The Task Parallel Library

a concurrency library for .NET

Daan Leijen
Wolfram Schulte
Sebastian Burckhardt
Microsoft Research

RISE

http://msdn.microsoft.com/concurrency
The Concurrency Elephant for Multicore

- Infrastructure (locks, threads, STM)
- Asynchronous agents (responsiveness)
- Task parallelism (performance)
A sequential matrix multiply

```c
void MatMult(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];
            }
        }
    }
}
```
Making the loop parallel today

```java
int N = size;
int P = 2 * Environment.ProcessorCount;
int Chunk = N / P; // size of a work chunk
AutoResetEvent signal = new AutoResetEvent(false);
int counter = P; // use a counter to reduce kernel transitions
for (int c = 0; c < P; c++) { // for each chunk
    ThreadPool.QueueUserWorkItem(delegate(Object o) {
        int lc = (int)o;
        for (int i = lc * Chunk; i < (lc + 1 == P ? N : (lc + 1) * Chunk); // respect upper bound
            i++) {
            // original body
            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];
                }
            }
        }
        if (Interlocked.Decrement(ref counter) == 0) // efficient interlocked ops
            signal.Set(); // and kernel transition only when done
    }, o);
}
signal.WaitOne();
```
Using the concurrency library

```csharp
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];
        }
    }
});

Compare with sequential version:

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];
        }
    }
}
Demo: Ray tracing [by Luke Hoban]

Single line change to parallelize:

```csharp
void RenderScene(Scene scene, Int32[] rgb)
{
    //for (int y = 0; y < screenHeight; y++)
    Parallel.For(0, screenHeight, delegate(int y)
    {
        for (int x = 0; x < screenWidth; x++)
        {
            Ray ray = new Ray(scene.Camera.Pos,
                                GetPoint(x, y, scene.Camera));
            Color color = TraceRay(ray, scene, 0);
            rgb[x+y*screenWidth] = color.ToInt32();
        }
    });
}```
TPL: Features

- Just a regular .NET library
- Works with VS 2005 and up (.NET 2.0+)
- No language extensions needed
- Compatible with existing infrastructure
- Small API (fits on 3 slides)

- You can use it today: http://msdn.microsoft.com/concurrency
Agenda

• Motivation
  – Demo, Features
• Gestalt
  – Language, Architecture, Work Stealing
• Foundation
  – Tasks, Replicable Tasks, Duplicating Queue
• Remarks
  – Limitations, Conclusion
Parameteric polymorphism (generics) and anonymous first-class functions (delegates) are essential to define custom control structures as a library and does not require language extensions.

```csharp
static void Parallel.For( int from, int to, Action<int> body );
```

Pass a delegate function that takes the loop index as its first argument.
Architecture

User abstractions: `Parallel.For`, `Task<T>`, etc.

Primitives: `Task`, `ReplicableTask`

Task scheduler and infrastructure: `DupQueue`

- Tasks `p1`, `p2`, `p3`, `p4`
- Queues:
  - One Thread per processor
  - Task Queues
Runtime

• TPL uses work stealing in combination with lock-free concurrent task queues
• Work stealing has provably good locality and work distribution properties
Static work distribution
Dynamic work distribution

p1  p2  p3  p4
Agenda

• Motivation
  – Demo, Features

• Gestalt
  – Language, Architecture, Work Stealing

• Foundation
  – Tasks, Replicable Tasks, Duplicating Queue

• Remarks
  – Limitations, Conclusion
class Task
{
    Task( Action action );
    void Wait(); // Wait for the task to finish ("Join")
    bool IsCompleted { get; } // Is the task finished?
    ...
}

class Task<T> : Task // Tasks that compute a value ("Futures")
{
    Task( Func<T> function );
    T Value { get; } // Value does an implicit Wait()
}

