Równoległość i współbieżność

Wykonanie sekwencyjne. Poszczególne akcje procesu są wykonywane jedna po drugiej. Dokładniej: kolejna akcja rozpoczyna się po całkowitym zakończeniu poprzedniej.

Praca współbieżna polega na tym, że składające się na nią zjawiska, czynności i działania odbywają się równocześnie. Istotny jest przy tym punkt widzenia obserwatora.

Wykonanie współbieżne. Kolejna akcja rozpoczyna się przed zakończeniem poprzedniej. Nie mówimy nic na temat tego, czy akcje te są wykonywane w tym samym czasie czy też „w przeplocie”.

Wykonanie równoległe. Kilka akcji jest wykonywanych w tym samym czasie. Jest to "prawdziwa" współbieżność, możliwa do uzyskania na komputerze z wieloma procesorami.

Dlaczego warto i należy rozważać programy współbieżne?

– Niektóre problemy są z natury współbieżne, ich rozwiązania dają się łatwo i elegancko wyrazić w postaci niezależnie wykonujących się procedur.

– Współczesne systemy operacyjne zezwalają na uruchamianie wielu procesów jednocześnie.

– Malejące ceny sprzętu sprzyjają powstawaniu architektur wieloprocesorowych, w których można uzyskać prawdziwą współbieżność, tzn. wykonywać jednocześnie wiele procesów na różnych procesorach.

– Rozpowszechnienie się sieci komputerowych.

Przykłady „prac”, które można realizować równolegle

1. Podstawienie

```
X = 3   <- 1 procesor
Y = 4   <- 2 procesor
```

Uwaga – instrukcji:

```
X = 3
Y = X
```

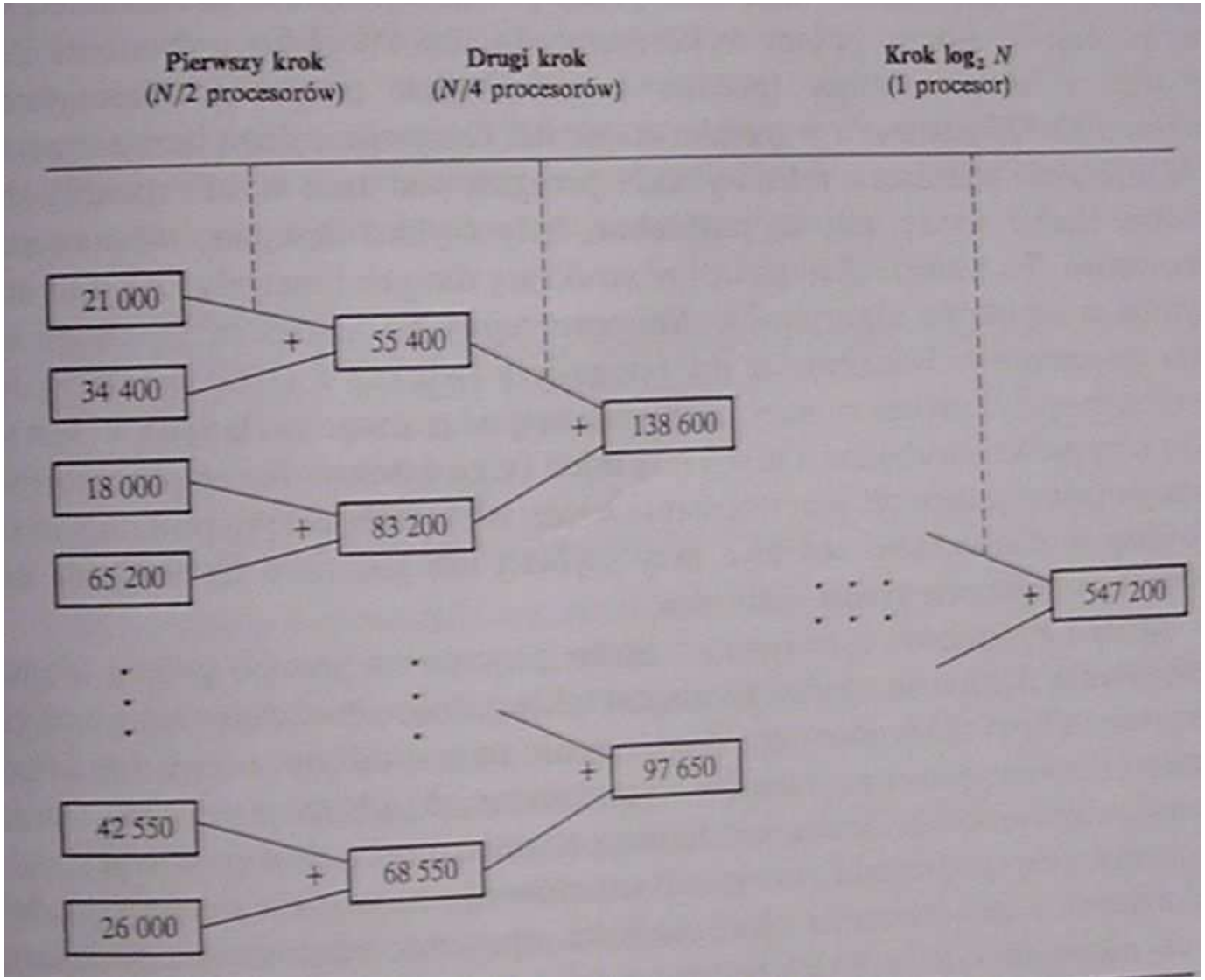
nie można w bezpośredni sposób zrównoleglić

