Łagodzenie zasad

Modele równoległości, współbieżności i alternatywne

Fakt, że informatyka nie przynosi tylko dobrych wiadomości, popchnął badaczy w wielu kierunkach, mających na celu próbę złagodzenia problemu. Omówimy niektóre z najbardziej interesujących z nich: równoległość i współbieżność, obliczenia kwantowe i obliczenia molekularne. Każdy z nich reprezentuje nowy paradygmat algorytmiczny i wszystkie robią to, rozluźniając podstawowe założenie leżące u podstaw konwencjonalnego przetwarzania, a mianowicie, że algorytm jest wykonywany przez mały procesor Runaround, który pracuje sam. Równoległość i współbieżność dotyczą bezpośredniego ustawiania tak, aby kilka procesorów (lub kilka małych Runarounds) pracowało razem, we współpracy. Obliczenia kwantowe przenoszą obliczenia do tajemniczej dziedziny mechaniki kwantowej, w której równoległość wynika ze zdolności cząstek do przebywania w więcej niż jednym miejscu jednocześnie. A obliczenia molekularne, czyli biologiczne, są próbą nakłonienia molekuł do wykonywania pracy za nas poprzez masowy, pozornie nadmiarowy paralelizm. Aby wyczuć równoległość, rozważ następujące kwestie. Kilka lat temu w rejonie Los Angeles odbył się konkurs o tytuł mistrza świata w budownictwie szybkim. Należało przestrzegać pewnych sztywnych zasad, takich jak liczba pokoi, wymagane media i dozwolone materiały budowlane. Prefabrykacja nie była dozwolona, ale fundamenty można było przygotować wcześniej. Dom uznano za skończony, kiedy ludzie mogli w nim dosłownie zacząć żyć; cała instalacja wodno-kanalizacyjna i elektryczność musiały być na miejscu i działać idealnie, drzewa i trawa musiały ozdabiać podwórko i tak dalej. Nie nałożono żadnych ograniczeń na wielkość zespołu budowlanego. Zwycięska firma wykorzystała zespół około 200 budowniczych i przygotowała dom w nieco ponad cztery godziny! To uderzająca ilustracja korzyści płynących z równoległości: jedna osoba pracująca sama potrzebowałaby znacznie więcej czasu na ukończenie domu. Tylko dzięki wspólnej pracy, wśród niesamowitych wyczynów współpracy, koordynacji i wzajemnego wysiłku, zadanie mogło zostać zrealizowane w tak krótkim czasie. Obliczenia równoległe umożliwiają równoległą pracę wielu komputerów lub wielu procesorów w jednym komputerze. Obliczenia kwantowe to zupełnie nowe podejście do obliczeń, oparte na mechanice kwantowej, tym kuszącym i paradoksalnym dziele fizyki XX wieku. Jak dotąd odkryto kilka zaskakująco wydajnych algorytmów kwantowych do rozwiązywania problemów, o których nie wiadomo, że są wykonalne w „klasycznym" sensie. Jednak do pracy wymagają zbudowania specjalnego komputera kwantowego, czego na razie nie ma. Obliczenia molekularne, kolejny bardzo niedawny paradygmat, umożliwiły naukowcom nakłonienie rozpuszczalnika molekularnego do rozwiązania przypadków pewnych problemów NP-zupełnych, co stwarza interesujące i ekscytujące możliwości. W pozostałej części omówiono te idee, przy czym paralelizm i współbieżność rozproszona - bardziej klasyczna z nich - zostały potraktowane bardziej szczegółowo.

Równoległość lub łączenie sił

Naszym głównym celem jest zbadanie konsekwencji tego, że wiele procesorów wspólnie osiąga cele algorytmiczne, współpracując ze sobą. To rozluźnienie jest częściowo motywowane chęcią wykorzystania paralelizmu w sprzęcie, a mianowicie dostępności tak zwanych komputerów równoległych, które składają się z wielu oddzielnych elementów przetwarzających, współpracujących i pracujących równolegle. Później uogólnimy również sztywne pojęcie wejścia/wyjścia problemu algorytmicznego na przypadki związane z wiecznością lub ciągłym zachowaniem, które wcale nie musi prowadzić do zakończenia. Takie problemy wynikają z rozproszonych środowisk o z natury równoległej naturze, takich jak systemy rezerwacji lotów czy sieci telefoniczne, a także są związane z rozwojem systemu.

Równoległość pomaga

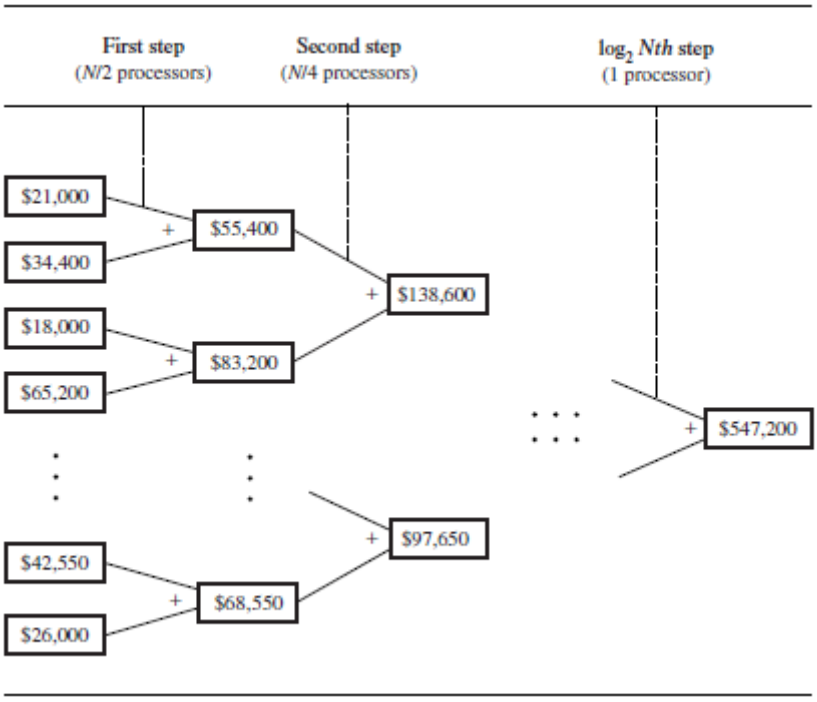
Historia budowy domu pokazuje, że robienie rzeczy równolegle może zdziałać cuda. Zobaczmy, do czego sprowadzają się te cuda pod względem wydajności algorytmicznej. Jeśli algorytm wymaga sekwencji instrukcji:

X ← 3; Y ← 4,

wtedy oczywiście moglibyśmy zaoszczędzić czas, wykonując je jednocześnie, równolegle. Musimy jednak być bardzo ostrożni, aby nie „paralelizować" byle czego; jeśli instrukcje były:

X ← 3; Y ← X

historia byłaby zupełnie inna, ponieważ po równoległym wykonaniu nowa wartość Y może być starą wartością X, a nie nową , 3. Tutaj efekt drugiej instrukcji zależy od wyników pierwszej, a więc dwóch nie może być zrównoleglony. Oczywiście można je modyfikować w sposób, który pozwoli obejść problem, ale w swojej pierwotnej formie muszą być wykonane w odpowiedniej kolejności. Aby lepiej zilustrować tę kwestię, rozważ problem kopania rowu o głębokości jednej stopy, szerokości jednej stopy i długości 10 stóp. Jeśli jedna osoba jest w stanie wykopać rów jeden po drugim w, powiedzmy, godzinę, 10 osób mogłoby w oczywisty sposób wykopać pożądany rów w godzinę. Paralelizm jest tutaj najlepszy. Jeśli jednak pożądany rów ma mieć jedną stopę szerokości, jedną stopę długości i 10 stóp głębokości, równoległość nic nie da, a nawet 100 osób potrzebowałoby 10 godzin, aby wykonać zadanie. Niektóre problemy algorytmiczne można łatwo zrównoleglić, mimo że pierwsze rozwiązania, które przychodzą na myśl, są mocno sekwencyjne; tak naprawdę nie są one z natury sekwencyjne. Wiele z nich można rozwiązać znacznie efektywniej dzięki przetwarzaniu równoległemu. Rozważmy problem sumowania wynagrodzeń z rozdziału 1. Być może konieczne byłoby przejrzenie listy pracowników w kolejności, w celu uzyskania opisanego tam algorytmu liniowego. Bynajmniej. Można wymyślić równoległy algorytm, który będzie działał w czasie logarytmicznym — rzeczywiście jest to doniosłe ulepszenie, jak pokazano w Części 6. Metoda polega na rozważeniu najpierw całej listy N pracowników w parach, <pierwszy, drugi> <trzeci, czwarty> i tak dalej, a także sumując jednocześnie dwie pensje we wszystkich parach, uzyskując w ten sposób listę N/2 nowych liczb (przypomnijmy, że N to całkowita liczba pracowników). Zajmuje to czas tylko jednego dodania, który tutaj liczymy jako pojedynczą jednostkę czasu. Nowa lista jest wtedy również rozpatrywana w parach, a dwie liczby w każdej z nich są ponownie dodawane jednocześnie, co daje nową listę N/4 liczb. Trwa to tak długo, aż pozostanie tylko jeden numer; jest to suma wynagrodzeń z całej listy. Rysunek



ilustruje ten prosty pomysł. Jak wyjaśniono w rozdziale 6, liczba razy N można podzielić przez 2, aż osiągnie 1, wynosi około $\log_2$ N, a zatem logarytmiczne ograniczenie czasowe. Mając na uwadze liczby z Części 6, wynika z tego, że 1000 pensji można zsumować w czasie potrzebnym na wykonanie zaledwie 10 uzupełnień, a milion pensji można zsumować w czasie zaledwie 20 uzupełnień.

Stały a rozszerzający się równoległość

Zauważ, że aby jednocześnie wykonać 500 dodatków wymaganych w pierwszym kroku sumowania 1000 wynagrodzeń, potrzebujemy 500 procesorów. Te same mogą być następnie użyte do przeprowadzenia równolegle 250 dodawania drugiego etapu (połowa z nich byłaby oczywiście bezczynna), następnie 125 dodanych trzeciego etapu i tak dalej. Oczywiście sama duża liczba procesorów nie wystarczy; musimy również ułożyć dane w taki sposób, aby odpowiednie liczby były szybko dostępne dla właściwych procesorów, gdy ich potrzebują. Dlatego dobre struktury danych i metody komunikacji są kluczowe dla szybkich algorytmów równoległych. Koncentrując się jednak na liczbie procesorów w tej chwili, widzimy, że aby osiągnąć skrócenie z czasu liniowego do logarytmicznego, potrzebujemy N/2 procesorów, liczby zależnej od N, czyli długości wejścia. Rzeczywiście, tak jest z konieczności, ponieważ gdybyśmy mieli tylko stałą liczbę procesorów, nie moglibyśmy poprawić rzeczy poza stałą ukrytą pod dużym-O: moglibyśmy być w stanie robić rzeczy dwa razy szybciej lub 100 razy szybciej, ale nadal byłaby liniowa, to znaczy O(N), i nie byłoby poprawy o rząd wielkości. Można by twierdzić, że rosnąca liczba procesorów jest po prostu niewykonalna. Ale nie ma też rosnącej ilości czasu ani pamięci. Celem miar złożoności jest zapewnienie środków do oszacowania nieodłącznej trudności w rozwiązywaniu problemów algorytmicznych w miarę zwiększania się danych wejściowych, przy czym oszacowanie jest podawane jako funkcja wielkości danych wejściowych. Jeśli mamy podsumowywać listy z nie więcej niż milionem pensji i jeśli mamy pod ręką pół miliona procesorów, to będziemy potrzebować bardzo mało czasu (mniej więcej z 20 dodatkami). Jeśli mamy mniej procesorów, możemy zrównoleglać do pewnego punktu; to znaczy do pewnej głębokości w drzewie rysunku powyżej. Od tego momentu trzeba będzie stosować mieszankę równoległości i sekwencyjności. Oczywiście wynik nie będzie tak dobry. Osiągnięcie poprawy o rząd wielkości wymaga zatem rozszerzania równoległości - to znaczy liczby procesorów, która rośnie wraz

ze wzrostem N. Jednak ta liczba nie musi koniecznie wynosić N/2. Na przykład można dodać pensje na liście o długości N w czasie O(√N), jeśli dostępnych jest √ N procesorów - na przykład milion pensji w czasie około 1000 uzupełnień z 1000 procesorów. (Widzisz jak?) Oczywiście, czas 1000 dodatków nie jest tak dobry jak 20, ale wtedy 1000 procesorów to mniej niż pół miliona, więc dostajemy to, za co płacimy, że tak powiem.

Sortowanie równoległe

Sortowanie listy o długości N (na przykład porządkowanie pomieszanej książki telefonicznej) to doskonały problem do omówienia korzyści z przetwarzania równoległego. Rozważ algorytm scalania z Części 4. Wymagał podzielenia listy danych wejściowych na połowy, posortowania ich rekurencyjnie, a następnie scalenia posortowanych połówek. Scalenie uzyskuje się poprzez wielokrotne porównywanie aktualnie najmniejszych (pierwszych) elementów w połówkach, co skutkuje wysłaniem jednego z nich na wyjście. Algorytm można opisać schematycznie w następujący sposób:

podprogram sort-L:

(1) jeśli L składa się z jednego elementu, to jest sortowane;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, L₁ i L₂;

(2.2) wywołanie sort-L₁;

(2.3) wywołanie sort-L₂;

(2.4) scalić listy wynikowe w jedną posortowaną listę.

W Części 6 stwierdzono, że złożoność czasowa tego algorytmu wynosi O(N × log N). Teraz oczywiście dwie czynności sortowania połówek (wiersze (2.2) i (2.3) w algorytmie) nie kolidują ze sobą, ponieważ dotyczą rozłącznych zbiorów elementów. Dlatego można je przeprowadzać równolegle. Nawet zrównoleglenie sortowania połówek tylko raz, na najwyższym poziomie algorytmu, skutkujące potrzebą tylko dwóch procesorów, poprawiłoby sytuację, ale tylko w obrębie stałej big-O, jak już wyjaśniono. Możemy jednak przeprowadzić oba rodzaje równolegle na wszystkich poziomach rekurencji, uzyskując następujące wyniki:

podprogram równoległy-sort-L:

(1) jeśli L składa się z jednego elementu, to jest sortowane;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, L₁ i L₂;

(2.2) jednocześnie wywołaj Parallel-sort-L₁ i Parallel-sort-L₂;

(2.3) scalić listy wynikowe w jedną posortowaną listę.

Przy skrupulatnym przestrzeganiu algorytm ten działa bardzo podobnie do algorytmu sumowania równoległego opartego na drzewie z rysunku powyżej. Po pierwsze, po zejściu do najniższego poziomu rekurencji, jednocześnie porównuje dwa elementy w każdej z par N/2, układając każdą parę we właściwej kolejności. Następnie, ponownie jednocześnie, łączy każdą parę par w posortowaną czwórkę, a następnie każdą parę z czterech krotek w ósemkę i tak dalej. Pierwszy krok zajmuje tylko jedno porównanie, drugi zajmuje trzy (dlaczego?), trzeci siedem, czwarty piętnaście i tak dalej. Zakładając dla uproszczenia, że N jest potęgą liczby 2, łączna liczba porównań wynosi:

1 + 3 + 7 + 15 + … (N − 1)

czyli mniej niż 2BA. Stąd całkowity czas jest liniowy. Ceną zapłaconą za ulepszenie O(N × log N) do O(N) jest zapotrzebowanie na N/2 procesorów. Tutaj też jest kompromis; możemy posortować N elementów w czasie O(N × log N) z jednym procesorem lub w czasie liniowym z liniową liczbą procesorów. Właściwie, jak zobaczymy później, możemy zrobić jeszcze lepiej.

