Analityczne modelowanie systemów równoległych

Modelowanie wydajności

- Program szeregowy jest modelowany przez czas wykonania. Na ogół jest to czas asymptotyczny w notacji O.
- Czas wykonania programu szeregowego zależy od jego rozmiaru danych wejściowych.
  - Algorytm mergesort użyty dla tablicy o rozmiarze n ma złożoność obliczeniową O(nlog(n)).
- Algorytm szeregowy ma tę samą złożoność obliczeniową na każdej maszynie szeregowej.
- W przypadku programów równoległych sytuacja się komplikuje. Czas pracy jest funkcją: rozmiaru danych, liczby procesorów i architektury maszyny równoległej (np prędkości wymiany danych pomiędzy procesami).
- Trudno jest analizować algorytm w oderwaniu od maszyny bez pewnej utraty dokładności analizy.
  - Aczkolwiek stworzone pewne wyidealizowane modele maszyny takie jak model logP czy maszyna PRAM.
- Zaproponowano (A. Grama i wsp.) aby kombinację algorytmu równoległego i maszyny równoległej nazywać ***systemem równoległym*** i analizować razem.

Przypomnienie: Notacje O, Θ, Ω

- Funkcja $f(x) = O(g(x))$ (czytamy ***f jest rzędu co najwyżej g***), jeżeli istnieje stała c, taka że $f(x) \leq cg(x)$ dla $x > x_{0.}$

- Funkcja $f(x) = \Theta(g(x))$ (czytamy ***f jest rzędu dokładnie g***), jeżeli istnieją stałe c_1 oraz c_2, takie że , $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla $x > x_0$

- Funkcja $f(x) = \Omega(g(x))$ (czytamy ***f jest rzędu co najmniej g***), jeżeli istnieje stała c, taka że $cg(x) \leq f(x)$ dla $x > x_{0.}$

Przykład:

$100n^2+0.5*n^3+10n+1=O(n^3)$.

Użyteczność notacji O, Θ, Ω w obliczeniach równoległych

- Notacje O, Θ, Ω są powszechnie stosowane w modelowaniu czasu obliczeń sekwencyjnych (patrz przedmiot algorytmy i struktury danych).
- Ich użyteczność (i w ogóle użyteczność analizy asymptotycznej) w modelowaniu czasy pracy programów równoległych jest ***o wiele mniejsza***.
  - Prawie zawsze czas pracy programu zależy od dwóch zmiennych liczby procesorów p i rozmiaru danych n.
  - Nie możemy przyjąć, że obie dążą do nieskończoności.
- Analiza asymptotyczna ignoruje czynniki niższych rzędów, które w praktyce mogą być bardzo istotne.
  - Np. wzór na czas transmisji komunikatu o długości n słów ma postać: $t_c=t_s+n*t_w$
  - Analiza asymptotyczna powie nam że ten czas jest Θ(n). Jednak to wyrażenie ignoruje zupełnie t_s, który dla praktycznych zastosowań może zdominować czas transmisji komunikatu.
  - Ponadto analiza asymptotyczna ignoruje inne czynniki, pojawiające się w rzeczywistych systemach jak np. obciążenie sieci połączeń.

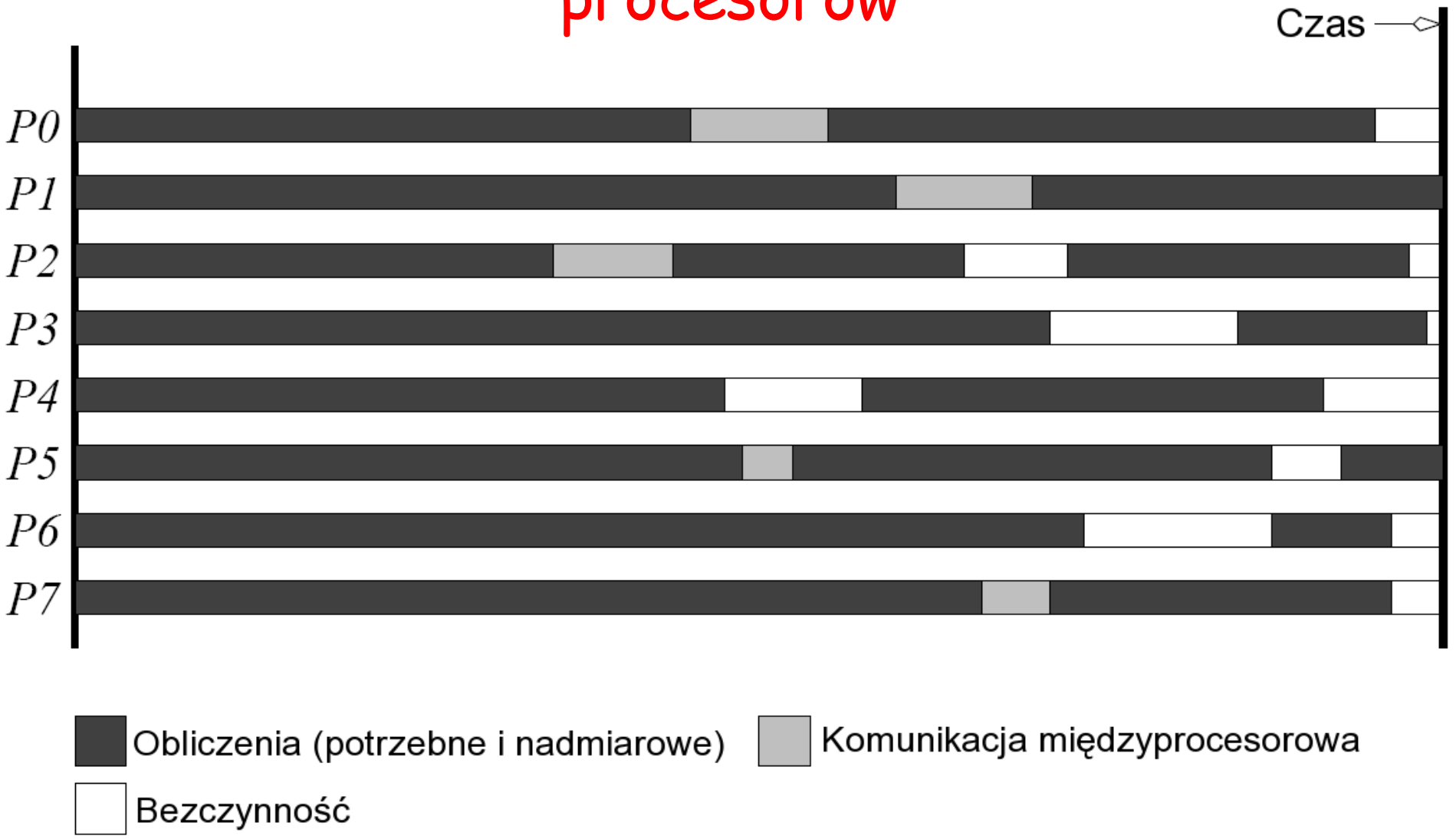
Miary wydajności dla programu równoległego

