in parallel, somewhere between the creation time of the task, and the first call to the `Wait` method. The associated action could potentially be executed on another thread that created the task, but it is guaranteed that actions do not migrate among threads. This is an important guarantee since it enables us to safely use thread affine operations.

The `Wait` method returns whenever the associated action has finished. Any exception that is raised in the action is stored, and re-raised whenever `Wait` is called which ensures that exceptions are never lost and properly propagated to dependent tasks.

Taking it all together, we can see tasks as an improved thread pool where work items return a handle that can be waited upon, and where exceptions are propagated.

On top of these basic tasks we can easily add more advanced abstractions. The parallel `Do` method, for example, can be implemented as follows:

```
static void Do(Action action1, Action action2){
  Task t = new Task(action1); // do potentially in parallel
  action2();                  // call action2 directly
  t.Wait();                   // wait for action1
}
```

Futures are a variation of tasks, where the associated action computes a result:

```
delegate T Func<T>();
class Future<T> : Task{
  Future (Func<T> function);
  T Value{ get; }    // does an implicit wait
}
```

A future is constructed with a delegate having the `Func<T>` type where `T` is the return type of the delegate. The result of the future is retrieved through the `Value` property, which calls `Wait` internally to ensure that the task has completed and the result value has been computed. Since `Wait` is called, calling `Value` will throw any exception that was raised during the computation of the future value. One can view futures as returning either a result value or an exceptional value.

Futures are an old concept already implemented in multilisp (Halstead jr. 1985). Our notion of a future is not "safe", in the sense that that the programmer is responsible for properly accessing shared memory. This is in contrast with some earlier approaches like Moreau (Moreau 1996) where the action of a future is wrapped automatically in a memory transaction.

3.2 Replicable tasks

The second primitive concept of our library are replicable tasks which form the basic abstraction to implement parallel distribution. A *replicable task* can potentially execute its associated action itself in parallel. It is defined as:

```
class ReplicableTask : Task{
  ReplicableTask(Action action);
}
```

The constructor takes an action that can potentially be executed in parallel. If an exception is raised in any of those executions, only one of them is stored and re-thrown by `Wait`.

Replicable tasks are a new concept and have proved to be invaluable for the implementation of the many parallel iteration abstractions that the library implements. In particular, all the `Parallel.For` variations of the library are implemented using replicable tasks. Here is for example a straightforward implementation of the basic `Parallel.For` abstraction[1]:

```
static void For(int from, int to, Action<int> body){
  int index = from;
  var rtask = new ReplicableTask(delegate{
    int i;
    while ((i = InterLocked.Increment(ref index)) <= to){ }
      body(i-1);
    }
  });
  rtask.Wait();
}
```

In this example, the action delegate associated with the replicable task captures a shared index variable `index`. This action can potentially execute in parallel by multiple threads and access to this shared index is coordinated through interlocked operations, where each execution 'claims' one index at a time. Note how we completely abstracted away over work distribution and only express the essential details of the algorithm. As we will see in Section 4.2, the scheduling algorithm in the library automatically ensures that idling processors will participate in doing the work for a replicable task, and that nested parallel for loops are scheduled efficiently.

Replicable futures are replicable tasks that return a value. They are defined as:

```
class ReplicableFuture<T> : Future<T>{
  ReplicableFuture<T> (Func<T> function
                      ,Func<T,T,T> combine);
}
```

A replicable future takes as arguments a function that returns a result, and a combining function that is commutative and associative. Just like replicable tasks, the `function` argument can potentially be executed in parallel by multiple processors, and in that case the `combine` function is used to combine the parallel results. Just as replicable tasks are a good abstraction for implementing parallel iteration, replicable futures are a good abstraction for implementing parallel aggregation. As an example, here is an implementation for the `Parallel.Aggregate` method[2]:

```
static T Aggregate<T>(int from, int to, T init,
                      Func<T> body,
                      Func<T,T,T> combine){
  int index = from;
  var rfuture = new ReplicableFuture<T>(
    delegate{
      int i;
      T   acc = init;
      while ((i = Interlocked.Increment(ref index)) <= to){
        acc = combine(acc, body(i-1));
      }
      return acc;
    },
    combine
 );
  return rfuture.Value;
```

Replicable tasks and futures are a powerful abstraction to implement different parallel iteration strategies. However, they need to be used with care and we mostly see them as a tool for writing extension libraries that capture domain specific parallel patterns. Also note that only tasks and replicable tasks are really primitive to the TPL, since both futures and replicable futures can be implemented in terms of tasks or replicable tasks.

4. Work distribution

As we have seen TPL is convenient to use, but it can only be successful if besides elegance, it is also performant. In this section we focus on important high-level design decisions and focus later on how these decisions inform the design of the work stealing implementation.

4.1 Design for efficiency

The most important contributor for efficiency is the decision to give *no concurrency guarantees*. Parallel tasks are only *potentially* run in parallel. The library specifies that a task is executed between its creation and the first call to `Wait`. This means that we can give a valid implementation that is fully sequential. Indeed, there is a special debug mode where all tasks are executed sequentially when `Wait` is called (and will therefore never have race conditions). Effectively, there are no fairness guarantees for parallel tasks. In contrast with OS threads, parallel tasks are only good for finite cpu-bound tasks, but not for asynchronous programming.

This means that an implementation has a lot of freedom in choosing the most efficient scheduling that processes all tasks in the quickest way possible. For example, on a sin-

[1] An `Action<T>` delegate is an action that returns a value of type `T`.