Złożoność produktu: czas × rozmiar

Liczba procesorów wymaganych przez algorytm równoległy jako funkcja długości jego danych wejściowych jest jednym ze sposobów pomiaru złożoności sprzętu wymaganego do uruchomienia algorytmu. Nieco nadużywając terminologii, możemy nazwać tę miarę po prostu złożonością rozmiaru algorytmu, rozumiejąc, że nie mamy na myśli ani długości algorytmu, ani ilości pamięci, jaką wymaga, ale raczej wielkość wymaganej armii procesorów. Ponieważ zarówno miara czasu, jak i rozmiar odgrywają rolę w analizie algorytmów równoległych, nie jest jasne, w jaki sposób powinniśmy określić względną wyższość takich algorytmów. Czy nieco szybszy algorytm jest lepszy, nawet jeśli używa o wiele więcej procesorów? A może powinniśmy poświęcić małą wielkość, aby uzyskać znaczną oszczędność czasu? Jednym ze sposobów szacowania jakości algorytmu równoległego jest połączenie obu miar – pomnożenie czasu przez rozmiar

	Name	Size (no. of processors)	Time (worst case)	Product (time × size)
SEQUENTIAL	Bubblesort	1	$O(N^2)$	$O(N^2)$
SEQUENTIAL	Mergesort	1	$O(N \times \log N)$	$O(N \times \log N)$ (optimal)
PARALLEL	Parallelized mergesort	$O(N)$	$O(N)$	$O(N^2)$
PARALLEL	Odd-even sorting network	$O(N \times (\log N)^2)$	$O((\log N)^2)$	$O(N \times (\log N)^4)$
PARALLEL	"Optimal" sorting network	$O(N)$	$O(\log N)$	$O(N \times \log N)$ (optimal)

Algorytm o lepszej złożoności produktu jest uważany za lepszy. Warto zauważyć, że najlepsza miara produktu nie może być mniejsza niż dolna granica złożoności sekwencyjno-czasowej problemu, ponieważ możliwe jest sekwencjonowanie dowolnego algorytmu równoległego. Odbywa się to poprzez symulację działań różnych procesorów na pojedynczym procesorze w kolejności zgodnej z kolejnością zaleconą przez algorytm równoległy. (Na przykład, gdybyśmy sekwencjonowali algorytm równoległego sortowania przez scalanie, moglibyśmy posortować dwie połówki w dowolnej kolejności, ale oba sortowania musiałyby zostać zakończone przed przeprowadzeniem scalania.) Całkowity czas trwania symulacji wynosi z grubsza suma całego czasu zajętego przez wszystkie procesory; to znaczy

czas produktu × rozmiar oryginalnego algorytmu równoległego. Tak więc, gdybyśmy hipotetycznie mogli znaleźć algorytm sortowania równoległego, który zajmowałby czas logarytmiczny i używał tylko procesorów O(N/log N), można by wyprowadzić zsekwencjonowaną wersję, która działałaby w czasie zgodnym z kolejnością produktu, który jest liniowa (dlaczego?). Ale byłoby to sprzeczne z dolnym ograniczeniem O(N × log N) przy sortowaniu sekwencyjnym. Najlepsze, na co możemy więc liczyć, to algorytm sortowania równoległego, który wykazuje optymalną wydajność produktu O(N × log N). W tym sensie algorytm sekwencyjnego sortowania przez scalanie jest na przykład optymalny. Ale z drugiej strony nie jest to algorytm równoległy. Szczególnie intrygujące jest pytanie, czy istnieje równoległy, oparty na porównaniach algorytm sortowania, który działa w czasie logarytmicznym, ale wymaga tylko liniowej liczby procesorów. W pewnym sensie reprezentowałoby to idealną procedurę sortowania równoległego - niezwykle szybką, ale rozsądnej wielkości. Jak zobaczymy, problem ten został rozwiązany twierdząco.

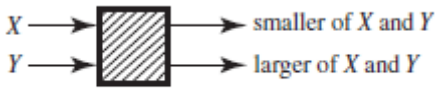
Sieci: równoległość połączenia stałego

Jak dotąd nie powiedzieliśmy nic o współpracy różnych procesorów. Oczywiście nie działają one w całkowitym odosobnieniu, ponieważ muszą przekazywać sobie wyniki pośrednie. Nawet prosty algorytm sumowania równoległego wymaga pewnego rodzaju współpracy między procesorami wytwarzającymi sumy pośrednie i tymi, które używają ich w następnej kolejności. Wydajność algorytmu równoległego może się znacznie różnić w zależności od różnych metod współpracy. Jedno z podejść opowiada się za pamięcią współdzieloną, co oznacza z grubsza, że pewne zmienne lub struktury danych są współdzielone przez wiele procesorów. W ramach tego podejścia ważne jest określenie, czy udostępnianie polega tylko na odczytaniu wartości z odpowiedniej pamięci, czy też ich zapisie. Jeśli zostanie wybrane drugie podejście, musimy zdecydować, jak rozwiązać konflikty zapisu (na przykład dwa procesory próbują jednocześnie pisać do tej samej zmiennej lub lokalizacji pamięci), a konkretna wybrana metoda może mieć duże znaczenie w wynikowym algorytmie moc. Nieograniczona pamięć współdzielona jest uważana za nierealistyczną, głównie dlatego, że jeśli chodzi o budowanie prawdziwych komputerów równoległych, wzorzec połączeń staje się niemożliwie skomplikowany. Albo każdy procesor musi być podłączony do zasadniczo każdej lokalizacji pamięci, albo (jeśli pamięć jest fizycznie rozdzielona między procesorami) każdy procesor musi być podłączony do każdego innego. W obu przypadkach jest to zazwyczaj sytuacja beznadziejna: jeśli wymagana liczba procesorów rośnie wraz z N, jak to ma miejsce w przypadku wszystkich nietrywialnych algorytmów równoległych, połączenia szybko stają się nierozsądnie skomplikowane. Bardziej realistycznym podejściem jest wykorzystanie sieci o stałym połączeniu lub w skrócie tylko sieci i zaprojektowanie specjalnie dla nich algorytmów równoległych. Słowo „stały” oznacza, że każdy procesor jest podłączony do co najwyżej pewnej stałej liczby sąsiednich procesorów. W wielu przypadkach oznacza to również, że cała sieć jest skonstruowana jako maszyna specjalnego przeznaczenia, bardzo wydajnie rozwiązująca jeden konkretny problem algorytmiczny. Procesory w sieci specjalnego przeznaczenia mają zazwyczaj bardzo ograniczone możliwości obliczeniowe. Jedną dobrze znaną klasą sieci są sieci logiczne lub obwody logiczne, nazwane na cześć dziewiętnastowiecznego logika George'a Boole'a, który wynalazł zasady manipulowania wartościami logicznymi prawda i fałsz (a zatem również zasady manipulowania odpowiadającymi im wartościami bitowymi). , 1 i 0). W sieci logicznej procesory nazywane są bramkami i obliczają proste funkcje logiczne jedno- lub dwubitowych wartości. Bramka AND wytwarza 1 dokładnie wtedy, gdy oba jej wejścia mają wartość 1, bramka OR wytwarza 1 dokładnie, gdy co najmniej jedno z jej dwóch wejść ma wartość 1, a bramka NOT odwraca wartość pojedynczego wejścia. W pewnym sensie każdy efektywnie rozwiązywalny problem algorytmiczny P może być rozwiązany przez efektywnie obliczalny jednolity zbiór sieci logicznych. Aby móc dokładniej określić ten fakt, wyobraź sobie, że dane wejściowe P są zakodowane przy użyciu tylko zer i jedynek. Twierdzi się, że dla każdego takiego problemu P istnieje algorytm (powiedzmy maszyna Turinga), który

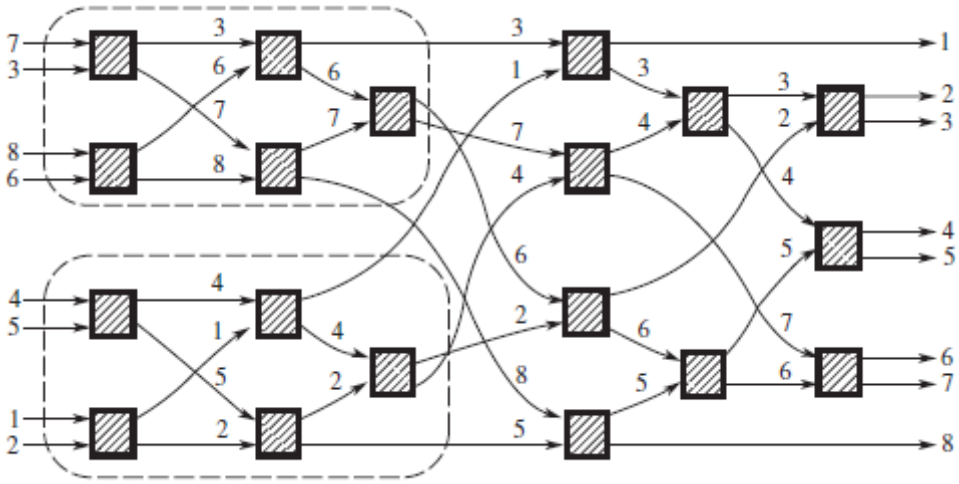
przyjmuje liczbę wejściową N i wyprowadza opis sieci logicznej, która rozwiązuje problem P dla danych wejściowych składających się z N bitów. Jednak ten nieco dziwny rodzaj uniwersalności nie jest powodem, dla którego wprowadzamy sieci. Bardziej interesują nas sieci, które są specjalnie zaprojektowane w celu zapewnienia skutecznych rozwiązań konkretnych problemów.

Nieparzysta i parzysta sieć sortowania

Sieci specjalnego przeznaczenia zostały znalezione dla wielu problemów, ale przede wszystkim do sortowania i łączenia. Sieć sortującą można postrzegać jako architekturę komputerową specjalnego przeznaczenia, zapewniającą stały wzorzec połączeń niezwykle prostych procesorów, które współpracują w celu równoległego sortowania N elementów. Większość sieci sortujących wykorzystuje tylko jeden bardzo prosty rodzaj procesora, zwany komparatorem, który wprowadza dwa elementy i wyprowadza je w kolejności, jak pokazano na rysunku



Zilustrujmy to podejście tak zwaną siecią sortowania nieparzysto-parzyste. Sieć ta jest konstruowana rekursywnie, stosując zasadę konstruowania sieci dla N elementów z sieci dla N/2 elementów. Nie będziemy tu podawać opisu tej ogólnej zasady, ale zilustrujemy ją przykładem. Rysunek

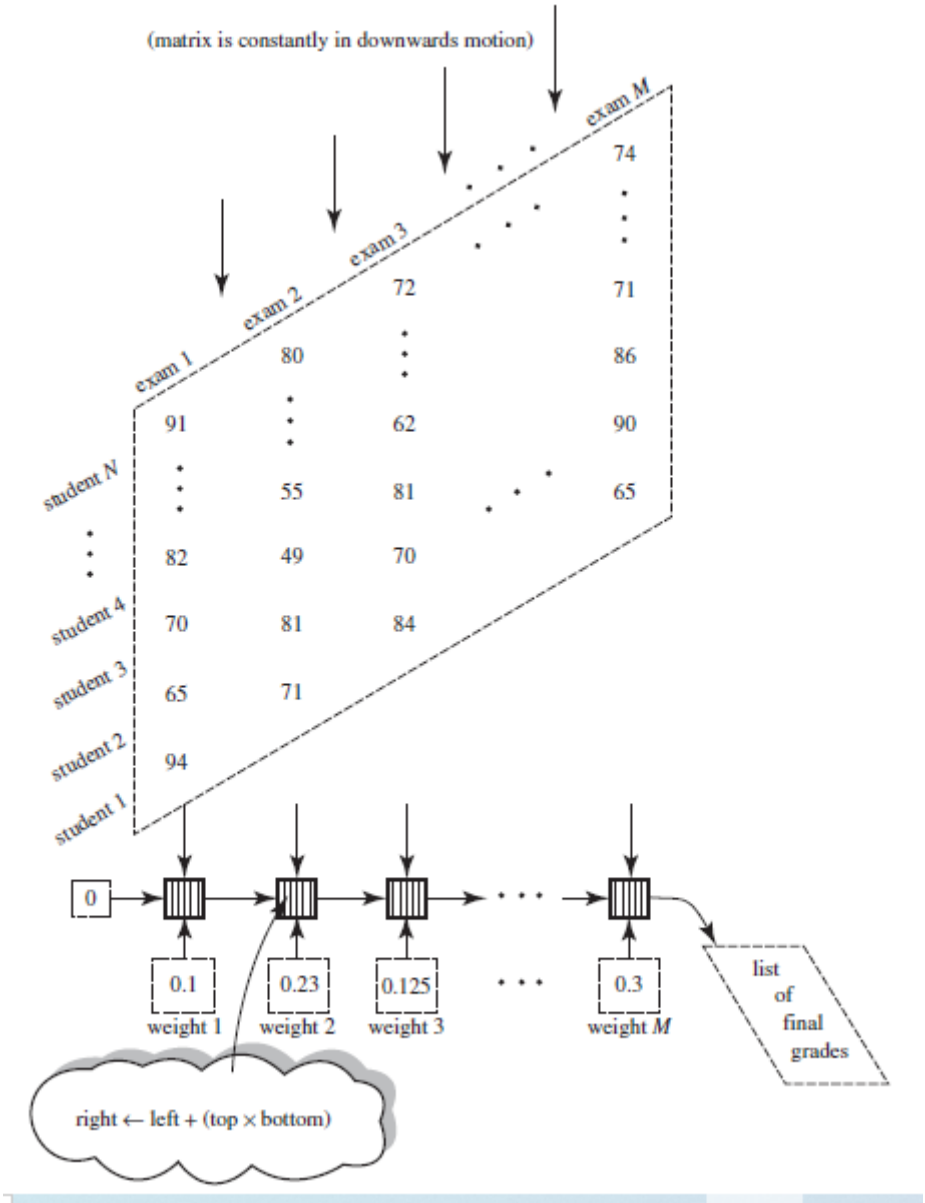


przedstawia sieć dla przypadku, gdy N wynosi 8, z przykładową listą wejść zapisaną wzdłuż linii. Części sieci ujęte w linie przerywane reprezentują dwie sieci nieparzyste-parzyste po 4 elementy każda. W podobnym duchu sieć dla 16 elementów miałaby po lewej stronie dwie sieci dla 8 elementów, identyczne jak na rysunku. Pozostałe części sieci nieparzysto-parzystej składają się z różnych podsieci do łączenia posortowanych list. To, co jest w tym wszystkim sprytne, to sposób, w jaki podsieci są ze sobą połączone, co odkryjesz, jeśli spróbujesz zdefiniować ogólną regułę konstrukcji rekurencyjnej. Czas zajmowany przez nieparzystą sieć sortującą można wykazać jako $O((\log N)_2)$, a jej rozmiar (liczba procesorów) to $O(N \times (\log N)_2)$, stąd nie jest optymalny, ponieważ ich iloczyn jest większy niż optymalny. Przełom w rodzaju został osiągnięty w 1983 r. wraz z pojawieniem się pomysłowej sieci sortowania, która wykorzystuje procesory $O(N \times \log N)$ i zajmuje tylko czas logarytmiczny. Później to rozwiązanie połączono z odmianą sieci nieparzysto-parzystej, aby ostatecznie uzyskać optymalną sieć logarytmiczną o rozmiarze liniowym. Jest to zatem optymalny algorytm sortowania równoległego

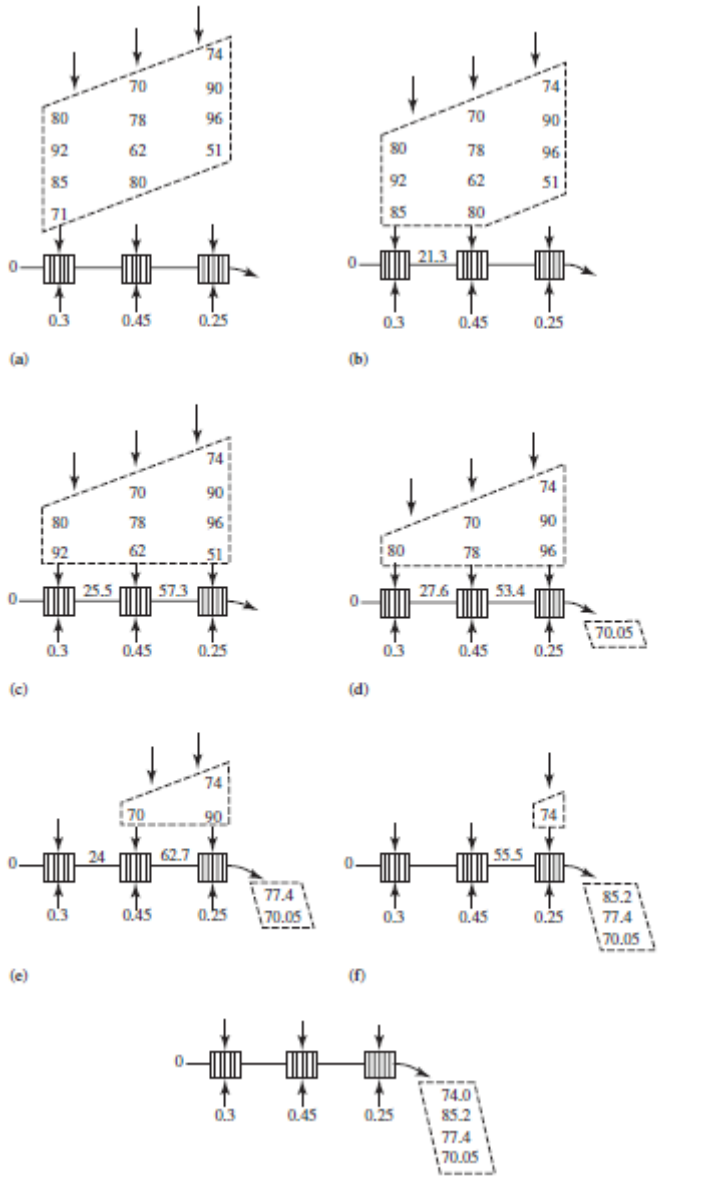
wspomniany wcześniej. Niestety, poza tym, że są niezwykle skomplikowane, stałe big-O w obu tych sieciach logarytmicznych są ogromne, co czyni je w praktyce całkiem bezużytecznymi dla rozsądnej wielkości N. Dla kontrastu, stałe dla wielu teoretycznie gorszych sieci są bardzo małe ; na przykład te z sieci nieparzystych i parzystych są mniejsze niż 1 (około 1/2 dla miary czasu i 1/4 dla rozmiaru). W szczególności możemy zbudować nieparzystą sieć sortującą z nieco ponad 1000 komparatorów, która posortuje 100 elementów w czasie potrzebnym do przeprowadzenia zaledwie 25 porównań. Dla 1000 elementów zajęłoby to tylko około 55 porównań, ale potrzebowalibyśmy około 23 000 komparatorów. Jeśli czas jest kluczowym czynnikiem, a wszystkie listy do posortowania będą mniej więcej tej samej długości, sieć sortująca, taka jak ta, staje się całkiem praktyczna.

