Wykorzystanie obliczeń równoległych w eksploracji danych na przykładzie algorytmów wyszukiwania zbiorów częstych

Marek Piotr Puścian
Politechnika Warszawska
mpuscian@elka.pw.edu.pl

Streszczenie

Gigantyczne rozmiary współczesnych kolekcji danych skutecznie uniemożliwiają jakiekolwiek próby ręcznej analizy zgromadzonych informacji. W ostatnich latach zaproponowano wiele metod i technik automatycznego lub półautomatycznego odkrywania wiedzy. Pomimo tego, wydajność systemów komputerowych nie wzrasta wystarczająco szybko, aby zaspokoić potrzeby aplikacji wykorzystywanych do drążenia danych. Prognozy dotyczące wzrostu ilości danych opublikowane w raporcie IDC oraz ograniczenie wzrostu wydajności maszyn jednoprocesorowych dają wyraźne powody do zastanowienia nad alternatywnymi metodami analizowania danych. Jedną z takich metod są obliczenia równoległe, które przez długi czas traktowane były jako egzotyczna dziedzina informatyki stosowana wyłącznie przez środowiska akademickie. W poniższym artykule zaprezentowano najpopularniejsze równoległe algorytmy wyszukiwania zbiorów częstych wykorzystywane przy odkrywaniu reguł asocjacyjnych. Ponadto, został w nim opisany nowoczesny paradygmat programowania równoległego Charm++, który można wykorzystać do implementacji przedstawionych algorytmów.

1. Problem analizy pozyskanych danych

Większość repozytoriów informacji wykorzystywanych komercyjnie lub naukowo zawiera użyteczną wiedzę ukrytą w danych pod postacią trendów, regularności, korelacji lub osobliwości. Niestety gigantyczne rozmiary współczesnych kolekcji danych skutecznie uniemożliwiają jakiekolwiek próby ręcznej analizy zgromadzonych informacji. Dla przykładu, baza danych wykorzystywana przez sieć sprzedaży Wal-Mart gromadzi dziennie informacje o ponad 20 milionach transakcji. System satelitarnej obserwacji EOS, który został zbudowany przez NASA, generuje w ciągu każdej godziny dziesiątki gigabajtów danych obrazowych. Nawet niewielkie supermarkety rejestrują codziennie sprzedaż tysięcy artykułów. Nasze możliwości analizowania i rozumienia tak dużych wolumenów danych są dużo mniejsze od możliwości ich zbierania i przechowywania. Problem analizy gromadzonych danych jest poważny i ciągle narasta. Zgodnie z raportem opublikowanym przez firmę IDC („The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011") ilość informacji przechowywanej elektronicznie jest większa i rośnie szybciej niż przewidywano. Wyniki przeprowadzonych badań pokazują, że do roku 2011 wszechświat danych cyfrowych osiągnie wielkość prawie 1,8 ZB (zattabajtów, 1 ZB = 1021 EB), co oznacza 10-krotny wzrost w ciągu 5 lat. Ponad 30 % z tej ilości będzie generowana i przechowywana przez organizacje, przedsiębiorstwa związane z różnymi dziedzinami przemysłu oraz ośrodki naukowe, natomiast pozostała część będzie efektem działalności osób prywatnych.

Właściciel	Zawartość	Rozmiar
World Data Centre for Climate	dane klimatyczne	6 PB + 220 TB
National Energy Research Scientific Computing Center	wyniki badań dotyczących fizyki wielkich energii	2,8 PB
AT&T	dane telekomunikacyjne	312 TB

Tabela 1. Przykłady największych baz danych (dane z 2007).

Odpowiedzią na gwałtowny wzrost ilości informacji gromadzonych w bazach i magazynach danych jest intensywnie rozwijająca się dziedzina odkrywania wiedzy (ang. *Knowledge Discovery*) oraz eksploracji danych (ang. *Data Mining*). Techniki eksploracji danych pozwalają na znajdowanie wcześniej nieznanych zależności i schematów, które mogą być wykorzystane do wspomagania podejmowania decyzji lub opisu bazy danych. Projektowane algorytmy umożliwiają automatyczne lub półautomatyczne przetwarzanie

zgromadzonych woluminów, dzięki czemu użytkownik nie musi samodzielnie przeszukiwać i analizować danych. Pomimo tego, wydajność systemów komputerowych nie wzrasta wystarczająco szybko, aby zaspokoić potrzeby aplikacji wykorzystywanych do drążenia danych. Aktualne trendy pokazują, że moc obliczeniowa systemów komputerowych wzrasta o 10-15 % w ciągu roku, a ilość zgromadzonych danych w tym samym czasie podwaja się. Sekwencyjne algorytmy eksploracji danych już przy obecnych ilościach informacji mogą nie dać wyników w rozsądnym czasie.
Pojawiają się również inne problemy - ilość pamięci dostępna w ramach jednego procesora może nie być wystarczająca do przechowania danych pośrednich, które są wymagane przez algorytm. Poza tym, sekwencyjne algorytmy drążenia danych nie nadają się do analizowania problemów wielkiej skali lub wymagają podziału obliczeń na części i częstego przenoszenia danych pośrednich poza obszar bezpośredniego dostępu jednostki obliczeniowej (np. na twardy dysk), a to znacząco spowalnia przetwarzanie.
Prognozy opublikowane w raporcie IDC oraz ograniczenie wzrostu wydajności maszyn jednoprocesorowych dają wyraźne powody do zastanowienia nad innymi metodami zapewniającymi efektywne analizowanie danych.

