softserve experience matters

Data Parallelism Patterns

Task Parallel Library

2016-07-19 by O. Shvets
Reviewed by S. Diachenko

- Introduction to Parallel Programming
- Parallel Loops
- Parallel Aggregation

- **Introduction to Parallel Programming**
- Parallel Loops
- Parallel Aggregation

Multicore system features

- Hardware trends predict more cores instead of faster clock speeds
- One core is loaded at 100%, the rest of the cores are inactive
- Solution - parallel multithreaded programming
- The complexities of multithreaded programming
  - Thread creation
  - Thread synchronization
  - Hard reproducible bugs

Potential parallelism

- Some parallel applications can be written for specific hardware
  - For example, creators of programs for a console gaming platform have detailed knowledge about the hardware resources that will be available at run time (number of cores, details of the memory architecture)
- In contrast, when you write programs that run on general-purpose computing platforms you may not always know how many cores will be available
- With potential parallelism applications will run
  - fast on a multicore architecture
    - The degree of parallelism is not encoded tough to get a win on the future multicore processors
  - the same speed as a sequential program when there is only one core available

Parallel programming patterns aspects

- Decomposition
- Coordination
- Scalable Sharing

Decomposition

- Tasks are sequential operations that work together to perform a larger operation
- Tasks are not threads
  - While a new thread immediately introduces additional concurrency to your application, a new task introduces only the potential for additional concurrency
  - A task's potential for additional concurrency will be realized only when there are enough available cores
- Tasks must be
  - large (running for a long time)
  - independent
  - numerous (to load all the cores)

Coordination

- Tasks that are independent of one another can run in parallel
- Some tasks can begin only after other tasks complete
- The order of execution and the degree of parallelism are constrained by
  - control flow (the steps of the algorithm)
  - data flow (the availability of inputs and outputs)
- The way tasks are coordinated depends on which parallel pattern you use

Scalable sharing of data

- Tasks often need to share data
- Synchronization of tasks
  - Every form of synchronization is a form of serialization
  - Adding synchronization (locks) can reduce the scalability of your application
  - Locks are error prone (deadlocks) but necessary in certain situations (as the goto statements of parallel programming)
- Scalable data sharing techniques:
  - use of immutable, readonly data
  - limiting your program's reliance on shared variables
  - introducing new steps in your algorithm that merge local versions of mutable state at appropriate checkpoints
- Techniques for scalable sharing may involve changes to an existing algorithm

Parallel programming design approaches

- Understand your problem or application and look for potential parallelism across the entire application as a whole
- Think in terms of data structures and algorithms; don’t just identify bottlenecks
- Use patterns

Concurrency & parallelism

- Concurrency is a concept related to multitasking and asynchronous input-output (I/O)
  - multiple threads of execution that may each get a slice of time to execute before being preempted by another thread, which also gets a slice of time
  - to react to external stimuli such as user input, devices, and sensors
  - operating systems and games, by their very nature, are concurrent, even on one core
- The goal of concurrency is to prevent thread starvation