- Czas pracy (ang. wall clock time). Czas mierzony od momentu startu pierwszego procesora do momentu gdy ostatni procesor w grupie rozwiązującej problem zakończy pracę. Oznaczamy jako T_p i jest on funkcją liczby procesorów p. Czas pracy wersji szeregowej oznaczamy jako T_S.
- Przyspieszenie (ang. speedup): ile razy wersja równoległa jest szybsza.
  - Natychmiast pojawia się pytanie: ***szybsza od czego?*** Nasz algorytm równoległy uproszczony dla jednego procesora nie musi wcale być algorytmem optymalnym, bo np. najlepszy algorytm szeregowy jest trudny do zrównoleglenia i zrównoleglilismy algorytm „gorszy”
  - Aby porównanie było uczciwe należy porównywać się z ***najlepszym*** algorytmem szeregowym.
  - Przykład załóżmy że sortowanie bąbelkowe zajmuje 40s. Zrównoleglone na czterech procesorach trwa 10s. Przyspieszenie dla czterech procesorów jest równe 4. Ale czy to jest rzetelna ocena zrównoleglenia, jeżeli algorytm quicksort w wersji szeregowej zajmuje 3s.
  - Rzeczywiste przyspieszenie wynosi więc 0.3
- Przyspieszenie definiujemy jako $S(p)=T_S/T_p(p)$.

Źródła narzutów (ang. overhead) w obliczeniach równoległych.

- Czy jeżeli użyję dwóch procesorów to mój program wykona się dwa razy szybciej ?
- ***Z reguły nie***. Program równoległy zawiera wiele dodatkowych narzutów w porównaniu z programem szeregowym. Są to:
- Komunikacja pomiędzy procesorami. Procesory rozwiązujące nietrywialny problem muszą się komunikować ze sobą, co zabiera czas. Koszt ten nie jest ponoszony w wersji szeregowej. Uwaga: koszty komunikacji w modelu ze wspólną pamięcią są ukryte (ale również ponoszone), dlatego tak popularny stał się model programowania z przesyłaniem komunikatów, w którym koszty te są jawne.
- Bezczynność (ang. idling). Procesor może być bezczynny z kilku powodów, np: nierównomiernego podziału pracy pomiędzy procesorami (niezrównoważenie obciążenia - ang. load imbalance), braku zrównoleglenia części algorytmu.
- Nadmiarowe obliczenia. Obliczenia wykonywane przez wersję równoległą, a nie wykonywane przez wersję szeregową. Ich występowanie może być spowodowane trudnościami w zrównolegleniu lub potrzebą minimalizacji komunikacji.

Diagram przestrzenno - czasowy obliczeń dla 8 procesorów



- Istnieją programy (np jumpshot dostarczany z MPICH) do tworzenia takiego diagramu. Jest to dobre narzędzie do diagnozowania i analizy programów równoległych

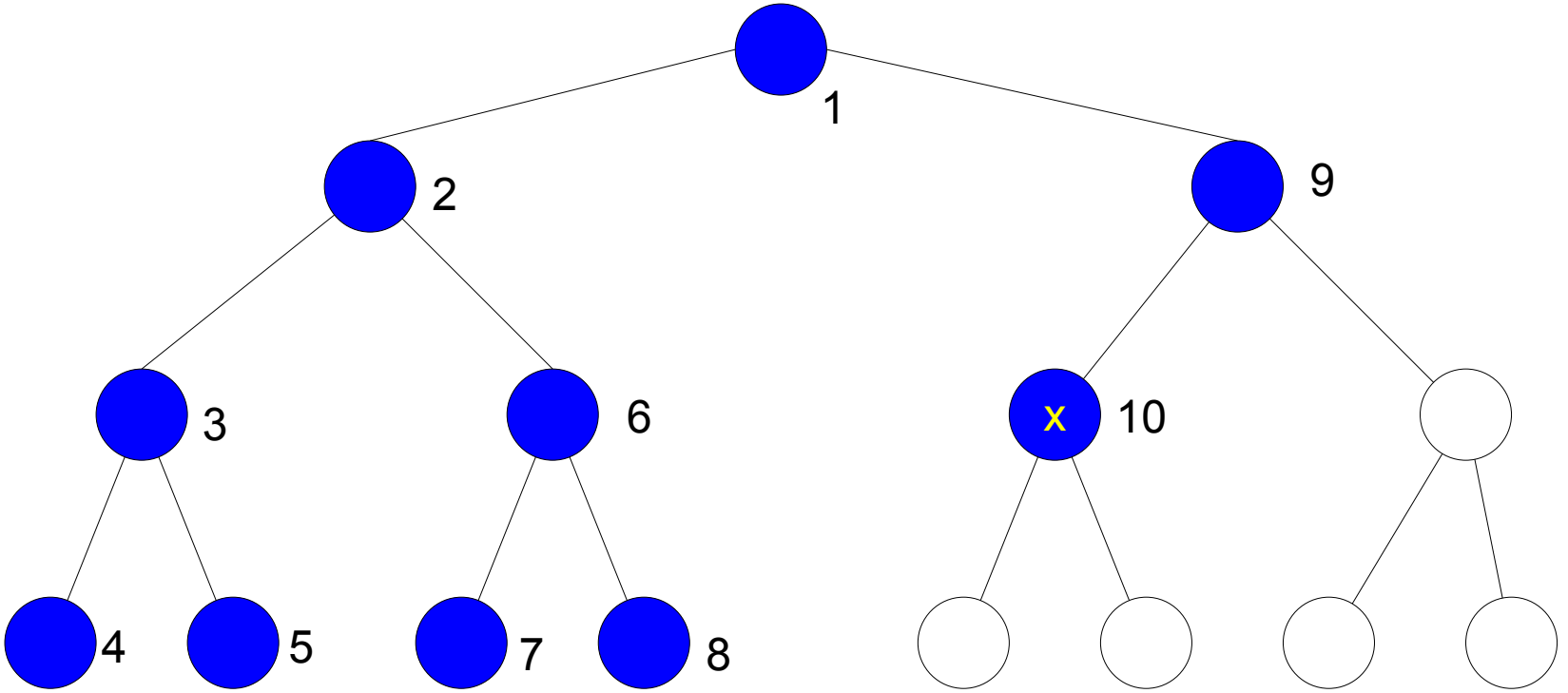
Ograniczenia przyspieszenia

- Minimalne przyspieszenie jest równe 0, w sytuacji w której program równoległy nigdy się nie kończy.
- Teoretycznie przyspieszenie jest ograniczone z góry przez liczbę procesorów p. (p krotny wzrost zasobów prowadzi do p-krotnego przyspieszenia).
- Przyspieszenia większe od p nazywane jest ***przyspieszeniem superliniowym***.
- Ma ono miejsce w tedy, gdy każdy procesor spędza mniej niż T_s/p czasu nad rozwiązywaniem problemu.
  - W takiej sytuacji możnaby zasymulować p procesorów przy pomocy jednego procesora i podziału czasu, co dałoby algorytm szeregowy o krótszym czasie wykonania niż T_s.
  - Powyższe przeczy założeniu, że porównujemy się z najszybszym algorytmem szeregowym.