2. Współczesne zastosowania obliczeń równoległych

Obliczenia równoległe przez długi czas traktowane były jako egzotyczna dziedzina informatyki - interesująca, ale nie mająca praktycznego zastosowania i niedostępna dla zwykłych programistów. Sytuacja powoli zaczyna się zmieniać - analiza rozwoju współczesnych programów, architektur komputerów i sieci dowodzi, że równoległość staje się wszechobecna. Na przykład:
- Intel, czołowy producent procesorów, potwierdził, że w drugiej połowie bieżącego roku na rynku pojawią się sześciordzeniowe jednostki z rodziny Dunnington, a na początku 2010 światło dzienne ujrzy pierwszy procesor 32-rdzeniowy.
- w połowie 2007 producent układów graficznych nVidia wypuścił na rynek jednostki oraz komputery dedykowane dla złożonych, równoległych obliczeń naukowych (m.in. serwery wyposażone w 8 układów z rodziny TESLA)
Programowanie równoległe powoli zaczyna odgrywać istotną rolę przy tworzeniu oprogramowania dla komputerów. W chwili obecnej komputery o dużej mocy są wykorzystywane nie tylko do modelowania złożonych zjawisk i problemów takich, jak np. pogoda, obwody elektryczne i elektroniczne, procesy technologiczne, reakcje chemiczne czy procesy biologiczne. Coraz częściej obliczenia równoległe są wykorzystywane w aplikacjach komercyjnych wymagających zdolności do operowania wielkimi zbiorami danych. Wśród nich można wymienić aplikacje wspomagające komputerową diagnostykę medyczną, bazy danych w systemach automatyzacji procesów decyzyjnych, jak również aplikacje służące do wizualizacji danych oraz symulujące wirtualną rzeczywistość. Coraz głośniej mówi się również o wspólnym przedsięwzięciu firmy Intel i producenta oprogramowania Microsoft, którego celem jest stworzenie warunków do rozwoju oprogramowania korzystającego z wielordzeniowych procesorów.

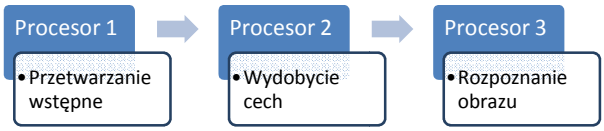
3. Obliczenia równoległe w eksploracji danych

Na przełomie ostatnich kilku lat można zaobserwować wzrost zainteresowania równoległymi algorytmami drążenia danych. W środowisku równoległym można bowiem inaczej spojrzeć na ograniczenia i problemy, z którymi borykają się algorytmy sekwencyjne. Dzięki wykorzystaniu mocy obliczeniowej n-równoległych procesorów czas wykonania może zostać skrócony nawet do n razy, a ograniczenia związane z ilością pamięci operacyjnej nie stanowią żadnego problemu – każdy z n procesorów może posiadać własną pamięć RAM o rozmiarze, który był dostępny dla algorytmu sekwencyjnego.
Warto jednak zauważyć, że zrównoleglenie istniejących algorytmów, prowadzące do uzyskania dobrej wydajności i skalowalności w przypadku ogromnych zbiorów danych, nie jest zadaniem trywialnym.
Pierwszym pojawiającym się problem jest zaprojektowanie właściwej organizacji danych oraz strategii dekompozycji zadań. Przy zrównoleglaniu algorytmów dąży się do równomiernego obciążenia wszystkich jednostek obliczeniowych przy jednoczesnym zachowaniu minimalnych zależności danych pomiędzy nimi. Ponadto, należy zminimalizować narzuty związane z synchronizacją oraz komunikacją – pozwoli to osiągną dobrą skalowalność algorytmu przy zwiększaniu liczby jednostek obliczeniowych (procesów). Aby efektywnie korzystać z mocy dostępnych procesorów, trzeba również zaprojektować mechanizm równoważenia obciążeń. Innym problemem, który musi zostać rozwiązany, jest minimalizacja operacji zapisu/odczytu danych z wolnych nośników pamięci, tj. dyski twarde.

4. Podstawowe zagadnienia związane z przetwarzaniem równoległym

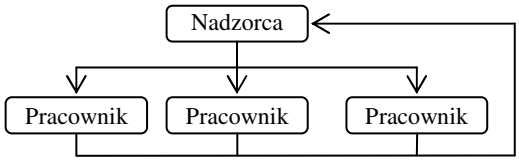
Najważniejszym etapem projektowania algorytmu równoległego jest dekompozycja na podzadania nazywane ziarnami (ang. *seeds*). Rozwiązywany problem może być podzielny na wiele różnych sposobów, np. podzadania mogą mieć taki sam albo różny rozmiar.