2. Obliczenie wartości wyrażenia `a*b + (c+d)*(e-f)`

3. Sortowanie tablicy liczb

```
qsort(dół, góra):
   jeżeli dół ≥ góra
      // nie więcej niż jeden element – nie rób nic
      koniec
    // cel: podzielić tablicę ze względu na środek.
   filter()
    // cel: posortować rekurencyjnie nowe przedziały
   qsort(dół, środek-1)
   qsort(środek+1, góra)
```

4. Sumowanie elementów tablicy



Rozszerzająca się równoległość

Polepszenie rzędu wielkości np. złożoności czasowej wymaga wzrostu liczby procesorów.

Sumowanie równoległe: aby otrzymać złożoność czasową `O(log n)` potrzebujemy `n/2` procesorów.

Mając do dyspozycji tylko $\sqrt{n}$ procesorów, złożoność czasowa będzie też rzędu $\sqrt{n}$.

Liczbę procesorów potrzebną do wykonania algorytmu równoległego, nazywamy złożonością rozmiaru.

Złożoność iloczynowa

Iloczyn czasu wykonania i rozmiaru (liczby potrzebnych procesorów).

- dla pełnego sumowania równoległego:
  $O(\log n) * O(n) = O(n \log n)$
- dla sumowania „niepełnego”: $O(\sqrt{n}) * O(\sqrt{n}) = O(n)$
- dla równoległego sortowania szybkiego: $O(n) * O(n) = O(n^2)$
- dla sekwencyjnego sortowania szybkiego:
  $1 * O(n \log n) = O(n \log n)$
- „optymalne sortowanie sieciowe”: $O(n) * O(\log n) = O(n \log n)$

Dla złożoności iloczynowej prawdziwe pozostaje dolne ograniczenie dla danego problemu, wyliczone przy założeniu sekwencyjności obliczeń.

Prawo Amdahla

Przyspieszenie uzyskiwane na n procesorach wynosi $\frac{1}{S + \frac{1-S}{n}}$,

gdzie S jest częścią kodu, która nie może być wykonana równolegle.

Przykłady:

Jeżeli na czteroprocesorowym komputerze uruchomimy program, który aż w 95% może być wykonywany równolegle, uzyskamy nieco ponad trzyipółkrotne przyspieszenie.

Jeżeli na szesnastoprocesorowym komputerze uruchomimy program, który tylko w 80% może być wykonany równolegle, zaobserwujemy zaledwie czterokrotny przyrost prędkości wykonania!

