Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Paweł Tryfon
Nr albumu: 248444

Biblioteka dla obliczeń równoległych w języku C++

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Marcina Benke
Zakład Logiki Stosowanej

Maj 2011

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Obecne architektury pozwalają na przyspieszanie działania programów dzięki ich wykonywaniu jednocześnie na kilku procesorach. Jednakże skorzystanie z tej możliwości przedstawia istotną trudność dla programistów, gdyż programy wielowątkowe w swojej strukturze bardzo różnią się od programów sekwencyjnych. Projektowanie i implementacja programów wykorzystujących współbieżność jest znacznie bardziej czasochłonna oraz wymaga wyższych kwalifikacji. Niniejsza praca podejmuje próbę stworzenia biblioteki Parallel, która ułatwiłaby zadanie programowania programów wielowątkowych. Głównym priorytetem było umożliwienie programiście zrównoleglania obliczeń w zwięzły, zrozumiały i prosty sposób. Obecnie nie istnieje w języku C++ żadna biblioteka oferująca taką funkcjonalność. Praca przedstawia model prowadzenia obliczeń równoległych w języku C++ oraz prezentuje proponowaną implementację. W pracy zostały szczegółowo opisane problemy, które zostały rozwiązane podczas projektowania biblioteki, takie jak: mechanizm przekazywania wyrażeń do wyliczenia równoległego, sposób prowadzenia równoległych obliczeń, sposób zwracania wyników obliczeń oraz metody zapobiegania problemom związanym z prowadzeniem równoległych obliczeń.

Słowa kluczowe

obliczenia równoległe, C++, wielowątkowość, leniwe wyliczanie, programowanie generyczne

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.3. Programming languages
D.3.3. Language Constructs and Features
Subject: Concurrent Programming Constructs

Tytuł pracy w języku angielskim

A library for parallel computing in C++ language

Spis treści

Wprowadzenie

Równoległe prowadzenie obliczeń nie jest w informatyce tematem nowym, gdyż liczy sobie już ponad 50 lat[ParHist]. Trudno jednakże wskazać konkretny punkt lub pracę, które zapoczątkowały ten kierunek rozwoju informatyki, ale niewątpliwie do jego pionierów należeli tak znani naukowcy jak Amdahl (prawo Amdahla), Flynn (klasyfikacja Flynna), Dijkstra (problem sekcji krytycznych oraz semafory) czy Petri (sieci Petriego).

Zrównoleglanie obliczeń jest jednym ze sposobów przyspieszania wydajności systemów komputerowych. Metoda ta nabrała ostatnio szczególnego znaczenia, ponieważ rozwój technologii mikroprocesorowej dotarł do takiego momentu, że przyspieszanie pojedyńczej jednostki obliczeniowej stało się trudne i w związku z tym nieopłacalne. Ze względu na to obecnie kierunek rozwoju wyznaczany jest przez równoległość, to znaczy umieszczanie w procesorach komputerów wielu układów wykonujących obliczenia równolegle w jednym czasie. Takie rozwiązanie pozwala teoretycznie na uzyskanie przyspieszenia wprost proporcjonalnego do liczby układów umieszczonych w procesorze. Aby uzyskać wspomniany wzrost wydajności, programy wykonywane na takim procesorze powinny być w stanie wykorzystać możliwości procesorów wielordzeniowych.

Jednakże, pomimo długo rozwijanej teorii oraz narzędzi wspierających, programowanie równoległe wciąż pozostaje bardzo trudne do zastosowania w praktyce. Dzieje się tak, ponieważ wykorzystanie kilku wątków wykonania programu, drastycznie zwiększa złożoność pracy programistycznej. Programista, aby napisać program współbieżny ([Barney]) musi w pierwszej kolejności zidentyfikować fragmenty obliczeń, które mogą zostać zrównoleglone. Następnie powinien zaprojektować sposób, w jaki poszczególne wątki wykonania się ze sobą komunikują i w jaki są synchronizowane. W celu zapewnienia efektywności programu programista powinien uwzględnić w projekcie również balansowanie rozkładu pracy pomiedzy poszczególne wątki wykonania.

Programistę w zmaganiach z pisaniem programów współbieżnych wspierają różnorodne narzędzia, języki programowania specjalnie zaprojektowane do obliczeń równoległych, takie jak Ada, lub biblioteki oferujące wykonywanie obliczeń równoległych w językach natywnie sekwencyjnych. Pierwsze wywiązują się zazwyczaj dobrze ze swojego zadania, gdyż są dedykowane do programowania równoległego, lecz nie są popularne w zastosowaniach praktycznych. Większym problemem jest korzystanie z drugiego typu narzędzi, ponieważ język sekwencyjny często nie pozwala na zaprojektowanie biblioteki mogącej konkurować prostotą i intuicyjnością z językiem do programowania równoległego. W praktyce potrzeba skorzystania z bibliotek dla języków sekwencyjnych występuje zdecydowanie częściej, ponieważ języki sekwencyjne są znacznie szerzej stosowane. Niniejsza praca podejmuje próbę stworzenia biblioteki Parallel dla języka C++ umożliwiającej równoległe prowadzenie obliczeń. Nadrzędnymi priorytetami podczas projektowania biblioteki było zapewnienie łatwości pisania programów równoległych z jednoczesnym pozostawieniem pełnej kontroli nad wykonaniem programu w rękach programisty. Dzięki temu miały zostać osiągnięte dwa główne cele projektowania biblioteki: zwiększenie produktywności programisty oraz zwiększenie szybkości działania programów

(przyspieszonych przez wykorzystanie równoległości).

Biblioteka Parallel pozwala na zrównoleglanie obliczeń w zwięzły sposób, który nie ingeruje znacząco w strukturę kodu. Aby z niej skorzystać wystarczy nieznacznie zmodyfikowane wyrażenie zgodne ze specyfikacją biblioteki Parallel przekazać do funkcji bibliotecznej `eval`. To powoduje przekazanie tego wyrażenia do wyliczenia przez aparat wykonawczy biblioteki Parallel i zwrócenie wyliczonej wartości. Najlepiej objaśni to przykład wykorzystania biblioteki. Posłużę się w tym celu definicją funkcji obliczającej n-tą liczbę Fibonacciego. Standardowo w języku C++ taki kod wygląda następująco:

```
1  unsigned fibo(unsigned n)
2  {
3    if (n == 0)
4      return 0;
5    else if (n == 1)
6      return 1;
7    else
8      return fibo(n-1) + fibo(n-2);
9  }
```

W przypadku wykorzystania biblioteki Parallel przyjmie on następującą postać:

```
1  #include <parallel.h>
2
3  unsigned fibo(unsigned n)
4  {
5    if (n == 0)
6      return 0;
7    else if (n == 1)
8      return 1;
9    else
10     return parallel::eval(parallel::lazyf(fibo, n-1)) + fibo(n-2);
11 }
```

Ten przykład pokazuje jak można wykorzystać bibliotekę Parallel do zrównoleglenia obliczeń w podejściu "dziel i zwyciężaj", czyli problem jest dzielony na kilka części, obliczanych równolegle, a następnie z wyników dla podproblemów obliczany jest końcowy rezultat. W tym przypadku zrównoleglenie będzie polegało na tym, że kod główny programu zleci równoległe obliczenie wyrażenia `fibo(n-1)` bibliotece Parallel i nie czekając na wynik przejdzie do następnej instrukcji, to jest obliczenia `fibo(n-2)`. Gdy obliczanie `fibo(n-1)` i `fibo(n-2)` się zakończy, zostanie obliczona suma obu wartości i zwrócona jako `fibo(n)`. Funkcja `eval` z przestrzeni nazw `parallel` służy do równoległej ewaluacji wyrażeń, a funkcja `lazyf` oznacza wywołanie funkcji podanej jako pierwszy argument z parametrami podanymi jako kolejne argumenty `lazyf`. Składnia służąca do korzystania z biblioteki Parallel zostanie opisana dokładniej w dalszej części pracy.

Pierwszy rozdział pracy opisuje koncepcję biblioteki, główne zasady nią rządzące oraz porównuje projektowane rozwiązanie do istniejących rozwiązań. Drugi rozdział traktuje o sposobie implementacji biblioteki. W trzecim rozdziale zostały przedstawione metody oraz wyniki ewaluacji biblioteki. Rozdział ostatni nakreśla możliwości dalszego rozwoju biblioteki.

Definicje pojęć i skrótów

Pojęcie	Definicja
API	Interfejs programowania aplikacji (z ang. Application Programming Interface) w kontekście bibliotek programistycznych jest opisem zestawu typów i funkcji, które dana biblioteka udostępnia.
AST	Abstrakcyjne Drzewo Syntaktyczne (z ang. Abstract Syntax Tree) jest drzewem reprezentującym strukturę kodu źródłowego w pewnym języku programowania.
Gorliwa semantyka	Semantyka, w której wyrażenia są obliczane w miejscu pojawienia się w programie. Przeciwieństwem jest semantyka leniwa, w której wyrażenia obliczane są tam, gdzie jest potrzebny ich wynik.
Idempotentność	Własność pewnej operacji, która pozwala na wielokrotne jej stosowanie bez zmiany wyniku.
Idiom C++	Konstrukcja języka C++, która często pojawia się w kodzie lub projektach doświadczonych programistów C++. Stosowanie jej uważane jest za dobrą praktykę.

Rozdział 1

Koncepcja biblioteki

W tym rozdziale zostaną przedstawione główne założenia oraz szkic projektu biblioteki Parallel. Ponadto biblioteka Parallel zostanie zestawiona z istniejącymi bibliotekami do programowania równoległego w celu pokazania różnic i uzasadnienia potrzeby stworzenia nowej biblioteki.

1.1. Cele biblioteki

Tworzeniu biblioteki Parallel przyświecały jasno sprecyzowane cele, których ideą przewodnią było ułatwienie wykorzystywania obliczeń równoległych w programach. Wymienionym poniżej celom było podporządkowane projektowanie API i implementacja biblioteki.

1.1.1. Wysoka efektywność

Jednym z głównych powodów stosowania zrównoleglania obliczeń jest przyspieszanie ich wykonania. Zatem sama biblioteka do zrównoleglania powinna działać szybko. Niedopuszczalną byłaby sytuacja, gdyby program współbieżny wykonywał się wolniej niż jego sekwencyjny odpowiednik. Biblioteka Parallel będzie biblioteką ogólnego zastosowania, przy pomocy, której będzie możliwe prowadzenie dowolnych obliczeń. W związku z tym, nie ma możliwości zoptymalizowania biblioteki pod kątem prowadzenia jednego z góry znanego typu obliczeń. Dlatego, oprócz szybkiego działania mechanizmów wbudowanych w bibliotekę, niezbędne jest pozwolenie programiście na podejmowanie decyzji o takim prowadzeniu obliczeń, że ich wykonanie przy użyciu biblioteki Parallel będzie efektywne. Kluczową rolę odgrywa odpowiedni podział zadań przez programistę.

1.1.2. Zwiększenie produktywności programisty

Problem z produktywnością programisty w przypadku pisania programów równoległych polega na tym, że takie programy są trude do napisania, więc wymagają znacznych nakładów pracy programistów. Zrównoleglenie choćby niewielkiego fragmentu programu wymaga znacznie więcej czasu niż napisanie jego sekwencyjnego odpowiednika. Być może dlatego obliczenia równoległe wykorzystywane są wyłącznie wtedy, gdy już nie ma innego sposobu osiągnięcia niezbędnego minimum wydajności programu. Biblioteka Parallel celuje w zmianę tego stanu rzeczy poprzez wprowadzenie modelu programowania równoległego, który będzie tak intuicyjny jak programowanie sekwencyjne. Dzięki temu napisanie kodu, który działa współbieżnie, będzie w wielu przypadkach prawie tak samo szybkie jak kodu sekwencyjnego, co pozwoliłoby uzyskać programistom szybsze programy przy zbliżonej produktywności ich pracy.

1.1.3. Czytelność kodu

Tym, co najbardziej utrudnia zrozumienie programów współbieżnych jest konieczność zrozumienia zależności pomiędzy odrębnymi równolegle działającymi częściami programu. Zazwyczaj te zależności dotyczą miejsc w kodzie, które są od siebie stosunkowo odległe. Mnogość niejawnych zależności i przeplotów wykonań programu sprawiają, że nawet pozornie proste operacje są trudne do poprawnego zaprogramowania, a przyczyny ewentualnych błędów są bardzo trudne do zidentyfikowania. Jednym z bardziej wymownych przykładów popierających to stwierdzenie jest problem implementacji semafora uogólnionego przy pomocy semaforów binarnych [GenSem], którego błędne rozwiązania pojawiały się w publikacjach naukowych i nawet przez kilka lat były uważane za poprawne. Stąd celem, który został postawiony przed biblioteką Parallel było stworzenie takiego modelu obliczeń, w którym obliczenia równoległe prowadzone byłyby w sposób czytelny. Oznacza to, iż miejsca wykorzystania równoległości powinny być wyraźnie widoczne i łatwe do odnalezienia, a sam zapis nie powinnien komplikować kodu. Najważniejsze jest to, że struktura programu napisanego przy pomocy biblioteki Parallel nie powinna istotnie różnić się od struktury programu sekwencyjnego. Pozwoli to na uzyskanie kodu, który będzie znacznie łatwiej zrozumieć.

1.1.4. Transparencja

Biblioteka Parallel powinna udostępniać programiście wgląd w to, w jaki sposób obliczenia równoległe będą prowadzone. Dzięki temu programista będzie mógł uwzględnić podczas programowania specyfikę biblioteki. Między innymi będzie mógł dostosować wielkość zlecanych fragmentów obliczeń (ziarnistość obliczeń), tak aby zmaksymalizować wydajność programu.

1.1.5. Abstrakcja

Abstrakcja ukrywa niepotrzebne szczegóły implementacji przed programistą, co pozwala na zwiększenie jego produktywności. Biblioteka powinna ofertować proste ogólne API, tak aby programista mógł, po zapoznaniu się z funkcjami oferowanymi przez bibliotekę, w krótkim czasie przystąpić do korzystania z Parallel.

1.1.6. Ograniczenie konieczności korzystania z mechanizmów komunikacji i synchronizacji procesów równoległych

Projektowawnie komunikacji i synchronizacji w programach współbieżnych jest czymś, co decyduje o fakcie, że programowanie równoległe jest tak trudnym zadaniem. Celem biblioteki Parallel jest zdjęcie w znacznym stopniu tego obciążenia z programisty. Komunikacja pomiędzy różnymi wątkami wykonania będzie koordynowana przez bibliotekę. Biblioteka nie może wyręczyć jednak programisty we wszystkim, ochrona spójności struktur danych programu pozostanie w rękach programisty.

1.1.7. Przenaszalność

Jest to bardzo istotna cecha biblioteki, dzięki której kod pisany przy użyciu biblioteki Parallel będzie mógł być kompilowany i wykonywany na dowolnych platformach. Zostanie to osiągnięte dzięki zaprogramowaniu biblioteki w pełnej zgodności z przyszłym standardem języka C++ (standard C++0x). Użycie nowego standardu jest niezbędne ze względu na zaawansowane konstrukcje językowe potrzebne do zaprogramowania biblioteki Parallel. Konsekwencją tego będzie niezgodność biblioteki z wcześniejszymi standardami języka C++, ale

umożliwi stworzenie lepszego, bardziej czytelnego kodu biblioteki przy użyciu nowoczesnych technik programowania w C++.

1.2. Prezentacja idei biblioteki Parallel

Niniejsza sekcja przedstawia inspirację oraz wynik końcowy pracy koncepcyjnej nad biblioteką. Zostaną zarysowane wysokopoziomowa architektura biblioteki oraz funkcjonowanie biblioteki z punktu widzenia programisty-użytkownika.

1.2.1. Inspiracja

Powstanie biblioteki zostało zainspirowane biblioteką do prowadzenia obliczeń równoległych w języku Haskell, Parallel Haskell[HasRef]. Ta biblioteka pozwala w sposób bardzo intuicyjny obliczać dwa wyrażenia równolegle. Oto przykład funkcji obliczającej w sposób równoległy n-tą liczbę Fibonacciego:

```
1  import Control.Parallel
2
3  nfib :: Int -> Int
4  nfib n | n <= 1 = 1
5      | otherwise = par n1 (seq n2 (n1 + n2))
6                    where n1 = nfib (n-1)
7                          n2 = nfib (n-2)
```

Analogiczny program sekwencyjny wyglądałby następująco:

```
1  import Control.Parallel
2
3  nfib :: Int -> Int
4  nfib n | n <= 1 = 1
5      | otherwise = (n1 + n2)
6                    where n1 = nfib (n-1)
7                          n2 = nfib (n-2)
```

W Haskellu funkcja `par` wskazuje, że wyliczenie dwóch wyrażeń może odbyć się równolegle i w czasie wykonania podejmowana jest decyzja o sposobie wyliczenia. Obliczenia odbywają się równolegle, gdy jest to bardziej efektywne od wykonania sekwencyjnego. Ta konstrukcja pokazuje z jak zadziwiającą prostotą można pisać programy równoległe. W Haskellu wystarczy dodać jedno słowo, aby oznaczyć wyrażenie jako przeznaczone do zrównoleglenia. Taka była pierwotna inspiracja dla zbudowania biblioteki Parallel. Przekazanie wyrażenia do odpowiedniej funkcji powinno poskutkować jego zrównolegleniem.