Niezależnie od sposobu podziału należy dążyć do tego, aby ziarna były od siebie nie zależne lub słabo zależne.
Jednym z parametrów określających strategię podziału jest ziarnistość dekompozycji. Jeżeli problem zostanie podzielony na dużą liczbę niewielkich zadań, mamy do czynienia z dekompozycją drobnoziarnistą (ang. *fine grained*). Natomiast o dekompozycji gruboziarnistej (ang. *coarse grained*) mówimy, gdy problem zostanie podzielony na niewielką liczbę stosunkowo dużych zadań. Inną miarą wykorzystywaną w trakcie projektowania algorytmów równoległych jest stopień współbieżności (ang. *degree of concurency*). Określa ona liczbę zadań, które mogą być wykonywane równolegle. Stopień współbieżności może zmieniać się w trakcie wykonywania programu, dlatego podawana jest jego maksymalna oraz średnia wartość. Wpływ na wartość miary ma m.in. ziarnistość dekompozycji. Stopień współbieżności zwiększa się, gdy podział staje się bardziej drobnoziarnisty i na odwrót.
Zwiększanie stopnia współbieżności nie zawsze prowadzi do szybszego wykonania programu. Wraz ze wzrostem liczby zadań zwiększa się również zakres interakcji pomiędzy nimi. Zadania mogą współdzielić dane wejściowe, wyjściowe lub pośrednie, np. rezultat wykonania jednego zadania jest danymi wejściowymi drugiego. Narzuty związane z komunikacją i synchronizacją pomiędzy równoległymi ziarnami mogą znacznie ograniczyć efektywność takiego przetwarzania.
Innym istotnym ograniczeniem jest maksymalna liczba zadań, na które można zdekomponować problem. Zwykle istnieje ograniczenie związane z zasadą działania algorytmu lub danymi, które wyznacza górną granicę stopnia współbieżności. Na przykład, maksymalna liczba zadań przy mnożeniu macierzy $n^x n$ i wektora n-elementowego jest równa $O(n^2)$, ponieważ wykonanie takiej operacji wymaga n^2 mnożeń i dodawań.
W zależności od rozwiązywanego zadania wykorzystuje się:
- dekompozycję funkcjonalną – najpopularniejsza technika polegająca na podziale algorytmu na niezależne grupy operacji, które będą wykonywane na osobnych jednostkach obliczeniowych. Można wyróżnić kilka rodzajów dekompozycji funkcjonalnej, np.:
  - wg idei przetwarzania taśmowego zadanie zostaje rozbite na bloki operacji, które dla określonych danych muszą być wykonywane sekwencyjne. Każdy procesor wykonuje jeden blok operacji na różnych danych – przetworzone dane są albo wejściem dla następnego bloku operacji albo wynikiem końcowym. Idea przetwarzania taśmowego jest stosowana np. przy rozpoznawaniu obrazów
  - wg idei farmy zadań - jeden z procesorów pełni funkcję nadzorcy, który zarządza pozostałymi procesorami (pracownikami). Głównym zadaniem nadzorcy jest rozdzielenie zadań oraz danych pomiędzy pracownikami. Po wykonaniu zadania pracownicy zwracają rezultat do kolektora wyników (jego funkcję zwykle pełni nadzorca)
- dekompozycję danych - technika powszechnie używana przy rozwiązywaniu problemów operujących na dużych ilościach danych. Zwykle składa się z dwóch etapów: podziału danych, na których przeprowadzane są obliczenia, podziału obliczeń w oparciu o podział danych. Dekompozycja może dotyczyć danych wejściowych, pośrednich lub wyjściowych.
- dekompozycję eksploracyjną – technika wykorzystywana w przypadku problemów, których rozwiązanie wymaga przeszukania pewnej przestrzeni rozwiązań (ang. *solution space*) np. optymalizacja dyskretna, dowodzenie twierdzeń, rozgrywanie gier. Polega ona po podziale przestrzeni rozwiązań na części, które będą niezależnie przeszukiwanie do momentu znalezienia rozwiązania.
- dekompozycję rekurencyjną – stosowana dla problemów, które mogą być rozwiązane przy pomocy strategii „dziel i zwyciężaj” (ang. *divide and conquer*). Technika ta opiera się na rekurencyjnym podziale problemu na coraz to mniejsze, niezależne zadania (do momentu uzyskania pożądanej ziarnistości).
- dekompozycję hybrydową



Rysunek 1. Idea przetwarzania taśmowego

Decydujący wpływ na wydajność algorytmu równoległego ma alokacja zadań . Dobrze zaprojektowany program równoległy powinien zapewnić efektywne wykorzystanie wszystkich procesorów (czyli tzw. równoważenie obciążeń ang. *load-blancing*) i zawierać tylko niezbędną komunikację między ziarnami. Na ogół liczba zadań jest znacznie większa niż liczba dostępnych procesorów. W związku z tym należy przygotować strategię przydziału w oparciu o wiedzę na temat środowiska, w którym będzie wykonywany program. Podczas konstrukcji odwzorowania (ang. *mapping*) trzeba brać pod uwagę zarówno zależności, jak i interakcje pomiędzy zadaniami, np. należy dążyć do tego, aby żaden proces nie oczekiwał bezczynnie na dane. Zadania, które są od siebie niezależne należy przydzielić różnym procesorom, natomiast przy alokacji zadań, które wymagają częstej interakcji musimy brać pod uwagę narzuty związane z komunikacją. Zwykle takie zadania umieszcza się na jednym procesorze lub na osobnych węzłach obliczeniowych posiadających pomiędzy sobą szybkie połączenie sieciowe.



Rysunek 2. Idea farmy zadań

Metody alokacji zadań możemy podzielić na statyczne i dynamiczne. W metodach statycznych odwzorowanie zadań na procesory odbywa się przed uruchomieniem programu. W związku z tym przed rozpoczęciem obliczeń trzeba posiadać wiedzę na temat środowiska wykonawczego oraz przygotować podział zadań i danych. Takie ograniczenia nie występują w przypadku metod dynamicznych. Umożliwiają one alokowanie zadań podczas pracy programu oraz pozwalają na ich swobodną migrację w celu efektywnego wykorzystania dostępnych zasobów obliczeniowych. Dzięki temu przed uruchomieniem programu nie musimy przygotowywać podziału oraz odwzorowania na procesory.

5. Równoległe odkrywanie reguł asocjacyjnych

Odkrywanie reguł asocjacyjnych (ang. *association rule discovery*) jest jedną z najważniejszych technik eksploracji danych polegającą na znajdowaniu związków pomiędzy występowaniem grup elementów w zadanych zbiorach danych. Znalezione korelacje są prezentowane jako reguły postaci $X \Rightarrow Y$, gdzie X i Y są rozłącznymi zbiorami elementów. Z każdą regułą postaci $X \Rightarrow Y$ związane są dwa parametry: wsparcie - liczba wystąpień zbioru elementów $X \cup Y$ w kolekcji danych oraz zaufanie – prawdopodobieństwo wystąpienia zbioru Y, jeśli w kolekcji wystąpił zbiór X (czyli $P(Y|X)$). Reguła asocjacyjna jest uważana za silną regułę asocjacyjną, jeżeli jej wsparcie i zaufanie jest większe od progów zadanych przez użytkownika.
Najpopularniejszym przykładem odkrywania asocjacji jest tzw. analiza koszyka - przetwarzanie baz danych supermarketów i hurtowni w celu znalezienia grup towarów, które są najczęściej kupowane wspólnie. Przykładowo, znalezione asocjacje mogą wskazywać, że kiedy klient kupuje kiełbasę i chleb, wtedy kupuje także musztardę.

