Patterns for Programming Shared Memory

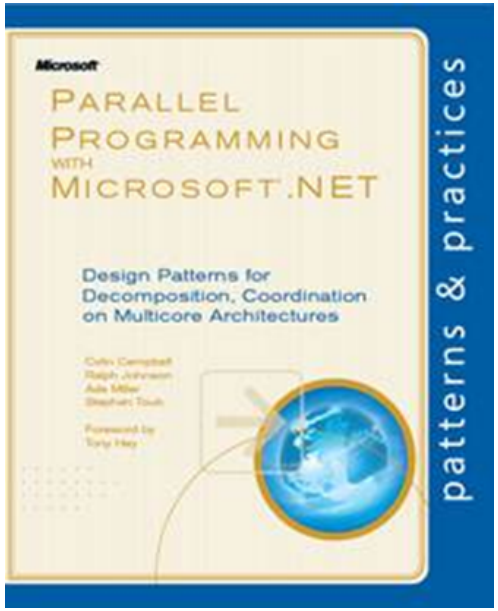
Keep Abstraction Level High

- Temptation: ad-hoc parallelization
  - Add tasks or parallel loops all over the code
  - Discover data races/deadlocks, fix them one-by-one
- Problem:
  - Complexity quickly adds up
  - Easy to get cornered by deadlocks, atomicity violations, and data races
  - And these bugs often are hard to expose
- Alternatives?
  - Use well-understood, simple high-level patterns

Book Recommendation: Parallel Programming with Microsoft .NET

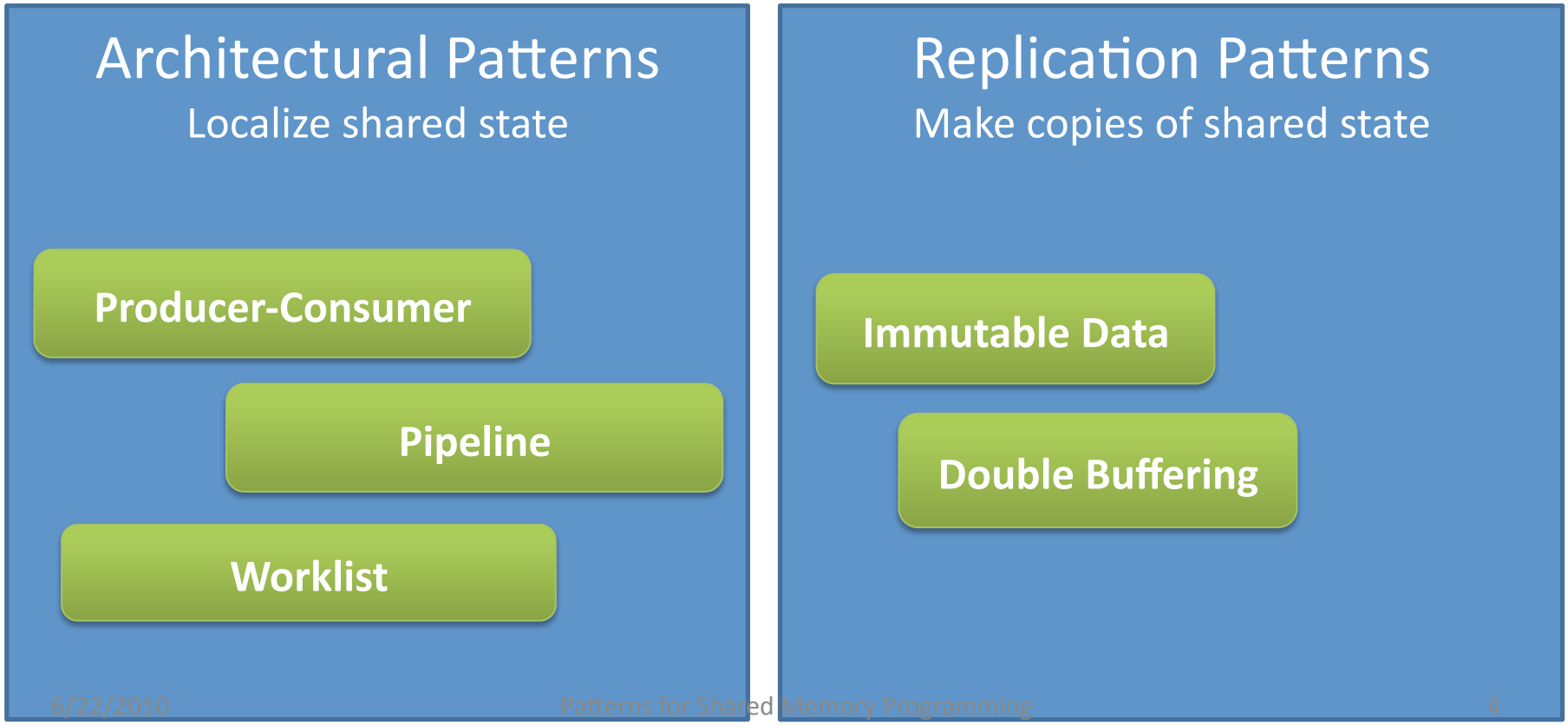
- Covers Common Patterns Systematically
- Free download at