Przyczyna przyspieszenia superliniowego - lepsze wykorzystanie pamięci cache

- Większa łączna pojemność pamięci cache procesorów w systemie wieloprocesorowym prowadzi to większego współczynnika trafień i w konsekwencji co superliniowego przyspieszenia.
- Przykład: procesor z 64KB pamięci cache ma współczynnik trafień 80% (dla danego algorytmu i rozmiaru danych).
- W systemie dwuprocesorowym łączna pojemność pamięci cache dwóch procesorów wynosi 128KB. Ponieważ rozmiar danych pozostaje taki sam współczynnik trafień może wzrosnąć. Przyjmijmy że wzrośnie do 90%. Z pozostałych 10%, 8% to dostępy do pamięci lokalnej DRAM 2% do pamięci zdalnej (zakładamy architekturę NUMA).
- Przyjmując czas dostępu do pamięci cache 2ns, lokalnej 100ns, zdalnej 400 ns dostajemy (obliczenia - A. Grama) przyspieszenie 2.43 dla dwóch procesorów.
- Powyższa analiza zakłada brak jakichkolwiek narzutów, co nie zdarza się w praktyce. W rezultacie przyspieszenie superliniowe, z powodu lepszego wykorzystania pamięci cache zdarza się bardzo rzadko (ale czasami ma miejsce).

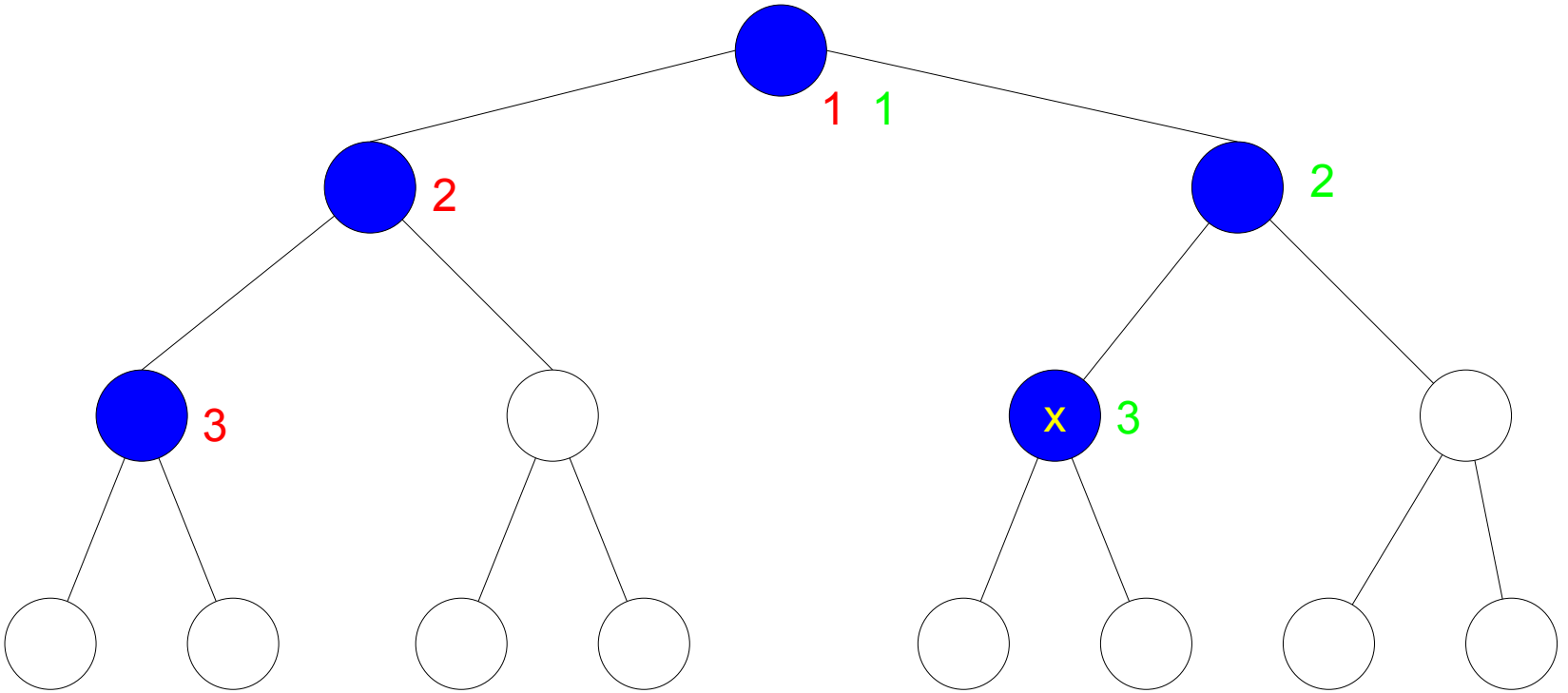
Przyczyna przyspieszenia superliniowego - program szeregowy wykonuje więcej pracy



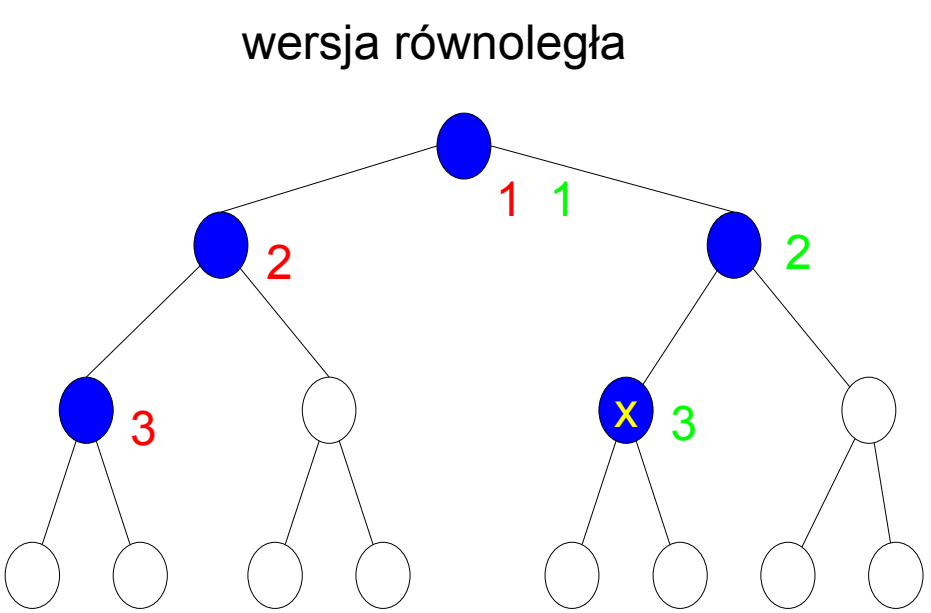
- Przeszukiwanie wgłąb drzewa od korzenia, w poszukiwaniu wierzchołka ze znakiem „x”.
- Wierzchołki odwiedzane są zaznaczone na niebiesko, liczby pokazują kolejność odwiedzania
- Razem 10 odwiedzanych wierzchołków.

