JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 6 (2018) 62-67
Wysłane: 2017-11-06
Przyjęte: 2017-11-13

Pięć sposobów wprowadzenia współbieżności do programu w języku C#

Paweł Szyszko*, Jakub Smołka
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie: Dzisiejsze procesory w komputerach osobistych i urządzeniach mobilnych umożliwiają coraz bardziej efektywne zrównoleglanie działań w celu szybszego uzyskania wyników. Twórcy oprogramowania mają wiele różnych możliwości zaimplementowania współbieżności, jednak zazwyczaj trzymają się jednej, najbardziej znanej sobie techniki. Warto prześledzić działanie każdej z nich, aby odkryć, kiedy można ją wykorzystać w sposób efektywny, a kiedy lepiej poszukać alternatywy. W poniższym artykule zostały przedstawione sposoby równoległej implementacji obliczeń matematycznych z wykorzystaniem wątków, zadań, puli wątków, puli zadań oraz równoległej pętli for z klasy Parallel. Wszystkie zostały napisane w języku C# na silniku Windows Presentation Foundation platformy .NET. Zaimplementowane obliczenia matematyczne to obliczenie liczby Pi z pomocą wzoru Leibniza.

Słowa kluczowe: programowanie równoległe; .Net; C#; pi

*Autor do korespondencji/

Adres e-mail: pawel.szyszko24@gmail.com

Five ways to insert concurrency to a program written in C#

Paweł Szyszko*, Jakub Smołka

Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract: Nowadays processors working in personal computers and mobile devices allow for more and more effective parallel computing. Developers have at their disposal many different methods of implementing concurrency, but usually use the one, that they now best. It is beneficial to know, when a particular technique is good and when it is better to find an alternative. This paper presents different ways of implementing parallel mathematical calculations using threads, tasks, thread pool, task pool and parallel for loop. Each method was used in a C# application running on Windows Presentation Foundation engine on .NET platform. Implemented operation is calculation value of Pi using Leibnitz's formula.

Keywords: parallel programming; .Net; C#; pi

*Corresponding author

E-mail address: pawel.szyszko24@gmail.com

1. Wstęp

Praca twórcy oprogramowania nie składa się jedynie z samego pisania kodu. Musi on także znaleźć odpowiednie rozwiązania postawionych przed nim problemów – takie rozwiązania, które będą najprostsze i najefektywniejsze. Jednym z takich problemów jest przyspieszenie i skrócenie czasu potrzebnego na wykonanie długich i skomplikowanych zadań, a jego rozstrzygnięciem – wprowadzenie współbieżności. Takie rozwiązanie generuje kolejne pytanie: w jaki sposób wprowadzić współbieżność do kodu aplikacji. Istnieje wiele możliwości na zrównoleglenie działań w programie, a większość z nich jest stosunkowo prosta w obsłudze.

Poniższy artykuł przedstawia pięć sposobów na zrównoleglenie działań w aplikacji napisanej w języku C# na platformie programistycznej .Net oraz stara się przedstawić odpowiedź na pytanie, która z nich jest najefektywniejsza i najprostsza do działań na szeregach matematycznych na przykładzie szeregu Leibniza pozwalającego na obliczenie wartości liczby π z zadaną dokładnością.

2. Dotychczasowe badania

Rozwój programowania równoległego jest na tyle szybki, że badania naukowe nie zawsze nadążają za nimi. Pomimo, że nad tym tematem pracowało i pracuje tysiące zespołów z całego świata wciąż wiele rzeczy pozostaje do odkrycia. Jedną z pierwszych prac porównujących dwie techniki wprowadzania współbieżności do programu był artykuł, napisany przez Niklausa Wirtha w 1996 [13]. Przedstawione zostały w nim wady i zalety podejścia wątkowego oraz jego alternatywy – opartego na przerwaniach systemu zadań, ale wybór technologii został pozostawiony twórcom oprogramowania. Od tamtego czasu większość badań skupia się na znalezieniu optymalnego rozwiązania (nie zawsze jest to możliwe [3,9]) – technologii zrównoleglania działań – zadanego problemu i w ten sposób przedstawia swoje wyniki [5,11]. Niestety większość badaczy skupia się jedynie na dwu wyżej wymienionych sposobach, co sprawia, że inne techniki, które z nich wyewoluowały pozostają nieprzebadane.