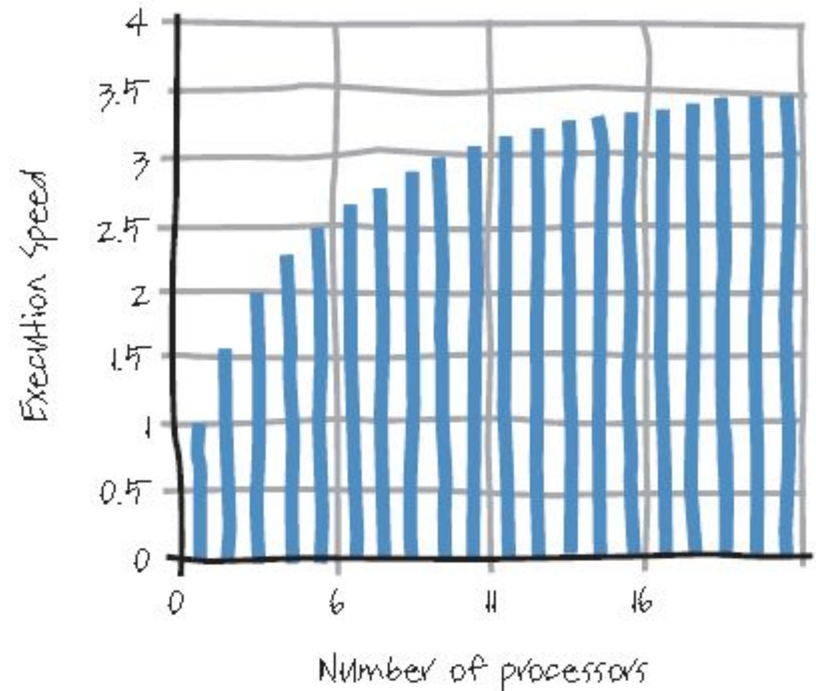
Concurrency & parallelism

- With parallelism, concurrent threads execute at the same time on multiple cores
- Parallel programming focuses on improving the performance of applications that use a lot of processor power and are not constantly interrupted when multiple cores are available
- The goal of parallelism is to maximize processor usage across all available cores

softserve experience matters

The limits of parallelism

- Amdahl's law says that no matter how many cores you have, the maximum speedup you can ever achieve is (1 / percent of time spent in sequential processing)



Parallel programming tips

- Whenever possible, stay at the highest possible level of abstraction and use constructs or a library that does the parallel work for you
- Use your application server's inherent parallelism; for example, use the parallelism that is incorporated into a web server or database
- Use an API to encapsulate parallelism, such as Microsoft Parallel Extensions for .NET (TPL and PLINQ)
  - These libraries were written by experts and have been thoroughly tested; they help you to avoid many of the common problems that arise in parallel programming
- Consider the overall architecture of your application when thinking about how to parallelize it

Parallel programming tips

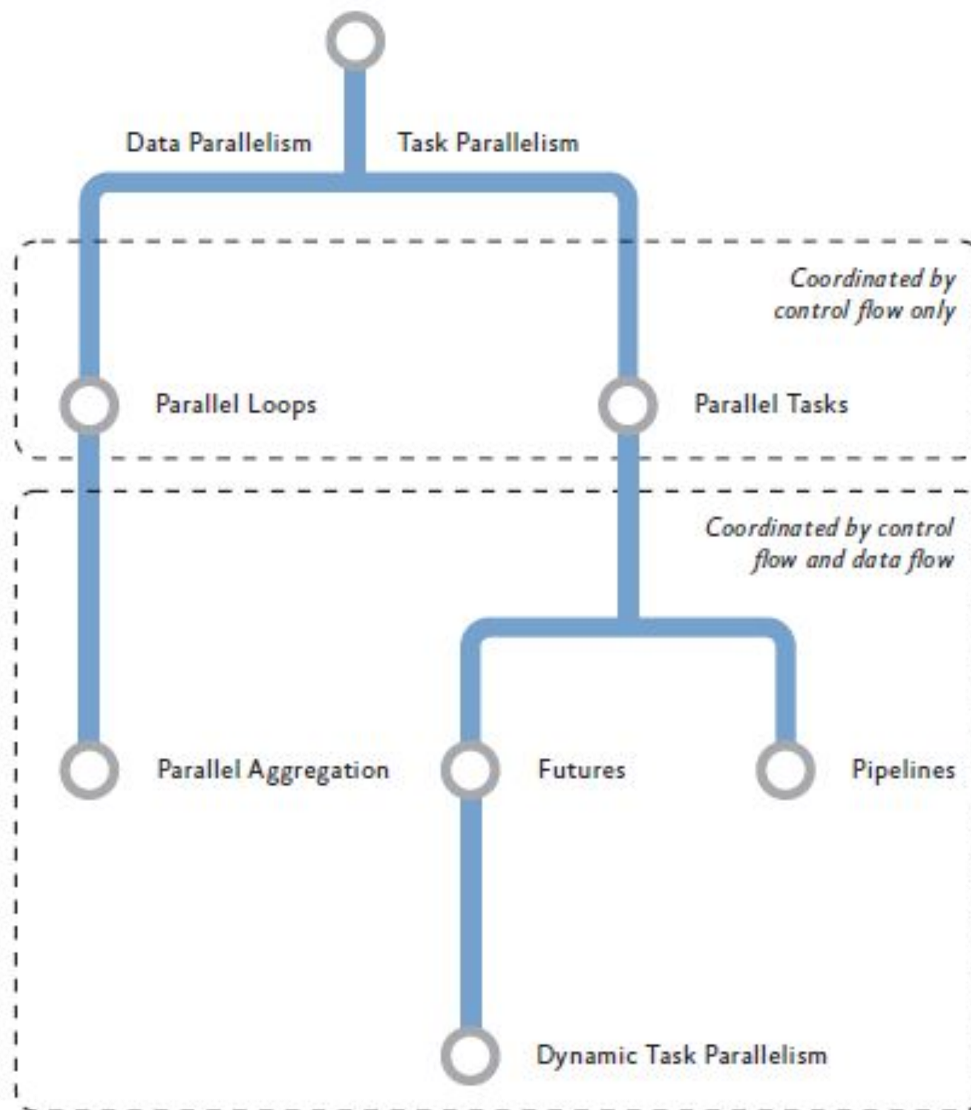
- Use patterns
- Restructuring your algorithm (for example, to eliminate the need for shared data) is better than making low-level improvements to code that was originally designed to run serially
- Don’t share data among concurrent tasks unless absolutely necessary
  - If you do share data, use one of the containers provided by the API you are using, such as a shared queue
- Use low-level primitives, such as threads and locks, only as a last resort
  - Raise the level of abstraction from threads to tasks in your applications

