Converting Parallel Code From Low-Level Abstractions to Higher-Level Abstractions

Semih Okur

Cansu Erdogan

Danny Dig

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

OSU

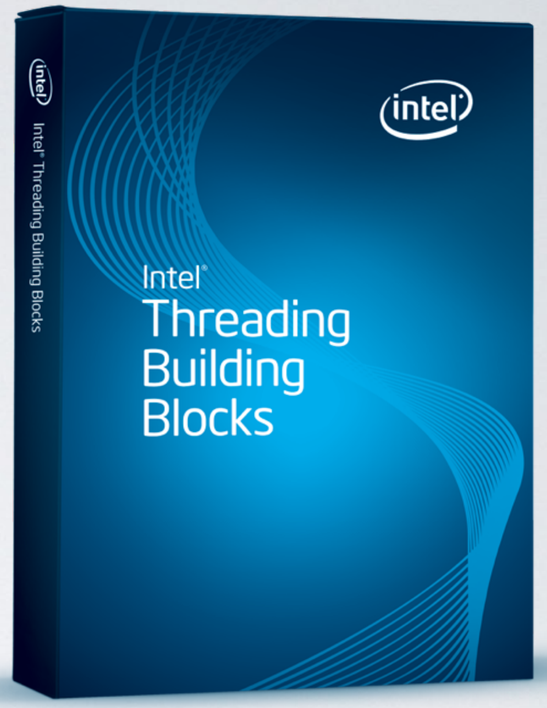
Oregon State UNIVERSITY

Libraries Make Parallel Programming Easier

Libraries Make Parallel Programming Easier

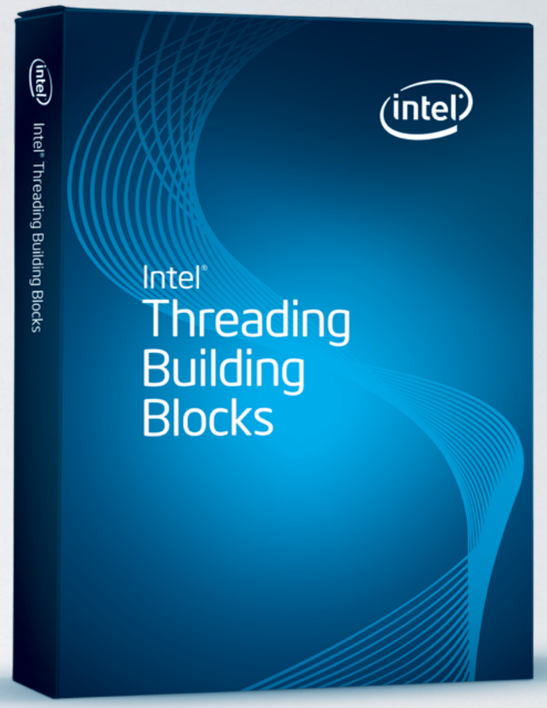
OpenMP™

Libraries Make Parallel Programming Easier



OpenMP™

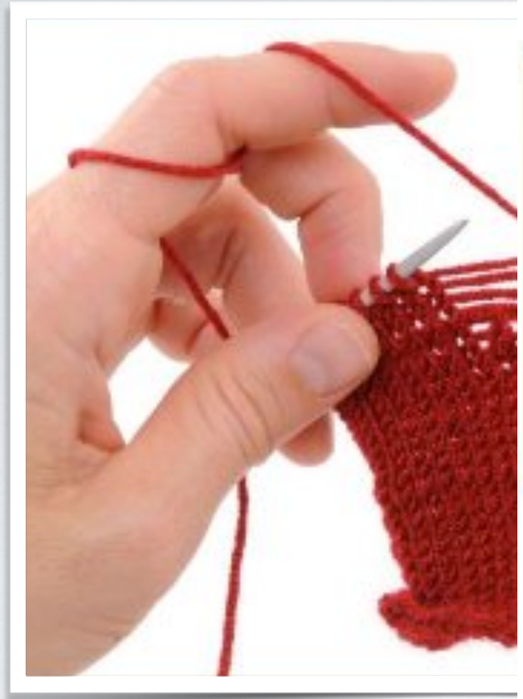
Libraries Make Parallel Programming Easier

