COMP 422, Lecture 13: Single-place X10 (contd), .NET Parallel Extensions

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Outline

- **Single-place programming in X10 (contd)**
  - **Acknowledgments**
    - **PLDI 2007 tutorial on X10 by V.Saraswat, V.Sarkar, N.Nystrom**

- **.NET Parallel Extensions**
  - **Acknowledgments**
    - **"Parallel Extensions to the .NET Framework a.k.a ParallelFX", Joe Duffy**
    - **"Parallel LINQ", Igor Ostrovsky, Seattle Code Camp presentation, Jan 2008**

Review of Single-place X10

Stm:

```
async [clocked ClockList ] Stm
atomic Stm
finish Stm
next;        c.resume()        c.drop()
for ( i : Region ) Stm
foreach ( i : Region ) Stm
ateach ( I : Distribution ) Stm
```

MethodModifier:

```
atomic nonblocking sequential
```

Type:

```
nullable<Type>
future <Type >
```

x10.lang has the following classes (among others)

point, range, region, array, clock

Some of these are supported by special syntax.

Cellular Automata Simulation: Game of Life

Acknowledgment:

“Barriers”, Chapter 5.5.4, Java Concurrency in Practice, Brian Goetz et al

Game of Life – Java version (1 of 2)

java.util.concurrent version (Listing 5.15, p102, JCiP)

```
public class CellularAutomata {
  private final Board mainBoard;
  private final CyclicBarrier barrier;
  private final Worker[] workers;

  public CellularAutomata(Board board) {
    this.mainBoard = board;
    int count = Runtime.getRuntime().availableProcessors();
    this.barrier = new CyclicBarrier(count,
                         new Runnable() { // barrier action
                           public void run(){mainBoard.commitNewValues();}});
    this.workers = new Worker[count];
    for (int i = 0; i < count; i++)
      workers[i] = new Worker(mainBoard.getSubBoard(count, i));
  } // constructor

  public void start() {
    for (int i = 0; i < workers.length; i++) new Thread(workers[i]).start();
    mainBoard.waitForConvergence();
  } // start()
} // CellularAutomata
```

Game of Life – Java version (2 of 2)

```
private class Worker implements Runnable {
  private final Board board;
  public Worker(Board board) { this.board = board; }

  public void run() {
    while (!board.hasConverged()) {
      for (int x = 0; x < board.getMaxX(); x++)
        for (int y = 0; y < board.getMaxY(); y++)
          board.setNewValue(x, y, computeValue(x, y));
      try { barrier.await(); }
      catch (InterruptedException ex) { return; }
      catch (BrokenBarrierException ex) { return; }
    } // while
  } // run()

  private int computeValue(int x, int y) {
    // Compute the new value that goes in (x,y)
    . . .
  }
} // Worker
```

Game of Life – X10 version

```
public class CellularAutomata {
  private final Cell[.] mainBoard1, mainBoard2;
  public CellularAutomata(Cell[.] board) {
    mainBoard1 = board; mainBoard2 = null;
  } // constructor

  public void start() {
    finish async {
      final clock barrier = clock.factory.clock();
      foreach ( point[i] : [0:numWorkers-1] ) clocked(barrier) {
        boolean red = true;
        while ( !subBoardHasConverged(mainBoard1,mainBoard2,red) ) {
          for ( point[x,y] : myRegion(mainBoard1.region,i) )
            if ( red ) mainBoard2[x,y] = computeValue(mainBoard1, x, y);
            else mainBoard1[x,y] = computeValue(mainBoard2, x, y);
          next;
          red = ! red;
        } // while
      } // foreach
      if (! red) mainBoard1 = mainBoard2; // answer is now in mainBoard1
    } // finish async
    // All boards have now converged
  } // start()
} // CellularAutomata
```

Example of transmitting clock from parent to child

NOTE: exiting from while loop terminates activity for iteration i, and automatically deregisters activity from clock