3. Pięć sposobów wprowadzenia współbieżności do kodu w języku C#

3.1. Kolekcja wątków

Pierwszym sposobem na oddelegowanie prac jest wykorzystanie wątku. Oczywiście wykorzystanie pojedynczego wątku nie sprawi, że działania będą wykonywane równolegle, więc należy posłużyć się większą kolekcją, jak na listingu 1. Praca ze zmiennymi globalnymi wymaga zabezpieczeń np. konstrukcji *lock* [4,10].

Każdy wątek w kolekcji należy zainicjować, przekazań mu akcję do wykonania (np. poprzez wyrażenie lambda jak w przykładzie) i wystartować poleceniem *Start()*. Język C# nie posiada wbudowanych elementów pozwalających na jednoczesne oczekiwanie na zakończenie pracy przez wiele wątków, aby więc osiągnąć taki efekt należy posłużyć się licznikiem, np. zastosowanym w przykładzie obiektem klasy *CountdownEvent*. Metoda *Signal()* tego obiektu jest po prostu dekrementacją licznika, która jest operacją atomową, więc nie ulegnie przerwaniu.

Wątki i ich kolekcje są dostępne jedynie dla aplikacji tworzonych z myślą o starszych systemach Windows przy użyciu silników WPF i Windows Forms.

Listing 1. Przykładowa implementacja kolekcji wątków

```
Thread[] threads = new Thread[workersNr];
CountdownEvent countdown = new
CountdownEvent(workersNr);
for (int j = 0; j < workersNr; j++)
{
threads[j] = new Thread(() =>
{
/.../
lock (key)
{
countdown.Signal();
}
});
}
for (int j = 0; j < workersNr; j++)
{
threads[j].Start();
}
countdown.Wait();
countdown.Dispose();
```

3.2. Kolekcja zadań

Alternatywą dla wątków w większości języków programowania wspierających współbieżność, a więc także w języku C# są zadania. Są one prostsze w obsłudze i zwalniają programistę z wielu działań związanych z synchronizacją (listing 2.), kosztem dodatkowych związanych np. ze zwolnieniem wykorzystywanych zasobów (metoda Dispose()).

Listing 2. Przykładowa implementacja kolekcji zadań

```
Task[] tasks = new Task[workersNr];
for (int j = 0; j < workersNr; j++)
{
tasks[j] =
Task.Factory.StartNew(() =>
{
/.../
});
}
Task.WaitAll(tasks);
foreach (var t in tasks) t.Dispose();
```

W przeciwieństwie do wątków zadania mogą być uruchamiane w momencie inicjalizacji, a klasa je reprezentująca posiada wbudowany mechanizm oczekiwania. Należy pamiętać, że aby zadanie było wykonywane współbieżnie trzeba posłużyć się metodami *Factory.StartNew()* lub *Run()*, a nie konstrukcją *await task*, która uruchamia kod asynchronicznie. Akcje przekazywane do zadań powinny być możliwie najkrótsze i niezależne [1].

3.3. Pula wątków

Pula wątków jest kolekcją wątków, której można przydzielić zadanie do wykonania. W przeciwieństwie do zwykłej kolekcji deklarowanej w aplikacji, jest ona zarządzana z poziomu biblioteki klas lub systemu operacyjnego co ułatwia pracę programisty. Wciąż musi on zapewnić bezpieczeństwo wątków (ang. thread-safety) i nie może przewidzieć kolejności ich zakończenia, a dodatkowo klasa *ThreadPool* tworzy tylko wątki tła, których działanie jest przerywane wraz z zakończeniem pracy aplikacji. Programista nie musi natomiast oddzielnie startować każdej akcji, która jest uruchamiana, kiedy zwolni się wątek roboczy i nadejdzie jej kolej (listing 3.).