[2] A `Func<R,A,B>` delegate is a function that takes two arguments of type `A` and `B` respectively, and returns a value of type `R`.

gle core machine, a parallel for loop can just be executed sequentially.

The ability to execute tasks sequentially also enables an *efficient implementation of waiting*. Let's look at the last line of of the `ParFib` function from § 2.2 to see why. When a thread executes `f2.Value` it has to distinguish three cases:

1. The task `f2` *has already been executed* by another worker thread in parallel. This is the ideal case and `f2.Value` immediately returns with the result value and we achieve a (significant) speedup relative to single core execution.
2. The task `f2` *is still being executed* by another worker thread in parallel. This is the worst case, and there is no other choice than to wait until the result is available. In this case, another worker thread will be scheduled on that particular core in order to maintain the ideal concurrency level.
3. The task `f2` *has not been started yet*. This is actually a very common scenario and always the case on single core machines. In this case, a valid strategy is to call the associated action of `f2` directly and compute it sequentially![3]

Case 3 is the major source of efficiency for the library and we put most of our efforts in trying to optimize this case. In particular, this lead us to abandon the usual implementation of work-stealing queues (Herlihy and Shavit 2008) and develop 'duplicating queues' instead (see § 5).

Also note that in case 3, we execute the associated action of the task on the calling thread. If this action does a thread affine operation, like changing the locale or other thread local state, this effect is seen afterwards. We feel that this is fair since we consider a call to `Value` as similar to a method call. Moreover, since we know what code is potentially executed, we can still reason about the thread local state.

Similarly, when we end up in case 2, we always completely block the calling thread and do *not* use the thread to execute other pending tasks. We schedule a fresh worker thread on that core instead. This is important for two reasons. First, unknown pending tasks could execute thread affine operations and prevent us from reasoning about the thread-local state. Second, it is vital to prevent deadlocks: if a task needs to wait on another task, and we would continue execution on the same stack, we may end up in a situation where a task is waiting (indirectly) for a task that is higher up on the stack and where no further progress is possible even if the tasks are not in a circular dependency themselves.

4.2 Work stealing

The TPL runtime reuses the ideas of the well known work stealing techniques as implemented for example in CILK (Danaher et al. 2005; Frigo et al. 1998) and the Java fork-join framework (Lea 2000). We shortly discuss its principles and focus on why our implementation differs.

The runtime system uses one *worker group* per processor. Each group contains one or more *worker threads* where only one of them is running and where all others are blocked. Whenever this running worker thread becomes blocked too (due to the previous case 2 for example) an extra running worker thread is added to that group such that the processor is always busy. A worker thread can also be retired and give control to another worker in its group when that worker can be unblocked (because the task on which it was waiting has completed for example).

Each worker group keeps tasks in its own *doubled-ended queue*. The task queue provides the operations `Push`, `Pop` and `Take`. When a task is created, the running worker thread pushes the task onto the local queue of its group. If it finishes its current task, it will try to pop a task from the group queue and continue with that. This way, a task is always pushed and popped locally, which benefits both the data locality of the task and reduces the amount of synchronization needed. If a worker thread finds there are no more tasks in its queue (and none of the other workers in its group can be unblocked), it becomes a *thief*: it chooses another task queue of another worker group at random and tries to steal a task or replicable task from the (non-local) end of the task queue using `Take`. For many parallel conquer-and-divide algorithms, this ensures that the largest tasks are stolen first. For loops, which are typically implemented via replicable tasks, it means that tasks are only replicated on demand, i.e. when another thread starts idling. Unfortunately, at this point we have to do more synchronization for our queues, since we have multiple parties that can try to steal at the same time from the same queue, and we can have interaction between the worker thread associated with that queue and the stealing thread.

The performance of work stealing is largely dependent on the performance of its task queue implementation. In particular, we need to ensure that each pushed task is only popped or taken once such that each task is only executed once. To achieve atomic take and pop operations, most implementations, like (Arora et al. 1998b), rely on the THE protocol (Dijkstra 1965), which allow the common `Push` and `Pop` operations to be implemented without using expensive atomic compare-and-swap operations. Unfortunately, this only works on machines with a sequentially consistent memory model, and in practice there are just few architectures that support this. On the x86 for example, one needs to insert a memory barrier in `Pop` operation which is almost as expensive as taking a lock in the first place.

Moreover, there is an even bigger disadvantage to queues based on the THE protocol which is more subtle. As shown in Section 4.1, whenever `f2.Value` is called where the future `f2` has not yet started, we should execute it directly on the calling thread. But if we use the THE protocol, there is

[3] Contrast this with waiting on operating system signals, where it is not possible to 'make the signal happen' and the only thing that can be done is blocking the calling thread.

no mechanism to get `f2` exclusively: we can only `Pop` it from our queue if it happens to be the last task that was pushed, otherwise one cannot determine whether `f2` is on the local queue, or resides in a queue of another worker group. Without gaining exclusive access in some other way, we cannot execute the task directly or otherwise we may execute a task more than once since it can concurrently be taken by another worker, or popped if it resided on a non-local task queue.