Code examples of this presentation

- Based on the .NET Framework 4
- Written in C #
- Use
  - Task Parallel Library (TPL)
  - Parallel LINQ (PLINQ)

- Introduction to Parallel Programming
- **Parallel Loops**
- Parallel Aggregation

Parallel programming patterns



Parallel Loops

- Use the Parallel Loop pattern when you need to perform the same independent operation for each element of a collection or for a fixed number of iterations
  - The steps of a loop are independent if they don't write to memory locations or files that are read by other steps
  - The word "independent" is a key part of the definition of this pattern
- Unlike a sequential loop, the order of execution isn't defined for a parallel loop

Parallel.For

```
int n = ...
for (int i = 0; i < n; i++)
{
   // ...
}
```

```
int n = ...
Parallel.For(0, n, i =>
{
   // ...
});
```

```
Parallel.For(int fromInclusive,
             int toExclusive,
             Action<int> body);
```

Parallel.ForEach

```
IEnumerable<MyObject> myEnumerable = ...

foreach (var obj in myEnumerable)
{
   // ...
}
```

```
IEnumerable<MyObject> myEnumerable = ...

Parallel.ForEach(myEnumerable, obj =>
{
   // ...
});
```

```
ForEach<TSource>(IEnumerable<TSource> source,
                 Action<TSource> body);
```

Parallel LINQ (PLINQ)

- Almost all LINQ-to-Objects expressions can easily be converted to their parallel counterpart by adding a call to the AsParallel extension method
- PLINQ's ParallelEnumerable class has close to 200 extension methods that provide parallel queries for ParallelQuery<T> objects

```
IEnumerable<MyObject> source = ...

// LINQ
var query1 = from i in source select Normalize(i);

// PLINQ
var query2 = from i in source.AsParallel()
                                    select Normalize(i);
```

PLINQ ForAll

- Use PLINQ’s ForAll extension method in cases where you want to iterate over the input values but you don’t want to select output values to return
- The ForAll extension method is the PLINQ equivalent of Parallel.ForEach
- It’s important to use PLINQ’s ForAll extension method instead of giving a PLINQ query as an argument to the Parallel.ForEach method

```
IEnumerable<MyObject> myEnumerable = ...

myEnumerable.AsParallel().ForAll(obj => DoWork(obj));
```

Exceptions

- The .NET implementation of the Parallel Loop pattern ensures that exceptions that are thrown during the execution of a loop body are not lost
  - For both the Parallel.For and Parallel.ForEach methods as well as for PLINQ, exceptions are collected into an AggregateException object and rethrown in the context of the calling thread

Parallel loops variations

- Parallel loops
  - 12 overloaded methods for Parallel.For
  - 20 overloaded methods for Parallel.ForEach
  - PLINQ has close to 200 extension methods
- Parallel loops options
  - a maximum degree of parallelism
  - hooks for external cancellation
  - monitor the progress of other steps (for example, to see if exceptions are pending)
  - manage task-local state

Dependencies between loop iterations

- Writing to shared variables

```
for(int i = 1; i < n; i++)
  total += data[i];
```

- Using properties of an object model

```
for(int i = 0; i < n; i++)
  SomeObject[i].Parent.Update();
```

Dependencies between loop iterations

- Referencing data types that are not thread safe
- Loop-carried dependence

```
for(int i = 1; i < N; i++)
  data[i] = data[i] + data[i - 1];
```