Listing 3. Przykładowa implementacja puli wątków

```
CountdownEvent countdown = new
CountdownEvent(workersNr);
for (int j = 0; j < workersNr; j++)
{
ThreadPool.QueueUserWorkItem((index) =>
{
/.../
lock (key)
{
countdown.Signal();
}
},j);
}
countdown.Wait();
countdown.Dispose();
```

3.4. Pula zadań

Analogicznie jak pula wątków, pula zadań jest obiektem pozwalającym na wprowadzenie współbieżności do programu. Aby ją wykorzystać należy posłużyć się obiektem *TaskPoolScheduler* (listing 4.) z przestrzeni nazw *System.Reactive.Concurency* [12], która nie jest standardowym obiektem, więc musi być dołączona do projektu. Ten sposób najlepiej sprawdza się przy krótkich operacjach. Podobnie jak pula wątków, to pula zadań zarządza wykonywaniem zadań [1].

Listing 4. Przykładowa implementacja puli zadań

```
CountdownEvent countdown = new
CountdownEvent(workersNr);
TaskPoolScheduler taskpool = TaskPoolScheduler.Default;
List<IDisposable> disposables = new List<IDisposable>();
for (int j = 0; j < workersNr; j++)
{
disposables.Add(taskpool.Schedule(j,
(IScheduler scheduler, int threadIndex) =>
{
/.../
lock (key)
{
countdown.Signal();
}
return Disposable.Empty;
}));
}
countdown.Wait();
countdown.Dispose();
foreach (var d in disposables) d.Dispose();
```

Możliwe jest posługiwanie się wieloma rodzajami planisty (*TaskPoolScheduler*), a nawet tworzenie własnych, ale można także, jak w przykładzie, wykorzystać obiekt domyślny.

3.5. Pętla równoległa

Równoległe implementacje pętli *for* i *foreach* pojawiły się w języku C# wraz z biblioteką TPL. Te dwie konstrukcje najlepiej sprawdzają się do wykonywania serii takich samych zadań (np. obliczania szeregu liczbowego). Wykorzystanie pętli równoległej *for* jest zdecydowanie najprostsze [8] i gdyby nie potrzeba oczekiwania na wynik, można by ją zapisać w jednej linii kodu (nie licząc akcji do wykonania) – listing 5.

Listing 5. Przykładowa implementacja równoległej pętli for

```
CountdownEvent countdown = new
CountdownEvent(workersNr);
Parallel.For(0, workersNr, (_) =>
{
/.../
lock (key)
{
countdown.Signal();
}
});
countdown.Wait();
countdown.Dispose();
```

4. Algorytmy badawcze

4.1. Algorytm sekwencyjny

Wykorzystanie algorytmu sekwencyjnego w połączeniu z ww. sposobami oddelegowania obliczeń, pozwala na odciążenie wątku głównego, ale nie ma na celu przyspieszenia obliczeń. Nie różni się on niczym od swego odpowiednika, który byłby uruchamiany w wątku GUI (listing 6.).

Listing 6. Algorytm sekwencyjny implementujący szereg Leibniza

```
for (int j = 0; j < calculations; j++)
{
Pi = 0;
for (int i = 0; i < sequenceLength; i++)
{
if (i % 2 == 0)
Pi = Pi + (decimal)4 / (decimal)(2 * i + 1);
else Pi = Pi - (decimal)4 / (decimal)(2 * i + 1);
}
}
```

Algorytm składa się z dwóch pętli *for*, z których pierwsza pozwalana na wielokrotne ponowienie obliczeń – pojedyncze obliczenie wartości szeregu było zbyt szybkie, aby możliwe było odczytanie czasu trwania obliczeń, a druga przebiega po całym szeregu i w zależności od parzystości lub nieparzystości licznika dodaje lub odejmuje kolejny element.