Since this is exactly the common case that we need to optimize, we opted for another approach where each task uses internally an atomic compare-and-swap operation to ensure that it only executes once. We assign to each task an associated state, `Init`, `Running`, and `Done`. The internal `Run` method on a task performs an atomic compare-and-swap operation to try to switch from `Init` to `Running`. If the atomic operation succeeds, the associated action is executed and the state is set to `Done` afterwards:

```
void Task.Run(){
  if(CompareAndSwap(ref state, Init, Running)){
    Execute();   // execute the associated action
    state = Done;
  }
}
```

Effectively, this ensures that each task is only executed once, or stated differently: running a task is an idempotent operation. Unfortunately, if we would use task queues based on the THE protocol we now use two mechanisms for mutual exclusion, and execute both a memory barrier instruction on a `Pop` operation, and an interlocked instruction when calling `Run`. Since these are expensive instructions that require synchronization among the processors, we would like to avoid these. As shown in the previous paragraphs, the interlocked instruction in the `Run` method is essential and since we now ensure exclusivity on the task level, it is possible to use a weaker data structure for the task queues. In particular, this leads to the development of the *duplicating queue*.

Note that only tasks need this additional check; for replicating tasks there is no need to do the atomic compare-and-swap operation since they already implement their own mutual exclusion.

5. Duplicating queues

A duplicating queue is a double-ended queue that potentially returns a pushed element more than once. In particular, the `Push` and `Pop` operations behave like normal, but the `Take` operation is allowed to either take an element (and remove it from the queue), or to just duplicate an element in the queue. While this nondeterminism might be dangerous for many clients, it is fine for our usage of the duplicating queue: the `Task`'s `Run` method is idempotent and `ReplicableTask`'s expect to be executed in parallel. Other properties of a duplicating queues are as usual: a duplicating queue never loses an element, and returns all pushed elements after a finite number of `Pop` (and `Take`) operations.

By allowing duplication we can avoid an expensive memory barrier instruction in the `Pop` operation on the x86 architecture. More generally, our duplicating queue is designed specifically to be correct on architectures satisfying the Total Store Order (TSO) memory model (Sindhu et al. 1991) or stronger model. To our knowledge, this is one of the first data structures that takes specific advantage of weaker memory models to avoid memory barriers. An interesting aspect is that the non-determinism that is introduced by the weaker memory model is captured in a benign way where the number of duplicated elements is non-deterministically determined.

5.1 Total Store Order

Before explaining and verifying the algorithm, we first give a short specification of the TSO memory model along the lines of (Collier 1992; Gharachorloo et al. 1992; Sindhu et al. 1991). We write L_A for load at address A, and S_A for a store at address A. The memory order is an order on all the memory transactions from the processors, and we write $op_1 <_m op_2$ if instruction op_1 precedes instruction op_2 in a particular memory order. Similarly, we can define a *program order* which is the order of instructions as executed by the processors. The order is total for each processor, while the order on all instructions is a partial order defined by an interleaving of the total order on each processor. We write $op_1 <_p op_2$ if an instruction op_1 precedes another instruction op_2 in program order.

We can now define the TSO memory model as a set of axioms on the possible memory order:

1. In TSO, the stores are in a total memory order, i.e.

$$\forall SS'.\ S <_m S' \ \vee \ S' <_m S$$

2. If two stores occur in program order, then these stores are in memory order too:

$$S <_p S' \ \Rightarrow \ S <_m S'$$

   Informally, this means that a local store buffer on a processor is kept in FIFO order where memory stores cannot be reordered.

3. If a load occurs before another operation in program order, it also precedes it in memory order:

$$L <_p op \ \Rightarrow \ L <_m op$$

   Together with the previous axiom, this effectively means that only a load that succeeds a store in program order, may be reordered to precede it in the global memory order.

4. Finally, loads will always return the last value stored where stores that precede it in program order are seen

even if such store occurs after the load in memory order. Writing $val(op)$ for the value that is stored or loaded, we formalize this as:

$$val(L_A) = val(max_m(\{S_A \mid S_A <_m L_A\} \cup \{S_A \mid S_A <_p L_A\}))$$

where max_m is the maximum store under the memory order relation ($<_m$). Informally, this axiom describes that processors always see their local stores where a load 'snoops' the store buffer.

The TSO memory model is for example supported by the Sparc architecture. The widely available x86 architecture almost implements TSO except that stores do not always form a total order (axiom 1). In particular, with four processors it is possible that two stores by two processors are seen in an opposite order by the other two processors, i.e. stores to memory may be seen at different times by different processors depending on the physical layout of the processors. However, as we will see below, we will arrange that the duplicating queue is only accessed by at most two concurrent processes, and therefore the x86 behaves just like a TSO architecture for our particular purposes.

5.2 Implementation

We describe the duplicating queue using C# code. We have chosen to present not the simplest possible implementation, but to use a fairly realistic implementation that represents the queue as an array and uses wrap around to prevent shifting the elements. The members of the queue are defined as:

```
class DupQueue{
  Task[]  tasks;
  int     size;              // size ⩾ 2
  int     tailMin   = ∞;
  volatile int tail = 0;
  volatile int head = 0;
  ...
}
```

The `tasks` array contains the elements of the queue, where `size` is the size of the array. The `tailMin` member is essential to prevent losing elements and is discussed later. The `head` and `tail` are indices in the array and represent the head and tail of the queue. When `head == tail` the queue is empty. There are three main operations on the queue: the `Take` operation is called by other worker threads to steal elements from the head of the queue by incrementing the `head` index. The `Push` and `Pop` operations are called by the worker thread that owns the queue and pushes and pops elements to the tail of the queue, by respectively incrementing and decrementing the `tail` index. By declaring the `head` and `tail` as **volatile** we prevent the compiler from rearranging accesses to those members.

