Programowanie współbieżne

Ćwiczenia 13 – Współbieżne dziel i rządź cz. 1

Zadanie 1: Redukcja

$\otimes$-redukcją tablicy $t[N]$, gdzie $N \geq 1$ zaś $\otimes$ jest pewną operacją łączną, nazwiemy wartość wyrażenia $t[0] \otimes t[1] \otimes t[2] \otimes \cdots \otimes t[N-2] \otimes t[N-1]$. Napisz efektywny algorytm równoległej redukcji wykorzystujący paradygmat dziel i rządź, to jest operacje **spawn** i **sync**. Oblicz pracę (ang. *work*), długość ścieżki krytycznej (ang. *span*) oraz równoległość (ang. *parallelism*) zaproponowanego algorytmu.

```
typedef ... val_t;
val_t parReduce(val_t * t, int s, int e) {
  if (s == e) {
    return t[s];
  } else {
    int m = (s + e) / 2;
    val_t lv = spawn parReduce(t, s, m);
    val_t rv = parReduce(t, m + 1, e);
    sync;
    return lv ⊗ rv;
  }
}
```

Powyższa funkcja jest wołana następująco: val_t wynik = parReduce(t, 0, N − 1);.

Work: $T_1(N) = 2T_1(\frac{N}{2}) + \Theta(1)$, gdzie $T_1(1) = \Theta(1)$. Z Twierdzenia Mistrza otrzymujemy zatem $T_1(N) = \Theta(N)$.

Span: $T_\infty(N) = T_\infty(\frac{N}{2}) + \Theta(1)$, gdzie $T_\infty(1) = \Theta(1)$. Rozwijając rekursję dostajemy $T_\infty(N) = \Theta(\log N)$.

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

Zadanie 2: „Krótsza" redukcja

Dla skrócenia notacji, pan Szymon Hakerski zaimplementował powyższy algorytm w następujący sposób:

```
typedef ... val_t;
val_t parReduce2(val_t * t, int s, int e) {
  val_t res = t[s];
  parallel for (int i = s + 1; i <= e; ++i) {
    res ⊗= t[i];
  }
  return res;
}
```

Czy powyższa implementacja jest poprawna? Czy jest ona efektywna?

Odpowiedź: Aby odpowiedzieć na te pytania, zauważmy, iż nasz model zakłada arbitra pamięci. Oznacza to, że zapisy i odczyty do tej samej komórki pamięci są sekwencjonowane. W szczególności, wartości, które znajdą się w zmiennej res i zostaną z niej odczytane, są wartościami, które musiały być wcześniej zapisane.

Z tego powodu, poprawność zależy od implementacji operatora $\otimes=$.

Jeśli jest on zaimplementowany w ten sposób, iż proces najpierw wczytuje wartości res oraz t[i], następnie wykonuje na nich $\otimes$ i wreszcie wynik zapisuje na res, to powyższy algorytm nie będzie poprawny, ponieważ możemy zgubić wyniki niektórych operacji $\otimes$.

Jeśli natomiast operacja res $\otimes$= t[i] jest zaimplementowana w ten sposób, iż najpierw wczytuje t[i] a następnie *atomowo* wczytuje res, wykonuje $\otimes$ i wynik zapisuje na res, to algorytm będzie poprawny. Innymi słowy, poprawność jest zapewniona przy atomowej instrukcji atomic_fetch_$\otimes$_set. Są jeszcze inne warianty instrukcji atomowych, dzięki którym można uzyskać ten sam efekt. Niemniej jednak nie mamy gwarancji, iż takie instrukcje atomowe istnieją, w szczególności jeśli $\otimes$ jest bardzo złożoną operacją.

Jednakże nawet przy poprawnej implementacji będzie ona nieefektywna.

Praca algorytmu się nie zmieni poniżej liniowej, ponieważ każda wartość t[i] musi być odczytana co najmniej raz. Jest to spójne ze sposobem liczenia pracy dla **parallel for** polegającym na wykreśleniu słowa **parallel** i policzeniu złożoności czasowej wynikowego algorytmu sekwencyjnego.

Ścieżka krytyczna algorytmu będzie co najmniej logarytmiczna, ponieważ rozwijanie pętli **parallel for** jest wykonywane w czasie logarytmicznym. W praktyce ścieżka krytyczna będzie natomiast jeszcze gorsza — liniowa. Jest to spowodowane tym, iż każda z N operacji res $\otimes$= t[i] wymaga zapisu do zmiennej res. Jako że arbiter pamięci sekwencjonuje takie zapisy, będą się one odbywały jeden po drugim. W efekcie ścieżka krytyczna będzie proporcjonalna do N.

Innymi słowy, nawet jeśli algorytm Szymona Hakerskiego byłby poprawny, co nie jest gwarantowane przy naszych założeniach, to i tak jest on nieefektywny.

