JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 247–251
Received: 9 June 2020
Accepted: 16 June 2020

Performance testing of STL and Qt library elements in multi-threaded processing

Badanie wydajności elementów bibliotek STL i Qt w przetwarzaniu wielowątkowym

Piotr Krasowski*, Jakub Smołka

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

Abstract
In recent years multithreaded processing has become a important programming aspect. Computers with a multi-core processor are now widely available, enabling the creation of more efficient applications. Many libraries support multi-threaded solutions, but performance information is often lacking. The use of appropriate data structures and algorithms significantly speeds up the process of creation and development of applications. Article describes selected elements of the Qt and STL library and compares their performance in concurrent programming. The test was performed with custom applications created with C++. The time needed to perform individual operations was analysed.

Keywords: concurrent computing; multithreading; container performance; data structures

Streszczenie
Przetwarzanie wielowątkowe na przestrzeni ostatnich lat stało się ważnym aspektem programistycznym. Komputery dysponujące procesorem wielordzeniowym są obecnie powszechnie dostępne co umożliwia tworzenie wydajniejszych aplikacji. Wiele bibliotek wspiera rozwiązania wielowątkowe lecz często brakuje informacji o wydajności. W artykule opisano wybrane elementy biblioteki Qt i STL oraz porównano ich wydajność w programowaniu współbieżnym. Testy zostały przeprowadzone za pomocą autorskich aplikacji napisanych w języku C++. Wyniki przedstawiono w postaci analizy czasów potrzebnych na wykonanie poszczególnych operacji.

Słowa kluczowe: przetwarzanie współbieżne; wielowątkowość; wydajność kontenerów; struktury danych

*Corresponding author
Email address: piotr.krasowski@pollub.edu.pl (P. Krasowski)

1. Wstęp

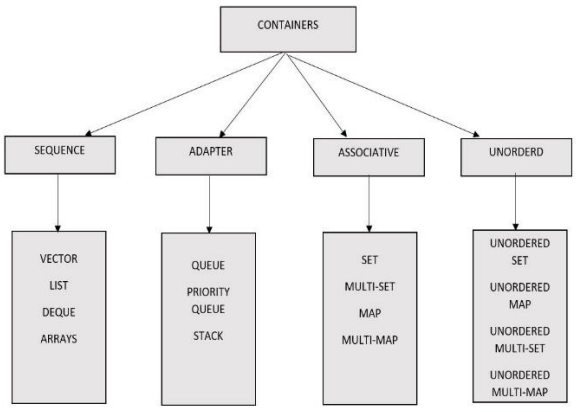
Gromadzenie i przetwarzanie danych to jedno z najważniejszych zagadnień we współczesnym świecie zdominowanym przez komputery. Pomimo imponujących maszyn potrzeba wiele czasu by przetworzyć dane tak by były użyteczne. Dzięki wydajnym komputerom i powszechnemu dostępowi do sieci zbieranie i przetwarzanie informacji jest dużo prostsze niż dawniej lecz nadal szybkość i wydajność tego procesu zależy głównie od programu. Istnieje wiele implementacji algorytmów i struktur przechowujących dane lecz ich wydajność w dużej mierze zależy od doboru odpowiednich rozwiązań do realizowanego zadania.

W dzisiejszych czasach procesory posiadają wiele rdzeni, z których każdy jest w stanie wykonywać niezależne operacje od pozostałych. Uwzględnienie aspektu wielowątkowości podczas tworzenia aplikacji pozwala na wykorzystanie pełni potencjału współczesnych maszyn. Programowanie współbieżne nie jest trywialnym zagadnieniem nawet dla doświadczonych programistów. Kluczowym aspektem na etapie projektowania aplikacji staje się dobór odpowiednich bibliotek.

Ze względu na dużą liczbę wątpliwości dotyczących wydajności poszczególnych bibliotek oraz mnogość dostępnych rozwiązań w ramach artykułu omówiono koncepcję programowania współbieżnego bibliotek Qt i STL. Zbadano i porównano wydajność wybranych kontenerów i funkcji dostępnych w bibliotece Qt w wersji 5.14 i zasobach STL.