The `Take` operation is called by thiefs and is the simplest and least often performed operation. Its definition is given in Figure 1. Since multiple thiefs may try to steal a task at the same time, the code first takes a lock. This allows us to reason about the algorithm as if there are only two concurrent threads: a thief that calls `Take`, and a worker thread that calls `Push` and `Pop` (to make tasks available for stealing). If the queue is not empty, the head element of the queue is returned. For efficiency, the elements are accessed modulo the `size` of the array. This way, we do not have to shift the elements in the array once the end of the array is reached but can simply wrap around.

```
public Task Take(){
  lock(this){
    if(head < tail){
      Task task = tasks[head%size];
      head = head + 1;
      return task;
    }
    else{
      return null;
    }
  }
}
```

Figure 1. The `Take` operation is called by a thief

The thief is the only thread that modifies the `head` field (and reads `tasks`). The worker is the only thread that modifies the `tail` field and the `tasks` array. Since these fields are modified without using atomic operations, this means that on a TSO architecture the thief may see an outdated `tail` field or element, while the worker thread may see an outdated `head` field. To make this more explicit in the code, we denote a potentially outdated `tail` field in a thief thread as $\boxed{\texttt{tail}}$, and a potentially outdated `head` field in the worker thread as $\boxed{\texttt{head}}$.

More formally we can have the order:

$$L^w_{\texttt{tail}} <_m L^t_{\boxed{\texttt{tail}}} <_m S^w_{\texttt{tail}} \text{ and } S^w_{\texttt{tail}} <_p L^w_{\texttt{tail}}$$

where we use the superscript ^{w} for operations in the worker thread, and ^{t} for operations in the stealing thread. In this case, the load $L^w_{\texttt{tail}}$ sees the value stored by $S^w_{\texttt{tail}}$ due to the program order (and axiom 4), while the load executed in the thief $L^t_{\boxed{\texttt{tail}}}$ sees the old value of the `tail` field before the $S^w_{\texttt{tail}}$ store happened.

Since the `head` field is only incremented we have the following invariants:

A. $\texttt{head} \geqslant \boxed{\texttt{head}}$

Also, in the `Take` operation, we can assume:

B. *Any* `elem[i%size]` *where* $\texttt{head} \leqslant \texttt{i} < \boxed{\texttt{tail}}$ *is valid or* **null**.

Note that $\texttt{tail} \geqslant \boxed{\texttt{tail}}$ does *not* hold since the tail can be decremented by the `Pop` operation (as described below).

```
public Task Pop(){
  tail = tail - 1;
  // can we pop safely?
  if (head <= Math.Min(tailMin,tail)){
    if(tailMin > tail) tailMin = tail;
    Task task = tasks[tail%size];
    tasks[tail%size] = null;
    return task;
  }
  else{
    lock (this){
      // adjust head and reset tailMin
      if(head > tailMin) head = tailMin;
      tailMin = ∞;

      // try to pop again
      if(head <= tail){
        Task task = tasks[tail%size];
        tasks[tail%size] = null;
        return task;
      }
      else{
        tail = tail + 1; // restore tail when empty
        return null;
      }
    }
  }
}
```

Figure 2. The `Pop` operation is called by a worker

```
public void Push(Task task)
{
  if(task == null) return ;

  // queue not full and no index overflow?
  if(tail < Math.Min(tailMin,head)+size &&
      tail < int.MaxValue/2){
    tasks[tail%size] = task;
    tail = tail + 1;
  }
  else{
    lock(this){
      if(head > tailMin) head = tailMin;
      tailMin = ∞;

      // adjust the indices to prevent overflow
      int count = Math.Max(0,tail - head);
      head = head % size;
      tail = head + count;
    }
    // just run this task eagerly
    task.Run();
  }
}
```

Figure 3. The `Push` operation is called by a worker

Informally this means that `Take` can return an arbitrary element of the queue. Also `head` might be overwritten, i.e. the returned element is not deleted.

Figures 2 and 3 define the `Pop` and `Push` operations. Both of these operations are only called from the same worker thread and therefore do not need to take a lock. The `tailMin` field contains the minimal index at which an element was popped.

The `Pop` operation decrements the `tail` field opportunistically and tests whether the queue still contains elements (`head <= tail`) and if the `head` field is smaller or equal to the `tailMin`. This test is written as `head <= Math.Min(tailMin, tail)` to combine both tests using a single load of the `head` field (as a subsequent load may give a different (higher) value). If the test succeeds, the `tailMin` field is updated and an element is popped. To prevent space leaks a **null** value is written to the popped location. If the test fails, we fall back on a safe routine and take an explicit lock. In this case, there are no concurrent thiefs and the `head` value is current due to the implicit memory barrier of a lock, and we can safely determine whether the queue was empty or not.

The initial test `head <= tail` ensures we only pop elements that have previously been pushed. Theoretically, many steals could have happened where `head` is much larger than `head`. However, it is still safe to pop those elements: i.e. return an element both from a `Take` and from a `Pop` operation even though the element has only been pushed once - the `Run` method, which consumes the returned task takes care that a duplicate task is executed only once. The second initial test `head <= tailMin` prevents losing elements. In particular, on a TSO architecture the following sequence of events could occur:

1) We push an element A and `tail` and `tail` are 1.