  - Sometimes, it’s possible to use a parallel algorithm in cases of loop-carried dependence (parallel scan and parallel dynamic programming are examples of these patterns)

Breaking out of loops early

- Sequential iteration

```
int n = ...
for (int i = 0; i < n; i++)
{
  // ...
  if (/* stopping condition is true */)
    break;
}
```

- With parallel loops more than one step may be active at the same time, and steps of a parallel loop are not necessarily executed in any predetermined order

Parallel Break

- Use Break to exit a loop early while ensuring that lower-indexed steps complete

```
int n = ...
Parallel.For(0, n, (i, loopState) =>
{
  // ...
  if (/* stopping condition is true */)
  {
    loopState.Break();
    return;
  }
});
```

```
Parallel.For(int fromInclusive,
             int toExclusive,
             Action<int, ParallelLoopState> body);
```

Parallel Break

- Calling Break doesn’t stop other steps that might have already started running
- To check for a break condition in long-running loop bodies and exit that step immediately
  - ParallelLoopState.LowestBreakIteration.HasValue == true
  - ParallelLoopState.ShouldExitCurrentIteration == true
- Because of parallel execution, it’s possible that more than one step may call Break
  - In that case, the lowest index will be used to determine which steps will be allowed to start after the break occurred
- The Parallel.ForEach method also supports the loop state Break method

ParallelLoopResult

```
int n = ...
var result = new double[n];

var loopResult = Parallel.For(0, n, (i, loopState) =>
{
   if (/* break condition is true */)
   {
      loopState.Break();
      return;
   }
   result[i] = DoWork(i);
});

if (!loopResult.IsCompleted &&
        loopResult.LowestBreakIteration.HasValue)
{
   Console.WriteLine("Loop encountered a break at {0}",
                         loopResult.LowestBreakIteration.Value);
}
```

Parallel Stop

- Use Stop to exit a loop early when you don’t need all lower-indexed iterations to run before terminating the loop

```
var n = ...
var loopResult = Parallel.For(0, n, (i, loopState) =>
{
   if (/* stopping condition is true */)
   {
      loopState.Stop();
      return;
   }
   result[i] = DoWork(i);
});

if (!loopResult.IsCompleted &&
     !loopResult.LowestBreakIteration.HasValue)
{
   Console.WriteLine(“Loop was stopped”);
}
```

External Loop Cancellation

```
void DoLoop(CancellationTokenSource cts)
{
  int n = ...
  CancellationToken token = cts.Token;

  var options = new ParallelOptions
                       { CancellationToken = token };

  try
  {
    Parallel.For(0, n, options, (i) =>
    {
      // ...

      // ... optionally check to see if cancellation happened
      if (token.IsCancellationRequested)
      {
         // ... optionally exit this iteration early
         return;
      }
    });
  }
  catch (OperationCanceledException ex)
  {
     // ... handle the loop cancellation
  }
}
```

Special handling of small loop bodies

```
int n = ...
double[] result = new double[n];
Parallel.ForEach(Partitioner.Create(0, n),
    (range) =>
    {
       for (int i = range.Item1; i < range.Item2; i++)
       {
         // very small, equally sized blocks of work
         result[i] = (double)(i * i);
       }
    });
```

```
Parallel.ForEach<TSource>(
     Partitioner<TSource> source,
     Action<TSource> body);
```

Special handling of small loop bodies

- The number of ranges that will be created by a Partitioner object depends on the number of cores in your computer
- The default number of ranges is approximately three times the number of those cores
- You can use an overloaded version of the Partitioner.Create method that allows you to specify the size of each range

```
double[] result = new double[1000000];
Parallel.ForEach(Partitioner.Create(0, 1000000, 50000),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
          // small, equally sized blocks of work
          result[i] = (double)(i * i);
        }
    });
```

Controlling the degree of parallelism

- You usually let the system manage how iterations of a parallel loop are mapped to your computer's cores, in some cases, you may want additional control
  - Reducing the degree of parallelism is often used in performance testing to simulate less capable hardware
  - Increasing the degree of parallelism to a number larger than the number of cores can be appropriate when iterations of your loop spend a lot of time waiting for I/O operations to complete

```
var n = ...
var options = new ParallelOptions()
                      { MaxDegreeOfParallelism = 2};
Parallel.For(0, n, options, i =>
{
  // ...
});
```