2. Struktury danych

Wszystkie wykorzystywane w aplikacji zmienne oraz stałe są wczytywane i przechowywane w pamięci komputera. Uzyskanie dostępu do konkretnej zmiennej wiąże się z odwołaniem się do odpowiedniego adresu w pamięci. Struktury danych [1] czyli tzw. kontenery umożliwiają łatwiejszy dostęp do danych. Typy kontenerów zostały zaprezentowane na rysunku 1.



Rysunek 1: Podział kontenerów ze względu na typ – źródło [7]

Według artykułu [2] kontenery w języku C++ to klasy szablonowe wyższego poziomu. Umożliwiają one programistom łatwiejsze zarządzanie kodem. Pozwalają także na łatwe przetwarzanie danych za pomocą różnego rodzaju algorytmów.

3. Przetwarzanie współbieżne

Wykorzystanie koncepcji programowania współbieżnego pozwala na realizację przez system wielu złożonych operacji jednocześnie przy czym sposób realizacji koncepcji jest ściśle uzależniony od jednostki centralnej danej maszyny. Dawniej procesory były w stanie wykonywać tylko jedną operację jednocześnie. W takim przypadku w celu umożliwienia przetwarzania współbieżnego stosowano tzw. mechanizm przełączania kontekstu [3]. Sprowadza się on do realizacji małych fragmentów poszczególnych zadań naprzemiennie co daje efekt iluzji współbieżności. Upowszechnienie się procesorów wielordzeniowych zmieniło to podejście. Takie jednostki mogą wykonywać wiele operacji jednocześnie bez sztucznego przełączania. Ten rodzaj przetwarzania określany jest mianem współbieżności sprzętowej. Dodatkowo współczesne jednostki centralne mogą posiadać wiele wątków sprzętowych co umożliwia realizację kilku zadań równolegle przez jeden procesor bez wykorzystania mechanizmu przełączania kontekstu.

4. Standardowa biblioteka szablonów

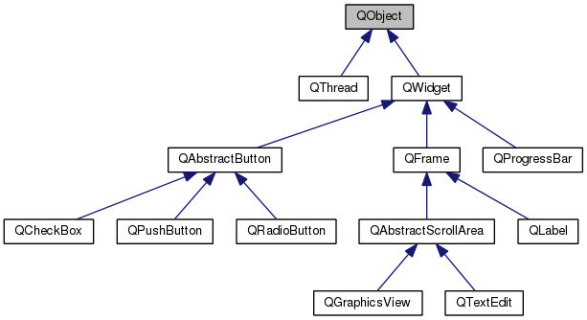
Standardowa biblioteka szablonów (ang. standard template library) [4] została stworzona w latach 90 przez Alexandra Stepanova na długo przed standaryzacją języka C++. Po tym jak jej elementy zostały przeniesione do współczesnej wersji standardu języka C++ zaprzestano używania przedrostka stl na rzecz std.

Biblioteka STL jest zbiorem ustandaryzowanych konstrukcji w formie szablonów służących do przechowywania danych [5]. Jest tzw. biblioteką generyczną, co oznacza, że jej elementy są kompatybilne z natywnymi typami języka C++ jak i typami zdefiniowanymi przez użytkownika. Główne komponenty biblioteki to::

- Algorytmy - funkcje kompatybilne z kontenerami STL, służące do wykonywania operacji takich jak sortowanie czy wyszukiwanie na danym przedziale danych,
- Kontenery - struktury danych umożliwiające łatwiejszy dostęp do danych,
- Funktory - obiekty funkcyjne umożliwiające między innymi na przekazywanie funkcji jako parametr lub przypisanie jej do zmiennej,
- Iteratory - obiekty umożliwiające dostęp do poszczególnych elementów w kontenerach i generalizację algorytmów.

5. Qt

Framework Qt [6] to zestaw bibliotek zawierający setki gotowych rozwiązań pozwalających na konstruowanie aplikacji okienkowych i konsolowych. Dostarcza także pakiet rozwiązań ułatwiający tworzenie złożonych aplikacji wielowątkowych i zarządzanie danymi. Qt jest biblioteką wieloplatformową co sprawia, że raz napisany kod może być kompilowany i uruchamiany na wielu platformach sprzętowych. Biblioteka wspiera takie języki programowania jak C++, Python czy Java Script. Rysunek 2 przedstawia hierarchie klas biblioteki Qt.



