

Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Paweł Tryfon

Nr albumu: 248444

**Parallel – biblioteka do
równoległego obliczania wyrażeń 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, 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 expression evaluation in C++

Spis treści

Wprowadzenie	5
1. Koncepcja biblioteki	9
1.1. Cele biblioteki	9
1.1.1. Wysoka efektywność	9
1.1.2. Zwiększenie produktywności programisty	9
1.1.3. Czytelność kodu	10
1.1.4. Transparencja	10
1.1.5. Abstrakcja	10
1.1.6. Ograniczenie konieczności korzystania z mechanizmów komunikacji i synchronizacji procesów równoległych	10
1.1.7. Przenaszalność	10
1.2. Prezentacja idei biblioteki Parallel	11
1.2.1. Inspiracja	11
1.2.2. Zlecanie obliczeń	11
1.2.3. Wykonanie zadań	12
1.2.4. Sposób przekazywania wyniku	13
1.2.5. Schemat interakcji kodu programu z biblioteką Parallel	14
1.3. Istniejące biblioteki do programowania równoległego w języku C++	16
1.3.1. Open Multi-Processing (OpenMP)	16
1.3.2. Threading Building Blocks (TBB)	18
1.3.3. Message Passing Interface (MPI)	20
1.3.4. Boost Threads	22
2. Opis implementacji	25
2.1. Architektura biblioteki	25
2.1.1. API biblioteki	25
2.1.2. Operacje wykonywane w kodzie biblioteki Parallel	26
2.1.3. Zwrócenie wyniku	26
2.2. Implementacja przekazywania wyrażeń do wyliczenia	26
2.2.1. Szablony w języku C++	27
2.2.2. Idiom C++ szablonu wyrażenia	27
2.2.3. Praktyczna implementacja szablonu wyrażenia – Boost.Proto	33
2.2.4. Alternatywne rozwiązanie	34
2.3. Implementacja mechanizmu ewaluacji	34
2.3.1. Zadania w bibliotece Parallel	34
2.3.2. Pula wątków	35
2.3.3. Kolejka zadań	35

2.3.4.	Procedura ewaluacji	35
2.3.5.	Ograniczenia mechanizmu ewaluacji	36
2.4.	Implementacja zwracania wyniku obliczeń	37
2.4.1.	Podstawowe właściwości wzorca typu <code>deferred_value</code>	37
2.4.2.	Przeładowanie operatorów szablonu typu <code>deferred_value</code>	38
2.4.3.	Szczególne postacie wartości zwracanych przez wyrażenie	39
2.4.4.	Obsługa wyjątków	40
3.	Podsumowanie	41
	Bibliografia	43

Wprowadzenie

Równoległe prowadzenie obliczeń nie jest nowym tematem w informatyce, gdyż liczy sobie już ponad 50 lat[ParHist]. Trudno wyznaczyć jeden punkt lub jedną pracę, która zapoczątkowała ten kierunek rozwoju informatyki, ale niewątpliwie do jej pionierów należeli tak znani naukowcy jak Amdahl (prawo Amdahl-a), Flynn (klasyfikacja Flynna), Dijkstra (problem sekcji krytycznych oraz semafor), Petri (sieci Petriego).

Zrównoleglanie obliczeń jest jednym ze sposobów przyspieszania wydajności systemów komputerowych. Szczęólnego znaczenia nabrało ostatnimi czasy, ponieważ rozwój technologii mikroprocesorowej dotarł do takiego momentu, że przyspieszanie pojedynczego układu stało się trudne i dlatego nieopłacalne. Stąd obecnie kierunek rozwoju wyznaczany jest przez równoległość, to znaczy umieszczanie w procesorach komputerów wielu układów wykonujących obliczenia równoległe w jednym czasie. Takie rozwiązanie teoretycznie pozwala na uzyskanie przyspieszenia wprost proporcjonalnego do liczby układów umieszczonych w procesorze. Żeby się jednak tak stało programy wykonywane na takim procesorze powinny być w stanie wykorzystać te możliwości.

Zatem w programach powinno być stosowane równoległe prowadzenie obliczeń, 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 ciągi wykonania się ze sobą komunikują i w jaki sposób są synchronizowane. W celu zapewnienia efektywności programu programista powinien uwzględnić w projekcie również balansowanie rozkładu pracy pomiędzy poszczególne wątki wykonania.

Programistę w zmaganiach z pisaniem programów wspierają różnorodne narzędzia, języki programowania specjalnie zaprojektowane do obliczeń równoległych jak Ada lub biblioteki oferujące wykonywanie obliczeń równoległych w językach natywnie sekwencyjnych. Zazwyczaj te pierwsze wywiązują się ze swojego zadania dobrze, gdyż zostały są dedykowane do programowania równoległego, lecz nie są popularne w zastosowaniach praktycznych. Większym problemem jest korzystanie z tych drugich, ponieważ język sekwencyjny zazwyczaj 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 takiej biblioteki dla języka C++. Nadrzednymi priorytetami podczas projektowania biblioteki Parallel było zapewnienie łatwości pisania programów równoległych (ukrycie niepotrzebnych detali przed programistą) z jednoczesnym pozostawieniem pełnej kontroli nad wykonaniem programu w rękach programisty. Dzięki temu miały zostać osiągnięte dwa główne cele projektowania biblioteki zwiększeni produktywności programisty oraz zwiększenie szybkości działania programów

(przyspieszonych przez wykorzystanie równoległości).

Biblioteka Parallel pozwala na zrównoleglanie obliczeń w zwiezły 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 tę sytuację przykład wykorzystania biblioteki. Zgodnie ze zwyczajem stosowanym w literaturze informatycznej jako pierwszy pokażę fragment kodu zrównoleglający wypisanie komunikatu “Hello World!”. Standardowo w języku C++ taki kod wygląda następująco:

```
1  #include <iostream>
2
3  using std;
4
5  int main()
6  {
7      cout << "Hello World!" << endl;
8      cout << "...lo World, ...orld, ...rld" << endl; /* Echo */
9      return 0;
10 }
```

W przypadku wykorzystania biblioteki Parallel przyjmie on następującą postać:

```
1  #include <parallel.h>
2  #include <iostream>
3
4  using std, parallel;
5
6  int main()
7  {
8      parallel::eval(ref(cout) << "Hello World!" << endl);
9      cout << "...lo World, ...orld, ...rld" << endl; /* Echo */
10     return 0;
11 }
```

Zrównoleglenie będzie polegało w tym przypadku na tym, że kod główny programu zleci równoległe obliczenie wyrażenia wypisującego “Hello World!” bibliotece Parallel i nie czekając na wynik przejdzie do następnej instrukcji. W szczególności w tym przypadku może się zdarzyć, że odpowiedź echa zostanie wypisana szybciej niż pierwszy komunikat. Wprawdzie ten fragment kodu nie pokazuje przydatności biblioteki, ale prezentuje prosty przykład składni i pomoże zrozumieć dalszy opis biblioteki. Bardziej obszerne przykłady znajdują się 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 podsumowujący nakreśla możliwości dalszego rozwoju biblioteki.

