Programowanie współbieżne

Ćwiczenia 14 – Współbieżne dziel i rządź cz. 2

Zadanie 1: Floyd-Warshall

Algorytm Floyda-Warshalla wyznacza najkrótsze ścieżki między każdą parą wierzchołków w grafie, w którym krawędzie mają nieujemne wagi. Poniżej funkcja implementująca ten algorytm.

```
void seqFW(int const * const * g, int * const * res, int n) {
  int i, j, k;
  for (j = 0; j < n; ++j) {
    for (i = 0; i < n; ++i) {
      res[j][i] = g[j][i];
    }
  }
  for (k = 0; k < n; ++k) {
    for (j = 0; j < n; ++j) {
      for (i = 0; i < n; ++i) {
        res[j][i] = min2(res[j][i], res[j][k] + res[k][i]);
      }
    }
  }
}
```

Napisz efektywną równoległą implementację tego algorytmu. Oblicz pracę (ang. *work*), długość ścieżki krytycznej (ang. *span*) oraz równoległość (ang. *parallelism*) zaproponowanego algorytmu.

```
void parFW(int const * const * g, int * const * res, int n) {
  int i, j, k;
  parallel for (j = 0; j < n; ++j) {
    parallel for (i = 0; i < n; ++i) {
      res[j][i] = g[j][i];
    }
  }
  for (k = 0; k < n; ++k) {
    parallel for (j = 0; j < n; ++j) {
      parallel for (i = 0; i < n; ++i) {
        res[j][i] = min2(res[j][i], res[j][k] + res[k][i]);
      }
    }
  }
}
```

Work: $T_1(N) = \Theta(N^2) + \Theta(N^3) = \Theta(N^3)$.
Span: $T_\infty(N) = \Theta(\log N) + \Theta(N \log N) = \Theta(N \log N)$.
Parallelism: $T_1(N)/T_\infty(N) = \Theta(\frac{N^2}{\log N})$.

Zauważmy, że dla pojedynczej komórki res[r][c] możliwy jest konflikt odczyt-zapis. Arbiter pamięci rozwiązuje te konflikty poprzez sekwencjonowanie operacji. Kolejność konfliktujących operacji w sekwencji nie ma wpływu na poprawność powyższego algorytmu – w „najgorszym" przypadku proces przeczyta wartość komórki z następnej iteracji (po k).

Zadanie 2: Transpozycja macierzy

Rozważmy następujący kod do transpozycji macierzy $N \times N$.

```
void parTranspose(double * const * m, int n) {
  int i, j;
  parallel for (i = 1; i < n; ++i) {
    parallel for (j = 0; j < i; ++j) {
      double tmp = m[i][j];
      m[i][j] = m[j][i];
      m[j][i] = tmp;
    }
  }
}
```

Przeprowadź analizę efektywności tego algorytmu. Jak zmieniłaby się efektywność, gdyby usunąć słówko **parallel** w wewnętrznej pętli?

Work: $T_1(N) = \sum_{i=1}^{N-1} \big(i \cdot \Theta(1)\big) = \Theta(N^2)$.

Span: $T_\infty(N) = \Theta(\log N) + \max_{1 \le i < N} iter_\infty(i)$, gdzie $iter_\infty(i)$ oznacza długość ścieżki krytycznej wewnętrznej pętli **parallel for** w zależności od wartości i z zewnętrznej pętli. Mamy więc $iter_\infty(i) = \Theta(\log i) + \max_{0 \le j < i} iter'_\infty(i,j)$, gdzie $iter'_\infty(i,j)$ to długość ścieżki krytycznej ciała wewnętrznej pętli **parallel for** w zależności od wartości i oraz j. Jako że dla dowolnych i oraz j mamy $iter'_\infty(i,j) = \Theta(1)$, to dostajemy $iter_\infty(i) = \Theta(\log i)$ i ostatecznie $T_\infty(N) = \Theta(\log N) + iter_\infty(N-1) = \Theta(\log N)$.

Parallelism: $T_1(N)/T_\infty(N) = \Theta(\frac{N^2}{\log N})$.

W przypadku usunięcia słówka **parallel** z wewnętrznej pętli, powyższe wartości zmieniłyby się następująco.

Work: Nie zmienia się – $T_1(N) = \Theta(N^2)$.

Span: $T_\infty(N) = \Theta(\log N) + \max_{1 \le i < N} iter_\infty(i)$, gdzie $iter_\infty(i)$ oznacza długość ścieżki krytycznej wewnętrznej pętli **for** w zależności od wartości i z zewnętrznej pętli. Mamy więc $iter_\infty(i) = \Theta(i)$ i ostatecznie $T_\infty(N) = \Theta(\log N) + iter_\infty(N-1) = \Theta(N)$.

Parallelism: $T_1(N)/T_\infty(N) = \Theta(N)$.

Zadanie 3: Mnożenie macierzy

Zapisz algorytm mnożenia macierzy $N \times N$ wykorzystując konstrukcję **parallel for** (bez możliwości używania operacji **spawn** i **sync**). Oblicz efektywność zaproponowanego algorytmu.

```
void parMatMul(
    double const * const * a, double const * const * b,
    double * const * c, int n
) {
  int i, j, k;
  parallel for (j = 0; j < n; ++j) {
    parallel for (i = 0; i < n; ++i) {
      c[j][i] = 0.0;
      for (k = 0; k < n; ++k) {
        c[j][i] += a[j][k] * b[k][j];
      }
    }
  }
}
```

Work: $T_1(N) = \Theta(N^3)$.

Span: $T_\infty(N) = \Theta(\log N) + \Theta(\log N) + \Theta(N) = \Theta(N)$.