Więcej o sieciach: Obliczanie średnich ważonych

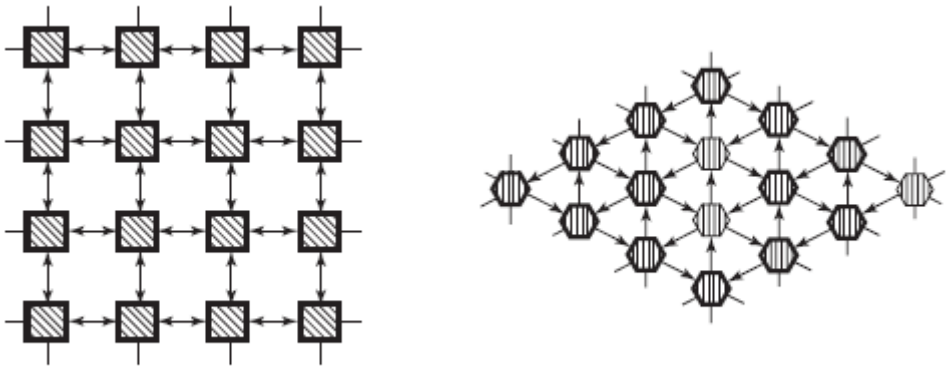
W sieci nieparzystej każdy komparator jest używany tylko raz przy sortowaniu danej listy. Lista wejść zbiera się razem, a cały cykl życia każdego komparatora polega na oczekiwaniu na dwa własne wejścia, porównywaniu ich, wysyłaniu maksimum i minimum wzdłuż odpowiednich linii wyjściowych i wyłączaniu się. Inne rodzaje sieci charakteryzują się tym, że procesory są wielokrotnie aktywowane w ramach jednego przebiegu, w regularny sposób. Takie sieci są czasami nazywane skurczowymi, co wywodzi się od fizjologicznego terminu „skurcz”, który odnosi się do powtarzających się skurczów odpowiedzialnych za pompowanie krwi przez nasze ciała. Jako przykład rozważmy nauczyciela, który jest zainteresowany obliczeniem końcowych ocen N uczniów z kursu, na którym są egzaminy M. Każdy egzamin ma inną wagę, a ocena końcowa ma być średnią ważoną tych egzaminów M. Oceny wejściowe są ułożone w tablicy N na M, a wagi są podane w postaci listy ułamków M, w sumie 1. Dla każdego ucznia wymagane jest pomnożenie każdej oceny przez odpowiednią wagę, a następnie zsumowanie wyników M. Ostatecznym wynikiem ma być lista N ocen końcowych. Sekwencyjny algorytm wymagałby oczywiście czasu O(N × M), ponieważ dla każdej oceny należy wykonać jedno mnożenie, a każdy z N uczniów ma ocenę z każdego egzaminu M. Rysunek



przedstawia sieć skurczową do rozwiązania tego problemu (który w terminologii matematycznej nazywa się problemem mnożenia macierzy przez wektor). Sieć składa się z liniowego układu M połączonych procesorów. Szereg stopni, matryca, pod względem technicznym, jest wprowadzana do sieci od góry w pokazany sposób, po przekątnej na raz. W ten sposób sieć najpierw akceptuje ocenę ucznia 1 z egzaminu 1, następnie jednocześnie ocenę ucznia 2 z egzaminu 1 i ocenę ucznia 1 z egzaminu 2, a następnie jednocześnie ocenę ucznia 3 z egzaminu 1, ocenę 2 z egzaminu 2 i ocena 1 z egzaminu 3 i tak dalej. Ostatnim krokiem jest zaakceptowanie oceny ucznia N z egzaminu M. Z drugiej strony wektor wag jest stale dostępny wzdłuż dolnej linii. Zera są wprowadzane z lewej strony, a wektor wyjściowy jest tworzony z prawej strony, element po elemencie. (Ta liniowa konfiguracja jest czasami nazywana układem potoku, co odnosi się do sposobu, w jaki każdy procesor łączy wyjście do tego po swojej prawej stronie.) Każdy procesor z osobna jest niezwykle prosty: za każdym razem, gdy otrzymuje zestaw nowych danych wejściowych, po prostu mnoży jego górny i dolny, dodaje produkt po lewej stronie i wysyła wynik po prawej stronie. Algorytm indukowany przez tę sieć jest wyraźnie liniowy w N + M. Rysunek



przedstawia prostą symulację sieci krok po kroku dla przypadku czterech studentów i trzech egzaminów. Sieci skurczowe zostały skonstruowane w celu rozwiązania wielu problemów. Układ procesorów jest zwykle liniowy, prostokątny lub w kształcie rombu, a procesory są albo kwadratowe, co daje tak zwaną siatkę połączoną z siatką, albo sześciokątne, dające w wyniku ul, jak na rysunku



Mesh connected

Beehive

Czy paralelizm można wykorzystać do rozwiązania tego, co nierozwiązywalne?

Fakty, które omówiliśmy do tej pory, nie pozostawiają wątpliwości, że paralelizm można wykorzystać do poprawy zachowania czasowego algorytmów sekwencyjnych. Problemy, które wymagają pewnej ilości czasu na rozwiązanie sekwencyjne, można rozwiązywać szybciej, nawet w kategoriach rzędu wielkości, jeśli (rozszerzający się) równoległość jest dozwolona. Naturalne jest pytanie, czy paralelizm można wykorzystać do rozwiązywania problemów, których bez niego w ogóle nie dałoby się rozwiązać. Czy możemy opracować równoległy algorytm dla nierozstrzygalnego problemu? Odpowiedź brzmi: nie, ponieważ, jak wyjaśniono wcześniej, każdy algorytm równoległy może być symulowany sekwencyjnie przez jeden procesor, który biega i wykonuje pracę wszystkich w odpowiedniej kolejności. W tym sensie teza Churcha/Turinga odnosi się również do równoległych modeli obliczeń: klasa problemów rozwiązywalnych jest niewrażliwa nawet na dodanie rozszerzającego się paralelizmu. Następnym pytaniem, które należy zadać, jest to, czy równoległość może zmienić trudne problemy w wykonalne. Czy istnieje problem wymagający nieuzasadnionego (powiedzmy wykładniczego) czasu na sekwencyjne rozwiązanie, które można rozwiązać równolegle w rozsądnym (tj. wielomianowym) czasie? Aby móc lepiej docenić subtelność tego pytania, rozważmy najpierw problemy NP z Części 7. Jak zapewne pamiętasz, wszystkie problemy w NP mają rozsądne rozwiązania, które są niedeterministyczne; używają magicznej monety, która, jeśli zostanie rzucona, kiedy skonfrontowany z wyborem, użyje swojej magii, aby wskazać kierunek, który prowadzi do pozytywnej odpowiedzi, jeśli taki kierunek istnieje. Teraz, jeśli mamy nieograniczoną liczbę procesorów, nie potrzebujemy magicznej monety: kiedy dochodzi do „rozdroża", możemy po prostu wysłać nowe procesory, aby podążały za obiema możliwościami jednocześnie. Jeśli jeden z wysłanych procesorów kiedykolwiek wróci i powie „tak", cały proces zostanie zatrzymany i również powie „tak"; jeśli upłynął z góry określony czas wielomianu i nikt nie powiedział „tak", proces zatrzymuje się i mówi „nie". Ponieważ NP-ność problemu gwarantuje, że jeśli odpowiedź brzmi „tak", to rzeczywiście zostanie ona znaleziona przez magiczną monetę w wielomianowym okresie czasu, nasze wyczerpujące, wieloprocesorowe przemierzenie wszystkich możliwości znajdzie „tak" w tyle samo czasu. Jeśli tak nie jest, to odpowiedź musi brzmieć „nie". Konsekwencja jest jasna. Wszystkie problemy w NP, w tym te NP-zupełne, takie jak łamigłówki z małpami, komiwojażerowie i rozkłady jazdy, mają wielomianowe rozwiązania równoległe. Jednak zanim zaczniesz pospiesznie powiedzieć wszystkim, że niewykonalność jest tylko nieco uciążliwą konsekwencją konwencjonalnych jednoprocesorowych modeli obliczeń, i że można ją wyeliminować stosując obliczenia równoległe, należy poczynić trzy uwagi. Po pierwsze, liczba procesorów potrzebnych do rozwiązania problemu NP-zupełnego w rozsądnym czasie sama w sobie jest wykładnicza. Jeśli chcielibyśmy dowiedzieć się, w czasie krótszym niż miliardy lat pracy komputera, czy nasze lokalne liceum może opracować plan zajęć spełniający wszystkie ograniczenia, potrzebowalibyśmy całkowicie nierozsądnego komputera zawierającego biliony misternie połączonych procesorów. To jest coś, do czego powrócimy za chwilę. Druga uwaga jest zakorzeniona w fakcie, że problemy NP-zupełne nie są nierozwiązywalne – są jedynie przypuszczenia, że tak jest. Tak więc fakt, że możemy rozwiązywać problemy NP-zupełne równolegle w czasie wielomianowym, nie oznacza, że paralelizm może uwolnić problem od jego wrodzonej nierozwiązywalności, ponieważ nie wiemy, czy problemy NP-zupełne są faktycznie nierozwiązywalne. Wreszcie, nawet jeśli mamy algorytm równoległy, który używa tylko wielomianowej liczby instrukcji, ale wymaga wykładniczej liczby procesorów, nie jest jasne, czy naprawdę możemy uruchomić algorytm w czasie wielomianowym na rzeczywistym komputerze równoległym. W rzeczywistości istnieją wyniki, które pokazują, że przy dość liberalnych założeniach dotyczących szerokości linii komunikacyjnych i szybkości komunikacji, superwielomianowa liczba procesorów wymagałaby superwielomianowej ilości czasu rzeczywistego, aby wykonać nawet wielomianową liczbę kroków , bez względu na to, jak procesory są zapakowane razem. Wyniki te opierają się na nieodłącznych ograniczeniach przestrzeni trójwymiarowej. Tak więc

analiza złożoności obliczeń równoległych o nierozsądnych rozmiarach wydaje się wymagać czegoś więcej niż tylko oszacowania liczby kroków przetwarzania; Istotna jest na przykład ilość komunikacji. Pozostaje zatem pytanie: czy możemy użyć paralelizmu, nawet przy nadmiernie wielu procesorach, do rozwiązania w rozsądnym czasie problemu, który może okazać się nierozwiązywalny sekwencyjnie w rozsądnym czasie? I to pytanie jest nadal otwarte, jak teraz zostanie pokazane

Teza obliczeń równoległych

W Części 9 widzieliśmy, że teza Churcha/Turinga jest słuszna również w wyrafinowanym sensie, zgodnie z którym, zgodnie z bardzo naturalnymi założeniami, wszystkie sekwencyjne modele obliczeń są równoważne w obrębie wielomianu czasu. W konsekwencji trakcyjność, a nie tylko rozstrzygalność, jest niewrażliwa na wybór takiego modelu. Teza głosi, że sytuacja ta nie zmieni się wraz z proponowaniem nowych modeli. Podobne twierdzenie dotyczy równoległych modeli obliczeń. Zgodnie z naturalnymi założeniami, wszystkie dotychczas sugerowane uniwersalne modele paralelizmu rozszerzającego, w tym liczne warianty modeli pamięci współdzielonej, modele komunikacyjne, jednolite klasy sieci itp., mogą być równoważne w czasie wielomianowym. Każdy model może być symulowany przez drugi z co najwyżej wielomianową stratą czasu. Oznacza to, że podobnie jak w przypadku sekwencyjnym, klasa problemów rozwiązywanych równolegle w czasie wielomianowym jest również odporna na swój własny sposób; nie zależy to od konkretnego rodzaju wybranego modelu komputera równoległego ani od języka programowania użytego do jego programowania. Podobnie jak w przypadku sekwencyjnym, tutaj musimy być ostrożni. Modele, które pozwalają na współbieżny odczyt i zapis tej samej zmiennej, mogą być wykładniczo wydajniejsze niż te, które pozwalają na odczyt i zapis tylko przez jeden procesor na raz. Nie wchodząc w szczegóły należy stwierdzić, że aby to twierdzenie było prawdziwe, podstawowe instrukcje związane z jednoczesnym manipulowaniem elementami niskiego poziomu muszą mieć podobny charakter (tak jak podstawowe instrukcje manipulowania liczbami muszą być porównywalne w celu zachowania wielomianowej równoważności modeli sekwencyjnych). Fakt ten prowadzi do jednej części tezy o tzw. obliczeniach równoległych. Twierdzi, że i ta sytuacja nie ulegnie zmianie, ponieważ sugerowane są nowe modele. Innymi słowy, aż do różnic wielomianowych, ważne klasy złożoności dla obliczeń równoległych są również solidne i pozostaną takie nawet w miarę postępu nauki i technologii. Ta połowiczna teza jednak nie wystarcza, ponieważ mówi tylko o paralelizmie. Chcemy również wiedzieć, czy istnieje jakiś nieodłączny związek między sekwencyjnością a paralelizmem, jeśli chodzi o efektywność. Wiemy tylko, że równoległość nie pomaga w rozwiązywaniu całkowicie nierozwiązywalnych problemów; ale o ile lepiej może pomóc w rozwiązaniu problemów? Z pomocą przychodzi nam druga część pracy o obliczeniach równoległych. Twierdzi, że (ponownie, aż do różnic wielomianowych) czas równoległy jest taki sam, jak sekwencyjna przestrzeń pamięci. Jeśli możemy rozwiązać problem sekwencyjnie, używając pewnej ilości przestrzeni dla danych wejściowych o długości N, to możemy rozwiązać go równolegle w czasie, który nie jest gorszy niż wielomian w tej ilości przestrzeni. Może to być ta liczba do kwadratu, sześcianu, a może nawet podniesiona do setnej potęgi, ale nie będzie w tym wykładnicza. Odwrotna sytuacja również obowiązuje: jeśli możemy rozwiązać problem równolegle w określonym czasie dla danych wejściowych o długości N, możemy również rozwiązać go sekwencyjnie, używając przestrzeni pamięci ograniczonej wielomianem w tym czasie. Szczególnie interesujący jest szczególny przypadek tego ogólnego faktu, w którym pierwotna ilość miejsca sama w sobie jest wielomianem: każdy problem rozwiązywalny sekwencyjnie przy użyciu tylko wielomianowej ilości przestrzeni pamięci jest rozwiązywalny równolegle w wielomianowym okresie czasu i na odwrót . Symbolicznie:

Sekwencyjny-PSPACE = Równoległy-PTIME

Tak więc pytanie, czy istnieją nierozwiązywalne problemy, które stają się wykonalne wraz z wprowadzeniem paralelizmu, sprowadza się do tego, czy sekwencyjna klasa złożoności PSPACE zawiera

nierozwiązywalne problemy; to znaczy takie, co do których można udowodnić, że nie dopuszczają rozwiązań sekwencyjnych w czasie wielomianowym. To pytanie, sformułowane wyłącznie w kategoriach sekwencyjnych, jest nadal otwarte i, podobnie jak problem P vs. NP, jest prawdopodobnie również bardzo trudne.