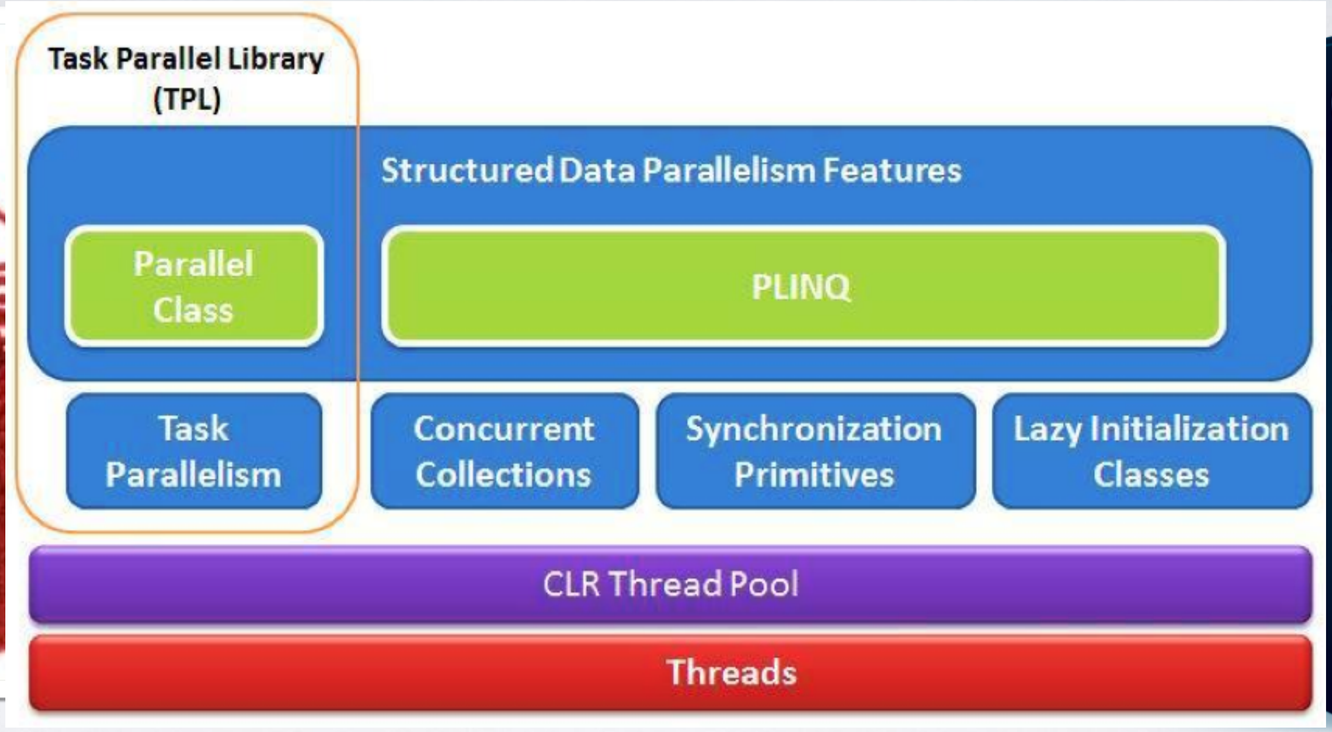


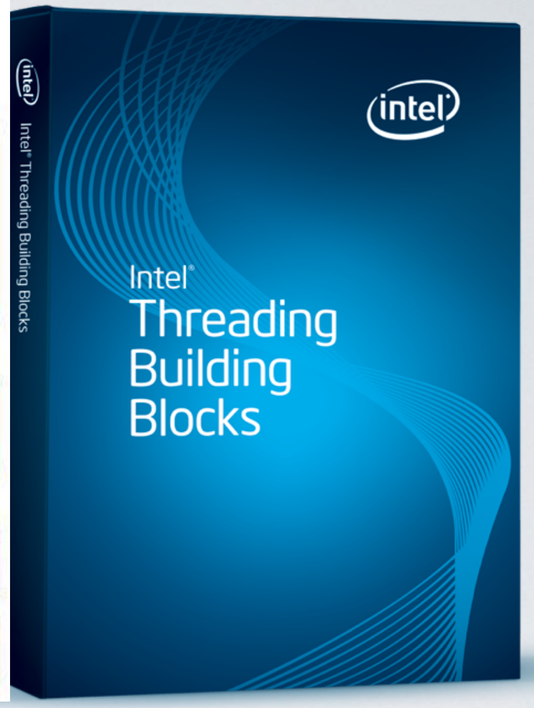
OpenMP™

java.util.concurrent

Libraries Make Parallel Programming Easier







OpenMP™

java.util.concurrent

Developers Still Use Old Libraries

Two recent empirical studies for C# and Java [1,2]

- Thread is still the primary choice for most developers.

1. Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are Java programmers transitioning to multicore?: a large scale study of java FLOSS. In *Proceedings of the SPLASH '11 Workshops*,
2. Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the FSE '12*.

Developers Still Use Old Libraries

Two recent empirical studies for C# and Java [1,2]

- Thread is still the primary choice for most developers.

Disadvantages of using old libraries:

1. Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are Java programmers transitioning to multicore?: a large scale study of java FLOSS. In *Proceedings of the SPLASH '11 Workshops*,
2. Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the FSE '12*.

Developers Still Use Old Libraries

Two recent empirical studies for C# and Java [1,2]

- Thread is still the primary choice for most developers.

Disadvantages of using old libraries:

- Threads are heavyweight, less scalable, more code bloat
- New platforms no longer support Thread

1. Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are Java programmers transitioning to multicore?: a large scale study of java FLOSS. In *Proceedings of the SPLASH '11 Workshops*,

2. Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the FSE '12*.

Developers Need Tool Support For Migration

Millions SLOC use old parallel libraries and need migration

Developers Need Tool Support For Migration

Millions SLOC use old parallel libraries and need migration

Challenges in migration

Developers Need Tool Support For Migration

Millions SLOC use old parallel libraries and need migration

Challenges in migration

- Understanding the shape of the computation
- Preserving program behavior even in the presence of exceptions

Developers Need Tool Support For Migration

Millions SLOC use old parallel libraries and need migration

Challenges in migration

- Understanding the shape of the computation