1.2.2. Zlecanie obliczeń

Zlecanie równoległego wykonania obliczeń powinno jak najmniej ingerować w sekwencyjny kod programu dla zachowania jego intuicyjności. (Dla wyjaśnienia dodam, że nie twierdzę, iż jedynie sekwencyjny kod programu może być intuicyjny, jednak praktyka pokazuje, że zrozumienie programu, który został zapisany jako kilka jednocześnie wykonywanych ciągów instrukcji jest znacznie trudniejsze od zrozumienia programu sekwencyjnego.) Do oznaczenia wyrażenia, które ma zostać wykonane równolegle będzie służyła funkcja `eval`, przyjmująca jako argument wyrażenie do wykonania.

Składnia wyrażeń powinna być jak najbardziej zbliżona do składni języka C++, po to, aby konstruowanie wyrażeń było łatwe dla programisty. Idealnie byłoby, gdyby programista mógł przekazać wyrażenie w jego standardowej postaci w języku C++, czyli w następujący sposób:

```
parallel::eval(fibo(40));
```

W tej chwili czytelnik dobrze zaznajomiony z semantyką języka C++ mógł dostrzec pewien problem, który jest związany z przekazaniem wyrażenia do wykonania. W podanym przykładzie takie wyrażenie najpierw zostałoby wyliczone, ponieważ język C++ posiada gorliwą semantykę wyliczania wyrażeń, a dopiero potem byłoby przekazane do funkcji `eval`. Zatem funkcja `eval` otrzymałaby gotowy wynik i żadne równoległe obliczenia nie byłyby już potrzebne. Takiej sytuacji należy uniknąć poprzez zaprojektowanie specjalnego sposobu przekazywania wyrażeń do funkcji `eval`. Zabieg, który należy zastosować nazywa się uleniwieniem wyrażenia i polega na odroczeniu obliczenia wartości wyrażenia do momentu, gdy ta wartość będzie potrzebna. Wyrażenie leniwe nie jest wyliczane w miejscu pojawienia się. Dzięki zastosowaniu takiej metody wyrażenie, które ma być wyliczone równolegle pojawia się w kodzie tam, gdzie jest to najwygodniejsze, w jawnej postaci, a potem może zostać przekazane do mechanizmu ewaluacji biblioteki Parallel.

Leniwe wyrażenia w języku C++

Stworzenie leniwego wyrażenia w języku C++ nie wydaje się prostym zadaniem. Domyślna semantyka obliczeń jest gorliwa, nie ma słów kluczowych pozwalających na dodanie leniwości, C++ nie pozwala również na rozszerzenie składni języka. Wskazówka do rozwiazanie problemu leniwych wyrażeń w języku C++ znajduje się w książce *More C++ Idioms*[Idioms], która przedstawia idiom C++ szablonu wyrażenia (z ang. Expression Template). Idea stworzenia szablonu wyrażenia polega na reprezentacji wyrażenia przez zbudowanie odpowiedniego drzewa typów, które jest Abstrakcyjnym Drzewem Syntaktycznym (z ang. Abstract Syntax Tree - AST) wyrażenia. Dokładny opis zastosowanej metody znajduje się w rozdziale poświęconym implementacji biblioteki.

Funkcja `eval`

Jeszcze słowo w rozdziale przedstawiającym koncepcje biblioteki należy się funkcji `eval`, która jest główną funkcją z API biblioteki Parallel. Funkcja służy do zlecania wykonania równoległego wyliczania wyrażeń, które następnie są przekazywane w formie leniwej do mechanizmu wykonawczego. Wartość zwracana przez funkcję jest uchwytem, którego można użyć do sprawdzenia wyniku obliczenia. Przekazywanie wyników obliczeń przez bibliotekę Parallel zostało omówione w sekcji Sposób przekazywania wyniku.

1.2.3. Wykonanie zadań

W tej podsekcji zostanie omówiony mechanizm wyliczania wyrażeń w bibliotece Parallel po przekazaniu ich do funkcji `eval`. Jest to serce biblioteki, dzięki któremu biblioteka potrafi pełnić swoje funkcje.

Działanie mechanizmu ewaluacji

Ewaluacja wyrażeń przekazanych do obliczenia będzie odbywała się zgodnie ze schematem producent-konsument. Producentem wyrażeń będzie kod programu korzystającego z biblioteki Parallel i przekazujący je do funkcji `eval`. Wyrażenia te trafiają do kolejki, w której

będą oczekiwały na wyliczenie. Natomiast konsumentem będą wątki-pracownicy biblioteki Parallel, których jedynym zadaniem będzie oczekiwanie na zadania (wyrażenia do ewaluacji), pobieranie ich z kolejki i wykonywanie. Ze względu na efektywność wykorzystany zostanie wzorzec puli wątków (z ang. thread pool opisany w [ThdPool]). Zatem wątek nie będzie tworzony dla wykonania każdego zadania, a będzie istniała grupa wątków dedykowanych do wykonywania obliczeń, która będzie zaalokowana w programie tak długo, jak długo program będzie korzystał z biblioteki Parallel. O liczbie wątków będzie decydował użytkownik, gdyż ich liczba może mieć znaczący wpływ na wydajność programu, a nie istnieje optymalna liczba wątków dla każdego rodzaju zastosowania biblioteki Parallel. Standardowo w tym modelu wyrażenia będą trafiały do kolejki zadań, w której będą oczekiwały na wyliczenie.

Może się zdarzyć, że wyrażenie nie zostanie wyliczone do momentu, gdy jego wynik będzie potrzebny. W tej sytuacji biblioteka Parallel reaguje w taki sposób, by nie blokować programu bez przyczyny. Główny wątek programu dotarłszy do takiego miejsca musiałby oczekiwać na wykonanie obliczeń. Lecz skoro nie ma nic innego do wykonania, to może sam wyliczyć wyrażenie, odpowiednio informując bibliotekę Parallel, aby porzuciła ewaluację danego wyrażenia. Całością procesu alternatywnego wyliczania zarządza biblioteka, więc nie jest to zauważalne dla programisty.

Jednym z rozszerzeń mechanizmu ewaluacji mogłoby być dodanie automatycznego dostosowywania liczby wątków w puli do intensywności prowadzonych obliczeń. W pierwszej wersji biblioteki nie przewidziano w projekcie takiej funkcjonalności.

Warto zauważyć, że w bibliotece nie występuje problem balansowania obciążenia różnych wątków, gdyż wszystkie wątki należące do puli są identycznie i dzięki wykorzystaniu wspólnej kolejki zadań, obciążenie jest równoważone samoczynnie. Wątek kończąc pracę nad jednym zadaniem sięga do kolejki po następne.

Mechanizmy synchronizacji

Dla poprawności działania biblioteki niezbędne było zaimplementowanie mechanizmów synchronizacji pomiędzy wątkami biblioteki a kodem programu. Zostały one opisane w sekcji Implementacja mechanizmu ewaluacji. Używając biblioteki Parallel programista nie musi martwić się o synchronizację działania biblioteki Parallel.

Biblioteka nie może jednak przewidzieć zależności w kodzie programu, dlatego ochrona danych, na których dokonywane są obliczenia pozostaje w rękach programisty. Parallel nie narzuca żadnych ograniczeń, co do korzystania w kodzie zrównoleglanych wyrażeń instrukcji dostępu do struktur danych, z których korzysta wiele wątków jednoczesnie. W tej sytuacji programista musi zadbać o odpowiednie umieszczenie sekcji krytycznych w kodzie obliczanych wyrażeń, tak aby program był bezpieczny i żywotny.

1.2.4. Sposób przekazywania wyniku

Kod korzystający z biblioteki Parallel powinien wyglądać naturalnie. Skoro do funkcji `eval` przekazujemy wyrażenie, to naturalnym oczekiwaniem jest to, że w wyniku ewaluacji otrzymamy wartość tego wyrażenia. Problem tkwi w tym, że jeśli wartość byłaby zwracana w momencie powrotu z funkcji `eval` to otrzymalibyśmy *de facto* program sekwencyjny, choć rozłożony na kilka wątków, nie byłoby możliwości wykorzystania korzyści z równoległego prowadzenia obliczeń na wielu wątkach. Dlatego wywołanie obliczenia wyrażenia przez funkcję `eval` biblioteki Parallel nie zwraca wyniku natychmiast. W momencie powrotu z funkcji `eval` będzie znany jedynie typ wyrażenia, a nie jego wartość. Wartość zostanie obliczona później przez mechanizm ewaluacji biblioteki Parallel, a kod programu przechodzi do kolejnych

instrukcji. Program może w dowolnym momencie poprosić o obliczoną wartość wyrażenia.

To podejście nazywa się komunikacją asynchroniczną i często pojawia się w kontekście programowania współbieżnego. Wykorzystują je takie technologie jak asynchroniczne zdalne wywołanie procedur bądź Message Passing Interface. Jego przeciwieństwem, w którym wynik funkcji jest zwracany w momencie powrotu z funkcji wywołującej równoległe obliczenie, jest komunikacja synchroniczna. Wybór asynchronicznego sposobu zwracania wyniku pozwala na następujące optymalizacje:

- Gdy wątek główny nie wymaga wyniku ani informacji o zakończeniu zadania, może przejść do wykonywania kolejnych instrukcji.
- Wątek główny może uruchomić kilka równoległych obliczeń jednocześnie.
- Wątek główny może uruchomić obliczenia, wykonywać inne instrukcje i odebrać wynik później, gdy będzie potrzebował.

Omówienie asynchronicznych sposobów zwracania wyniku

Programowanie używając komunikacji asynchronicznej jest trudniejsze, gdyż programista musi w odpowiedni sposób nią zarządzać. Generalnie istnieją dwa asynchroniczne sposoby zwracania wyniku. Oba zostały opisane w ksiażce [DisSys] w rozdziale dotyczącym Asychronicznego Zdalnego Wywołania Procedur (z ang. Asychronous RPC).

Pierwszy z nich to sposób jawny, w którym asynchroniczne wywołanie pewnej funkcji zwraca identyfikator wyniku. Następnie wątek, który wywołał tę funkcję może odpytywać proces wykonujący obliczenie o wynik (z ang. polling) przy pomocy identyfikatora.

Drugi sposób jest niejawny, ponieważ kod programu nie obsługuje zwracanego wyniku wprost. Proces wykonujący obliczenie wywołuje pewną funkcję, której przekazuje obliczony wynik i ta funkcja odpowiada za dalsze przetwarzanie wyniku. Jest to tak zwane zwrotne wywołanie funkcji (z ang. callback), a ten wzorzec projektowy określany jest jako przetwarzanie sterowane wydarzeniami (z ang. event-driven processing).

W kontekście projektowania biblioteki Parallel natychmiast odrzucone zostało wykorzystanie drugiego z opisanych sposobów, ponieważ stałby on w zdecydowanej sprzeczności z celami biblioteki. W przypadku niejawnego sposobu przetwarzania wyniku, kod programu używającego biblioteki Parallel stałby się mniej czytelny i trudniejszy do analizy. Zlecenie obliczenia i odebranie wyniku byłoby opisane przez dwa różne odległe od siebie miejsca w kodzie.

Zdecydowanie bliższy idei biblioteki Parallel jest sposób pierwszy. Jednakże zdecydowana większość tego typu rozwiązań zrzuca na programistę obowiązek aktywnego odpytywania o wynik. Nie jest to rozwiązaniem złym, ale zmusza programistę do zaprogramowania dodatkowych czynności. Na przykład, programista musi obsłużyć sytuacje, gdy wynik już jest dostępny lub gdy obliczenie nie zostało jeszcze wykonane. Aby uprościć użycie biblioteki Parallel warto było poszukać innego rozwiązania.

Emulowanie synchronicznego zwracania wyniku

Rozwiązanie zaprojektowane dla biblioteki Parallel opiera się na asynchronicznym zwracaniu wyniku, ale asynchroniczność jest ukryta przed programistą. Z jego punktu widzenia kod programu działa tak, jakby wynik został zwrócony w sposób synchroniczny. To wyjaśnia tytuł nadany tej podsekcji, ponieważ Parallel emuluje zachowanie synchronicznego zwracania wyniku.

Zostało to osiągnięte poprzez wprowadzenie klasy uchwytu do wyniku, obudowującej wartość zwracaną, którą nazwiemy wartością odroczoną (z ang. deferred value). Aby wartość odroczona ściśle odpowiadała typowi wynikowemu wyrażenia, będzie ona szablonem typu parametryzowanym typem wynikowym wyrażenia. Dzięki temu, że znany był typ zwracany, możliwe jest takie zaprojektowanie klasy wartości odroczonej, że przypomina w swoim zachowaniu typ wynikowy wyrażenia. W momencie, gdy wartość wyrażenia będzie niezbędna nastąpi wymuszenie ewaluacji, jeśli wartość nie została jeszcze wyliczona, i pobranie wyniku. Przykładem takiej sytuacji jest przypisanie wartości odroczonej na zmienną o typie wartości obudowywanej. Odbędzie się wtedy niejawna konwersja, do której będzie potrzebna rzeczywista wartość wyrażenia.

Obsługa sytuacji wyjątkowych

Należy również przyjrzeć się sytuacji, w której w czasie obliczania wyrażenia wystąpi błąd, zasygnalizowany przez rzucenie wyjątku. Biblioteka Parallel wspiera obsługę takiej sytuacji. Aby reakcja na takie zdarzenie była możliwa wyjątek musi zostać złapany i zasygnalizowany programiście. Przepuszczenie wyjątku przez bibliotekę skutkowałoby natychmiastowym błędnym zakończeniem programu, gdyż programista nie miałby żadnej możliwości złapania wyjątku, ponieważ wyjątek ten pochodziłby z kodu biblioteki Parallel wykonywanym w innym wątku. Biblioteka nie powinna tłumić żadnych wyjątków, gdyż wtedy programista nie wiedziałby co naprawdę dzieje się w programie.