Id	Pozycje
1	kiełbasa, chleb, musztarda
2	ogórki, pomarańcze, sól
3	ogórki, kiełbasa, chleb
4	kiełbasa, chleb, musztarda
5	paluszki, pomarańcze

$\{\text{kiełbasa,chleb}\} \Rightarrow \{\text{musztarda}\}$

Tabela 2. Analiza koszyka zakupów - przykład wyszukanej reguły asocjacyjnej.

Odkryte reguły asocjacyjne mogą być wykorzystane do np.: organizowania promocji i sprzedaży wiązanej, konstruowania katalogów wysyłkowych, ustalania rozmieszczenia towarów na półkach
Najważniejszym etapem znajdowania reguł asocjacyjnych jest wyszukiwanie zbiorów częstych. Wśród najpopularniejszych i najbardziej wydajnych algorytmów można wymienić:
- ***Apriori*** - Rakesh Agrawal, John C. Shafer
- ***Eclat*** - Mohammed J. Zaki, Mitsunori Ogihara, Wei Li
- ***FP-growth*** - Jiawei Han, Jian Pei, Yiwen Yin
- ***D-Club*** - Jianwei Li, Alok Choudhary, Nan Jiang

Dla każdego z powyższych algorytmów przygotowano wersję równoległą.

6. Algorytmy z rodziny Apriori

Większość równoległych metod wykrywania reguła asocjacyjnych bazuje na algorytmie Apriori. Technika zaproponowana m.in. przez Pakesh'a Agrawal'a, polega na iteracyjnym generowaniu i testowaniu kandydatów na zbiory częste o długości od 1 do k do momentu, aż znalezione zostaną wszystkie zbiory częste. Kandydaci na zbiory częste są generowani w taki sam sposób jak w algorytmie Apriori - sklejane są takie zbiory częste, które po usunięciu ostatnich elementów są identyczne.
Algorytmy równoległe należące do rodziny Apriori można podzielić na następujące grupy:
- ***Count Distribution*** – algorytmy w tej grupie bazują na dekompozycji danych. Baza danych dzielona jest na statyczne, poziomie partycje, w których niezależne procesy zliczają wsparcie dla kandydatów na zbiory częste. Pod koniec każdej iteracji, procesy wymieniają się licznikami swoich kandydatów, tak aby każdy posiadał globalny stan wyszukiwania.
- ***Data Distribution*** – algorytmy należące do tej grupy dążą do lepszego wykorzystania pamięci. Podobnie jak w poprzedniej grupie baza danych jest dzielona na partycje. Najważniejszą różnicą jest rozdzielenie listy wyszukiwanych kandydatów pomiędzy procesory. Każdy kandydat jest wyszukiwany przez tylko jeden proces, dlatego procesy muszą „wymieniać się" partycjami w czasie każdej iteracji (wspólne wykorzystanie podzielonych danych)
- ***Candidate Distribution*** **–** algorytmy w tej grupie również rozdzielają listę wyszukiwanych kandydatów. W odróżnieniu od techniki *Data Distribution*, zamiast podziału i wymiany partycji stosowana jest replikacja części transakcji z bazy danych na węzłach obliczeniowych. Dzięki temu obliczenia mogą być prowadzone niezależnie.

Wyniki przeprowadzonych danych pokazują, że algorytmy z grupy ***Count Distribution*** wykazują najlepszą wydajność spośród wszystkich algorytmów bazujących na Apriori. Algorytm *Count Distribution* przeznaczony dla maszyny wieloprocesorowej z pamięcią rozproszoną można podzielić na następujące kroki:
1.Baza danych jest dzielona na równe partycje w orientacji poziomej, które zostają przydzielone poszczególnym procesom
2.Każdy proces skanuje lokalną partycje w celu zliczenia występowania każdego elementu
3.Wszystkie procesy wymieniają i sumują lokalne liczniki, aby uzyskać globalną liczbę wystąpień każdego elementu , po czym wyszukują wszystkie zbiory częste o długości 1
4.Ustaw długość kandydata k = 2
5.Wszystkie procesy generują kandydatów o długości k ze zbiorów częstych o długości k – 1
6.Każdy proces skanuje lokalną partycje w celu policzenia wystąpień każdego zbioru elementów o długości k
7.Wszystkie procesy wymieniają i sumują lokalne liczniki, aby uzyskać globalną liczbę wystąpień każdego zbioru elementów o długości k, po czym wyszukują wszystkie k-zbiory częste.
8.Powtarzaj kroki (5) – (8) z k := k + 1 do momentu, aż nie znaleziony zostanie żaden zbiór częsty

TID	Pozycje	Proces
1	f d b e	P0
2	f e b	
3	a d b	P1
4	a e f c	
5	a d e	P2
6	a c f e	

Tabela 3. Przykładowy podział bazy danych (Apriori)

Element	Lokalny licznik			Globalny licznik
	P0	P1	P1	
a	0	2	2	4
b	2	1	0	3
c	0	1	1	2
d	1	1	1	3
e	2	1	2	5
f	2	1	1	4

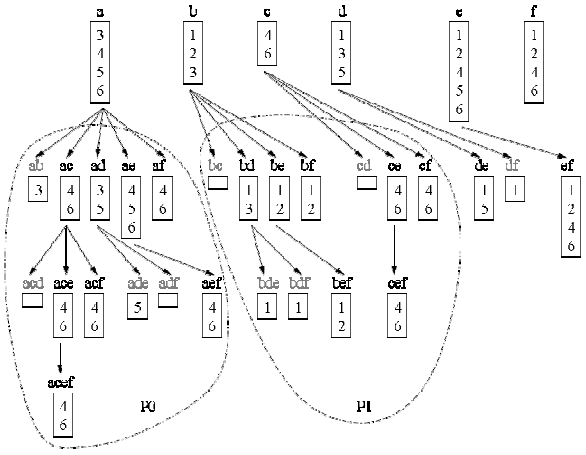
Tabela 4. Stan lokalnych i globalnych liczników po 1 iteracji algorytmu dla podziału z tabeli 3

