JCSI 2 (2016) 120-126
Journal of Computer Sciences Institute

Wysłane: 2016-11-18
Przyjęte: 2016-11-24

Wpływ zastosowania programowania równoległego na wydajność algorytmów kryptograficznych

Mateusz Kraska*, Piotr Kozieł*

Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule zostały porównane możliwości zrównoleglenia dwóch algorytmów z dziedziny kryptografii: szyfrowania XOR oraz wyszukiwania kolizji funkcji skrótu MD5. Prezentowane rozwiązania zostały zaimplementowane w taki sposób, by używały wybraną przez użytkownika liczbę rdzeni procesora. Wydajność algorytmów została sprawdzona na różnych procesorach i przedstawiona na wykresach.

Słowa kluczowe: programowanie; równoległe; synchroniczne; kryptografia.

*Autor do korespondencji.

Adresy e-mail: Mateusz.kraska@pollub.edu.pl; Piotr.koziel@pollub.edu.pl

The influence of the parallel programming on the performance of cryptographic algorithms

Mateusz Kraska*, Piotr Kozieł*

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

Abstract. Paper compares possibility to parallelize two cryptographic algorithms: Xor cipher and MD5 collision search. Presented solutions allows user to specify number of used processors. Performance of algorithms where tested on a different types of processors and visualized on graphs.

Keywords: programming; parallel; asynchronous; cryptography.

*Corresponding author.

E-mail addresses: Mateusz.kraska@pollub.edu.pl; Piotr.koziel@pollub.edu.pl

1. Wstęp

Programowanie równoległe jest stosowane niemal od początku istnienia informatyki. Zrównoleglanie obliczeń w wielu przypadkach jest jednym możliwym sposobem by skrócić czas obliczeń. Jednak dopiero początek XXI wieku spowodował popularyzację tego zagadnienia. W wieku XX, większość procesorów posiadała jeden rdzeń. Producenci procesorów zwiększali ich wydajność poprzez zwiększenie częstotliwości taktowana. Zgodnie z obserwacjami Gordona Moor'a, częstotliwość taktowania podwajała się co około 2 lata. Na początku XXI wieku, produkowano procesory osiągające częstotliwość pracy bliską 4 GHz, jednak od tego czasu wzrost częstotliwości taktowania został zahamowany. Tranzystory w układach scalonych osiągnęły na tyle małe rozmiary, że dalsza ich miniaturyzacja stała się bardzo trudna. Aby zwiększać wydajność procesorów, producenci rozpoczęli produkcję procesorów wielordzeniowych. Komputery posiadające takie procesory, pojawiły się masowo już w pierwszych latach XXI wieku. Obecnie, nawet w najtańszych komputerach, laptopach, a także urządzeniach mobilnych typu smartphone procesory jednordzeniowe są spotykane bardzo rzadko. Programiści, aby w pełni wykorzystać moc urządzeń, projektują swoje aplikacje w taki sposób by wykorzystywać wiele rdzeni procesora.

Celem opisanej w niniejszym artykule pracy badawczej była analiza wydajności zastosowania programowania asynchronicznego. Posłużono się przy tym porównaniem wyników zrównoleglenia popularnych algorytmów i języków programowania stosowanych w informatyce.

2. Stanowisko badawcze

Do wykonania wszystkich badań i obliczeń posłużono się prywatnym sprzętem autorów artykułu. Uznano, że badania te będą najbardziej wartościowe gdy zostaną wykonane na sprzęcie domowym, dostępnym dla zwykłego użytkownika.

Tabela 1. Parametry sprzętu komputerowego wykorzystanego do pomiarów

Procesor	**Liczba rdzeni**	**Pamięć RAM**	**System operacyjny**	**Wersja JVM / .NET**
Intel Core i3-4160	2 fizyczne, 4 logiczne	2 X 4GB DDR3 800Mhz	Windows 10 64 bit	.NET Framework 4.6.2
AMD Athlon 64 X2 Dual Core 4200+	2 fizyczne	2 x 2GB DDR2 800MHz	Windows 7 64 bit	.NET Framework 4.6.2
MediaTek MT6580	4 fizyczne	1 x 1GB	Android 5.1 Lollipop	Nie dotyczy

3. Zrównoleglenie algorytmu XOR

Algorytm XOR jest jednym z najprostszych i jednocześnie najmniej bezpiecznych algorytmów służących do szyfrowania danych [1,2]. Jego działanie polega na wykonaniu operacji XOR na kolejnych bajtach danych, które należy zaszyfrować z kluczem. Deszyfrowanie jest operacją identyczną

wykorzystującą ten sam klucz. Szyfr wykonany tym algorytmem można złamać w bardzo prosty sposób metodą ataku statystycznego. Ze względu na łatwość implementacji oraz niską złożoność obliczeniową, algorytm XOR jest przedmiotem badań w niniejszej pracy.