Parallelism: $T_1(N)/T_\infty(N) = \Theta(N^2)$.

Z powodu konfliktów w dostępie do zmiennej `c[j][i]`, w wewnętrznej pętli nie można użyć **parallel for**.

Zadanie 4: Mnożenie macierzy raz jeszcze

Popraw powyższy algorytm poprzez implementację redukcji przy wykorzystaniu operacji **spawn** i **sync**. Oblicz efektywność zaproponowanego algorytmu.

```
void parMatMul2(
    double const * const * a, double const * const * b,
    double * const * c, int n
) {
  int i, j, k;
  parallel for (j = 0; j < n; ++j) {
    parallel for (i = 0; i < n; ++i) {
      c[j][i] = parDotProd(a, b, j, i, 0, n − 1);
    }
  }
}
double parDotProd(
    double const * const * a, double const * const * b,
    int j, int i, int s, int e
) {
  if (s == e) {
    return a[j][s] * b[s][i];
  } else {
    int m = (s + e) / 2;
    double lv = spawn parDotProd(a, b, j, i, s, m);
    double rv = parDotProd(a, b, j, i, m + 1, e);
    sync;
    return lv + rv;
  }
}
```

Work: Oznaczmy $T_1^{DPji}(N)$ jako pracę funkcji parDotProd(∘, ∘, j, i, 0, N − 1). Mamy wtedy $T_1^{DPji}(N) = 2T_1^{DPji}(\frac{N}{2}) + \Theta(1)$. Z Twierdzenia Mistrza otrzymujemy $T_1^{DPji}(N) = \Theta(N)$. Jako że $T_1^{DPji}(N)$ nie zależy od i oraz j, to sumaryczna praca algorytmu wynosi $T_1(N) = N^2 * \Theta(N) = \Theta(N^3)$.

Span: Analogicznie, $T_\infty^{DPji}(N) = T_\infty^{DPji}(\frac{N}{2}) + \Theta(1) = \Theta(\log N)$ nie zależy od i ani j. Wobec tego sumaryczna długość ścieżki krytycznej wynosi $T_\infty(N) = \Theta(\log N) + \Theta(\log N) + \Theta(\log N) = \Theta(\log N)$.

Parallelism: $T_1(N)/T_\infty(N) = \Theta(\frac{N^3}{\log N})$.

Zadanie 5: Transpozycja macierzy raz jeszcze

Napisz efektywny równoległy algorytm transpozycji macierzy kwadratowej bez wykorzystywania **parallel for**, to jest używając tylko **spawn** i **sync**. Oblicz pracę, długość ścieżki krytycznej oraz równoległość zaproponowanego algorytmu.

```
void parTransposeRec(double * const * m, int r, int c, int n) {
  if (n > 1) {
    int hn = n / 2;
    spawn parTransposeRec(m, r, c, nh);
    spawn parTransposeRec(m, r + nh, c + nh, n − nh);
    parSwapRec(m, r, c + nh, r + nh, c, nh, n − nh);
    sync;
  }
}
```

```
void parSwapRec(double * const * m, int r1, int c1, int r2, int c2, int n1, int n2) {
  /* Swap the n1 x n2 submatrix starting at (r1, c1)  */
  /* with the n2 x n1 submatrix starting at (r2, c2). */
  if (n1 < n2) {
    parSwapRec(m, r2, c2, r1, c1, n2, n1);
  } else if (n1 == 1) {
    double tmp = m[r1][c1];
    m[r1][c1] = m[r2][c2];
    m[r2][c2] = tmp;
  } else {
    int nm = n1 / 2;
    spawn parSwapRec(m, r2, c2, r1, c1, n2, nm);
    parSwapRec(m, r2, c2 + nm, r1 + nm, c1, n2, n1 - nm);
    sync;
  }
}
```

Work: Oznaczmy $T_1^S(M)$ jako pracę funkcji parSwapRec na M-elementowej macierzy (zwróćmy uwagę, że mówimy o liczbie elementów a nie wierszy czy kolumn). Mamy wtedy $T_1^S(M) = 2T_1^S(\frac{M}{2}) + \Theta(1)$. Z Twierdzenia Mistrza otrzymujemy $T_1^S(M) = \Theta(M)$.

Pracę całego algorytmu na M-elementowej macierzy obliczymy zakładając, że macierz jest kwadratowa. Mamy wtedy $T_1(M) = 2T_1(\frac{M}{4}) + T_1^S(\frac{M}{4}) = 2T_1(\frac{M}{4}) + \Theta(M)$. Z Twierdzenia Mistrza otrzymujemy $T_1(M) = \Theta(M) = \Theta(N^2)$. Algorytm jest więc optymalny pod względem pracy.

Span: Analogicznie, $T_\infty^S(M) = T_\infty^S(\frac{M}{2}) + \Theta(1) = \Theta(\log M)$.

Długość ścieżki krytycznej całego algorytmu dla macierzy M-elementowej jest to maksimum z długości ścieżki krytycznej dwóch zejść rekurencyjnych dla macierzy $\frac{M}{4}$-elementowej (plus czynnik stały) oraz długości ścieżki krytycznej $T_\infty^S(\frac{M}{4}) = \Theta(\log M)$. Jako że równanie $R(i) = R(\frac{i}{4}) + \Theta(1)$ ma rozwiązanie $R(i) = \Theta(\log i)$, to ostatecznie długość ścieżki krytycznej ma postać $T_\infty(M) = \Theta(\log M) = \Theta(\log N)$. Tak więc algorytm jest też optymalny pod względem długości ścieżki krytycznej.

Parallelism: $T_1(N)/T_\infty(N) = \Theta(\frac{N^2}{\log N})$.