Klasa Nicka: w kierunku rozsądnego równoległości

Ogólnie rzecz biorąc, wielomianowe algorytmy równoległe nie mogą być uważane za rozsądne, ponieważ mogą wymagać całkowicie nieuzasadnionej (tj. wykładniczej) liczby procesorów. Ponadto jednym z celów wprowadzenia równoległości jest radykalne skrócenie czasu pracy. W rzeczywistości jednym z celów jest tutaj znalezienie algorytmów podliniowych; czyli algorytmy, które są równoległe do tego stopnia, że nawet nie odczytują całego wejścia sekwencyjnie (w przeciwnym razie wymagałyby co najmniej liniowego czasu). Stąd wydaje się, że Parallel-PTIME nie jest najlepszym wyborem dla klasy problemów, które są wykonalne w obecności równoległości. Jaka jest zatem „właściwa" definicja podatności równoległej? Jedną z ciekawszych propozycji dotyczących tego zagadnienia jest klasa problemów zwana NC (klasa Nicka, od nazwiska jednego z jej pierwszych badaczy). Problem jest w NC, jeśli dopuszcza bardzo szybkie rozwiązanie równoległe, które wymaga tylko wielomianowo wielu procesorów. „Bardzo szybko" oznacza, że działa w czasie polilogarytmicznym; czyli w czasie jest to pewna stała potęga logarytmu N, jak O(log N) lub O((log N)²). Sumowanie wynagrodzeń, sortowanie, obliczanie średniej ważonej i wiele innych problemów są w NC (pierwsze dwa algorytmy opisane wcześniej, a trzeci algorytm nie opisany tutaj). Na przykład problemy NP-zupełne mogą, ale nie muszą być w NC-nie wiemy. Udało nam się jedynie wykazać, że dopuszczają rozwiązania równoległe w czasie wielomianowym; bycie w NC jest silniejszym wymogiem. Klasa NC jest solidna w tym samym sensie, co klasy takie jak P, NP i PSPACE. Pozostaje niewrażliwy na różnice między różnymi modelami obliczeń równoległych, a zatem może być używany w badaniu wrodzonej mocy równoległości. Można na przykład pokazać, że wszystkie problemy w NC są również w P (to znaczy w Sekwencyjnym-PTIME), ale nie wiadomo, czy jest odwrotnie: czy istnieje problem, który jest wykonalny w sensie sekwencyjnym (to jest w P), ale nie w tym równoległym sensie? Również tutaj, podobnie jak w pytaniu P vs. NP, większość badaczy uważa, że te dwie klasy są różne, a więc P różni się od NC. W szczególności problem znalezienia największego wspólnego dzielnika dwóch liczb całkowitych (jego gcd) występuje w P - Eukclidesowe znalazł dla niego algorytm wielomianowy już 2300 lat temu, jak wspomniano w rozdziale 1. Jednak problem gcd jest podejrzewa się, że nie jest w NC. Nikt nie wie, jak wykorzystać równoległość, aby znacząco przyspieszyć obliczanie tej najważniejszej wielkości, tak jak możemy przyspieszyć sortowanie i sumowanie wynagrodzeń. Oznacza to, że każdy krok klasycznego algorytmu gcd zależy od poprzednich kroków i nikt nie był w stanie znaleźć sposobu na usunięcie części tej zależności, aby można było z korzyścią wykorzystać równoległość. Tak więc mamy

$NC \subseteq P \subseteq NP \subseteq PSPACE$

a wielu informatyków uważa, że wszystkie trzy inkluzje są w rzeczywistości ścisłe. Przypominając wyżej wspomniane połączenie między PSPACE i Parallel-PTIME, ten zestaw przypuszczalnych nierówności można opisać (od prawej do lewej) mówiąc:

1. Istnieją problemy, które można rozwiązać w rozsądnej przestrzeni sekwencyjnej - to znaczy w rozsądnym czasie równoległym (ale nierozsądnym rozmiarze sprzętu) - których nie można rozwiązać w rozsądnym czasie sekwencyjnym, nawet przy magicznym niedeterminizmie.

2. Istnieją problemy, które można rozwiązać w rozsądnym czasie sekwencyjnym za pomocą magicznego niedeterminizmu, których bez niego nie da się rozwiązać w takim czasie.

3. Istnieją problemy, które można rozwiązać w rozsądnym czasie sekwencyjnym, których nie można rozwiązać w bardzo krótkim czasie równoległym przy rozsądnym rozmiarze sprzętu.

Jednak żadna z tych nierówności nie jest tak naprawdę ścisła, a zatem jest po prostu możliwe (choć bardzo mało prawdopodobne), że te klasy problemów są w rzeczywistości wszystkie równe. Należy wspomnieć, że podobnie jak w przypadku NP vs P, pytanie P vs NC również rodzi naturalne pojęcie kompletności. Tak więc, chociaż nie wiemy, czy P = NC, bardzo pomocne jest wykazanie, że problem jest P-zupełny, co oznacza, że jeśli jest w NC, to wszystkie problemy w P również są w NC.

Współbieżność rozproszona i ciągła

Byłoby miło móc zakończyć w tym miejscu dyskusję na temat jednoznacznej jednoczesności. A dlaczego nie? Omówiliśmy algorytmy równoległe i ich złożoność oraz zobaczyliśmy, jak mogą one ulepszyć sytuację i do jakiego stopnia. Przyznaliśmy nawet, że nie wiemy o paralelizmie prawie tyle, ile powinniśmy. Czego zatem brakuje? Cóż, wprowadzenie paralelizmu w celu efektywniejszego rozwiązywania konwencjonalnych problemów algorytmicznych to tylko jeden z jego aspektów. Drugi dotyczy sytuacji, w których paralelizm nie jest czymś, co wprowadzamy, aby coś poprawić, ale czymś, z czym musimy żyć, ponieważ po prostu tam jest. Wiąże się to również z innym rodzajem problemu algorytmicznego, który nie zawsze wiąże się z przekształcaniem danych wejściowych podanych na początku w pożądane wyniki, które są wytwarzane na końcu. Jest to raczej to, co można znaleźć w wielu (a właściwie większości!) reaktywnych i wbudowanych systemach, które omówimy w części 14. Problem polega na określeniu protokołów pożądanego zachowania w czasie, aby zagwarantować różne właściwości wymagane od tego zachowania trzymać. To, co szczególnie utrudnia te problemy, to fakt, że w wielu przypadkach protokół, który należy opisać jako ich rozwiązanie, w ogóle nie musi się kończyć. Musi po prostu siedzieć tam na zawsze, robiąc rzeczy, które zawsze są zgodne z tymi wymogami. Aby rozróżnienie było jasne, w tym przypadku użyjemy terminu współbieżność, a nie równoległość. W przeciwieństwie do Części 13 i 14, tutaj nie interesuje nas ogólny problem metod i języków służących do inżynierii rozwoju dużych i złożonych systemów, ale pojawiające się tam problemy algorytmiczne na małą skalę, ale zawsze tak subtelne. Rozważmy przykład. Załóżmy, że pewien hotel ma tylko jedną łazienkę na każdym piętrze. (Wiele niedrogich hoteli jest tego typu.) Załóżmy też, że ogrzewanie jest tylko w pokojach - na korytarzu jest nieprzyjemnie chłodno. (Wiele niedrogich hoteli również spełnia to założenie.) Co jakiś czas goście muszą brać prysznic. (Większość gości w takich hotelach rzeczywiście to robi). Jak powinni się zająć zaspokajaniem tych potrzeb? Nie mogą ustawić się w kolejce przed drzwiami łazienki, z powodu związanego z tym czynnika drżenia. Jeśli gość po prostu raz na jakiś czas spróbuje otworzyć drzwi łazienki, może pozostać nieczysty na zawsze, ponieważ całkiem możliwe, że za każdym razem będzie zajęty. Dzieje się tak pomimo faktu, że prysznice zajmują tylko skończoną ilość czasu; ktoś inny może zawsze wejść pierwszy. Zakładamy, że goście pochodzą z różnych krajów i wszyscy mówią różnymi językami, więc bezpośrednia komunikacja nie wchodzi w rachubę. Jedno z oczywistych rozwiązań wymaga przymocowania małej tablicy do zewnętrznej części drzwi do łazienki. Każdy gość wychodzący z łazienki kasuje zapisany na tablicy numer pokoju (będzie on jego) i zapisuje numer kolejnego pokoju w ustalonej kolejności. (Pomyśl o pokojach na piętrze jako rozmieszczonych cyklicznie, tak aby każdy pokój miał swojego niepowtarzalnego poprzednika i następcę). na tablicy i wracać do swojego pokoju, jeśli tak nie jest. Najwyraźniej każdy gość w końcu dostanie swoją kolej na prysznic. Jednak to rozwiązanie nadal ma poważne problemy. Po pierwsze, możliwe jest, że pokój numer 16 nigdy nie jest zajęty, że jego mieszkaniec nigdy nie chce wziąć prysznica, albo został właśnie uprowadzony i nigdy więcej go nie widziano. Na tablicy będzie wtedy na zawsze napisane 16 i żaden z pozostałych gości nigdy nie będzie czysty. Po drugie, nawet jeśli we wszystkich pomieszczeniach mieszkają żywi, dostępni ludzie, to rozwiązanie narzuca wszystkim gościom niemożliwą do spełnienia dyscyplinę jeden prysznic na cykl,

zmuszając tych bardziej wybrednych do brania prysznica tak rzadko, jak tych, którym nie zależy na tym. Problem ten ilustruje niektóre z głównych problemów pojawiających się podczas projektowania systemów, które są z natury rozproszone. Dystrybucja to szczególny rodzaj współbieżności, w którym współbieżne komponenty są fizycznie oddalone. W związku z tym, ludzie troszczą się nie tylko o to, co każdy komponent powinien zrobić i jak należy wykonywać inżynierię oprogramowania i zarządzanie projektami, ale także o zminimalizowanie ilości komunikacji, która ma miejsce między komponentami. Komunikacja, czy to jawna, czy niejawna (powiedzmy w postaci pamięci współdzielonej), może stać się niezwykle kosztowna w systemie rozproszonym. Tak jak poprzednio, będziemy używać terminu „procesory" dla oddzielnych podmiotów lub komponentów w systemie rozproszonym. Procesory w z natury współbieżnym lub rozproszonym systemie nie muszą osiągać jedynie relacji wejścia/wyjścia, ale raczej wykazywać określone pożądane zachowanie w czasie. W rzeczywistości system może w ogóle nie być zmuszony do zakończenia pracy, tak jak w problemie z prysznicem. Części pożądanego zachowania można określić jako ograniczenia globalne. Na przykład prysznic niesie ze sobą ograniczenie mówiące, że może pomieścić najwyżej jedną osobę na raz, a inne istotne ograniczenie mówi, że każdy potrzebuje od czasu do czasu prysznica. Tak więc prysznic jest tak naprawdę kluczowym zasobem, takim, jakiego każdy potrzebuje przez pewien ograniczony czas, po którym można z niego zrezygnować na rzecz kogoś innego. Innym ważnym ograniczeniem w problemie z prysznicem jest to, że musimy zapobiegać sytuacjom impasu, w których żaden procesor nie może zrobić postępu (na przykład w rozwiązaniu tablicy, gdy gość 16 nigdy się nie pojawia) oraz sytuacjom głodu - czasami nazywanych blokadami - w któremu co najmniej jednemu procesorowi, ale nie każdemu z nich, nie udaje się zrobić postępu (na przykład, gdy goście są poinstruowani, aby po prostu spróbowali od czasu do czasu drzwi do łazienki, a pechowcy mogą zostać na zawsze). Pojęcia te są typowe dla wielu rzeczywistych systemów, takich jak omówione w częściach 13 i 14. Na przykład większość standardowych komputerów ma kilka kluczowych zasobów, takich jak napędy taśm i dysków, drukarki i plotery oraz kanały komunikacyjne. W pewnym sensie pamięć komputera może być również postrzegana jako kluczowy zasób, ponieważ nie chcielibyśmy, aby dwa działające razem zadania jednocześnie zapisywały w tej samej lokalizacji. Innym przykładem jest system rezerwacji lotów, który jest bardzo rozpowszechniony. Może składać się z ogólnoświatowej sieci zawierającej jeden duży komputer w Nowym Jorku, drugi w Los Angeles i 2000 terminali. Byłoby żenujące, gdyby dwa z terminali przydzielały miejsce 25D na ten sam lot dwóm różnym pasażerom. Nie można również zakończyć systemu rezerwacji lotów. Musi nadal działać, stale udostępniając swoje zasoby wszystkim procesorom, którzy o to poproszą, jednocześnie zapobiegając zakleszczeniu i głodowi. To naprawdę nie są problemy algorytmiczne w zwykłym znaczeniu tego słowa. Obejmują różne rodzaje wymagań i nie są rozwiązywane przez zwykłe algorytmy. Rozwiązanie musi dyktować protokoły algorytmiczne dla zachowania każdego z procesorów, które gwarantują spełnienie wszystkich wymagań i ograniczeń określonych w zadaniu oraz przez cały (potencjalnie nieskończony) czas życia systemu.

Rozwiązywanie problemów z prysznicem w hotelu

Tablica sugerowana dotycząca problemu kąpieli pod prysznicem w niedrogich hotelach może być traktowana jako pamięć współdzielona, a prysznic jako kluczowy zasób. Zauważ, że omijaliśmy problem konfliktów w pisaniu, pozwalając na pisanie na tablicy tylko jednemu gościowi w tym samym czasie. Opiszmy teraz satysfakcjonujące rozwiązanie tego problemu. Właściwie rozwiążemy trudniejszy problem, w którym łazienki są w każdym pokoju, ale jednorazowo tylko jeden gość może brać prysznic. (Powód może być związany z ciśnieniem lub temperaturą wody, co jest kolejnym rozsądnym założeniem w przypadku niedrogich hoteli.) Ta sytuacja jest wyraźnie bardziej delikatna, ponieważ nie ma bezpośredniego sposobu, aby dowiedzieć się, czy ktoś rzeczywiście bierze prysznic. Niemniej jednak rozwiązanie musi zapewniać, że wszyscy biorą prysznic, ale dwoje gości nigdy nie bierze prysznica jednocześnie. Bardziej ogólny opis problemu jest następujący. Istnieje N procesorów, z

których każdy musi wielokrotnie wykonywać pewne „prywatne” czynności, po których następuje sekcja krytyczna:

cykl życia I procesora:

(1) wykonaj następujące czynności raz za razem:

(1.1) wykonywać prywatne czynności (np. jeść, czytać, spać);

(1.2) przeprowadzić sekcję krytyczną (na przykład wziąć prysznic).

Prywatne działania to moja własna działalność przetwórcy - nie mają nic wspólnego z nikim innym. Z drugiej strony, sekcja krytyczna to sprawa wszystkich, ponieważ żadne dwa procesory nie mogą znajdować się w swojej sekcji krytycznej jednocześnie. Problem polega na tym, aby znaleźć sposób, aby procesory N mogły żyć wiecznie, bez impasu i głodu, przy jednoczesnym poszanowaniu krytycznej natury sekcji krytycznych. Musimy poinstruować procesory, aby wykonywały określone czynności, takie jak sprawdzanie tablic i pisanie na nich, przed i/lub po wejściu w ich krytyczne sekcje, aby te wymagania zostały spełnione. Przedstawiony w tej formie problem bywa nazywany problemem wzajemnego wykluczania, ponieważ procesorom należy zagwarantować wyłączność w zakresie wchodzenia w krytyczne sekcje.