  http://parallelpatterns.codeplex.com



Sharing State Safely

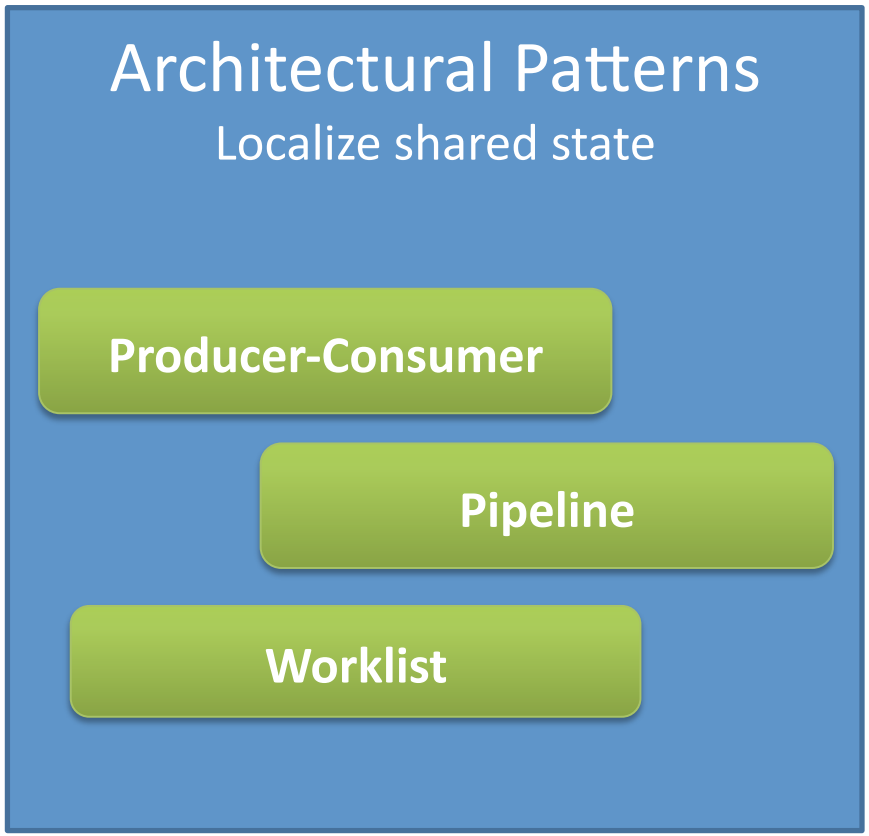
- We discuss two types of patterns in this unit
  - Not an exhaustive list, read book for more