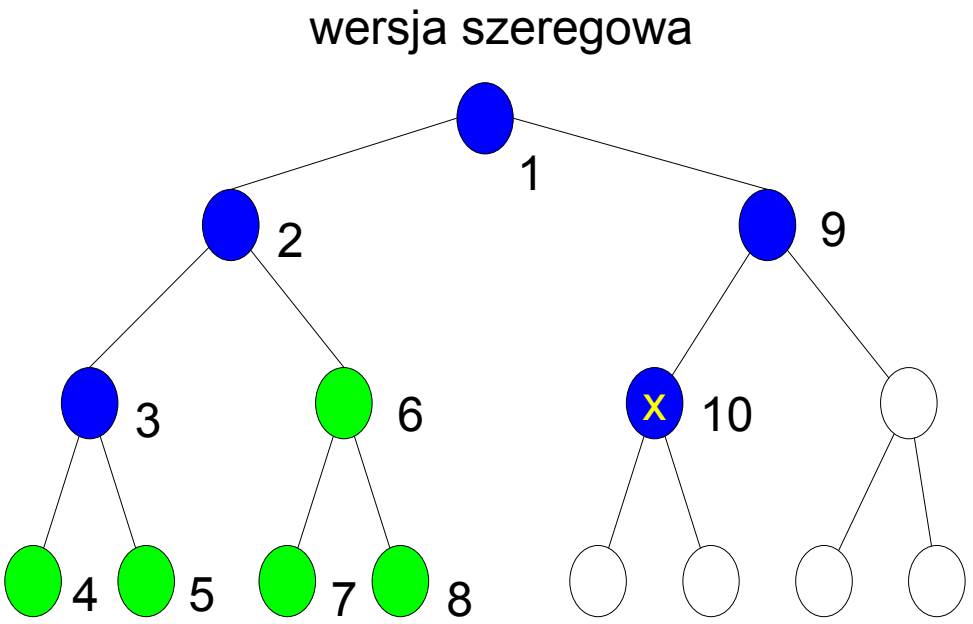
Przeszukiwanie wgłąb - wersja równoległa

- Wersja dla dwóch procesorów. Obydwa zaczynają od korzenia.
- **Procesor pierwszy** otrzymuje lewe poddrzewo a **procesor drugi** prawe. Obydwa poddrzewa przeszukiwane są równolegle
- Zakładamy, że pierwszy procesor, który znajdzie wierzchołek kończy pracę całego programu.
- W trzecim kroku procesor drugi znajduje „x”, przyspieszenie 10/3=3.33 !



Przyspieszenie superliniowe - przyczyna





- Wyraźnie widać, że program równoległy wykonuje mniej pracy.
- Dzięki "lepszemu" punktu startowemu jednego z procesorów natrafiamy szybciej na rozwiązanie i nie ma potrzeby odwiedzania wierzchołków drzewa zaznaczonych na zielono.
- W tej sytuacji algorytm składający się z dwóch procesów na jednym procesorze również szybciej odnalazłby rozwiązanie od algorytmu szeregowego.
  - Kłóci się to z założeniem porównywania z najlepszym algorytmem szeregowym, ale dla tego problemów trudno sformułować algorytm "najlepszy"

Wydajność (ang. Efficiency)

- Wydajność jest miarą użytecznego wykorzystania procesora. Definiuje się ją jako stosunek przyspieszenia do idealnego przyspieszenia liniowego równego p:

$$E(p) = \frac{S(p)}{p} = \frac{T_s}{p * T_p(p)}$$

- Wydajność w praktyce zawiera się w przedziale (0,1).
- Wydajność równa 1 oznacza przyspieszenie liniowe.
- W przypadku przyspieszenia superliniowego wydajność jest większa niż 1.

Koszt

- Koszt jest iloczynem czasu pracy i liczby procesor

$$C(p) = p * T_p(p)$$

- Koszt odzwierciedla sumę czasu spędzonego przez wszystkie procesory pracujące nad rozwiązaniem problemu.
- System równoległy jest ***optymalny ze względu na koszt***, jeżeli jego koszt jest tego samego rzędu (jako funkcja rozmiaru danych), co czas pracy najlepszego algorytmu szeregowego.
- Znaczenie praktyczne tego pojęcia: Można wykazać że algorytm optymalny ze względu na koszt ma ***stałą wydajność***.
  - Jeżeli nie jest optymalny ze względu na koszt, wydajność spada przy liczbie procesorów dążącej do nieskończoności.

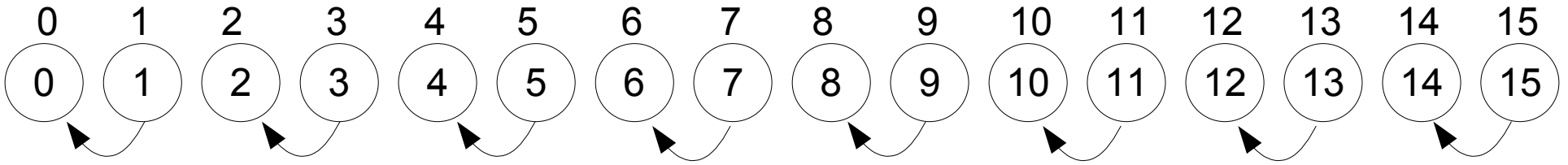
Całkowity narzut równoległy (ang. total parallel overhead)

- T_p - czas wersji równoległej, T_s - czas wersji szeregowej, p liczba procesorów.
- Całkowity czas spędzony przez wszystkie procesory pracujące nad rozwiązaniem problemu jest równy pT_p.
- T_s jednostek tego czasu jest spędzona na użytecznej pracy pozostałość jest całkowitym narzutem równoległym (T_o).

