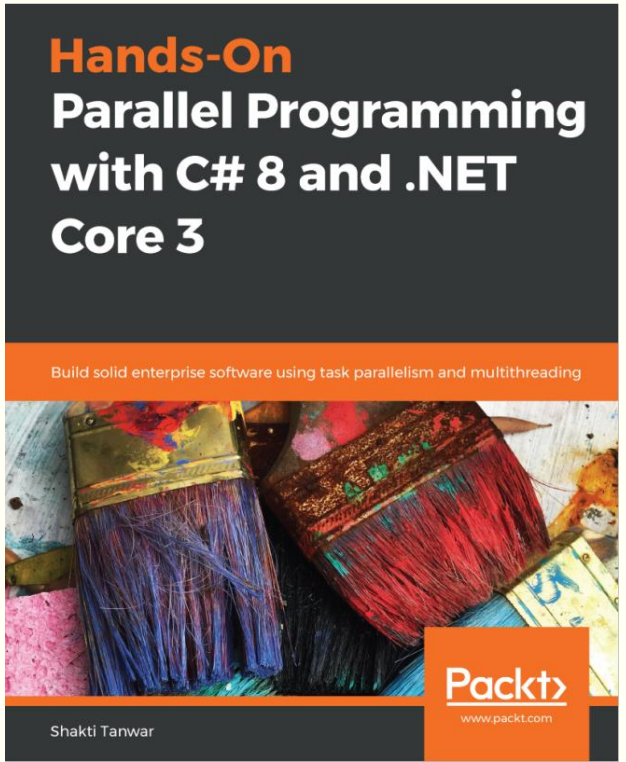
ПАРАЛЕЛЬНЕ ВИКОНАННЯ КОДУ

Лекція 12
Об’єктно-орієнтоване програмування

План лекції

- Основи паралельного виконання коду.
- Паралельна обробка даних за допомогою Parallel LINQ.
- Бібліотека розпаралелення задач TPL Dataflow.
  - https://dimuthublog.wordpress.com/2017/04/24/implementing-concurrent-work-flow-in-c-sharp/



ОСНОВИ ПАРАЛЕЛЬНОГО ВИКОНАННЯ КОДУ

Питання 12.1. (глава 4 з книги)

Паралельна обробка даних

- Паралельне програмування використовується для розбиття блоків роботи, обмежених по обчисленнях, та їх розподілу між кількома потоками.
- ***Завдання:*** маємо колекцію даних. Потрібно виконати одну і ту ж операцію з кожним елементом даних.
  - Ця операція обмежена по обчисленнях та може тривати деякий час.
- ***Вирішення:*** Тип Parallel містить метод ForEach, розроблений спеціально для цього.
  - Приклад отримує колекцію матриць та обертає ці матриці:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

Паралельна обробка даних

- Можливі ситуації, в яких передчасно потрібно перервати цикл (наприклад, при виявленні недійсного значення).
  - Приклад обертає кожну матрицю, проте при виявленні недійсної матриці цикл буде перервано:

```
void InvertMatrices(IEnumerable matrices) {
    Parallel.ForEach(matrices, (matrix, state) =>
    {
        if (!matrix.IsInvertible)
            state.Stop();
        else matrix.Invert();
    });
}
```

  - Код використовує ParallelLoopState.Stop для зупинки циклу та запобігання будь-яких подальших викликів тіла циклу.
  - Враховуючи паралельність циклу, можуть виконуватись інші виклики тіла циклу, зокрема виклики для наступних після поточного елементів.
  - Якщо 3тю матрицю неможливо обернути, то цикл переривається, і нові матриці оброблятись не будуть, проте може виявитись, що вже здійснюється обернення інших матриць (наприклад, 4ї та 5ї).

Більш розповсюджена ситуація: потрібно скасувати паралельний цикл

- Цикл зупиняється зсередини та скасовується за своїми межами.
  - Наприклад, кнопка скасування може скасувати CancellationTokenSource (і паралельний цикл):

```
void RotateMatrices(IEnumerable matrices, float degrees, CancellationToken token) {
    Parallel.ForEach(matrices,
                        new ParallelOptions { CancellationToken = token },
                        matrix => matrix.Rotate(degrees)
    );
}
```

  - Метод Parallel.ForEach надає можливість паралельної обробки для послідовності значень.
  - Аналогічне рішення Parallel LINQ (PLINQ) надає практично ті ж можливості в LINQ-подібному синтаксисі.
  - Одна з відмінностей: PLINQ передбачає, що може використовувати *всі* ядра на комп'ютері, тоді як Parallel може динамічно реагувати на зміни умов процесора.