Rysunek 2: Schemat hierarchii klas w bibliotece Qt – źródło [7]

Mechanizmem odróżniającym Qt od innych bibliotek jest system sygnałów i slotów [7]. Odpowiada on za komunikację pomiędzy poszczególnymi obiektami.

6. Metoda badań

Celem badań było zbadanie wydajności wybranych kontenerów bibliotek STL i Qt oraz struktur odpowiadających za przetwarzanie wielowątkowe. Podobnie jak w testach opisanych w [8] stworzono aplikacje testujące wydajność każdego z badanych obiektów. Weryfikacja wydajności została przeprowadzona poprzez zmierzenie czasu potrzebnego na wykonanie określonych operacji. Do pomiaru czasu został użyty moduł QTest zawierający się w bibliotece Qt. Tabela 1 przedstawia zestawienie obiektów badanych klas obu bibliotek.

Tabela 1: Zestawienie kontenerów bibliotek Qt i STL

Qt	STL
QVector	std::vector
QList	std::list
QMap	std::map
QHashMap	std::unordered_map

Weryfikacja wydajności kontenerów została przeprowadzona poprzez zmierzenie czasu potrzebnego na wykonanie podstawowych operacji. W przypadku kontenerów sekwencyjnych były to funkcje odczytu i dodawania nowego elementu. W przypadku kontenerów asocjacyjnych i haszowanych zmierzono czas odczytu wartości za pomocą klucza, wstawiania elementu.

Każdy przypadek testowy został powtórzony kilkukrotnie dla różnych rozmiarów kontenera. W pojedynczym teście dana operacja jak np. dodawanie elementu została wykonana n-razy w celu uzyskania dokładniej-

szych wyników. Algorytm pojedynczego testu w przypadku kontenera można przedstawić w następujących krokach:

1. Deklaracja kontenera o rozmiarze r,
2. Wypełnienie kontenera losowymi danymi,
3. Rozpoczęcie pomiaru czasu,
4. Wykonanie danej operacji n razy,
5. Zakończenie pomiaru czasu.

W celu porównania obiektów std::thread i QThread stworzono aplikację obliczającą n-ty wyraz ciągu Fibonacciego, a następnie zmierzono czas potrzebny na wykonanie obliczenia. Każdy przypadek testowy został wykonany kilkukrotnie dla różnej liczby wątków i różnych wartości indeksów kolejnych wyrazów ciągu. Algorytm pojedynczego testu w przypadku wątku został przedstawiony w poniższych krokach:

1. Deklaracja kontenera o rozmiarze r,
2. Wypełnienie kontenera losowymi indeksami wyrazu z zakresu 5-35,
3. Rozpoczęcie pomiaru czasu,
4. Deklaracja liczby użytych wątków,
5. Podział danych w kontenerze i przekazanie ich do poszczególnych wątków,
6. Wyznaczenie wyrazu ciągu n razy na podstawie indeksu,
7. Zakończenie pomiaru czasu.

7. Wyniki badań

Pierwsza część testów dotyczyła porównania czasu odczytu dla kontenerów sekwencyjnych. Wyniki testów dla implementacji poszczególnych bibliotek przedstawia tabela 2 i 3.

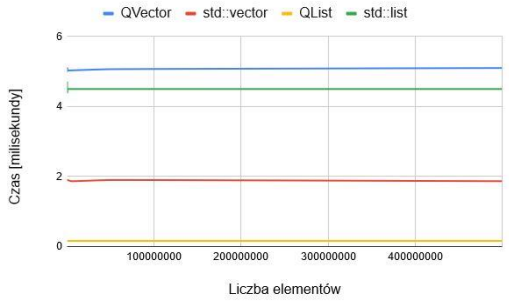
Tabela 2: Czas odczytu i wstawiania dla wektora o rozmiarze r w zależności od pozycji elementu