- Preserving program behavior even in the presence of exceptions

We provide two refactorings for C#

Example - Thread (Since 2003)

```
Thread thread = new Thread(foo);
thread.Priority = ThreadPriority.BelowNormal;
thread.Start(arg);
thread.Join();
```

+ Represents an actual OS-level thread, highest degree of control (stack size, priority, etc.)

Example - Thread (Since 2003)

```
Thread thread = new Thread(foo);
thread.Priority = ThreadPriority.BelowNormal;
thread.Start(arg);
thread.Join();
```

+ Represents an actual OS-level thread, highest degree of control (stack size, priority, etc.)

- Heavyweight (200,000 CPU cycles to create a new thread, and about 100,000 cycles to retire a thread)

- Creating a Thread needs about 1.5 MB memory space

Example - Thread (Since 2003)

```
Thread thread = new Thread(foo);
thread.Priority = ThreadPriority.BelowNormal;
thread.Start(arg);
thread.Join();
```

+ Represents an actual OS-level thread, highest degree of control (stack size, priority, etc.)

- Heavyweight (200,000 CPU cycles to create a new thread, and about 100,000 cycles to retire a thread)

- Creating a Thread needs about 1.5 MB memory space

- Developers need to manage the thread life cycle

Example - ThreadPool (Since 2003)

```
ThreadPool.QueueUserWorkItem(new WaitCallback(foo), arg);
```

+ Reusing the threads (amortize the cost of creating and destroying threads)

Example - ThreadPool (Since 2003)

```
ThreadPool.QueueUserWorkItem(new WaitCallback(foo), arg);
```

+ Reusing the threads (amortize the cost of creating and destroying threads)

- Developers have no control at all

Example - ThreadPool (Since 2003)

```
ThreadPool.QueueUserWorkItem(new WaitCallback(foo), arg);
```

+ Reusing the threads (amortize the cost of creating and destroying threads)

- Developers have no control at all

- No simple ‘joining’ or ‘waiting’ operation

Example - Task (Since 2010)

```
Task task = Task.Run(()=>foo(arg));
task.Wait();
```

+ Offers the best of both worlds: lightweight, efficient, scalable

Example - Task (Since 2010)

```
Task task = Task.Run(()=>foo(arg));
task.Wait();
```

+ Offers the best of both worlds: lightweight, efficient, scalable

+ Developers have advanced control (cancellation, continuation, futures, async/await, etc.)