Part I

ARCHITECTURAL PATTERNS

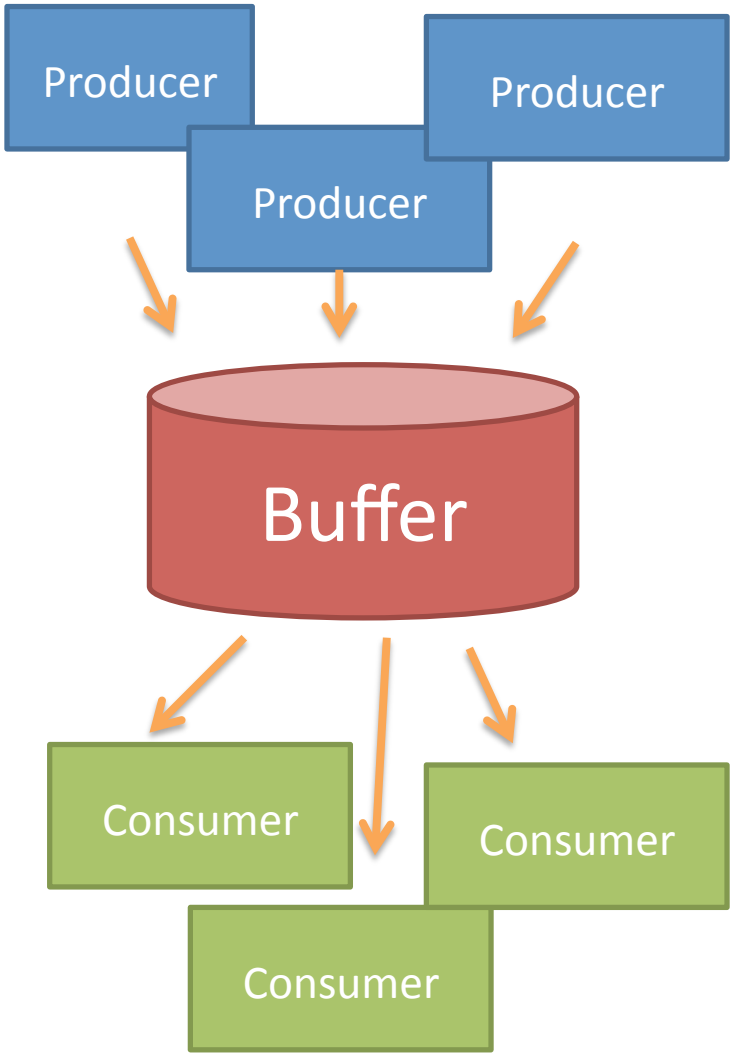
Architectural Patterns



- Visualize "factory": each worker works on a separate item
- Items travel between workers by means of buffers

Example: Producer-Consumer

- One or more producers add items to the buffer
- One or more consumers remove items from the buffer