Futures

future

Expr ::= future PlaceExpSingleListopt {Expr }

future S

- Creates a new child activity that executes statement S;
- Returns immediately.
- S may reference **final** variables in enclosing blocks.

future vs. **async**

- Return result from asynchronous computation
- Tolerate latency of remote access.

```
// shared variables
final double a[R] =  …;
final int idx = …;

// create future with a & idx
// as implicit parameters
future<double> fd =
  future { f(a[idx]); }
. . .
// Wait for result
double retval = fd.force();
```

future type

- no subtype relation between T and future<T>

future example

```
public class TutFuture1 {
    static int fib (final int n) {
        if ( n <= 0 ) return 0;
        if ( n == 1 ) return 1;
        future<int> x = future { fib(n-1) };
        future<int> y = future { fib(n-2) };
        return x.force() + y.force();
    }

    public static void main(String[] args) {
        System.out.println("fib(10) = " + fib(10));
    }
}
```

- Divide and conquer: recursive calls execute concurrently.

Example: rooted exception model (future)

```
double div (final double divisor)
   future<double> f = future { return 42.0 / divisor; }
   double result;
   try {
     result = f.force();
   } catch (ArithmeticException e) {
     result = 0.0;
   }
   return result;
}
```

- Exception is propagated when the future is forced.

Futures can deadlock

```
nullable<future<int>> f1=null;
nullable<future<int>> f2=null;

void main(String[] args) {
  f1 = future(here){a1()};
  f2 = future(here){a2()};
  f1.force();
}
```

```
int a1() {
  nullable<future<int>> tmp=null;
  do {
    tmp=f2;
  } while (tmp == null);
  return tmp.force();
}

int a2() {
  nullable<future<int>> tmp=null;
  do {
    tmp=f1;
  } while (tmp == null);
  return tmp.force();
}
```

cyclic wait condition

X10 guidelines to avoid deadlock:

- avoid futures as shared variables
- force called by same activity that created body of future, or a descendant.

Outline

- **Single-place programming in X10 (contd)**
  - **Acknowledgments**
    - **PLDI 2007 tutorial on X10 by V.Saraswat, V.Sarkar, N.Nystrom**

- **.NET Parallel Extensions**
  - **Acknowledgments**
    - **"Parallel Extensions to the .NET Framework a.k.a ParallelFX", Joe Duffy**
    - **"Parallel LINQ", Igor Ostrovsky, Seattle Code Camp presentation, Jan 2008**

Basic Parallelism in .NET 3.5

- **Work scheduling**
  - —Explicit threads – too specific, high overhead
  - —Thread pool – inefficient, scaling bottlenecks, lacking common features (wait, cancel, etc.)
  - —BackgroundWorker – still good for UI responsiveness
- **Synchronization**
  - —Monitors – Enter, Exit, Wait, Pulse, PulseAll
  - —Kernel objects – Mutex, Semaphore, AutoResetEvent, ManualResetEvent

Parallel Extensions to .NET Framework

- **Concurrency libraries for the .NET Framework**
  - **Usable from any .NET language**
- **Includes:**
  - **Common work-stealing scheduler**
  - **Task and data parallel APIs (parallel loops, parallel blocks, tasks, futures, etc)**
  - **PLINQ – data parallel queries**
  - **Communication and synchronization primitives**
- **Community Technology Preview has been released**
  - **Requires .NET Framework v3.5 (Visual Studio 2008)**

Resources for .NET Parallel Extensions

- **MSDN Dev Center downloads (http://msdn.microsoft.com/concurrency)**
  - **ParallelExtensions.zip --- contains documentation for CTP release in CHM format**
  - **ParallelExtensions_Dec07CTP.msi --- full CTP release, including documentation, samples, and the library itself**
    - **Need .NET 3.5 to install**
    - **To just extract the files (e.g. to view the samples), you can use the MSIEXEC command as follows:**

```
MSIEXEC /a ParallelExtensions_Dec07CTP.msi
TARGETDIR=C:\ParallelExtensionsCTP /qb!
```