Example - Task (Since 2010)

```
Task task = Task.Run(()=>foo(arg));
task.Wait();
```

+ Offers the best of both worlds: lightweight, efficient, scalable

+ Developers have advanced control (cancellation, continuation, futures, async/await, etc.)

Preferred way to write multithreaded code

Overview of Our Study

Conducted a formative study about parallel programming to answer two RQs:

Overview of Our Study

Conducted a formative study about parallel programming to answer two RQs:

RQ1) What level of parallel abstractions do developers use?

Overview of Our Study

Conducted a formative study about parallel programming to answer two RQs:

RQ1) What level of parallel abstractions do developers use?

RQ2) What do developers think about parallel abstractions?

Overview of Our Study

Conducted a formative study about parallel programming to answer two RQs:

RQ1) What level of parallel abstractions do developers use?

RQ2) What do developers think about parallel abstractions?

We provide two refactorings for C#: Taskifier & Simplifier

Code Corpus

Downloaded the most popular 1000 C# apps which was modified at least once since June 2013

Eliminated 120 apps which cannot be compiled and target old platforms (.Net 3.5, WP 7, Silverlight 4)

Code Corpus

Downloaded the most popular 1000 C# apps which was modified at least once since June 2013

Eliminated 120 apps which cannot be compiled and target old platforms (.Net 3.5, WP 7, Silverlight 4)

Analyzed 880 open source C# apps, comprising 42M SLOC, produced by 1859 developers

Wide domain: libraries, Windows Phone, silverlight, web, tablet applications

Code Corpus

Downloaded the most popular 1000 C# apps
which was modified at least once since June 2013

Eliminated 120 apps which cannot be compiled and
target old platforms (.Net 3.5, WP 7, Silverlight 4)

Analyzed 880 open source C# apps, comprising 42M SLOC, produced by 1859 developers

Wide domain: libraries, Windows Phone, silverlight, web, tablet applications

Built a static analysis tool with Microsoft's Roslyn
a set of open-source compilers and code analysis APIs for C# and VB
roslyn.codeplex.com

RQ1) What Level Of Parallel Abstractions Do Developers Use?

	#	App%
Thread	2105	31%
ThreadPool	1244	22%
Task	1542	19%

RQ1) What Level Of Parallel Abstractions Do Developers Use?

	#	App%
Thread	2105	31%
ThreadPool	1244	22%
Task	1542	19%

96 apps use Thread, ThreadPool, and task at the same time

RQ2) What Do Developers Think About Parallel Abstractions?

Contacted the developers of 10 apps that use Thread, ThreadPool, and Task

Why are you using the old abstractions?

RQ2) What Do Developers Think About Parallel Abstractions?

Contacted the developers of 10 apps that use Thread, ThreadPool, and Task

Why are you using the old abstractions?

"We intend to move most stuff to tasks, but that is on an as needed basis, since the code works" Oren, developer of ravendb

"never had time to change"

RQ2) What Do Developers Think About Parallel Abstractions?

Contacted the developers of 10 apps that use Thread, ThreadPool, and Task

Why are you using the old abstractions?