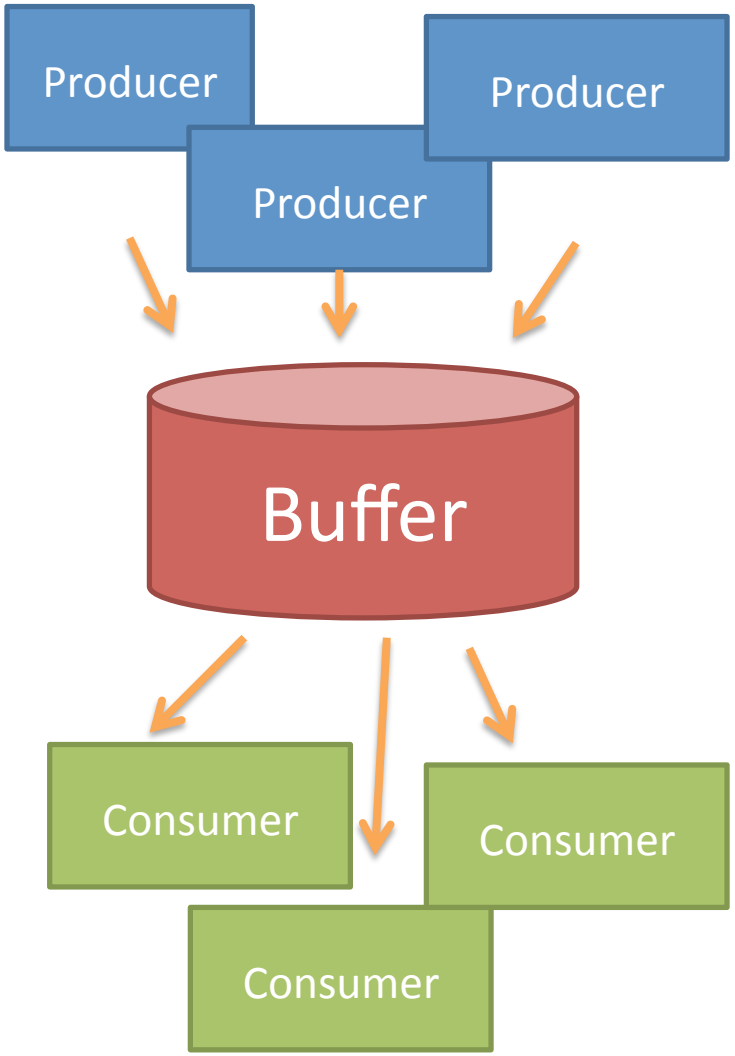


Example: Producer-Consumer

No data races!

1. Item is local to Producer before insertion into buffer
2. Item is local to Consumer after removal from buffer
3. What about buffer?
   - Buffer is thread-safe
   - Blocks when full/empty
   - Not trivial to implement
   - More about this in Unit 3
   - For now, use

     `BlockingCollection<T>`



Producer-Consumer Code

```
int buffersize = 1024;
int num_producers = 4;
int num_consumers = 4;

// create bounded buffer
var buffer = new BlockingCollection<Item>(buffersize);

// start consumers
for (int i = 0; i < num_consumers; i++)
   Task.Factory.StartNew(() => new Consumer(buffer).Run());

// start producers
for (int i = 0; i < num_producers; i++)
   Task.Factory.StartNew(() => new Producer(buffer).Run());
```

Producer Code

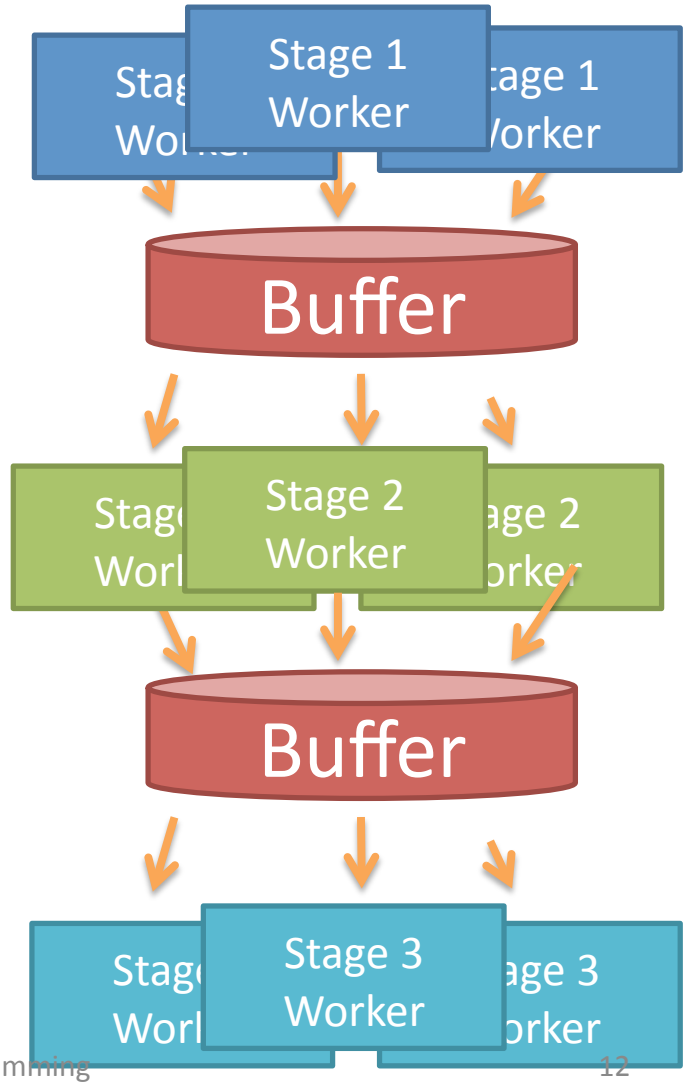
```
class Producer
{
    public Producer(BlockingCollection<Item>
buffer)
    {
        this.buffer = buffer;
    }
    public void Run()
    {
        while (!Done())
        {
            Item i = Produce();
            buffer.Add(i);
        }
    }
    private BlockingCollection<Item> buffer;
    private Item Produce() { … }
    private bool Done() { … }
}
```