Listing 1. Przykładowy kod realizujący szyfrowanie XOR

```
void Method1(byte[] array, byte key)
{
    for (inti = 0; i<array.length; i++)
    {
     array[i] ^= key;
    }
}

(...)

byte[] dataArray = new byte[] { 2, 4, 17, 0, 255, 170, 1 }
Method1(dataArray, 170);
```

Pierwszy parametr metody Method1 służy do przekazania danych do zaszyfrowania lub deszyfrowania. Drugi parametr to klucz. Metoda nie zwraca danych. Zaszyfrowane zostają dane wejściowe. Po wykonaniu operacji przedstawionej na listingu 1, tablica dataArray zawierać będzie elementy: 168, 174, 187, 170, 0, 171. Aby przystosować algorytm do wykorzystywania wielu rdzeni należało zrównoleglić operację szyfrowania oraz deszyfrowania danych. W tym celu zmodyfikowano metodę Method1.

Listing 2. Przykładowy kod realizujący szyfrowanie XOR

```
void Method1(byte[] array, int start, int end, byte key)
{
    for (inti = start; i< end; i++)
    {
     array[i] ^= key;
    }
}
byte[] dataArray = new byte[] { 2, 4, 17, 0, 255, 170, 1 }
Method1(dataArray, 170,0,4);
```

Do metody dodano dwa parametry: start oraz end. Oznaczają one pierwszy oraz ostatni indeks, które zostaną zaszyfrowane. Dzięki nim obliczenia możemy rozdzielić na kilka wątków.

3.1. Wykorzystanie klasy Thread do równoległego wykonania obliczeń

Podstawową metodą do zrównoleglenia obliczeń na platformie .NET jest wykorzystanie klasy Thread [3]. Inicjalizacja obiektu Thread powoduje utworzenie nowego zarządzanego wątku na platformie .NET. Bazowy konstruktor tej klasy przyjmuje jako argument instancję klasy ThreadStart. Jednak dzięki zastosowaniu wyrażenia lambda nie musimy jawnie tworzyć obiektu klasy ThreadStart.

Poniższy kod (Listing 3) inicjalizuje wątek, który szyfruje 4 pierwsze elementy tablicy. Następnie wywołanie metody Start powoduje rozpoczęcie obliczeń w nowym wątku. Metoda Join() [3,4] służy do synchronizacji wątku głównego z nowo stworzonym wątkiem. Dzięki temu mamy pewność, że po wykonaniu metody Join() na obiekcie wątku, zakończył on swoje działanie.

Przedstawiony na listingu 3 przykład tworzy wątek za pomocą klasy Thread, ale obliczenia nie są zrównoleglone. Aby zrównoleglić obliczenia potrzebujemy co najmniej dwa wątki, pracujące jednocześnie.

Listing 3. Przykładowy kod realizujący szyfrowanie w osobym wątku

```
byte[] dataArray = new byte[] { 2, 4, 17, 0, 255, 170, 1 }
var thread = new Thread(() => {
    Method1(dataArray, 0, 3, 170);
});
thread.Start();
thread.Join();
```

Metoda przedstawiona na listingu 4 przyjmuje 4 parametry. Pierwszy z nich to dane do zaszyfrowania (lub odszyfrowania). Drugi parametrem jest klucz. Kolejny, trzeci parametr, to liczba wątków jaka ma być wykorzystana do wykonania zadania. Ostatni, czwarty parametr, to delegat do metody wykonującej szyfrowanie, czyli do metody Method1().

Listing 4. Metoda realizująca równoległe szyfrowanie za pomocą obiektów klasy Thread

```
void CalculateInManyThreads(byte[] dataArray, byte key,
intthreadCount, Action<byte[], int, int, byte> action){
    Thread[] threads = new Thread[threadCount];
    intelementsForThread = dataArray.Length / threadCount;
    for (inti = 0; i<threadCount; i++){
     int start = i * elementsForThread; int end;
     if (i == threadCount - 1){
          end = dataArray.Length;
     }
     else{
          end = (i + 1) * elementsForThread;
     }
     threads[i] = newThread(() =>     {
          action(dataArray, start, end, key);
     });
    }
    for (inti = 0; i<threadCount; i++){
     threads[i].Start();
    }
    for (inti = 0; i<threadCount; i++){
     threads[i].Join();
    }
}
```

Przedstawiona na listingu 4 Funkcja CalculateInManyThreads() najpierw tworzy tablicę, która przechowywać będzie instancje klasy Thread. Rozmiar tej tablicy jest określany przez parametr threadCount. Następnie, w pętli for, jest obliczany zakres tablicy, który ma być zaszyfrowany przez poszczególne wątki. Tworzone są także poszczególne instancje klas Thread. Kolejna pętla uruchamia wszystkie wątki za pomocą metody Start(). Ostatni fragment metody, to trzecia pętla for, której wykonywanie kończy się dopiero wtedy, gdy wszystkie wątki zakończą swoje działanie. Wywołanie metody Join na obiekcie klasy Thread blokuje wątek wywołujący, aż do czasu zakończenia pracy wątku reprezentowanego przez ten obiekt.

