УДК 004.432.2, 004.422.83

ПОБУДОВА БАГАТОПОТОКОВИХ ПРОГРАМ ЗАСОБАМИ ПЛАТФОРМИ .NET

Ср. Ярошко[1], **Св. Ярошко**[2]

[1] *Львівський національний університет імені Івана Франка, вул. Університетська, 1, Львів, 79000, e-mail: serhiy.yaroshko@lnu.edu.ua*
[2] *Національний університет "Львівська політехніка", вул. С. Бандери, 12, Львів, 79013, e-mail: svitlana.m.yaroshko@lpnu.ua*

Описано загальні підходи до створення багатопотокових програм мовою C#. З'ясовано, як за допомогою класу *Task* платформи .Net і нових ключових слів *async*, *await* відокремити потік обчислень від потоку, що обслуговує інтерфейс користувача Windows Forms аплікації, як розпаралелити обчислення та відобразити процес їхнього виконання. Таку методику використано в програмі, що розв'язує задачу комівояжера для великої кількості міст за допомогою генетичного алгоритму. Наведено результати експериментальної перевірки швидкодії алгоритму на багатоядерному процесорі залежно від кількості паралельних потоків виконання обчислень.

Ключові слова: потік виконання, багатопотоковий застосунок, клас Task, розпаралелювання, зупинка потоку виконання, відображення ходу обчислень, генетичний алгоритм, тур комівояжера.

1. Вступ

Більшість сучасних програм використовують декілька потоків виконання. Наприклад, у редакторі текстів окремий потік відповідає за перевірку правопису, інший – за автоматичне збереження, ще один – за опрацювання команд користувача. Для розробника програм потоки надають зручний спосіб структурування функціональних можливостей: архітектура застосунку буде надійнішою, якщо різні його можливості запрограмовано незалежно одна від одної. Уміння використовувати потоки виконання стане в пригоді навіть обчислювачу. Ітераційні методи зазвичай передбачають виконання значного обсягу обчислень для отримання наближеного розв'язку певної задачі. Цю обставину доводиться враховувати під час проектування відповідних комп'ютерних програм, щоб користувач мав змогу спостерігати за ходом обчислень, і не переживав, чи не зависла його програма. У консольній програмі проблему можна вирішити доволі просто: достатньо, наприклад, додати інструкції виведення на екран номера виконаної ітерації. Проте користувач ніяк не зможе впливати на хід обчислень, йому доведеться терпеливо очікувати їхнього завершення. У програмі з графічним інтерфейсом ситуація складніша: якщо обчислення виконує той самий потік, що й обслуговує інтерфейс, то вікно програми залишатиметься "замороженим" аж до завершення обчислень. Для того, щоб інформувати користувача про хід обчислень і залишити інтерфейс здатним до взаємодії, доведеться запускати обчислення в окремому потоці виконання.

Платформа .Net надає програмістові декілька засобів для запуску окремих потоків виконання [1]. Для безпосередньої взаємодії з потоком використовують клас *Thread* з простору імен *System.Threading*, для асинхронного виконання окремого методу – можливості класів *Delegate* та *MulticastDelegate*, для асинхронного читання

з файла – метод *ReadAsync* класу *FileStream* тощо. Починаючи з версії 2.0 платформи .Net для зручної взаємодії з асинхронними потоками почали використовувати асинхронний шаблон на підставі подій [2] і його реалізації у стандартних компонентах, наприклад, *BackgroundWorker*. Найбільш ефективним щодо використання обчислювальних потужностей комп'ютера є сучасний асинхронний шаблон на підставі задач [3], що використовує класи *Task* і *Task<TResult>* простору імен *System.Threading.Tasks*. Вони з'явилися у версії 4.0 платформи .Net, і сьогодні фахівці компанії Microsoft радять використовувати саме їх для побудови багатопотокових застосунків. Екземпляри цих класів створюються швидко і потребують небагато системних ресурсів, ефективно використовують час процесора, добре взаємодіють з системою планування потоків. З їхньою допомогою можна легко виділити обчислення в окремий потік виконання, який може запускати декілька паралельних потоків обчислень. Такий підхід суттєво пришвидшує виконання на багатопроцесорному (чи багатоядерному) комп'ютері.

Для вивчення сучасних технологій програмування доцільно використовувати реалістичні приклади їхнього застосування. Такі приклади мали б бути досить простими для розуміння і, водночас, досить складними, щоб продемонструвати можливості технології щодо вирішення практичних завдань. Зокрема, на практиці потрібно враховувати деякі неочевидні особливості використання класу *Task*. У нашій праці як навчальний приклад описано, як за допомогою класу *Task* побудувати застосунок, що виконує обчислення у декількох паралельних потоках та інформує користувача про хід обчислень. Для прикладу створено програму, яка наочно демонструє застосування генетичного алгоритму відшукання туру комівояжера для великої кількості міст. Експериментально перевірено, як впливає кількість потоків виконання на швидкодію програми. Повний текст програми можна завантажити з репозитарію [4].