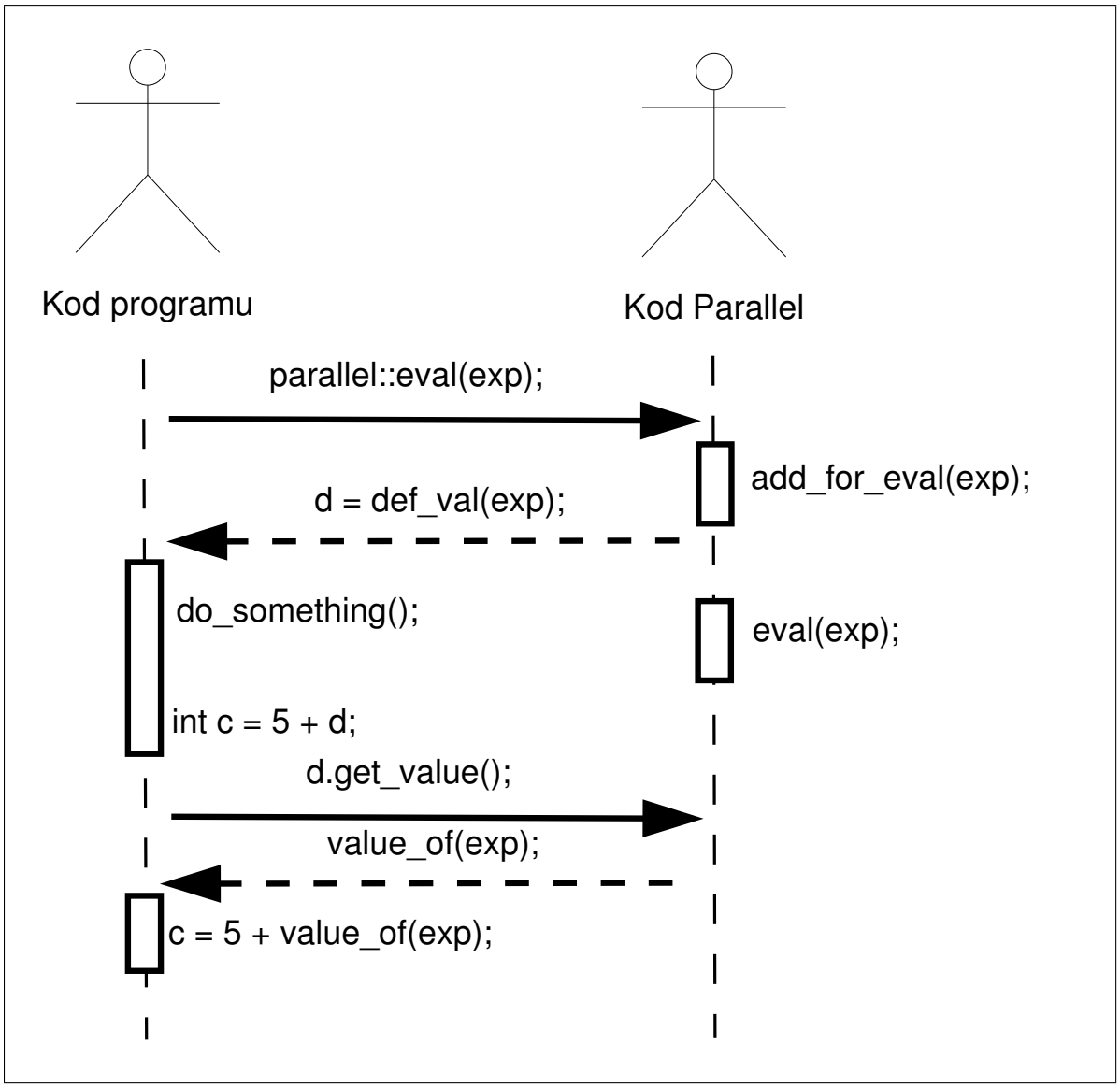
Nasuwa się zatem rozwiązanie, w którym biblioteka wyłapuje wyjątki wywołane w obliczanych wyrażeniach i przekazuje je do wątku głównego programu. Może to nastąpić jedynie za pośrednictwem wartości odroczonej, ponieważ powrót z funkcji `eval` następuje przed rozpoczęciem obliczania wyrażenia. Wartość odroczona ma ustalony typ, więc pobranie wartości nie może zwrócić wyjątku. Natomiast rozwiązaniem, na które się zdecydowano, jest ponowne rzucenie wyjątku przy próbie pobrania wartości lub wymuszenia jej wyliczenia. Programista używający Parallel powinien to mieć na uwadze.

1.2.5. Schemat interakcji kodu programu z biblioteką Parallel

Dokonam podsumowania opisu idei biblioteki Parallel poprzez opisanie pokrótce tego jak kod aplikacji pisany przez programistę współpracuje z kodem biblioteki Parallel. Schematycznie ilustruje to poniższy diagram.

Schemat pokazuje jak w pierwszej kolejności kod programu wywołuje funkcję `eval`, która służy do zlecenia obliczenia równoległego wyrażenia. Następnie sterowanie przechodzi do ciała funkcji `eval`, czyli do kodu biblioteki Parallel. Funkcja `eval` tworzy niezbędne struktury danych oraz dodaje wyrażenie `exp` do wyrażeń czekających na wyliczenie. Potem następuje powrót z funkcji `eval` ze zwróceniem wartości odroczonej `d` i na jakiś czas kod programu i biblioteki Parallel zaprzestają komunikacji. Program wykonuje własne instrukcje, a biblioteka Parallel wykonuje własne, między innymi wyliczając wyrażenia, które zostały przekazane do wyliczenia równoległego. Takie wyliczenie zachodzi w pewnym momencie równiez dla wyrażenia `exp`.

Gdy potrzebna jest wyliczona wartość wyrażenia `exp` następuje automatycznie próba pobrania wartości wyrażenia z wartości odroczonej `d`. Wtedy sterowanie przechodzi do kodu biblioteki, który przekazuje z powrotem wyliczoną wartość wyrażenia `exp`. Kod programu otrzymuje wartość i może dalej prowadzić swoje obliczenia. Tak wygląda jeden cykl skorzystania z mechanizmu zlecania wyrażeń do równoległego wyliczenia w bibliotece Parallel.



Rysunek 1.1: Schemat interakcji z biblioteką Parallel

1.3. Istniejące biblioteki do programowania równoległego w języku C++

W tej sekcji opis idei biblioteki Parallel zestanie przeciwstawiony przeglądowi obecnie istniejących bibliotek do programowania współbieżnego w jezyku C++. Powstało ich wiele, o różnych cechach, jednak żadna z nich nie realizuje zestawu celów, który postawiłem przed biblioteką Parallel. Dlatego, moim zdaniem, istniała potrzeba stworzenia nowej biblioteki, która różni się znacznie od istniejących rozwiazań i pokrywa inny zakres potrzeb programistów. Swój przegląd oprę na przeglądzie najpopularniejszych bibliotek do programowania równoległego w C++ dokonany na podstawie ich dokumentacji oraz publikacji naukowych.

1.3.1. Open Multi-Processing (OpenMP)

OpenMP został stworzony do pisania wielowątkowych programów dla systemów wieloprocesorowych z pamięcią dzieloną. Dzięki temu, że został uzgodniony przez największe firmy dostarczające oprogramowanie i sprzęt komputerowy wspiera wiele platform, takich jak Microsoft Windows, Unix, oraz wiele języków, na przykład C, C++, Fortran.

Nie jest typową biblioteką, gdyż oprócz pewnego zbioru funkcji udostępnia również zbiór dyrektyw kompilatora oraz zmiennych środowiskowych, które modyfikują działanie programu. Programowanie odbywa się w sposób jawny, to znaczy programista wyraźnie opisuje w kodzie jak powinno przebiegać równoległe wykonanie programu. Ten opis dokonywany jest w większości przez użycie odpowiednich dyrektyw kompilatora. Oczywiście nie wszystkie kompilatory zgodne ze standardem C++, wspierają OpenMP, to wsparcie musiało zostać specjalnie dołączone, tak aby kompilator rozumiał dyrektywy i kierowany tymi dyrektywami mógł wygenerować współbieżny kod.

Oto prosty przykład kodu napisanego przy pomocy OpenMP:

```
1  #include <omp.h>
2  #include <iostream>
3  int main (int argc, char *argv[]) {
4   int th_id, nthreads;
5  #pragma omp parallel private(th_id)
6   {
7    th_id = omp_get_thread_num();
8    std::cout << "Hello World from thread" << th_id << "\n";
9  #pragma omp barrier
10   if ( th_id == 0 ) {
11    nthreads = omp_get_num_threads();
12    std::cout << "There are " << nthreads << " threads\n";
13   }
14  }
15  return 0;
16 }
```

Program wypisuje komunikat “Hello World” z dołączonym numerem wątku. Zrównoleglenie tego kodu przez kompilator osiągnięto stosując dyrektywę `#pragma omp parallel private(th_id)`. Wynika z niej, że kompilator powinien zrównoleglić zaznaczony blok kodu, przy czym zmienna `th_id` ma być prywatna, czyli każdy wątek powinien posiadać swoją kopię.

OpenMP wykorzystuje model Fork-Join. W programie napisanym przy użyciu OpenMP występuje wątek głowny, który koordynuje pracę. Gdy wykonanie dochodzi do początku re-

gionu kodu, który został oznaczony odpowiednimi dyrektywami do zrównoleglenia, wtedy następuje faza fork, czyli tworzenia wątków. Każdy z wątków otrzymuje unikalny identyfikator, którego wartość można odczytać i przy jego pomocy sterować pracą tylko określonego wątku. Wątki przetwarzają kod przeznaczony do zrównoleglenia niezależnie od siebie, aczkolwiek istnieją mechanizmy pozwalające na podział zadań zdefiniowany przez programistę. Dzięki temu możliwe jest zrównoleglenie zarówno na poziomie zadań, różne wątki mogą wykonywać różny kod, jak i na poziomie danych, gdy wątki wykonują ten sam kod, ale na różnych danych. Na końcu wykonania kodu oznaczonego do zrównoleglenia następuje faza-join, w której wątek główny oczekuje na zakończenie pracy wszystkich wątków pracowników. Liczba wątków może być kontrolowana przez programistę za pomocą funkcji OpenMP lub zmiennych środowiskowych.

Kolejny przykład pokazuje jak w OpenMP można wykonać dodawanie dwóch tablic:

```
1  #include <omp.h>
2  #define CHUNKSIZE 100
3  #define N      1000
4
5  main ()
6  {
7
8  int i, chunk;
9  float a[N], b[N], c[N];
10
11 /* Some initializations */
12 for (i=0; i < N; i++)
13   a[i] = b[i] = i * 1.0;
14 chunk = CHUNKSIZE;
15
16 #pragma omp parallel shared(a,b,c,chunk) private(i)
17   {
18
19   #pragma omp for schedule(dynamic,chunk) nowait
20   for (i=0; i < N; i++)
21     c[i] = a[i] + b[i];
22
23   } /* end of parallel section */
24
25 }
```

W tym przykładzie następuje zrównowleglenie pętli for, w której sumowane są elementy dwóch tablic i wynik przypisywany jest na trzecią tablicę. Dyrektywa `#pragma omp for schedule(dynamic,chunk) nowait` mówi, iż pętla powinna zostać zrównoleglona, ale w taki sposób, że każdy z wątków zajmie się fragmentem tablicy o wielkości zapisanej w zmiennej `chunk`. Kolejne obroty pętli nie są ze sobą synchronizowane, o czym mówi słowo `nowait`.

Porównanie OpenMP vs. Parallel

Podobieństwa	Różnice
• Równoległość inkrementacyjna, możliwe jest dodawanie zrównoleglania obliczeń stopniowo, bez drastycznych zmian w kodzie. • Mała potrzeba zmian w kodzie przy zrównoleglaniu • OpenMP i Parallel działają tylko na platformach z pamięcią współdzieloną.	• W OpenMP dekompozycja zadań domyślnie jest dokonywana automatycznie, a w Parallel odpowiada za nią programista. • OpenMP nie jest zwykłą biblioteką języka i potrzebuje wsparcia kompilatora. • W przypadku OpenMP możliwa jest kompilacja do kodu sekwencyjnego bez żadnych modyfikacji w kodzie. • Parallel zostało zaprojektowane do zrównoleglania zadaniowego, a nie zrównoleglania na poziomie danych (mniejsza ziarnistość). W OpenMP oba te podejścia są wspierane. • OpenMP nie wspiera obsługi wyjątków, a biblioteka Parallel tak. • OpenMP pozwala na mniejszą dowolność synchronizacji równoległych fragmentów kodu. W Parallel wątki są synchronizowane, gdy jest to niezbędne.

1.3.2. Threading Building Blocks (TBB)

Threading Building Blocks jest biblioteką, która służy do pisania programów wykorzystujących wielowątkowość w języku C++. Biblioteka składa się z szablonów typów i algorytmów, które działają w sposób równoległy, jednocześnie pozwalają uniknąć trudności i złożoności związanych z programowaniem przy wykorzystaniu standardowych mechanizmów oferujących równoległość, takich jak wątki POSIX, Windows lub wątki z biblioteki Boost.Thread. W standardowej bibliotece oferującej wielowątkowość programista obciażony jest tworzeniem, usuwaniem lub synchronizacją wątków i przypisywaniem do nich zadań.

W przypadku TBB programista, zamiast definować działanie współbieżnego fragmentu programu manualnie, używa szkieletów algorytmów dostępnych w tej bibliotece. Następnie biblioteka sama dzieli wykonanie algorytmu na podzadania, przypisuje je do wątków, zajmuje się równoważeniem obciążenia procesorów i przypisywaniem wątków do procesorów w taki sposób, by zminimalizować migotanie pamięci podręcznej. Nawet optymalna liczba wątków jest dobierana automatycznie przez TBB zależnie od konfiguracji komputera.

Przykład wykorzystania biblioteki TBB do zrównoleglenia pętli for wygląda następująco:

```
1 #include "tbb/blocked_range.h"
2
3 class ApplyFoo {
```

```
4     float *const my_a;
5     public:
6       void operator()( const blocked_range<size_t>& r ) const {
7         for( size_t i=r.begin(); i!=r.end(); ++i ) {
8           Foo(my_a[i]);
9         }
10      }
11    ApplyFoo( float a[] ) :
12      my_a(a) {}
13  };
14
15  #define A_SIZE 1000
16  int main()
17  {
18    float a[A_SIZE];
19    parallel_for(blocked_range<size_t>(0,n,IdealGrainSize),
20      ApplyFoo(a) );
21  }
```

W tym przykładzie klasa ApplyFoo definiuje obiekt funkcyjny, który jeśli otrzyma za argument obiekt typu `blocked_range<size_t>` to przypisze na każdy element tablicy `my_a` wartość zwracaną przez wywołanie funkcji `Foo` od tego elementu. Niżej w funkcji `main` znajduje się wywołanie funkcji TBB `parallel_for`, która równolegle aplikuje obiekt funkcyjny `ApplyFoo(a)` do fragmentów tablicy `a` wielkości `IdealGrainSize`.

TBB zawiera, oprócz schematu pętli for, również wiele innych: schemat pętli while, schemat pipeline, schemat reduce.

Porównanie TBB vs. Parallel

Podobieństwa	Różnice
• Równoległość inkrementacyjna, możliwe jest dodawanie zrównoleglania obliczeń stopniowo, bez drastycznych zmian w kodzie. • TBB i Parallel działają tylko na platformach z pamięcią współdzieloną i nie jest możliwe przeskalowanie programu na wiele maszyn. • Obie biblioteki zostały zaprojektowane do zrównoleglania kodu na poziomie zadań do wykonania. • Obie biblioteki wspierają obsługę wyjątków.	• Użycie TBB zazwyczaj wymaga zmian w kodzie. Choć jego struktura pozostaje w większości niezmieniona, to niezbędne jest zdefinowanie klas - obiektów funkcyjnych przekazywanych do algorytmów z TBB. • Kod TBB jest mniej czytelny, gdyż to co się dzieje w programie opisane jest w miejscu wywołującym równoległe wykonanie, jak i przez obiekty funkcyjne przekazywane do TBB zdefinowane w innym miejscu w kodzie. • W TBB dekompozycja zadań domyślnie jest dokonywana automatycznie. • Kod pisany przy użyciu TBB zazwyczaj jest dłuższy, ze względu na konieczność definowania dodatkowych klas.

1.3.3. Message Passing Interface (MPI)

Message Passing Interface jest biblioteką nieco odmienną od poprzednio opisanych, gdyż pozwala na pisanie programów równoległych na systemy komputerowe z pamięcią rozproszoną. Jest to możliwe dzięki komunikacji poprzez wiadomości, którą zapewnia biblioteka. W typowym przypadku program równoległy składa się z wielu procesów komunikujących się poprzez wywołania odpowiednich funkcji MPI do wysyłania lub odbierania wiadomości. MPI zostało ustandaryzowane i jest dostępne w wielu językach i na wielu platformach.

MPI jest uważane za stosunkowo niskopoziomowy sposób pisania programów równoległych. Programista ustala liczbę procesów, kierunki komunikacji, mechanizmy synchronizacji, podział danych i ich rozkład pomiędzy procesy oraz przydział procesów do procesorów. Do korzyści z programowania przy pomocy MPI można zaliczyć wydajność tej biblioteki oraz łatwość z jaką można skalować programy na większą liczbę procesorów. Ponadto dzięki szerokiemu wsparciu MPI przez największych dostawców sprzętu i oprogramowania, programy pisane przy pomocy MPI łatwo przenosi się na inne platformy.