"We intend to move most stuff to tasks, but that is on an as needed basis, since the code works" Oren, developer of ravendb

"never had time to change"

Contacted the top 10 users for the tags "multithreading" and "C#"

We asked them some interesting questions about parallel abstractions

in the paper

TASKIFIER

A tool that migrates Thread and ThreadPool abstractions to Task

Visual Studio plugin, on top of the Roslyn API

TASKIFIER

A tool that migrates Thread and ThreadPool abstractions to Task

Visual Studio plugin, on top of the Roslyn API

Workflow

TASKIFIER

A tool that migrates Thread and ThreadPool abstractions to Task

Visual Studio plugin, on top of the Roslyn API

Workflow

(1) Taskifier identifies Thread or ThreadPool instances
(2) Converts the member accesses and method calls to the corresponding ones
(3) Optimizes the new code for special cases

(1) From Thread To Task

1- Identify the Thread instances

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

```
Task task = Task.Run(()=> doWork(e));
if(task.Status == TaskStatus.Running)
   ...
task.Wait();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

- Many syntactic variations

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

```
Task task = Task.Run(()=> doWork(e));
if(task.Status == TaskStatus.Running)
   ...
task.Wait();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

- Many syntactic variations
- Not always one-to-one mapping

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

```
Task task = Task.Run(()=> doWork(e));
if(task.Status == TaskStatus.Running)
   ...
task.Wait();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

- Many syntactic variations
- Not always one-to-one mapping

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

```
Task task = Task.Run(()=> doWork(e));
if(task.Status == TaskStatus.Running)
   ...
task.Wait();
```

(1) From Thread To Task

1- Identify the Thread instances

2- Replace its method calls and member accesses with their correspondents from the Task class.

- Many syntactic variations
- Not always one-to-one mapping
- Not using the same name and type

```
ThreadStart t =
     new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
if(thread.IsAlive)
   ...
thread.Join();
```

```
Task task = Task.Run(()=> doWork(e));
if(task.Status == TaskStatus.Running)
   ...
task.Wait();
```

(3) Challenges

1- Understanding the shape of the computation (I/O and CPU-bound)

Naively transforming I/O-bound Thread to a Task, can cause starvation in the thread pool which severely degrades performance

(3) Challenges

1- Understanding the shape of the computation (I/O and CPU-bound)

Naively transforming I/O-bound Thread to a Task, can cause starvation in the thread pool which severely degrades performance

Manually determining whether the code in a Thread transitively calls some blocking operations is non-trivial

(3) Challenges

1- Understanding the shape of the computation (I/O and CPU-bound)

Naively transforming I/O-bound Thread to a Task, can cause starvation in the thread pool which severely degrades performance

Manually determining whether the code in a Thread transitively calls some blocking operations is non-trivial

32% of existing tasks have at least one I/O blocking operation and 9% use Thread.Sleep(>1 second).

(3) Challenges

1- I/O and CPU-bound

(i) Checks the members of the Stream if there is any …Async method (naming convention)

```
thread.Start(foo);
...
public static void foo()
{
  ...
  var str = stream.Read();
  ...
}
```

```
Task.Run(()=> foo());
...
public async static void foo()
{
  ...
  var str = await stream.ReadAsync();
  ...
}
```

(3) Challenges

1- I/O and CPU-bound

(i) Checks the members of the Stream if there is any …Async method (naming convention)

```
thread.Start(foo);
...
public static void foo()
{
  ...
  var str = stream.Read();
  ...
}
```

```
Task.Run(()=> foo());
...
public async static void foo()
{
  ...
  var str = await stream.ReadAsync();
  ...
}
```

(3) Challenges

1- I/O and CPU-bound

(i) Checks the members of the Stream if there is any …Async method (naming convention)

(ii) Replace it with the new call

(iii) Add async/await keywords

```
thread.Start(foo);
...
public static void foo()
{
  ...
  var str = stream.Read();
  ...
}
```

```
Task.Run(()=> foo());
...
public async static void foo()
{
  ...
  var str = await stream.ReadAsync();
  ...
}
```

(3) Challenges

1- I/O and CPU-bound

(i) Checks the members of the Stream if there is any …Async method (naming convention)

(ii) Replace it with the new call

(iii) Add async/await keywords

```
thread.Start(foo);
...
public static void foo()
{
  ...
  var str = stream.Read();
  ...
}
```

```
Task.Run(()=> foo());
...
public async static void foo()
{
  ...
  var str = await stream.ReadAsync();
  ...
}
```

(3) Challenges

1- I/O and CPU-bound

(i) Checks the members of the Stream if there is any …Async method (naming convention)

(ii) Replace it with the new call

(iii) Add async/await keywords