Відомо, що еволюційне числення не гарантує отримання точного розв'язку задачі, й автори не претендують на те, що їхня програма знайде найкоротший тур. Вона лише надає графічну візуалізацію процесу обчислень. Посилання на програмну реалізацію генетичного алгоритму містить екземпляр узагальненого інтерфейсу, що дає змогу легко адаптувати програму для відображення виконання довільного обчислювального алгоритму.

2. Асинхронні фонові задачі

Багато застосунків з графічним інтерфейсом користувача потребують здатності виконувати асинхронні фонові завдання. Конкретні вимоги можуть відрізнятися, але більшості програм потрібна можливість запускати на виконання деяку операцію так, щоб не блокувати інтерфейс користувача й отримувати від неї звіт про завершення.

Загальні вимоги щодо такої можливості сформульовано у [5]. Перелічимо їх і опишемо, як вони реалізовані в моделі асинхронних методів, що використовують *async*, *await* , та в бібліотеці Task Parallel Library (TPL).

Отримання результату. Зазвичай метою виконання асинхронної операції є отримання певного значення (чи набору значень), яке далі використовують для оновлення стану програми та подальших обчислень. Асинхронні методи природно повертають екземпляри *Task* або *Task<TResult>*. З об'єкта *Task<TResult>* результат отримують через властивість *Result*. Треба зазначити, що читання цієї власти-

вості синхронізує потік виконання: інструкція читання затримає потік виконання аж до завершення асинхронної операції.

Сигналізування про помилки. Якщо під час виконання операції сталася помилка, то потік виконання інтерфейсу мав би отримати виняток з інформацією про проблему. Асинхронні методи коректно справляються з цим завданням, головний потік отримує коректний стек викликів. Якщо виняток стався всередині *Task*, то він також дістанеться до головного потоку, але як вкладений виняток у *AggregateException*.

Хід виконання. У багатьох випадках корисно, коли, крім повернення результату після успішного виконання та сигналізування про помилку, фонове завдання покроково інформує про хід свого виконання. Така можливість є в асинхронних методів: вони використовують абстракцію *IProgress<T> progress* для "спілкування" з потоком, що їх викликав. У цьому випадку асинхронний метод у потрібний момент викликає *progress.Report(data)*, а головний потік перед викликом асинхронного методу визначає, як опрацьовувати отримані *data*.

Скасування. Для тривалих завдань, які виконують обчислення, має бути певний механізм скасування. Обчислювальний алгоритм мав би перевіряти наявність вимоги припинити роботу і завершуватися належно. І асинхронні методи, і завдання *Task* використовують уніфікований механізм скасування платформи .Net. Перед запуском асинхронної операції потрібно створити і передати їй маркер зупинки *CancellationToken token*. Для "м'якого" припинення виконання асинхронний алгоритм у зручний для себе момент має перевіряти властивість *token.IsCancellationRequested* і припиняти виконання на вимогу. Вимогу зупинки може ініціювати головний потік методом *token.Cancel()*.

Вкладення. Одне фонове завдання може запускати дочірні завдання. Така здатність важлива для проектування шару бізнес логіки застосунку. Дуже часто асинхронний метод запускає декілька *Task*, де всі вони можуть спільно використовувати один маркер зупинки та синхронно припиняти роботу.

Синхронізація. Потік інтерфейсу користувача має відображати прогрес фонового завдання, оновлюватися після його завершення тощо. Контекст виконання асинхронного методу створюється і звільняється автоматично. Екземпляри *Task* і *Task<TResult>* мають декілька методів синхронізації, а також можуть використовувати досконалі можливості *TaskScheduler*.