Oto przykład programu napisanego przy pomocy MPI, który pokazuje podstawowe operacje związane z wysyłaniem i odbieraniem komunikatów.

```
1 /*
2  * "Hello World" MPI Test Program
3  */
4 #include <mpi.h>
5 #include <stdio.h>
6 #include <string.h>
7
```

```
 8 #define BUFSIZE 128
 9 #define TAG 0
10
11 int main(int argc, char *argv[])
12 {
13  char idstr[32];
14  char buff[BUFSIZE];
15  int numprocs;
16  int myid;
17  int i;
18  MPI_Status stat;
19
20  /* kazdy program MPI musi najpierw wywolac MPI_Init */
21  MPI_Init(&argc,&argv);
22  /* sprawdzenie ile jest procesow w grupie */
23  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
24  /* sprawdzenie numeru danego procesu w grupie */
25  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
26
27  if(myid == 0)
28  {
29    printf("%d: We have %d processors\n", myid, numprocs);
30    for(i=1;i<numprocs;i++)
31    {
32      sprintf(buff, "Hello %d! ", i);
33      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
34    }
35    for(i=1;i<numprocs;i++)
36    {
37      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
38       MPI_COMM_WORLD, &stat);
39      printf("%d: %s\n", myid, buff);
40    }
41  }
42  else
43  {
44    /* odebranie wiadomosci od procesu o identyfikatorze 0 */
45    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG,
46     MPI_COMM_WORLD, &stat);
47    sprintf(idstr, "Processor %d ", myid);
48    strncat(buff, idstr, BUFSIZE-1);
49    strncat(buff, "reporting for duty\n", BUFSIZE-1);
50    /* wysłanie wiadomości do procesu z identyfikatorem 0 */
51    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
52  }
53
54  /* Program MPI powinien zakonczyć się wywołaniem MPI_Finalize,
55   * ktory jest dla procesow punktem synchronizacji.
56   */
```

```
57   MPI_Finalize();
58   return 0;
59 }
```

W tym programie wątek główny o identyfikatorze 0 wysyła wiadomość "Hello" do każdego z wątków, a następnie oczekuje na odpowiedź od wszystkich wątków. Wątki pozostałe czekają na wiadomość, a następnie wysyłają odpowiedź.

W kodzie można zauważyć, że MPI udostępnia niskopoziomowe API. Operacje wysyłania i odbierania wiadomości działają analogicznie do funkcji systemowych read i write, operują na poziomie bitów. Wiąże się to z narzuceniem na programistę obowiązku zadbania o odpowiednie zakodowanie i rozkodowanie wiadomości.

Porównanie MPI vs. Parallel

Podobieństwa	**Różnice**
• Obie biblioteki zostały zaprojektowane do zrównoleglania kodu na poziomie zadań do wykonania.	• W MPI trudno jest stopniowo zrównoleglać program. Fragmenty programu, które mają działać równoległe muszą zostać zaimplementowane w całości i ze sobą współgrać. • MPI wymaga znacznych strukturalnych zmian w kodzie. • Ze względu na to, że MPI koordynuje pracę wielu niezależnych procesów, które komunikują się asynchronicznie trudno jest dokładnie prześledzić i zrozumieć działanie takiego programu. • MPI działa także w systemach komputerowych z pamięcią rozproszoną, Parallel tylko w środowisku z pamięcią dzieloną. • W MPI całość komunikacji, synchronizacji, podział zadań, zbieranie wyników musi zostać zapisane *explicite* przez programistę. • Komunikacja w MPI odbywa się pomiędzy różnymi procesami, a w Parallel pomiędzy wątkami o wspólnej przestrzeni adresowej. MPI można stosować w ogólniejszych przypadkach.

1.3.4. Boost Thread

Istnieje kilka bibliotek oferujących programiście możliwość uruchamiania wielu watków w ramach jednego programu. Wśród nich można wymienić POSIX Threads, Windows Threads,

najbardziej typowe rozwiązania dla platform odpowiednio Unix i Windows. Do opisu wybrano jednak Boost Thread ze względu na przenaszalność, natomiast zasada działania i oferowane możliwości są podobne we wszystkich tego typu bibliotekach.

Biblioteka Boost Thread umożliwia zarządzanie wątkami, jak i udostępnia typy oraz funkcje służące do synchronizacji pomiędzy wątkami. Mechanizmy synchronizacyjne dostępne w Boost Thread to między innymi różnego rodzaju blokady, zmienne warunkowe, bariery. Boost Thread pozwala również na tworzenie grup wątków, którymi można zarządzać, ale nie ma funkcji puli wątków, która optymalizowałaby zużycie zasobów przy posługiwaniu się wątkami. Komunikacja w programie pisanym przy użyciu Boost Thread odbywa się zazwyczaj przez współdzielone struktury danych, o ochronę których programista musi zatroszczyć się samodzielnie.

Poniżej znajduje się przykład prezentujący użycie biblioteki Boost Thread:

```
1
2  #include <boost/thread.hpp>
3  #include <boost/bind.hpp>
4
5  #include <vector>
6  #include <iostream>
7
8  void hello_function(int n)
9  {
10   std::cout << "Hello from thread nr " << n << std::endl;
11 }
12
13 const size_t threads_num = 20;
14
15 int main()
16 {
17   std::vector<boost::thread> vt(threads_num);
18
19   for (int i = 0; i < threads_num; i++)
20   {
21     vt[i] = thread(boost::bind(hello_function, i));
22   }
23   for (int i = 0; i < threads_num; i++)
24     vt.[i].join();
25 }
```

Przykład pokazuje w jaki sposób tworzy się wątki i przydziela im zadanie do wykonania, każdemu z osobna. Następnie wątek główny oczekuje na zakończenie działania każdego z wątków wywołując funkcję `join`.

Porównanie Boost Threads vs. Parallel

Podobieństwa	Różnice
• Boost Thread i Parallel działają tylko na platformach z pamięcią współdzieloną i nie jest możliwe przeskalowanie programu na wiele maszyn. • Obie biblioteki zostały zaprojektowane do zrównoleglania kodu na poziomie zadań do wykonania. • W przypadku użycia obu bibliotek programista musi zadbać o dekompozycje zadań.	• Stopniowe dodawanie równoległości w przypadku użycia Boost Thread jest trudniejsze, ponieważ użycie biblioteki wymaga zmian w strukturze kodu i dodania mechanizmu komunikacji między wątkami. • Wątki z Boost Thread przyjmują do wykonania jedynie obiekty funkcyjne, co wiąże się z koniecznością dodania do kodu definicji tych obiektów. • Kod Boost Thread jest mniej czytelny, gdyż składnia obiektów funkcyjnych jest mniej czytelna niż składnia wyrażeń. • Chcąc przekazać wartość pomiędzy wątkami Boost Thread trzeba skorzystać z dodatkowej synchronizacji i globalnych zmiennych lub wskaźników obecnych w pamięci współdzielonej. • W Parallel zadania są przydzielane wątkom dynamicznie, dzieki czemu równoważone jest obciążenie wątków. Natomiast programista korzystający z Boost Thread odpowiada za jawny przydział zadań dla każdego wątku. • Programista używający Parallel jest w znacznym stopniu odciążony z używania mechanizmów synchronizacji i komunikacji pomiędzy wątkami. • W Boost Thread wyjątki wywołane w kodzie wykonywanym przez wątek nie są sygnalizowane wątkowi głównemu, który uruchomił dany wątek.

1.4. Zaawansowane przykłady użycia biblioteki Parallel

Opis koncepcji biblioteki Parallel zakończę pokazaniem zaawansowanych przykładów wykorzystania biblioteki Parallel do prowadzenia równoległych obliczeń. W pierwszym z przykładów pokaże w jaki sposób biblioteka Parellel może służyć do zrównoleglenia algorytmu szybkiego sortowania (z ang. quicksort). Drugi przykład wykorzystujący w jeszcze większej mierze możliwości biblioteki Parallel pokaże zdecydowaną przewagę biblioteki Parallel nad standardową biblioteką do obsługi wielu wątków Boost.Threads.

1.4.1. Równoległe szybkie sortowanie

Moja implementacja zrównoleglonego szybkiego sortowania opiera się na prostej idei "dziel i zwyciężaj". W każdym rekurencyjnym wywołaniu funkcji `parallel_quicksort` na początku wywoływana jest funkcja `partition`, dzieląca tablicę na części o elementach mniejszych od pewnego wyróżnionego elementu i o elementach większych od tegoż elementu. Następnie, jeśli podtablice do posortowania są wystarczająco duże (w przykładzie limit został ustalony na 100 elementów) to wywoływana jest rekurencyjnie funkcja `parallel_quicksort`. Jeśli podtablica jest mniejsza to jest sortowana standardowym algorytmem sekwencyjnym szybkiego sortowania.

Implementacje dostarczyłem w dwóch wersjach. Jedna jest zaimplementowana przy użyciu biblioteki Boost.Threads, a druga przy użyciu biblioteki Parallel. Oto implementacja wykorzystująca standardową bibliotekę do obsługi wielowątkowości:

```
1  const unsigned limit = 100;
2
3  template <typename Item>
4  void thread_quicksort(Item array[], size_t l, size_t r)
5  {
6    if (l < r)
7    {
8      size_t s = partition(array, l, r);
9      boost::thread thread;
10     if (s - l > limit)
11       thread = boost::thread(
12         boost::bind(quicksort<int>, array, l, s - 1));
13     else
14       quicksort(array, l, s - 1);
15     quicksort(array, s + 1, r);
16     if (s - l > limit)
17       thread.join();
18   }
19 }
```

Jak widzimy dla każdego równoległego wywołania funkcji `paralell_quicksort` musi zostać stworzony wątek, ponieważ Boost.Threads nie posiada puli wątków. [1] W przypadu dużych tablic może to skutkować spowolnieniem działania programu z powodu zbyt dużego obciążenia systemu wątkami i zbyt częstych przełączeń kontekstu.

[1] Oczywiście pulę wątków można zaimplementować wykorzystując Boost.Threads, ale moim celem było pokazanie przykładów zrównoleglenia szybkiego sortowania, których napisanie było podobne pod względem struktury kodu i pracochłonności.

W poniższy sposób wygląda analogiczna implementacja szybkiego sortowania korzystająca z biblioteki Parallel:

```
1  const unsigned limit = 100;
2
3  template <typename Item>
4  void parallel_quicksort(Item array[], size_t l, size_t r)
5  {
6    if (l < r)
7    {
8      size_t s = partition(array, l, r);
9      deferred_value<void> tmp;
10     if (s - l > limit)
11       tmp = parallel::eval(
12         parallel::lazyf(quicksort<int>, array, l, s - 1));
13     quicksort(array, s + 1, r);
14     if (s - l > limit)
15       tmp.force();
16   }
17 }
```

Ta implementacja ma tę przewagę, że wątki są tworzone tylko raz, podczas inicjalizacji biblioteki Parallel i ich ilość jest ustalona. Zatem nie jest możliwe przeciążenie systemu niekontrolowanym rozmnożeniem się wątków. Poza istotną korzyścią związaną z brakiem potrzeby zarządzania wątkami, w samej strukturze programu nie widać znaczących korzyści z wyboru biblioteki Parallel. Kod wygląda bardzo podobnie. Aby pokazać zalety Parallel w porównaniu ze standardowymi bibliotekami będzie potrzebny kolejny przykład.

1.4.2. Równoległe sumowanie elementów tablicy

Przykład zrównoleglenia sumowania elementów tablicy pokaże jak wygodna jest składnia bibioteki Parallel w porównianiu ze składnią biblioteki Boost.Threads, gdy równoległe obliczenia muszą zwrócić pewną wartość.

Tak wygląda implementacja wielowątkowa napisana w oparciu o Boost.Threads:

```
1  void sum(int* wynik, int* begin, int* end)
2  {
3    *wynik = 0;
4    for (int* i = begin; i != end; i++) *wynik += *i;
5  }
6
7  int array_sum_thread(int* array, size_t size)
8  {
9    int s1, s2, s3, s4;
10   boost::thread t1(
11     boost::bind(sum, &s1, array, array + size/4));
12   boost::thread t2(
13     boost::bind(sum, &s2, array + size/4 + 1, array + size/2));
14   boost::thread t3(
15     boost::bind(sum, &s3, array + size/2 + 1, 3*size/4));
```

```
16    boost::thread t4(
17      boost::bind(sum, &s4, array + 3*size/4 + 1, array + size));
18    t1.join();
19    t2.join();
20    t3.join();
21    t4.join();
22    return s1 + s2 + s3 + s4;
23  }
```

W tej implementacji niezbędne było zaimplementowanie funkcji sumującej pewien odcinek tablicy, który zwraca wynik przez jeden z argumentów, poniważ wątki nie zwracają żadnych wartości. Oczywiście, w przypadku tak prostej implementacji nie ma możliwości żadnej obsługi błędów, które mogą pojawić się podczas obliczeń. Ponadto programista musiał zadbać o synchronizację wątków, w tym przypadku poprzez wywołanie funkcji `join` na każdym z wątków.

Nieco inaczej wygląda funkcja napisana przy pomocy biblioteki Parallel:

```
1   int sum(int* begin, int* end)
2   {
3     int s = 0;
4     for (int* i = begin; i != end; i++) s += *i;
5     return s;
6   }
7
8   int array_sum_parallel(int* array, size_t size)
9   {
10    deferred_value<in> sum = parallel::evaluate(
11      *parallel::lazyf(sum, array, array + size/4));
12    for (int i = 1; i < 4; i++)
13    {
14      sum += parallel::evaluate(parallel::lazyf(
15        sum, array + (i * size)/4 + 1, array + ((i+1) * size)/4));
16    }
17    return sum;
18  }
```

Funkcja sumująca tablicę w tym przykładzie ma częściej spotykaną postać, w której wynik zwracany jest w standardowy sposób. W związku z tym często w praktycznych zastosowaniach będzie można zastosować istniejącą funkcję, co zwiększy częstość ponownego użycia kodu. W przykładzie nie widać żadnego śladu jawnej synchronizacji pomiędzy wątkami. Dzięki odpowiedniej implementacji wartości odroczonych synchronizacja zarządzana jest automatycznie przez bibliotekę. Pomimo tego, że w linii 13. przykładu sum jest używane to przeładowanie operatora `+=` sprawia, że obliczenie wartości odroczonej nie jest wymuszane i wszystkie obliczenia wykonują się równocześnie. Dopiero w linii 15, gdzie wartość odroczona `sum` jest konwertowana do `int` następuje pobranie wyniku ze wszystkich zleconych obliczeń i zwrócenie wartości.

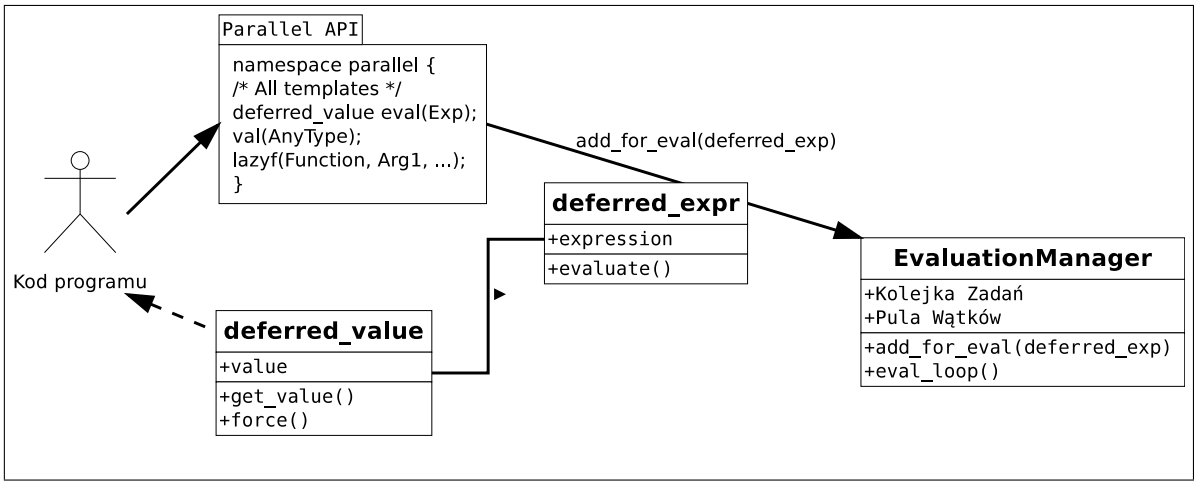
Rozdział 2

Opis implementacji

W tym rozdziale zostaną opisane szczegóły implementacyjne biblioteki Parallel. Nie jest moim celem pełne opisanie całości kodu, które byłoby bardzo obszerne i męczące dla czytelnika. Swoją uwagę skoncentruję na najciekawszych aspektach implementacji biblioteki, które niosły ze sobą pewne problemy do rozwiązania i wywarły największy wpływ na wynik finalny.

2.1. Architektura biblioteki

Poniżej znajduje się diagram pokazujący główne komponenty biblioteki wraz z zależnościami między nimi. Podczas omawiania implementacji biblioteki Parallel będę odwoływał się do elementów przedstawionego schematu. Pozwoli to na lepsze zrozumienie późniejszych rozważań.



Rysunek 2.1: Schemat architektury biblioteki Parallel

Na diagramie nie znajdują się wszystkie metody zaimplementowane w poszczególnych klasach, a jedynie te niezbędne do zaprezentowania architektury biblioteki.