2) There is steal of A and `head` $= 1$ but `head` is still 0.

3) There is a pop of A and `tail` and `tailMin` become 0.

4) There is a push of B where `tail` becomes `1` again.

5) By now the value of `head` becomes equal to `head` (i.e. 1).

Suppose we have a `Pop` operation now. If we would only test for `head <= tail`, we would assume that the queue is empty and the B element would be lost! This situation is prevented by the `tailMin` field. Since the test `head <= Math.Min(tailMin, tail)` fails, the safe path is taken. After the lock is taken, the `Pop` operation potentially resets the `head` field to the `tailMin` field, and resets the `tailMin` field to ∞. In the above scenario, `head` becomes 0 again and the

B element is immediately popped. With the addition of the `tailMin` field, we have the following invariants:

C. *Any* `elem[i%size]` *where* $min(\boxed{\texttt{head}}, \texttt{tailMin}, \texttt{tail}) \leqslant i < \texttt{tail}$ *is valid or* **null**.

D. $\texttt{tailMin} \geqslant \boxed{\texttt{head}}_0$ *where* $\boxed{\texttt{head}}_0$ *is the first* $\boxed{\texttt{head}}$ *value loaded since initialization or taking a lock.*

The `Push` operation in Figure 2 also avoids a lock in almost all cases. If a **null** value is pushed the method returns immediately. It then tests whether there is still room in the queue. One may think that the test `tail < head + size` would suffice but that would not be strong enough. Indeed, as captured in invariant C, the `tailMin` field could be lower as `head` and we should be careful not to overwrite those values. Therefore, the push operation tests whether the `tail` field is smaller as $min(\texttt{tailMin}, \texttt{head})$ `+ size`. Also, to prevent integer overflow after many steals and pushes, there is a test to prevent the `tail` value from getting too large.

If there is still room in the queue, the element is pushed and the tail is incremented. Otherwise, a lock is taken to prevent concurrent thiefs. Just like in `Pop`, the `head` value potentially is set to the `tailMin` field, and `tailMin` is set to ∞. To prevent integer overflow, the `head` and `tail` field are both adjusted to their minimal values. Finally, the pushed task is just run eagerly: indeed, pushing a task is only done to enable it to be stolen by other worker threads. If the work queue is full, there is no reason to enlarge it further since there are still enough tasks that can be stolen and we can just as well execute it directly.

5.3 Verification

As apparent from the above description, the individual operations of the duplicating queue are simple, but the concurrent interaction between them is very subtle. To gain confidence in the correctness of the algorithms, we applied *CheckFence* (Burckhardt et al. 2007) to verify the data structure formally under the TSO memory model.

CheckFence takes three inputs: (1) a sequential reference implementation of the duplicating queue which serves as a specification, (2) a suite of concurrent unit tests, and (3) an axiomatic specification of the memory model. It then uses a SAT solver to verify that all possible concurrent executions of the tests on the given memory model are observationally equivalent to some interleaved, atomic execution of the reference implementation. Our experience confirmed that this methodology is very effective for finding subtle concurrency bugs. We had to revise our initial implementation several times based on the counterexample traces produced by CheckFence. The specific test suite we used is described in Appendix A.

To specify the permitted behaviors of the duplicating queue, we expressed the 'duplication' by introducing nondeterminism into the `Take` operation. Our specification operates like a normal double-ended queue insofar `Push` and `Pop` operations are concerned (they simply add/remove elements at the tail end of the queue). In contrast, the `Take` operation nondeterministically chooses between two behaviors, either (1) removing an element from the head end of the queue as usual, or (2) returning a nondeterministically chosen element of the queue without removing it. Clearly, this specification is strong enough to ensure correct work stealing: no elements are lost or invented, and repeated `Pop` calls will reliably empty the queue.

CheckFence indeed verified that our queue algorithm never produces answers outside of the specification. Even though the correctness proof is limited to test sequences described in the appendix, they are verified under *all* possible interleavings allowed by the TSO memory model. In this respect our methodology goes well beyond common practice, and given the complexity of reasoning about concurrent data structures under weak memory models, we feel that this is a strong result.

6. Performance

Given the continuous improvements in virtual machines and compilers for managed code, we feel that absolute comparisons against other languages and frameworks is of limited value. Therefore, we measure only intrinsic properties of our library like scalability and relative speedups.

6.1 Matrix multiplication revisited

TPL can only succeed if it is simple to use *and* well-performing. So let's compare the performance of the introductory matrix-multiplication with a hand written solution using the standard thread pool to introduce parallelism. Figure 4 shows a fairly sophisticated implementation: for instance, the code statically divides the loop into chunks depending on the number of processors, creating twice as many as necessary to adapt better to dynamic workloads; also to minimize the number of kernel transitions the code introduces a counter together with a single wait handle.

Clearly, this code is harder to write, and more error prone than the `Parallel.For` method. Perhaps it performs better? Even though it is hand tuned and uses a near optimal division of work, it performs generally worse than `Parallel.For`. Figure 5 shows the speedups obtained on an eight core machine relative to running the sequential for loop. Note how the `Parallel.For` version is just slightly slower ($\approx$99%) than a direct **for** loop on a single core machine.