Komunikacja pomiędzy procesami w algorytmach z rodziny Count Distribution jest ograniczona do wymiany liczników na końcu każdej iteracji. W pozostałych częściach algorytmu procesy działają niezależnie i wykorzystują lokalny fragment bazy danych. Niestety, przedstawiona metoda wyszukiwania zbiorów częstych nie jest pozbawiona wad. Pamięć węzłów obliczeniowych nie jest wykorzystywana efektywnie – kandydaci i zbiory częste są przechowywane na każdej jednostce wykonawczej. Ponadto praca związana z wygenerowaniem kandydatów i wyborem zbiorów częstych jest powtarzana na każdej jednostce (co może być kosztowne, jeżeli jest dużo takich zbiorów). W takich wypadkach, skalowalność będzie się pogarszać wraz ze wzrostem liczby procesów przetwarzających. Nie można również uniknąć kosztu związanego z odczytem bazy danych oraz synchronizacją pomiędzy procesami w każdej iteracji.

7. Parallel Eclat

Równoległy Eclat (ang. *ParallelEclat, ParEclat*) jest zrównoleglonym odpowiednikiem popularnego algorytmu Eclat. Technika zaproponowana m.in. przez Mohammed'a Zaki'ego charakteryzuje się lepszym wykorzystaniem zasobów komputerowych oraz doskonałą skalowalnością. Wykorzystano w niej pionową orientację danych – zamiast bazy zawierającej rekordy typu {id transakcji, lista pozycji} używane są posortowane listy identyfikatorów transakcji (w skrócie, tid-listy), w których występuje dana pozycja (lub zbiór elementów). Zbiory częste o długości k są zorganizowane w rozłączne klasy równoważności (zbiory posiadające identyczne k-1 elementowe prefiksy). Dzięki temu kandydaci o długości k+1 mogą być generowani poprzez połączenie dwóch częstych k-zbiorów (tzn. zbiorów o długości k) z tej samej klasy. Wsparcie kandydującego zbioru obliczamy poprzez wyznaczenie części wspólnej tid-list podzbiorów, z których go utworzono.
Zrównoleglenie algorytmu uzyskujemy poprzez rozdzielenie zadań generacji i weryfikacji kandydatów dla poszczególnych klas równoważności pomiędzy dostępne procesy. Klasy równoważności wszystkich częstych 2-zbiorów zostają przypisane do wszystkich dostępnych procesorów. W kolejnym kroku pomiędzy procesorami rozdystrybuowane zostają odpowiednie tid-listy potrzebne do wykonania obliczeń. Następnie, każdy proces niezależnie wyszukuje zbiory częste spośród wszystkich zbiorów wygenerowanych za pomocą przypisanych mu klas równoważności. Algorytm dla maszyny wieloprocesorowej z pamięcią rozproszoną można podzielić na następujące kroki:
1.Baza danych zostaje podzielona na równe partycje w orientacji poziomej, które zostają przydzielone każdemu procesowi
2.Każdy proces odczytuje swoją lokalną partycję w celu zliczenia wystąpień zbiorów o długości 1 oraz 2
3.Wszystkie procesy wymieniają i sumują lokalne liczniki, aby uzyskać globalną liczbę wystąpień każdego

1-zbioru i 2-zbioru po czym wyszukują pośród nich wszystkie zbiory częste
4.W oparciu o uzyskane dane utworzone zostają klasy równoważności dla wszystkich częstych 2-zbiorów
5.Wytworzone klasy równoważności zostają przypisane poszczególnym procesom
6.Każdy proces przekształca swoją lokalną część danych do orientacji pionowej (tid-listy) dla wszystkich częstych 2-zbiorów
7.Każdy proces wymienia lokalne tid-listy z innymi procesami, aby otrzymać ich globalną postać dla przypisanych klas równoważności
Dla każdej przypisanej klasy równoważności na każdej jednostce wykonawczej wykonywane jest rekursywne wykrywanie zbiorów częstych (każde dwie dwójki zbiorów z tej samej klasy równoważności są łączone i wyznaczana część wspólna tid-list)



Rysunek 3. Przykład rozdziału klas równoważności w algorytmie Eclat

Istnieją również inne odmiany zrównoleglonego Eclat'a, które różnią się organizacją oraz strategiami przeszukiwania danych. Wśród nich można wyróżnić: ParMaxEclat , ParClique, ParMaxClique
W porównaniu do algorytmów Apriori, w metodzie zaproponowanej przez Zaki'ego nastąpiła znacząca redukcja narzutu związanego z operacjami we/wy. ParEclat wykonuje trzy odczyty z bazy danych, natomiast algorytmy z rodziny Apriori muszą przeszukiwać bazę danych tyle razy, ile wynosi maksymalna długość zbioru częstego. Ponadto, już na początku wykonania algorytmu ParEclat uzyskujemy niezależność pomiędzy procesami. Dzięki czemu nie jest potrzebna komunikacja/synchronizacja w trakcie generacji kolejnych kandydatów na zbiory częste. Algorytm posiada również istotne wady**.** Główne narzuty związane z komunikacją wynikają z potrzeby ustalenia globalnych tid-list na początku algorytmu. W większości przypadków udaje się je zamortyzować w kolejnych iteracjach algorytmu. W trakcie wykonania algorytmu dla różnych danych może dochodzić do nierównego obciążenia procesorów. Aby tego uniknąć, liczba procesów powinna być dużo mniejsza niż liczba klas równoważności dla zbiorów częsty o długości 2. Aby osiągnąć dobrą wydajność algorytmu dla dowolnych danych może być konieczne przygotowanie strategii rozdziału klas równoważności oraz równoważenia obciążeń.

8. Implementacja algorytmów równoległych

Istnieje wiele narzędzi oraz języków programowania równoległego. Najbardziej popularne i wykorzystywane w praktycznych zastosowaniach są biblioteki MPI (ang. *Message Passage Interface*) oraz PVM (ang. *Parallel Virtual Machine*). Niestety, rozwiązania przygotowane za ich pomocą są mało elastyczne i wymagają dużo pracy. Interesującą alternatywą może okazać się Charm++.