W granicy, gdy $n \to \infty$, przyspieszenie wynosi $\frac{1}{S}$. Np. $S = 25\% = 1/4$, przyspieszenie maksymalnie 4–krotne, niezależnie od liczby procesorów.

Systemy rozproszone

Składniki współbieżne są fizycznie od siebie oddalone.
Pojawia się problem komunikacji między składnikami systemu (procesorami).

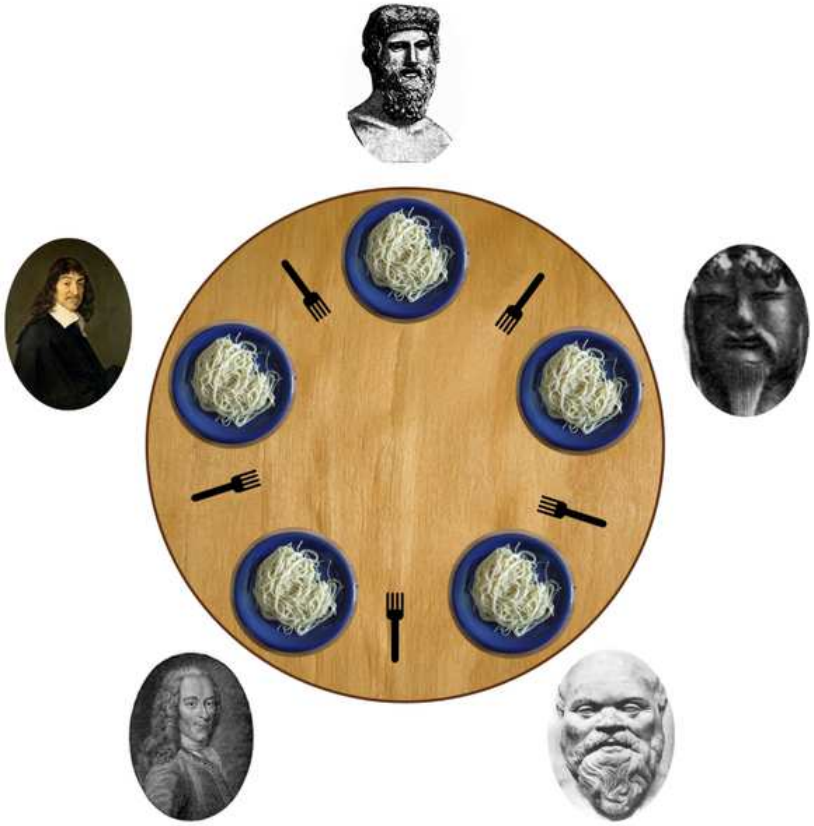
Przykład: tani hotel – jedna łazienka na piętro, wielonarodowi goście, zimny korytarz. Jak sprawić, żeby każdy gość miał możliwość skorzystania z prysznica?

Zasób krytyczny – każdy proces potrzebuje go na wyłączność przez jakiś czas.
Przykłady: prysznic, drukarka, stacja dysków itp.

Problemy do rozwiązania:

- Zapobieganie zastojom (sytuacjom, w której żaden procesor nie może działać dalej)
- Zapobieganie zagłodzeniom (jeden lub więcej procesor, ale nie wszystkie, nie mogą działać dalej)

Problem jedzących filozofów



Ograniczenia: jeść można tylko przy użyciu dwóch widelców, filozof nie ma dostępu do widelców poza zasięgiem.

Jedno z możliwych rozwiązań: odźwierny, który pilnuje, żeby przy stole siedziało co najwyżej `n-1` filozofów.

Każde rozwiązanie musi opierać się na instytucji odźwiernego bądź na pamięci współdzielonej.

Problem czytelników i pisarzy

Problem synchronizacji dostępu do jednego zasobu (pliku, rekordu bazy danych) dwóch rodzajów procesów – dokonujących i nie dokonujących w nim zmian.