Consumer Code

```
class Consumer
{
    public Consumer(BlockingCollection<Item> buffer)
    {
        this.buffer = buffer;
    }
    private BlockingCollection<Item> buffer;

    public void Run()
    {
        foreach (Item i in buffer.GetConsumingEnumerable())
            Consume(i);
    }
    private void Consume(Item item)
    {
      ...
    }
}
```

Pipeline Pattern

- Generalization of Producer-Consumer
  - One or more workers per stage
  - First stage = Producer
  - Last stage = Consumer
  - Middle stages consume and produce
  - No Data Races: Data is local to workers



Pipeline Code

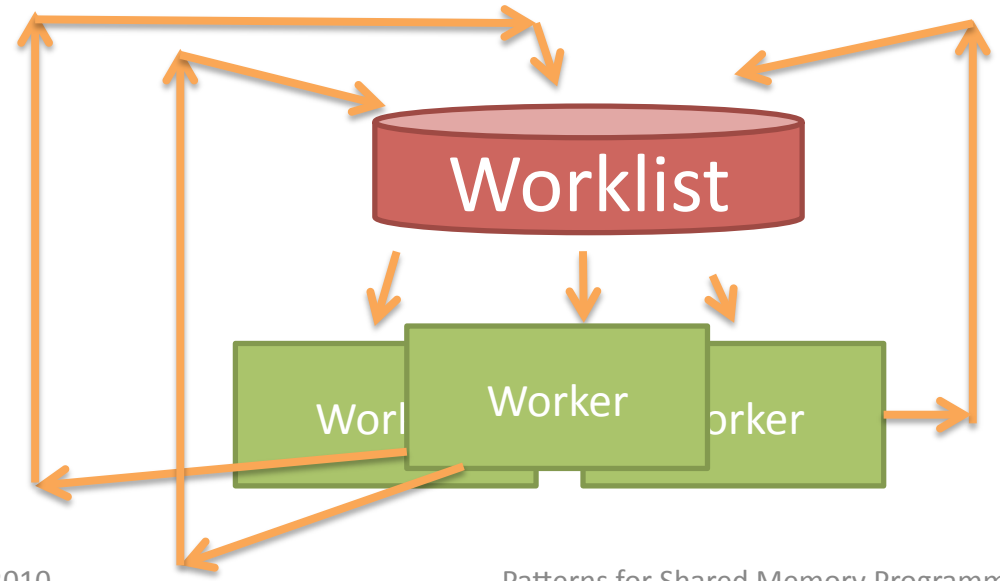
```
// create bounded buffers
var buffer_1_2 = new BlockingCollection<Item>(buffersize);
var buffer_2_3 = new BlockingCollection<Item>(buffersize);

// start workers
for (int i = 0; i < num_workers_stage1; i++)
   Task.Factory.StartNew(() =>
      new Stage1Worker(buffer_1_2).Run());
for (int i = 0; i < num_workers_stage2; i++)
   Task.Factory.StartNew(() =>
      new Stage2Worker(buffer_1_2, buffer_2_3).Run());
for (int i = 0; i < num_workers_stage3; i++)
   Task.Factory.StartNew(() =>
      new Stage3Worker(buffer_2_3).Run());
```