• If an exception is raised in the task, it is stored and re-thrown from Wait.
Example: Fibonacci

class Fib
{
    public static int Fib(int n)
    {
        if (n <= 1) return n;
        var f1 = Fib(n-1);
        var f2 = Fib(n-2);
        return f1+f2;
    }

    public static void main(String[] args) { // sample driver
        Console.WriteLine("Answer: " + Fib(35));
    }
}
Using Java

class Fib extends FJTask {
    private volatile int number;
    public Fib(int n) { number = n; }

    public int getAnswer() {
        if (!isDone()) throw new Error("Not yet computed");
        return number;
    }

    public void run() {
        int n = number;
        if (n > 1) {
            Fib f1 = new Fib(n - 1);
            Fib f2 = new Fib(n - 2);
            coInvoke(f1, f2);
            number = f1.number + f2.number;
        }
    }

    public static void main(String[] args) {
        FJTaskRunnerGroup group = new FJTaskRunnerGroup(2); // 2 worker threads
        Fib f = new Fib(35);
        group.invoke(f);
        System.out.println("Answer: " + f.getAnswer());
    }
}
Using C#

class Fib
{
    public static int Fib(int n)
    {
        if (n <= 1) return n;
        var f1 = new Task<int>(() => Fib(n-1));
        var f2 = Fib(n-2);
        return f1.Value + f2;
    }

    public static void main(String[] args) {    // sample driver
        Console.WriteLine("Answer: " + Fib(35));
    }
}

The structure of the original code is maintained through the use of an anonymous delegate expression (that captures the free variable n).
Tasks are first-class values

- We can put tasks in data structures, or return them from functions.
- This is good for modularity:
  E.g. a game could calculate the health of actors as a `Task<int>` value, and in later phases access those values using the `Value` property.
Abstractions

- Delegate expressions can capture computations and allow us to abstract over common control patterns.
- Everything is build on just 2 primitives (`Task` and `ReplicableTask`).

```csharp
static void Parallel.Do( Action action1, Action action2 )
{
    Task t = new Task( action1 );
    action2();
    t.Wait();
}
```
Foundation: Replicable tasks

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

- The action of a Repl. Task can be done in parallel with itself
- This is the fundamental pattern for a parallel “apply to all”

- All Parallel.For methods are implemented with Repl.Tasks

- Repl. Tasks can be stolen but don’t have to be.
A naïve parallel for method:

```csharp
static void ParallelFor( int from, int to, Action<int> body )
{
    int index = from;
    ReplicableTask task = new ReplicableTask( delegate {
        int i;
        while ((i = Interlocked.Increment(ref index)-1) < to) {
            body(i);
        }
    });
    task.Wait();
}
```

- By exposing replicable tasks, users can create their own high-level iteration patterns, like a for loop with a stride, or iteration over external data
int sum = 0;
for(int i = 0; i < 100; i++)
{
    sum += Fib(i);
}
Shared variables

```csharp
int sum = 0;
Parallel.For(0, 100, delegate(int i)
{
    sum += Fib(i);
});
```

Two problems:
- Data race
- Cache contention

```csharp
int f = Fib(i);
lock(this) {
    sum += f;
}
```
Parallel Aggregate ("Map-Reduce")

```csharp
int sum = Parallel.Aggregate( 0, 100,
    0,  // initial value
    i => Fib(i),  // mapper
    (x, y) => x + y  // combiner
);
```

The above expression is the parallel equivalent of:

```csharp
int sum = 0 + Fib(0) + Fib(1) + .. + Fib(100);
```
Replicable futures

class ReplicableTask<T> : Task<T>
{
    ReplicableTask<T>( Func<T> function, Func<T,T,T> combine );
}

• A replicable future is a function that can be executed in parallel with itself, together with a combine function to combine the parallel results.

• A replicable future is the fundamental pattern for parallel aggregation over elements (ie. “map reduce”).