2.1.1. API biblioteki

Kod programu komunikuje się z biblioteką poprzez API. Oto krótkie wyjaśnienie znaczenia poszczególnych funkcji z API: Główną funkcją z API jest funkcja `eval`, której znaczenie zostało opisane w sekcji Funkcja `eval`. Pozostałe funkcje mają zadanie pomocnicze w stosunku do `eval`. Są niezbędne do stworzenia wyrażenia, które funkcja `eval` może przyjąć do wyliczenia. I tak funkcja `val` pozwala przekazać swój argument do wyrażenia przez wartość,

`cval` przez wartość stałą, `ref` przez referencję, `cref` przez stałą referencję. Ponadto, w API znajduje sie wzorzec funkcji pozwalający na uleniwione wywoływanie funkcji, `lazyf`. Rola i sposób implementacji tych funkcji są opisane w sekcji Implementacja przekazywania wyrażeń do wyliczenia.

Należy dodać, iż program korzystający z biblioteki Parallel musi przed rozpoczęciem zlecania wyrażeń do obliczenia wywołać funkcję `init` inicjalizującą bibliotekę i ustalającą liczbę wątków w puli mechanizmu ewaluacji. Korzystanie z biblioteki Parallel powinno zakończyć się wywołaniem funkcji `close`, która zwalnia zasoby zaalokowane przez bibliotekę.

2.1.2. Operacje wykonywane w kodzie biblioteki Parallel

Odpowiednio stworzone wyrażenie jest przetwarzane przez funkcję `eval`. W ciele funkcji z wyrażenia tworzony jest obiekt typu `deferred_expr`, który reprezentuje zadanie do wykonania. Zadanie jest dodawane do kolejki zadań w obiekcie `evaluation_mgr` przy użyciu metody `add_for_eval`. Aparat wykonawczy w postaci instancji typu `evaluation_mgr` zawiera, oprócz kolejki zadań do wykonania, również wątki, którymi zarządza, i które wykonują zadania działając w nieskończonej pętli w funkcji `eval_loop`.

Każdy obiekt typu `deferred_expr` jest połączony 1-1 z obiektem typu `deferred_value`, który reprezentuje wynik obliczenia wyrażenia. Jeśli zadanie zdefinowane przez wyrażenie zapamiętane w obiekcie `deferred_expr` zostaje wykonane, to wynik (jeśli istnieje) zostaje przypisany na pole `m_value` w skojarzonym obiekcie typu `deferred_value`.

2.1.3. Zwrócenie wyniku

Wynik jest przekazywany do kodu programu poprzez obiekt typu `deferred_value`, który jest zwracany jako wynik funkcji `eval`. Jeśli nastąpi wywołanie pobrania wartości z obiektu typu `deferred_value` metodą `get_value` lub wymuszenie wyliczenia wartości przez metodę `force`, wyliczenie wyrażenia zostaje wymuszone, o ile nie zostało już wykonane.

2.2. Implementacja przekazywania wyrażeń do wyliczenia

Jednym z najtrudniejszych zadań podczas fazy implementacji biblioteki Parallel było zaprojektowanie przekazywania wyrażeń do wykonania. Mechanizm miał stanowić ważną część API biblioteki, które powinno wspierać spełnienie wyznaczonych dla biblioteki celów czytelności i intuicyjności. Te cele niewątpliwie byłyby zrealizowane, gdyby możliwe było przekazywanie wyrażeń w ich standardowej postaci w języku C++. Jednak problem stanowił fakt, iż język C++ posiada gorliwą semantykę wyliczania wyrażeń. Należało zatem uleniwić wyrażenia przekazywane do wyliczenia.

Najbardziej naturalne byłoby użycie słowa kluczowego lub funkcji, która uleniwiałaby wyrażenie. Pierwszy pomysł był nie do zrealizowania, ponieważ nie można rozszerzyć języka C++ o nowe słowa kluczowe, a standard C++ nie przewidział słowa kluczowego dla uleniwiania wyrażeń. Uleniwienie poprzez użycie funkcji mogłoby wyglądać następująco:

```
make_lazy_expression(4 + fibonacci(20));
```

Jednak semantyka języka C++ również nie pozwalała na taką realizację uleniwiania wyrażeń, gdyż wyrażenie podane jako argument jest wyliczane przed wywołaniem funkcji.

Metoda rozwiązania problemu uleniwiania wyrażeń okazała się być bardziej skomplikowana i została zainspirowana przez idiom C++ szablonu wyrażenia, który jako jedna z dobrych

praktyk języka C++ został opisany w książce *More C++ Idioms* [Idioms]. Implementacja tego szablonu nie byłaby możliwa gdyby nie bardzo silny mechanizm szablonów typów i funkcji w języku C++.

2.2.1. Szablony w języku C++

Ta sekcja nie zawiera podstawowych informacji na temat szablonów w C++. Wprowadzenie do tej tematyki mozna przeczytać w książce [C++Lang]. Moim celem jest pokazanie, dlaczego implementacja przekazywania wyrażeń, tak jak tego dokonałem, była możliwa.

Szablony są bardzo silnym mechanizmem. Ciekawostką jest fakt, że siła ekspresyjna szablonów nie była zamierzona przez twórców w momencie projektowania i możliwości, które dają, zostały odkryte później. Szablony C++ są w rzeczywistości pewnym funkcyjnym językiem programowania o ekspresywności rachunku lambda. Udowodniono, iż mechanizm szablonów jest równoważny maszynie Turinga ([TuringCom]). W praktyce oznacza to tyle, że szablony C++ pozwalają dokonywać dowolnych obliczeń w oparciu o wzorce typów. Proszę spojrzeć na poniższy przykład:

```
1 template< unsigned n >
2 struct factorial {
3   static const unsigned value =
4     n * factorial<n-1>::value;};
5 template<>
6 struct factorial<0> {
7   static const unsigned value = 1; };
```

Wzorzec `factorial` pozwala obliczyć dowolne wartości silni w czasie kompilacji [1] i korzystać z nich w czasie stałym podczas wykonania programu. Obliczenie odbywa się rekurencyjnie, warunkiem kończącym rekurencję jest `factorial<0>`, które jest specjalizacją szablonu `factorial`.

2.2.2. Idiom C++ szablonu wyrażenia

Intuicyjnie rzecz ujmując, idiom szablonu wyrażenia polega na opisaniu wyrażenia przy pomocy konkretyzacji pewnego szablonu typu. Ten typ nie jest nigdy zapisywany w postaci jawnej w programie, gdyż można przypuszczać, że składnia szablonu typu byłaby wtedy bardzo nieczytelna. Zamiast tego szablon typu jest tworzony w wyniku obliczeń na typach udostępnianych przez mechanizm rozwijania szablonów w języku C++. Idiom szablonu wyrażenia wykorzystuje inny idiom Rekurencyjnego Składania Typów (z ang. Recursive Type Composition). Polega on na tym, że szablon typu zawiera pola, których typem jest jest konkretyzjacja tego samego szablonu typów. W tym przypadku szablon wyrażenia może zawierać pola, które są szablonem wyrażenia. W ten sposób powstaje reprezentacja wyrażenie w postaci drzewa typów. Taka konstrukcja nazywana jest Abstrakcyjnym Drzewem Syntaktycznym (z ang. Abstract Syntax Tree - AST). W liściach drzewa znajdują się terminale wyrażenia, czyli wartości, referencje bądź wskaźniki. Natomiast w węzłach pośrednich reprezentowane są operatory użyte w wyrażeniu, które jako pola posiadają szablony wyrażenia, będące argumentami operatora. Każdemu operatorowi odpowiada reprezentujący go szablon typu, z odpowiednio zaimplementowaną funkcją aplikującą operator, w momencie, gdy szablon wyrażenia jest wyliczany.

[1] Pewnym ograniczeniem jest limit kompilatora na rekurencyjne konkretyzowanie szablonów, ale ten limit można zmienić przy pomocy odpowiednich opcji

Zaprezentuję ideę szablonu wyrażenia na przykładzie, w którym jedyną dozwoloną operacją będzie dodawanie liczb całkowitych lub zmiennoprzecinkowych.

```
1  /*** TYPY ODPOWIEDZIALNE ZA BUDOWĘ DRZEWA AST ***/
2
3  template <typename T> struct Exp;
4
5  template <typename T>
6  struct Term : Exp<Term<T> >
7  {
8    Term(T v) : val(v){}
9    const T val;
10 };
11
12 /* Definicje wezłów reprezentujących operatory */
13
14 template <typename T1, typename T2>
15 struct Sum : Exp<Sum<T1, T2> >
16 {
17   Sum (T1 l, T2 r) : lhs(l), rhs(r) {}
18   const T1 lhs;
19   const T2 rhs;
20 };
21
22 template <typename T1, typename T2>
23 struct Mul: Exp<Mul<T1, T2> >
24 {
25   Mul(T1 l, T2 r) : lhs(l), rhs(r) {}
26   const T1 lhs;
27   const T2 rhs;
28 };
29
30 /* Wzorzec Infer służy do obliczania typu wynikowego wyrażenia */
31
32 template <typename E>
33 struct Infer {
34   typedef E type ;
35 };
36
37 template <typename T>
38 struct Infer<Term<T> >
39 {
40   typedef T type;
41 };
42
43 /* Wnioskowanie dla sumy zgodnie z semantyką języka C++ */
44
45 template < >
46 struct Infer< Sum<double,double> >
```

```
47 {
48   typedef double type;
49 };
50
51 template < >
52 struct Infer< Sum<double,int> >
53 {
54   typedef double type;
55 };
56
57 template < >
58 struct Infer< Sum<int,double> >
59 {
60   typedef double type;
61 };
62
63 template < >
64 struct Infer< Sum<int,int> >
65 {
66   typedef int    type;
67 };
68
69 /* Typ(T1 + T2) to typ sumy(typ(T1),typ(T2)) */
70 template <typename T1, typename T2>
71 struct Infer < Sum<T1,T2> > {
72   typename Infer<T1>::type typedef T1T ;
73   typename Infer<T2>::type typedef T2T ;
74   typename Infer<Sum<T1T,T2T> >::type typedef type;
75 } ;
76
77 /* Wnioskowanie o typach dla wzorca Mul jest analogiczne */
78
79 /* Wzorzec Exp umożliwia uniknięcie definicji operatorów
80  * w każdym typie reprezentujacym wezeł pośredni w drzewie AST */
81
82 template <typename T>
83 struct Exp
84 {
85   template <typename U>
86   inline Sum<T,U> operator + (U u)
87   {
88     /* static_cast jest potrzebny i prawidłowy,
89      * ponieważ Exp jedynie obudowuje typ T */
90     return Sum<T,U>(static_cast<T>(*this), u);
91   }
92
93   template <typename U>
94   inline Mul<T,U> operator * (U u)
95   {
```

```
96      return Mul<T,U>(static_cast<T>(*this), u);
97    }
98  };
99
100 template <typename T, typename U>
101 Sum<T,U> operator + (T t, Exp<U> u)
102 {
103   return Sum<T,U>(t, static_cast<U>(u));
104 }
105
106 template <typename T, typename U>
107 Mul<T,U> operator * (T t, Exp<U> u)
108 {
109   return Mul<T,U>(t, static_cast<U>(u));
110 }
111
112 /*** DEFINICJE FUNKCJI EWALUUJĄCYCH DRZEWA AST ***/
113
114 inline template <typename T>
115 Infer<T>::type eval(Exp<T> exp)
116 {
117   return eval(static_cast<T>(exp));
118 }
119
120 inline template <typename T>
121 Infer<Term<T> >::type eval(Term<T> term)
122 {
123   return term.val;
124 }
125
126 inline template <typename T1, typename T2>
127 Infer<Sum<T1, T2> >::type eval(Sum<T1, T2> sum)
128 {
129   return eval(sum.lhs) + eval(sum.rhs);
130 }
131
132 inline template <typename T1, typename T2>
133 Infer<Mul<T1, T2> >::type eval(Mul<T1, T2> mul)
134 {
135   return eval(mul.lhs) + eval(sum.rhs);
136 }
137
138 /* Potrzebna jest także definicja int eval (int)
139  * oraz double eval(double), ponieważ nie ma wymogu,
140  * aby zawsze te typy proste były obudowane w Term. */
141
142 inline int eval(int i) {return i;}
143 inline double eval(double d) {return d;}
```

Wzorzec `Term` (linie 6-11) umożliwia oznaczenie dowolnej wartości jako liścia w drzewie reprezentujacym szablon wyrażenia. Teoretycznie wszystkie wartości powinny byc oznaczone w ten sposób. Jednak jak sie później dowiemy identyczne działanie można uzyskać oznaczając tylko niektóre z wartości.

Wzorce `Sum` oraz `Mul` reprezentują wezły pośrednie w drzewie AST (linie 15-29). Ich pierwszy i drugi parametr reprezentują odpowiednie podwyrażenia, będące argumentami operatora binarnego. Najciekawszym zabiegiem użytym w ich definicji jest podanie definicji tegoż samego wzorca jako argumentu wzorca Exp, będącego klasą bazową wzorców operatorów. Ma to miejsce przykładowo w tym fragmencie kodu: `struct Sum : Exp<Sum<T1, T2> >` i jest w pełni dozwolone przez składnię języka. Dzięki takiemu zabiegowi można podać implementację operatorów tylko we wzorcu `Exp`, a nie jest to konieczne we wszystkich innych wzorcach węzłów pośrednich. `Sum<T1, T2>` dziedziczy metody przedefiniujące operatory, z tym, że te definicje są dopasowane do `Sum<T1, T2>`, ponieważ są parametryzowane tym typem.

Kolejna część przykładu (linie 33-78) to definicja wzorca `Infer`, który oblicza typ wynikowy wyliczenia wyrażenia, będzie on potrzebny do definicji funkcji ewaluujących wyrażenie. Wykorzystywany jest tutaj mechanizm specjalizacji wzorców, który pozwala określić typ wynikowy wyrażenia w zależności od postaci wzorca. Ogólnie, podczas definicji wnioskowania o typach należy pamiętać o uwzględnieniu wszystkich istotnych specjalizacji wzorca, czyli dla węzłów pośrednich jak `Term`, `Sum`, `Mul` oraz wszystkich możliwych typów, które mogą wystąpić w węzłach pośrednich. Jak widać implementacja szablonu wzorca wnioskującego o typie wynikowym wyrażenia jest bardzo długa i czasochłonna (do tego stopnia, że pominąłem część poświęconą wnioskowaniu typu dla wzorca `Mul`, gdyż jest ona analogiczna jak część dotycząca wzorca `Sum`. Na koniec sekcji kodu, która tworzy AST (linie 83-99) znajduje się definicja wzorca `Exp`, który przeciąża operatory dla szablonu wyrażenia.

Kolejna sekcja kodu przykładu (linie od 103 do 132) definiuje funkcję `eval`, która służy do obliczania wartości szablonu wyrażenia. Funkcja `eval` przyjmuje jako argument szablon wyrażenia (albo wartość prostą int lub double), a następnie oblicza jego wartość zgodnie z semantyką języka C++. Typ zwracany przez `eval` jest obliczany przy pomocy wzorca `Infer`.

Szablon wyrażenia w działaniu

Zrozumienie kodu szablonu wyrażenia przedstawionego powyżej będzie łatwiejsze po przeanalizowaniu przykładów jego działania. Weźmy najprostszy możliwy przykład:

```
Term(4) + Term(5);
```

Wyrażenie dodaje dwie liczby całkowitych. Szablon wyrażenia, który je reprezentuje, zostanie stworzony w wyniku obliczeń na typach i będzie wyglądał następująco:

```
Sum<Term<int>,Term<int> >
```

Już na bazie tego prostego przykładu nasuwa się kilka ciekawych kwestii do omówienia.

Po pierwsze można się zastanowić, po co składania jest tak "przegadana" oraz czy potrzebne jest pisanie `Term` przy każdej wartości przekazywanej do wyrażenia. Wiadomo na pewno, że co najmniej jedna wartość musi być oznaczona jako terminal szablonu wyrażenia. W przeciwnym przypadku, byłoby to zwykłe wyrażenie C++, żadna z wartości nie zostałaby uleniwiona i zostałoby obliczone gorliwie.

Zastanówmy się, czy poniższe wyrażenie jest poprawne składniowo i jeśli tak, to jaki będzie szablon typu, który powstanie.

```
Term(4) + 5;
```

W tym wyrażeniu mamy do czynienia z operacją dodawania wartości typu `Term<int>` oraz typu `int`. Ponieważ `Term<int>` dziedziczy operator dodawania z `Exp<Term<int> >` to zostanie wywołany operator dodawania zdefinowany w linii 87 przykładu. Warto zauważyć, że typ prawego argumentu przeciążonego operatora nie jest w żaden sposób ograniczony, więc zastosowanie typu int jest poprawne. W związku z tym wynikiem budowy drzewa tego wyrażenia będzie `Sum<Term<int>, int>`. Pokazuje to, że szablon wyrażenia zostanie zbudowany poprawnie również wtedy, gdy nie wszystkie wartości zostaną oznaczone jako terminale. Jednakże, część wartości, a co najmniej jedna musi zostać oznaczona, gdyż inaczej budowa szablonu wyrażenia nie zostanie zapoczątkowana i wyrażenie nie zostanie uleniwione, a obliczy się normalnie.