Stage Worker Code

```
class Stage2Worker {
   public Stage2Worker(BlockingCollection<Item> buffer_in,
                       BlockingCollection<Item> buffer_out) {
      this.buffer_in = buffer_in;
      this.buffer_out = buffer_out;
   }

    public void Run() {
      try {
         foreach (Item i in buffer_in.GetConsumingEnumerable()) {
            Item j = Process(i);
            buffer_out.Add(j);
         }
      }
      finally { buffer_out.CompleteAdding(); }
   }
   . . .
```

Worklist Pattern

- Worklist contains items to process
  - Workers grab one item at a time
  - Workers may add items back to worklist
  - No data races: items are local to workers



Worklist Code (1/2)

```
public void Run()
{
    int num_workers = 4;

    // create worklist, filled with initial work
    worklist = new BlockingCollection<Item>(
        new ConcurrentQueue<Item>(GetInitialWork()));

    cts = new CancellationTokenSource();
    itemcount = worklist.Count();

    for (int i = 0; i < num_workers; i++)
        Task.Factory.StartNew(RunWorker);
}

IEnumerable<Item> GetInitialWork() { … }
```

```
BlockingCollection<Item> worklist;
CancellationTokenSource cts;
int itemcount;
```

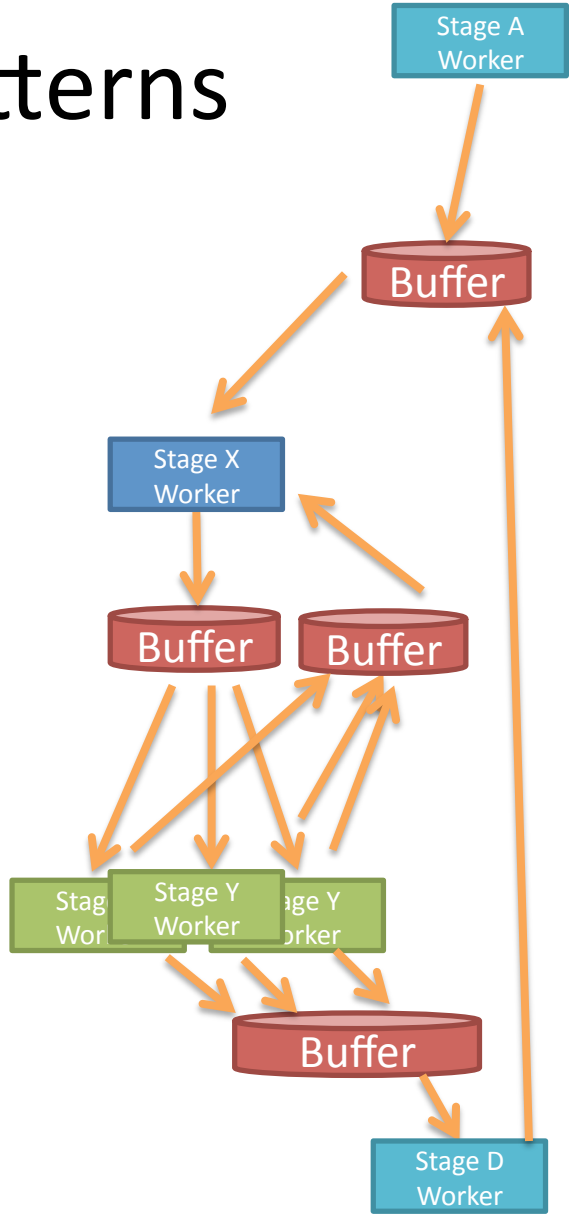
Worklist Code (2/2)

```
BlockingCollection<Item> worklist;
CancellationTokenSource cts;
int itemcount;
```

```
public void RunWorker() {
    try {
       do {
          Item i = worklist.Take(cts.Token);
          // blocks until item available or cancelled
          Process(i);
          // exit loop if no more items left
        } while (Interlocked.Decrement(ref itemcount) > 0);
    } finally {
        if (! cts.IsCancellationRequested)
           cts.Cancel();
    }
}
public void Process(Item i) { . . . }
// may call AddWork() to add more work to worklist
public void AddWork(Item item) {
    Interlocked.Increment(ref itemcount);
    worklist.Add(item);
}
```