Паралельна обробка даних

```
// Примечание: это не самая эффективная реализация.
// Это всего лишь пример использования блокировки
// для защиты совместного состояния.
int InvertMatrices(IEnumerable<Matrix> matrices)
{
  object mutex = new object();
  int nonInvertibleCount = 0;
  Parallel.ForEach(matrices, matrix =>
  {
    if (matrix.IsInvertible)
    {
      matrix.Invert();
    }
    else
    {
      lock (mutex)
      {
        ++nonInvertibleCount;
      }
    }
  });
  return nonInvertibleCount;
}
```

- Зауважте, що кожна паралельна задача може виконуватись у іншому потоці, тому будь-який спільний стан повинен бути захищеним.
  - У прикладі кожну матрицю обертають та підраховують кількість матриць, які обернути не вдалось.
- Parallel.ForEach реалізує паралельний цикл foreach.
  - Для виконання паралельного циклу for клас Parallel також підтримує метод Parallel.For.
  - Метод Parallel.For особливо корисний при роботі з кількома масивами даних, які отримують один індекс.

Рецепт 2: паралельне агрегування

- ***Завдання:*** потрібно агрегувати результати при завершенні паралельної операції.
- ***Вирішення:*** клас Parallel використовує концепцію локальних значень – змінних, які існують локально всередині паралельного циклу.
  - Тіло циклу може просто звертатись до значення напряму, без необхідності синхронізації.
  - Коли цикл готовий до агрегування всіх своїх локальних результатів, він робить це за допомогою делегата localFinally.
  - Делегату localFinally не потрібно синхронізувати доступ до змінної для зберігання результату.

```
// Це не найефективніша реалізація, а лише приклад
// використання блокування для захисту спільного стану.
int ParallelSum(IEnumerable<int> values)
{
    object mutex = new object();
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue => {
            lock (mutex)
                result += localValue;
        });
    return result;
}
```

- В Parallel LINQ реалізована зрозуміліша підтримка агрегування:
  - int ParallelSum(IEnumerable<int> values) {
    return values.AsParallel().Sum();
    }
  - Дешевий трюк: в PLINQ є вбудована підтримка розповсюджених операторов (наприклад, Sum). Також передбачена узагальнена підтримка агрегування оператором Aggregate:
  - int ParallelSum(IEnumerable<int> values) {
    return values.AsParallel().Aggregate(
    seed: 0,
    func: (sum, item) => sum + item);
    }

Рецепт 3: Паралельний виклик

```
void ProcessArray(double[] array)
{
  Parallel.Invoke(
      () => ProcessPartialArray(array, 0, array.Length / 2),
      () => ProcessPartialArray(array, array.Length / 2, array.Length)
  );
}

void ProcessPartialArray(double[] array, int begin, int end)
{
  // Обработка, интенсивно использующая процессор...
}
```

```
void DoAction20Times(Action action)
{
  Action[] actions = Enumerable.Repeat(action, 20).ToArray();
  Parallel.Invoke(actions);
}
```

```
void DoAction20Times(Action action, CancellationToken token)
{
  Action[] actions = Enumerable.Repeat(action, 20).ToArray();
  Parallel.Invoke(new ParallelOptions { CancellationToken = token },
     actions);
}
```

- ***Завдання:*** маємо набір методів, які повинні викликатись паралельно.
  - Ці методи переважно незалежні один від одного.
- ***Вирішення:*** клас Parallel має простий метод Invoke для таких сценаріїв.
  - У прикладі масив розбивається навпіл, і половини обробляються незалежно.
  - Метод Parallel.Invoke уже не такий доречний для випадків, коли потрібно активізувати дію для кожного елементу вхідних даних (Parallel.ForEach краще) або якщо кожна дія виконує деякий вивід (Parallel LINQ краще).
- Методу Parallel.Invoke також можна передати масив делегатів, якщо кількість викликів невідома до моменту виконання.
  - Parallel.Invoke підтримує скасування, як і інші методи класу Parallel.