Controlling the degree of parallelism

- The PLINQ query in the code example will run with a maximum of eight tasks at any one time

```
IEnumerable<T> myCollection = // ...

myCollection.AsParallel()
            .WithDegreeOfParallelism(8)
            .ForAll(obj => /* ... */);
```

- If you specify a larger degree of parallelism, you may also want to use the ThreadPool class's SetMinThreads method so that these threads are created without delay

Task-local state in a loop body

- Sometimes you need to maintain thread-local state during the execution of a parallel loop
  - For example, you might want to use a parallel loop to initialize each element of a large array with random values

```
ForEach<TSource, TLocal>(
   OrderablePartitioner<TSource> source,
   ParallelOptions parallelOptions,
   Func<TLocal> localInit,
   Func<TSource, ParallelLoopState, TLocal, TLocal> body,
   Action<TLocal> localFinally)
```

Random initialization of the large array

```
int numberOfSteps = 10000000;
double[] result = new double[numberOfSteps];

Parallel.ForEach(

     Partitioner.Create(0, numberOfSteps),

     new ParallelOptions(),

     () => { return new Random(MakeRandomSeed()); },

     (range, loopState, random) =>
     {
       for (int i = range.Item1; i < range.Item2; i++)
         result[i] = random.NextDouble();
       return random;
     },

     _ => {});
```

Random class in parallel

- Calling the default Random constructor twice in short succession may use the same random seed
  - Provide your own random seed to prevent duplicate random sequences
- The Random class isn’t the right random generator for all simulation scenarios
- If your application really requires statistically robust pseudorandom values, you should consider using the RNGCryptoService Provider class or a third-party library

Using a custom task scheduler

- You can substitute custom task scheduling logic for the default task scheduler that uses ThreadPool worker threads

```
int n = ...
TaskScheduler myScheduler = ...
var options = new ParallelOptions()
                      { TaskScheduler = myScheduler };
Parallel.For(0, n, options, i =>
{
  // ...
});
```

- It isn’t possible to specify a custom task scheduler for PLINQ queries

Anti-Patterns

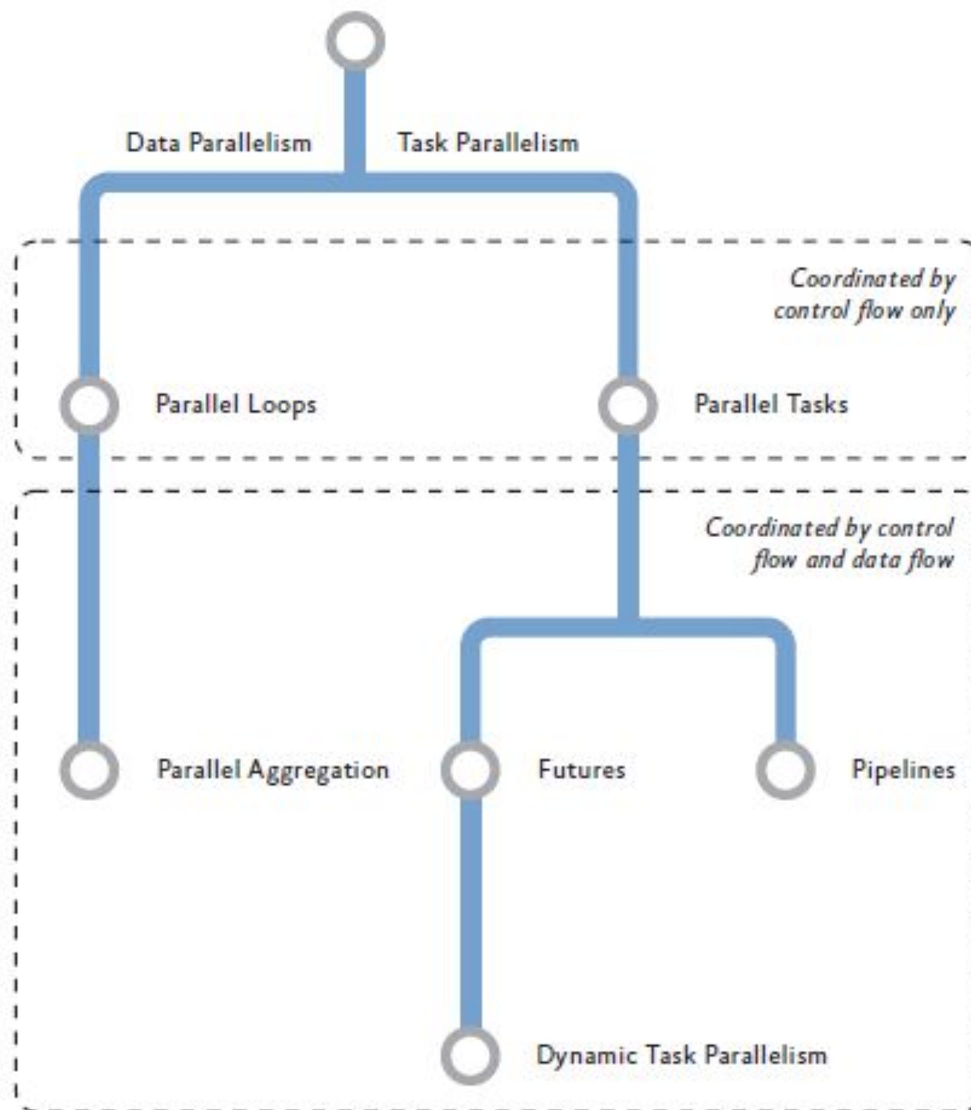
- Step size other than one
- Hidden loop body dependencies
- Small loop bodies with few iterations
- Processor oversubscription and undersubscription
- Mixing the Parallel class and PLINQ
- Duplicates in the input enumeration