Typ	QVector			std::vector		
Pozycja	0	r/2	r	0	r/2	r
Czas odczytu (ms)	4,9	5,1	4,8	1,9	1,8	1,9
Czas wstawiania (ms)	23177	22533	0,01	22351	23198	0,01

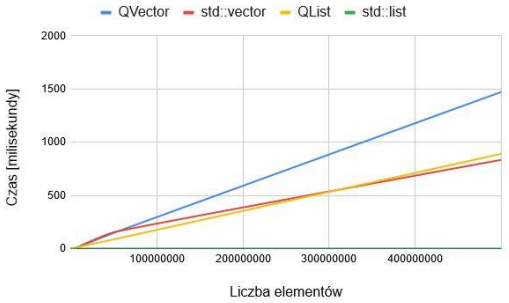
Tabela 3: Czas odczytu i wstawiania dla listy o rozmiarze r w zależności od pozycji elementu

Typ	QList			std::list		
Pozycja	0	r/2	r	0	r/2	r
Czas odczytu (ms)	0,2	0,2	0,2	0,01	0,4	0,01

Wyniki dla poszczególnych rozmiarów kontenera przedstawiono na rysunku 3 i 4.



Rysunek 3: Uśredniony czas odczytu elementu dla kontenerów sekwencyjnych



Rysunek 4: Uśredniony czas dodawania elementu do kontenerów sekwencyjncyh

Druga część testów dotyczyła porównania czasów odczytu dla kontenerów asocjacyjnych i haszujących. Wyniki badań zostały przedstawione w tabelach 4 i 5.

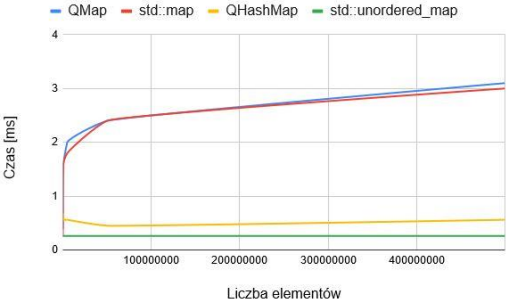
Tabela 4: Czas wykonania poszczególnych operacji dla kontenerów asocjacyjnych

Typ	QMap		std::map	
Operacja	Odczyt	Wstawianie	Odczyt	Wstawianie
Czas (ms)	2,1	2,3	2,1	10,7

Tabela 5: Czas wykonania poszczególnych operacji dla kontenerów haszujacych

Typ	QHashMap		std::unordered_map	
Operacja	Odczyt	Wstawianie	Odczyt	Wstawianie
Czas (ms)	0,58	1,4	0,26	7,2

Wyniki dla danej liczby elementów zawartych w kontenerze przedstawiono na rysunku 5 i 6.



Rysunek 5: Porównanie czasów odczytu

Rysunek 6: Uśredniony czas dodawania elementu dla kontenerów asocjacyjnych i haszowanych

Ostatnia część testów dotyczyła porównania wydajności kontenerów STL i Qt w środowisku wielowątkowym. Wyniki testów dla implementacji z użyciem poszczególnych bibliotek zostały przedstawione w tabelach 6 i 7.

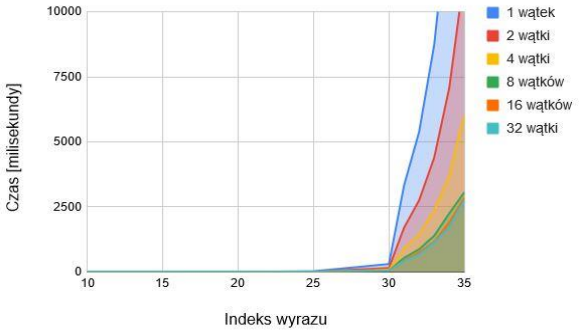
Tabela 6: Uśredniony czas wykonania operacji wyznaczenia n-tego wyrazu ciągu dla liczby użytych wątków z użyciem Qt

Nazwa biblioteki	Qt					
Liczba wątków	1	2	4	8	16	32
Czas (s)	9,4	4,2	1,9	1,2	0,9	1,5