Ważnym faktem jest to, że budowa drzewa AST ma miejsce w czasie kompilacji, więc budowanie szablonów wyrażeń nie nakłada narzutów na czas wykonania programu. Natomiast narzuty podczas ewaluacji uleniwionego wyrażenia zależą od tego w jakim stopniu kompilator zoptymalizował wywołania funkcji `eval` wyliczającej wartość wyrażenia poprzez jej rekurencyjne rozwinięcie w miejscu wykonania. W najgorszym przypadku będzie miało miejsce jedno wywołanie funkcji dla każdego węzła pośredniego.

Teraz na bardziej zaawansowanym przykładzie zbadamy jakie są zasady decydujące o tym, kiedy dodanie oznaczenia `Term` jest wymagane, aby uleniwienie wyrażenia działało zgodnie z życzeniem programisty. Do tej analizy posłużą dwa przykłady uleniwionych wyrażeń:

```
4 + 5 * Term(6); /* (1) */

Term(4) + 5 * 6; /* (2) */
```

Należy mieć świadomość, że dużą rolę w procesie budowy drzewa AST odgrywają własności operatorów. Liczba ich argumentów, priorytety i kierunek wiązania decydują o postaci drzewa, a tym samym o uleniwieniu wyrażenia.

W pierwszym wyrażeniu najpierw kompilator zajmie się podwyrażeniem `5 * Term(6)`, które da w rezultacie `Mul<int, Term<int> >`. Warto odnotować, że tym razem tylko prawy argument operatora binernego był typem szablonu wyrażenia, więc został użyty operator zdefinowany w linii 108 przykładu. W nastepnej kolejności nastąpi dodawanie 4 oraz wyrażenia typu `Mul<int, Term<int> >`, co da w wyniku kolejny bardziej rozbudowany szablon wyrażenia `Sum<int, Mul<int, Term<int> > >`. Udało się pokazać, że pierwsze z analizowanych wyrażeń zachowa się zgodnie z intencją programisty, to znaczy całe wyrażenie zostanie uleniwione.

Analogicznie badamy zachowanie wyrażenia drugiego. Na początku spostrzegamy, że najpierw dojdzie do obliczenia podwyrażenia `5 * 6`. Zatem ewaluacja odbędzie się gorliwie, ponieważ budowa szablonu wyrażenia nie została zapoczątkowana, mamy tu do czynienia ze zwykłym mnożeniem. W następnym kroku odbędzie się dodawanie `Term(4)` i wyniku `5 * 6`, czyli `30`, czego konsekwencją będzie stworzenie szablonu wyrażenia `Sum<Term<int>, int>`. To nie jest to czego oczekiwaliśmy, gdyż założyliśmy, że chcemy uleniwić całe wyrażenie, a zaszło to jedynie dla jego części.

Wnioskiem z powyższej analizy jest reguła propagacji leniwości w czasie budowania szablonu wyrażenia. Propagowanie zachodzi od wartości oznaczonej jako terminal w górę drzewa. Stąd każdy operator na ścieżce pomiędzy uleniwionym terminalem, a korzeniem drzewa (włącznie) będzie uleniwiony, czyli nie będzie wywołany od razu, a zostanie zakodowany w strukturze szablonu wyrażenia i wywołany, gdy nastąpi wymuszenie obliczenia wyrażenia. W szczególności, pisząc kod tworzący szablony wyrażenia w oparciu o przedstawiony przykład należy mieć na uwadze to, że tworzenie drzew AST przez kompilator nie jest jednoznaczne, ponieważ kompilator może w przypadku wystąpienia obok siebie operatorów o identycznym

priorytecie stworzyć różne drzewa AST. Przykładem jest wyrażenie `4 + 5 + 6`.

2.2.3. Praktyczna implementacja szablonu wyrażenia – Boost Proto

Przedstawiony powyżej sposób da się uogólnić dla dowolnego wyrażenia w języku C++. To znaczy, że stworzenie dowolnego leniwego wyrażenia jest możliwe, do czego dążyłem podczas tworzenia implementacji Parallel. Jednakże napisanie takiej biblioteki od podstaw wykraczało poza ramy tej pracy magisterskiej. Dlatego sięgnąłem po rozwiązania już istniejące, publicznie dostepne i sprawdzone. W celu implementacji uleniwionych wyrażeń metodą szablonów wyrażeń wybrałem bibliotekę Boost Proto.

Boost Proto jest biblioteką służącą do tworzenia Wbudowanych Języków Domenowych (z ang. Domain Specific Embedded Language) w języku C++. Zawiera narzędzia do tworzenia, sprawdzania typów, przetwarzania oraz ewaluacji szablonów wyrażeń. Proto zapewnia:

- Szablony wyrażeń w postaci AST
- Mechanizm modyfikacji zachowania wyrażeń
- Przeciążanie operatorów dla budowy AST z wyrażenia
- Narzędzia do definiowania gramatyki wyrażeń
- Rozszerzalny mechanizm wyliczenia szablonu wyrażenia
- Rozszerzalny zestaw operacji przetwarzania szablonów wyrażeń

Biblioteka Boost Proto została wykorzystana do implementacji następujących komponentów biblioteki Parallel:

- Funkcje do oznaczania terminali:
  - `val` – funkcja przekazująca terminal do wyrażenia przez wartość
  - `cval` – funkcja przekazująca terminal do wyrażenie przez wartość stałą
  - `ref` – funkcja przekazująca terminal do wyrażenia przez referencję
  - `cref` – funkcja przekazująca terminal do wyrażenia przez referencję stałą
- `lazyf` – funkcja służąca do tworzenia uleniwionego wywołania funkcji
- `deferred_expr` – szablon typu przechowującego uleniwione wyrażenia i zarządzającego jego wyliczeniem

Biblioteka Parallel korzysta z Boost Proto do tworzenia szablonów wyrażeń w postaci AST, również przy pomocy mechanizmu przeciążania operatorów, dzięki czemu budowa AST jest uproszczona. Ponadto ewaluacja odbywa się przy pomocy dostarczonej przez Boost Proto funkcji `proto::eval` z domyślnym zachowaniem. Zasada działania Boost Proto jest analogiczna do działania przykładu pokazanego powyżej.

2.2.4. Alternatywne rozwiązanie

Do wyboru była również inna możliwość realizacji przekazywania wyrażenia niż forma uleniwiona opisana to powyżej. Podobny efekt można uzyskać stosując obiekty funkcyjne, które reprezentowałyby dane wyrażenie. Istnieje jednak istotny powód, dla którego zostało wybrane pierwsze z przedstawionych rozwiązań. Leniwe wyrażenia mają bardziej intuicyjną składnię, natomiast tworzenie odpowiednich obiektów funkcyjnych wymaga znajomości stosownych bibliotek. Przykładami są Boost.Lambda, Boost.Function i Boost.Bind, jednakże pomimo tego, że są to jedne z najlepszych bibliotek w swojej klasie, w przypadku pisania rozbudowanego wyrażenia ich składnia jest zdecydowanie nieintuicyjna. Szczególnie problematyczny jest zapis zagnieżdżonych funkcji, w tym operatorów. Następujący kawałek kodu nie należy do czytelnych:

```
bind(f, bind(operator+,4, (bind(operator*, 5, 6)))); //f(4 + 5 * 6)
```

Zamiast bardziej intuicyjnego:

```
lazyf(val(4) + val(5) * val(6));
```

Tak naprawdę wystarczyłoby

```
lazyf(f, 4 + val(5)*6);
```

gdyż `val(5)` wprowadziłoby leniwość na najniższym poziomie w drzewie wyrażenia, która następnie propagowałaby się w górę drzewa.

Dlatego, ze względu na znacznie bardziej intuicyjną składnię, wybrano leniwe wyrażenia do realizacji przekazywania obliczeń do wykonania równoległego.

2.3. Implementacja mechanizmu ewaluacji

Zarys mechanizmu ewaluacji został przedstawiony w sekcji Wykonanie zadań. Główną funkcją mechanizmu ewaluacji jest pobieranie zadań (wyrażeń do obliczenia) zlecanych przez kod programu i ich ewaluacja. Do implementacji tej części biblioteki wykorzystano wzorzec projektowy puli wątków oraz kolejki zadań. Główną klasą tej części biblioteki jest `evaluation_mgr` (od ang. evaluation manager – menedżer ewaluacji), która jest singletonem i zarządza całością procesu wykonywania zadań.

2.3.1. Zadania w bibliotece Parallel

Zadaniem w kontekście biblioteki Parallel nazywam uleniwione wyrażenie, które zostało przekazane bibliotece do obliczenia. Zadanie jest reprezentowane przez konkretyzację wzorca typu `deferred_expr` (od ang. deferred expression – wyrażenie odroczone). Jedyny konstruktor publiczny tego wzorca przyjmuje jako argument szablon wyrażenia, taki jaki został opisany w sekcji Implementacja przekazywania wyrażeń do wyliczenia. Szablon wyrażenia jest przechowywany jako jedno z pól `deferred_expr` i poddawany ewaluacji po wywołaniu metody `evaluate`. Pozostałe pola i metody `deferred_expr` służa kontrolowaniu wyliczenia wyrażenia oraz przekazywaniem wyniku do skojarzonego obiektu typu `deferred_value` [2].

[2]Zgodnie z opisem z Operacje wykonywane w kodzie biblioteki Parallel obiekt `deferred_expr` jest skojarzony 1-1 z obiektem `deferred_value`.

2.3.2. Pula wątków

Wzorzec puli wątków został wybrany ze względu na efektywność. Dzięki takiej implementacji unika się tworzenia wątku dla każdego zadania do wykonania, co jest operacją dosyć drogą i warto zadbać, aby nie była wykonywana zbyt często. Każdy z wątków działa w nieskończonej pętli w funkcji `eval_loop`, w której wątek próbuje pobrać zadanie do wykonania z kolejki zadań, a jeśli nie jest ono dostępne to czeka na zmiennej warunkowej. Do implementacji została wykorzystana biblioteka Boost.Threads.

Pula wątków pozwala uniknąć innego problemu związanego z tworzeniem wątku dla każdego zadania. W przypadku gdyby pojawiło się zbyt wiele zadań jednocześnie liczba wątków mogłaby wzrosnąć do takiej liczby, że wydajność programu znacznie by spadła z powodu częstego przełączania kontekstu pomiędzy różnymi wątkami. Pula wątków pozwala ustalić maksymalną liczbę wątków, co przedziwdziała przeciążeniu systemu. Obecnie liczba wątków jest ustalana przez programistę, nic nie stoi na przeszkodzie, aby w przyszłości liczba wątków była dobierana automatycznie w zależności od wydajności systemu bądź innych parametrów.

2.3.3. Kolejka zadań

Kolejka zadań jest standardową kolejką FIFO przechowującą obiekty `deferred_expr`. Ponieważ jednocześnie kod programu lub każdy z wątków z puli przechowywanej w `evaluation_mgr` może chcieć skorzystać z kolejki, dostęp do niej jest chroniony sekcją krytyczną. Dodawanie zadań do kolejki odbywa się w ciele funkcji `eval` przy użyciu funkcji `add_for_eval`.

2.3.4. Procedura ewaluacji

Procedura ewaluacji z punktu widzenia mechanizmu ewaluacji wygląda bardzo prosto. Wątek-robotnik pobrawszy zadanie wywołuje jego metodę `evaluate`. To skutkuje wyliczeniem wyrażenia, a jeśli typ wynikowy wyrażenia jest inny niż `void` to wartość zostaje przekazana do skojarzonego obiektu `deferred_value`. Po wyliczeniu wyrażenia obiekt `deferred_expr` jest niszczony, gdyż nie jest już potrzebny.

Ten obraz komplikuje się, gdy rozważymy pewien bardzo istotny scenariusz. Otóż nie ma problemu jeśli, wątki-robotnicy dokonują ewaluacji zadań z kolejki odpowiednio szybko. Jednak, gdy długość kolejki wzrośnie może dojść do tego, że wątek programu, który zlecił wyliczenie wyrażenia oczekującego w kolejce, będzie potrzebował jego wyniku. Wtedy nastąpi próba pobrania wartości z obiektu `deferred_value`, który jeszcze nie otrzymał obliczonej wartości. To oznacza, że wątek programu musiałby zawiesić wykonywanie i poczekać, aż wartość zostanie obliczona.

Gdy wątek programu oczekuje na wyrażenie, którego ewaluacja już się rozpoczęła, to nie ma innego wyjścia niż poczekanie na dokończenie obliczeń. Jednak, gdy zadanie, na które program oczekuje, jeszcze nie zostało pobrane do wykonania doszłoby do absurdalnej sytuacji, ponieważ program zawiesiłby się i czekałby na wykonanie zadania, podczas gdy sam mógłby wykonać potrzebne obliczenia. Odpowienio zaimplementowana procedura ewaluacji uwzględnia ten problem.

W analogicznej sytuacji wątek programu, gdy zorientuje się, że wyrażenie nie zostało obliczone sam dokona ewaluacji. Zostanie wywołana ta sama procedura `evaluate`, którą wywołują wątki-robotnicy. Może to skutkować zdublowanym wyliczeniem wyrażenia, w przypadku, gdy zarówno wątek programu, jak i jeden z wątków-robotników obliczyliby wyrażenie. Taka sytuacja byłaby niedopuszczalna, ponieważ wykonywanie wyrażenia w C++ nie jest idempotentne z powodu efektów ubocznych.

Najbardziej standardowym rozwiązaniem byłoby umieszczenie w każdym obiekcie `deferred_expr` flagi wraz mutexem ją chroniącym w celu zapobieżenia podwójnej ewaluacji wyrażenia. Jednakże umieszczenie mutex-a w każdym obiekcie ma pewien narzut pamięciowy i wydajnościowy.

Istnieje lepsze rozwiązanie tego problemu wykorzystujące funkcję z biblioteki Boost Threads `call_once` gwarantującą jednokrotne wywołanie pewnej instrukcji. Przekazanie tej funkcji zmiennej typu `once_flag` wraz z funkcją do wykonania (w tym przypadku funkcją `deferred_expr::evaluat` sprawia, że dla danej flagi funkcja wykona się tylko raz. Wywołanie `call_once` z już raz wykorzystaną flagą `once_flag` zakończy się natychmiast bez wywołania przekazanej funkcji. Ponieważ typ `once_flag` ma niewielki rozmiar (na większości platform jest to typ `long`) to umieszczenie go jako pola każdego obiektu `deferred_expr` dodaje mniejszy narzut pamięciowy niż stosowanie mutexów.

Innym ważnym aspektem procedury ewaluacji odroczonego wyrażenia jest obsługa wyjątków. Zgodnie z koncepcją biblioteki wyjątki, które wystąpią podczas ewaluacji wyrażenia (w ciele funkcji `evaluate` są wywoływane, a następnie przekazywane do obiektu `deferred_value` zamiast wyniku. Dzięki temu wyjątek będzie mógł zostać przekazany z obiektu `deferred_value` do głównego wątku programu.

2.3.5. Ograniczenia mechanizmu ewaluacji

W związku ze sposobem przekazywania wyrażeń do obliczenia, możliwości projektowania mechanizmu ewaluacji były dość poważnie ograniczone. W ogólności moglibyśmy wyobrazić sobie sytuację, w której wyrażenie z programu działającego na jednym komputerze byłoby przekazywane do wyliczenia do innych programów lub nawet do innych komputerów, w celu większego rozproszenia i jeszcze lepszego zrównoleglenia wykonania programu. W przypadku biblioteki parallel wystąpiło kilka ograniczeń, które uniemożliwiły zaprojektowanie ogólniejszego mechanizmu obliczeń.

Migracja kodu (taka jak została opisana w książce [DisSys]) nie jest wspierana przez język C++, ponieważ kod kompilowany jest do natywnego kodu maszynowego, a nie kodu pośredniego. Nie ma możliwości zserializowania fragmentu obliczeń i przesłania do wykonania na innym komputerze, o nieznanej architekturze. W przeciwieństwie do języka C++ to jest wykonalne w języku Java.