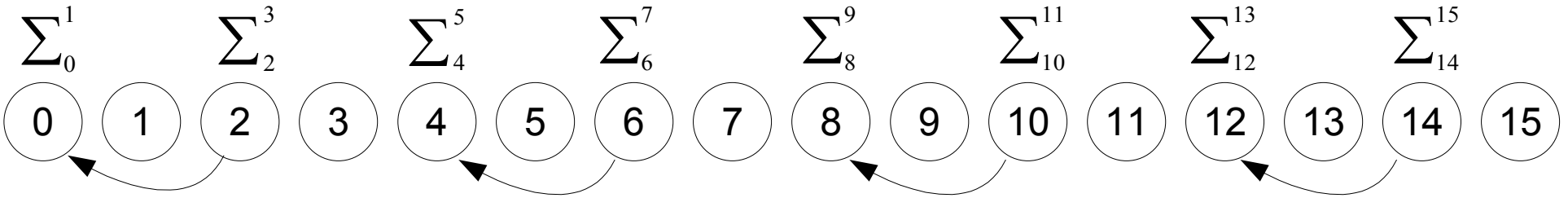
$$T_o(p) = p * T_p(p) - T_s$$

Przykład (A. Grama i wsp.): dodawanie liczb

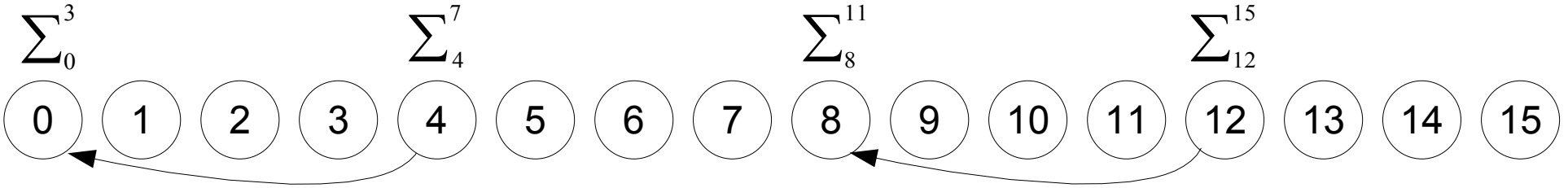
- 16 procesorów, każdy przechowuje jedną z liczb. Krok 1. (Strzałka oznacza wysłanie liczby a jej odbiorca wykonuje sumowanie swojej liczby z nadesłaną)



- Krok 2. Sumowanie ośmiu sum cząstkowych, otrzymujemy cztery sumy cząstkowe

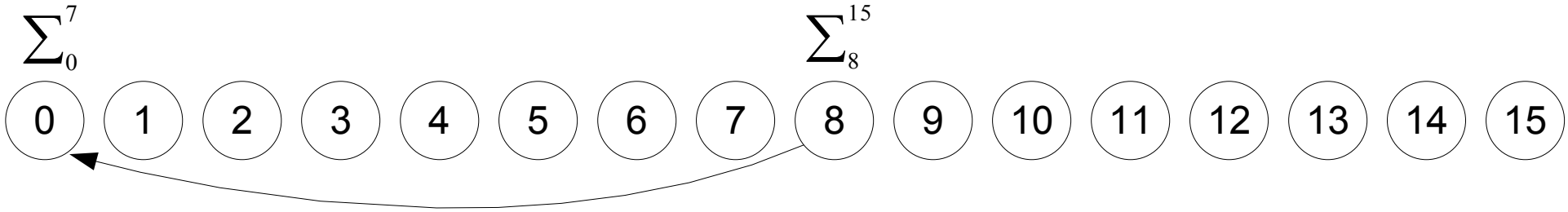


- Krok 3. Sumowanie czterech sum cząstkowych, otrzymujemy dwie sumy cząstkowe

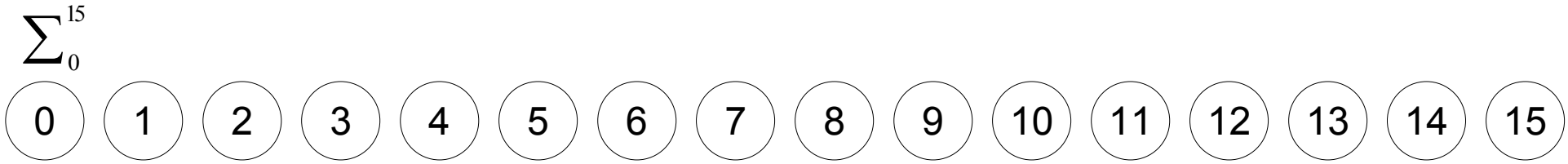


Dodawanie liczb - c.d.

- Krok 4. Sumowanie dwóch sum cząstkowych otrzymujemy wynik w procesorze 0



- Po zakończeniu algorytmu procesor 0 zawiera sumę liczb.



- Algorytm zakończyliśmy po czterech krokach. Nietrudno zauważyć że w każdym kroku zakres sum cząstkowych podwaja się. Stąd liczba kroków algorytmu sumowania p liczb na p procesorach wynosi Θ(log(p)).
- Czas jednego kroku t obejmuje wysłanie jednej liczby (t_S+1*t_W) + wykonanie dodawania (t_D). Zatem $t=t_S+t_W+t_D$. Czas ten jest więc stały nie zależy od liczby procesorów (i liczb) p.
- Z powyższych dwóch wniosków wynika czas pracy algorytmu szeregowego Θ(log(p)).

Dodawanie liczb - skalowalność

- Czas algorytmu szeregowego sumowania p liczb: Θ(p).
- Stąd otrzymujemy przyspieszenie:

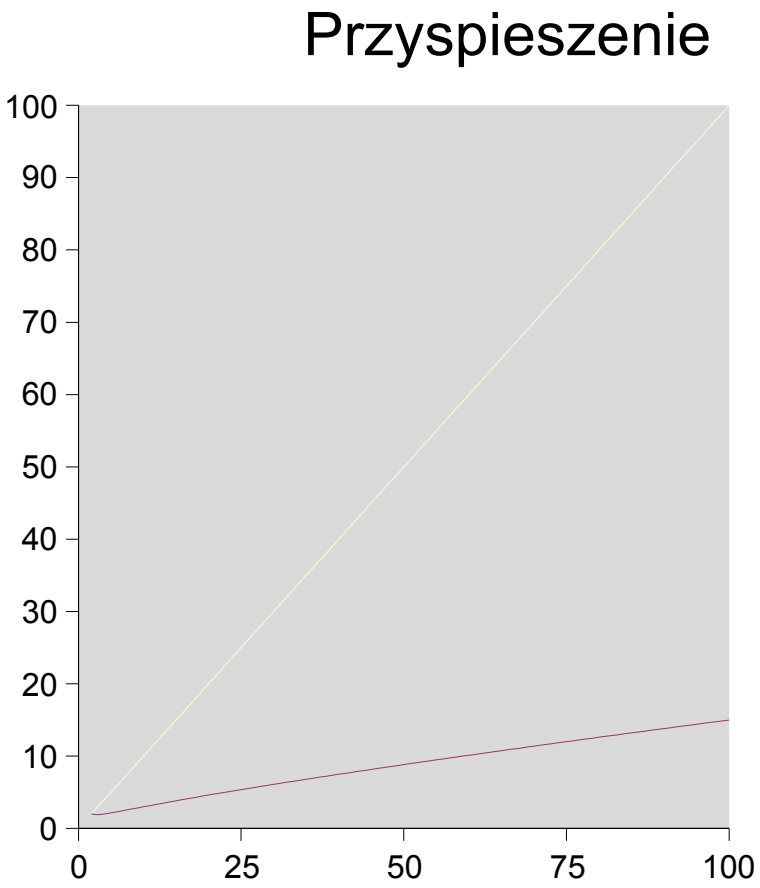
$$S(p) = \Theta\left(\frac{p}{\log(p)}\right)$$