3.2. Wykorzystanie klasy Task do równoległego wykonania obliczeń

W czwartej wersji .NET Framework, wprowadzona została klasa Task [4, 5]. Podobnie jak Thread, służy ona do zrównoleglania operacji. Jej działanie opiera się na ThreadPool. Oznacza to, że stworzenie instancji obiektu klasy Task nie musi oznaczać utworzenia nowego wątku. O tym, czy zainicjowanie nowego obiektu Task powoduje również stworzenie obiektu Thread decyduje TaskScheduler. Jeśli TaskScheduler działa w domyślnej konfiguracji to liczba utworzonych wątków powinna być równa ilości rdzeni procesora. Ma to znaczenie wtedy, gdy chcemy wykonywać wiele różnych zadań jednocześnie, ponieważ inicjalizacja obiektu Thread jest operacją kosztowną.

Listing 5. Metoda realizująca równoległe szyfrowanie za pomocą obiektów klasy Task

```
void CalculateInManyTasks(byte[] dataArray, byte key,
inttaskCount, Action<byte[], int, int, byte> action){
    Task[] tasks = new Task[taskCount];
    intelementsForThread = dataArray.Length / taskCount;
    for (inti = 0; i<taskCount; i++){
     int start = i * elementsForThread;
     int end;
     if (i == taskCount - 1)    {
          end = dataArray.Length;
     }
     else{
          end = (i + 1) * elementsForThread;
     }
    tasks[i] = new Task(() => { action(dataArray, start, end,
key); });}
    for (inti = 0; i<taskCount; i++){
     tasks[i].Start();
    }
    Task.WaitAll(tasks);
}
```

Metoda CalculateInManyTasks() (Listing 5) jest bardzo podobna do poprzedniej o nazwie CalculateInManyThreads(). Podstawowa różnica polega na wykorzystaniu klasy Task zamiast Thread. Kolejną różnicą jest wykorzystanie metody Task.WaitAll() w celu poczekania na zakończenie pracy przez wszystkie Taski.

W CalculateInManyThreads() do tego celu służyła pętla, która czekała na wszystkie wątki za pomocą metody Join(). W przypadku tego algorytmu, synchronizacja wątków jest dość prosta w implementacji, co może nie przekonywać do zastosowania klasy Task. Oprócz tego klasa Task, posiada bliźniacze funkcje Task.WaitAny() oraz Task.WhenAny() [6]. Obie metody służą do oczekiwania na zakończenie pracy przez dowolny wątek, spośród kilku przekazanych w parametrze. Dodatkowo, metoda Task.WhenAny() zwraca Task, który jako pierwszy zakończył pracę. Task.WhenAny() została wykorzystana w rozdziale dotyczącym wyszukiwania kolizji md5. Aby przeprowadzić podobne działanie z wykorzystaniem klasy Thread zamiast Task, należałoby wykorzystać mechanizmy synchronizacji oparte na klasach AutomaticResetEvent, ManualResetEventlubSemaphor lub Semaphor co wymagało by bardziej złożonej implementacji, oraz czyni kod trudniejszym w interpretacji.

3.3. Równoległe wykonanie obliczeń z wykorzystaniem Simple Instruction Multiple Date (SIMD)

W wersji 4.6 platformy .NET Framework pojawiło się wiele nowych funkcjonalności. Jedną z nich jest wsparcie dla instrukcji procesorów Intel o nazwie Simple Inctruction Multiple Date (SIMD) [7]. Instrukcje SIMD, są to instrukcje wykonywane nie na pojedynczych danych, lecz na całych wektorach. Instrukcje SIMD są obsługiwane zarówno w procesorach w architekturze x86 jak i x64. Instrukcje w procesorach mogą się różnić. Procesory x86 Intel Pentium 4 oraz AMD Athlon 64 obsługują zestaw instrukcji SSE2. Zestaw instrukcji AVX posiada część procesorów Intel z serii Sandy Bridge, Ivybridge, Haswell. [7].

Poniższa metoda Method2() (Listing 6), realizuje to samo działanie co Method1(), ale korzystając z SIMD.