4.2. Algorytm równoległy

Wątek główny zapełnia kolekcję obiektami odpowiedniego typu w zależności od wykorzystywanej technologii w liczbie przekazanej przez zmienną *workersNr*. Do robotników nie są przekazywane żadne argumenty startowe. Każdy z nich deklaruje własną zmienną *myI* przechowującą informację o aktualnej wartości iteratora, a następnie w sekcji krytycznej [6] pobiera wartość globalną i inkrementuje zmienną globalnego iteratora [7]. Kiedy robotnik uzyska informację nad którą wartością ma pracować wchodzi w pętlę, której nie opuści dopóki pobrana wartość *myI* nie będzie większa niż zadana liczba obliczeń π. W pętli robotnik oblicza wartość zgodnie z sekwencyjnym algorytmem, a następnie zapisuje ją do tablicy wynikowej i pobiera kolejną wartość iteratora. Kiedy pętla się zakończy kończy się także działanie robotnika (listing 7.).

Listing 7. Algorytm równoległy obliczania liczby π

```
for (int j = 0; j < workersNr; j++)
{
threads[j] = new Thread(() =>
{
int myI;
lock (key)
{
myI = i;
i++;
}
while (myI < calculations)
{
decimal myPi = 0;
for (int k = 0; k < sequenceLength; k++)
{
if (k % 2 == 0)
myPi = myPi + (decimal)4 / (decimal)(2 * k + 1);
else
myPi = myPi - (decimal)4 / (decimal)(2 * k + 1);
}
lock (key)
{
PiValues.Add(myPi);
myI = i;
i++;
}
}
lock (key)
{
countdown.Signal();
}
});
}
```