Рецепт 4: динамічний паралелізм

```
void Traverse(Node current)
{
  DoExpensiveActionOnNode(current);
  if (current.Left != null)
  {
    Task.Factory.StartNew(
        () => Traverse(current.Left),
        CancellationToken.None,
        TaskCreationOptions.AttachedToParent,
        TaskScheduler.Default);
  }
  if (current.Right != null)
  {
    Task.Factory.StartNew(
        () => Traverse(current.Right),
        CancellationToken.None,
        TaskCreationOptions.AttachedToParent,
        TaskScheduler.Default);
  }
}

void ProcessTree(Node root)
{
  Task task = Task.Factory.StartNew(
      () => Traverse(root),
      CancellationToken.None,
      TaskCreationOptions.None,
      TaskScheduler.Default);
  task.Wait();
}
```

- ***Завдання:*** потрібно реалізувати складнішу паралельну ситуацію: структура і кількість паралельних задач залежить від інформації, яка стає відомою тільки в ході виконання.
- ***Вирішення:*** у бібліотеці TPL центральне місце займає тип Task.
  - Клас Parallel і Parallel LINQ – лише зручні обгортки для нього.
  - Если потребуется реализовать динамический параллелизм, проще использовать тип Task напрямую.
- У прикладі для кожного вузла бінарного дерева необхідно виконати певну затратну обробку.
  - Структура дерева невідома до стадії виконання, тому цей сценарій доречний для динамічного паралелізму.
  - Метод Traverse обробляє поточний вузол, а потім створює 2 дочірні задачі, по одній для кожної гілки (тут передбачається, що батьківські вузли повинні бути оброблені до переходу до дочірніх вузлів).
  - Метод ProcessTree починає обробку, створюючи батьківську задачу верхнього рівня та очікуючи на її завершення.
  - Батьківські задачі виконують свій делеагат, а потім очікують завершення своїх дочірніх задач.

Рецепт 4: динамічний паралелізм

```
Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Task is done"),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// Аргумент "t" для продолжения - то же, что "task".
```

- Якщо ваша ситуація не відноситься до категорії «батько/потомок», ви можете запланувати запуск будь-якої задачі після іншої задачі, використовуючи продовження.
  - Тут застосовуються CancellationToken.None і TaskScheduler.Default.
  - Завжди краще явно задати планувальник TaskScheduler, що використовується StartNew та ContinueWith.
  - Така структура батьківських та дочірніх задач типова для динамічного паралелізму, проте не обов’язкова.
  - Також можна зберегти кожну нову задачу в потокобезпечній колекції, а потім очікувати завершення їх усіх за допомогою Task.WaitAll().

Parallel LINQ

- ***Завдання:*** потрібно виконати паралельну обробку послідовності даних, щоб згенерувати іншу їх послідовність або узагальнення цих даних.
- ***Вирішення:*** Parallel LINQ (PLINQ) розширяє підтримку LINQ паралельною обробкою.
  - PLINQ добре працює в потокових сценаріях, які отримують послідовність вхідних значень і утворюють послідовність вихідних значень.
  - Клас Parallel доречний для багатьох сценаріїв, проте код PLINQ виявляється більш простим при агрегуванні або перетворення однієї послідовності в іншу.
  - Пам'ятайте, що клас Parallel веде себе коректніше з іншими процесами в системі, ніж PLINQ; цей фактор стає особливо суттєвим при виконанні паралельної обробки на серверній машині.
- Простий приклад: множення кожного елементу послідовності на 2:

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
  return values.AsParallel().Select(value => value * 2);
}
```

  - Приклад може генерувати свої вихідні значення в довільному порядку; ця поведінка використовується в Parallel LINQ за умовчанням.

Паралельна обробка даних

- Можна вимагати збереження початкового порядку:

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
  return values.AsParallel().AsOrdered().Select(value => value * 2);
}
```

- Інше застосування Parallel LINQ – агрегування чи узагальнення даних у паралельному режимі. Приклад з паралельним додаванням:

```
int ParallelSum(IEnumerable<int> values)
{
  return values.AsParallel().Sum();
}
```

  - PLINQ надає паралельні версії багатьох операторів, включаючи фільтрацію (Where), проєкцію (Select) та різні види агрегування, зокрема Sum, Average та більш загальну форму Aggregate.
  - Загалом, все, що можна зробити за допомогою LINQ, також можливо здійснити в паралельному режимі з PLINQ.

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: Паралельна обробка даних за допомогою PLINQ