- **ParallelFX team blog: http://blogs.msdn.com/pfxteam**
- **MSDN forums on parallel computing: http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=551&SiteID=1**

Parallel Loops

- Common source of work in sequential programs
  - for (int i = 0; i < n; i++) work(i);
  - foreach (T e in data) work(e);
- Parallelism when iterations are independent
  - Body doesn't depend on mutable state
  - E.g. static vars, writing to local vars to be used in subsequent iterations
- Using System.Threading.Parallel class:
  - Parallel.For(0, n, i => work(i));
  - Parallel.ForEach(data, e => work(e));
  - Synchronous: all finish regularly or exceptionally
    - In case of exception, further iterations are stopped from executing on a best-effort basis.
- Dynamic decomposition
  - All, some, or no iterations may run in parallel
  - If a processor becomes available, it "steals" iterations
  - Works well for both:
    - large iteration count + small work/iteration
    - small iteration count + large work/iteration

Parallelizing Blocks

- **Program statements (imperative task parallelism)**
  - ```
    {
        DoA();
        DoB();
        DoC();
    }
    ```
- **When all statements are independent, they can be parallelized (same as loops)**
  - ```
    Parallel.Do(
        () => DoA(),
        () => DoB(),
        () => DoC()
    );
    ```
- **As with For, Do is synchronous**

Explicit Task Parallelism

- All concurrency is represented as Task objects
  - Available in System.Threading.Tasks.*
  - Task t = Task.Create(o => …);
- Features of Tasks
  - Lightweight
  - Can wait for them to finish
  - Can cancel them
  - Form parent/child relationships (Parent)
  - Can get the current one (Task.Current static property)

Waiting on Tasks

- Use the Wait method to wait for completion
  - Task t = Task.Create(o=> …);
  - t.Wait(); // no timeout
  - t.Wait(250); // 250 ms timeout
- Wait for many of them:
  - TaskCoordinator.WaitAll(new Task[] { t0, t1, t2 });
- When a task completes due to exception
  - Reraised in the calling thread
  - Thrown as a wrapper AggregateException (more on next slide)
- Can also check status via IsCompleted property

Dealing with Exceptions

- Most of ParallelFX uses AggregateException
  - Has an Exception[] InnerExceptions property
  - Leaves original exceptions' stack traces intact
  - When migrating sequential code, changes "throws" contracts
- Usually incorrect to just "deal w/ the 1st one"
  - InnerExceptions[] { OOM, ArgNullEx, FileNotFoundEx }
  - Picking just one would lead to bugs
- Offers a Handle method to process certain exceptions
  -
```
try { … seq … }
catch (FooException fex) { S; }
```
  -
```
try { … par … }
catch (AggException aex) {
    aex.Handle(delegate(Exception e) {
        FooException fex = e as FooException;
        if (fex != null) { S; return true; }
        return false;
    });
}
```

Canceling Tasks

- Cancel method does three things:
  - Set IsCanceled to true
  - Delete from the runnable queue if it's not running
    - Task is canceled only if it hasn't been scheduled yet; if it's already running, then it is not interrupted
  - Wake up those waiting for it (TaskCanceledException)
- A new Task's parent is the active Task when created
  - Use RespectParentCancelation to inherit cancelations
  - IsCanceled property walks ancestor chain
- Opt-in prevents bugs due to unexpected cancels
  - A lot like ThreadInterruptedException, use with care

Task Managers

- A single global TaskManager
  - Contains policy, work queues, and threads
  - Uses work stealing and cooperative blocking
  - Very efficient for recursive queueing
- Ability to construct individual TaskManagers
  - Different policies:
    - Min, Ideal, and Max number of threads (def. 0, ProcessorCount, none)
    - Stack size (def. 1MB)
    - ExecutionContext capture/flow suppression (def. false)
  - Achieves a level of isolation from other managers
- Parallel APIs accept TaskManager as optional parameter e.g.,
  - `TaskManager myTm = new TaskManager(…);`
    `Parallel.For(…, myTm);`