3. Генетичний алгоритм для задачі комівояжера

Генетичні алгоритми розв'язування задач [6] розглядають як альтернативу до традиційних методів, які використовують складний математичний апарат для того, щоб за мінімальну кількість кроків отримати якнайліпший результат. На противагу їм генетичний алгоритм мало турбується про оптимальні характеристики кандидатів на розв'язок задачі, проте генерує їх чимало і швидко для того, щоб відбракувати гірші та продовжити генерувати нові, враховуючи позитивний досвід, здобутий ліпшими кандидатами. Протягом процедури виконання такого алгоритму кандидати на розв'язок удосконалюються, покращують свої характеристики – еволюціонують від початкового наближення до оптимального розв'язку. Тому розв'язування задач за допомогою генетичних алгоритмів часто називають *еволюційним численням*. Такий підхід не може гарантувати відшукання точного (чи оптимального) розв'язку, тому еволюційне числення розглядають як евристику, або спосіб отримання хорошого наближеного розв'язку за малий час. Нагадаємо, що один

загальний підхід до побудови евристичних алгоритмів полягає у переліченні всіх вимог до точного розв'язку і поділі їх на дві групи: ті, які необхідно задовольнити обов'язково, і ті, які можна виконати не повною мірою. Тоді будують алгоритм, який виконує вимоги з першої групи і намагається виконати вимоги другої групи.

Природна еволюція розпочинається з первинної популяції організмів. Протягом багатьох поколінь випадкова мінливість і природний відбір формують поведінку та здібності особин популяції так, щоб максимально пристосуватись до вимог оточення. У найзагальніших термінах еволюцію можна описати як двокроковий ітераційний процес, який складається з випадкової зміни та подальшого відбору. Зв'язок між таким описом еволюції та оптимізаційним алгоритмом концептуально простий. Як і природна еволюція, алгоритмічний підхід започатковують з вибору первинної множини альтернативних розв'язків певної проблеми. Далі ці "батьківські" розв'язки генерують "нащадків" за допомогою вибраних засобів випадкової варіації. Усі розв'язки – і батьків, і нащадків – оцінюють з огляду на те, наскільки добре вони виконують поставлене завдання. Остаточно застосовують критерій вибору, щоб відкинути ті розв'язки, цінність яких замала. Так само, як у природі "виживають найсильніші". Процес повторюють з відібраною множиною розв'язків, щоб отримати наступні покоління, аж доки не буде задоволено умову щодо отримання прийнятного розв'язку. Схематично цей алгоритм можна описати так

1. Ініціалізувати популяцію.
2. Утворити нащадків за допомогою випадкових змін і додати їх до популяції.
3. Обчислити пристосованість членів популяції.
4. Застосувати відбір.
5. Перевірити, чи виконано умову завершення: якщо так, то зупинити, інакше – перейти до кроку 2.

Еволюційні алгоритми не потребують таких припущень, як традиційні методи оптимізації. Наприклад, алгоритми лінійного програмування передбачають, що цільова функція є лінійною, градієнтні методи не можна використати, якщо цільова функція не гладка, має стрибки чи злами. Такі вимоги звужують область застосовності методів. Для еволюційного алгоритму важливо лише, щоб існував спосіб впорядкувати два конкурентні розв'язки, визначивши, який з них з певних причин ліпший від іншого. Така невибагливість методу сприяє його широкому застосуванню до тих задач, які не можна розв'язати традиційними числовими методами. Еволюційне числення можна застосовувати і тоді, коли окремі умови задачі змінюються. У цьому випадку поточна популяція розв'язків зберігається як резервуар накопиченого знання, яке можна використати для пристосування до зміненої задачі.

Щоб реалізувати еволюційний підхід, треба виконати такі підготовчі кроки: вибрати зображення розв'язку, визначити цільову функцію, винайти оператор випадкових змін, задати правило виживання розв'язків, ініціалізувати популяцію. Немає одного найкращого вибору зображення розв'язку чи оператора змін, тому потрібна певна винахідливість. Остаточний успіх еволюційного алгоритму залежить від того, наскільки добре узгоджені оператор мутацій, зображення розв'язку та цільова функція. Проілюструємо ці кроки на прикладі класичної задачі відшукання туру комівояжера для великої кількості міст. Йтиметься про обчислення розв'язку для дещо спрощеної моделі, а саме: задано натуральне n – кількість міст, координати цих міст на "карті" – декартовій площині; вважається, що кожну пару міст з'єднує дорога, довжину якої можна обчислити як відстань між точками координатної площини. Потрібно відшукати замкнуту ламану найменшої довжини, що

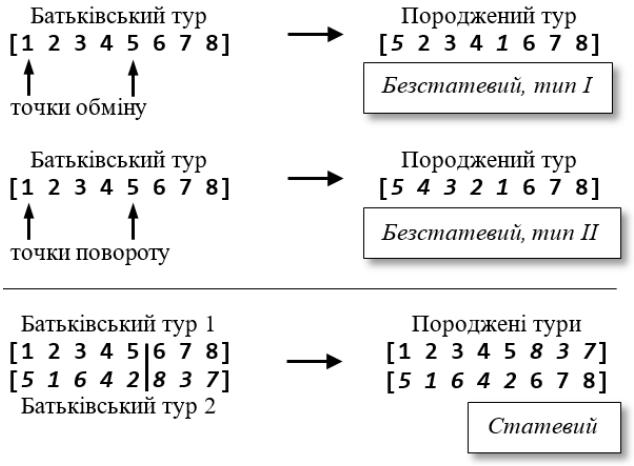

Рис. 1. Оператори випадкових змін

з'єднує всі міста на карті, де кожне місто є кінцевим вузлом рівно двох сусідніх ланок ламаної.