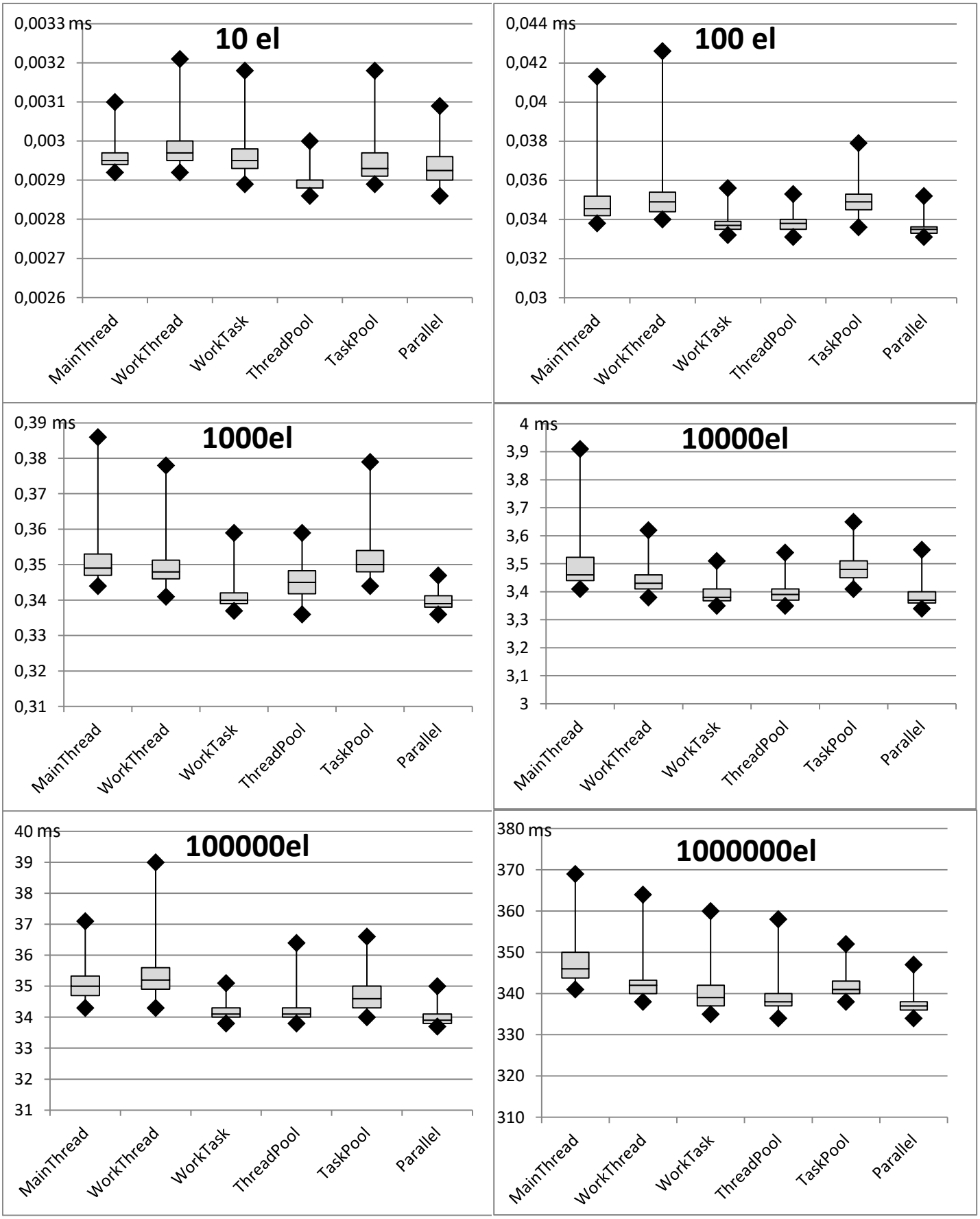
5. Wyniki badań

W procesie badawczym posługiwano się dwoma dodatkowymi zmiennymi liczbowymi. Zmienna *sequenceLength* pozwalała na zmianę długości szeregu, a co za tym idzie rozmiar pracy do wykonania, a zmienna *calculations* – liczby powtórzeń całego procesu obliczenia szeregu. Manipulowano nimi tak, aby pomiar czasu pozwalał na odczytanie ok. dwóch cyfr znaczących.