Work Stealing Pros and Cons

- **Pros:**
  - More scalable queue management
  - Processor-local queues enable lock freedom
  - Tasks are lighter weight than threads
  - Intelligent runtime manages thread creation/retirement
- **Cons:**
  - Global queue is FIFO, local queues are LIFO
  - Lack of fairness can be surprising
  - Cooperative blocking can lead to delays in unblocks
  - Use of reentrancy on waiting can be surprising
  - Different model, not decided how best to surface debugging

Dataflow Parallelism with Future<T>

- Future<T> is just a Task with a Value property
- Synchronization hidden and based on data dependence
  - Accepts a Func<T> vs. Action<object>
    - `Future<T> f = Future.Create<T>(() => someT);` or
    - `Future<T> f = Future<T>.Create(() => someT);`
  - Can also create a "promise-style" Func<T>
    - No function provided at construction
    - Code just sets the Value property (or Exception for failure)
  - Accessing Value returns the value, or throws the exception
    - If not running, executes on calling thread
    - If already running, waits for it to complete

LINQ Overview

- **LINQ = Language Integrated Query**
  - **Announced by Microsoft in 2005**

- **Unified model to query data**
- **Query is a chain of operators:**
  - **Filters (Where)**
  - **Projections (Select, SelectMany)**
  - **Aggregations(Sum, Count, Aggregate, …)**
  - **Joins**
  - **Etc.**
- **Different LINQ providers handle different data sources: .NET objects, XML, SQL, web service, …**

LINQ to Objects

- Queries any data structure implementing IEnumerable
  - —All .NET collections implement IEnumerable
  - —User-writen classes can implement IEnumerable
- Example:

```
int[] array = new int[] { 3, 6, 2, 7 };
IEnumerable<int> query = array
            .Where(i => i > 5)
            .OrderBy(i => i);
```

- Same example, but using the C# query syntax:

```
int[] array = new int[] { 3, 6, 2, 7 };
IEnumerable<int> query =
      from x in array
      where x > 5
      orderby x
      select x;
```

Introducing Parallel LINQ (PLINQ)

- **Add AsParallel() to a LINQ to Objects query to create a PLINQ query:**

```
IEnumerable<int> query =
        from x in array
        where x > 5
        orderby x
        select x;
```

```
IEnumerable<int> query =
        from x in array.AsParallel()
        where x > 5
        orderby x
        select x;
```

- **PLINQ will spread out the work on all available cores**

Parallel LINQ (PLINQ)