Definicje pojęć i skrótów

Pojęcie	Definicja
Idiom C++	Konstrukcja języka C++, która często pojawia się w kodzie lub projektach doświadczonych programistów C++. Stosowanie jej uważane jest za dobrą praktykę.
AST	Abstrakcyjne Drzewo Syntaktyczne (z ang. Abstract Syntax Tree) jest drzewem reprezentującym strukturę kodu źródłowego w pewnym języku programowania.

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 idea 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. Dlatego 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, które 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 (granularność obliczeń).

1.1.2. Zwiększenie produktywności programisty

Problem z efektywnością programisty w przypadku pisania programów równoległych polega na tym, że takie programy są trudne do pisania, stąd 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, dzięki wprowadzeniu modelu programowania równoległego, który będzie tak intuicyjny jak programowanie sekwencyjne. Dzięki czemu napisanie kodu, który działa współbieżnie, będzie 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 równoległe 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], gdzie 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 powinien 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 zleczanych 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

Projektowanie 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 wyřęczyć 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 skutkować 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 równoległe będzie służyła funkcja **eval**, przyjmująca

jako argument wyrażenie do wykonania. Obliczenie przekazanego wyrażenia odbędzie się równolegle.

Składnia wyrażeń powinna być jak najbardziej zbliżona do składni języka C++, bo to by 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(fibonacci_od(40));
```

W tej chwili czytelnik dobrze zaznajomiony z semantyką języka C++ mógł dostrzec pewien problem do pokonania związany z przekazaniem wyrażenia do wykonania. W podanym przykładzie takie wyrażenie najpierw zostałoby wyliczone, ponieważ język C++ posiada gorliwą semantykę wyliczania wyrażeń, a dopiero następnie byłoby przekazane do funkcji `eval`. Zatem funkcja `eval` otrzymałaby gotowy wynik i żadne równoległe obliczenia nie byłyby już potrzebne. Takiej sytuacji należy unikać 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 równoległe pojawia się w kodzie tam, gdzie jest to najwygodniejsze, w jawnej postaci, a potem może zostać przekazane od mechanizmu ewaluacji biblioteki `Parallel`.

Leniwe wyrażenia w języku C++

Stworzenie leniwego wyrażenia w języku C++ nie wydaje się prostym zadaniem. Domyśla semantyka obliczeń jest gorliwa, nie ma słów kluczowych pozwalających na dodanie leniwości, C++ nie pozwala również na rozszerzenie składni języka. Wskazówka do rozwiązania 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 dostępnego dla programisty. Funkcja służy do zlecenia wykonania obliczeń równoległych. W ciele funkcji znajduje się przekazanie wyrażenia w formie leniwej do mechanizmu ewaluacyjnego. Wartość zwracana przez funkcję powinna udostępniać możliwość 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 z wzorcem producent-konsument. Producentem wyrażeń będzie kod programu korzystającego z biblioteki Parallel i przekazujący wyrażenia przy pomocy funkcji `eval`. Natomiast konsumentem będą wątki-pracownicy biblioteki Parallel, których jedynym zadaniem będzie oczekiwanie na zadania (wyrażenia do ewaluacji) i ich wykonywanie. Ze względu na efektywność wykorzystany zostanie wzorzec puli wątków (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 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 będzie potrzebne w programie, który przekazał je do obliczenia. W tej sytuacji biblioteka Parallel reaguje w taki sposób, by nie blokować programu bez przyczyny. Program dotarłby do takiego miejsca, że potrzebuje wyliczonego wyrażenia przekazanego do Parallel, a nie zostało jeszcze wykonane, program musiałby oczekiwać na wykonanie obliczeń. Skoro program nie ma nic innego do wykonania to wątek programu sam może wyliczyć wyrażenie, odpowiednio informując bibliotekę Parallel, aby wyrażenie nie zostało wykonane kilkakrotnie, gdyż nierozsądne byłoby narzucanie ograniczenia nakazującego, żeby wyrażenia przekazywane do biblioteki Parallel były idempotentne. Całością procesu alternatywnego wyliczania obliczenia przez wątek programu powinna zarządzać biblioteka, tak aby nie było 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.