Note also that this hand-optimized code is not compositional, for instance you have to do a major rewrite if you wanted to parallelize both outer loops; TPL however is composable, so you can simply exchange both outer loops with `Parallel.For`s. Also, in contrast to TPL, the hand-optimized version does not propagate exceptions raised in the loop body.

```
void ThreadMatrixMul(int size, double[,] m1
                     ,double[,] m2, double[,] res)
{
  int N = size;
  int P = 2 * Environment.ProcessorCount;
  int Chunk = N / P;                  // size of a work chunk
  AutoResetEvent sig = new AutoResetEvent(false);
  int counter = P;
  for(int c = 0; c < P; c++){   // for each chunk
    ThreadPool.QueueUserWorkItem(
    delegate(Object o){
      int lc = (int)o;
      for(int i = lc * Chunk;     // for each item
            i < (lc + 1 == P ? N : (lc + 1) * Chunk);
            i++){
        // original inner loop body
        for(int j = 0; j < size; j++){
          res[i, j] = 0;
          for(int k = 0; k < size; k++){
            res[i, j] += m1[i, k] * m2[k, j];
          }
        }
      }
      // use efficient interlocked instructions
      if(Interlocked.Decrement(ref counter) == 0){
        sig.Set();   // kernel transition only when done
      }
    }, c);
  }
  sig.WaitOne();
}
```

Figure 4. Parallel matrix multiplication without using the TPL leads to static work distribution and explicit synchronization.

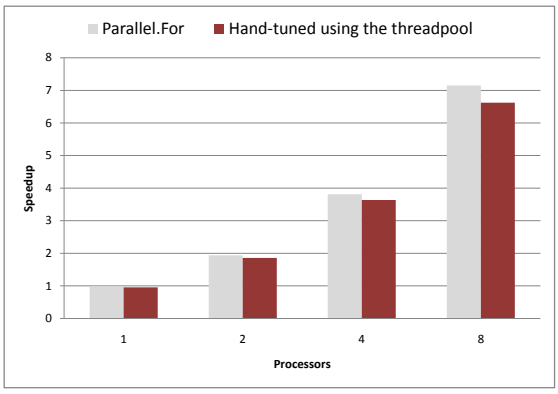

Figure 5. Relative speedup of parallelizing the outer loop of a matrix multiplication with 750x750 elements, where 1 is the running time of a normal for-loop. The tests were run on a 4-socket dual-core Intel Xeon machine with 3Gb memory running Windows Vista.

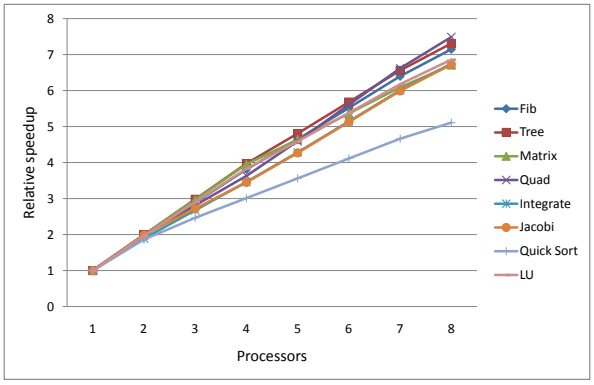

Figure 6. Relative speedup of standard benchmarks. The benchmarks were run on a 4-socket dual-core Intel Xeon machine with 32Gb running Windows Server 2008. Note that Quick Sort cannot be fully parallelized.

6.2 Scaling

The library should also scale well with common parallel patterns. We used a collection of common benchmarks adapted from the CILK and Java fork/join library benchmarks:

- Fib: The fibonacci of 44 using a threshold of 18.
- Tree: Summation of nodes in a balanced tree of depth 28 with a threshold of 11.
- Matrix: Naïve matrix multiplication of a 1024x1024 matrix of doubles.
- Quad: Divide-and-conquer quad matrix multiplication of a 1024x1024 matrix of doubles.
- Integrate: Recursive Gaussian quadrature of $(2i-1)x^{2i-1}$ summing over odd values of $1 \leqslant i \leqslant 6$ and integrating from -11 to 12.
- Jacobi: Iterative mesh with 100 steps of nearest neighbor averaging on a 4096x4096 matrix of doubles.
- Quick Sort: Traditional quick sort algorithm on a 1.6 million element array with a sequential quick sort threshold of 4000 elements. This algorithm will not scale linearly when run in parallel but the parallel speedup is bounded by $\frac{Plog_2(N)}{2P-2+log_2(N/P)}$ (Thornley 1995).
- LU: Matrix decomposition of a 4096x4096 matrix of doubles.

Figure 6 shows that all these algorithms have good linear speedups when run on multiple processors. The exception is the traditional quick sort benchmark which does not scale linearly but we come very close to the ideal speedup. Another important point is that all of the benchmarks run (almost) as fast on a single processor as their sequential coun-

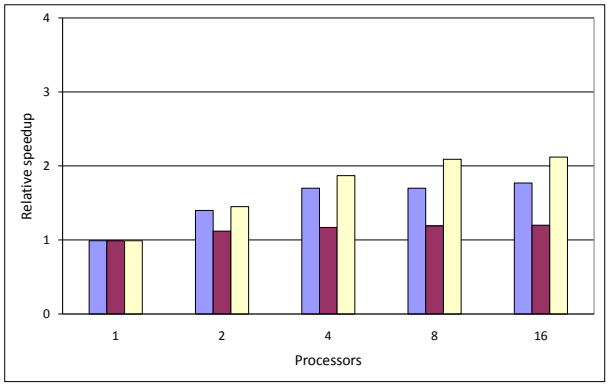

Figure 7. Relative speedup of laying out three extreme graphs using MSAGL. The test were run on a 8- socket dual-core Intel Xeon machine with 4Gb memory running Windows Vista.

terparts which is important when running parallelized software on older machines.