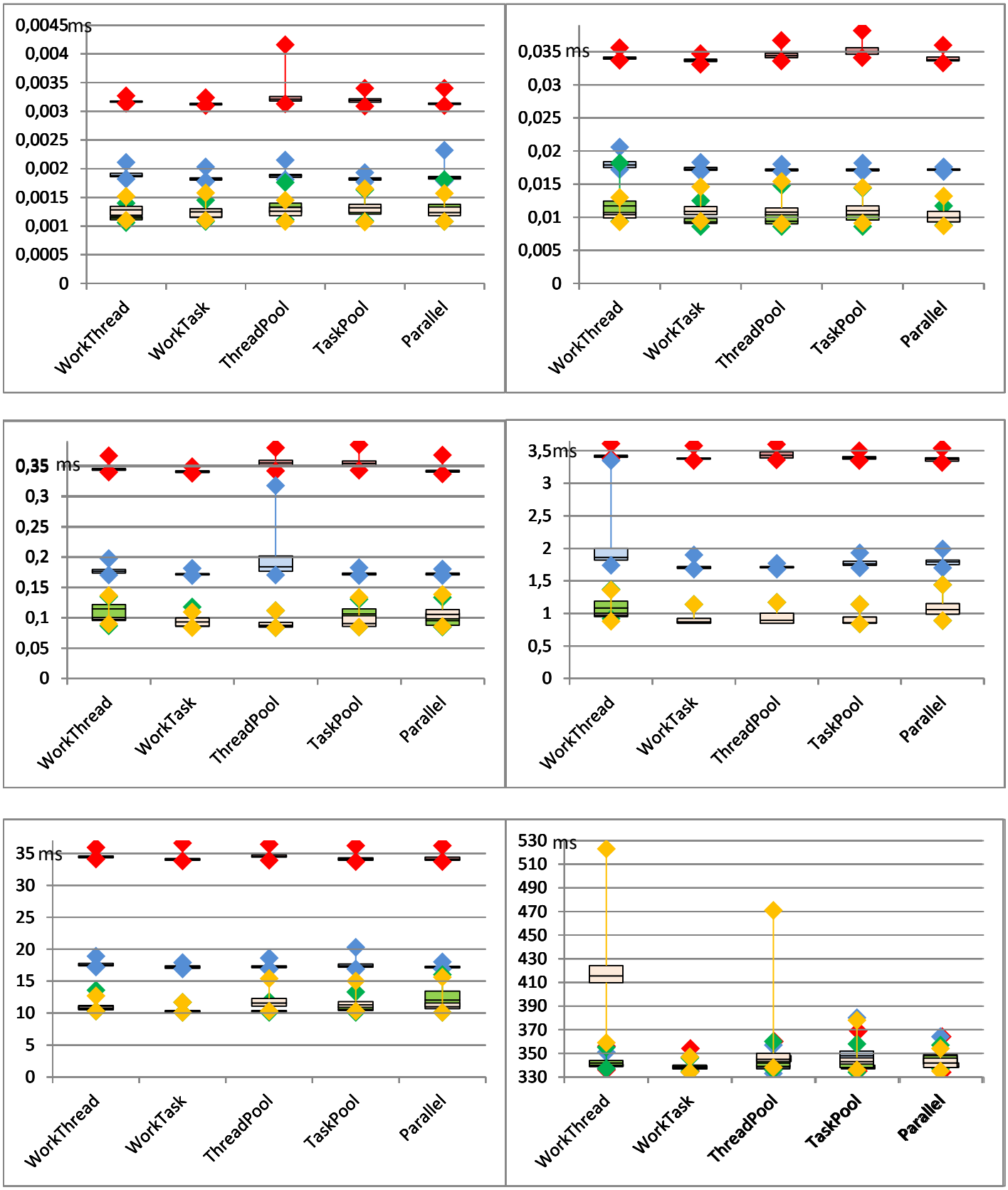
5.1. Algorytm sekwencyjny

Wykorzystanie algorytmu sekwencyjnego, poza wątkiem głównym, okazało się być tak samo wydajne, jak jego uruchomienie w wątku GUI (wykresy na rys. 1. str. 4). Widoczne są różnice niewielkie statystyczne, ale są one na tyle małe, że można uznać, że czas wykonania algorytmu nie zależy od tego, czy jest uruchamiany w wątku interfejsu użytkownika, czy poza nim. Oznacza to, że oddelegowanie prac powinno mieć miejsce zawsze, w najgorszym wypadku zwolnione zostaną zasoby wątku głównego.

Wartymi uwagi są „pudełka" przedstawiające czas wykonania obliczeń przez WorkTask, ThreadPool i Parallel. Są one zbliżone w większości przypadków zbliżone do siebie, co jest zgodne z informacją, że pracują w oparciu o tę samą technologię.



Rys. 1. Wykresy prezentujące czas obliczenia szeregu Leibniza z wykorzystaniem algorytmu sekwencyjnego



Rys. 2. Wykresy prezentujące czas obliczenia szeregu Leibniza z wykorzystaniem algorytmu równoległego; kolory odróżniają od siebie wykresy czasu obliczeń wykonywanych przez różną liczbę robotników: czerwony – 1, niebieski – 2, zielony – 4, pomarańczowy – 8

5.2. Algorytm równoległy

Prezentowane na rysunku 2. (str. 5) wykresy przedstawiają statystyki czasu obliczeń wartości liczby π dla każdej z pięciu technologii programowania równoległego i różnych długości szeregu Leibniza. W przeciwieństwie do swoich odpowiedników prezentujących czas obliczeń z wykorzystaniem algorytmu sekwencyjnego zawierają dodatkową informację na temat liczby robotników wykonujących zadanie. I tak: czerwonym kolorem zostały zaznaczone „pudełka" prezentujące statystyki czasu obliczeń wykonywanych przez jednego robotnika, niebieskim – dwóch, zielonym – czterech i pomarańczowym – ośmiu.