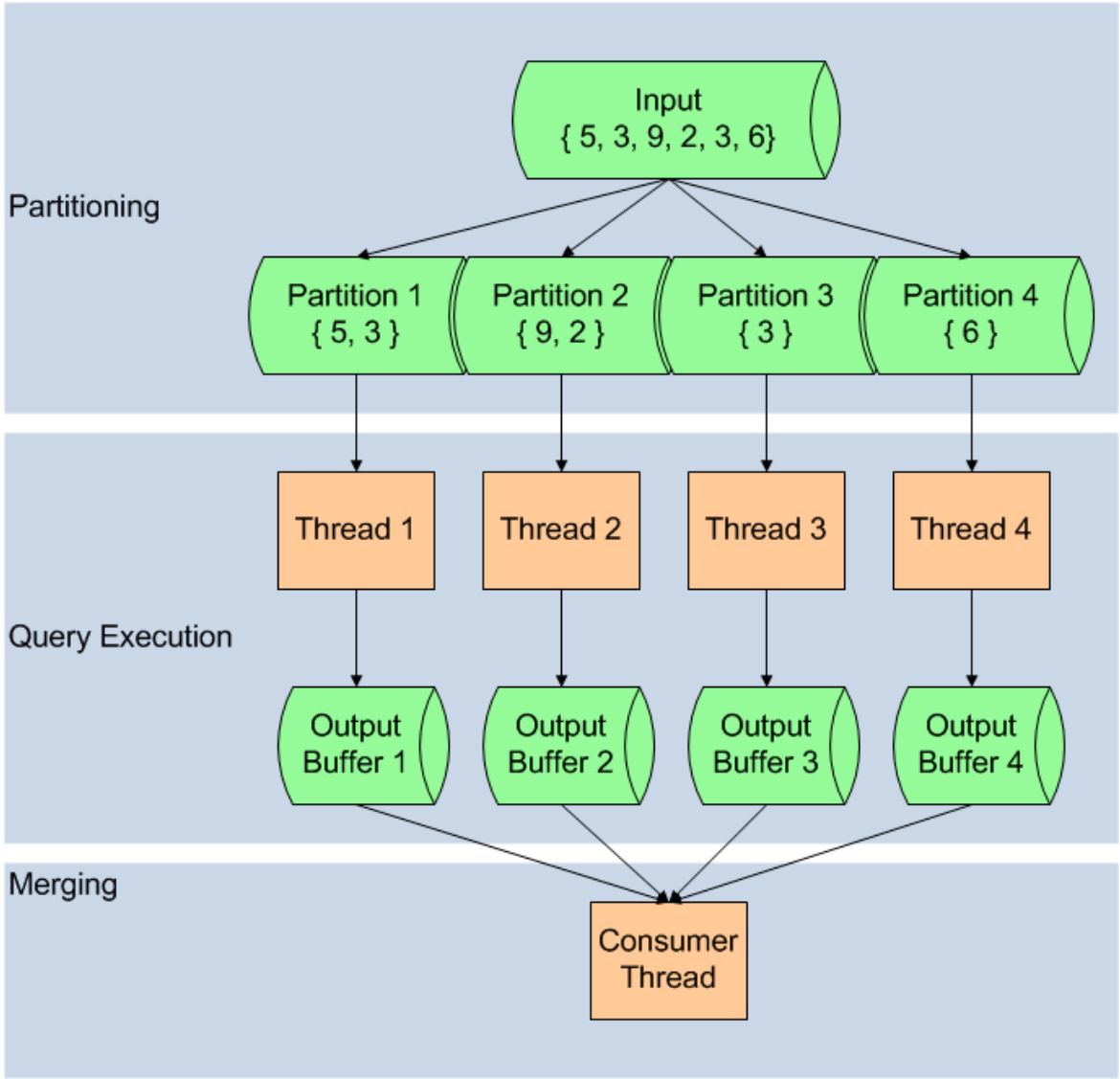
- **Declarative data parallelism via LINQ-to-Objects**
  - **PLINQ supports all LINQ operators**
    - **Select, Where, Join, GroupBy, Sum, etc.**
  - **Activated with the AsParallel extension method:**
    - **var q = from x in data where p(x) orderby k(x) select f(x);**
      var q = from x in data. AsParallel() **where p(x) orderby k(x) select f(x);**
  - **Works for any IEnumerable<T>**

- **Query syntax enables runtime to auto-parallelize**
  - **Automatic way to generate more Tasks, like Parallel**
  - **Graph analysis determines how to do it**
  - **Classic data parallelism: partitioning + pipelining**

PLINQ "Gotchas"

- **Ordering not guaranteed**
  - `int[] data = new int[] { 0, 1, 2 };`
    `var q = from x in data.AsParallel(QueryOptions.PreserveOrdering)`
    `select x * 2;`
    `int[] scaled = q.ToArray(); // == { 0, 2, 4 }?`
- **Exceptions are aggregated**
  - `object[] data = new object[] { "foo", null, null };`
    `var q = from x in data.AsParallel() select o.ToString();`
  - NullRefExceptions on data[1], data[2], or both?
  - PLINQ will always aggregate under AggregateException
- **Side effects and mutability are serious issues**
  - Most queries do not use side effects… but:
  - `var q = from x in data.AsParallel() select x.f++;`
  - OK if all elements in data are unique, else race condition