Alternatywą dla wsparcia języka dla migracji kodu jest rozszerzenie biblioteki o narzędzia automatycznie generujące kod dla klienta (programu zlecającego obliczenia) i serwera (programu wykonującego obliczenia). To przypomina metodę tworzenia RPC i rodzi szereg innych problemów również opisanych w [DisSys]. Implementacja takiego modelu prowadzenia obliczeń w bibliotece Parallel wykraczałaby poza ramy nakreślonej pracy oraz mogłaby ograniczyć użyteczność biblioteki ze względu na bardziej skomplikowany proces programowania i kompilacji.

Kolejne z ograniczeń jest związane z obecnością w wyrażeniu przekazywanym do obliczenia wartości typu referencje lub wskaźniki, które są ściśle zależne od przestrzeni adresowej programu. Przesłanie ich nawet do innego programu na tym samym komputerze wymagałoby wykorzystania specjalnej procedury, gdyż proste przekazanie wartości tego typu powodowałoby błędy w dostępie do pamięci. Obejście tego problemu oferuje mechanizm pamięci współdzielonej, ale powoduje znaczny narzut związany z dostępnem do tego rodzaju pamięci.

Wymienione powyżej ograniczenia uzasadniają podjęcie decyzji o przyjęciu dla biblioteki Parallel mechanizmu ewaluacji wyrażeń opartego o wątki.

2.4. Implementacja zwracania wyniku obliczeń

Możliwość zwracania wyników z obliczeń wykonywanych przez inny wątek jest jedną z najważniejszych cech przemawiających na korzyść biblioteki Parallel w porównaniu do standardowych bibliotek oferujących wielowątkowość. Najważniejszym elementem procesu zwracania wyników przez bibliotekę Parallel jest wzorzec typu `deferred_value` parametryzowany typem wyniku wyrażenia. Obiekty tego typu są pośrednikami, które przekazują informacje z kodu biblioteki do kodu programu.

2.4.1. Podstawowe właściwości wzorca typu `deferred_value`

Przekazanie wartości

Obiekt typu `deferred_value` po utworzeniu i zwróceniu przez funkcję `eval` posiadają wartość nieokreśloną. Dopiero po obliczeniu wyrażenia zapamiętanego w skojarzonym obiekcie `deferred_expr`, otrzymana wartość jest przekazywana do obiektu `deferred_value` w celu jej zapamiętania. Ponieważ obiekt `deferred_expr` jako jedyny ma prawo dokonania takiego przypisania, nie jest konieczna ochrona przed współbieżnym zapisem do obiektu `deferred_value`.

Wymuszenie wyliczenia wyrażenia

Obiekt typu `deferred_value` pozwala na kontrolowanie w pewnym stopniu wyliczenia wyrażenia przez programistę. Mianowicie, wywołanie metody `force` powoduje wymuszenie wyliczenia wyrażenia. Zatem programista w razie takiej potrzeby może uzyskać pewność, że ewaluacja wyrażenia została zakończona. Ponadto, do wymuszenia wyliczenia wyrażenia zawsze dochodzi wtedy, gdy pobierana jest wartość wyrażenia.

Pobranie wartości

Obiekt typu `deferred_value` pozwala na pobranie wartości poprzez wywołanie metody `get_value` lub też niejawnie, ponieważ poprzez operator konwersji do typu, który reprezentuje. Zatem taki ciąg instrukcji:

```
deferred_value<int> d = parallel::eval(parallel::val(4) + 5);
/* ... */
int c = d + 42;
```

jest poprawny syntaktycznie. Zatem obiektu typu wzorcowego `deferred_value` można użyć w każdym miejscu, gdzie dozwolone jest użycie typu wynikowego wyrażenia. Skutkuje to wywołaniem operatora konwersji i pobraniem wartości.

Alternatywnym sposobem definicji wzorca `deferred_value` było nie umieszczanie w jego interfejsie operatora konwersji. Uniemożliwiłoby to stosowanie obiektów typu `deferred_value` w miejsce typu wynikowego wyrażenia, co sprawiłoby, że pobieranie wartości stałoby się operacją zawsze wywoływaną jawnie przez programistę. W trakcie projektowania interfejsu biblioteki podjęto decyzję o umieszczeniu operatora konwersji, ponieważ czyni to składnię bardziej naturalną. Ponadto używanie obiektów typu `deferred_value` w wyrażeniach bez jawnego pobierania wartości ma bardzo istotne znaczenie dla pełnego wykorzystania funkcji przeciążających operatory dla wzorca typu `deferred_value`. Ta kwestia zostanie wyjaśniona w następnej sekcji.

2.4.2. Przeciążanie operatorów szablonu typu `deferred_value`

Motywacja dla przeciążania operatorów

Można wyobrazić sobie scenariusz, w którym chcąc skorzystać z wartości przechowywanej w zmiennej typu `deferred_value`, programista zawsze wywoływałby jawną metodę pobrania wartości bądź stosowałby niejawną metodę, przy użyciu operatora konwersji. Wyglądałoby to w kodzie w sposób następujący. Wersja z jawnym pobraniem wartości:

```
deferred_value<int> d = parallel::eval(parallel::val(4) + 5);
/* ... */
auto c = d.get_value() + 42;
```

Wersja z niejawnym pobraniem wartości:

```
deferred_value<int> d = parallel::eval(parallel::val(4) + 5);
/* ... */
auto c = d + 42;
```

Zanim przejdziemy do dalszej części rozważań bardzo ważne jest przypomnienie motywacji stosowania biblioteki Parallel. Mianowicie celem programisty używającego biblioteki jest zrównoleglenie jak największej części obliczeń. Podkreśliłem słowa "jak największej", ponieważ im większa część obliczeń zostanie wykonana przez wątki biblioteki Parallel, tym potencjalnie szybciej może działać program. Natomiast, tym większa będzie część obliczeń wykonana przez bibliotekę Parallel, im później będzie wymuszane pobranie wartości z obiektów `deferred_value`.

W jawnym przypadku wszystko jest jasne, biblioteka nie ma pola manewru, gdyż programista zażądał pobrania wartości, więc wartość musi zostać policzona i zwrócona przez metodę `get_value`. Czy podobnie jest w drugim przypadku? Czy zmienna `c` musi zostać oznaczona jako typ `int` i powinna zostać do niej natychmiast przypisana wartość 51? Byłoby to poprawne, ale nie jest to konieczne.

Z punktu widzenia zwiększania efektywności wykorzystania biblioteki Parallel lepiej będzie, gdy obliczenie wartości wyrażenia skojarzonego z obiektem `d` nie będzie w tej sytuacji wymuszone, gdyż może zostać odroczone. Aby to uzyskać wystarczy przeciążyć operator dodawania, w taki sposób, aby wyrażenie `d + 42` zwracało wartość odroczoną, zawierającą uleniwione wyrażenie w postaci szablonu wyrażenia. Działanie głównego wątku programu mogłoby się wtedy toczyć dalej bez wymuszania obliczenia wartości `d`, natomiast później gdy zostanie wymuszone obliczenie wartości `c` to rekurencyjnie zostanie wymuszone również wyliczenie `d`, w celu ewaluacji szablonu wyrażenia zapisanego w `c`.

Implementacja przeciążania operatorów

Język C++ pozwala na przeciążenie wszystkich operatorów, oprócz `::`, `.` oraz `.*`. Przecieżanie operatorów dla obiektów `deferred_value` ma sens dla zdecydowanej większości, ale nie dla wszystkich operatorów. Za bezcelowo uznano przeciążanie operatorów `->*` i `->`. Przedefiniowanie operatorów `new`, `new []`, `delete` oraz `delete []` nie było potrzebne. Ponadto domyślne zachowanie operatora `,` jest odpowiednie dla typu `deferred_value`. Pozostałe operatory zostały przedefinowane w taki sposób, aby uzyskać taki efekt, że obiektów `deferred_value` można używać wszędzie tam, gdzie można użyć typu, który dany obiekt `deferred_value` reprezentuje. Przedefinowania operatorów zwracają jako wynik obiekty `deferred_value`, z szablonem wyrażenia, obliczenie którego pozwoli uzyskać pożądany wynik. Oto pełna lista przeciążonych operatorów:

```
+  -  *  /  %  ^  &  |  ~  !  =  <  >  <<   >>  +=  -=  *=  /=  %=  ^=
&=  |=  >>=  <<=  ==  !=  <=  >=  &&   ||  ++  --  []  ()
```

Dla ilustracji sposobu implementacji funkcji przeciążających operatory dla obiektów `deferred_value` posłużę się przykładem przeciążenia dodawania:

```
1    inline deferred_value<T> operator + (T& rhs)
2       { return make_deferred_value_from_exp(lazyf(&deferred_value<T>::get_valu
```

W funkcji przeciążającej operator tworzona jest inna niż znana do tej pory wersja obiektu `deferred_value`. Nie jest zwracana przez funkcję `eval`, a przez `make_deferred_value_from_exp`, do której przekazywany jest szablon wyrażenia. W tym przypadku tworzony jest szablon wyrażenia, w którym funkcja `|get_value|`jest uleniwiona i do wyniku jej wywołania dodawany jest drugi argument dodawania. Pozwala to na uzyskanie oczekiwanego zachowania, o którym była mowa powyżej.

2.4.3. Szczególne postacie wartości zwracanych przez wyrażenie

Niektóre z postaci wyrażeń przekazywanych do obliczenia posiadają typ wynikowy, który wymaga specjalnego traktowania, ponieważ domyślny wzorzec typu `deferred_value` nie zadziałałby w ich przypadku. Poniżej znajduje się opis takich przypadków wraz z prezentacją rozwiązania zastosowanego w implementacji biblioteki Parallel.

Wyrażenie zwracające typ `void`

Może się zdarzyć, że typem wynikowym wyrażenia jest typ `void`. Nie można wtedy mówić o wartości, którą wyrażenie zwraca, nie można również zadeklarować zmiennej typu void. Dlatego obiekt typu `deferred_value`, dla którego typem zwracanym jest typ `void`, powinien być zaimplementowany inaczej. Dla poradzenia sobie z tym przypadkiem powstała odpowiednia specjalizacja wzorca `deferred_value`. Nie posiada ona żadnej wartości, którą można byłoby pobrać ani służących do tego metod.

W tym przypadku rozważałem całkowitą rezygnację ze zwracania obiektu typu `deferred_value` z funkcji `eval`. Istnieje jednak bardzo ważny scenariusz, w którym konkretyzacja wzorca `deferred_value` sparametryzowana typem `void` jest niezbędna. Ilustruje to poniższy przykład:

```
1   #include <parallel.h>
2
3   int main()
4   {
5     /* Array initialized with some numbers */
6     std::vector a = { ... };
7     auto d = parallel::eval(lazyf(sort<int*>, a.begin(), a.end()));
8     /* Do something without using a */
9     d.force();
10    for_each(a.begin(), a.end(), process_a);
11  }
```

Po zleceniu równoległego posortowania tablicy `a` kod może wykonywać czynności niekorzystające z wartości zapisanych w tablicy. Ale w momencie, gdy dochodzi do przetwarzania `a`

programista musi uzyskać pewność, że sortowanie się zakończyło. Wystarczy zatem, że wywołą metodę `force` na obiekcie `deferred_value`, która wymusi wykonanie sortowania, jeśli nie zostało rozpoczęte i poczeka do jego zakończenia. Po powrocie z `force` programista może używać elementów tablicy `a`, mając pewność, że są posortowane.

Wyrażenie zwracające referencję

Również zwracanie referencji do zmiennej jako wyniku obliczenia wyrażenia okazało się problematyczne. Przykładem takiego wyrażenia jest:

```
/* using namespace std, parallel; */
eval(ref(cout) << "Operacja zakończona sukcesem." << endl);
```

W tym przypadku problemem jest inicjalizacja składowej `m_value` wzorca typu `deferred_value`, którego zadaniem jest przechowywanie zwracanej wartości. W przykładzie typ zwracany to `ostream&`. Referencja może być zainicjalizowana wyłącznie w liście inicjalizacyjnej konstruktora obiektu. Ponieważ obiekt, do którego miałaby się odnosić referencja z obiektu `deferred_value` jeszcze nie został obliczony to w konstruktorze nie można ustawić tej referencji. Stąd zastosowanie typu identycznego z typem wynikowym wyrażenia zwracającego referencję nie jest możliwe. Konieczne było stworzenie odpowiedniej specjalizacji wzorca `deferred_value`.

Należy wykluczyć rozwiązanie polegające na skopiowaniu wartości obiektu. Zmieniłoby to semantykę wyrażenia i ograniczyłoby możliwość używania Parallel dla wyrażeń zwracających obiekty niekopiowalne.

Istnieje wśród programistów języka C++ przekonanie, iż referencja jest *de facto* wskaźnikiem, ale z ładniejszą składnią. Pomimo, że to stwierdzenie nie jest prawdziwe, ponieważ istnieją pewne subtelne różnice w semantyce wskaźników i referencji, to rozwiązanie problemu ze zwracaniem referencji zainspirowane tym stwierdzeniem świetnie sprawdziło się w praktyce. Wskaźniki C++ mają bowiem taką istotną różnice w stosunku do referencji, że można je przestawić na inny obiekt, więc nie muszą być inicjalizowane w konstruktorze. Stąd składowa `deferred_value`, która przechowuje wynik obliczenia wyrażenia jest wskaźnikiem, natomiast wyspecjalizowany dla opisywanego przypadku wzorzec `deferred_value` udostępnia interfejs, który emuluje referencję.

2.4.4. Obsługa wyjątków

Jednym z założeń biblioteki Parallel było umożliwienie niezawodnej obsługi sytuacji wyjątkowych, które mogą się zdarzyć podczas wykonywania zleconych obliczeń. Gdy zostaje wyłapany wyjątek jest on przekazywany do odpowiedniego obiektu `deferred_value`, gdzie zostaje zapamiętany. Natomiast w przypadku wymuszenia obliczenia wartości obiektu `deferred_value`, w którym zapamiętano wyjątek, jest on ponownie rzucany i powinien zostać przechwycony oraz obsłużony w kodzie programu.

Rozdział 3

Ewaluacja biblioteki

Ewaluacja biblioteki miała na celu weryfikację czy wykonana biblioteka działa poprawnie i jaka jest jej wydajność. Poszczególne etapy weryfikacji zostały opisane w odpowiednich sekcjach tego rozdziału.

3.1. Testy poprawności

Testy dotyczące poprawności biblioteki Parallel zostały przeprowadzone przy wykorzystaniu narzędzia do przeprowadzania testów Boost Unit Test Framework. Testy te koncentrowały się na sprawdzeniu czy wyniki wyrażeń obliczanych przez bibliotekę Parallel są identyczne do oczekiwanych oraz czy podczas ich obliczania nie występują niespodziewane błędy. Wyodrębnione zostały następujące przypadki użycia biblioteki Parallel, definowane przez różne typy wyrażeń przekazanych do obliczenia, dla których zostały zaimplementowane odpowiednie testy jednostkowe (z ang. unit test):

- wyrażenie z różnymi rodzajami typów:
  - wyrażenie z typami prostymi,
  - wyrażenie zawierajace obiekty złożone.
- wyrażenia z różnymi formami przekazywania zmiennych:
  - zmienna przekazywane przez wartość,
  - zmienna przekazywana przez referencję,
  - zmienna przekazywana przez wskaźnik.
- wyrażenia z różnymi rodzajami typów zwracanych:
  - wyrażenie o typie wynikowym `void`,
  - wyrażenie zwracające typ prosty,
  - wyrażenie zwracające typ złożony,
  - wyrażenie zwracające referencję,
  - wyrażenie zwracające wskaźnik,
  - wyrażenie, którego obliczenie wywołało wyjątek.
- wyrażenie zawierające wywołania funkcji:

– wywołanie funkcji bezargumentowej,
– wywołanie funkcji z argumentami będącymi typami prostymi,
– wywołanie funkcji z argumentami będącymi typami złożonymi,
– wywołanie metody obiektu,
– wywołanie metody statycznej obiektu.

Wymienione przypadki testowe można kombinować ze sobą i tworzyć w nieskończoność coraz bardziej złożone wyrażenia. Zaimplementowany zestaw testów pokrywa wszystkie z wymienionych przypadków.

Dla wszystkich przypadków biblioteka uzyskała wynik pozytywny. Pokazuje to, że biblioteka działa poprawnie dla reprezentatywnego zbioru przykładów wyrażeń w języku C++. Pozwala to sądzić, że biblioteka potrafi obsłużyć poprawnie dowolne wyrażenie języka C++.