Application Architecture May Combine Patterns

- "Stages" arranged in a graph and may have
  - Single worker (only inter-stage parallelism)
  - Multiple workers (also intra-stage parallelism)
- No data races: items local to worker
- Buffers may have various ordering characteristics
  - ConcurrentQueue = FIFO
  - ConcurrentStack = LIFO
  - ConcurrentBag = unordered



Provisioning

- How to choose number of workers?
  - Single worker
    - If ordering of items matter
  - Workers are CPU-bound
    - Can not utilize more than one per processor
      `num_workers = Environment.ProcessorCount`
  - Workers block in synchronous IO
    - Can potentially utilize many more
- How to choose buffer size?
  - Too small -> forces too much context switching
  - Too big -> uses inappropriate amount of memory

Manual Provisioning

```
BlockingCollection<Item> buffer;
class Item { … }
void Consume(Item i) { … }

public void FixedConsumers() {
   int num_workers = Environment.ProcessorCount;
   for (int i = 0; i < num_workers; i++)
      Task.Factory.StartNew(() => {
         foreach (Item item in buffer.GetConsumingEnumerable())
            Consume(item);
      });
 }
```

Automatic Provisioning

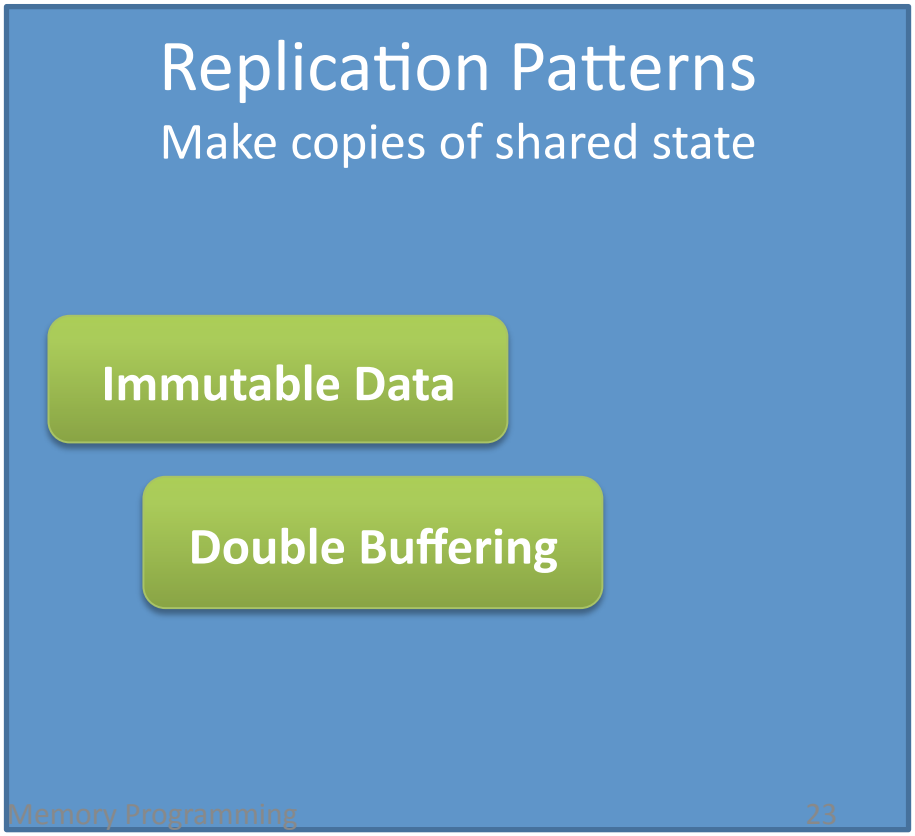
```
BlockingCollection<Item> buffer;
class Item { … }
void Consume(Item i) { … }



public void FlexibleConsumers()
{
    Parallel.ForEach(
        new BlockingCollectionPartitioner<Item>(buffer),
        Consume);
}

private class BlockingCollectionPartitioner<T> : Partitioner<T>
{
    // see http://blogs.msdn.com/b/pfxteam/archive/2010/04/06/9990420.aspx
}
```

Part II

REPLICATION PATTERNS

Replication Patterns

- Idea: Many workers can work on the "same" item if we make copies



Immutability

- Remember: concurrent reads do not conflict
- Idea: never write to shared data
  - All shared data is immutable (read only)
  - To modify data, must make a fresh copy first
- Example: strings
  - Strings are immutable!
  - Never need to worry about concurrent access of strings.