How PLINQ Works



Why LINQ To Objects != PLINQ

- **Why is LINQ to Objects not parallel by default?**
- **Parallelism inhibitors:**
  - **Side effects**
  - **Exceptions**
  - **Output ordering**
  - **Performance overhead**
  - **Thread affinity**
- **Conclusion: parallelization is opt-in** per query

Parallelism Blockers: Side Effects

- Query contains arbitrary user code
- The code may have side effects
- For example:

```
int count = 0;
var query = from x in source
        where count++ < 5
        select x;
```

- The query above works in LINQ, but would not in PLINQ
- PLINQ queries should only contain observationally pure delegates

Parallelism Blockers: Concurrent Exceptions

- **Consider query:**
  - new String[](”?”,null)
    .Select(s => int.Parse(s))
    .ToArray()
- **FormatException and NullReferenceException may be thrown concurrently!**
- **What should the user see?**
  - One of the exceptions. But which one?
    - The one to win the race
    - The first one in position in the IEnumerable
    - The most “serious” one
  - All of the exceptions, fired one by one
    - Does not seem to fit with the existing exception model
  - An aggregate exception containing all exceptions that occurred
    - Approach we took in PLINQ

Parallelism Blockers: Output Ordering

- **PLINQ merges results generated by different threads**
- **Better performance if results can be consumed without reordering**
- **Here, user may expect { 0, -1, -2 } in order:**

```
int[] arr = { 0, 1, 2 };
var q = from x in arr.AsParallel();
        select -x;
```

- **Solution: opt-in model for order**

Managing State

- **Isolation**
  - Memory space is “partitioned”
  - No two threads ever access the same state
  - Pros: no overhead, easy to reason about
  - Cons: sharing is usually needed, leading to message passing
- **Immutability**
  - Data is only read, not written (e.g. readonly fields in C#)
  - Pros: no overhead, easy to reason about
  - Cons: C# and VB encourage mutability! Difficult to program
- **Synchronization**
  - Lock access to shared state
  - Pros: flexible, programming techniques remain similar
  - Cons: perf overhead, deadlocks, races, …

Performance Tips

- Compute intensive and/or large data sets
  - Assume overhead of parallelism is ~1,000s of cycles
- Prefer isolation, immutability over synchronization
  - Synchronization == !Scalable
  - Parallel.For(0, n, i => lock(myLock) { … }); is very bad!
- Do not be gratuitous in task creation
  - Lightweight, but still requires object allocation, etc.

```
int SumTree(TreeNode n, int depth) {
    int left = 0, right = 0;
    if (depth < 8) { // 2^8 == 256 tasks
        Parallel.Do(() => left = SumTree(n.Left, depth+1),
            () => right = SumTree(n.Right, depth+1));
    } else {
        left = SumTree(n.Left, depth+1);
        right = SumTree(n.Right, depth+1);
    }
    return left + right + this.Value;
}
```

Summary: Choosing the Right Model

- **Declarative data parallelism (PLINQ) preferred**
  - —Encourages functional-style of programming
  - —Handles static nesting elegantly
  - —But not all problems can (easily) take the form of queries
- **Imperative data parallelism most common**
  - —Easier to migrate existing sequential apps
  - —But be careful: side-effects and mutability are elusive!
- **Task parallelism is great, but can be tricky**
  - —Prefer structured forms: Parallel.Do, Future<T>
  - —Unstructured tasks can lead to races, debugging headaches
  - —Most power and flexibility of the models: useful for building the next Parallel, PLINQ, etc.

BACKUP SLIDES START HERE

Continuation Passing

- **Blocking is often "bad"**
  - On GUI threads, no responsiveness
  - Burns threads
- **Continuation passing can be used instead**
  - `Task t = Task.Create(…);`
    `t.Wait();`
    `DoStuff();`
  - becomes …
  - `Task t = Task.Create(…);`
    `t.Completed += delegate { DoStuff(); };`
- **Can be difficult because the stack must be forfeited**