9. Ogólna charakterystyka Charm++

Charm++ jest paradygmatem programowania równoległego opartym o asynchroniczną wymianę komunikatów i zorientowanym obiektowo. Przez paradygmat programowania należy tutaj rozumieć sposób pisania programu (czyli model programistyczny). Charm++ nie jest nie zależnym językiem programowania – językiem bazowym wykorzystywanym przy pisaniu programów jest C++. Udostępnia on programiście dodatkowe funkcje oraz warstwę abstrakcji, która ułatwia przygotowanie aplikacji równoległych.
Język oraz biblioteka wchodząca w skład systemu Charm++ powstała jako grupowy wysiłek osób pracujących w Laboratorium Programowania Równoległego Uniwersytetu Illinois w Urbana-Champaign. Projekt po raz pierwszy został zaprezentowany we wrześniu 1993 roku na konferencji OOPSLA (ang. conference on Object Oriented Programming, Systems, Languages and Application) i od tego czasu jest nieustannie rozwijany. Pracownicy laboratorium stale aktualizują bibliotekę, aby ta umożliwiała wykonywanie obliczeń na najnowszych maszynach, wspierała najnowsze paradygmaty programowania równoległego, jak również umożliwiała wygodne i szybkie przygotowanie wymagających aplikacji wykonujących przetwarzanie równoległe.

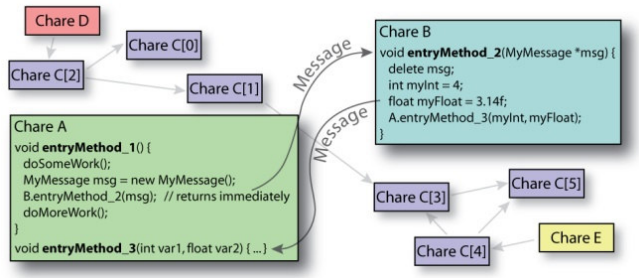
10. Środowisko sprzętowe Charm++

W chwili obecnej Charm++ wspiera następujące platformy sprzętowe: BlueGene/L, BlueGene/P, PSC Lemieux, IBM SP, IBM RS-6000 (AIX) SGI (IRIX 5.3 or 6.4), SGI Origin2000, Cray XT3/4, Cray X1, Cray T3E, pojedynczą stację roboczą lub sieć stacji roboczych firmy Sun Microsystems (Solaris), HP (HP-UX), Intel x86

(Linux, Windows 98/2000/XP), Intel IA64, Intel x86 64, wielordzeniowe x86 oraz x86 64, Apple Mac
Ponadto, programista może wykorzystać wsparcie dla komunikacji w następujących protokołach i sieciach komunikacyjnych: UDP, TCP, Myrinet, Infiniband, Quadrics Elan, Shmem, MPI, NCSA VMI

11. Aplikacja w Charm++

W ogólnym przypadku Charm++ umożliwia przygotowanie programów współbieżnych. Jednostki programowe mogą być wykonywane współbieżnie na jednym procesorze (podział czasu jednostki przetwarzającej pomiędzy wszystkie wątki) lub równolegle na wielu procesorach (wszystkie wątki są wykonywane jednocześnie).
Główną metodą komunikacji pomiędzy jednostkami programowymi jest asynchroniczna wymiana komunikatów. Operacja przekazania wiadomości jest nie blokująca – po wysłaniu komunikatu kontener równoległy (ang. chare) kontynuuje wykonanie swojego kodu. Charm++ umożliwia również synchroniczną wymianę komunikatów – nadawca wiadomości wstrzymuje wykonanie swojego kodu do momentu, gdy odbiorca wiadomości wykona jej przetwarzanie. W tym wypadku możliwe jest odesłanie wyniku przetwarzania (komunikatu zwrotnego).



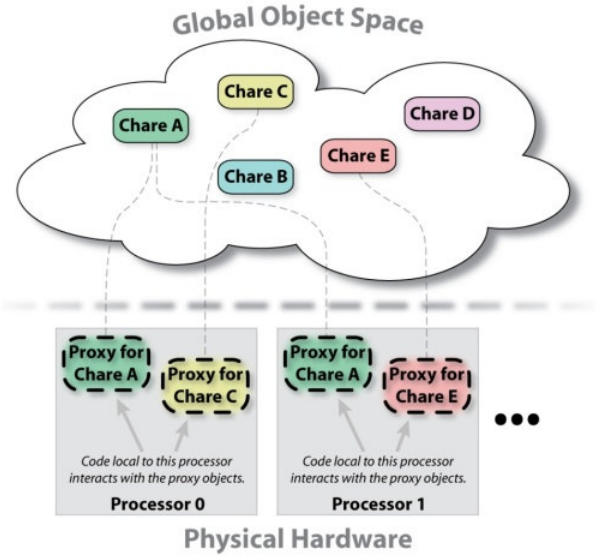
Rysunek 4. Aplikacja w Charm++ z punktu widzenia programisty

W trakcie przygotowania rozwiązania w Charm++ programista nie musi wiedzieć, ile jednostek wykonawczych posiada maszyna na której będzie uruchamiany program ani zajmować się przydzielaniem kontenerów równoległych do dostępnych procesorów. Przed programistą zostały również ukryte szczegóły związane z komunikacją pomiędzy poszczególnymi kontenerami. Za przydzielenie kontenerów do dostępnych, fizycznych jednostek wykonawczych oraz zapewnienie komunikacji niezależnie od fizycznej realizacji połączenia pomiędzy komputerami odpowiada system wykonawczy Charm++ (ang. Charm++ Runtime System). Głównym zadaniem programisty jest rozdzielenie zadań pomiędzy kontenerami zdefiniowanymi w aplikacji.
Każda instancja kontenera równoległego w Charm++ jest traktowana jako całkowicie niezależny obiekt. Kontener (podobnie jak instancja klasy) posiada pola przechowujące jego stan. Kontenery nie mają bezpośredniego dostępu do danych przechowywanych w innych kontenerach, natomiast mogą „bezpośrednio" komunikować się z innymi kontenerami. Zbiór wszystkich kontenerów zdefiniowanych w aplikacji Charm++ jest nazywany globalną przestrzenią obiektów (ang. global object space). Dowolny kontener należący do globalnej przestrzeni obiektów może zażądać jakieś informacji od innego kontenera z tej przestrzeni poprzez wysłanie odpowiedniego komunikatu. Każdy kontener równoległy jest instancją klasy kontenera. W klasie kontenera zawarta jest informacja o typach jego pól i metod składowych.