Omówimy rozwiązanie w przypadku dwóch gości, czyli procesorów, P_1 i P_2. Bardziej ogólny przypadek dla procesorów N jest przedstawiony później. Użyjemy trzech zmiennych, X_1, X_2 i Z, które w przykładzie z prysznicem są reprezentowane przez trzy małe obszary na tablicy wiszącej na korytarzu. Z może wynosić 1 lub 2, a oba procesory mogą zmieniać jego wartość. W takich przypadkach mówimy, że Z jest zmienną wspólną. Znaki X mogą być tak lub nie i mogą być odczytywane przez oba procesory, ale zmieniane tylko przez procesor z odpowiednim indeksem; to znaczy, P1 może zmienić X_1, a P_2może zmienić X_2 W takich przypadkach mówimy, że X są zmiennymi rozłożonymi. Początkowa wartość obu X to nie, a Z to 1 lub 2, to nie ma znaczenia. Oto protokoły dla dwóch procesorów:

protokół dla P_1:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) $X_1 \leftarrow$ tak;

(1.3) $Z \leftarrow 1$;

(1.4) poczekaj, aż X_2 stanie się nie lub Z stanie się 2 (lub oba);

(1.5) przeprowadzić sekcję krytyczną;

(1.6) $X_1 \leftarrow$ nie.

protokół dla P_2:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) $X_2 \leftarrow$ tak;

(1.3) $Z \leftarrow 2$;

(1.4) czekać, aż X_1 stanie się nie lub Z stanie się 1 (lub oba);

(1.5) przeprowadzić sekcję krytyczną;

(1.6) $X_2 \leftarrow$ nie.

Co się tutaj dzieje? Pomyśl o X_1 jako o tym, że P_1> chce wejść do jego krytycznej sekcji: X_1 oznacza, że tak oznacza, że chciałby wejść, a X_1 nie oznacza, że zakończył i chciałby wrócić do jego prywatnej działalności. X_2 odgrywa tę samą rolę dla p_2 Trzecia zmienna, Z, jest pewnego rodzaju wskaźnikiem grzecznościowym: po tym, jak P1 jasno określił, że chce wejść do swojej sekcji krytycznej, natychmiast ustawia Z na 1, wskazując w bardzo hojny sposób, że jeśli chodzi o , P_2może wejść, jeśli chce. Następnie czeka, aż albo P_2 wskaże, że opuścił własną sekcję krytyczną i wróci do swoich prywatnych działań (to znaczy, że X_2 oznacza nie), albo P_2 ustawi zmienną grzecznościową Z na 2, hojnie dając P_1 pierwszeństwo . P_2 działa w podobny sposób. (Możemy myśleć o Z jako o reprezentowaniu podpisów procesorów w dzienniku. Ostatnim, który się loguje, jest ten, który ostatnio dał drugiemu pierwszeństwo.) Oczekiwanie w klauzuli (1.4) jest czasami nazywane zajętym czekanie, ponieważ procesor nie może po prostu bezczynnie, dopóki nie zostanie poparty przez kogoś innego. Raczej aby nadal sprawdzać wartości Z i samej zmiennej X drugiego procesora, nie robiąc w międzyczasie nic więcej. Zobaczmy teraz, dlaczego to rozwiązanie jest poprawne. Cóż, przede wszystkim twierdzimy, że P_1 i P_2 nie mogą być w swoich krytycznych sekcjach jednocześnie. Załóżmy, że mogą. Oczywiście, w czasie, gdy znajdują się w swoich krytycznych sekcjach razem, zarówno X_1, jak i X_2mają wartość tak, więc ostatni procesor, który wszedł, musiał wejść na mocy części Z swojego testu oczekiwania w klauzuli (1.4), ponieważ wartość X innych była tak. Oczywiście oba procesory nie mogły przejść testów w tym samym czasie. (Dlaczego?) Powiedzmy, że P_1 zdał test jako pierwszy. Wtedy Z musiało wynosić 1, gdy P_2 później zdał test. Oznacza to, że między momentem, w którym P_1 ustawił Z na 2 w swoim punkcie (1.3) a momentem, w którym przeszedł test w punkcie (1.4), P_1musiał ustawić Z na 1 i od tego momentu P_1 nie robił nic więcej, dopóki nie wszedł jego sekcję krytyczną ze względu na fakt, że Z wynosi 1. Ale w jaki sposób P_1 zdołał wcześniej wejść do swojej sekcji krytycznej? Nie mógł wejść ze względu na fakt, że Z wynosi 2, ponieważ, jak już powiedzieliśmy, ustawił Z na 1 po tym, jak P_1 ustawił go na 2. Ponadto nie mógł wejść z tego powodu, że X_1 jest nie, ponieważ X_2 było ustawiony na tak przez P_2, zanim ustawi Z na 2. A zatem nie ma możliwości, aby P_2 wszedł w swoją sekcję krytyczną, o ile P_1 znajduje się we własnej sekcji krytycznej. Podobny argument dotyczy podwójnej możliwości, gdzie P_1 wchodzi, podczas gdy P_2 już jest. Wniosek jest taki, że rzeczywiście obowiązuje wzajemne wykluczenie. Zobaczmy, dlaczego głód i impas są niemożliwe. (W tym przypadku głód pojawia się, gdy jeden z procesorów chce wejść do swojej krytycznej sekcji, ale na zawsze nie może tego zrobić, a zakleszczenie występuje, gdy żaden z procesorów nie może poczynić postępów.) Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że zablokowanie obu procesorów jest niemożliwe. w ich klauzulach (1.4) razem, ponieważ Z jest zawsze 1 lub 2, a zatem jeden z nich może zawsze uwolnić się od oczekiwania. Teraz załóżmy, że P_1 jest zmuszony do pozostania w swojej klauzuli (1.4), podczas gdy P_2 nie. Jeśli P_2 nigdy nie chce wejść do swojej sekcji krytycznej, to znaczy pozostaje w klauzuli (1.1), wartość X_2 będzie wynosić nie, a P_1 będzie mógł wejść, kiedy tylko zechce. Jedyną inną możliwością jest to, że P_2 kontynuuje cykle wokół swojego protokołu na zawsze. W takim przypadku prędzej czy później ustawi Z na 2, a ponieważ nigdy nie może zmienić go z powrotem na 1 (może to zrobić tylko P_1), P_1 w końcu będzie mógł przejść test i wejść (1.5). W ten sposób zapobiega się głodowi i impasowi, a rozwiązanie spełnia wszystkie wymagania.

Rzeczy są trudniejsze niż się wydaje

To dwuprocesorowe rozwiązanie wydaje się na pierwszy rzut oka dość elementarne, a argument dotyczący poprawności, choć nieco zawikłany, również nie wygląda na zbyt skomplikowany. Jednak taka prostota może być dość zwodnicza. Jako ilustrację tego, jak naprawdę delikatne są rzeczy, przestudiujmy niewielką odmianę powyższego rozwiązania. Co by się stało, gdybyśmy zamienili

kolejność klauzul (1.2) i (1.3) w protokołach? Innymi słowy, gdy na przykład P_1 chce wejść do swojej sekcji krytycznej, najpierw robi P_2 dzięki uprzejmości ustawienia Z na 1, przepuszczając go, jeśli chce, i tylko następnie ustawia X_1 na tak, aby wskazać, że chce wejść. Na pierwszy rzut oka wydaje się, że nie ma różnicy: P_1 może nadal wejść do swojej sekcji krytycznej tylko wtedy, gdy P_2 jest albo bezinteresowny (to znaczy, jeśli X_2 jest nie) lub wyraźnie daje P_1 pierwszeństwo (to znaczy, jeśli Z wynosi 2). Co może pójść źle? W tym momencie powinieneś spróbować przepracować podany nieformalny dowód poprawności, ale w przypadku poprawionej wersji, aby zobaczyć, gdzie zawodzi. To się nie udaje, a oto scenariusz, który prowadzi do tego, że oba procesory znajdują się jednocześnie w swoich krytycznych sekcjach. Początkowo, zgodnie z ustaleniami, X_1 to nie, a X_2 to nie, a oba procesory są zajęte swoimi prywatnymi czynnościami. Teraz pojawia się następująca sekwencja działań:

1. P_2 ustawia Z na 2;

2. P_1 ustawia Z na 1;

3. P_1 ustawia X1 na tak;

4. P_1 wchodzi w swoją sekcję krytyczną, ponieważ X_2 jest nie;

5. P_2 ustawia X_2 na tak;

6. P_1 wchodzi w swoją sekcję krytyczną, ponieważ Z wynosi 1.

Problem polega oczywiście na tym, że teraz P_2 może wziąć prysznic (to znaczy wejść do swojej sekcji krytycznej), mimo że P_1 sam bierze prysznic, ponieważ P_1 był ostatnim, który był uprzejmy, a zatem wartość Z wynosi 1, gdy P_2 wskazuje jego chęć wejścia. W oryginalnym protokole było to niemożliwe, ponieważ ostatnią rzeczą, jaką musiał zrobić P2 przed wejściem, było ustawienie Z na niekorzystną wartość 2. Tak więc nawet to pozornie proste rozwiązanie dla dwóch procesorów kryje w sobie pewną subtelność. Przyjrzyjmy się teraz ogólnemu rozwiązaniu dla procesorów N, które jest jeszcze delikatniejsze.

Rozwiązanie problemu wzajemnego wykluczania procesorów N

Gdy gości jest więcej niż dwóch, nie wystarczy powiedzieć „chciałbym wziąć prysznic”, a następnie być uprzejmym wobec jednego z pozostałych. Nie pomoże też być uprzejmym dla wszystkich innych razem i mieć nadzieję, że wszystko się ułoży. Musimy uczynić nasze procesory bardziej wyrafinowanymi. Procesorzy zostaną poinstruowani, aby zwiększyć nacisk na wejście do sekcji krytycznej, gdy będą czekać. W ten sposób każdy procesor przejdzie przez pętlę, której wykonanie będzie odpowiadać wyższemu poziomowi nacisku. Liczba poziomów to dokładnie N, całkowita liczba procesorów, a pierwszy poziom (właściwie zero) wskazuje na brak zainteresowania wejściem do sekcji krytycznej. Każdy z poziomów nalegań ma swoją własną zmienną grzecznościową, a na każdym poziomie procesor jest uprzejmy dla wszystkich innych, wpisując w tej zmiennej własną liczbę. Gdy przetwórca zasygnalizował chęć zwiększenia poziomu nalegań i był uprzejmy dla wszystkich innych, czeka z podwyższeniem poziomu, aż albo wszyscy inni będą mniej natarczywi niż on sam, albo otrzyma uprzejmość i przystąpi do następny poziom przez kogoś innego. Sekcja krytyczna jest ostatecznie wprowadzana, gdy pojawi się zielone światło umożliwiające przekroczenie najwyższego poziomu. Po opuszczeniu sekcji krytycznej procesor rozpoczyna całą procedurę od nowa na poziomie zerowym. Oto protokół dla I procesora. W nim wektor Z jest indeksowany przez poziomy nalegań, a nie przez procesory. Zatem procesor I ustawia Z[J] na I (nie Z[I] na J, jak w przypadku wektora X), aby wskazać, że na poziomie J daje pierwszeństwo wszystkim innym. Warto zauważyć, że jeśli N = 2 to protokoły te

są dokładnie tymi właśnie przedstawionymi, więc jest to rzeczywiście bezpośrednie rozszerzenie rozwiązania dwuprocesorowego.

protokół dla I procesora, PI:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) dla każdego J od 1 do N − 1 wykonaj następujące czynności:

(1.2.1) X[I] ← J ;

(1.2.2) Z[J] ← I ;

(1.2.3) czekaj, aż X[K] < J dla wszystkich K = I lub Z[J] = I ;

(1.3) przeprowadzić sekcję krytyczną:

(1.4) X[I] ← 0.

Możliwe jest dostarczenie nieformalnego dowodu poprawności dla tego ogólnego przypadku, na wzór dowodu dla przypadku dwuprocesorowego. Tutaj pokazujemy, że na każdym poziomie procesory działają tylko jeden na raz, tak że w szczególnym przypadku w danej chwili w krytycznej sekcji znajduje się najwyżej jeden procesor. Podobnie ustala się wolność od impasu i głodu.

Właściwości bezpieczeństwa i żywotności

Rozproszone systemy współbieżne są pod wieloma względami bardziej wymagające niż normalne systemy sekwencyjne. Trudności zaczynają się już od samego zadania sprecyzowania problemu i pożądanych właściwości rozwiązania. „Problemy algorytmiczne” związane z takimi systemami nie mogą być już dłużej definiowane naiwnie poprzez określenie zestawu legalnych danych wejściowych wraz z funkcją opisującą pożądane wyniki. Poprawność nie polega już po prostu na zakańczaniu i generowaniu poprawnych wyjść, a wydajności nie można już naiwnie mierzyć liczbą kroków wykonanych przed zakończeniem lub rozmiarem używanej pamięci. W związku z tym techniki opracowane w celu udowodnienia poprawności lub szacowania wydajności konwencjonalnych algorytmów są niewystarczające, jeśli chodzi o protokoły rozwiązujące problemy o charakterze współbieżnym i rozproszonym. Niektóre aspekty tych problemów omówiono w dalszej części, w częściach 13 i 14. Okazuje się, że większość wymagań dotyczących poprawności protokołów dla bieżących systemów współbieżnych można podzielić na dwie główne kategorie, właściwości bezpieczeństwa i żywotności. Właściwości bezpieczeństwa stwierdzają, że pewne „złe” rzeczy nigdy nie mogą się wydarzyć lub, równoważnie, że odpowiadające im „dobre” rzeczy zawsze będą miały miejsce, a właściwości żywotności stwierdzają że pewne „dobre” rzeczy w końcu się wydarzą. Wzajemne wykluczanie - zapobieganie jednoczesnemu przebywaniu dwóch procesorów w ich krytycznych sekcjach - jest właściwością bezpieczeństwa, ponieważ wymaga gwarancji, że zawsze będzie co najwyżej jeden procesor w krytycznej sekcji, podczas gdy zapobieganie głodowi jest właściwością żywotności, ponieważ wymaga dla gwarancji, że każdy procesor, który chce wejść do swojej krytycznej sekcji, w końcu będzie mógł to zrobić. Co ciekawe, częściowa poprawność i właściwości terminacyjne konwencjonalnych algorytmów są szczególnymi przypadkami bezpieczeństwa i żywotności. Częściowa poprawność oznacza, że program nigdy nie może kończyć się z niewłaściwymi danymi wyjściowymi – wyraźnie jest to właściwość bezpieczeństwa – a zakończenie oznacza, że algorytm musi w końcu osiągnąć swój logiczny koniec i zakończyć – wyraźnie właściwość żywotności. Aby wykazać, że dany protokół narusza właściwość bezpieczeństwa, wystarczy wykazać

skończony, zgodny z prawem ciąg czynności, który prowadzi do sytuacji zabronionej. Dokonano tego w przypadku błędnej wersji dwuprocesorowego rozwiązania problemu z prysznicem i jest to po prostu pokazanie, że algorytm narusza częściową poprawność, wyświetlając kończącą sekwencję wykonywania z błędnymi wynikami. Zupełnie inną sprawą jest pokazanie, że zostaje naruszona właściwość życia. Musimy jakoś udowodnić istnienie nieskończonej sekwencji działań, która nigdy nie prowadzi do obiecanej sytuacji. Ta różnica jest w rzeczywistości jednym ze sposobów scharakteryzowania dwóch klas właściwości. Jeśli chodzi o ciągłą współbieżność, techniki testowania zwykle nie są zbyt pomocne. Możemy być rozsądnie przekonani o poprawności zwykłego algorytmu sortującego, wypróbowując go na wielu standardowych listach elementów oraz w niektórych przypadkach „z pogranicza", takich jak: jako listy z jednym elementem lub bez, lub listy z równymi wszystkimi elementami. Z drugiej strony wiele błędów pojawiających się w sferze systemów współbieżnych nie ma nic wspólnego z nieoczekiwanymi danymi wejściowymi. Wynikają one z nieoczekiwanych interakcji między procesorami oraz nieprzewidywalnej kolejności wykonywania akcji. Udowodnienie poprawności takich systemów jest zatem bardziej śliskie i podatne na błędy niż konwencjonalne algorytmy sekwencyjne. Wiele opublikowanych „rozwiązań" współbieżnych problemów programistycznych okazało się później zawierać subtelne błędy, które wymykały się zarówno recenzentom, jak i czytelnikom, pomimo obecności pozornie ważnych argumentów za poprawnością. Potrzeba formalnej weryfikacji jest więc tutaj bardziej dotkliwa niż w przypadku sekwencyjnym i rzeczywiście zaproponowano kilka podejść. Zazwyczaj te, które zajmują się właściwościami bezpieczeństwa, są rozszerzeniem metody pośrednio-potwierdzenia do weryfikacji częściowej poprawności, a te dostosowane do właściwości związanych z żywotnością rozszerzają metodę zbieżności do udowodnienia zakończenia. Wykazanie, że np. algorytm scalania równoległego jest częściowo poprawny, wymaga m.in. formalnego dowodu, że dwa procesory przypisane do równoległego sortowania dwóch połówek listy, na dowolnym poziomie rekurencji, nie kolidują ze sobą. inny. W tym konkretnym przypadku procesory pracujące równolegle w ogóle nie wchodzą w interakcje, a taki dowód jest stosunkowo łatwy do zdobycia. Kiedy procesory wchodzą w interakcję, jak w przypadku rozwiązania problemu z prysznicem, sprawy stają się znacznie trudniejsze. Musimy w jakiś sposób zweryfikować zachowanie każdego pojedynczego procesora w izolacji, biorąc pod uwagę wszystkie możliwe interakcje z innymi, a następnie połączyć dowody w jedną całość. Asercji pośrednich nie można tutaj używać w zwykły sposób. Twierdzenie, że twierdzenie jest prawdziwe za każdym razem, gdy zostanie osiągnięty pewien punkt w protokole procesora P, nie zależy już wyłącznie od działań P. Co więcej, nawet jeśli twierdzenie jest rzeczywiście prawdziwe w tym momencie, może stać się fałszywe przed następnym działaniem P, ponieważ inny procesor był wystarczająco szybki, aby w międzyczasie zmienić wartość jakiejś zmiennej. Pewne formalne metody dowodowe przezwyciężają ten problem, wymagając oddzielnych dowodów, aby spełnić pewien rodzaj własności wolności ingerencji, która musi być udowodniona oddzielnie. Wszystko to brzmi dość skomplikowanie. W rzeczywistości tak jest. Jednym z powodów przedstawienia jedynie nieformalnego dowodu poprawności dla prostego rozwiązania problemu wzajemnego wykluczania dwóch procesorów był raczej techniczny charakter dowodu formalnego. W zasadzie jednak istnieją odpowiednie metody dowodowe i, jak w przypadku sekwencyjnym, dowody formalne istnieją zawsze, jeśli rozwiązania są rzeczywiście poprawne, chociaż ich odkrycie jest w zasadzie nieobliczalne.