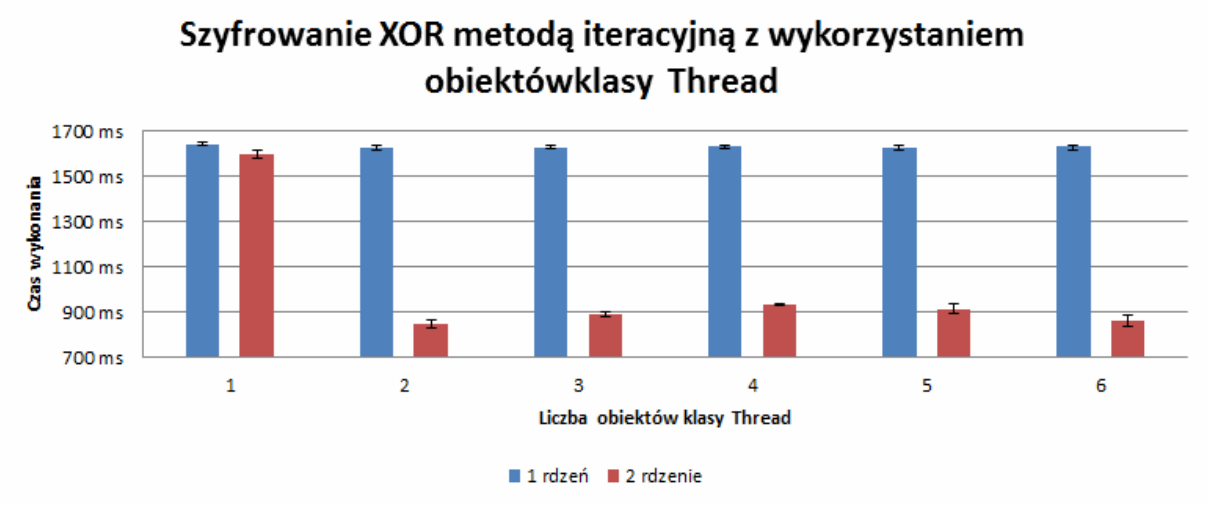
Listing 6. Metoda realizująca równoległe szyfrowanie za pomocą obiektów klasy Task

```
private static void Method2(byte[] array, int start, int end,
byte key)
{
    byte[] keysArray = new byte[Vector<byte>.Count];
    for (int j = 0; j <Vector<byte>.Count; j++){
     keysArray[j] = key;
    }
    Vector<byte>keysVector = new
Vector<byte>(keysArray);
    vartmpEnd = end - Vector<byte>.Count; inti;
    for (i = start; i<tmpEnd; i += Vector<byte>.Count){
     Vector<byte>outputVector = newVector<byte>(array, i)
^ keysVector;
     outputVector.CopyTo(array, i);
    }
    byte[] b = new byte[Vector<byte>.Count];
    Array.Copy(array, i, b, 0, end - i);
    Vector<byte>outputVector = new Vector<byte>(b) ^
keysVector;
    outputVector.CopyTo(b);
    b.Take(end - i).ToArray().CopyTo(array, i);
    }
}
```

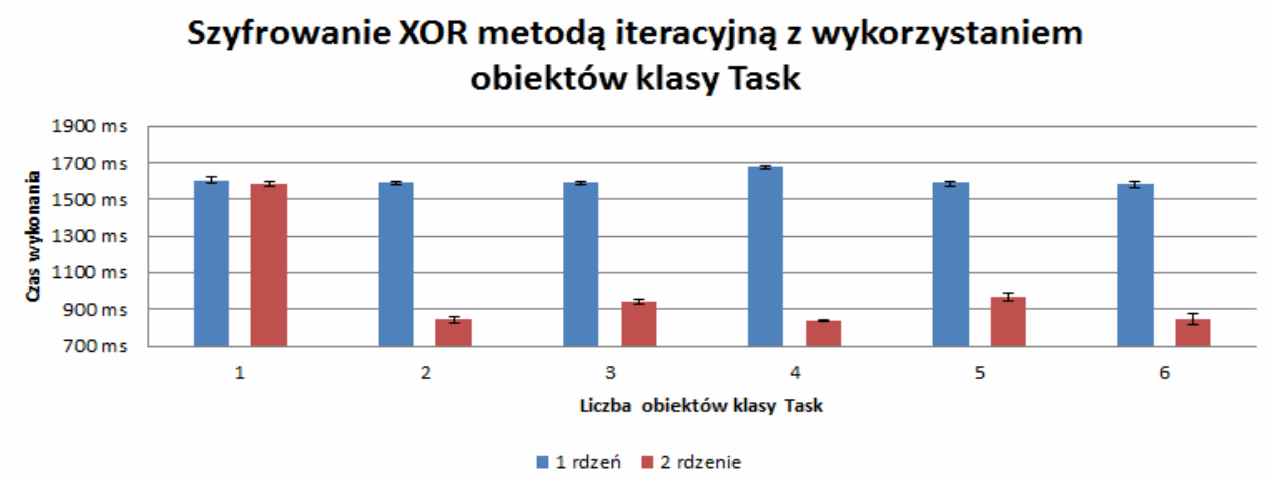
Na początku Method2 (Listing 6) tworzony jest wektor kluczy keysVector. Następnie w pętli for tworzone są wektory kolejnej porcji danych, oraz wykonywana jest operacja XOR wektora danych z wektorem klucza.