Wykorzystanie algorytmu równoległego jest uzasadnione tylko, jeżeli działanie, które ma być wykonane jest podzielne na niezależne od siebie elementy. Jeżeli nie jest (jak w przypadku ostatniego zadania – obliczenia jednego długiego szeregu), obliczenia są wykonywane i tak przez jednego robotnika, więc czas wykonania jest zbliżony do czasu działania algorytmu sekwencyjnego, a nawet większy, gdyż dodawany jest narzut czasowy związany z synchronizacją. W pozostałych przypadkach, gdy do wykonania była praca możliwa do rozdzielenia, udział kolejnych robotników w pozytywny i dość przewidywalny sposób wpływał na czas obliczeń, nigdy nie udało się jednak osiągnąć m-krotnego przyspieszenia działania, a jedynie bliski tej wartości wynik (gdzie m to liczba robotników).

Analiza wyników pozwala stwierdzić, że żadna z badanych technologii programowania równoległego nie wybija się ponad inne, więc każda z nich może być z równie dobrym skutkiem wykorzystywana do obliczeń zbliżonych do działań na szeregach liczbowych. Jest to o tyle pozytywna wiadomość, że każdy programista może sam zdecydować, która technologia najbardziej mu odpowiada.

6. Podsumowanie

Przeprowadzenie badań nad pięcioma technologiami programowania równoległego w języku C# pozwoliło stwierdzić, że każda z nich tak samo dobrze nadaje się do obliczeń współbieżnych oraz sekwencyjnych, oraz, że samo przeniesienie obliczeń poza wątek GUI pozwala na zwiększenie wydajności aplikacji.

Każda z przedstawionych technologii ma inną implementację, inne wymagania i inne ograniczenia, ale wybór konkretnej warto pozostawić twórcom oprogramowania opierającym się na swoich doświadczeniach i wykorzystujących określone biblioteki i frameworki [2].

Literatura

[1] Blaar H., Lange T., Winter R., Karnstedt M.: *Possibilities to solve the clique problem by thread parallelism using task pools.* 19th IEEE International Parallel and Distributed Processing Symposium 2005.

[2] Bugnion Laurent: *MVVM : Multithreading and Dispatching in MVVM Applications.* MSDN Magazine, 2014.

[3] Choudhury Olivia i inni: *Balancing Thread-Level and Task-Level Parallelism for Data-Intensive Workloads on Clusters and Clouds*. IEEE International Conference on Cluster Computing, 2015.

[4] Jin Jiangming, Zhang Yang, Tang Shanjiang, Fan Hongfei: *Performance Modeling and Analysis for Critical Section Contention in Parallel Codes.* IEEE Trustcom/BigDataSE/ISPA, 2016.

[5] Lee Jiyeon, Chwa Hoon Sung, Lee Jinkyu, Shin Insik: *Thread-level priority assignment in global multiprocessor scheduling for DAG tasks.* Journal of Systems and Software, 2016.

[6] Nowak Robert *Współdzielenie obiektów w aplikacjach współbieżnych.* Software Developers Journal 2010

[7] Paolieri Marco i inni: *A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-threaded Hard Real-Time Tasks.* 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2011.

[8] Schubring Tadeusz: *Algorytmy równoległe w języku programowania C#.* TTS Technika Transportu Szynowego, 2016.

[9] Schwan K., Zhou H.: *Dynamic scheduling of hard real-time tasks and real-time threads.* IEEE Transactions on Software Engineering, 1992.

[10] Smoliński Mateusz: *Elimination of task starvation in conflictless scheduling concept.* Information Systems in Management vol. 5 (2) 2016, 237−247.

[11] Tousimojarad Ashkan i inni: *Number of Tasks, not Threads, is Key.* 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015.

[12] Warczak Mateusz i inni: *Programowanie równoległe i asynchroniczne w C# 5.0*. Wydawnictwo Helion, 2014.

[13] Wirth Niklaus: *Tasks versus Threads: An Alternative Multiprocessing Paradigm*. Springer-Verlag 1996.