Even though the above benchmarks show great speedups, we believe that for most realistic programs the speedups will be much more modest. According to Amdahl's law the sequential part of a program will eventually dominate the running time. For example, even when just 10% of your application is intrinsically sequential, we can at most make this program run 10 times as fast when perfectly parallelizing the other 90%. In practice this means that especially the first 8 cores can make a substantial difference in running times, but adding more cores will have little effect.

Microsoft product groups have started using TPL and have observed this effect. For instance, Figure 7 describes the relative speedup of the Microsoft Automatic Graph Layout library (MSAGL) (Nachmanson and Powers 2008) for three extreme graphs. MSAGL is a large and complex library and we just replaced a single **for** loop with a `Parallel.For` to parallelize MSAGL's spline computations. Unfortunately, only part of the layout algorithm can be parallelized this easily, as the other parts currently consist of intrinsically sequential algorithms, like a sequential simplex algorithm. When looking at the figure, we see that the sequential parts dominate and the speedups are modest, between 1.2 to 2.2 times faster on 8 processors. Still, since MSAGL is used for example to layout large XML diagrams in real time in Visual Studio, the speedup is often noticeable in practice.

Of course, in many cases we *can* take advantage of ever more cores, as long as we can make the parallel part relatively larger by just adding more data. This happens in particular in domains like games or speech analysis.

6.3 Duplicating queues

Measuring the performance of the duplicating queue in isolation is not very useful. The reason is that the main performance benefit does not come from the duplicating queue, but from being able to guarantee mutual exclusivity in the tasks themselves, such that a task can be executed directly when `Wait` is called and the task has not started yet. Since the task queues no longer need to guarantee mutual exclusivity, the duplicating queue is mostly an optimization to avoid using too many expensive interlocked instructions.

Therefore, it is better to measure the benefit of being able to directly execute a task for which `Wait` is called (and which has not started yet). We created two versions of the library: one based on a duplicating queue with direct execution of tasks, and a traditional implementation based on the standard THE protocol. The standard fibonacci benchmark represents one of the worst-case examples since most waits are for tasks that were just pushed on the stack (and reside on the top of the stack). We ran this benchmark on a 4 processor machine which used 196418 tasks ($\approx$ 500.000 tasks per second). The implementation based on the duplicating queue was 1.4 times faster when using all processors. Here are some statistics, where `DUP` refers to the duplicating queue implementation, and `THE` to the implementation based on the THE protocol.

	speedup	steal	switch	migrate	workers
`DUP`	3.89	29	20	6	20
`THE`	2.78	37	2452	9596	53

As we can see, the performance of `THE` mostly suffered because there were many more switches between threads, and many more thread migrations. This happened precisely when `Wait` was called for a task that had not yet started. Since one cannot reliably determine whether this task was on the local queue, a fresh worker thread was needed to execute the task resulting in a thread switch. Of course, this automatically also lead to more migration of threads where ready worker threads were stolen by another worker group. Interestingly, the number of task steals are about the same as this is mostly determined by the particular algorithm and sizes of the tasks.

Of course, for most fork-join parallelism (including the fibonacci benchmark), the task to be waited upon is often right on the top of the local task queue. When `Wait` is called on a task that has not started yet, we can optimize for this case in the `THE` implementation by simply popping the top of the stack if that happens to be our task and execute it directly. When applying this simple optimization, both implementations perform very similar for this benchmark. Of course, the `DUP` implementation outperforms again when the parallelism is less structured using futures for example, and in general at any time when `Wait` is called on a task that is not on the top of the stack.

7. Related work

There is a wealth of research into parallel scheduling algorithms, data structures, and language designs, and we neces-

sarily restrict this section to work that is directly relevant to work stealing and embedded library designs.

The idea of duplicating queues has recently been descibed by Maged Michael et al (2009) as "idempotent" queues. We were not aware of this work at the time of writing this paper and arrived at our results independently (doing our first implementation in January 2008). Their general motivation and the semantics of the idempotent queue seem largely identical, but the implementation is quite different. Their elegant implementation packs fields together in a memory word and relies strictly on atomic compare-and-swap and memory ordering instructions. In contrast, our implementation uses a simple lock on all but the critical paths which can simplify many implementation aspects and also removes a level of indirection on the critical path.

Futures are a well known concept and were already implemented for multi-lisp (Halstead jr. 1985). It is important to distinguish *safe* futures from the futures as described in this article. Safe futures have no observable side effect and can therefore always be safely evaluated in parallel. In contrast, TPL futures have no such restrictions and programmers have to be careful when accessing shared state. Safe futures in the context of Java have been studied for example by Pratikakis et al. (2004) and Welc et al. (2005). The semantics of futures with side effects and exceptions has been studied by Moreau (Moreau 1996) and by Flanagan and Felleisen in the context of optimization (Flanagan and Felleisen 1995).

CILK (Randall 1998) championed lightweight task execution frameworks for C. The CILK runtime used the THE protocol to implement task queues (Frigo et al. 1998; Dijkstra 1965). Tasks are created using the `spawn` primitive, and synchronized with the `sync` keyword. In contrast to TPL, the liveness of CILK tasks is determined by their lexical scope which enables the CILK compiler to allocate tasks on the stack instead of the heap which can be more efficient. On the other hand, it makes tasks second-class citizens and restricts how they can be used. More recent work on a Java based CILK implementation studies the interaction of Java exceptions with the CILK `sync` and `spawn` primitives (Danaher et al. 2005, 2006).