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 wątków-pracowników oraz kodu 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 jednocześnie. 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 leży w tym, że w momencie powrotu z funkcji `eval`

będzie znany jedynie typ wyrażenia, a nie jego wartość. Wartość zostanie obliczona później przez mechanizm ewaluacji biblioteki Parallel. Wynika stąd potrzeba wprowadzenie klasy obudowującej wartość zwracaną, uchwytu do wyniku, nazwijmy go wartością odroczoną (z ang. deferred value). Aby wartość odroczone 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.

Wartości odroczonej można używać prawie zamiennie z typem, który obudowuje. To dzięki temu, że jest dostępna niejawna konwersja do obudowywanego typu, a zastosowanie operatorów będzie dawało identyczny wynik. 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

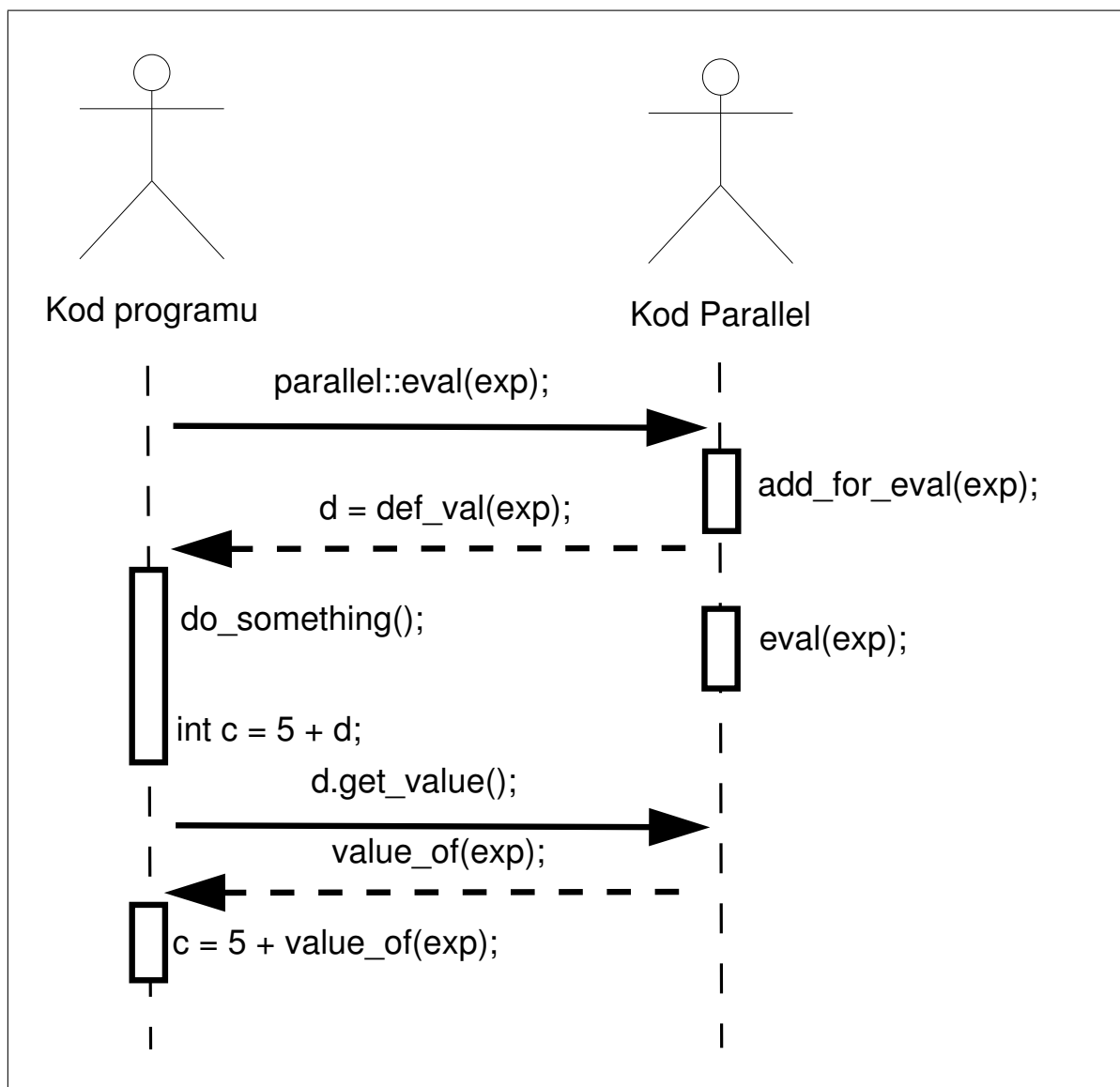
Należy również przyjrzeć się sytuacji, w której w czasie obliczania wyrażenia wystąpi błąd, zasygnalizowany przez rzucenie wyjątku. Biblioteka Parallel powinna wspierać obsługę takiej sytuacji w kodzie. Aby reakcja ze strony programisty była możliwa wyjątek musi zostać złapany i zasygnalizowany programiście. Przepuszczenie wyjątku przez bibliotekę skutkowałoby natychmiastowym błędnym zakończeniem programu, gdyż programista nie miałby żadnej możliwości złapania wyjątku, gdyż wyjątek ten pochodziłby z kodu biblioteki Parallel wykonywanym w innym wątku. Biblioteka nie powinna również tłumić wszystkich wyjątków, gdyż programista nie wiedziałby wtedy co się dzieje w programie, a wartość przypisana na wartość odroczoną byłaby niezdefiniowana.

Nasuwa się zatem rozwiązanie, w którym biblioteka wyłapywałaby wyjątki wywołane w obliczanych wyrażeniach, ale przekazywane byłyby one do wątku głównego programu. Może to nastąpić jedynie za pośrednictwem wartości odroczonej, ponieważ powrót z funkcji `eval` następuje zazwyczaj przed rozpoczęciem obliczania wyrażenia. Wartość odroczone ma ustalony typ, więc wartość odroczone nie może zwrócić wyjątku. Jednakże, to co może zostać wykonane to ponowne rzucenie wyłapanego i przekazanego do wartości odroczonej wyjątku przy próbie pobrania wyliczonej wartości. Wtedy wyjątek powstały w obliczeniach byłby przekazywany do kodu głównego wątku i programista miałby możliwość jego obsługi w dobrze zdefiniowanym miejscu.

1.2.5. Schemat interakcji kodu programu z biblioteką Parallel

Podsumowania opisu idei biblioteki Parallel dokonam 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 wyrażen 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. Kod programu wykonuje własne instrukcje, a kod Parallel wykonuje własne zadania, między innymi wyliczając wyrażenia, które zostały wskazane do wyliczenia równoległego. Takie wyliczenie zachodzi w pewnym momencie w



Rysunek 1.1: Schemat interakcji z biblioteką Parallel

czasie również dla wyrażenia `exp`.

Gdy nadchodzi moment w kodzie programu, gdy potrzebna jest wyliczona wartość wyrażenia to następuje automatycznie próba pobrania wartości wyrażenia z wartości odroczonej `d`. Wtedy następuje przekazanie sterowania 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 zlecenia wyrażeń do równoległego wyliczenia w bibliotece Parallel.

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

W tej sekcji opis idei biblioteki Parallel zostanie przeciwstawiony przeglądowi obecnie istniejących bibliotek do programowania współbieżnego w języku 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 rozwiązań i pokrywa inny zakres potrzeb programistów. Swoją przeglądnę oparę 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 w dla systemów wieloprocessorowych 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 raz 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 główny, 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 zdefiniowany przez programistę podział zadań. 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. Następnie wątki wykonują zadanie, po czym na końcu kodu zaznaczonego 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 prosty 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 zrównoleglenie pętli for, w której sumowane są elementy dwóch tablic i wynik przypisywany jest na trzecią tablicę. Dyrektywa `#pragma omp for schedule(dynamic,chunk)` 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