```
thread.Start(foo);
...
public static void foo()
{
  ...
  var str = stream.Read();
  ...
}
```

```
Task.Run(()=> foo());
...
public async static void foo()
{
  ...
  var str = await stream.ReadAsync();
  ...
}
```

```
Thread.Sleep => await Task.Delay
```

(3) Special Cases

1- Understanding the shape of the computation (I/O and CPU-bound)

2- Behavior difference in exception handling

An unhandled exception in Thread and ThreadPool abstractions results in termination of the applications.

An unhandled exception in Task is propagated back to the joining thread when it is waited.

Naive migration from Thread to Task can make the unhanded exceptions silenced

(3) Special Cases

1- Understanding the shape of the computation (I/O and CPU-bound)

2- Behavior difference in exception handling

```
Task.Run(()=> foo()).FailFastOnException();

public static Task FailFastOnException(this Task task) {
    task.ContinueWith(c => Environment.FailFast("Task faulted",
c.Exception), TaskContinuationOptions.OnlyOnFaulted |
TaskContinuationOptions.ExecuteSynchronously |
TaskContinuationOptions.DetachedFromParent);
    return task;
}
```

(3) Special Cases

1- Understanding the shape of the computation (I/O and CPU-bound)

2- Behavior difference in exception handling

```
Task.Run(()=> foo()).FailFastOnException();

public static Task FailFastOnException(this Task task) {
    task.ContinueWith(c => Environment.FailFast("Task faulted",
c.Exception), TaskContinuationOptions.OnlyOnFaulted |
TaskContinuationOptions.ExecuteSynchronously |
TaskContinuationOptions.DetachedFromParent);
    return task;
}
```

3- Foreground and Background is

in the paper

Developers Still Miss Opportunities

Tasks affords more flexibility and control but developers do not use the extra flexibility in most cases

Parallel Class (2010)

Support parallel programming design patterns

- Parallel.For/ForEach: data parallelism
- Parallel.Invoke: fork-join task parallelism

```
Parallel.For(0, n, (i)=> foo(i));

Parallel.Invoke( ()=> foo(arg1),
                 ()=> foo(arg2));
```

Developers Are Not Aware Of Parallel

From Formative Study

	#	App%
Data Parallelism with Parallel.For	432	6%
Task Paralleism with Parallel.Invoke	53	1%

Developers Are Not Aware Of Parallel

From Formative Study

	#	App%
Data Parallelism with Parallel.For	432	6%
Task Paralleism with Parallel.Invoke	53	1%

Asked some questions the developers of 10 applications which use Thread, ThreadPool and Task at the same time.

Developers Are Not Aware Of Parallel

From Formative Study

	#	App%
Data Parallelism with Parallel.For	432	6%
Task Paralleism with Parallel.Invoke	53	1%

Asked some questions the developers of 10 applications which use Thread, ThreadPool and Task at the same time.

Are you aware of Parallel class?

Developers Are Not Aware Of Parallel

From Formative Study

	#	App%
Data Parallelism with Parallel.For	432	6%
Task Paralleism with Parallel.Invoke	53	1%

Asked some questions the developers of 10 applications which use Thread, ThreadPool and Task at the same time.

Are you aware of Parallel class?

"Is this in .NET framework? It is the most elegant way of a parallel loop"

Simplifier

Refactoring tool that converts multiple Task instances to one of three Parallel operations.

Fork-join Tasks => Parallel.Invoke

Data parallelism => Parallel.For / ForEach

Simplifier suggests code snippets that can be transformed to Parallel operations

Simplifier

Refactoring tool that converts multiple Task instances to one of three Parallel operations.

in the paper

Fork-join Tasks => Parallel.Invoke

Data parallelism => Parallel.For / ForEach

Simplifier suggests code snippets that can be transformed to Parallel operations

Simplifier

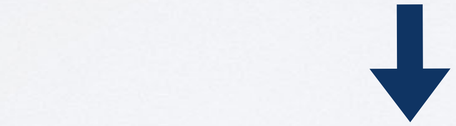
from ravendb app