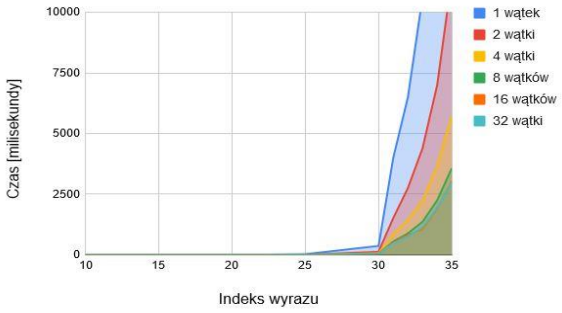
Tabela 7: Uśredniony czas wykonania operacji wyznaczenia n-tego wyrazu ciągu dla liczby użytych wątków z użyciem stl i std

Nazwa biblioteki	STL + elementy standardu C++					
Liczba wątków	1	2	4	8	16	32
Czas (s)	7,8	3,2	1,6	1,1	0,7	0,9

Wyniki dla poszczególnych indeksów ciągu i liczby wątków przedstawiono na Rys 7 i 8 .



Rysunek 7: Czas wyznaczenia wyrazu ciągu Fibonacciego o danym indeksie dla implementacji Qt



Rysunek 8: Czas wyznaczenia wyrazu ciągu Fibonacciego o danym indeksie dla implementacji stl i std

8. Wnioski

Badania pokazały, że wydajność implementacji kontenerów poszczególnych bibliotek jest uzależniona od typu użytego kontenera.

Analizując tabele 2 można stwierdzić, że wektor jest strukturą zoptymalizowaną do umieszczania elementów na końcu kontenera. Dostęp do elementów ma stały czas i nie zależy od rozmiaru struktury. W przypadku operacji wstawiania wydajniejsza jest implementacja QVector, natomiast przy odczycie lepiej sprawdza się std::vector co widać na rysunkach 3 i 4.

Na podstawie wyników przedstawionych w tabeli 3 można stwierdzić, że listy są wydajnymi kontenerami jeżeli chodzi o operacje wstawiania elementów niezależnie od pozycji. W porównaniu do wektora są mniej wydajne w operacjach odczytu.

Wyniki przedstawione w tabeli 4 potwierdzają, że struktury asocjacyjne są zoptymalizowane pod względem wyszukiwania i dostępu do elementu na podstawie klucza. Analizując tabele 5 można stwierdzić, że kontenery haszujące są wydajniejsze przy operacji odczytu i wstawiania. Na podstawie rysunków 5 i 6 można wywnioskować, że w większości przypadków testowych wydajniejszą implementacją kontenera była wersja STL.

Wyniki przedstawione w tabelach 6 i 7 udowadniają, że standard języka C++ dostarcza wydajniejsze struktury wielowątkowe. Analizując rysunki 7 i 8 trzeba stwierdzić, że nadmierny podział zadania głównego na mniejsze zadania wykonywane w środowisku wielowątkowym jest złym podejściem. Czas wykorzystywany inicjalizację obiektów i synchronizację wątków wydłuża wykonywanie obliczeń i komplikuje zarządzanie aplikacją.

Literatura

[1] M. Matsuda, M. Sato, Y. Ishikawa, Parallel array class implementation using C++ STL adaptors, 2006.

[2] H. Bischof H, Generic Parallel Programming Using C++ Templates and Skeletons, 2016, 43–55.

[3] A. Williams, Język C++ i przetwarzanie współbieżne w akcji, 2019.

[4] Opis struktury i zasady działania kontenerów C++. https://www.geeksforgeeks.org/containers-cpp-stl/, 2016, 42-55.

[5] Dokumentacja techniczna C++ dotycząca standardu, https://isocpp.org/std/, [23.05.2020].

[6] R. Penea, Mastering Qt 5, 2016, 45-68.

[7] Dokumentacja biblioteki Qt dotycząc kontenerów, https://doc.qt.io/qt-5/containers.html, [29.05.2020].

[8] B. Kyle, QThreads: An api for programming with millions of lightweight threads, 2010 24-26