Кожному місту присвоїмо порядковий номер від 1 до n, тоді потенційним розв'язком задачі буде довільна перестановка перших n натуральних чисел. *Зображенням розв'язку* буде масив, що зберігає таку перестановку – порядок відвідання міст. Згенерувати новий розв'язок означатиме створити нову перестановку. Так завжди буде виконано обов'язкову вимогу до розв'язків задачі: тур проходить через кожне місто, і через кожне місто – один раз. Такий спосіб зображення турів можна використати і в тому випадку, коли між деякими містами немає сполучення. Достатньо буде вилучати з розгляду ті перестановки, в яких номери відповідних міст розташувалися поруч.

Подорож має бути найкоротшою, тому "ціною" розв'язку є довжина туру. *Цільова функція* обчислюватиме довжину ламаної за заданими координатами вузлів. Можна використати й інші способи побудови цільової функції. Наприклад, якщо б картою був граф, то для обчислення довжини можна було б використати елементи матриці суміжності. При відборі перевагу надаватимемо коротшому турові порівняно з довшим.

Оператор (оператори) *випадкових змін* за батьківськими розв'язками генерує породжені. Для його конструювання можна використати багато різних способів. У природі існує два основні способи відтворення: статевий і нестатевий. У статевому двоє батьків одного виду обмінюються генетичним матеріалом, який рекомбінується у потомстві. Безстатеве відтворення є за змістом клонуванням, однак під час передачі генетичного матеріалу від батька до нащадків цей матеріал може зазнавати різних змін. Такі оператори відтворення можна змоделювати в еволюційному алгоритмі. Розглянемо можливі варіанти генерування потомства для задачі комівояжера, який використовує перестановки (рис. 1). Тут "генами" розв'язку є номери міст. Безстатеві способи полягають у випадковому виборі двох міст з батьківського туру й обміні їх місцями у породженому турі (тип I) або зміні порядку слідування між ними на протилежний (тип II). Такі способи завжди генеруватимуть допустимі тури: у них буде кожне з міст, і кожне – один раз. Спеціалізовані

безстатеві оператори можуть вносити мутації не у весь ланцюжок міст, а в окрему його частину. Детальніший огляд публікацій, присвячених різним реалізаціям генетичного алгоритму розв'язування задачі комівояжера, можна знайти в [7]. Зауважимо, що статевий оператор у літературі називають також оператором схрещування (crossover operator), а безстатеві – операторами мутацій (exchange mutation operator - тип I, simple inversion mutation - тип II).

Статевий спосіб з'єднує частини двох батьківських турів, розітнутих у випадково вибраній точці. Він може генерувати недопустимі тури, які містять деякі міста двічі і не містять інших окремих міст. Це не означає, що в цій задачі не можна використовувати статевого розмноження – просто треба передбачити в операторі додаткові дії з відбракування чи виправлення генерованих турів. Виявлення дефектів геному суттєво впливає на швидкодію алгоритму, тому в цій статті такі оператори не розглядали.

Правило виживання у найпростішому випадку залишає в популяції певну кількість найпристосованіших розв'язків і вилучає всіх інших. Його легко реалізувати програмно, якщо використати для зберігання популяції впорядковану колекцію. Тоді виживання залишає в ній задану кількість початкових елементів і вилучає хвіст. Альтернативою може бути застосування змагального підходу, під час якого випадково створені пари (чи m-ки) розв'язків змагаються за виживання. Тоді слабші розв'язки в популяції часом виживають кілька поколінь, що може бути перевагою у складних проблемах, де буває легше поліпшити розв'язок, змінюючи гірший, ніж змінюючи тільки найліпші. Можливостей безліч, і будь-яке правило, яке загалом віддає перевагу у виживанні ліпшому розв'язку над гіршим, має сенс. Для *початкового створення популяції* розв'язки вибирають випадково з простору можливих, або залучають розв'язки, близькі до добрих, отриманих за допомогою деякого іншого алгоритму чи з урахуванням апріорної інформації щодо задачі.