3.2. Test wydajnościowy

Test wydajnościowy biblioteki został przeprowadzony poprzez porównanie wyników wykonania algorytmu szybkiego sortowania (z ang. quicksort) zaimplementowanego sekwencyjnie, przy wykorzystaniu biblioteki Parallel lub przy użyciu standardowego mechanizmu wątków Boost.Threads. Celem testów było sprawdzenie prawdziwości hipotez:

- Programy używajace biblioteki Parallel działają nie wolniej niż programy sekwencyjne.
- Odpowiednio napisany program używajacy biblioteki Parallel potrafi uzyskać przyspieszenie proporcjonalne do liczby wykorzystywanych procesorów.
- Czas działania programów napisanych używajać bibliotek Parallel i Boost.Threads jest zbliżony, z przewagą biblioteki Parallel wynikającej z zastosowania puli wątków.

Dla pisania testu przyjęto zastosowanie uproszczonych metod programowania równoległego, gdyż aby różne implementacje można było ze sobą porównywać powinny one działać podobnie. Dlatego najpierw pisany był program sekwencyjny. Następnie, w celu stworzenia implementacji korzystających z Parallel i Boost.Threads, zrównoleglane były fragmenty kodu, które zostały zidentyfikowane jako odpowiednie do zastosowania obliczeń równoległych. Dla tak przygotowanych programów mierzono czasy wykonania przy użyciu biblioteki Boost Posix Time.

3.2.1. Parametry testu

Do testów wykorzystano tablicę liczb całkowitych generowanych przy pomocy liczb pseudolosowych z ustaloną wartością początkową[1]. Tablica mieściła 200 000 000 liczb całkowitych, osiągając rozmiar około 800 000 000 MB.

W każdej z równoległych implementacji zastosowano zasadę “Dziel i zwyciężaj”, to znaczy problem sortowania był dzielony na mniejsze, które następnie były rozwiązywane równolegle. Fragmenty tablicy poniżej 1 000 000 elementów były sortowane sekwencyjnie. Implementacja testów jest do wglądu w załączonych do pracy plikach źródłowych w folderze ptlib-test. Dane na temat systemu komputerowego użytego do testów znajdują się w dodatku B niniejszej pracy. Najważniejszą informacją jest fakt, iż komputer posiadał procesor wyposażony w dwa rdzenie.

[1] Ustalenie wartości początkowej ciągu liczb pseudolosowych ma służyć powtarzalności testu.

3.2.2. Wyniki testu

Każdy z testów został przeprowadzony 30 razy, a wyniki zostały uśrednione, w celu uniezależnienia wyniku od chwilowych wahań wydajności komputera. Oto otrzymane rezultaty:

- sortowanie szybkie sekwencyjne: 36,29 s.
- sortowanie szybkie z Boost.Thread: 20,96 s.
- sortowanie szybkie z Parallel: 20,82 s.

Przyspieszenie osiągnięte w wyniku zrównoleglenia algorytmu szybkiego sortowania wyniosło 1,74 w przypadku implementacji używającej biblioteki Parallel i 1,73 dla implementacji z Boost.Thread.

3.2.3. Podsumowanie wyników testu

Wyniki przeprowadzonych testów potwierdziły postawione hipotezy. Algorytm szybkiego sortowania działał po zrównolegleniu znacznie szybciej niż wersja sekwencyjna. Uzyskane przyspieszenie było proporcjonalne do liczby jednostek obliczeniowych, wykorzystanych do wykonania programu. Czasy wykonania programów zaimplementowanych w oparciu o Parallel i Boost.Thread różniły się nieznacznie, z niewielką przewagą na rzecz biblioteki Parallel. Przewaga ta znajdowała się w granicy błędu statystycznego, jednak mogłaby zostać uwypuklona w przypadku, gdy wystąpiłoby więcej operacji tworzenia wątków w implementacji korzystającej z biblioteki Boost.Thread.

Inną obserwacją poczynioną podczas przeprowadzania testów było błędne wykonanie programu korzystającego z biblioteki Boost.Thread, w przypadku zmniejszenia rozmiaru tablicy, która była sortowana sekwencyjnie do 50 000. Powodem była próba stworzenia większej liczby wątków niż pozwalała na to konfiguracja wykorzystanego do testów komputera. Dla tych samych parametrów program używający Parallel wykonał się bezbłędnie.

Rozdział 4

Podsumowanie

Biblioteka Parallel stworzona w ramach tej pracy służy zrównoleglaniu obliczeń w języku C++. Niniejsza praca argumentuje, iż cele postawione przed biblioteką, które kierowały procesem jej projektowania, zostały spełnione. Biblioteka Parallel pozwala na zwiększenie produktywności programisty piszącego programy równoległe. Jest to możliwe dzięki zaprojektowaniu takiego modelu biblioteki, który pozwala na pisanie czytelnego i intuicyjnego kodu. Równie istotnym priorytetem podczas tworzenia biblioteki było uzyskanie wysokiej efektywności programów pisanych przy pomocy Parallel. Liczba operacji wykonywanych przez bibliotekę została ograniczona do niezbędnego minimum. Bardzo ważnym czynnikiem wpływającym na osiągnięcie wysokiej efektywności było zapewnienie programiście wglądu w sposób działania biblioteki. Jest to istotne ze względu na to, że programista zlecając równoległe obliczenie wyrażenia wie, jak zostanie ono wykonane przez bibliotekę Parallel, dzięki czemu może dostosować podział zadań w taki sposób, aby zmaksymalizować wydajność programu.

Kolejnym ułatwieniem, które biblioteka Parallel zapewnia programiście podczas jej użytkowania, jest zdjęcie z niego obciążenia związanego z projektowaniem komunikacji i synchronizacji pomiędzy różnymi wątkami. Programista musi jedynie zatroszczyć się o spójność danych wykorzystywanych do równoległych obliczeń. Biblioteka Parallel jest w pełni przenaszalna na wszystkie platformy posiadające kompilator języka C++ zgodny z przyszłym standardem C++0x oraz są kompatybilne z biblioteką Boost.

Należy podkreślić, że w ramach pracy przeprowadzono analizę istniejących rozwiązań w celu weryfikacji czy biblioteka Parallel nie dubluje funkcjonalności dostępnych w innych bibliotekach. Wynik poszukiwań okazał się negatywny, żadna z popularnych bibliotek do programowania równoległego w języku C++ nie spełnia wymogów, które zostały zdefiniowane dla biblioteki Parallel. Uzasadniło to potrzebę stworzenia Parallel, która ma szansę wprowadzić nową jakość do programowania równoległego w C++.

W czasie pracy nad biblioteką rozwiązano kilka ciekawych problemów badawczych. Najtrudniejsze okazało się zaprojektowanie sposobu przekazywania wyrażeń do obliczenia. Problem został rozwiązany przy pomocy uleniwiania wyrażeń zgodnie z idiomem C++, zwanym szablonem wyrażenia, który zaimplementowano wykorzystując bibliotekę Boost.Proto. W oparciu o bibliotekę Boost.Threads wykonano implementację mechanizmu wykonawczego biblioteki, w której został wykorzystany wzorzec puli wątków i kolejki zadań. Ponadto, zaprojektowano sposób zwracania wyniku obliczenia wyrażenia, który opiera się na autorskim pomyśle polegającym na wprowadzeniu wartości odroczonych. Pozwoliło to na emulowanie synchronicznego zwracania wyniku (ułatwiającego pracę programisty), podczas, gdy w rzeczywistości wynik jest zwracany asynchronicznie. Rozwiązano również kilka pomniejszych problemów takich jak obsługa wyjątków wywołanych w trakcie równoległych obliczeń, unik-

nięcie zdublowanego obliczenia wyrażenia przekazanego do obliczenia, dostosowanie wartości odroczonej do specjalnych postaci wartości zwracanych oraz odroczenie wymuszenia obliczenia wartości odroczonej do jak najpóźniejszego momentu poprzez przeciążenie operatorów.

Do implementacji biblioteki użyto wielu zaawansowanych technik programowania w języku C++. Były to między innymi obliczenia na wzorcach typów, specjalizacja szablonów klas, pozyskiwanie zasobu przez inicjalizację (z ang. Resource Acquisition Is Initialization), inteligentne wskaźniki, referencje do r-wartości i konstruktor przenoszący (z ang. move constructor). Implementacja Parallel korzysta z bibliotek z kolekcji Boost, docenianych ze względu na ich wysoką jakość oraz będących wzorcem nowoczesnego programowania w C++. Są to biblioteki Boost Proto, Boost Thread, Boost Bind, Boost TypeTraits, Boost SmartPointers, Boost MPL, Boost Ref, Boost Exception, a do testowania biblioteki Parallel została wykorzystana biblioteka Boost.UnitTestFramework.

Biblioteka Parallel jest kompletnym funkcjonalnie i gotowym do użycia produktem. Wszystkie funkcjonalności opisane w niniejszej pracy działają, co zostało potwierdzone testami. W przyszłości biblioteka może zostać rozszerzona o mechanizm automatycznego dobierania optymalnej liczby wątków wykonujących obliczenia oraz wzorce równoległych algorytmów, które oferowałyby zoptymalizowane funkcje do popularnych zastosowań.

Podsumowując, stworzenie biblioteki Parallel zostało poprzedzone szeroką analizą teoretyczną, jak również przy jej tworzeniu uwzględniono praktyczne aspekty stosowania biblioteki. Parallel jest dobrym kandydatem, aby stać się biblioteką szeroko używaną przez programistów języka C++ piszących programy równoległe.

Dodatek A

Zawartość załączonej płyty

Do pracy załączono płytę CD z zawartością:

- **praca_magisterska/mgr_pawel_tryfon.pdf** – elektroniczna wersja tego dokumentu
- **praca_magisterska/src/** – katalog zawierający pliki źródłowe tego dokumentu
- **ptlib/** – folder zawierający implementację biblioteki Parallel
- **ptlib/src/** – folder zawierający pliki źródłowe biblioteki Parallel
- **ptlib/Release/** – folder zawierający pliki do budowania biblioteki w formie plików makefile oraz skompilowany plik biblioteki
- **ptlib-test/** – folder zawierający implementację testów dla biblioteki Parallel
- **ptlib-test/src** – folder zawierający pliki źródłowe testów biblioteki Parallel
- **ptlib-test/Release/** – folder zawierający pliki do budowania testów biblioteki oraz skompilowany program testujący bibliotekę Parallel

Dodatek B

Konfiguracja komputera wykorzystanego do testów wydajnościowych

Do testów użyto komputera z systemem operacyjnym Ubuntu w wersji 10.10. Dokładną konfigurację komputera, zaprezentowaną poniżej, uzyskano poleceniem `lshw`. Załączoną jedynie dane istotne z punktu widzenia interpretacji testów, to znaczy informacje o liczbie i wydajności rdzeni procesorów oraz dane na temat pamięci RAM.

```
  pthink
   description: Notebook
   product: 2089WFY
   vendor: LENOVO
   version: ThinkPad T500
   serial: L3AXL6X
   width: 32 bits
   capabilities: smbios-2.4 dmi-2.4 smp-1.4 smp
 *-core
      description: Motherboard
      product: 2089WFY
      vendor: LENOVO
      physical id: 0
      version: Not Available
      serial: VF28895W17A
    *-firmware
         description: BIOS
         vendor: LENOVO
         physical id: 0
         version: 6FET66WW (2.16 ) (04/22/2009)
         size: 128KiB
         capacity: 8128KiB
         capabilities: pci pcmcia pnp upgrade shadowing escd cdboot bootselect socketedr
   *-cpu:0
         description: CPU
         product: Intel(R) Core(TM)2 Duo CPU     P8600  @ 2.40GHz
```

```
     vendor: Intel Corp.
     physical id: 6
     bus info: cpu@0
     version: 6.7.10
     serial: 0001-067A-0000-0000-0000-0000
     slot: None
     size: 2400MHz
     capacity: 2400MHz
     width: 64 bits
     clock: 266MHz
     capabilities: boot fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic mtrr p
     configuration: id=0
   *-cache:0
        description: L1 cache
        physical id: a
        slot: Internal L1 Cache
        size: 64KiB
        capacity: 64KiB
        capabilities: synchronous internal write-back instruction
   *-cache:1
        description: L2 cache
        physical id: c
        slot: Internal L2 Cache
        size: 3MiB
        capacity: 3MiB
        capabilities: burst internal write-back unified
   *-logicalcpu:0
        description: Logical CPU
        physical id: 0.1
        width: 64 bits
        capabilities: logical
   *-logicalcpu:1
        description: Logical CPU
        physical id: 0.2
        width: 64 bits
        capabilities: logical
*-cache
     description: L1 cache
     physical id: b
     slot: Internal L1 Cache
     size: 64KiB
     capacity: 64KiB
     capabilities: synchronous internal write-back data
*-memory
     description: System Memory
     physical id: 2b
     slot: System board or motherboard
     size: 2GiB
   *-bank:0
```

```
          description: SODIMM Synchronous 1066 MHz (0.9 ns)
          product: M471B5673EH1-CF8
          vendor: 80CE
          physical id: 0
          serial: 877BBB48
          slot: DIMM 1
          size: 2GiB
          width: 64 bits
          clock: 1066MHz (0.9ns)
     *-bank:1
          description: SODIMM DDR2 Synchronous 1066 MHz (0.9 ns) [empty]
          physical id: 1
          slot: DIMM 2
          clock: 1066MHz (0.9ns)
*-cpu:1
     physical id: 1
     bus info: cpu@1
     version: 6.7.10
     serial: 0001-067A-0000-0000-0000-0000
     size: 2401MHz
     capacity: 2401MHz
     capabilities: vmx ht cpufreq
     configuration: id=0
   *-logicalcpu:0
          description: Logical CPU
          physical id: 0.1
          capabilities: logical
   *-logicalcpu:1
          description: Logical CPU
          physical id: 0.2
          capabilities: logical
```

Programy testowe były kompilowane przy pomocy kompilatora GCC w wersji 4.4.5. Do testów zostały wykorzystane biblioteki z kolekcji Boost w wersji 1.42.0.

Bibliografia

[Ben-Ari06] Mordechai Ben-Ari, *Principles of Concurrent and Distributed Programming, Second Edition*, Addison-Wesley, 2006.

[ParC++] Cameron Hughes, Tracey Hughes, *Parallel & Distributed Programming using C++*, Adison-Wesley, 2003.

[TemG] David Vandevoorde, Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, Addison Wesley, 2002.

[BerLand] Krste AsanovĂc, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick, *The Landscape of Parallel Computing Resarch: A view from Berkley*, Electrical Engineering and Computer Science, University of California at Berkeley, 2006, Technical Report No. UCB/EECS-2006-183, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[ParHist] Wilson, Gregory V Virginia, *The History of the Development of Parallel Computing*, Tech/Norfolk State University, Interactive Learning with a Digital Library in Computer Science, 1994.

[Barney] Blaise Barney, Lawrence Livermore National Laboratory, *Introduction to Parallel Computing*, Livermore Computing, `https://computing.llnl.gov/tutorials/tutorials/parallel_comp/`.

[Foster] Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[GenSem] John A. Trono, William E. Taylor, *Further comments on "A Correct and Unrestrictive Implementation of General Semaphores"*, ACM SIGOPS Operating Systems Review, Volume 34 Issue 3, July 2000.

[HasRef] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.6*, `http://www.haskell.org/ghc/docs/6.6/html/users_guide/index.html`.

[Proto] Eric Niebler, Boost.Proto Library `http://www.boost.org/doc/libs/release/libs/proto/index.`

[BoostThread] Anthony Williams, Boost.Thread Library `http://www.boost.org/doc/libs/release/libs`

[Idioms] Sumant Tambe (as the initiator and the lead contributor) and many other authors, *More C++ Idioms*, Wikibooks, `http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms`.

[ExpTem] Klaus Kreft, Angelika Langer, *An Introduction to the Principles of Expression Templates*, C/C++ Users Journal, March 2003, `http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.ht`

[DisSys] Andrew Tanenbaum, Marteen van Steen, *Distributed Systems*, Prentice Hall, 2002.

[SmtPool] Ami Bar, *Smart Thread Pool*, `http://www.codeproject.com/KB/threads/smartthreadpool.aspx`

[ThdPool] Brian Goetz, *Thread pools and work queues*, `http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html`.

[UseC++] Gregory V. Wilson, Paul Lu, *Parallel Programming Using C++ Scientific and Engineering Computation*, MIT Press, 1996.

[Oracle] *Developing Parallel Programs ? A Discussion of Popular Models*, Oracle White Paper September 2010, `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-paralle`

[OpenMP] Ruud van der Pas - Sun Microsystems, *An Overview of OpenMP*, 2009, `http://openmp.org/mp-documents/ntu-vanderpas.pdf`.

[C++Lang] Bjarne Stroustrup, *Język C++*, WNT, 2004.

[TuringCom] Todd L. Veldhuizen, *C++ Templates are Turing Complete*, Indiana University Computer Science.