3.4. Pomiary Wydajności

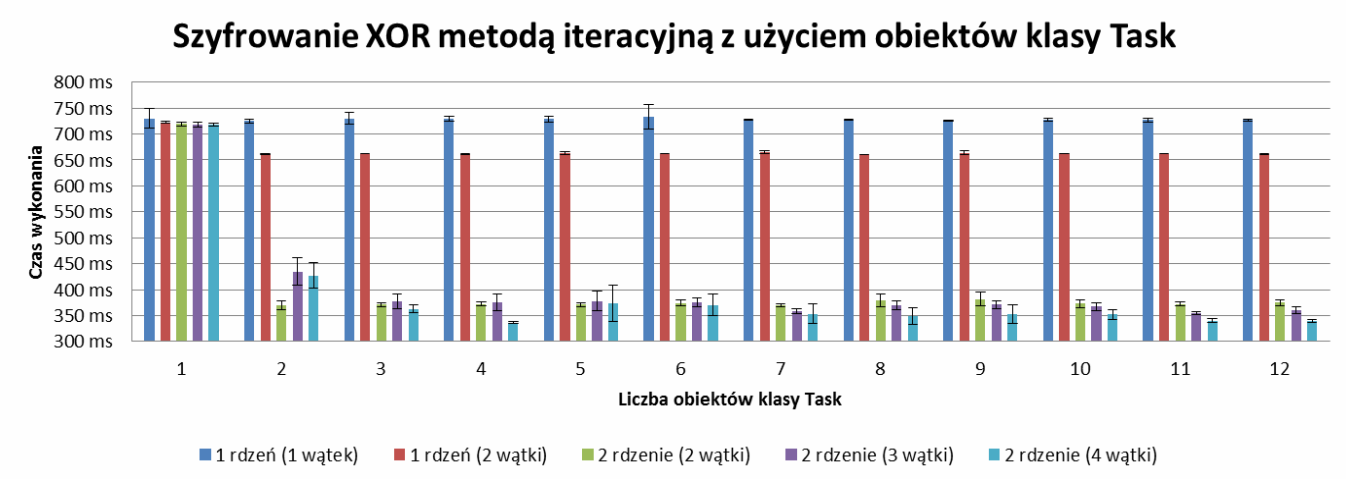
W celu porównania wydajności prezentowanych rozwiązań przeprowadzono szereg testów. Ich wyniki zaprezentowano na wykresach zamieszczonych na rysunkach 1, 2, 3 oraz 4. Na osiach rzędnych zamieszczonych wykresów przedstawiono czas szyfrowania 1GB danych, natomiast osie odciętych wskazują liczbę wątków (obiektów klasy Thread lub Task). Wykresy przedstawiają uśrednione czasy wykonywania obliczeń z 50 prób. Aby otrzymać tablicę bajtów o takim rozmiarze, utworzono plik o wielkości 1 GB z losową zawartością. Następnie, plik został wczytany do tablicy. Pomiary zostały wykonane na kilku różnych komputerach.



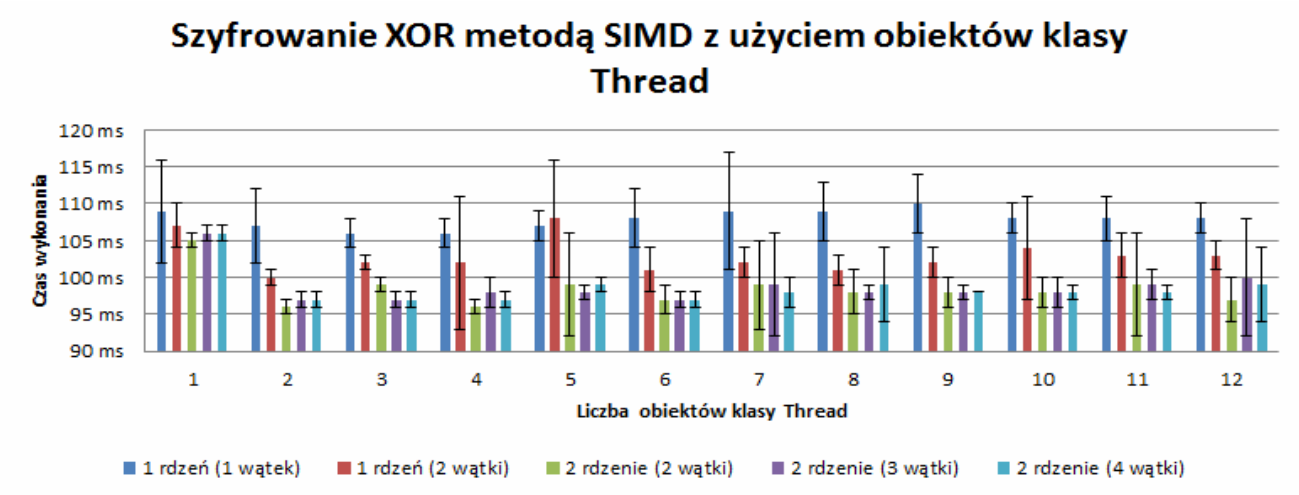
Rys. 1. Pomiar wydajności algorytmu szyforwania XOR z wykorzystaniem obiektów klas Thread na komputerze z procesorem AMD Athlon 64 X2Dual Core 4200+