4. Програмна реалізація

Клас *Tour* моделює розв'язок задачі, містить перестановку номерів міст, статичне посилання на координати міст, уміє обчислювати довжину туру і генерувати нові розв'язки за допомогою операторів випадкових змін. Перевірка на практиці засвідчила, що статевий спосіб генерування ускладнює реалізацію без суттєвого виграшу в ефективності, тому в програмі використано безстатеві оператори першого та другого типу. З'ясувалося також, що лише оператори другого типу позбавляють маршрут від петель і перехрещень, не притаманних оптимальному розв'язку.

Клас вікна застосунку *MainForm* зберігатиме вхідні дані задачі, параметри алгоритму (обсяг вибірки, кількості мутацій для обчислення нащадків, граничну кількість поколінь) і графічне середовище для відображення розв'язку: *Point[] towns* – екранні координати "міст", *Panel pnlMap* надає графічний контекст для відображення карти, *ISolver<Tour> solver* – посилання на реалізацію генетичного алгоритму, яке ми задали за допомогою інтерфейсу, щоб випробувати в програмі різні способи його реалізації.

Узагальнений інтерфейс *ISover<T>* описує мінімальні вимоги до реалізації.

```
interface ISolver<T>
{
    void Solve(CancellationToken token, IProgress<int> progress);
```

```
    T Best();
}
```

Тут метод *Solve* містить реалізацію числового методу, який підтримує можливість завершення роботи на вимогу користувача (параметр *CancellationToken token*) та інформування про хід виконання (параметр *IProgress<int> progress*). Метод *Best* повертає найліпший знайдений розв'язок.

У програмі використано дві різні реалізації цього інтерфейсу. Клас *SSolver* виконує всі обчислення в одному потоці, для зберігання популяції використовує колекцію *SortedDictionary<double, Tour> population*, яка автоматично впорядковує свої елементи-тури за довжиною. Метод *SSolver.Solve* заповнює початкову популяцію випадковими перестановками номерів міст, а далі послідовно генерує для кожного члена популяції певну кількість нащадків і додає їх до *population*. Перед кожним новим поколінням метод перевіряє стан властивості маркера завершення *token.IsCancellationRequested*, щоб завершити обчислення за першою вимогою.

Зупинимося детальніше на іншій реалізації – класі *PSolver*, який може виконувати обчислення в декількох паралельних потоках. Найбільш трудомісткими у генетичному алгоритмі є генерування й оцінювання нового покоління. Проте при безстатевому розмноженні його можна легко розпаралелити, адже генерувати нащадків різних частин популяції можна незалежно, а додавати їх – до безпечної щодо потоків колекції *ConcurrentDictionary<double, Tour> population*. Розглянемо, як це зробити на практиці за допомогою класу *Task*.

Екземпляри класу *Task* можна запустити на виконання тільки один раз. Після того, як задачу було завершено, або зупинено, об'єкт не можна використати повторно, тому для генерування нового покоління метод *PSolver.Solve* щоразу створює новий масив *Task[] tasks* – по одному завданню на кожен потік – і кожному його екземпляру передає межі частини колекції найкращих турів, маркер завершення і генератор випадкових чисел для операторів випадкових змін. Найпростіше це зробити, коли наперед відома кількість потоків. Наприклад, якщо потоків два, то створення *tasks* набуде вигляду

```
Task[] tasks = new Task[] {
    Task.Run(() => MakeGeneration(0, topTours.Length / 2,
                        token, random[0])),
    Task.Run(() => MakeGeneration(
                        topTours.Length / 2, topTours.Length,
                        token, random[1])) };
```

і, завдяки використанню ініціалізатора масиву, обидві задачі буде створено коректно.

Виявилося, що для створення масиву задач довільного розміру необхідно використати додаткові локальні змінні, як у фрагменті нижче.

```
Task[] tasks = new Task[threadsCount];
for (int i = 0; i < threadsCount; ++i)
{   Random rnd = random[i];
    int lowerLimit = bounds[i]; int upperLimit = bounds[i + 1];
    tasks[i] = Task.Run(() =>
           MakeGeneration(lowerLimit, upperLimit, token, rnd));
}
```

Без такого посередництва створені задачі поділятимуть спільні змінні та непередбачувано впливатимуть на виконання одна одної. Зокрема, за спостереженнями авторів, внутрішній цикл генерування нащадків у методі *MakeGeneration* може завершуватися достроково, що призводить до виродження популяції (вона порожніє) і до аварійного завершення всього алгоритму.

Завершення генерування покоління потребує синхронізації потоків. Це легко зробити за допомогою статичного методу *Task.WhenAll(tasks).Wait()*.