Build your own immutable objects

- Mutable list: changes original

```
interface IMutableSimpleList<T>
{
  int Count { get; }  // read only property
  T this[int index] { get; } // read only indexer
  void Add(T item);   // mutator method
}
```

- Immutable list: never changes original, copies on mutation

```
interface IImmutableSimpleList<T>
{
  int Count { get; }  // read only property
  T this[int index] { get; } // read only indexer
  ImmutableSimplelist<T> Add(T item); // read only method
}
```

Copy on Write

```
class SimpleImmutableList<T> : IImmutableSimpleList<T> {
    private T[] _items = new T[0];

    public int Count { get { return _items.Length; } }
    public T this[int index] { get { return _items
[index]; } }

    public IImmutableSimpleList<T> Add(T item)
    {
        T[] items = new T[_items.Length + 1];
        Array.Copy(_items, items, _items.Length);
        items[items.Length - 1] = item;

        return new SimpleImmutableList<T>(items);
    }
}
```

Example: Game

```
class GameObject {
    public void Draw()
    { // ... reads state
    }
    public void Update()
    { // ... modifies state
    }
}

GameObject[] objects
 = new GameObject[numobjects];

void DrawAllCells()
{
   foreach(GameObject g in objects)
      g.Draw();
}

void UpdateAllCells()
{
   foreach(GameObject g in objects)
      g.Update();
}
```

```
// GOAL: draw and update in parallel
void GameLoop()
{
    while (!Done()) {
        DrawAllCells();
        UpdateAllCells();
    }
}
```

```
// incorrect parallelization:
// data race between draw and update
void ParallelGameLoop()
{
    while (!Done())
        Parallel.Invoke(DrawAllCells,
                        UpdateAllCells)
}
```

Can we use immutability to fix this?

Example: Game

```
class GameObject {
    public void Draw()
    { // ... reads state
    }
    public GameObject ImmutableUpdate()
    { // ... return updated copy
    }
}

GameObject[] objects
   = new GameObject[numobjects];

void DrawAllCells()
{
   foreach(GameObject g in objects)
      g.Draw();
}

GameObject[] UpdateAllCells()
{
    var newobjects = new GameObject[numobjects];
    for (int i = 0; i < numobjects; i++)
        newobjects[i] = objects[i].ImmutableUpdate();
    return newObjects;
}
```

```
// correct parallelization of loop
void ParallelGameLoop()
{
    while (!Done())
    {
        GameObject[] newarray = null;
        Parallel.Invoke(() => DrawAllCells(),
          () => newarray = UpdateAllCells());
        objects = newarray;
    }
}
```

Correct parallelization.

Can we do this without allocating a new array every iteration?

Trick: Double Buffering

```
class GameObject {
    public void Draw()
    { // ... reads state
    }
    public GameObject ImmutableUpdate()
    { // ...
    }
}

GameObject[,] objects
   = new GameObject[2, numobjects];

void DrawAllCells(int bufferIdx)
{
    for (int i = 0; i < numobjects; i++)
      objects[bufferIdx, i].Draw();
}

void UpdateAllCells(int bufferIdx)
{
    for (int i = 0; i < numobjects; i++)
      objects[1- bufferIdx, i] = objects[bufferIdx, i].ImmutableUpdate();
}
```

```
// correct parallelization of loop
void ParallelGameLoop()
{
    int bufferIdx = 0;
    while (!Done())
    {
        Parallel.Invoke(
            () => DrawAllCells(bufferIdx),
            () => UpdateAllCells(bufferIdx)
        );
        bufferIdx = 1 - bufferIdx;
    }
}
```