Parallel loops design notes

- Adaptive partitioning
  - Parallel loops in .NET use an adaptive partitioning technique where the size of units of work increases over time
  - Adaptive partitioning is able to meet the needs of both small and large input ranges
- Adaptive concurrency
  - The Parallel class and PLINQ work on slightly different threading models in the .NET Framework 4
- Support for nested loops
- Support for server applications
  - The Parallel class attempts to deal with multiple AppDomains in server applications in exactly the same manner that nested loops are handled
  - If the server application is already using all the available thread pool threads to process other ASP.NET requests, a parallel loop will only run on the thread that invoked it

- Introduction to Parallel Programming
- Parallel Loops
- **Parallel Aggregation**

Parallel programming patterns



The Parallel Aggregation pattern

- The pattern is more general than calculating a sum
  - It works for any binary operation that is associative
  - The .NET implementation expects the operations to be commutative
- The pattern uses unshared, local variables that are merged at the end of the computation to give the final result
  - Using unshared, local variables for partial, locally calculated results is how the steps of a loop can become independent of each other
  - Parallel aggregation demonstrates the principle that it’s usually better to make changes to your algorithm than to add synchronization primitives to an existing algorithm

Calculating a sum

- Sequential version

```
double[] sequence = ...
double sum = 0.0d;
for (int i = 0; i < sequence.Length; i++)
{
  sum += Normalize(sequence[i]);
}
return sum;
```

- LINQ expression

```
double[] sequence = ...
return (from x in sequence select Normalize(x)).Sum();
```

Calculating a sum

- PLINQ

```
double[] sequence = ...
return (from x in sequence.AsParallel()
        select Normalize(x)).Sum();
```

  - PLINQ has query operators that count the number of elements and calculate the average, maximum, or minimum
  - PLINQ also has operators that create and combine sets (duplicate elimination, union, intersection, and difference), transform sequences (concatenation, filtering, and partitioning) and group (projection)
  - If PLINQ's standard query operators aren't what you need, you can also use the Aggregate extension method to define your own aggregation operators

```
double[] sequence = ...
return (from x in sequence.AsParallel() select Normalize(x))
             .Aggregate(1.0d, (y1, y2) => y1 * y2);
```

Parallel aggregation pattern in .NET

- PLINQ is usually the recommended approach
- You can also use Parallel.For or Parallel.ForEach to implement the parallel aggregation pattern
  - The Parallel.For and Parallel.ForEach methods require more complex code than PLINQ