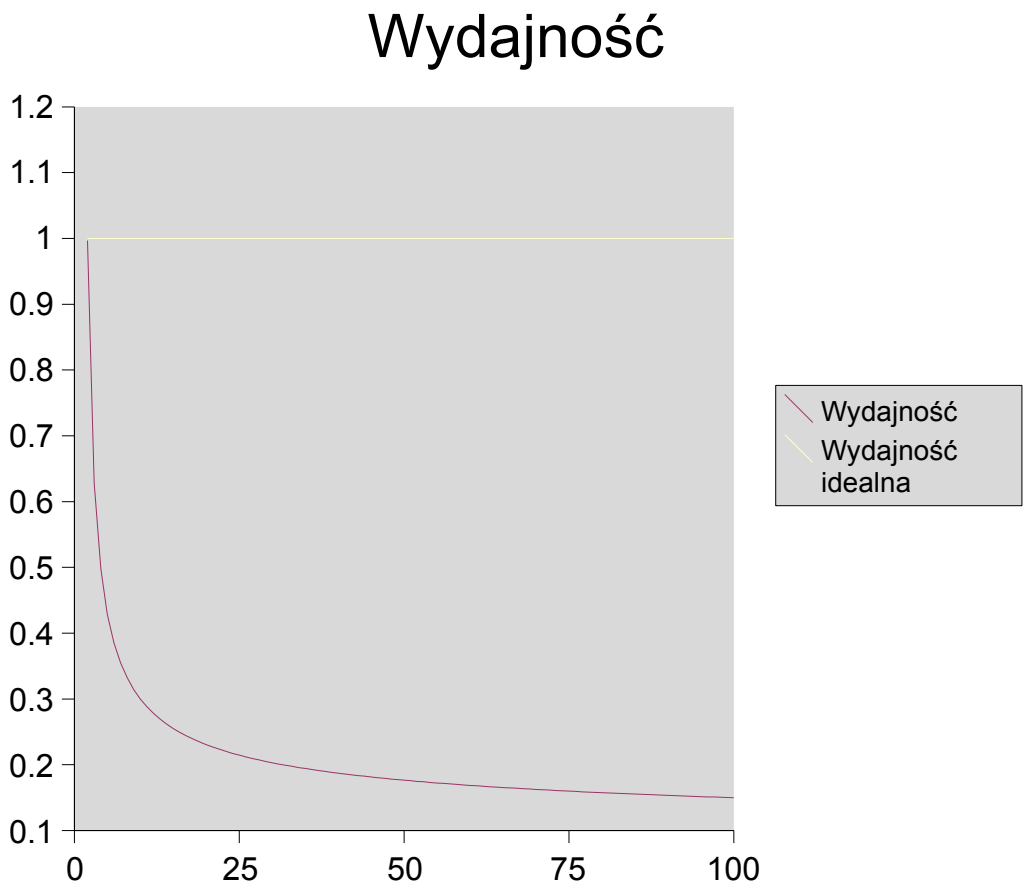
- Oraz wydajność:

$$E(p) = \frac{\Theta\left(\frac{p}{\log(p)}\right)}{p} = \Theta\left(\frac{1}{\log(p)}\right)$$

- Analiza jest asymptotyczna: pomijamy wpływ stałych t_S t_W oraz t_D.
- Ponadto w praktycznych zastosowaniach nie ma dowolnej liczby procesorów (O tym za chwilę).

Miary skalowalności - wykresy





- Przyspieszenie rośnie w nieskończoność, Wydajność spada.
- Ale ważna uwaga: wraz ze wzrostem ***liczby procesorów wzrasta rozmiar danych***, co wynika ze specyfiki algorytmu.

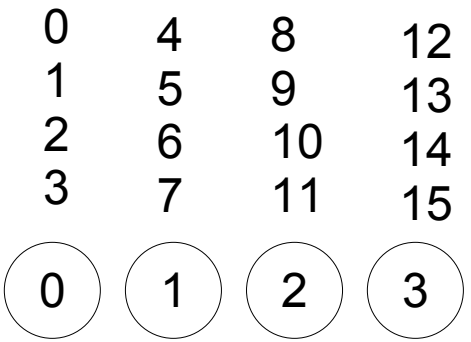
Dodawanie liczb - koszt

$$C(p) = p * T(p) = p * \Theta(\log(p)) = \Theta(p \log(p))$$

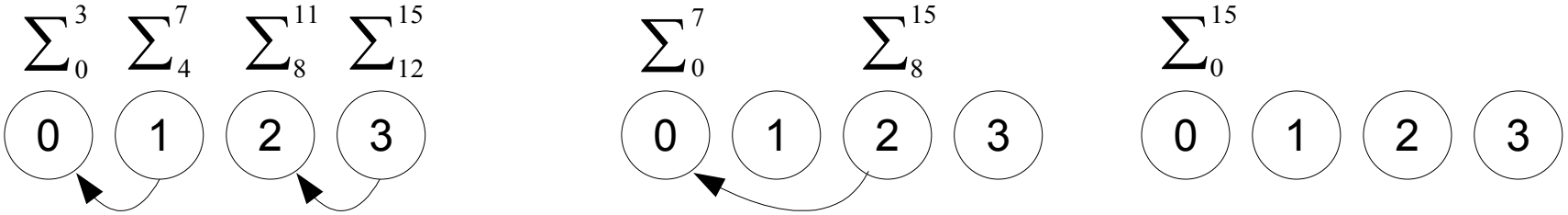
- Ponieważ czas pracy wersji szeregowej jest Θ(p), algorytm ***nie jest optymalny ze względu na koszt***.

Dodawanie n liczb na p procesorach

- Przykład dla p=4 procesorów i n=16 liczb. Każdy procesor otrzymuje 16/4=4 liczby. Najpierw każdy procesor oblicza lokalną sumę n/p liczb. Czas Θ(n/p)



- Potem postępujemy tak jak w poprzednim algorytmie wykonując dwa (ogólnie log(p)) kroki.