W problemie czytelników i pisarzy zasób jest dzielony pomiędzy dwie grupy procesów:

czytelnicy – wszystkie procesy nie dokonujące zmian w zasobie
pisarze – pozostałe procesy

Jednoczesny dostęp do zasobu może uzyskać dowolna liczba czytelników. Pisarz może otrzymać tylko dostęp wyłączny. Równocześnie z pisarzem dostępu do zasobu nie może otrzymać ani inny pisarz, ani czytelnik, gdyż mogłoby to spowodować błędy.

Rozwiązywanie problemów związanych z programowaniem równoległym

Mamy n – procesorów. Każdy z nich wykonuje wielokrotnie pewną własną działalność, po której występuje sekcja krytyczna.

Cykl życia i-tego procesora:

```
Powtarzaj bez końca
   - wykonuj własne czynności
   - wykonaj sekcję krytyczną
```

Własne czynności każdego z osobna procesora nie mają nic wspólnego z czynnościami pozostałych.
Sekcja krytyczna jest sprawą wszystkich procesorów: żadne dwa procesory nie mogą być równocześnie w swoich sekcjach krytycznych – problem wzajemnego wykluczania.

Rozwiązanie

`n = 2`, procesory `P1` i `P2`
zmienne pomocnicze: `X1`, `X2`, `Z`

`Z` może mieć wartość `1` albo `2` i mogą ją zmieniać oba procesory – zmienna dzielona

`X1`, `X2` mogą mieć wartości `tak` lub `nie`, czytać ją mogą oba procesory, ale zmieniać tylko procesory z odpowiednim indeksem – zmienne rozproszone

Wartości początkowe: `X1 = nie`, `X2 = nie`, `Z = 1` lub `2` (bez znaczenia)

Procesor P1

Powtarzaj bez końca:

- wykonuj własne czynności, aż do momentu w którym wejście do sekcji krytycznej stanie się pożądane
- X1 = tak
- Z = 1
- czekaj, aż albo X2 = nie, albo Z = 2 (albo obie relacje są spełnione)
- wykonaj sekcję krytyczną
- X1 = nie

Procesor P2

Powtarzaj bez końca:
- wykonuj własne czynności, aż do momentu w którym wejście do sekcji krytycznej stanie się pożądane
- X2 = tak
- Z = 2
- czekaj, aż albo X1 = nie, albo Z = 1 (albo obie relacje są spełnione)
- wykonaj sekcję krytyczną
- X2 = nie

Xi – wskaźnik chęci procesora Pi wejścia do sekcji krytycznej

Z – wskaźnik uprzejmości

Zamieńmy dwie linijki w kodzie:

Procesor P1

```
Powtarzaj bez końca:
    - wykonuj własne czynności, aż do momentu
      w którym wejście do sekcji krytycznej
      stanie się pożądane
    - Z = 1
    - X1 = tak
    - czekaj, aż albo X2 = nie, albo Z = 2
      (albo obie relacje są spełnione)
    - wykonaj sekcję krytyczną
    - X1 = nie
```

Przykład scenariusza błędu

```
X1 = nie  (wartości początkowe)
X2 = nie

P2: Z = 2
P1: Z = 1
P1: X1 = tak
P1: wchodzi do sekcji krytycznej na mocy tego,
    że X2 = nie
P2: X2 = tak
P2: wchodzi do sekcji krytycznej na mocy tego,
    że Z = 1
```

Protokół dla n procesorów

Wprowadzamy n poziomów nalegania X[i]

Procesor Pi

```
Powtarzaj bez końca:
   - wykonuj własne czynności, aż do momentu
     w którym wejście do sekcji krytycznej
     stanie się pożądane
   - dla każdego j od 1 do n-1 wykonuj
       - X[i] = j
       - Z[j] = i
       - czekaj, aż
         albo X[k] < j dla wszystkich k ≠ i,
         albo Z[j] ≠ i
   - wykonaj sekcję krytyczną
   - X[i] = 0
```

Metody programowania równoległego

MPI – Message Passing Interface (1994 r.)

Przesyłanie komunikatów dla poszczególnych procesorów.
Biblioteki dla wielu języków programowania (C/C++, Fortran, Ada itp.)
Szczególnie nadaje się do systemów rozproszonych.