Logika temporalna

W poprzednich rozdziałach wspominano logikę dynamiczną. Są to ramy formalne, które można wykorzystać do określenia i udowodnienia różnych właściwości algorytmów. Byłoby miło, gdybyśmy mogli wykorzystać taką logikę do określania i udowadniania właściwości bezpieczeństwa i żywotności systemów współbieżnych. Jak wspomniano w części 5, centralną konstrukcją logiczną używaną w logikach dynamicznych jest after(A, F), co oznacza, że F jest prawdziwe po zakończeniu A. Taka logika

opiera się zatem na paradygmacie wejścia/wyjścia - paradygmacie odniesienia sytuacji przed wykonaniem części algorytmu do sytuacji po wykonaniu. W przypadku współbieżności niezbędna wydaje się umiejętność mówienia bezpośrednio także o tym, co dzieje się podczas egzekucji. O wiele bardziej pasującym tu formalizmem jest logika temporalna, czyli TL. Jest to odmiana logiki klasycznej znanej matematykom jako logika czasów gramatycznych, specjalnie dostosowana do celów algorytmicznych. Jej formuły zawierają stwierdzenia o prawdziwości twierdzeń w miarę upływu czasu, a także mogą wyraźnie odnosić się do aktualnego miejsca kontroli w protokołach. Dwie z centralnych konstrukcji to odtąd (F) i ostatecznie (F). Pierwsza stwierdza, że F jest prawdziwe od teraz — aż do zakończenia, jeśli protokół się zakończy, i na zawsze, jeśli tak się nie stanie — a druga mówi, że F w końcu stanie się prawdą, to znaczy w pewnym momencie w przyszłości. Przypominając nasze użycie symbolu ∼ do oznaczenia „nie”, pierwsza z tych konstrukcji może być użyta do określenia właściwości bezpieczeństwa, pisząc odtąd (∼F) (czyli F nigdy nie stanie się prawdą), a druga do określenia żywotności nieruchomości. Jako przykład rozważ rozwiązanie problemu prysznica dwuprocesorowego i następujące formuły TL:3

P1-jest -na (1.4) → ostatecznie (P1-jest na (1.5))

P2-jest-na(1.4) → ostatecznie (P2-w-(1.5))

∿ (P1-jest przy-(1.5) & P2-jest-(1.5))

Pierwsze dwie formuły stwierdzają, że jeśli procesor czeka na wejście do sekcji krytycznej, w końcu wejdzie, a trzecia stwierdza, że oba procesory nigdy nie znajdą się w swojej sekcji krytycznej jednocześnie. Wzięte razem i stwierdzone, że są prawdziwe we wszystkich punktach obliczeń (używane odtąd), formuły te zapewniają, że rozwiązanie jest poprawne. Formalnym dowodem na to jest logiczna manipulacja, która trzyma się ścisłych aksjomatycznych reguł logiki temporalnej, o których tutaj nie będziemy się rozwodzić. Jednak, aby dokonać porównania z metodami dowodowymi z rozdziału 5, okazuje się że czyste właściwości bezpieczeństwa (takie jak w trzecim wzorze powyżej) mogą być udowodnione w sposób podobny do metody niezmiennej asercji dla częściowej poprawności. Ewentualności (takie jak te w pierwszych dwóch wzorach) można udowodnić, najpierw analizując każdy możliwy pojedynczy krok protokołów, a następnie stosując indukcję matematyczną, aby wydedukować ewentualności, które stają się prawdziwe w ciągu pewnej liczby kroków od tych, które stają się prawdziwe w mniej. Jak wspomniano, jeśli protokoły są rzeczywiście poprawne, istnieje dowód poprawności, a jeśli zostanie znaleziony, jego części można sprawdzić algorytmicznie. W rzeczywistości w wielu zastosowaniach logiki temporalnej każda zmienna ma tylko skończoną liczbę możliwych wartości (omówiony wcześniej protokół wzajemnego wykluczania wykorzystuje trzy zmienne, każda z tylko dwiema możliwymi wartościami). Te protokoły skończonych stanów, jak się je czasami nazywa, można zweryfikować algorytmicznie. Oznacza to, że istnieją algorytmy, które akceptują jako dane wejściowe pewne rodzaje protokołów i formułę logiki temporalnej zapewniającą poprawność i weryfikują te pierwsze z tymi drugimi. Tak więc, o ile znalezienie dowodów na poprawność transformacyjnych lub obliczeniowych części współbieżnych systemów nie jest na ogół możliwe, możliwe jest skuteczne znalezienie dowodów na części kontrolne, takie jak mechanizmy harmonogramowania procesorów i zapobiegania głodowaniu i zakleszczeniu, jak zazwyczaj dotyczą one tylko zmiennych o skończonym zakresie. Jednak te automatyczne weryfikatory nie zawsze są tak wydajne, jak byśmy chcieli, ponieważ między innymi liczba kombinacji wartości rośnie wykładniczo wraz z liczbą procesorów. Dlatego nawet krótkie i niewinnie wyglądające protokoły mogą być dość trudne do automatycznej weryfikacji, a w ręcznie tworzonych dowodach oczywiście subtelne błędy są regułą, a nie wyjątkiem. W ciągu ostatnich kilku lat opracowano potężne metody weryfikacji systemów współbieżnych względem formuł logiki temporalnej. Korzystając na przykład ze sprawdzania modelu, można konstruować wspomagane komputerowo dowody, że system spełnia formuły logiki

temporalnej, w tym właściwości bezpieczeństwa i żywotności. Może to stanowić pewną niespodziankę, biorąc pod uwagę nierozstrzygalność weryfikacji, o której była mowa w części 8. Co więcej, nawet jeśli ograniczymy się do systemów skończonych, np. zabraniając zmiennym przyjmowania wartości powyżej pewnej skończonej granicy, a poprzez ograniczenie a priori liczby elementów w systemie, odpowiednie problemy weryfikacyjne są nie do rozwiązania. Mimo to istnieją sposoby radzenia sobie z tymi problemami, które działają wyjątkowo dobrze w wielu przypadkach pojawiających się w praktyce. Rzeczywiście istnieje duża nadzieja na weryfikację, nawet w śliskiej sferze współbieżności.

Uczciwość i systemy czasu rzeczywistego

W tym miejscu warto wspomnieć o dwóch dodatkowych kwestiach, choć żadnej z nich nie będziemy szczegółowo omawiać. Pierwsza dotyczy globalnego założenia, które jest zwykle przyjmowane w przypadku bieżącej współbieżności. Zazwyczaj nie przyjmujemy żadnych założeń dotyczących względnej szybkości procesorów biorących udział w rozwiązywaniu problemów współbieżności, ale zakładamy, że wszystkie odpowiadają i robią postęp w skończonym, choć prawdopodobnie długim czasie. W naszych rozwiązaniach problemu wzajemnego wykluczania użyliśmy oczywiście tego założenia, ponieważ w przeciwnym razie P_1 mógłby osiągnąć klauzulę (1.3) i nigdy nie przejść do ustawienia Z na 1 i kontynuować. Założenie to jest czasami nazywane uczciwością, ponieważ jeśli myślisz o współbieżności jako implementowanej przez centralny procesor, czasami nazywany harmonogramem, który daje każdemu z procesorów współbieżnych kolejkę w wykonywaniu kilku instrukcji, to mówisz, że każdy procesor w końcu wykonuje postęp jest jak powiedzenie, że symulujący procesor jest sprawiedliwy wobec wszystkich, ostatecznie dając każdemu swoją kolej. Rozważ następujące protokoły dla dwóch procesorów P i Q.

protokół dla P:

(1) X ← 0;

(2) wykonaj następujące czynności raz za razem:

(2.1) jeśli Z = 0 to zatrzymaj się;

(2.2) X ← X + 1.

protokół dla Q:

(1) Z 0;

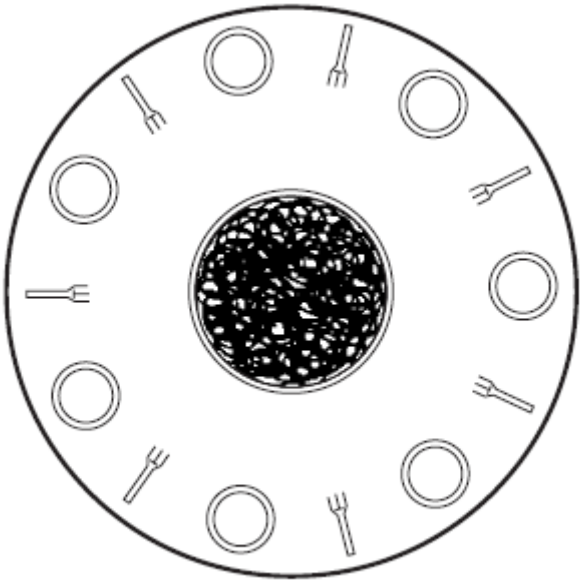
(2) zatrzymaj się.

Niech wartość początkowa Z wynosi 1. Jeśli nie przyjmiemy uczciwości, P może stale mieć prawo do kontynuacji, a słabe Q pozostaje na zawsze wykluczone z gry. To oczywiście prowadzi do nieskończonej pętli. Jedynym sposobem na zapewnienie zakończenia protokołów jest umożliwienie Q postępu. Z drugiej strony, przy założeniu słuszności, ten współbieżny algorytm kończy działanie, ustawiając X na jakąś nieznaną, prawdopodobnie bardzo dużą, nieujemną liczbę całkowitą. Sposób, w jaki to robi, jest następujący. Program planujący może dowolnie zaplanować kolejność P i Q, a w szczególności może pozwolić P na wiele razy, zanim wpuści Q po raz pierwszy. Musi jednak pozwolić Q na zmianę w pewnym momencie, zgodnie z założeniem uczciwości. W ten sposób P zwiększy X od 0 nieznaną liczbę razy, ale gdy Q dostanie, wykonanie zielonego światła zakończy się. Oczywiście ostateczna wartość X może być dowolną liczbą od zera w górę, ale proces w końcu się zakończy w wyniku sprawiedliwości. Oczywiście, że protokoły te mogą generować bardzo duże liczby, opiera się na założeniu, że niektóre procesory mogą być dowolnie wolniejsze niż inne. Czasami przydatne są silniejsze pojęcia uczciwości, zwłaszcza te, które ograniczają opóźnienie, które może wystąpić między akcjami. Możemy powiedzieć,

że Q ma wolniejszą reakcję niż P, ale nie jest gorsza niż dwa razy wolniej. Takie twierdzenia o uczciwości mają charakter ilościowy i wprowadzają do gry problemy z wyczuciem czasu. Dodatkowa komplikacja pojawia się w systemach, dla których ograniczenia czasowe mają kluczowe znaczenie, zwłaszcza w systemach czasu rzeczywistego. Są one zobowiązane do natychmiastowej reakcji na określone zdarzenia, a przynajmniej w niezauważalnym i znikomym czasie. Przykłady obejmują kontrolę lotu, naprowadzanie pocisków i systemy szybkiej komunikacji. Tutaj również może pomóc logika temporalna, wykorzystująca specjalny operator następnego kroku, który mówi o tym, jaka będzie prawdziwa jednorazowa jednostka czasu od chwili obecnej. Możliwe jest na przykład napisanie formuły TL, która stwierdza, że jeśli jeden dany fakt jest prawdziwy, inny stanie się prawdziwy, powiedzmy, 10 kroków później. To nadal nie rozwiązuje problemów programowania takich systemów, które zwykle są nie tylko krytyczne czasowo, ale także duże i złożone.

Problem jedzących filozofów

Jednym z najpopularniejszych przykładów ciągłej współbieżności, który ilustruje wiele zagadnień synchronizacji i współpracy, jest następujący. Mamy stół, wokół którego zasiada N-filozofów. Pośrodku duży talerz zawierający nieograniczoną ilość spaghetti. W połowie drogi między każdą parą sąsiednich filozofów znajduje się pojedyncze rozwidlenie



Teraz, skoro nikt, nawet filozof, nie może jeść spaghetti jednym widelcem, pojawia się problem. Pożądany cykl życia filozofa polega na wykonywaniu jego prywatnych czynności (np. myślenie, a następnie zapisywanie wyników do publikacji), odczuwanie głodu i próby jedzenia, jedzenia, a następnie powrót do prywatnych zajęć, ad infinitum. (Podobny, ale nieco trudniejszy do opanowania problem dotyczy chińskich filozofów, w których spaghetti i widelce zastępują ryż i pałeczki). Jak filozofowie powinni odprawiać swoje rytuały bez głodowania? Możemy poinstruować ich, aby po prostu podnieśli widelce po obu stronach, gdy są głodni, i jedzą, a po zakończeniu ponownie odkładają widelce. To rozwiązanie nie zadziała, ponieważ jeden lub oba widelce mogą zostać przejęte przez sąsiednich filozofów w tym czasie. Ponadto dwóch sąsiednich filozofów może próbować podnieść ten sam widelec w tym samym czasie. Używanie widelców, które są poza zasięgiem filozofa, jest zabronione. Tutaj jedzenie można uznać za część krytyczną, ponieważ dwóch sąsiednich filozofów nie

może jeść jednocześnie, a widelce są swego rodzaju kluczowymi zasobami, ponieważ nie mogą być używane przez dwóch filozofów jednocześnie. Ten problem jest również typowy dla wielu rzeczywistych sytuacji, takich jak na przykład komputerowe systemy operacyjne, w których wiele procesorów konkuruje o pewne współdzielone zasoby. Schemat połączeń takich systemów jest zwykle dość rzadki (nie wszystkie zasoby są dostępne dla każdego procesora), a liczba zasobów jest zbyt mała, aby wszyscy byli razem szczęśliwi. Jednym z rozwiązań tego problemu jest wprowadzenie do gry nowego gracza, portiera z jadalni. Filozofom instruuje się, aby wychodzili z pokoju, gdy nie są zainteresowani jedzeniem, i próbują wejść ponownie, gdy są. Portier ma obowiązek liczyć filozofów aktualnie przebywających w pokoju, ograniczając ich liczbę do N-1. Oznacza to, że jeśli w pokoju znajduje się wszystkich filozofów oprócz jednego, ten ostatni będzie czekał przy drzwiach, aż ktoś wyjdzie. Otóż, jeśli w danym momencie przy stole siedzi co najwyżej N - 1 filozofów, to jest co najmniej dwóch filozofów, którzy nie mają co najmniej jednego sąsiada, a zatem co najmniej jeden filozof może jeść. (Dlaczego?) W przypadku odpowiedniego sformalizowania rozwiązanie to może okazać się satysfakcjonujące. Wykorzystuje jednak dodatkową osobę (której miesięczna pensja będzie prawdopodobnie większa niż suma potrzebna na zakup dodatkowego zestawu widelców, przynajmniej za rozsądnie małe N). Czy problem kulinarnych filozofów można rozwiązać bez odźwiernego i bez uciekania się do pamięci wspólnej lub jej odpowiedników? Innymi słowy, czy istnieje w pełni rozproszone, w pełni symetryczne rozwiązanie problemu, które nie wymaga żadnych dodatkowych procesorów? W tym przypadku „w pełni rozproszony” oznacza, że nie ma centralnej pamięci współdzielonej, a protokoły mogą wykorzystywać tylko zmienne rozproszone (współdzielone, powiedzmy, tylko przez dwóch sąsiednich filozofów). „W pełni symetryczny” oznacza, że protokoły dla wszystkich filozofów są zasadniczo identyczne — nie pozwalamy różnym filozofom działać inaczej i nie pozwalamy im zaczynać od różnych wartości w swoich zmiennych. Te warunki mogą wydawać się zbyt restrykcyjne. Jeśli jednak filozofowie mają różne programy lub różne wartości początkowe, to tak, jakby powiedzieć, że wykorzystują swoją osobistą wiedzę i talenty w przyziemnym poszukiwaniu pożywienia. Chcielibyśmy to wszystko zachować na myślenie i publikowanie; jedzenie powinno być procedurą standardową i wspólną dla wszystkich. Odpowiedź na te pytania brzmi: nie. Do rozwiązania problemu potrzebujemy czegoś więcej, np. pamięci współdzielonej, bezpośredniej komunikacji między procesorami, specjalnego scentralizowane sterowanie procesorami lub wykorzystanie różnych informacji dla różnych filozofów.