Rys. 2. Pomiar wydajności algorytmu szyforwania XOR z wykorzystaniem obiektów klas Task na komputerze z procesorem AMD Athlon 64 X2 Dual Core 4200+



Rys. 3. Pomiar wydajności algorytmu szyforwania XOR z wykorzystaniem obiektów klas Task na komputerze z procesorem Intel i3-4160



Rys. 4. Pomiar wydajności algorytmu szyforwania XOR z wykorzystaniem obiektów klas Thread oraz SIMD na komputerze z procesorem Intel i3-4160

4. Znajdowanie kolizji funkcji skrótu MD5

MD5 jest popularnym algorytmem funkcji skrótu. Z ciągu danych o dowolnej (także równej zero) długości generowany jest skrót o wielkości 128 bitów. Algorytm, mimo, że jest dość stary oraz potencjalnie niebezpieczny, nadal jest często wykorzystywany. W niniejszym rozdziale przedstawiony zostanie również algorytm znajdujący kolizję metodą siłową (bruteforce). Zostaną również zbadane różnice w wydajności, gdy algorytm jest zrównoleglony.

Metoda FindCollision (Listing 7), jako parametry przyjmuje skrót, dla którego szukana będzie kolizja; losowy ciąg znaków; oraz obiekt klasy CancellationTokencancelationToken [6], który służy do przerywania obliczeń.

Listing 7. Metoda FindCollision

```
string FindCollision(string inputHash, string randomString,
CancellationTokencancelationToken) {
    string hash2 = randomString, input2;
    do{
     if (cancelationToken.IsCancellationRequested){
          return null;
     }
     input2 = hash2;
     hash2 = CalculateMD5Hash(input2);
    }
    while (inputHash.Equals(hash2));
    return input2;
}
```

Aby zrównoleglić obliczenia, napisano metodę, która w wielu wątkach uruchamia metodę CalculateInManyTasks (Listing 8).

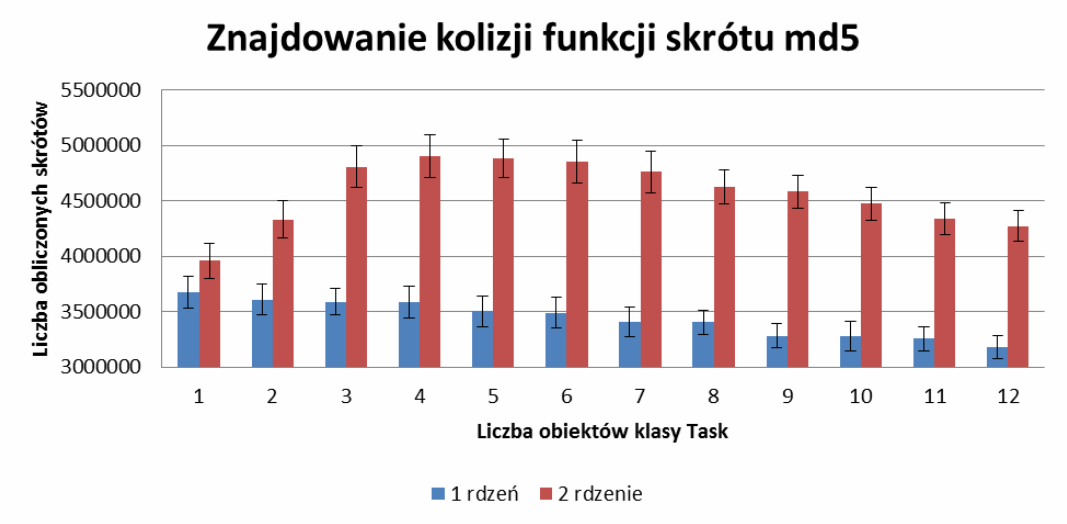
Listing 8. Metoda CalculateInManyTasks

```
static Task<string>CalculateInManyTasks(int taskCount){
    Task<string>[] tasks = new Task<string>[taskCount];
    int[] numberOfHashedCalculated = new int[taskCount];
    varcancelToken = new CancellationTokenSource();
    string input = RandomString(r);
    string hash = CalculateMD5Hash(input);
    for (inti = 0; i<taskCount; i++){
     string randomString = RandomString(r);
     int i2 = i;
     tasks[i] = new Task<string>(() =>{
          return FindCollision(hash, randomString,
     cancelToken.Token);
     });
    }
    for (inti = 0; i<taskCount; i++){
     tasks[i].Start();
    }
    TaskcompletedTask = await Task.WhenAny(tasks);
    cancelToken.Cancel();
    returncompletedTask.Result;
}
```