There also exist quite a few task based libraries for lightweight parallel programming in standard C and C++ based on workstealing. Some of the more well-known include StackThreads (Taura et al. 1999), the Filaments library (Engler et al. 1993), and Hood (Blumofe and Papadopoulos 1999; Blumofe et al. 1995).

Blumofe and Leiserson (Blumofe and Leiserson 1999) proved that the work-stealing algorithm is efficient with respect to time, space, and communication for the class of fully strict multithreaded computations. Arora, Blumofe, and Plaxton (Arora et al. 1998a) extended the time bound result to arbitrary multithreaded computations. In addition, Acar, Blelloch, and Blumofe (Acar et al. 2000) show that work-stealing schedulers are efficient with respect to cache misses for jobs with nested parallelism. Agrawal looked into improving the heuristics of work stealing through adaptive scheduling (Agrawal et al. 2006).

There exist many languages with light weight task parallelism. For instance, the Fortress language (Steele 2006; Allen et al. 2007; Aditya et al. 1995) makes parallelism the default execution mode, and internally uses work stealing to schedule tasks efficiently.

Our main inspiration is the widely used Java fork-join framework by Doug Lea (Lea 2000, 1999) which is now part of the standard Java libraries. Just like CILK, tasks are restricted to fork-join style parallelism only and should not escape their lexical scope. This is not enforced though which can be dangerous as the library can deadlock when tasks are used outside the lexical scope (even if there is no circular dependency among tasks).

8. Conclusion

We described the lessons learned in the design of a library for parallel programming. TPL is an example of the possibilities of an embedded domain specific language that relies heavily on parametric polymorphism and first-class anonymous functions, and we hope to apply this to other domains as well.

To the best of our knowledge, the duplicating queue is one of the first data structures that explicitly takes the properties of weak memory models into account, and it is surprising we can capture the resulting non-determinism in a benign way without for example losing or inventing elements. It would be interesting to see if we can adapt the structure such that it can be applied for other parallel algorithms too.

Acknowledgments

The authors would like to thank Vance Morrison, Joe Duffy, and Stephen Toub for their feedback and help in the design and implementation of TPL.

References

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, 2000.

Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 100–109, 2006.

Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project fortress: A multicore language for multicore processors. In *Linux Magazine*, September 2007.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998a.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, 1998b.

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

Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas, Austin, 1999.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

William W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. The jcilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in jcilk. *Science of Computer Programming (SCP)*, 63(2):147–171, December 2006.

E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

Dawson R. Engler, Dawson R. Engler, Gregory R. Andrews, Gregory R. Andrews, David K. Lowenthal, and David K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report TR 93-13a, University of Arizona, 1993.

Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Rice University*, pages 209–220, 1995.

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.

Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

R.H. Halstead jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201310090.

Doug Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, 2009.

Microsoft. Parallel extensions to .NET. June 2008. URL `http://msdn.microsoft.com/en-us/concurrency`.

Luc Moreau. The semantics of scheme with future. In *In In ACM SIGPLAN International Conference on Functional Programming (ICFP'96*, pages 146–156, 1996.

Lev Nachmanson and Lynn Powers. Microsoft automatic graph layout library (msagl). `research.microsoft.com/en-us/projects/msagl`, 2008.

Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. *SIGPLAN Not.*, 39 (10):206–223, 2004. ISSN 0362-1340.

Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, December 1991.

Guy Steele. Parallel programming and parallel abstractions in fortress. In *Invited talk at the Eighth International Sym-*

posium on Functional and Logic Programming (FLOPS), April 2006.

Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: integrating futures into calling standards. *SIGPLAN Not.*, 34(8):60–71, 1999.

John Thornley. Performance of a class of highly-parallel divide-and-conquer algorithms. Technical Report 1995.cs-tr-95-10, California Institute of Technology, 1995.

Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 439–453, 2005.

A. Verification test sequences

We modelled the following test sequences in *checkfence*, where `i2` and `i4` initialize a queue of size 2 and 4 respectively, `ps` is a push operation, `pp` a pop operation, `tk` a take operation, and the bar (`|`) operator composes operation sequences in parallel:

```
i2 (ps pp | tk)
i4 (ps ps pp | tk tk)
i4 (ps ps pp | tk)
i4 (ps pp pp | tk | tk)
i4 (ps ps pp pp | tk)
i4 (ps pp ps pp | tk | tk tk)
i4 (ps ps pp pp | tk | tk)
i4 (ps ps pp pp | tk tk)
i4 (ps ps ps pp pp pp | tk)
i4 (ps pp ps ps pp pp | tk)
i4 (ps pp ps pp pp ps | tk | tk)
i4 (ps ps ps pp pp pp | tk | tk tk)
i4 (ps ps pp ps ps pp pp pp | tk tk)
```

B. Differences with the productized TPL

At the time of writing, there are the following API differences with the productized TPL (.NET Framework 4.0, Beta 1):

- A task is started as `Task.Factory.StartNew(action)`, and a future as `Task.Factory.StartNew(function)`.
- The value of a future is retrieved as `future.Result`.
- `Parallel.Aggregate` is only available through PLINQ.
- Replicable tasks are not exposed.