<ul style="list-style-type: none">• Równoległość inkrementacyjna, możliwe jest dodawanie zrównoleglania obliczeń stopniowo, bez drastycznych zmian w kodzie.• Mała potrzeba zmian w kodzie przy zrównoleglaniu• W obu przypadkach możliwa jest kompilacja do kodu sekwencyjnego bez żadnych modyfikacji w kodzie.• OpenMP i Parallel działają tylko na platformach z pamięcią współdzieloną.	<ul style="list-style-type: none">• W OpenMP dekompozycja zadań domyślnie jest dokonywana automatycznie• OpenMP nie jest zwykłą biblioteką języka i potrzebuje wsparcia kompilatora.• Parallel zostało zaprojektowane do zrównoleglania zadaniowego, a nie zrównoleglania na poziomie danych (niższa 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 obciążony jest tworzeniem, usuwaniem lub synchronizacją wątków i przypisywaniem do nich zadań.

W przypadku TBB programista, zamiast definiować działanie współbieżnego fragmentu programu manualnie, używa szkieletów algorytmów dostępnych w tej bibliotece. Następnie biblioteka już 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 liczba wątków jest dobierana automatycznie przez TBB zależnie od konfiguracji komputera.

Przykład wykorzystania biblioteki TBB do zrównoleglania 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 }
```

```

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. Poniżej 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
<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • Użycie TBB zazwyczaj wymaga zmian w kodzie. Choć jego struktura pozostaje w większości niezmieniona, to niezbędne jest zdefiniowanie 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 zdefiniowane 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 definiowania 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 uważane jest za dosyć niskopoziomowy sposób pisania programów równoległych, gdyż całe działanie programu musi zostać zapisane jawnie przez programistę. Programista ustala liczbę procesów, kierunki komunikacji, mechanizmy synchronizacji, podział danych i ich rozkład pomiędzy procesy oraz przydział procesów do procesorów. Jako korzyść z trudności programowania przy pomocy MPI można zaliczyć wydajność z jaką działają dobrze napisane programy oraz łatwość z jaką można skalować je na większą liczbę procesorów. Ponadto dzięki szerokiemu wsparciu MPI przez największych dostawców sprzętu i oprogramowania, programy pisane przy pomocy MPI są przenośne.

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(MPLCOMM_WORLD,&numprocs);
24     /* sprawdzenie numeru danego procesu w grupie */
25     MPI_Comm_rank(MPLCOMM_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, MPLCHAR, i, TAG, MPLCOMM_WORLD);
34 }
35 for (i=1; i<numprocs; i++)
36 {
37     MPI_Recv(buff, BUFSIZE, MPLCHAR, i, TAG,
38             MPLCOMM_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, MPLCHAR, 0, TAG,
46             MPLCOMM_WORLD, &stat);
47     sprintf(idstr, "Processor %d ", myid);
48     strncat(buff, idstr, BUFSIZE-1);
49     strncat(buff, "reporting for duty\n", BUFSIZE-1);
50     /* wyslanie wiadomosci do procesu z identyfikatorem 0 */
51     MPI_Send(buff, BUFSIZE, MPLCHAR, 0, TAG, MPLCOMM_WORLD);
52 }
53
54 /* Program MPI powinien zakonczyc sie 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.

Kod pokazuje, że programowanie w MPI można uznać za dosyć niskopoziomowe. Operacje wysyłania i odbierania wiadomości działają analogicznie do funkcji systemowych read i write, operują na poziomie bitów. Stąd programista powinien także zatroszczyć się o odpowiednie zakodowanie i rozkodowanie wiadomości.

Porównanie MPI vs. Parallel

Podobieństwa	Różnice
<ul style="list-style-type: none">• Obie biblioteki zostały zaprojektowane do zrównoleglania kodu na poziomie zadań do wykonania.	<ul style="list-style-type: none">• W MPI raczej trudno jest stopniowo zrównoleglać program. Fragmenty programu, które mają działać równolegle muszą zostać zapisane w całości i muszą dokładnie sobie odpowiadać.• MPI wymaga znacznych strukturalnych zmian w kodzie.• Ze względu na to, że MPI koordynuje pracę wielu niezależnych procesów, które działają w sposób asynchroniczny 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 <i>explicite</i> 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 Threads

Istnieje kilka bibliotek oferujących programiście możliwość uruchamiania wielu wątków w ramach jednego programu. Wśród nich można wymienić POSIX Threads, Windows Threads, najbardziej typowe rozwiązania dla platform odpowiednio Unix i Windows. Do opisu została Boost.Threads ze względu na przenaszalność, natomiast zasada działania i oferowane możliwości są analogicznie jak w innych tego typu bibliotekach.

Biblioteka Boost Threads 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 Threads:

Porównanie Boost Threads vs. Parallel

Podobieństwa	Różnice
<ul style="list-style-type: none">• Boost Threads 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.	<ul style="list-style-type: none">• Stopniowe dodawanie równoległości przy użyciu Boost Threads jest trudniejsze, ponieważ użycie biblioteki wymaga zmian w strukturze kodu i dodania mechanizmu komunikacji między wątkami.• Nie ma możliwości przekazania do wątku wyrażenia do wykonania, wątki przyjmują do wykonania jedynie obiekty funkcyjne, co wiąże się z koniecznością dodania do kodu definicji tych obiektów.• Kod Boost Threads jest mniej czytelny, gdyż składnia obiektów funkcyjnych jest mniej czytelna niż składnia wyrażeń.• Przekazywanie wartości między wątkami wymaga dodatkowej synchronizacji i użycia specjalnych fragmentów pamięci współdzielonej lub wskaźników.• W Parallel zadania są przydzielane wątkom dynamicznie, dzięki czemu równoważone jest obciążenie wątków.• 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. Można taki efekt uzyskać pisząc dodatkowy do przekazywania wyjątków pomiędzy wątkami.

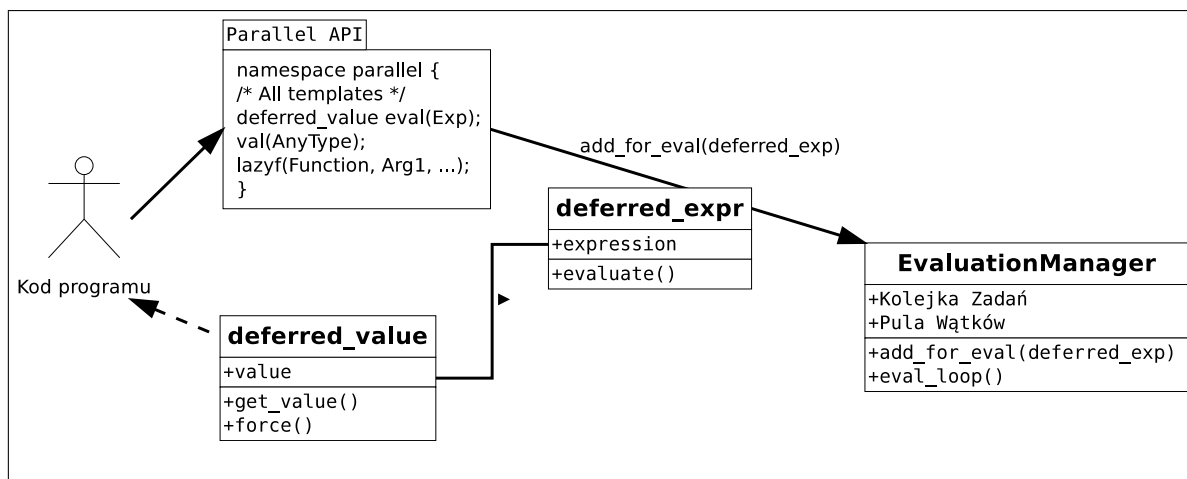
Rozdział 2

Opis implementacji

W tym rozdziale zostaną opisane szczegóły implementacyjne biblioteki Parallel. Nie jest moim celem opisanie 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 wyzwania i wywierają największy wpływ na wynik finalny.

2.1. Architektura biblioteki

Poniżej znajduje się diagram pokazujący główne komponenty biblioteki i to jak one od siebie wzajemnie zależą. Podczas omawiania implementacji biblioteki Parallel będę odwoływał się często do elementów przedstawionego schematu. Pozwoli to na lepsze zrozumienie późniejszych rozważań.



Rysunek 2.1: Schemat architektury biblioteki Parallel

Diagram przedstawia najważniejsze elementy statycznej struktury biblioteki wraz z zależnościami między nimi. Na diagramie nie znajdują się wszystkie metody zaimplementowane w poszczególnych klasach, a jedynie te niezbędne do zaprezentowania sposobu 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 zosta-

ło 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 odpowiednio 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 się wzorzec funkcji pozwalający na uleniwione wywoływanie funkcji, `lazyf`. Dokładna rola i sposób tych funkcji jest opisana w sekcji Implementacja przekazywania wyrażeń do wyliczenia.

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` z przypisanym odpowiednim wyrażeniem, który reprezentuje zadanie do wykonania. Zadanie do wykonania 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 te zadania wykonują 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 zdefiniowane przez wyrażenie zapisane w obiekcie `deferred_expr` zostaje wykonane, to wynik (jeśli istnieje) zostaje przypisany na pole `value` w skojarzonym obiekcie typu `deferred_value`.

2.1.3. Zwrócenie wyniku

Wynik jest przekazywany do kodu programu poprzez wartość typu `deferred_value`, która jest zwracana jako wynik funkcji `eval`. Jeśli nastąpi wywołanie pobrania wartości z obiektu typu `deferred_value` metodą `get_value` lub wymuszenie wyliczenia wartości przez metodę `force` to jeśli skojarzone `deferred_expr` nie zostało wykonane to wyliczenie wyrażenia zostanie wymuszone, o ile się jeszcze nie rozpoczęło.

2.2. Implementacja przekazywania wyrażeń do wyliczenia

Zaprojektowanie przekazywania wyrażeń do wykonania było jednym z najtrudniejszych zadań podczas fazy implementacji biblioteki Parallel. 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. Definicja problemu podpowiadała rozwiązanie, skoro problemem jest gorliwość należało wyrażenia przekazywane do wyliczenia uleniwić.

Najbardziej naturalne byłoby użycie słowa kluczowego lub funkcji, która uleniwiłaby wyrażenie. Pierwszy pomysł był nie do zrealizowania, ponieważ języka C++ nie można rozszerzyć o nowe słowa kluczowe, a standard języka nie przewidywał 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 szablonów typów i funkcji w języku C++.

2.2.1. Szablony w języku C++

Nie chciałbym zamieszczać tutaj opisu podstawowych informacji na temat szablonów. Wprowadzenie do tej tematyki można przeczytać w książce [C++Lang]. Moim celem jest pokazanie, dlaczego implementacja przekazywania wyrażeń tak jak tego dokonałem była możliwa.

Szablony są bardzo silnym mechanizmem, aczkolwiek siła ekspresyjna szablonów nie była zamierzona przez twórców i dopiero przypadkiem zostały odkryte możliwości, które dają. 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 typy. Oto przykład obliczania silni w czasie kompilacji przy pomocy szablonów:

```
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; };
```

Obliczenie odbywa się rekurencyjnie, granicą rekurencji jest `factorial<0>`, które jest specjalizacją szablonu `factorial`. Taka konstrukcja pozwala obliczyć dowolne wartości silni w czasie kompilacji¹ i korzystać z nich w czasie stałym podczas wykonania programu.

2.2.2. Idiom C++ szablonu wyrażenia

Intuicyjnie rzecz ujmując, idiom szablonu wyrażenia polega na opisaniu wyrażenia przy pomocy instancji pewnego szablonu typu. Ten typ nie jest nigdy zapisywany w postaci jawnej w programie, gdyż należałoby oczekiwać, 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 instancja tego samego szablonu typów. W tym przypadku szablon wyrażenia może zawierać pola, które są szablonem wyrażenia. W ten sposób powstaje drzewo reprezentujące wyrażenie w postaci drzewa typów. Taka konstrukcja nazywana jest Abstrakcyjnym Drzewem Syntaktycznym (z ang. Abstract Syntax Tree - AST). W takiej konstrukcji w liściach drzewa znajdują się terminale wyrażenia, czyli szablony wyrażenia nie zawierające już w sobie żadnych szablonów wyrażenia. 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.

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

Zaprezentuje ideę szablonu wyrażenie 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 węzłów 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 reprezentującym węzeł 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(mul.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 reprezentującym szablon wyrażenia.² Teoretycznie wszystkie wartości powinny być oznaczone w ten sposób. Jednak jak się później dowiemy identyczne działanie można uzyskać oznaczając tylko część wartości.

Wzorce **Sum** oraz **Mul** reprezentują węzły pośrednie w drzewie AST (linie 15-29). Ich lewa i prawa strona 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 zdefiniowane w `Exp`, z tym, że te definicje są dopasowane do `Sum<T1, T2>`, ponieważ są parametryzowane tym typem.

Kolejna część przykładu (linie 33-78) zajmuje się definicją wzorca **Infer**, który oblicza typ wynikowy wyliczenia wyrażenia, będzie on potrzebny do definicji funkcji ewaluujących wyrażenie. Wykorzystywany jest tutaj mechanizm specjalizacji wzorców w języku C++, 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 dokładnie analogiczna jak część dotycząca wzorca **Sum**).

Na koniec sekcji kodu, która tworzy AST (linie 83-99) znajduje się definicja wzorca **Exp**, którego rola została opisana powyżej. **Exp** uwspólnia definicje operatorów w klasach węzłów pośrednich.

Kolejna duża 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 reprezentuje dodanie dwóch liczb całkowitych i szablon wyrażenia, który zostanie stworzony w wyniku obliczeń na typach 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. Jest tak, ponieważ jeśli, żadna z wartości nie byłaby oznaczona byłoby to zwykłe wyrażenie C++ i zostałyby obliczone gorliwie. Ponadto, nie da się tego zmienić przy pomocy przeładowywania

²Jednak wzorce **Term** będą zaliczane do węzłów pośrednich, wartość, którą on trzyma jest liściem.

operatorów, gdyż co najmniej jednym argumentem operatora musi być klasa lub typ wyliczeniowy. Zastanówmy się, czy poniższe wyrażenie jest poprawne składniowo i jeśli tak, to jaki będzie szablon typu, który powstanie.

$\text{Term}(4) + 5;$

W tym wyrażeniu mamy do czynienia z operacją dodawania typu $\text{Term}\langle\text{int}\rangle$ oraz typu int . Ponieważ $\text{Term}\langle\text{int}\rangle$ dziedziczy operator dodawania z $\text{Exp}\langle\text{Term}\langle\text{int}\rangle\rangle$ to zostanie wywołany operator dodawania zdefiniowany w linii 87 przykładu. Warto zauważyć, że typ prawego argumentu nie jest w żaden sposób ograniczony, więc zastosowanie typu int jest poprawne i podane wyrażenie jest poprawne składniowo. W związku z tym wynikiem budowy drzewa tego wyrażenia będzie $\text{Sum}\langle\text{Term}\langle\text{int}\rangle, \text{int}\rangle$. To pokazuje, 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 nastąpi jedno wywołanie funkcji dla każdego węzła pośredniego.

Teraz na bardziej zaawansowanym przykładzie zbadamy to 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 poniższe przykłady uleniwionych wyrażeń:

$4 + 5 * \text{Term}(6); /* (1) */$

$\text{Term}(4) + 5 * 6; /* (2) */$

Należy mieć świadomość, że dużą rolę w procesie budowy drzewa AST odgrywają priorytety operatorów. To one decydują w dużej mierze o postaci drzewa i o tym w jaki sposób wyrażenie zostanie uleniwione.

W pierwszym wyrażeniu najpierw kompilator zajmie się podwyrażeniem $5 * \text{Term}(6)$, które da w rezultacie $\text{Mul}\langle\text{int}, \text{Term}\langle\text{int}\rangle\rangle$. Warto odnotować, że tym razem tylko prawy argument operatora binarnego był typem szablonu wyrażenia, więc został użyty operator zdefiniowany w linii 108 przykładu. W następnej kolejności nastąpi dodawanie 4 oraz wyrażenia typu $\text{Mul}\langle\text{int}, \text{Term}\langle\text{int}\rangle\rangle$, co da w wyniku kolejny bardziej rozbudowany szablon wyrażenia $\text{Sum}\langle\text{int}, \text{Mul}\langle\text{int}, \text{Term}\langle\text{int}\rangle\rangle\rangle$. 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 $\text{Term}(4)$ i wyniku $5 * 6$, czyli 30, czego konsekwencją będzie stworzenie szablonu wyrażenia $\text{Sum}\langle\text{Term}\langle\text{int}\rangle, \text{int}\rangle$. 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 to w jaki sposób dochodzi do propagowania się leniwości w czasie budowania szablonu wyrażenia. To 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 pisania pracy. Jednakże napisanie takiej biblioteki od podstaw wykroczało poza ramy tej pracy magisterskiej. Dlatego sięgnąłem po rozwiązania już istniejące, publicznie dostępne. Wybrałem bibliotekę Boost.Proto w celu implementacji uleniwionych wyrażeń, metodą szablonów wyrażeń.

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 wykonywania szablonów wyrażeń. Proto zapewnia:

- Szablony wyrażeń w postaci AST
- Mechanizm modyfikacji zachowania wyrażeń
- Przeładowania operatorów dla budowy AST z wyrażenia
- Narzędzia do definiowania gramatyki wyrażeń
- Rozszerzalny mechanizm natychmiastowego wyliczenia szablonu wyrażenia
- Rozszerzalny zestaw operacji przetwarzania na szablonach 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 wyrażenia przez stałą wartość (modyfikator `const`)
 - `ref` – funkcja przekazująca terminal do wyrażenia przez referencję
 - `cref` – funkcja przekazująca terminal do wyrażenia przez stałą referencję
- `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 przeładowywania 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

Oczywiście istnieje alternatywa do przekazywania wyrażenia do ewaluacji w formie leniwej, tak jak opisano 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 ich składnia jest w przypadku pisania rozbudowanego wyrażenia zdecydowanie nieintuicyjna. Szczególnie problematyczny jest zapis zagnieżdżonych funkcji, w tym operatorów. Następujący kawałek kodu nie należy do czytelnych:

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

Zamiast bardziej intuicyjnego:

```
fun(f)( val(5) + val(4) );
```

Tak naprawdę wystarczyłoby

```
f(val(5)+4);
```

gdyż `val(5)` "zaszczepiłoby" leniwość na najniższym poziomie w drzewie wyrażenia, która następnie propagowałaby się w stronę potomków i rodzeństwa tego węzła w drzewie wyrażenia.

Stąd ze względu na znacznie bardziej intuicyjną składnię dla przekazywania obliczeń do wykonania równoległego zostały wybrane leniwe wyrażenia.

2.3. Implementacja mechanizmu ewaluacji

Zarys mechanizmu ewaluacji został przedstawiony w sekcji Wykonanie zadań. Główną funkcją mechanizmu ewaluacji jest pobieranie zadań (wyrażeń do obliczenia) zleczanych 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 wzorzec typu `deferred_expr` (od ang. deferred expression – wyrażenie odroczone). Najistotniejszą własnością `deferred_expr` jest to, że jedyny konstruktor publiczny 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`. Ten szablon wyrażenia jest poddawany ewaluacji po wywołaniu metody `evaluate`. Pozostałe pola i metody `deferred_expr` służą kontrolowaniu wyliczenia wyrażenia oraz przekazywaniem wyniku do skojarzonego obiektu typu `deferred_value` ³.

³Zgodnie 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 w łatwy sposób 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, więc pewna liczba nigdy nie zostanie przekroczona i jeśli zostanie ustawiona odpowiednio, nie dojdzie przeciążenia 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 niszczone, gdyż nie jest już potrzebny.

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

Nie jest złą sytuacją, gdy wątek programu oczekuje, gdy ewaluacja wyrażenia już się rozpoczęła, ponieważ wartość zostanie przekazana tak szybko jak to możliwe. Jednak, gdy zadanie, na które kod programu oczekuje, jeszcze nie zostało pobrane do wykonania dochodzi do absurdalnej sytuacji, dlatego, że wątek programu nic nie robi i czeka na wykonanie zadania, podczas gdy mógłby sam wykonać potrzebne obliczenia. Odpowieniem zaimplementowana procedura ewaluacji w bibliotece `Parallel` unika tego problemu.

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 obliczyłoby wyrażenie. Taka sytuacja byłaby niedopuszczalna, ponieważ obliczanie w C++ nie jest idempotentne z powodu efektów ubocznych.

Najbardziej standardowym rozwiązaniem byłoby umieszczenie w każdym obiekcie `deferred_expr` flagi wraz mutex-em ją chroniącym w celu zapobieżenia podwójnej ewaluacji wyrażenia. Zapewniłoby to również drobną ziarnistość ochrony, co zwiększałoby współbieżność i jednocześnie wydajność działania biblioteki. Jednakże umieszczenie mutex-a w każdym obiekcie ma pewien narzut pamięciowy i wydajnościowy.

Na szczęście istnieje lepsze rozwiązanie tego problemu możliwe dzięki funkcji z biblioteki Boost Threads `call_once` gwarantującej jednokrotne wywołanie funkcji. Przekazanie tej funkcji zmiennej typu `once_flag` wraz z funkcją do wykonania (w tym przypadku funkcją `deferred_expr::evaluate`) sprawi, że dla danej flagi funkcja wykona się tylko raz. Wywołanie `call_once` z innego wątku lub ponownie z tego samego 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 mutex-ów.

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 nieznannej 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 już zaczyna przypominać mechanizm 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 specjalnego mechanizmu, 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` modelu obliczania wyrażeń opartego o wątki działające w ramach programu zlecają-

cego obliczenia.

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 stosunku do standardowych bibliotek oferujących wielowątkowość. Najważniejszym elementem procesu zwracania wyników przez bibliotekę `Parallel` jest wzorzec typu `deferred_value`. 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` nie posiadają wartości, a dokładnie mają wartość nieokreśloną. Dopiero po tym wyrażenie zapisane w skojarzonym obiekcie `deferred_expr` zostanie obliczone, wartość jest przekazywana do obiektu `deferred_value` w celu jej zapamiętania. Ponieważ jako jedyny obiekt `deferred_expr` ma prawo dokonania takiego przypisania, nie jest konieczna ochrona przed współbieżnym zapisem do obiektu `deferred_value`.

Wymuszenie wyliczenia wyrażenia

Obiekt typu `deferred_value` pozwala w pewnym stopniu na kontrolowanie wyliczenia wyrażenia przez programistę. Mianowicie, wywołanie metody `force` powoduje wymuszenie wyliczenia wyrażenia i metoda nie zakończy się dopóki to się nie wykona. Zatem programista jeśli tego potrzebuje, może uzyskać pewność, że ewaluacja wyrażenia została zakończona. Ponadto kolejną możliwością wymuszenia wyliczenia wyrażenia jest pobranie wartości zwróconej przez wyrażenie.

Pobranie wartości

Obiekt typu `deferred_value` pozwala na pobranie wartości metodą `get_value` lub też pobranie wartości może odbyć się niejawnie, ponieważ obiekt typu `deferred_value` ma zdefiniowany 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 w pełni 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

przeładowujących operatory dla wzorca typu `deferred_value`. Ta kwestia zostanie wyjaśniona w następnej sekcji.

2.4.2. Przeładowanie operatorów szablonu typu `deferred_value`

Motywacja dla przeładowywania 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 odpowiedzenie na pytanie “Po co programista używa (będzie używał) biblioteki `Parallel`?”, która brzmi “Aby możliwe *jak największą* część obliczeń w prosty sposób zrównoleglić”. Podkreśliłem słowa “jak największą” nie bez przyczyny, 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 wymagane 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 na szczęście 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 przeładować operator dodawania, w taki sposób, aby wyrażenie `d + 42` zwracało wartość odroczoną, zawierającą ułoniwione 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 przeładowywania operatorów

Język C++ pozwala na przeładowanie wszystkich operatorów, oprócz `::`, `.` oraz `.*`. Nie dla wszystkich pozostałych operatorów przeładowywanie ich dla obiektów `deferred_value` ma sens. Przededefiniowanie operatorów `new`, `new []`, `delete` oraz `delete []` nie było potrzebne. Ponadto domyślne zachowanie operatora `,` jest odpowiednie dla typu `deferred_value`. Pozostałe operatory zostały przededefiniowane 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. Przededefiniowania operatorów zwracają jako wynik obiekty `deferred_value`,