W przeciwieństwie do poprzedniego przykładu – szyfrowania danych metodą XOR, w tym doświadczeniu program nie oczekuje na zakończenie pracy wszystkich obiektów klasy Task. W momencie gdy jeden z wątków znajdzie kolizję i tym samym zakończy swoje działanie, praca pozostałych wątków nie jest już potrzebna, a więc jest kończona poprzez uruchomienie metody Cancel() obiektu cancelToken. Po zakończeniu pracy wszystkich wątków, zwracany jest wynik.

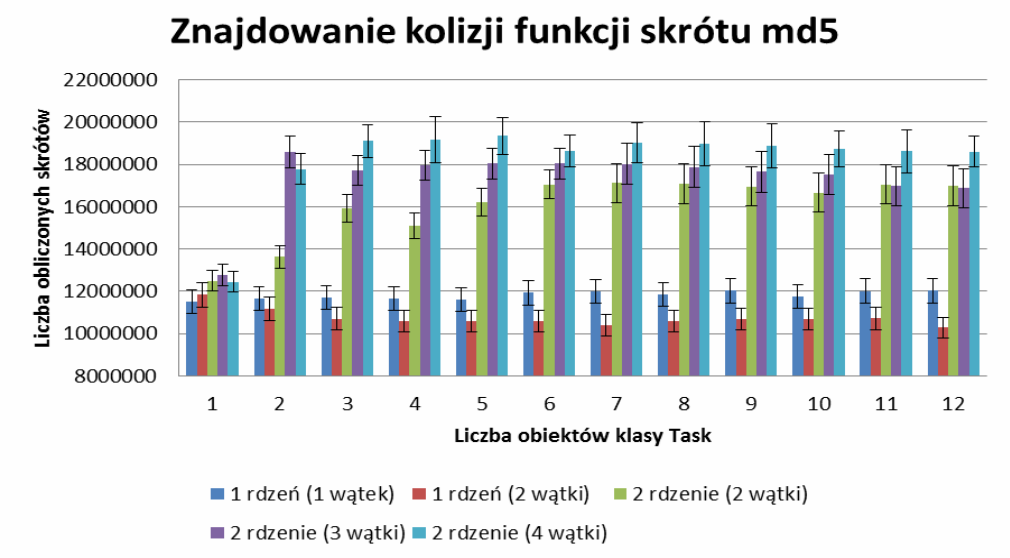
4.1. Pomiary wydajności

Do wykonania wszystkich pomiarów posłużono się sprzętem komputerowym opisanym w rozdziale 4. Wyniki zostały przedstawione w postaci wykresów zamieszczonych na rysunkach 5, 6 oraz 7. Na osi rzędnych przedstawiono liczbę obliczonych skrótów w ciągu 60 sekund pracy programu. Wykresy przedstawiają wartość średnią z 50 prób. Oś odciętych wskazuje liczbę obiektów klasy Task, które zostały użyte do obliczeń. Zastosowanie obiektów klasy Task zamiast Thread nie spowodowało zmiany wydajności. Na urządzeniu mobilnym VKWorld VK700 Max nie ograniczano liczby pracujących rdzeni procesora tak jak w przypadku pozostałych pomiarów. Dzięki zastosowaniu frameworka Xamarin, kod napisany w języku C# został bez istotnych zmian skompilowany na urządzenie mobilne pracujące na systmie Android 5.1. Warto zauważyć, że w przypadku tego doświadczenia, zrównoleglanie obliczeń dało największy zysk wydajności na urządzeniu mobilnym.