- Czas pracy tego algorytmu: Θ(n/p)+Θ(log(p))-Θ(n/p+log(p)) .
- Koszt tego algorytmu Θ(n+plog(p)) i w przypadku gdy n=Ω(plog(p)) (czyli mamy dostatecznie dużo liczb do dodania algorytm jest optymalny ze względu na koszt).

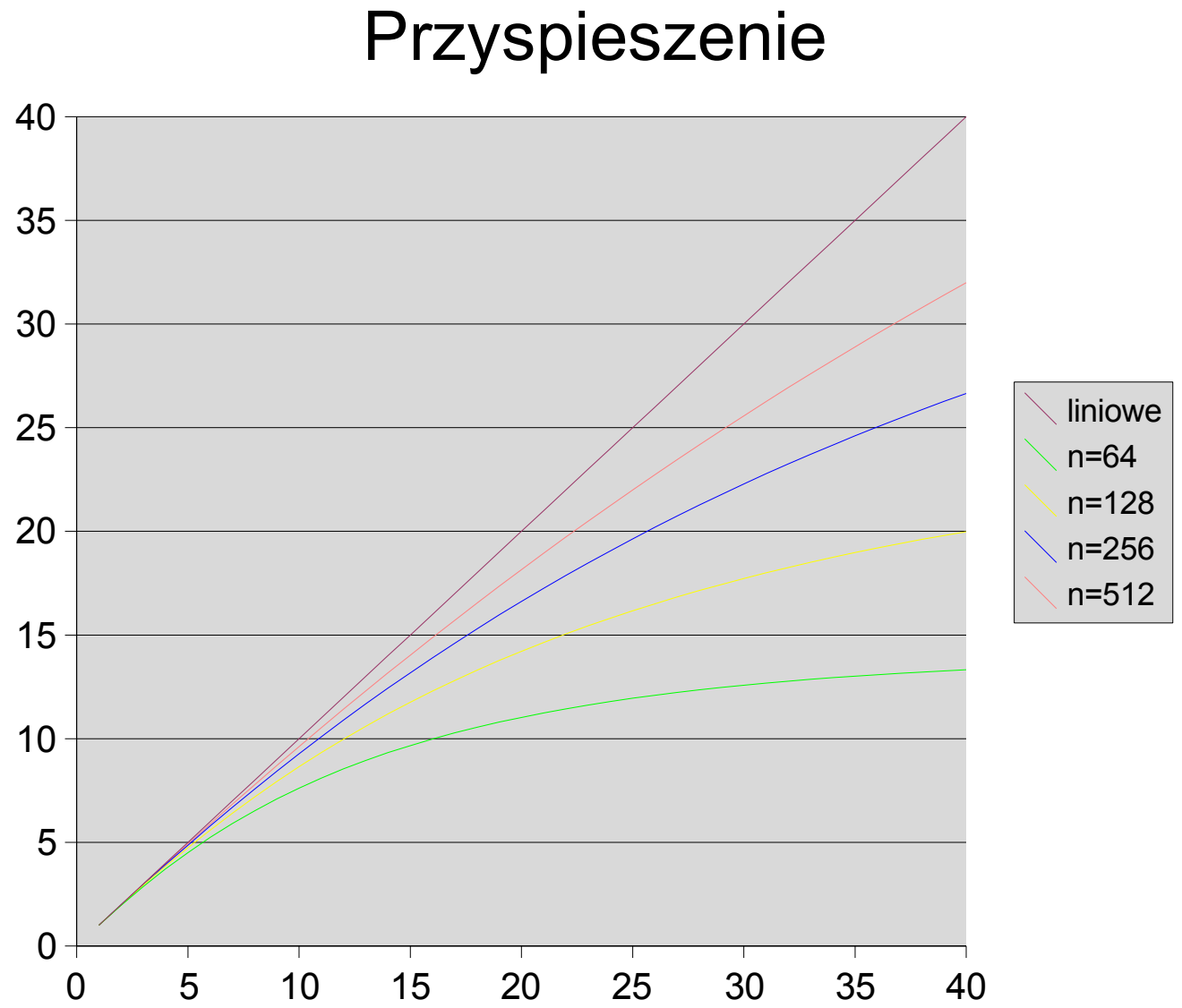
Dodawanie n liczb na p - procesorach - skalowalność

- Tym razem przyspieszenie i wydajność są funkcjami dwóch wielkości: rozmiaru danych n i liczby procesorów p.
- Tym razem przeprowadzimy analizę ze stałymi, a nie asymptotyczną. Załóżmy, że czas przesłania jednej liczby i czas dodawania są sobie równe i wynoszą jedną jednostkę. Zatem każdy krok po zsumowaniu lokalnych liczb zajmuje dwie jednostki. Zatem Tp(n,p)=n/p+2log(p). Przyspieszenie i wydajność wyrażają się jako:

$$S(n,p)=\frac{n}{n/p+2\log(p)} \qquad E(n,p)=\frac{1}{1+\frac{2p\log(p)}{n}}$$

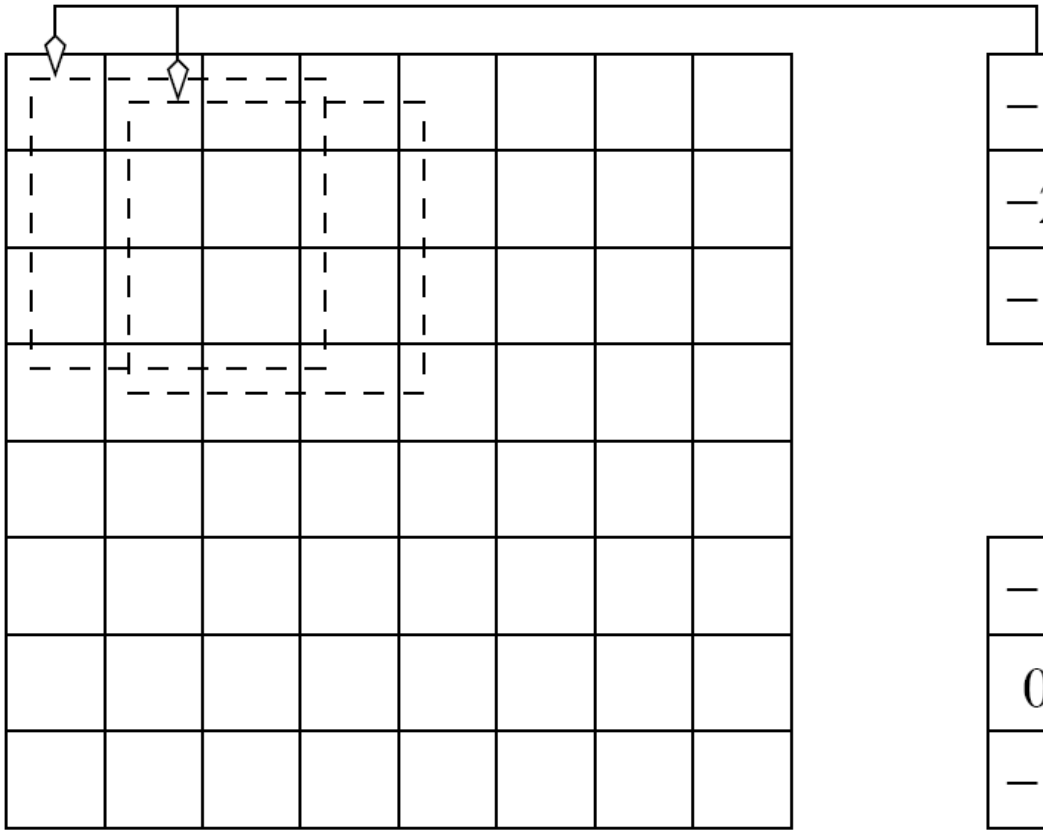
- Widać, że przy wzroście liczby procesorów p i stałym rozmiarze danych n wydajność spada.
- Z drugiej strony przy wzroście n i stałym p wydajność rośnie.
- Jeżeli 2plog(p)=n to wydajność jest stała.
- Jeżeli system (algorytm+maszyna) ma taką własność, że równocześnie zwiększając liczbę procesorów oraz rozmiar danych możemy utrzymać stałą wydajność, to nazywamy go ***systemem skalowalnym***.