12. Komunikacja w Charm++

Kontenery komunikują się ze sobą poprzez wymianę komunikatów. W Charm++ proces ten jest nazywany zdalnym wywoływaniem metod (ang. remote method invocation), ponieważ kontener wysyłający wiadomość wywołuje jedną z metod wejściowych (ang. entry methods) kontenera odbierającego. Z punktu widzenia programisty jest to normalne wywołanie jednej z metod kontenera docelowego.
Metody wejściowe są specjalnymi składowymi klasy kontenera równoległego. Główna różnica pomiędzy zwykłą metodą w języku C++, a metodą wejściową Charm++ jest to, że metoda wejściowa pełni rolę punktu odbiorczego dla nadanych wiadomości (innymi słowy, metody wejściowe mogą być zdalnie wywoływane przez inne kontenery równoległe). Kiedy jeden z kontenerów wywoła metodę wejściową innego, zdalnego kontenera, dane przekazane jako argumenty metody są pakowane (formowane) do komunikatu, który zostanie przekazany do zdalnego kontenera. Każda klasa kontenera równoległego posiada również konstruktor zadeklarowany jako metoda wejściowa (w uproszczeniu, konstruktor kontenera Charm++ pełni identyczną rolę jak konstruktor klasy C++). Kiedy kontener równoległy jest tworzony, system wykonawczy Charm++ automatycznie wywołuje konstruktor zadeklarowany jako metoda wejściowa. Wykonanie programu w Charm++ zaczyna się od wywołania konstruktora kontenera głównego (ang. main chare) zadeklarowanego jako metoda wejściowa.
Istnieją dwie zasadnicze różnice pomiędzy zwykłymi metodami klas w C++, a metodami wejściowymi kontenerów Charm++. Metody wejściowe mają nie zwracać żadnej wartości (typ zwracany należy zadeklarować jako void). Ponadto, wywołania metod wejściowych są nie blokujące. Po wywołaniu metody wejściowej zdalnego kontenera następuje natychmiastowy

powrót do wykonania kodu kontenera wywołującego. Nie oznacza to jednak, że metoda wejściowa została wykonana na zdalnym kontenerze. Powrót z metody gwarantuje jedynie wysłanie komunikatu. Kiedy komunikat dotrze do zdalnego kontenera (czas jest zależny od medium transmisyjnego oraz „fizycznej" odległości od kontenera), zostanie przekazany do odpowiedniej metody wejściowej (metoda zostanie wywołana).



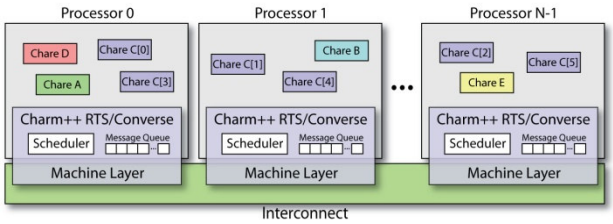
Rysunek 5. Lokalizacja kontenerów przy użyciu obiektów pośredniczących (ang. proxies)

Wszystkie kontenery równoległe należą do globalnej przestrzeni obiektów (ang. global objects space) i mogą być rozproszone pomiędzy różnymi jednostkami przetwarzającymi. W związku z tym bezpośrednia komunikacja pomiędzy dwoma kontenerami nie zawsze jest możliwa. Aby ukryć szczegóły związane z wysyłaniem wiadomości (Charm++ umożliwia komunikacje w różnych protokołach i mediach komunikacyjnych), w Charm++ zdefiniowano specjalne obiekty pośredniczące (ang. proxies), wykorzystywane w trakcie wykonywania operacji na kontenerach. Przed wywołaniem zdalnej metody wejściowej kontener wołający musi pobrać referencje do obiektu pośredniczącego kontenera wołanego. Następnie kontener wołający wywołuje metodę wejściową na obiekcie pośredniczącym kontenera wołanego (w taki sam sposób, jak gdyby kontener wołany był wykonywany na tym samym fizycznym procesorze co kontener wołający). System wykonawczy Charm++ zadba o zlokalizowanie właściwego kontenera w globalnej przestrzeni adresowej.
W większości aplikacji równoległych istnieje potrzeba wykorzystania co najmniej kilkunastu kontenerów, dlatego w Charm++ zdefiniowano kilka kolekcji pozwalających wygodnie na nich operować. Kontenery wchodzące w skład takich kolekcji wykonują identyczne operacje na różnych danych (ang. SIMD – single instructions, multiple data) i mogą być rozproszone pomiędzy fizycznymi procesorami. Każda kolekcja w programie Charm++ ma unikalny identyfikator nazywany uchwytem (ang. handle). Uchwyty kolekcji są dostępne z poziomu każdego procesora i mogą zostać wykorzystane do wykonania operacji na kontenerach wchodzących w skład danej kolekcji.