На виконання метод *solver.Solve* запускаємо за допомогою асинхронного методу та класу *Task*, що дає змогу відображати поточний розв'язок у вікні застосунку, доки програма шукає наступне наближення приймати команди від користувача.

```
private async Task RunGeneticAlgorithmAsync(ISolver<Tour> solver)
{
    // До запуску завдання необхідно приготуватися:
    // - створити маркер зупинки
    cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;
    // - з його допомогою можна задати дію після завершення
    token.Register(() => isSolved = true);
    // - створити об'єкт-інформатор, що відображатиме хід обчислень
    Progress<int> progressHandler = new Progress<int>(value =>
    {
        lblStopwatch.Text = "Минуло " +
                            stopwatch.Elapsed.ToString(@"mm\:ss\.ff");
        lblGenerations.Text = string.Format(
                                  "Перевірено {0} поколінь", value);
        pnlMap.Invalidate();
    });
    stopwatch.Restart();
    // Власне обчислення в окремому потоці
    //  та наступне після обчислень завдання
    await Task.Run(() => solver.Solve(token, progressHandler))
         .ContinueWith(task => pnlMap.Invalidate());
    stopwatch.Stop();
}
```

З тексту методу видно, що перед запуском завдання треба створити маркер завершення *token* і налаштувати його. Методом *Register* можна зареєструвати дію, яка спрацює одразу, як тільки маркер отримає повідомлення *Cancel*, незалежно від того, чи завдання вже завершило свою роботу. *ProgressHandler* інкапсулює дію, що відображає хід обчислень. У нашому прикладі вона відображає тривалість обчислень, кількість поколінь і схему карти міст.

Ключове слово *async* у заголовку методу змінює спосіб опрацювання результату його виконання і дає змогу використати в його тілі інструкцію *await* для запуску довільного асинхронного процесу. У середовищі .Net для цього зазвичай використовують екземпляри класів *Task* і *Task<Result>*. Завдання запущено в інструкції *await*, що не блокує потік обслуговування інтерфейсу користувача під час обчислень. Дію, яку буде виконано після завершення завдання, задають методом *ContinueWith*. У нашій програмі після відшукання туру буде виведено номери міст на карті.

Таблиця 1

Тривалість розв'язування задачі комівояжера (у *сек.*)

Кількість міст, n	Кількість мутацій		Розмір популяції, s		
	m	r	40	80	120
100	7	3	32.9	66.2	99.2
100	14	6	65.7	131.8	200.0
300	7	3	95.5	192.0	288.8

Таблиця 2

Швидкодія багатопотокового алгоритму

Кількість міст, n	Розмір популяції, s	Кількість мутацій		Тривалість виконання (у *хв.:сек.*) для заданої кількості потоків				
		m	r	1	2	4	6	8
100	80	7	3	1:06.21	0:47.35	0:33.93	0:38.93	0:35.96
120	100	8	4	1:58.79	1:19.06	0:59.30	1:04.28	1:01.56
160	80	8	4	2:08.71	1:21.24	1:01.33	1:06.37	1:02.40
230	100	10	5	3:20.07	2:04.25	1:30.72	2:20.75	1:37.09
300	100	12	5	6:58.20	4:10.31	3:02.16	3:24.38	3:03.93

5. Аналіз швидкодії

Оцінимо обчислювальну складність описаного раніше генетичного алгоритму розв'язування задачі комівояжера. Щоб згенерувати наступне покоління турів, клонують кожного члена популяції й до отриманих копій застосовують безстатеві оператори мутації. Нехай n – кількість міст, s – розмір популяції, m і r – кількості мутацій одного члена популяції за допомогою операторів типу I і типу II, відповідно. Легко бачити, що оператор типу I виконує один обмін пари номерів міст, а оператор типу II – в середньому $n/4$ обмінів і для турів великої довжини спричиняє основні обчислювальні затрати. Отже, генерування одного покоління потребує $k = s(m+r)$ копіювань туру, пов'язаних з виділенням динамічної пам'яті, і $s(m+r\times n/4)$ обмінів міст. Операція відбору потребує k обчислень довжини туру і стільки ж вставлень у впорядковану колекцію турів, яку ми використовуємо в програмі для зберігання популяції. Зауважимо, що складність вставляння у стандартний впорядкований контейнер є не гіршою за $O(k\log_2 k)$. Звільнення динамічної пам'яті після відбору виконується одною командою, її тривалість залежить від особливостей роботи системного збирача сміття.