Przyspieszenie przy stałym rozmiarze danych



- Przyspieszenie zbiega do pewnej granicy - tym szybciej im mniejszy rozmiar danych.

Przykład 2 (A. Grama i wsp.) : detekcja krawędzi w obrazach



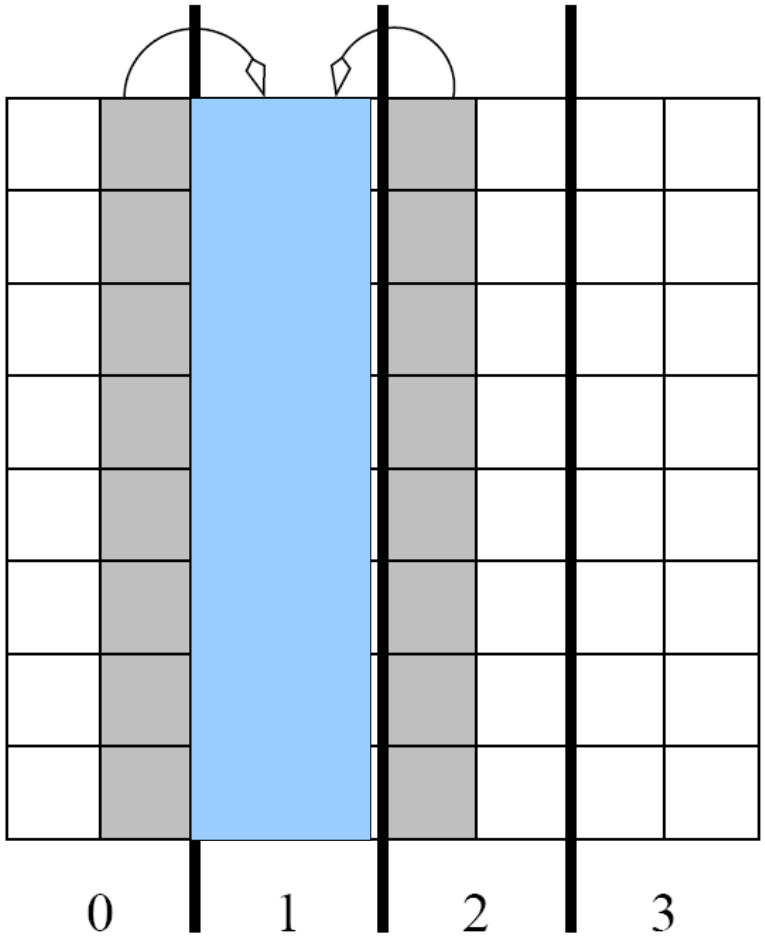
−1	0	1
−2	0	2
−1	0	1

−1	−2	1
0	0	0
−1	2	1

(a) (b)

- Obraz jest dwu wymiarową (nxn) kwadratową macierzą pikseli (a). Detekcja krawędzi polega na przemnożeniu każdego piksela przez maskę. (b) Dwie przykładowe maski o rozmiarach 3x3

Detekcja krawędzi - zrównoleglenie



- Zrównoleglenie oparte na podziale danych. Każdy z procesorów otrzymuje n/p kolumn macierzy.
- Każdy procesor odpowiada za swoje piksele.
- Aby obliczyć wartość pikseli w dwóch swoich skrajnych kolumnach procesor 1 potrzebuje dwóch kolumn: skrajnie lewej z jednego sąsiada i skrajnie prawej z lewego sąsiada.
  - Trzeba mu przesłać dwa komunikaty po n pikseli.
- Skrajne procesory 0 i 3 otrzymują tylko jeden komunikat.
- Na maszynie z przesłaniem komunikatów algorytm wykonuje się w dwóch krokach
  - przesłanie sąsiednich kolumn
  - obliczenie wartości pikseli lokalnego podobrazu.

Detekcja krawędzi - skalowalność

- Zakładamy maszynę z przesyłaniem komunikatów.
- Pierwszy krok - transmisja dwóch sąsiednich kolumn danych, z których każda składa się z n pikseli zajmuje, zgodnie z modelem komunikacji, $2*(t_s+t_w*n)$ czasu, gdzie t_s jest czasem inicjalizacji połączenia, a t_w czasem transmisji jednego słowa (piksela).
- W drugim kroku każdy procesor aplikuje maskę 3x3 do każdego ze swoich n*n/p pikseli. Aplikacja maski o wymiarze 3x3 zajmuje 9 kroków. Czas tego kroku jest równy $9t_c n*n/p$.
- Stąd otrzymujemy:

$$T_p(n,p)=9t_c\frac{n^2}{p}+2(t_S+t_W n)$$

$$T_S(n)=9t_c n^2$$

$$S(p)=\frac{T_S(n)}{T_p(n,p)}=\frac{9t_c n^2}{9t_c\frac{n^2}{p}+2(t_S+t_W n)}$$

Zakończenie

- Takiej analizy jak w ostatnim przykładzie oczekuję w Państwa projektach.

- Ze wzoru na T_p widać że pierwszy krok jest przyspieszany p-krotnie a drugi nie jest przyspieszany w ogóle, co oznacza że spełnione jest prawo Amdahla.

- Jako ćwiczenie proponuję wybranie przykładowych wartości t_c, t_w, t_s i zrobienie wykresów przyspieszenia dla różnych wartości n.
  - Aby symulacja była realistyczna powinno być $t_c \ll t_w \ll t_s$.