• All Parallel.Aggregate methods are implemented with replicable futures
A naïve Aggregate implementation

```csharp
static T Aggregate<T>( int from, int to, T init,
    Func<T,T,T> combine, Func<T> body ) {
    int index = from;
    ReplicableTask<T> future = new ReplicableTask<T>(
        delegate {
            int i;
            T localResult = init;
            while ((i = Interlocked.Increment(ref index)-1) < to) {
                localResult = combine(localResult, body(i));
            }
            return localResult;
        },
        combine
    );
    return future.Value;
}
```
Foundation: Duplicating queue

- Queue performance is critical for work stealing
- Two parties operate on queue:
  - The worker thread pushing/popping tasks on one end
  - The thief (another thread) taking tasks from other end
- Good lock-free implementations exist, but don’t guarantee the most efficient schedule
Choosing the most efficient schedule

Let’s look at the last line of fib

```csharp
return f1.Value + f2;
```

1. If f1 has already been executed by another thread, then return the value 😊
2. If f1 is still being executed, then wait and execute another thread on the core 😞
3. If f1 has not been started yet, the execute directly

This is the common case, so Pop “should have preference over” Take
Optimizing case 3

• Usually, a work stealing queue is used for mutual exclusion – the only way to execute $f_1$ directly is to search for it on our local queue!

• Instead, we add guarantee mutual exclusion in each task (using an interlocked flag)

• Therefore, the queues can be weakened and are allowed to sometimes duplicate elements
Compensation in Task.Run

```csharp
class Task {
    enum TaskState { Init, Running, Done);
    volatile TaskState state;

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

    ...
```

- Executes each task only once
Duplicating queues

• Now that we are allowed to sometimes duplicate elements, we can implement the pop and push operations on the queue without any interlocked instructions even on weak memory architectures

• We now capture the non-determinism of the weak memory model in a benign way (i.e. by sometimes duplicating a queue element)
The DupQueue Data Structure

Queue with head and tail pointers into array

```java
class DupQueue {
    Task[] tasks;
    int size;       // size > 2
    int tailMin = ∞ // minimal index at which an element was popped
    volatile int tail = 0; volatile int head = 0;
    ...
}
```
Non-atomic Operations and Stale Data

Assume Total Store Order Memory (X86)

- Push/Pop access data without atomic operation
  - Thieves reads tasks and r/w head
  - Worker r/w tasks and r/w tail
- So
  - Thieves might see outdated tasks and tail
  - Worker might see outdated tasks and head
Possible Duplication by DupQueue.Take

The thief

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

The worker

```csharp
public Task Pop()
{
    tail = tail - 1;
    if (head <= Min(tailMin, tail))
    {
        if (tailMin > tail)
            tailMin = tail;
        Task task = tasks[tail % size];
        Tasks[tail % size] = null;
        return task;
    } else {
        lock (this)
        {
            ...
        }
    }
    ... 
}
```

• Take can return an arbitrary element, Head can be overwritten
Agenda

• Motivation
  – Demo, Features
• Gestalt
  – Language, Architecture, Work Stealing
• Foundation
  – Tasks, Replicable Tasks, Duplicating Queue
• Remarks
  – Limitations, Conclusion
Thread-local storage

- A task executes on an unspecified thread but never migrates.
- Only tasks `t` on which one waits, `t.Wait()`, can execute on the same thread stack.
- Therefore, we can still reason locally over thread local storage and use for example windows critical sections.

- A task does run in the same security and remoting context in which it was created (ie. the logical call context flows with a task)
Things to watch out for...

- **Exceptions** are parallel too! exceptions are raised as they occur, not sequentially.

- **Thread affinity**
  - UI operations can be posted (BeginInvoke)

- **Shared state**
  - You still need to do locking yourself
  - STM would really help here!
Summary

- A lightweight C# library
- Simple to use
- Abstractions to introduce parallelism
- Works with existing infrastructure

- Just covers Task parallelism: not locking or asynchronous agents.
Questions?

- http://msdn.microsoft.com/concurrency

- Thanks to: **Daan Leijen**, who is the main author of TPL, **Sebastian Burckhardt**, who verified the duplicating queue, Vance Morrison, Joe Duffy, Vinod Grover, Davor Runje, Niklas Gustafson, Alessandro Catorcini, Todd Reifsteck, Judd Hall, Sid Sidhartha, and many others.