Після створення застосунку було перевірено, як зміна параметрів алгоритму впливає на тривалість його виконання, як кількість потоків виконання впливає на його швидкодію. Програму запускали на комп'ютері з чотириядерним процесором Intel® Core™ i5-5200U 2.20 GHz і 64-розрядною ОС для розв'язування задач з різною кількістю міст і різними параметрами генетичного алгоритму. У всіх випадках кількість поколінь обмежена значенням 5000. У табл. 1 наведено середні значення тривалостей роботи алгоритму для задач зі 100 і 300 містами для різних

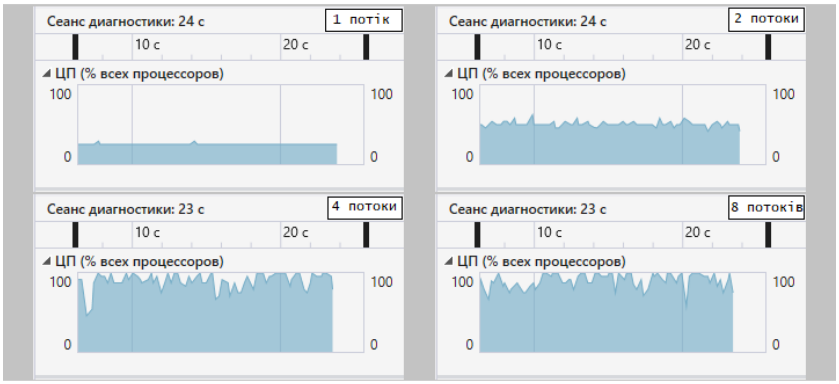

Рис. 2. Завантаженість процесорів під час роботи з різною кількістю потоків

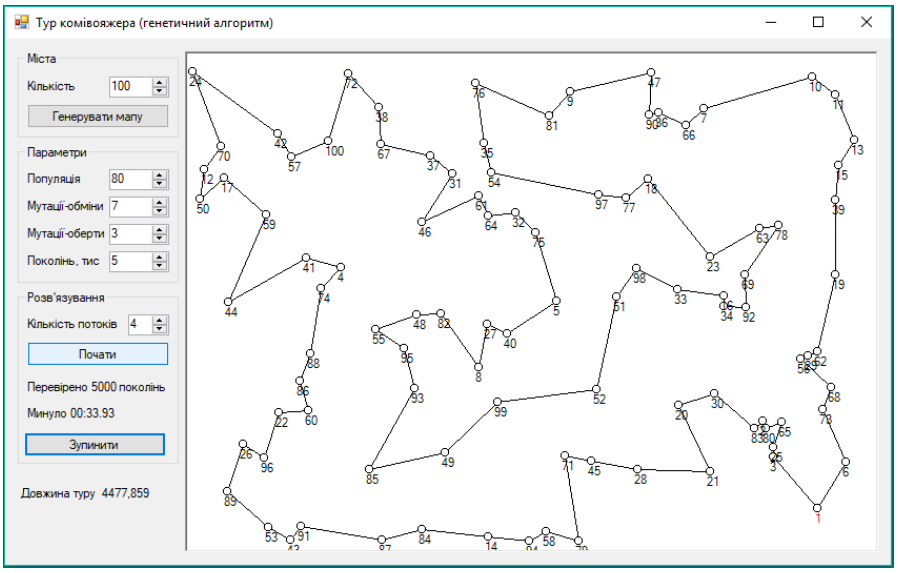

Рис. 3. Графічний інтерфейс програми

значень параметрів s, m і r. Видно, що від кожного з них тривалість виконання залежить лінійно, що узгоджується з теоретичними оцінками.

Зміна кількості потоків виконання суттєво вплинула на швидкодію алгоритму. Видно (табл. 2), що збільшення кількості потоків до 4 суттєво зменшує час виконання. Наступне збільшення уже не має такого ефекту, оскільки тоді операційна система планує декілька потоків на одне ядро процесора, даються взнаки накладні витрати на перемикання контексту виконання.

Завантаженість процесора комп'ютера для різної кількості потоків виконання зображено на рис. 2. Знімок засобів діагностики зроблено під час розв'язування задачі для 160 міст. Видно, що найщільніше процесори завантажені для 4 потоків.

На рис. 3 показано вікно застосунку під час розв'язування однієї з задач.

6. Висновки

Клас *System.Threading.Tasks.Task* є зручним інструментом для побудови багатопотокових застосунків, що дає змогу виконувати тривалі обчислення у фоновому потоці. Його можна також використовувати для розпаралелювання обчислень, що підвищує ефективність використання процесорного часу, але потребує уважного ставлення до створення завдань, використання відповідних структур даних і синхронізації процесів у застосунку. Еволюційні алгоритми, реалізовані в багатопотокових застосунках дають змогу отримувати за невеликий час наближені розв'язки навіть таких *NP*-повних задач, як задача відшукання туру комівояжера для великої кількості міст.