13. System wykonawczy Charm++

Kiedy programista pisze aplikacje w Charm++, zajmuje się jedynie zagadnieniami związanymi z rozdziałem kodu pomiędzy kontenerami równoległymi oraz wymianą informacji pomiędzy nimi opartą o zdalne wywoływanie metod (lub przekazywanie komunikatów). Szczegóły dotyczące środowiska wykonania takie jak liczba procesorów, ich typ, medium komunikacyjne oraz inne nie muszą (i nie są) brane pod uwagę. Programista przygotowuje aplikacje, w której pewna kolekcja obiektów współdziała ze sobą w celu wykonania określonego zadania. Taki sposób postrzegania aplikacji w Charm++ jest nazywany widokiem użytkownika lub programisty (ang. user's view of a Charm++ application).

Szczegóły związane ze środowiskiem wykonania zaczynają odgrywać role w momencie, kiedy aplikacja zostanie skompilowana. W czasie wykonania do dyspozycji będziemy mieli pewien zbiór zasobów fizycznych (tj. liczba procesorów). Głównym zdaniem systemu wykonawczego Charm++ (ang. Charm++ Runtime System) jest zarządzanie jak największą liczbą szczegółów związanych z fizycznymi zasobami w imieniu aplikacji oraz programisty. Taki sposób widzenia aplikacji w Charm++ jest określany jako widok systemowy (ang. system's view of a Charm++ application).



Rysunek 6 Aplikacja w Charm++ z punktu widzenia systemu

Wśród zadań, którymi w imieniu aplikacji zajmuje się system wykonawczy Charm++ można wyróżnić:
- rozmieszczanie kontenerów równoległych na fizycznych jednostkach wykonawczych – w systemie wykonawczym Charm++ zdefiniowano kilka strategii postępowania w trakcie przydziału.
- równoważenie obciążeń – system wykonawczy potrafi w trakcie wykonania aplikacji dynamicznie

przemieszczać kontenery równoległe pomiędzy fizycznymi procesorami. Dzięki temu można efektywnie wykorzystywać dostępną moc jednostek wykonawczych
- trasowanie wiadomości – kontenery równoległe mogą migrować pomiędzy procesorami, dlatego system wykonawczy Charm++ utrzymuje informacje o zmianach położenia. Dzięki temu wiadomości wysyłane do określonego kontenera są trasowane w locie (ang. dynamically routed), czyli kierowane do fizycznej jednostki wykonawczej, w której aktualnie znajduje się kontener. Wysyłanie wiadomości z punktu widzenia programisty jest realizowane tak samo - niezależnie od tego, czy mamy do czynienia z kontenerem lokalnym czy też kontenerem, który został przeniesiony do innego procesora.
- punkty kontrolne lub migawki (ang. checkpoints) – system wykonawczy Charm++ potrafi również utrwalić aktualny stan kontenerów równoległych – są one serializowane i zapisywane na dysku węzła obliczeniowego.
- tolerowanie błędów – jeżeli pojawią się jakiekolwiek problemy z fizycznym elementem obliczeniowym lub procesor ulegnie awarii, system wykonawczy potrafi dynamicznie odtworzyć stan kontenerów i rozpocząć dalsze wykonywanie ich kodu na pozostałych procesorach
- dynamiczna realokacja zasobów fizycznych – jeżeli klaster, który wykonuje aplikację Charm++ otrzyma nagle dodatkowe prace (lub gdy wykonanie dodatkowych prac zakończy się), system wykonawczy może dynamicznie przenieść kontenery równoległe z (lub do) fizycznych procesorów. Umożliwi to aplikacji dynamiczne korzystanie z większej lub mniejszej liczby procesorów w zależności od całkowitego obciążenia klastru.

Każda jednostka wykonawcza wykorzystywana w aplikacji ma uruchomiony system wykonawczy Charm++ (ang. Charm++ Runtime System). Każda instancja systemu wykonawczego jest odpowiedzialna za lokalne zasoby związane z jednostką na której został uruchomiony. Instancje systemu wykonawczego mogą również komunikować się między sobą w celu przeprowadzenia wspólnych operacji, tj. tworzenie migawek, odtwarzanie kontenerów po błędzie, równoważenie obciążeń jednostek wykonawczych.

14. Podsumowanie

Gwałtowny wzrost rozmiaru współczesnych baz danych skutecznie uniemożliwia próby ręcznej analizy zgromadzonych informacji. Chociaż w ostatnich latach zaproponowano wiele metod i technik automatycznego odkrywania wiedzy, wydajność systemów komputerowych nie wzrasta wystarczająco szybko, aby zaspokoić potrzeby aplikacji służących do drążenia danych. W niedalekiej przyszłości będzie konieczne zastanowienie nad alternatywnymi metodami analizowania danych. Jedną z nich mogą stać się obliczenia równoległe. Warto zauważyć, że w ostatnich latach zaproponowano wiele zrównoleglonych algorytmów związanych z eksploracją danych i odkrywaniem wiedzy. Wykorzystanie mocy wielu jednostek przetwarzających pozwoli skrócić czas obliczeń i obejść ograniczenia związane ilością pamięci RAM wykorzystywanej w czasie działania algorytmów. Ponadto, dzięki zastosowaniu nowoczesnych paradygmatów programowania, tj. jak Charm++, implementacja i uruchomienie takich algorytmów w dowolnym środowisku równoległym jest dziś dużo łatwiejsza.

15. Literatura

[1] Praca zbiorowa pod redakcją Andrzeja Karbowskiego i Ewy Niewiadomskiej-Szynkiewicz, „Obliczenia równoległe i rozproszone”, Oficyna Wydawnicza Politechniki Warszawskie, 2001.

[2] Jianwei Li, Ying Liu, Wei-keng Liao, Alok Choudhary „ Parallel Data Mining Algorithms for Association Rules and Clustering”, CRC Press, LLC, 2006.

[3] Mohammed J. Zaki. „Parallel and distributed association mining: A survey” *IEEE Concurrency, 7(4):14–25, 1999.*

[4] Mohammed J. Zaki, Mitsunori Ogihara, Srinivasan Parthasarathy, and Wei Li. „*Parallel data mining for association rules on shared-memory multi-processors*”. In *Proc. of the ACM/IEEE Conf. on Supercomputing, November 1996.*

[5] *The Charm++ Programming Language Manual.*