```
Task[] tasks = new Task[n];
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                   ()=> Queues[temp].Stop());
    tasks[i].Start();
}
Task.WaitAll(tasks);
```

Simplifier

from ravendb app

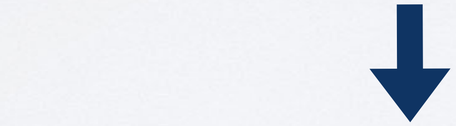
```
Task[] tasks = new Task[n];
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
Task.WaitAll(tasks);
```



Simplifier

from ravendb app

```
Task[] tasks = new Task[n];
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
Task.WaitAll(tasks);
```



```
Parallel.For(o, n, (i)=> Queues[i].Stop());
```

Algorithm (Parallel.For)

1) Locates a global barrier for the tasks (e.g., Task.WaitAll(tasks))

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

1) Locates a global barrier for the tasks (e.g., Task.WaitAll(tasks))

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

2) Argument of the barrier is only modified inside one ‘for’ loop

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

2) Argument of the barrier is only modified inside one ‘for’ loop

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

2) Argument of the barrier is only modified inside one ‘for’ loop

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

2) Argument of the barrier is only modified inside one ‘for’ loop

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

3) The iterator should be ‘++’ or ‘+=1’

Algorithm (Parallel.For)

4) Analyze the statements inside the loop. Allowed statements:

- Task creation, task starting
- Adding task to the array
- Creating a temporary variable

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

4) Analyze the statements inside the loop. Allowed statements:

- Task creation, task starting
- Adding task to the array
- Creating a temporary variable

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

4) Analyze the statements inside the loop. Allowed statements:

- Task creation, task starting
- Adding task to the array
- Creating a temporary variable

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                  ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

4) Analyze the statements inside the loop. Allowed statements:

- Task creation, task starting
- Adding task to the array
- Creating a temporary variable

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                  ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Algorithm (Parallel.For)

4) Analyze the statements inside the loop. Allowed statements:

- Task creation, task starting
- Adding task to the array
- Creating a temporary variable (Loop-carried dependence analysis)

```
Task[] tasks = new Task[n];
…
for(int i=0; i<n; i++)
{
    int temp = i;
    tasks[i] = new Task(
                    ()=> Queues[temp].Stop());
    tasks[i].Start();
}
…
Task.WaitAll(tasks);
```

Empirical Evaluation

Used the same code corpus from the Formative Empirical Study: 880 C# apps, 42M SLOC

Research Questions

(1) Are the refactorings applicable?

(2) Do the refactorings reduce the code bloat?

(3) Are the refactorings useful?

in the paper

(4) How much programmer effort is saved by the tools?

(5) Are the automated transformations safe?

(1) Our Tools Are Applicable

Taskifier

Taskifier migrated 78% of 1782 Thread instances.

Taskifier migrated all of 1244 ThreadPool instances to Task

Simplifier

Refactored all of 85 Task-based fork-join patterns to Parallel.Invoke

Refactored 92% of Task-based data-parallelism patterns to Parallel.For/ForEach

(2) Our Tools Reduce The Code Bloat

Taskifier

It is mostly one-to-one transformation

Achieved on average a 16% reduction in SLOC of the instance

Simplifier

Transforms multiple Task operations and helper operations to one equivalent method in the Parallel class

Achieved on average a 44% reduction for Parallel.Invoke

Achieved on average a 62% reduction for Parallel.For(Each)

(3) Our Transformations Are Useful

Chose 10 most recently updated apps for each Taskifier and Simplifier
Applied the tools ourselves and offered patches.
15 of 20 apps responded and accepted 53 patches.

Migration to Task is on their TODO list but they always postponed it because of working on new features.

Developers are looking forward to using our toolkit.

Conclusions

Describe the novel problem of migrating low-level parallel abstractions into their high-level counterparts

A toolkit inspired by our empirical study:

Taskifier migrates Thread(Pool) to Task in a fast & safe way

Simplifier finds parallel patterns and replace them with Parallel class

Evaluation:

highly applicable, useful,
saves programmer effort,
safer than manual migration
reduce the code bloat (62% on average for Parallel.For)