Список використаної літератури

1. *Троелсен Э.* Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен; пер. с англ. – Москва: ООО "И.Д.Вильямс", 2013. – 1312 с.
2. Event-based Asynchronous Pattern (EAP) [Електронний ресурс] / [Ron Petrusha, Peter Kulikov, Maira Wenzel, others] // Microsoft Docs, 2018. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous - programming - patterns /event - based - asynchronous-pattern-eap – Назва з екрану.
3. Task-based Asynchronous Pattern (TAP) [Електронний ресурс] / [Ron Petrusha, Maira Wenzel, Mike Jones, others] // Microsoft Docs. – 2017. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous - programming - patterns/task - based - asynchronous - pattern - tap – Назва з екрану.
4. Traveling Salesman Problem [Електронний ресурс] / S. Yaroshko // github. – 2019. – Режим доступу: https://github.com /mr-Serg /TravelingSalesmanProblem – Назва з екрану.
5. Various Implementations of Asynchronous Background Tasks [Електронний ресурс] / S. Cleary. – Режим доступу: http://blog.stephencleary.com/2010/08/various - implementations - of - asynchronous.html – Назва з екрану.
6. *Fogel B.* Fogel Evolutionary computing / David B. Fogel // IEEE Spectrum. – February. – 2000. – P. 26–32.
7. *Larranaga P.* Genetic algorithms for the travelling salesman problem: a review of representations and operators / P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, S. Dizdarevic // Artificial Intelligence Review. – 1999. – Vol.v13, No. 2. – P. 129–170.

Стаття: надійшла до редколегії 18.06.2019
доопрацьована 04.09.2019
прийнята до друку 18.09.2019

CONSTRUCTION OF MULTITHREADED PROGRAMS IN THE .NET FRAMEWORK

Sr. Yaroshko[1], Sv. Yaroshko[2]

[1]*Ivan Franko National University of Lviv,*
Universytetska Str, 1, Lviv, 79000, e-mail: serhiy.yaroshko@lnu.edu.ua
[2]*National University "Lviv Polytechnic",*
S. Bandery Str, 12, Lviv, 79013, e-mail: svitlana.m.yaroshko@lpnu.ua

The majority of modern software applications make use of several execution threads. For example, in the text editor, one thread is responsible for the spell cheking, another – for autosaving, one more thread – for handling the user input. The mechanism of multiple threads provides the software developer with a convenient way of structuring the functional possibilities: the software application architecture will be more stable, if it's various possibilities are programmed separately from each other. The ability to use multiple threads will be useful even for a specialist of numerical computing. The iteration methods usually need to execute a large amount of computations in order to receive an approximate solution of a certain problem. This circumstance should be taken into account at the stage of software design architecture so, that a user could follow the computational process and not worry about the responsiveness of the application.

In order to launch a separate execution thread, the .Net framework provides a developer with an effective template of task-based asynchronous programming [3] that uses some new keywords *async*, *await* and classes *Task*, *Task<TResult>* from the namespace *System.Threading.Tasks*. The instances of this class are created very fast, they need only a little amount of system resources and they effectively use the processors time. That's why it is convenient to move the computations to a separate thread, which on its turn can also launch different parallel computational threads. This approach significantly accelerates a software performance on a multiprocessor (or multi-core) computer.

The keyword *async* in the name of an asynchronous method changes the way of how the result of this method is handled, and gives a possibility to use the instruction *await* for launching any awaitable object. For this purpose, in the .Net framework there are usually used the instances of the classes *Task* and *Task<Result>*. The task, launched with the instruction *await*, does not block the user interface thread during the computation. Before launching an synchronous task, there are different preparation steps: create a stop-marker *CancellationToken token*, an object *Progress<T> progress* in order to be able to send the data to the main thread, and finally a set of local variables to correctly catch the execution context. Then the asynchronous method in the right moment calls *progress.Report(data)*, for a "soft" termination of an asynchronous algorithm. In a convenient point in time, this algorithm should check the property *token.IsCancellationRequested* and terminate the execution on request. The cancellation request can be initiated from the main thread by calling the method *token.Cancel()*. In order to synchronize the finishing of the array of asynchronous tasks, the static method *Task.WhenAll(tasks).Wait()* is used.

For the demonstration of the described approaches an application has been created. It demonstrates how to apply the genetic algorithm in order to solve the traveling sales-man problem [6] for a large number of cities. It is experimentally analyzed how the number of threads influences the performance of the application. The complete code base can be downloaded from the repository [4].

Key words: thread, multithread application, class Task, parallelization, cancellation of a thread execution, representation of a calculation progress, genetic algorithm, travelling salesman problem.