z szablonem wyrażenia, który po obliczeniu pozwoli uzyskać pożądany wynik. Oto pełna lista przeładowanych operatorów:

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

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 jaką wyrażenie zwraca, nie można również zadeklarować zmiennej typu `void`. Dlatego inaczej powinna wyglądać obiekt typu `deferred_value`, dla którego typem zwracanym jest typ `void`. 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 można byłoby rozważyć 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 wywoła metodę `force` na obiekcie `deferred_value`, która wymusi wykonanie sortowania, jeśli nie zostało rozpoczęte i poczeka do jego zakończenia. Po powrocie z funkcji `force` programista może bez obaw używać elementów tablicy `a`.

Wyrażenie zwracające referencję

Problematyczne okazuje się również, gdy wartością zwracaną jest referencja do obiektu, a nie sam obiekt. Przykładem takiego wyrażenia jest:

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

W tym przypadku problemem jest inicjalizacja składowej `m_val` 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 miałyby 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 często wywoływałoby błędy w kompilacji, gdyż referencje do obiektów często stosuje się, gdy obiektu nie można kopiować. Niestety zastosowanie wzorca klasy `boost::reference_wrapper` okazało się niemożliwe, gdyż on również musi zostać zainicjalizowany przez podanie referencji do obiektu.

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 oparte na tym stwierdzeniu świetnie sprawdziło się w praktyce. Wskaźniki C++ mają bowiem taką istotną różnicę w stosunku do referencji, że można je przestawić na inny obiekt, więc nie muszą być inicjalizowane w konstruktorze. Stąd składowa `deferred_value`, która ma przechowywać wynik obliczenia wyrażenia jest wskaźnikiem, natomiast obiekt `deferred_value` udostępnia interfejs, który w pełni oddaje fakt, że reprezentuje referencję.

2.4.4. Obsługa wyjątków

Jednym 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. Programista może sprawdzić czy podczas wykonania wyrażenia wystąpił wyjątek przy pomocy metody `has_exception`. Ponadto pobrany wyjątek można pobrać używając funkcji `get_exception`. Natomiast w przypadku użycia obiektu `deferred_value`, w którym zamiast wartości jest przechowywany wyjątek to ten wyjątek zostaje ponownie rzucony i powinien zostać przechwycony oraz obsłużony w kodzie programu.

Rozdział 3

Podsumowanie

Stworzona w ramach pracy biblioteka Parallel spełnia cele zdefiniowane w rozdziale Koncepcja biblioteki, które kierowały jej projektowaniem. Biblioteka Parallel pozwala na zwiększenie produktywności programisty piszącego programy równoległe. Było to możliwe dzięki zaprojektowaniu takiego modelu biblioteki, który pozwala na pisanie czytelnego i intuicyjnego kodu przy pomocy API biblioteki. Równie istotnym priorytetem podczas tworzenia biblioteki było uzyskanie wysokiej efektywności pisanych przy pomocy Parallel programów. Liczba operacji wykonywanych przez bibliotekę została ograniczona do niezbędnego minimum. Bardzo ważnym czynnikiem osiągnięcia wysokiej efektywności było zapewnienie programiście wglądu w to jaki sposób biblioteka działa. Udało się to osiągnąć, gdyż programista zlecając równoległe obliczenie wyrażenie wie w jaki sposób zostanie ono wykonane przez bibliotekę Parallel, dzięki czemu programista może dostosować podział zadań w taki sposób, aby zmaksymalizować wydajność programu.

Ułatwienie pracy programiście podczas korzystania z biblioteki Parallel polega również na tym, że biblioteka zdejmuje z niego obciążenie związane z projektowaniem i programowaniem 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, które posiadają kompilator języka C++ zgodny z przyszłym standardem C++0x oraz wspierają bibliotekę Boost.

W czasie pracy nad biblioteką zostało rozwiązanych kilka problemów badawczych, niektóre z nich okazały się bardzo wymagające. Najtrudniejsze okazało się zaprojektowanie sposobu przekazywania wyrażeń do obliczenia. Problem został rozwiązany przy pomocy idiomu C++ szablonu wyrażenia, a zaimplementowany 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 został oparty na autorskim pomysśle polegającym na wprowadzeniu wartości odroczonej. Rozwiązano również kilka pomniejszych problemów takich jak obsługa wyjątków wywołanych w trakcie równoległych obliczeń, uniknięcie zdublowanego obliczenia wyrażenia przekazanego do obliczenia, dostosowanie wartości odroczonej do specjalnych postaci wartości zwracanych, odroczenie wymuszenia obliczenia wartości odroczonej do jak najpóźniejszego momentu.

W implementacji biblioteki użyto wielu zaawansowanych technik programowania w języku C++. Były to między innymi obliczenia na typach w oparciu o wzorce, specjalizacja wzorców, inicjowanie przy pozyskiwaniu zasobu (z ang. Resource Acquisition Is Initialization), inteligentne wskaźniki, referencje do r-wartości i konstruktor przenoszący (z ang. move constructor). Tam, gdzie było to potrzebne i możliwe wykorzystane zostały biblioteki z

kolekcji Boost poważane ze względu na ich wysoką jakość oraz będące wzorcem nowoczesnego programowania w C++. Zastosowano biblioteki Boost.Proto, Boost.Threads, Boost.Bind, Boost.TypeTraits, Boost.SmartPointers, Boost.MPL, Boost.Ref, a do testowania biblioteki Parallel została wykorzystana biblioteka Boost.UnitTestFramework.

Podsumowując, stworzenie biblioteki Parallel zostało poprzedzone szeroką analizą teoretyczną, jak również zostały uwzględnione względy praktycznego stosowania biblioteki. Parallel jest dobrym kandydatem, aby stać się biblioteką szeroko używaną przez programistów języka C++ piszących programy równoległe.

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++*, Addison-Wesley, 2003.

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

[BerLand] Krste Asanović, 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 Research: 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>.

[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.htm>.

- [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-parallel-programming-146487.pdf>
- [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.