OpenMP (1997 r.)

Uwalnia programistę od bezpośredniego sterowania procesorami.
Zbiór dyrektyw dla kompilatora, kompilator rozdziela zadania
Szczególnie nadaje się do systemów z pamięcią współdzieloną.

Przykłady programowania równoległego

Przykład 1

```
int A[1000];
int i;
for (i=0; i<1000; i++)
   A[i] = 100 * i;
```

1 procesor:

```
for (i=0; i<500; i++)
   A[i] = 100 * i;
```

2 procesor:

```
for (i=500; i<1000; i++)
   A[i] = 100 * i;
```

Realizacja w OpenMP:

```
int A[1000];
int i

#pragma omp parallel for private(i) shared(A)
for (i=0; i<1000; i++)
   A[i] = 100 * i;
#pragma omp end parallel for
```

Przykład 2

```
int a;
int i;
a = 1;
for (i=1; i<1000; i++)
   a = a * i;
```

Próbujemy jak poprzednio:

1 procesor:

```
for (i=1; i<500; i++)
   a = a * i;
```

2 procesor:

```
for (i=500; i<1000; i++)
   a = a * i;
```

Procesor nr 1 może odczytać zmienną `a`, wyliczyć `a*i,` a w tym czasie procesor nr 2 może odczytać zmienną `a`, wyliczyć `a*i` i zapisać nową wartość do zmiennej `a`!

Zmienna `a` jest zasobem krytycznym, a operacja jej odczytu, wyliczenia nowej wartości i zapisu jest sekcją krytyczną.

1 procesor:

```
for (i=1; i<500; i++)
   sekcja krytyczna: a = a * i;
```

2 procesor:

```
for (i=500; i<1000; i++)
   sekcja krytyczna: a = a * i;
```

Inne rozwiązanie:

1 procesor:

```
for (i=1; i<500; i++)
   a1 = a1 * i;
```

2 procesor:

```
for (i=500; i<1000; i++)
   a2 = a2 * i;
```

teraz czekamy, aż oba procesory skończą działanie i zwrócą wynik i wykonujemy (na dowolnym z nich)

```
a = a1 * a2
```

Realizacja w OpenMP jest prosta (problem jest bardzo często spotykany):

```
#pragma omp parallel for private(i) shared(a) reduction(*:a)
for (i=0; i<1000; i++)
   a = a * i;
#pragma omp end parallel for
```

Przykład 3

```
wynik1 = jakieś czasochłonne obliczenie nr 1
         (niezależne od obliczenia nr 2)
zapisz wynik1 do pliku
wynik2 = jakieś czasochłonne obliczenie nr 2
         (niezależne od obliczenia nr 1)
zapisz wynik2 do pliku
wynik = wynik1 + wynik2
zapisz wynik do pliku
```

Procesor1:

```
wynik1 = jakieś czasochłonne obliczenie nr 1
         (niezależne od obliczenia nr 2)
sekcja krytyczna: zapisz wynik1 do pliku
```

Procesor2:

```
wynik2 = jakieś czasochłonne obliczenie nr 2
         (niezależne od obliczenia nr 1)
sekcja krytyczna: zapisz wynik2 do pliku

czekaj na wyniki z poszczególnych procesorów
```

Procesor 1 lub 2 (albo obydwa naraz, nic się złego nie stanie)

```
wynik = wynik1 + wynik2
```

Procesor 1 lub 2 (ale tylko jeden z nich!)

```
zapisz wynik do pliku
```

Realizacja w OpenMP

```
#pragma omp parallel sections
{
#pragma omp section

  wynik1 = obliczenie nr 1
  #pragma omp critical
    zapisz wynik1 do pliku
  #pragma omp end critical

#pragma omp section

  wynik2 = obliczenie nr 2
  #pragma omp critical
    zapisz wynik2 do pliku
  #pragma omp end critical
}
```

```
#pragma omp barrier
wynik = wynik1 + wynik2

#pragma omp master
zapisz wynik do pliku
#pragma omp end master

#pragma omp end parallel sections
```