

Wektory zamiast tortu – sprawiedliwy podział w wielu wymiarach

Igor JANKOWSKI*

* Politechnika Gdańska
Kontakt: igorjankowski@gmail.com

O problemie krojenia tortu i jego rozwiązaniu za pomocą lematu Spernera można przeczytać w Δ_3 .

CPU to procesor, GPU to karta graficzna, nazywana też dawniej akcelerorem graficznym.

Można podzielić coś idealnie po równo i zmarnować przy tym jedną trzecią. Ten pozorny paradoks zna każdy administrator klastra obliczeniowego: użytkownicy dostają identyczne przydziały procesorów i kart graficznych, a mimo to część maszyn stoi bezczynnie ze względu na zróżnicowane zapotrzebowanie na te zasoby. Jest w tym dużo podobieństwa do tzw. *problemu krojenia tortu*, który Hugo Steinhaus postawił w latach czterdziestych XX wieku. Nasz tort jest jednak specyficzny, nie każdy kawałek da się z niego wykroić. Przyjrzyjmy się tej trudności na konkretnym przykładzie.

Rozważmy klastę obliczeniową o pojemności $C = (180 \text{ CPU}, 180 \text{ GPU})$. Użytkownik A zajmuje się analityką danych i potrzebuje do każdego zadania wektora $D_A = (2 \text{ CPU}, 1 \text{ GPU})$. Użytkownik B trenuje sieci neuronowe, wymagając $D_B = (1 \text{ CPU}, 3 \text{ GPU})$. Zakładamy przy tym, że ich popyt jest nienasycony – obaj mają w kolejce setki procesów i chcą zrównoleglić obliczenia na tyle, na ile pozwoli im sprzęt.

Naturalnym podejściem wydaje się podział każdego z zasobów po równo (*Max-Min Fairness*). Limitujemy użytkowników do 50% serwerowni, udostępniając im maksymalnie 90 CPU oraz 90 GPU. Przy naiwnym podziale „po równo” każdy otrzymuje przydział ograniczony przez ten wymiar, który wyczerpie się u niego jako pierwszy. Biorąc pod uwagę takie obostrzenia, użytkownik A zdoła uruchomić jedynie 45 zadań, ponieważ wyczerpie przydzieloną pulę procesorów ($45 \times 2 = 90 \text{ CPU}$) – mimo ogromnego zapasu przyznanych mu kart graficznych. Z kolei użytkownik B zostanie zablokowany przez limit akceleratorów, uruchamiając zaledwie 30 zadań ($30 \times 3 = 90 \text{ GPU}$). Łącznie zużyją 120 CPU i 135 GPU. Reszta cennych zasobów – 60 CPU i 45 GPU – będzie leżała odłogiem.

Kolejnym rozważanym podejściem jest naiwny mechanizm rynkowy ze stałymi cenami. Nadajemy zasobom równe wartości. Ponieważ serwerownia składa się z 360 jednostek, przy cenie 10 kredytów za sztukę jej wartość wynosi 3600 kredytów. Dajemy każdemu z użytkowników budżet w wysokości 1800 kredytów. Użytkownik A sfinansowałby z tego 60 swoich zadań, a użytkownik B kupiłby ich 45. Łącznie generuje to zapotrzebowanie na 165 CPU oraz aż 195 GPU. Drugi zasób przekracza fizyczne możliwości klastra. Gdybyśmy chcieli rozwiązać ten problem poprawnie i wprowadzili dynamiczny mechanizm rynkowy (np. równowagę Walrasa), w którym ceny rosną wraz z popytem, zbalansowalibyśmy system. Taki rynek staje się jednak natychmiast podatny na spekulacje – uczestnikom opłaca się fałszować swoje zgłoszenia: zawyżać popyt na pożądany zasób, by wyczerpać budżety rywali, lub ukrywać preferencje, by uniknąć wzrostu ceny.

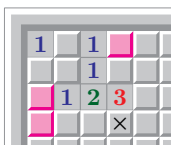
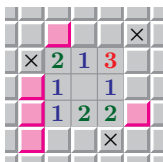
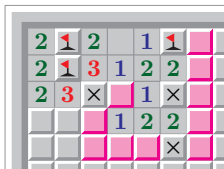
W poszukiwaniu sprawiedliwości: algorytm DRF