Dlaczego? Argument jest naprawdę prosty. Pomyśl o prawidłowym rozwiązaniu jako takim, które gwarantuje wszystkie pożądane właściwości, nawet w przypadku najbardziej złośliwego programu planującego (to znaczy nawet w przypadku programu planującego, który stara się tak bardzo, jak tylko może, spowodować zakleszczenie lub zagłodzenie). Innymi słowy, aby pokazać, że nie ma w pełni rozłożonego, w pełni symetrycznego rozwiązania, wystarczy wykazać jedną konkretną kolejność, w jakiej filozofowie są zaplanowani do wykonywania działań kandydata na rozwiązanie, a następnie pokazanie, że coś musi pójść nie tak. Załóżmy zatem, że mamy jakieś takie kandydujące rozwiązanie. Załóżmy również, że ponumerowaliśmy filozofów 1, 2, . . . , N. (Sami filozofowie nie są świadomi nawet swoich liczb; nie mogą mieć żadnych prywatnych informacji, które mogą okazać się przydatne). Harmonogram, który przyjmujemy, jest następujący. Na każdym etapie każdy filozof w kolejności 1, 2, . . . , N może wykonać jedną podstawową akcję. Akcja może być zajętym testem oczekiwania, w takim przypadku testowanie i decydowanie, czy czekać, czy kontynuować, są traktowane razem jako jedno niepodzielne działanie podstawowe. Można wykazać, że na każdym etapie dokładnie to samo działanie wykonuje każdy procesor. Wynika to z faktów, że zarówno sytuacja wyjściowa a protokoły są w pełni symetryczne, nie ma procesorów innych niż sami filozofowie, a tabela i jej zawartość są symetrycznie cykliczne. W konsekwencji sytuacja na końcu każdego etapu będzie nadal w pełni symetryczna; to znaczy, że wartości zmiennych będą takie same dla wszystkich filozofów, podobnie jak ich lokalizacje

w protokołach. Czy jakikolwiek filozof mógł jeść na jednym lub kilku etapach? Nie, ponieważ etapy obejmują tylko podstawowe czynności, a pod koniec etapu można wykryć fakt, że filozof je lub jadł. Ale niemożliwe jest, aby wszyscy filozofowie jedli na raz, a proces jedzenia dwóch filozofów, jeden po drugim, nie może być przeprowadzony w jednym etapie. Z niezbędnej symetrii wynika zatem, że na końcu każdego etapu żaden filozof nie będzie jadł. Dlatego nikt nigdy nie będzie. W części 11 zobaczymy, że problem filozofów jedzenia można rozwiązać w sposób w pełni rozproszony i w pełni symetryczny, ale z bardziej liberalnym pojęciem poprawności.

Semafory

Istnieje wiele języków programowania, które w taki czy inny sposób obsługują współbieżność, w tym języki zorientowane obiektowo i formalizmy wizualne do rozwoju systemu, z rodzaju omówionego później. Nie będziemy tutaj opisywać żadnego z tych języków, ale pokrótce rozważymy jedną z głównych konstrukcji wymyślonych specjalnie do radzenia sobie ze współbieżnością. Stanowi on podstawę części sposobu implementacji współbieżności w niektórych z tych języków i może być używany jawnie w innych. Wiemy już, że pamięć współdzielona i komunikacja bezpośrednia reprezentują dwa główne podejścia do opisywania współpracy, jaka ma zachodzić między równolegle realizowanymi procesami. Jeśli używa się tego pierwszego, musi istnieć mechanizm rozwiązywania konfliktów pisarskich, które wywołuje pamięć współdzielona. Jednym z najpopularniejszych z nich jest semafor.

Semafor to specjalny rodzaj elementu programistycznego, który wykorzystuje dwie operacje do kontrolowania użycia sekcji krytycznych (takich jak te, które wiążą się z zapisem w części pamięci współdzielonej). Próba wejścia do takiej sekcji jest reprezentowana przez operację żądania, a zwolnienie oznacza wyjście. Semafor S może właściwie być postrzegany jako zmienna o wartości całkowitej. Wykonywanie żądania(S) to niepodzielna akcja, która próbuje zmniejszyć S o 1, robiąc to bez przerwy, jeśli jego wartość jest dodatnia i czeka, aż stanie się dodatnia w przeciwnym razie. Skutkiem zwolnienia (S) jest po prostu zwiększenie S o 1. Ważne jest to, że z samej definicji semafor S poddaje się tylko jednemu żądaniu lub operacji zwolnienia na raz. W typowym użyciu semaforów w celu osiągnięcia wzajemnego wykluczenia, S otrzymuje wartość początkową 1, a sekcja krytyczna każdego procesora jest zawarta w następujący sposób:

. . .

upraszanie (S);

przeprowadzić sekcję krytyczną;

zwolnienie(S);

. . .

Powoduje to dopuszczenie tylko jednego procesora na raz do jego krytycznej sekcji. Pierwszy, który próbuje wejść, udaje się zmniejszyć S do 0 i wchodzi. Pozostali muszą poczekać, aż pierwszy z nich wyjdzie, zwiększając w tym procesie S do 1. Semafor, który zaczyna się od 1, a zatem przyjmuje tylko wartości 1 i 0, nazywany jest semaforem binarnym. Prostym sposobem użycia semaforów dla sekcji krytycznych, które mogą obsłużyć do K procesorów na raz, jest użycie niebinarnego semafora o początkowej wartości K. (Dlaczego ma to pożądany efekt?) Na przykład w problemie dotyczącym jadalni , portier może być modelowany przez semafor o wartości początkowej N - 1, kontrolujący sekcję krytyczną, która obejmuje czynności związane z próbą jedzenia, jedzeniem i opuszczeniem pokoju. Użycie każdego widelca może być modelowane przez semafor binarny. Warto zauważyć, że ta prosta definicja semaforów nie zakłada, że wszystkie procesory oczekujące na zablokowane operacje żądania

ostatecznie otrzymają prawo do wejścia, gdy S staje się niezerowe. Jest całkiem możliwe, że złośliwa implementacja semaforów zawsze daje pierwszeństwo najnowszemu procesorowi, blokując inne na zawsze. To jeden z przypadków, w których pewna uczciwość założenia wydaje się konieczne, dzięki czemu, powiedzmy, każdy oczekujący procesor ma zagwarantowany ewentualny postęp. Semafory można zatem opisać jako bardzo prosty typ danych, którego operacje (inkrementacja, test-i-dekrementacja) są chronione przed konfliktami zapisu przez wbudowany mechanizm wzajemnego wykluczania. Wzajemne wykluczanie dla bardziej skomplikowanych typów danych, które wykorzystują wiele operacji, można osiągnąć przez otoczenie każdego wystąpienia a operacja pisania z odpowiednimi operacjami na semaforach. Jednak w pewnym sensie semafory są jak wypowiedzi goto; zbyt wiele operacji żądań i wydawania rozrzuconych po całym długim programie może stać się niejasne i podatne na błędy. Semafory mogą być używane do rozwiązywania standardowych rodzajów problemów w programowaniu współbieżności, ale stanowią niestrukturalną konstrukcję programistyczną.

Badania nad równoległością i współbieżnością

Jeśli w poprzednich rozdziałach stwierdziliśmy, że prowadzone są intensywne badania nad wieloma omawianymi tematami, to jest to prawdziwsze niż kiedykolwiek w dziedzinie paralelizmu i współbieżności. Badacze starają się pogodzić praktycznie ze wszystkimi aspektami współpracy algorytmicznej i nie będzie przesadą stwierdzenie, że badania większości informatyków mają jakiś związek z poruszanymi w tej części tematami. Prowadzone są intensywne badania nad znajdowaniem szybkich algorytmów równoległych dla różnych problemów algorytmicznych, a rozwiązania wykorzystują szerokie spektrum wyrafinowanych struktur danych i mechanizmów współbieżności. Wiele problemów (takich jak obliczenia gcd) oparło się próbom użytecznego wykorzystania wszelkiego rodzaju równoległości. Obwody Boole'a i sieci skurczowe są również przedmiotem wielu aktualnych badań i odkryto interesujące powiązania między tymi podejściami a konwencjonalną algorytmiką sekwencyjną. Abstrakcyjna teoria złożoności paralelizmu stawia wiele ważnych i najwyraźniej bardzo trudnych, nierozwiązanych pytań dotyczących klas takich jak NC i PSPACE, z których niektóre zostały opisane wcześniej. Podczas gdy, jak widzieliśmy, sekwencyjność stwarza już wiele nierozwiązanych problemów, równoległość niewątpliwie rodzi o wiele więcej. W rzeczywistości wydaje się jasne, że rozumiemy podstawy algorytmów sekwencyjnych znacznie lepiej niż algorytmów równoległych, a przed nami długa i trudna droga. Inne tematy aktualnych badań obejmują równoległe projektowanie komputerów, techniki sprawdzania i analizy w celu wnioskowania o współbieżnych procesach oraz tworzenie użytecznych i wydajnych języków programowania współbieżnego. Jak ilustruje historia budowy domu na początku rozdziału, współbieżność jest faktem i im lepiej ją rozumiemy, tym bardziej możemy ją wykorzystać na naszą korzyść. W pewnym sensie najnowsze postępy naukowe i technologiczne w zakresie współbieżności wyprzedzają się. Wiele z najlepszych znanych algorytmów równoległych nie może zostać zaimplementowanych, ponieważ istniejące komputery równoległe są w jakiś sposób nieodpowiednie. Z drugiej strony wciąż nie wiemy wystarczająco dużo o projektowaniu współbieżnych programów i systemów, aby w pełni wykorzystać funkcje oferowane przez te same komputery. Prace są jednak kontynuowane i ciągle osiągane są znaczące wyniki, chociaż głębokie kwestie związane z prawdziwą złożonością paralelizmu pozostają nieuchwytne. Przejdziemy teraz do dwóch nowszych podejść do równoległości, które atakują go z zupełnie innych punktów widzenia.

Obliczenia kwantowe

Więc co to za modne nowe rzeczy do obliczeń kwantowych? Cóż, jest to głęboki i skomplikowany temat, oparty na złożonym materiale matematycznym i fizycznym, a przez to bardzo trudny do opisania w sposób wyjaśniający tej książki. Obliczenia kwantowe opierają się na mechanice kwantowej, niezwykłym temacie we współczesnej fizyce, który niestety jest śliski i trudny do uchwycenia i często

jest sprzeczny z intuicją. Naiwna próba wykorzystania ziemskiego zdrowego rozsądku, aby to zrozumieć, może łatwo stać się przeszkodą w zrozumieniu, a nie pomocą. Kolejne sekcje będą więc traktować ten temat wyjątkowo powierzchownie, nawet stosując standardy tej techniczności, unikając książki. Przepraszamy za to. Notatki bibliograficzne zawierają jednak kilka wskazówek do badań w literaturze dla bardziej dociekliwych, biegłych matematycznie czytelników. Po jaśniejszej stronie jest szansa - że komputery kwantowe mogą przynieść dobre wieści. Jak, dlaczego i kiedy są pytania, którymi postaramy się odpowiedzieć, bardzo krótko, w dalszej części pracy. Jedną z głównych zalet fizyki kwantowej jest jej zdolność do zrozumienia pewnych zjawisk eksperymentalnych na poziomie cząstek, których fizyka klasyczna wydawała się nie być w stanie. Dwie z głównych ciekawostek świata kwantowego, o których mówi się bardzo nieformalnie, to fakt, że cząstki nie można już uważać za znajdujące się w jednym miejscu w przestrzeni w określonym czasie oraz że jej sytuacja (w tym lokalizacje) może się zmienić w wyniku po prostu to obserwując. Pierwsza z nich wydaje się dobrą wiadomością dla komputerów: czy nie bylibyśmy w stanie wykorzystać właściwości przebywania w wielu miejscach razem, aby przeprowadzić masową równoległość obliczeń? Druga jednak wydaje się złą wiadomością: próba „zobaczenia" lub „dotknięcia" wartości podczas obliczeń, powiedzmy, aby przeprowadzić porównanie lub aktualizację, może w nieprzewidywalny sposób zmienić tę wartość! Obliczenia kwantowe to bardzo nowy pomysł. Wczesne prace były motywowane twierdzeniem, że gdyby można było zbudować komputer działający zgodnie z prawami fizyki kwantowej, a nie klasycznej, można by uzyskać wykładnicze przyspieszenie niektórych obliczeń. Komputer kwantowy, podobnie jak klasyczny, ma być oparty na jakimś elemencie skończonym, analogicznym do klasycznego bitu dwustanowego. Kwantowy analog bitu, zwany kubitem i wymawiany jako „bit kolejki", można wyobrazić sobie fizycznie na wiele sposobów: zgodnie z kierunkiem polaryzacji fotonu (pozioma lub pionowa), przez spin jądrowy (specjalny dwuwartościowy kwant obserwowalne) lub przez poziom energii atomu (ziemnego lub wzbudzonego). Dwa tak zwane stany bazowe kubitu, analogiczne do 0 i 1 zwykłego bitu, są oznaczone odpowiednio przez $|0>$ i $|1>$. To, czego nie mamy w systemie kwantowym, to proste, deterministyczne pojęcie, że kubit znajduje się w takim czy innym stanie bazowym. Jego pojęcie bycia lub nie bycia jest raczej nieokreślone: wszystko, co możemy powiedzieć o statusie kubitu, to to, że znajduje się on w obu stanach jednocześnie, z których każdy ma pewne „prawdopodobieństwo". Ale jakby celowo uczynić rzeczy jeszcze mniej zrozumiałymi dla śmiertelników, nie są to zwykłe, dodatnie prawdopodobieństwa, jak bycie w stanie $|0>$ z prawdopodobieństwem 1/4 i w $|1>$ z prawdopodobieństwem 3/4 . Te „prawdopodobieństwa" mogą być ujemne, a nawet urojone (tj. liczby zespolone, które zawierają pierwiastki kwadratowe z liczb ujemnych), a wynikowy stan kombinacji nazywany jest superpozycją. Gdy już „przyjrzymy się" kubitowi, czyli dokonamy pomiaru, nagle decyduje, gdzie być, widzimy go w jednym lub drugim stanie bazowym, prawdopodobieństwa znikają, a superpozycja zostaje zapomniana. Ten rodzaj „wymuszonej dyskrecji" prowadzi do przymiotnika „kwant". To tyle, jeśli chodzi o pojedynczy kubit. Co dzieje się z wieloma kubitami razem, obok siebie, których potrzebujemy jako podstawy do prawdziwych obliczeń kwantowych? Jak połączone są stany kilku kubitów, aby uzyskać złożony stan całego urządzenia obliczeniowego? W klasycznym przypadku każdy zbiór N bitów, z których każdy może być w dwóch stanach 0 lub 1, powoduje powstanie 2^N stanów złożonych. W kwantowym świecie kubitów również zaczynamy od 2^N stanów złożonych zbudowanych ze stanów bazowych N kubitów (na przykład w przypadku dwóch kubitów cztery stany złożone są oznaczone $|00>$, $|01>$, $|10>$ i $|11>$). Następnie stosujemy do nich złożone kombinacje, tak jak w przypadku pojedynczego kubitu. Jednak w tym przypadku sposób zdefiniowania kombinacji powoduje dodatkowy kluczowy skręt zwany odpowiednio splątaniem: niektóre stany złożone są czystymi kompozytami, które można uzyskać – za pomocą operacji zwanej „iloczynem tensorowym" – ze stanów oryginalne kubity, ale niektóre nie; są uwikłane. Splątane kubity, termin, który pojawia się w precyzyjnym matematycznym ujęciu, stanowią z natury nierozłączny „misz-masz" oryginalnych kubitów. Mają dziwną właściwość natychmiastowej

komunikacji: obserwowanie jednego, a tym samym naprawianie jego stanu, powoduje, że drugi jednocześnie blokuje się w podwójnym stanie, bez względu na to, jak daleko się od siebie znajdują. Splątanie okazuje się być podstawowym i niezbędnym pojęciem w obliczeniach kwantowych, ale niestety dalsze omówienie ich technicznych aspektów i sposobu, w jaki jest ono wykorzystywane w samych obliczeniach, wykracza poza zakres tej książki.