Rys. 5. Pomiar wydajności algorytmu wyszukiwania kolizji funkcji skrótu na komputerze z procesorem Athlon 64 X2 Dual Core 4200+

Rys. 6. Pomiar wydajności algorytmu wyszukiwania kolizji funkcji skrótu na urządzeniu mobilnym z procesorem MediaTek MT6582



Rys. 7. Pomiar wydajności algorytmu wyszukiwania kolizji funkcji skrótu na komputerze z procesorem Intel i3-4160

5. Wnioski

Wydajność programów została sprawdzona na kilku komputerach klasy PC. Uruchamiane aplikacje wykorzystywały różną liczbę wątków oraz różną liczbę rdzeni procesora. Pozwoliło to na zbadanie jaki zysk wydajności można osiągnąć dzięki wykorzystaniu kilku rdzeni procesora.

Dzięki wykorzystaniu dwóch rdzeni na komputerze z procesorem AMD Athlon 64 X2 Dual Core 4200+ udało się obliczyć 33% więcej skrótów, niż w przypadku użycia jednego rdzenia. Na komputerze z procesorem Intel i3-4160, dzięki wykorzystaniu dwóch rdzeni logicznych, na jednym rdzeniu fizycznym, udało się obliczyć 17% więcej skrótów niż w przypadku użycia tylko jednego rdzenia logicznego. Wykorzystanie dwóch rdzeni logicznych, na dwóch osobnych rdzeniach fizycznych, pozwoliło obliczyć 40% więcej funkcji skrótu, natomiast wykorzystanie wszystkich 4 rdzeni logicznych pozwoliło na obliczenie 54% więcej skrótów niż w przypadku wykorzystania jednego rdzenia logicznego.

W przypadku algorytmu szyfrowania XOR również na każdym urządzeniu, użycie większej liczby rdzeni procesora spowodowało skrócenie czasu operacji. Wykorzystanie klasy Task zamiast klasy Thread nie powoduje spadku wydajności. Na komputerze, z procesorem AMD Athlon 64 X2 Dual Core 4200+, wykonywanie obliczeń z wykorzystaniem dwóch rdzeni procesora zajmuje 50% czasu w porównaniu do obliczeń na jednym rdzeniu. Komputer z procesorem Intel i3-4160 posiada 2 rdzenie fizyczne. Na każdy z nich przypadają dwa rdzenie logiczne. Wykorzystanie dwóch rdzeni logicznych na jednym fizycznym rdzeniu procesora daje zysk wydajności około 13% w porównaniu do obliczeń na tylko jednym rdzeniu logicznym. Natomiast użycie dwóch rdzeni fizycznych daje aż 99% zysk wydajności w porównaniu do obliczeń na jednym rdzeniu. Wykorzystanie wszystkich 4 rdzeni logicznych daje 120% wzrost wydajności w porównaniu do obliczeń na jednym rdzeniu, oraz 10% wzrost wydajności w porównaniu do obliczeń na dwóch osobnych rdzeniach fizycznych. Dzięki zastosowaniu technologii SIMD, w przypadku algorytmu szyfrowania XOR, udało się osiągnąć wzrost wydajności o ponad 600% w porównaniu do metody iteracyjnej, używając tylko jednego rdzenia procesora. Użycie 4 liczby rdzeni zamiast jednego w przypadku metody SIMD, daje zysk wydajności o wartości zaledwie 10%. Warto zaznaczyć, że obliczenia metodą SIMD na jednym rdzeniu procesora wykonują się ponad 200% szybciej w porównaniu do tradycyjnej metody iteracyjnej z użyciem wszystkich 4 rdzeni logicznych procesora. Oprócz zysku wydajności, warto zaznaczyć, że dzięki zastosowaniu klasy Task, zamiast klasy Thread uzyskano krótszy, zwięźlejszy i prostszy w zrozumieniu kod.

Literatura

[1] Stallings W.: Kryptografia i bezpieczeństwo sieci komputerowych. Matematyka szyfrów i techniki kryptologii, Helion, Gliwice 2011-11-24, ISBN Książki drukowanej: 978-83-246-2986-2, 9788324629862

[2] Karbowski M.: Podstawy kryptografii. Wydanie III, Helion, Data wydania ebooka: 2015-01-08, ISBN Ebooka: 978-83-283-0958-6, 9788328309586

[3] Warczak M., Matulewski J., Pawłaszek R.: Programowanie równoległe i asynchroniczne w C# 5.0, Helion, Gliwice 2013-11-22 , ISBN Książki drukowanej: 978-83-246-6698-0, 9788324666980

[4] Lee Wei-Meng: 2008. Warsztat programisty C#, Wrox, ISBN Książki drukowanej: 978-83-246-2244-3, 9788324622443

[5] Alex Davies: Async in C#,O'Reilly Media,2012, ISBN:978-1-4493-3711-7

[6] Stephen Cleary: Concurrency in C# Cookbook, O'Reilly Media,2014, ISBN:978-1-4493-6755-8

[7] Numerics in the .NET Framework, (stan na dzień 20.04.2016r.)https://msdn.microsoft.com/en-us/library/dn879696(v=vs.110).aspx