Rozwiązania tego problemu dostarcza algorytm *Dominant Resource Fairness* (DRF). Porzuca on koncepcję „wspólnej waluty” na rzecz patrzenia przez pryzmat fizycznych ograniczeń maszyny. Załóżmy, że oferuje ona m zasobów, o które konkuruje N użytkowników (w przykładzie wyżej $N = m = 2$). Sprowadzamy żądania do abstrakcyjnej przestrzeni ułamków pojemności całego klastra. Dla jednego zadania i -tego użytkownika ułamek zużycia j -tego zasobu wynosi d_{ij}/C_j . Największy z tych ułamków nazywamy potrzebą dominującą i -tego użytkownika:

$$\mu_i = \max_{j \in \{1, \dots, m\}} \left(\frac{d_{ij}}{C_j} \right).$$

W naszym wariancie ułamki dla użytkownika A wynoszą $2/180 = 1/90$ dla CPU oraz $1/180$ dla GPU. Zasobem dominującym są procesory, stąd $\mu_A = 1/90$. Dla użytkownika B ułamki to $1/180$ dla CPU oraz $3/180 = 1/60$ dla GPU. Dominującą kartę graficzną, więc $\mu_B = 1/60$.

Rozwiązania zadań ze strony 15.
Zakolorowane pola można odsłonić.
Krzyżyki wskazują miejsca, w których na pewno jest mina.



Zasada algorytmu DRF opiera się na tym, że wszyscy uczestnicy mają otrzymać taki przydział, w którym ich *udział dominujący* osiągnie identyczną, globalną wartość s . Jeżeli użytkownik uruchomi x_i zadań, jego całkowity udział dominujący wyniesie $s_i = x_i \mu_i$. Zrównanie ich ($s_1 = s_2 = \dots = s$) oznacza w ostatecznym rozrachunku, że każdy kontroluje dokładnie taki sam ułamek swojego zasobu dominującego. Liczba przydzielonych zadań x_i staje się więc funkcją pożądanego poziomu s , zgodnie ze wzorem $x_i = s/\mu_i$. Intuicyjnie rzecz ujmując, algorytm traktuje udziały dominujące jako miarę tego, jak duży kawałek serwerowni faktycznie zajmujemy. W idealnym scenariuszu miara ta powinna rosnąć dla każdego w identycznym tempie. Należy tu jednak wprost zaznaczyć istotne założenie: popyt użytkowników jest nieograniczony (w prawdziwych systemach produkcyjnych system po prostu odcina dalszy przydział w momencie nasycenia).

Skąd zatem wziąć ostateczną wartość s ? Wyobraźmy sobie, że powoli proporcjonalnie zwiększamy przydziały wszystkich uczestników. Robimy to tak długo, aż w klastrze fizycznie wyczerpie się któryś z zasobów. Skoro $x_i = s/\mu_i$, to całkowite zużycie konkretnego zasobu j w całym systemie wynosi $\sum_{i=1}^N (s/\mu_i) d_{ij}$. Ze względów fizycznych suma ta nie może przekroczyć pojemności klastra C_j . Nierówność $s \sum (d_{ij}/\mu_i) \leq C_j$ po odpowiednim przekształceniu dostarcza nam poszukiwany wzór na s . Ponieważ bariera musi być zachowana dla wszystkich wymiarów, wybieramy tę najbardziej wymagającą (najmniejszą dopuszczalną wartość):

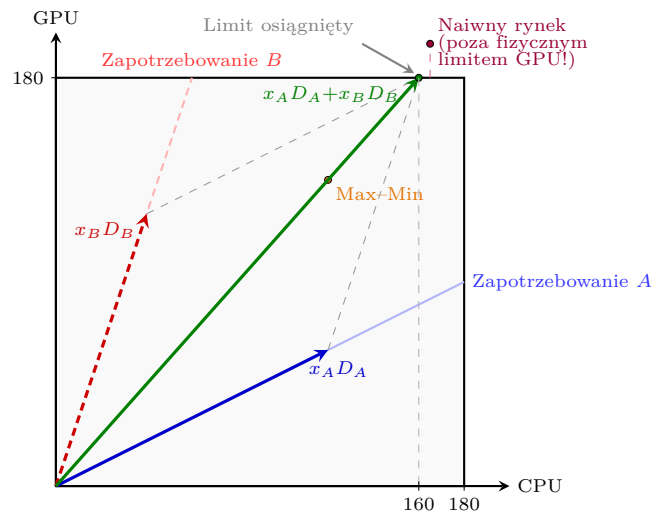
$$s = \min_{j \in \{1, \dots, m\}} \left(\frac{C_j}{\sum_{i=1}^N \frac{d_{ij}}{\mu_i}} \right).$$