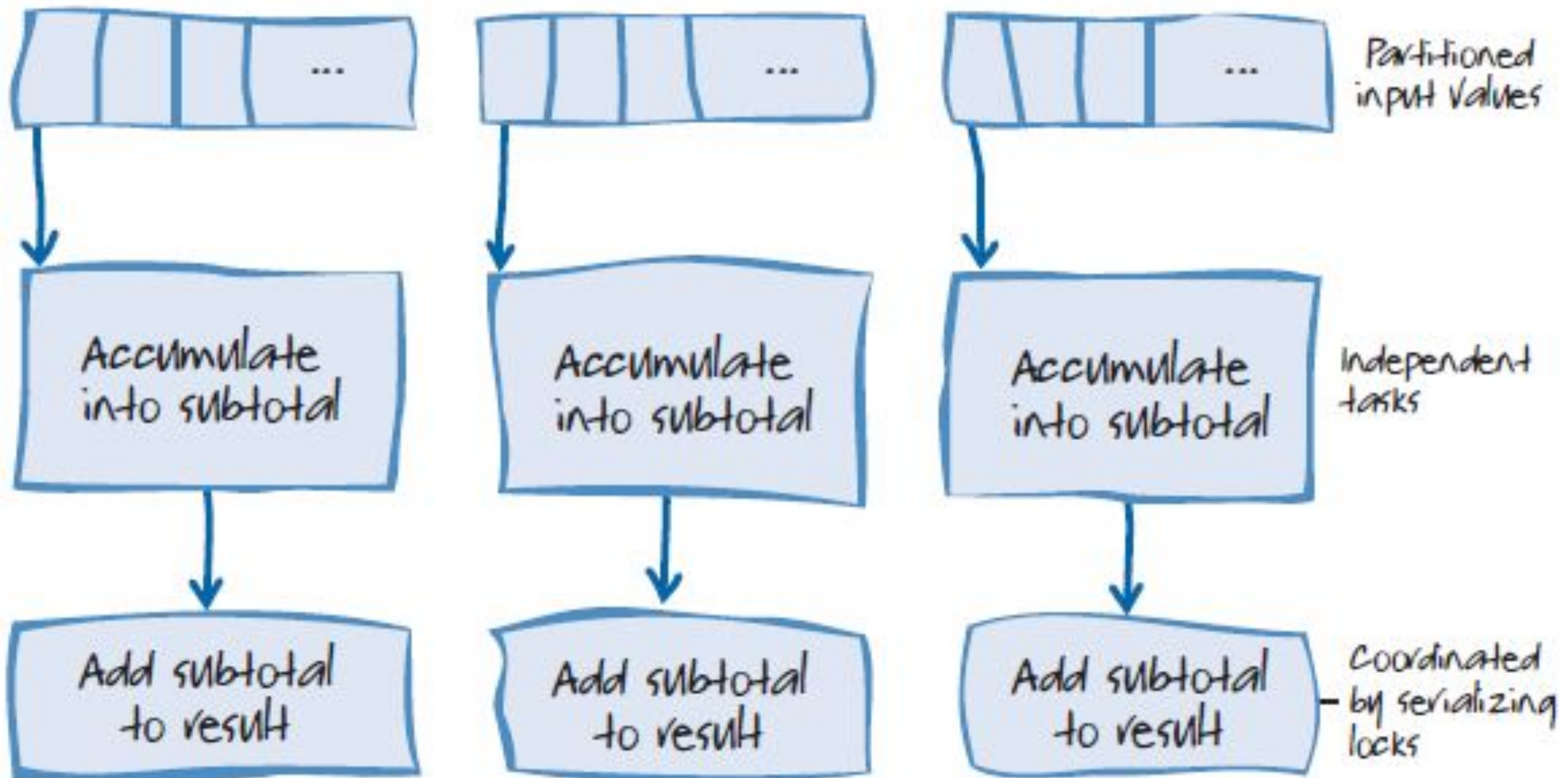
Using PLINQ aggregation with range selection

- The PLINQ Aggregate extension method includes an overloaded version that allows a very general application of the parallel aggregation pattern

```
Aggregate<TSource, TAccumulate, TResult>(
  this ParallelQuery<TSource> source,
  Func<TAccumulate> seedFactory,
  Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,
  Func<TAccumulate, TAccumulate, TAccumulate>
                                       combineAccumulatorsFunc,
  Func<TAccumulate, TResult> resultSelector);
```

Design notes

- Aggregation using Parallel For and ForEach



Design notes

- Aggregation in PLINQ does not require the developer to use locks
    - The final merge step is expressed as a binary operator that combines any two partial result values (that is, two of the subtotals) into another partial result
    - Repeating this process on subsequent pairs of partial results eventually converges on a final result
- One of the advantages of the PLINQ approach is that it requires less synchronization, so it's more scalable

softserve experience matters

Task Parallel Library
Data Parallelism Patterns

USA HQ
Toll Free: 866-687-3588
Tel: +1-512-516-8880

Ukraine HQ
Tel: +380-32-240-9090

Bulgaria
Tel: +359-2-902-3760