Zależności

Przeszkody przy zrównoleglaniu algorytmów

- Zależności – wzajemne uzależnienie instrukcji nakładające ograniczenia na kolejność ich realizacji
- Zależności zasobów
  - kiedy wiele wątków jednocześnie usiłuje korzystać z wybranego zasobu (np. pliku lub terminala)
- Zależności sterowania
  - kiedy wykonanie danej instrukcji zależy od rezultatów poprzedzających instrukcji warunkowych
- Zależności danych (zależności przepływu)
  - kiedy instrukcje wykonywane w bezpośrednim sąsiedztwie czasowym operują na tych samych danych i choć jedna z tych instrukcji dokonuje zapisu

Przeszkody przy zrównoleglaniu algorytmów

- Zależności danych
  - zależności wyjścia (zapis po zapisie, *write-after-write*)

    a := .....

    a := .....
    - Zależności wyjścia nie są rzeczywistym problemem; odpowiednia analiza kodu może doprowadzić do wniosku o możliwej eliminacji instrukcji lub przemianowaniu zmiennych
  - anty-zależności (zapis po odczycie, *write-after-read*)

    ... := a

    a := .....
    - także tutaj odpowiednie przemianowanie zmiennych może zlikwidować problem

Problemy współbieżności – zależności

- Zależności danych
  - zależności rzeczywiste (odczyt po zapisie, *read-after-write*)

    a := .....

    ... := a
  - zależności rzeczywiste uniemożliwiają zrównoleglenie algorytmu – konieczne jest jego przeformułowanie w celu uzyskania wersji możliwej do zrównoleglenia
- Wszystkie powyższe zależności mogą występować albo jawnie jak w podanych przykładach, lub niejawnie jako tzw. zależności przenoszone przez pętle
- Badając zależności należy zwrócić uwagę na instrukcje modyfikujące wartości zmiennych (*read-modify-write*, np a++), które zawierają i odczyt, i zapis

Przeszkody przy zrównoleglaniu algorytmów

- Zależności danych przenoszone przez pętle
  - zależności wyjścia (*output dependencies)*
    - ewentualne zrównoleglenie działania po przemianowaniu zmiennych musi zachować kolejność zapisu danych
  - anty-zależności (*anti-dependencies*)
    - możliwe zrównoleglenie działania poprzez przemianowanie zmiennych
  - zależności rzeczywiste (*true dependencies*)
- W przypadku użycia wskaźników automatycznie zrównoleglające kompilatory mogą podejrzewać występowanie utożsamienia (zamienności) nazw (*aliasing)*

Przykłady

- Zależności w kodzie:

```
x = 2*y + w;
y = 2*z - cos(w);
z = log(w) + y;
x += sin(w);
```

- Możliwe zrównoleglenie
  - usunięcie zależności
  - narzut wykonania równoległego
- Przykłady zależności przenoszonych w pętli
  - zrównoleglenie
  - narzut wykonania równoległego
  - poprawność zależna od operacji w „skrajnych” iteracjach

Przeszkody przy zrównoleglaniu algorytmów

- W celu przeprowadzenia analizy zależności w programie można konstruować i badać grafy zależności
- Dla praktycznie występujących programów konstrukcja i analiza grafu zależności przekracza możliwości komputerów (problem jest NP-trudny)
- Kompilatory zrównoleglające stosują rozmaite metody heurystyczne analizy kodu
- W wielu wypadkach analiza nie powodzi się (zwłaszcza, gdy np. poszczególne wątki wywołują procedury)
- Ingerencja projektanta algorytmu i programisty okazuje się być często jedynym sposobem na uzyskanie efektywnego programu równoległego

Przykład – wersja sekwencyjna

```
int main(){ // całkowanie f(x) w przedziale (a,b) metodą trapezów
    int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
    ... // nadanie wartości zmiennym: a, b, n
    dx = (b-a) / n;
    x1 = a;
    for(i=0; i<n; i++){
            x2 = x1 + dx;
            calka += 0.5*( f(x1) + f(x2) )*dx;
            x1 = x2;
} }   }
```

Przykład – wersja sekwencyjna

```
int main(){ // całkowanie f(x) w przedziale (a,b) metodą trapezów
    int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
    ... // nadanie wartości zmiennym: a, b, n
    dx = (b-a) / n;
    x1 = a;
    for(i=0; i<n; i++){
            x2 = x1 + dx;
            calka += 0.5*( f(x1) + f(x2) )*dx;
            x1 = x2;
} }   }
```

Przykład – wersja równoległa OpenMP

```
int main(){   // całkowanie f(x) w przedziale (a,b) metodą trapezów
    int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
    ... // nadanie wartości zmiennym: a, b, n
#pragma omp  parallel  firstprivate(a,b,n)  private(dx,x1,x2)
   shared(calka)
    {
        dx = (b-a) / n;
#pragma omp  for  reduction(+:calka)
        for(i=0; i<n; i++){
            x1 = a + i*dx;   x2 = x1 + dx;
            calka += 0.5*( f(x1) + f(x2) )*dx;
} }    }
```