To wyrażenie matematyczne to po prostu poszukiwanie tzw. wąskiego gardła – sprawdzamy, przy jakiej wartości s osiągniemy fizyczną granicę któregoś z zasobów. Prześledźmy obliczenia na przykładzie. Dla procesorów mianownik wynosi

$$\frac{d_{A,CPU}}{\mu_A} + \frac{d_{B,CPU}}{\mu_B} = \frac{2}{1/90} + \frac{1}{1/60} = 240.$$

Ograniczenie narzucone przez CPU to bariera $s \leq 180/240 = 0,75$. Mianownik dla układów GPU wynosi z kolei $1/(1/90) + 3/(1/60) = 270$. Ograniczenie dla kart graficznych okazuje się ciśniejsze: $s \leq 180/270 = 2/3 \approx 0,66$. Optymalizacja zatrzymuje się na limicie kart graficznych, więc $s = 2/3$. Użytkownik A może uruchomić $x_A = (2/3)/(1/90) = 60$ zadań (otrzymując przydział $x_A D_A = (120 \text{ CPU}, 60 \text{ GPU})$). Użytkownik B może uruchomić $x_B = (2/3)/(1/60) = 40$ zadań (jego przydział to $x_B D_B = (40 \text{ CPU}, 120 \text{ GPU})$). Łączne zużycie klastra wynosi (160 CPU, 180 GPU). Zasób limitujący został wykorzystany w 100%, a marnotrawstwo w wąskim gardle zniknęło.

Geometria tego problemu sprowadza się do znalezienia odpowiedniego punktu w przestrzeni zasobów wykres powyżej.



Geometria podziałów. Suma wektorów dla DRF opiera się o krawędź klastra, co oznacza 100% wykorzystania limitującego zasobu (GPU). Metoda Max-Min zostawia znaczny margines obu typów sprzętu, natomiast mechanizm stałych cen wychodzi poza prostokąt, wskazując wartości fizycznie niemożliwe do zrealizowania.

Wektory uzyskanych przydziałów ($x_A D_A$ oraz $x_B D_B$) sumują się zgodnie z regułą równoległoboku. Ich suma znajduje się na górnej krawędzi prostokąta, co obrazuje wyczerpanie GPU, pozostawiając margines bezpieczeństwa na osi procesorów.

Matematyka bez zawiści

Jedną z podstawowych cech, których wymaga się często od algorytmów podziału, jest *brak zawiści*. Oznacza to, że w wyniku zastosowania algorytmu żadnemu z uczestników nie opłacałoby się wymienić otrzymanym kawałkiem z kimś innym. W naszym przypadku opłacalność mierzona jest liczbą wykonanych zadań. Ponieważ zasoby przyznane k -temu użytkownikowi wynoszą $(x_k d_{k1}, \dots, x_k d_{km})$, to użyteczność dla i -tego użytkownika z tych zasobów jest równa:

$$U_i(k) = \min_{j \in \{1, \dots, m\}} \left(\frac{x_k d_{kj}}{d_{ij}} \right).$$

Twierdzenie. Algorytm DRF realizuje podział bez zawiści (dla każdego użytkownika i oraz k zachodzi $U_i(i) \geq U_i(k)$).

Dowód. Zastanówmy się najpierw nad wartością użyteczności z własnego przydziału. Ponieważ algorytm przyznał nam zasoby idealnie dopasowane do naszych potrzeb, możemy z nich uruchomić dokładnie x_i zadań. Formalnie $U_i(i) = x_i$, co zgodnie z wyliczonym wcześniej wzorem algorytmu wynosi s/μ_i .

Założmy teraz, że użytkownik i spogląda na cudzy przydział (zasoby sąsiada dla x_k zadań). Niech r_i oznacza zasób dominujący dla użytkownika i . Zapotrzebowanie naszego pojedynczego zadania na ten zasób wynosi $d_{i,r_i} = \mu_i C_{r_i}$.

Ile zasobu dominującego r_i posiada w swoim przydziale sąsiad k ? Zgromadził go w ilości $x_k d_{k,r_i}$. Ponieważ μ_k z definicji określa największe zapotrzebowanie ze wszystkich zasobów sąsiada, to dla interesującego nas zasobu r_i musi zachodzić nierówność $d_{k,r_i}/C_{r_i} \leq \mu_k$, czyli $d_{k,r_i} \leq \mu_k C_{r_i}$.