Algorytmy kwantowe

Co ludzie byli w stanie zrobić z obliczeniami kwantowymi? Z góry należy podać kilka faktów. Po pierwsze, pełne obliczenia kwantowe ogólnego przeznaczenia obejmują obliczenia klasyczne. Oznacza to, że jeśli i kiedy zostanie zbudowany, komputer kwantowy będzie w stanie emulować klasyczne obliczenia bez znaczącej straty czasu. Po drugie, choć pozornie słabszy, klasyczny komputer może symulować dowolne obliczenia kwantowe, ale może to pociągać za sobą wykładniczą stratę czasu. Fakt, że taka symulacja jest możliwa, oznacza, że obliczenia kwantowe nie mogą zniszczyć tezy Churcha/Turinga: obliczalność pozostaje nienaruszona również w świecie obliczeń kwantowych. Jeśli i kiedy zostaną zbudowane rzeczywiste komputery kwantowe, nie będą w stanie rozwiązać problemów, których bez nich nie da się rozwiązać. To powiedziawszy, głównym pytaniem jest, czy wykładnicza strata czasu w drugim stwierdzeniu jest rzeczywiście nie do pokonania. Podobnie jak w przypadku równoległości, pytamy, czy w świecie kwantowym istnieją nierozwiązywalne problemy, które stają się wykonalne. To znaczy, czy istnieje problem z dolną granicą wykładniczą czasu wykładniczego w klasycznych modelach obliczeń, które mają algorytm kwantowy w czasie wielomianowym? I tutaj, jeśli używamy QP do oznaczenia quantum-PTIME, mamy:

PTIME ⊆ QP ⊆ PSPACE (= równolegle-PTIME)

Zatem rozsądny, tj. wielomianowy, czas kwantowy leży w tym samym miejscu co NP, tj. między rozsądnym czasem deterministycznym a rozsądną przestrzenią pamięci. Niestety, tak jak poprzednio, nie wiemy, czy któraś z tych inkluzji jest ścisła. Pomijając złożoność obliczeniową i niezależnie od technologicznego problemu faktycznego zbudowania komputera kwantowego, dokonano już kilku niezwykle ekscytujących postępów w algorytmice kwantowej. Oto niektóre z najważniejszych wydarzeń. Rzeczywiście osiągnięto równoległość kwantową, w której superpozycja wejść jest wykorzystywana do wytworzenia superpozycji wyjść. Co ciekawe, chociaż wydaje się, że rzeczywiście wykonuje się wiele rzeczy równolegle, wyników nie można naiwnie oddzielić i odczytać z ich superpozycji; każda próba odczytu lub pomiaru da tylko jeden wynik, a reszta zostanie po prostu utracona. Potrzebny jest algorytm, aby sprytnie obliczyć wspólne właściwości wspólne dla wszystkich wyników i poradzić sobie z nimi. Przykłady mogą obejmować pewne zagregowane wartości arytmetyczne wyników liczbowych lub „i" i „lub" logicznych wyników tak/nie . Później odkryto dość zaskakujący algorytm kwantowy do wyszukiwania na nieuporządkowanej liście, powiedzmy w dużej bazie danych. Zamiast około N operacji, element można znaleźć tylko z √N operacjami (pierwiastek kwadratowy z N). Jest to sprzeczne z intuicją, prawie paradoksalne, ponieważ wydaje się konieczne przynajmniej spojrzenie na wszystkie wejścia N, aby dowiedzieć się, czy rzeczywiście tam jest to, czego szukasz.

Jednak wielką niespodzianką, a nawet szczytem dotychczasowych algorytmów kwantowych, jest algorytm faktoryzacji Shora. Wielokrotnie wspominaliśmy o faktoryzacji w książce, a jego znaczenie jako centralnego problemu algorytmicznego jest niepodważalne. Jak widzieliśmy, faktoring nie okazał się jeszcze wykonalny w zwykłym sensie – nie wiadomo, czy jest w PTIME (co, jak teraz wiemy, nie dotyczy testowania pierwszości) i sam fakt, że się pojawia trudność obliczeniowa odgrywa kluczową rolę w kryptografii, jak zobaczymy w części 12. Do tego stopnia, że znaczna część ścian, które podtrzymują współczesną kryptografię, zawaliłaby się, gdyby dostępny był wydajny algorytm

faktoryzacji . Na tym tle należy spojrzeć na znaczenie tej pracy, która dostarcza wielomianowego algorytmu kwantowego dla problemu. Aby docenić subtelność faktoryzacji kwantowej, rozważ naiwny algorytm, który próbuje znaleźć czynniki liczby N metodą prób i błędów, przechodząc przez wszystkie pary potencjalnych czynników i mnożąc je, aby sprawdzić, czy ich iloczyn jest dokładnie N. Dlaczego nie możemy to zrobić za pomocą równoległości kwantowej na wielką skalę? Moglibyśmy użyć zmiennych kwantowych do zachowania superpozycji wszystkich czynników kandydujących (powiedzmy, wszystkich liczb od 0 do N − 1), a następnie obliczyć równolegle, w najlepszym duchu kwantowym, wszystkie iloczyny wszystkich możliwych par tych liczb. mógł następnie spróbować sprawdzić, czy istnieje para, która wykonała zadanie. Niestety, to by nie zadziałało, ponieważ przyjrzenie się – to znaczy wykonanie pomiaru – tego ogromnego nałożonego wyniku nie powiedziałoby wiele. Może się zdarzyć, że trafimy na faktoryzację, ale możemy też trafić na dowolny inny spośród wielu produktów, które różnią się od N. Jak już wspomnieliśmy, gdy zmierzysz, to właśnie zobaczysz, a reszta jest stracona. Tak więc samo mieszanie wielu informacji nie wystarczy. Okazuje się, że trzeba wszystko tak zaaranżować, żeby była ingerencja. Jest to pojęcie kwantowe, zgodnie z którym możliwe rozwiązania „walczą" ze sobą o dominację w subtelny sposób. Te, które okażą się nie dobrymi rozwiązaniami (w naszym przypadku pary liczb, których iloczyn nie jest N) będą kolidować destrukcyjnie w superpozycji, a te, które są dobrymi rozwiązaniami (ich iloczynem jest N) będą przeszkadzać konstruktywnie. Wyniki tej walki pokażą się jako zmienne amplitudy na wyjściu, tak więc pomiar superpozycji wyjścia da dobrym rozwiązaniom znacznie lepszą szansę na pojawienie się. Należy zauważyć, że to liczby ujemne w definicji superpozycji umożliwiają tego rodzaju interferencję w algorytmie kwantowym. Łatwiej to powiedzieć niż zrobić, i właśnie tutaj matematyka obliczeń kwantowych staje się skomplikowana i wykracza poza zakres i poziom naszej prezentacji. Ale możemy powiedzieć, że osiągnięto właściwy rodzaj splątania dla faktoringu. Sam algorytm jest dość niezwykły, zarówno pod względem techniki, jak i, jak zobaczymy później, w jego rozgałęzieniach. Jego wydajność czasowa jest z grubsza sześcienna, czyli niewiele większa niż M^3, gdzie M jest liczbą cyfr w numerze wejściowym N. Dla bardziej zainteresowanego technicznie czytelnika algorytm wykorzystuje wydajną metodę kwantową do obliczania kolejności liczby Y modulo N, to znaczy znaleźć najmniejszą liczbę całkowitą a taką, że $Y^a = 1 \pmod N$. Wiadomo, że jest to wystarczające, aby umożliwić szybkie faktoring, a resztę pracy wykonuje się przy użyciu konwencjonalnych algorytmów. Algorytm ten nie zamienił jeszcze trudnego do udowodnienia problemu w wykonalny z dwóch powodów, z których jeden wielokrotnie wspominaliśmy, a jeden o którym wspomnieliśmy, ale wkrótce zajmiemy się bardziej szczegółowo. Po pierwsze, faktoring nie jest trudny do wyegzekwowania; po prostu nie byliśmy w stanie znaleźć dla niego algorytmu wielomianowego. Przypuszcza się, że jest to trudne, ale nie jesteśmy pewni. Po drugie, praktyczne trudności związane ze zbudowaniem komputera kwantowego są naprawdę ogromne.

Czy może istnieć komputer kwantowy?

Omawiając wcześniej równoległość, zauważyliśmy, że istnieje pewna niezgodność między istniejącymi algorytmami równoległymi a równoległymi komputerami, które zostały zbudowane do ich obsługi. Aby mogły zostać skutecznie zaimplementowane, wiele znanych algorytmów wymaga funkcji sprzętowych, które nie są jeszcze dostępne, i, z drugiej strony, teoria algorytmów równoległych musi jeszcze dogonić to, co jest w stanie zrobić dostępny sprzęt. W dziedzinie obliczeń kwantowych sytuacja jest mniej symetryczna. Mamy do dyspozycji kilka naprawdę fajnych algorytmów kwantowych, ale żadnych maszyn, na których można by je uruchamiać. Czemu? Znowu sprawa ta kręci się wokół głębokich szczegółów technicznych, ale tym razem barierą uniemożliwiającą szczegółowe przedstawienie nie jest matematyka, ale fizyka. Tak więc, ponownie, przedstawimy tylko bardzo krótką relację, a zainteresowany czytelnik będzie musiał szukać więcej informacji gdzie indziej. Jaki jest problem? Dlaczego nie możemy zwiększyć skali? Pomimo faktu, że same algorytmy kwantowe, a w szczególności ten faktoring, są zaprojektowane do pracy zgodnie z rygorystycznymi i powszechnie akceptowanymi

zasadami fizyki kwantowej, istnieją poważne problemy techniczne związane z samą konstrukcją komputera kwantowego. Po pierwsze, fizykom eksperymentalnym nie udało się połączyć nawet niewielkiej liczby kubitów (powiedzmy 10) i kontrolować ich w jakiś rozsądny sposób. Trudności wydają się wykraczać poza dzisiejsze techniki laboratoryjne. Szczególnie niepokojącym problemem jest dekoherencja: nawet jeśli udałoby się zebrać dużą liczbę kubitów i sprawić, by same zachowywały się ładnie, rzeczy, które znajdują się w pobliżu układu kwantowego, mają natarczywy zwyczaj wpływania na niego. Zachowanie kwantowe wszystkiego, co otacza komputer kwantowy - obudowa, ściany, ludzie, klawiatura, cokolwiek! - potrafi zepsuć delikatną konfigurację konstruktywnej i destrukcyjnej ingerencji w obliczeniach kwantowych. Nawet pojedynczy niegrzeczny elektron może wpłynąć na wzór interferencji, który jest tak istotny dla prawidłowego wykonania algorytmu, poprzez splątanie z kubitami biorącymi udział w tym wykonaniu, w wyniku czego pożądana superpozycja może się nie powieść. W związku z tym komputer musi być bezwzględnie odizolowany od otoczenia. Ale musi również odczytywać dane wejściowe i generować dane wyjściowe, a jego proces obliczeniowy może być kontrolowany przez niektóre elementy zewnętrzne. W jakiś sposób te sprzeczne wymagania trzeba pogodzić. Jakich rozmiarów naprawdę potrzebujemy? Niektóre protokoły kodowania kwantowego na małą skalę wymagają tylko około 10 kubitów, a nawet algorytm faktoryzacji kwantowej potrzebuje tylko kilku tysięcy kubitów, aby można go było zastosować w rzeczywistych sytuacjach. Ale ponieważ fizyka eksperymentalna radzi sobie obecnie tylko z siedmioma kubitami, a nawet to jest niezwykle trudne, wiele osób jest pesymistami. Prawdziwy przełom nie jest spodziewany w najbliższym czasie. Z drugiej strony, podekscytowanie związane z tym tematem już powoduje lawinę pomysłów i propozycji, którym towarzyszą złożone eksperymenty laboratoryjne, tak że z biegiem czasu z pewnością zobaczymy interesujące postępy. Podsumowując, wielomianowy algorytm faktoringu kwantowego Shora stanowi duży postęp w badaniach obliczeniowych pod każdą miarą. Jednak w tej chwili musi zostać zdegradowany do statusu półek i prawdopodobnie pozostanie takim przez jakiś czas. Nieusuwalność nie została jeszcze pokonana.

Obliczenia molekularne

Aby zakończyć naszą dyskusję na temat modeli obliczeniowych mających na celu złagodzenie niektórych złych wiadomości, wspominamy jeszcze jedną: obliczenia molekularne, czasami nazywane obliczeniami DNA. Główne podejście tutaj polega na tym, aby obliczenia odbywały się zasadniczo same, w starannie skomponowanej „zupie" cząsteczek, które bawią się ze sobą, dzieląc, łącząc i scalając. W ten sposób otrzymujesz miliardy lub biliony molekuł do rozwiązania trudnego problemu za pomocą brutalnej siły, sprytnie konfigurując rzeczy, aby zwycięskie przypadki można później wyizolować i zidentyfikować. W eksperymencie z 1994 roku stworzono molekuły w celu rozwiązania małego przykładu problemu ścieżki Hamiltona, który, jak wyjaśniono w rozdziale 7, jest w rzeczywistości rodzajem jednostki długości wersji problemu komiwojażera. Później inne problemy — w zasadzie wszystkie problemy w NP - okazały się podatne na podobne techniki. To, że natura może być dostrojona do rozwiązywania rzeczywistych problemów algorytmicznych, zasadniczo sama i w skali molekularnej, jest raczej zdumiewające. Podczas gdy oryginalny eksperyment dla instancji z siedmiu miast trwał kilka dni w laboratorium, problem został później rozwiązany przez innych w mniej brutalny sposób i dla znacznie większych instancji (50–60 miast). Poświęcenie laboratoriów biologii molekularnej do tego rodzaju prac może skutkować znacznym przyspieszeniem procesu i rzeczywiście trwa wiele prac, aby spróbować zwiększyć skalę technik. Z purystycznego punktu widzenia rzeczy przypominają konwencjonalne algorytmy równoległe: chociaż w zasadzie złożoność czasowa takich algorytmów molekularnych jest wielomianowa ze względu na wysoki stopień paralelizmu, który zachodzi w zupie molekularnej, liczba cząsteczek biorących udział w procesie rośnie wykładniczo. Ale z drugiej strony, jedną z głównych zalet używania DNA jest jego niesamowita gęstość informacji. Niektóre wyniki pokazują, że obliczenia DNA mogą zużywać miliard razy mniej energii niż komputer

elektroniczny wykonujący te same czynności i mogą przechowywać dane w bilionach razy mniejszej przestrzeni. W każdym razie informatyka molekularna jest zdecydowanie kolejnym ekscytującym obszarem badań, przyciągającym wyobraźnię i energię wielu utalentowanych informatyków i biologów. W przyszłości czeka nas wiele ekscytujących prac w tej dziedzinie, a niektóre specyficzne, trudne problemy mogą stać się wykonalne przy rozsądnych nakładach. Musimy jednak pamiętać, że zdecydowanie nie może wyeliminować nieobliczalności, ani nie oczekuje się, że zlikwiduje fatalne skutki nierozwiązywalności.