Nawet jeśli inne zasoby u sąsiada występują w obfitości, to ewentualne braki w naszym zasobie dominującym r_i będą stanowiły dla nas główną barierę, powyżej której nie uruchomimy więcej procesów. Ponieważ minimum po wszystkich zasobach nie przekracza wartości dla pojedynczego, konkretnego zasobu r_i , użyteczność całego cudzego przydziału jest ograniczona z góry:

$$U_i(k) \leq \frac{x_k d_{k,r_i}}{d_{i,r_i}} \leq \frac{x_k (\mu_k C_{r_i})}{\mu_i C_{r_i}} = \frac{x_k \mu_k}{\mu_i}.$$

Skoro wiemy dodatkowo, że DRF wszystkim zrównał udziały dominujące ($x_k \mu_k = s$), możemy podstawić ten fakt do licznika naszej nierówności:

$$U_i(k) \leq \frac{s}{\mu_i} = x_i = U_i(i).$$

Zatem mechanizm DRF sprawia, że nikomu nie opłaca się zazdrościć sąsiadowi jego przydziału. Mechanizm DRF zmniejsza też opłacalność oszustwa, bo sztuczne zawyżanie zapotrzebowania na zasób dominujący obróci się przeciwko użytkownikowi (powiększona wartość trafia do mianownika w naszym wzorze).

Cena kompromisu

Dlaczego więc algorytm pozwala, by w naszej symulacji pozostało niewykorzystane 20 CPU? Wyobraźmy sobie, że zabiegając o *silną* optymalność Pareto (wymóg bezwzględnego rozdysponowania całości sprzętu), postanawiamy oddać te wolne procesory komuś, kogo zadania nie wymagają użycia GPU. Kiedy to zrobimy, udział dominujący tej osoby wzrośnie, niszcząc fundament metody DRF: równe udziały dominujące. Uczestnicy systemu szybko zorientowaliby się, że opłaca się modyfikować zgłoszenia, by celowo przejmować wolne okrychy. Ratując pełne wykorzystanie maszyn, tracimy odporność na manipulacje. Ten konflikt to nie słabość konkretnej implementacji, lecz matematyczna właściwość problemu.

W pracy *Dominant Resource Fairness: Fair Allocation of Multiple Resource Types* autorzy algorytmu DRF argumentują, że w klasycznym modelu alokacji

(obejmującym podzielne zasoby i strategicznych graczy), żaden algorytm nie jest w stanie jednocześnie pogodzić ze sobą czterech własności:

- (i) braku zawiści;
- (ii) odporności na manipulacje (prawdomówność opłaca się w każdej sytuacji);
- (iii) gwarancji opłacalności (*sharing incentive*) – pewność, że nikt nie otrzyma mniej, niż gdyby samodzielnie kupił sprzęt wielkości $1/N$ klastra;
- (iv) silnej optymalności Pareto – bezwzględny wymóg niepozostawiania wolnych okrychów.

To napięcie wynika z prostego faktu: jeśli zmusimy system do rozdania wszystkich nieużywanych okrychów (własność (iv)) tym, którzy akurat ich potrzebują, zaburzamy proporcje, na których zbudowano sprawiedliwość (co łamie własność (i)) i tworzymy pole do strategicznych nadużyć (co całkowicie niweczy własność (ii)).

Oznacza to, że w tym klasycznym modelu idealny algorytm podziału wielowymiarowego tortu nie istnieje. Metoda DRF świadomie idzie na kompromis – doskonale spełnia pierwsze trzy aksjomaty, akceptując cenę w postaci niewykorzystanych okrychów. Współczesne badania sięgają jednak znacznie dalej. W prawdziwych systemach procesy dołączają do klastra i opuszczają go w ułamkach sekund. Jak zagwarantować sprawiedliwość w środowisku, gdzie algorytm – taki jak w systemie Apache Mesos, dla którego oryginalnie stworzono DRF – na bieżąco kontroluje przydział bez wiedzy o przyszłości? To wciąż otwarty problem – czy można pokroić tort sprawiedliwie, gdy nie wiemy, kto jeszcze zapuka do drzwi i z jak wielkim apetytem. Podział wielowymiarowego tortu okazuje się zatem zadaniem znacznie bardziej złożonym niż krojenie jego klasycznego, płaskiego odpowiednika Steinhausa. Choć idealna sprawiedliwość w chmurze pozostaje matematycznie niemożliwa, mądre kompromisy – takie jak DRF – pozwalają nam dzielić zasoby bez zawiści, nawet jeśli czasem na stole pozostaną wolne okrychy.

Studencka Konferencja Naukowa

organizowana w ramach obchodów dni prof. Antoniego Hoborskiego

elements X

23-25 października 2026

STATYSTYKA | RACHUNEK PRAWDOPODOBIEŃSTWA | MACHINE LEARNING
ANALIZA DANYCH

exploruj z nami świat data science!
www.elements.agh.edu.pl