Zadanie 3: Obliczenia prefiksowe

Problemem związanym z $\otimes$-redukcją jest $\otimes$-obliczenie-prefiksowe. Mając daną tablicę $x[N]$, gdzie $N \geq 1$ zaś $\otimes$ jest pewną operacją łączną, tablicę $y[N]$ nazwiemy wartością $\otimes$-obliczenia-prefiksowego w.t.w., gdy:

$$\begin{aligned} y[0] &= x[0], \\ y[1] &= x[0] \otimes x[1], \\ y[2] &= x[0] \otimes x[1] \otimes x[2], \\ &\dots, \\ y[N-1] &= x[0] \otimes x[1] \otimes x[2] \otimes \dots \otimes x[N-2] \otimes x[N-1]. \end{aligned}$$

Innymi słowy, sekwencyjny algorytm dla obliczeń prefiksowych może wyglądać następująco:

```
typedef ... val_t;
void seqScan(val_t * x, val_t * y, int n) {
  y[0] = x[0];
  for (int i = 1; i < n; ++i) {
    y[i] = y[i - 1] ⊗ x[i];
  }
}
```

Napisz efektywny algorytm równoległego obliczenia prefiksowego wykorzystujący paradygmat dziel i rządź. Oblicz pracę, długość ścieżki krytycznej oraz równoległość zaproponowanego algorytmu.

Rozwiązanie można zacząć od następującego algorytmu, wykorzystującego w oczywisty sposób wcześniejszy algorytm redukcji.

```
typedef ... val_t;
void parScan1(val_t * x, val_t * y, int n) {
  parallel for (int i = 0; i < n; ++i) {
    y[i] = parReduce(x, 0, i);
  }
}
```

Work: $T_1(N) = \sum_{i=0}^{N-1} \big(T_1\big(parReduce(\circ, 0, i)\big) + \Theta(1)\big) = \sum_{i=0}^{N-1} \big(\Theta(i - 0 + 1) + \Theta(1)\big) = \sum_{i=0}^{N-1} \Theta(i) = \Theta(N^2)$.

Span: $T_\infty(N) = \Theta(\log N) + \max_{0 \le i \le N-1} T_\infty\big(parReduce(\circ, 0, i)\big) = \Theta(\log N) + \Theta(\log N) = \Theta(\log N)$.

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

Jeśli chodzi o długość ścieżki krytycznej w naszym algorytmie równoległym, to jest ona optymalna. Natomiast praca jest nieoptymalna w porównaniu z pracą algorytmu sekwencyjnego ($\Theta(N)$), wobec czego ten powyższy algorytm równoległy jest nieefektywny.

Drugie podejście do algorytmu może zakładać także, że robimy równoległą redukcję mniejszych podprzedziałów a następnie wyniki ostatniego elementu w lewym podprzedziale są propagowane do wszystkich elementów prawego podprzedziału. Algorytm wygląda więc następująco.

```
typedef ... val_t;
void parScan2(val_t * x, val_t * y, int s, int e) {
  if (s == e) {
    y[s] = x[s];
  } else {
    int m = (s + e) / 2;
    spawn parScan2(x, y, s, m);
    parScan2(x, y, m + 1, e);
    sync;
    /* Invariant: for all i in [s, m], y[i] == x[s] ⊗ x[s + 1] ⊗ ··· ⊗ x[i − 1] ⊗ x[i] */
    /* Invariant: for all i in [m + 1, e], y[i] == x[m + 1] ⊗ x[m + 2] ⊗ ··· ⊗ x[i − 1] ⊗ x[i] */
    parallel for (int i = m + 1; i <= e; ++i) {
      y[i] = y[m] ⊗ y[i];
    }
  }
  /* Invariant: for all i in [s, e], y[i] == x[s] ⊗ x[s + 1] ⊗ ··· ⊗ x[i − 1] ⊗ x[i] */
}
```

Work: $T_1(N) = 2T_1(\frac{N}{2}) + \Theta(N)$, gdzie $T_1(1) = \Theta(1)$. Z Twierdzenia Mistrza otrzymujemy zatem $T_1(N) = \Theta(N \log N)$.

Span: $T_\infty(N) = T_\infty(\frac{N}{2}) + \Theta(\log N)$, gdzie $T_\infty(1) = \Theta(1)$. Rozwijając rekursję dostajemy $T_\infty(N) = T_\infty(\frac{N}{4}) + \Theta(\log \frac{N}{2}) + \Theta(\log N) = T_\infty(\frac{N}{4}) + \Theta(-1 + \log N) + \Theta(\log N) = \Theta(1 + 2 + 3 + \cdots + \log N - 1 + \log N) = \Theta(\log^2 N)$.

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

Innymi słowy, poprawiliśmy pracę kosztem ścieżki krytycznej. Jednakże obecnie w obu tych kryteriach algorytm jest nieoptymalny.

Trzecie podejście do algorytmu zakłada wcześniejszą redukcję jedynie „kluczowych” wartości y[i] a następnie spropagowanie tych wartości w drugiej fazie tam, gdzie trzeba. Te „kluczowe” wartości przechowywane są w dodatkowej tablicy t[N]. Algorytm może działać w dwóch fazach, ponieważ zejścia rekurencyjne w drugiej fazie będą obrabiały te same przedziały, co zejścia w fazie pierwszej. Jego treść jest następująca.

```
typedef ... val_t;
void parScan3(val_t * x, val_t * y, int n) {
  y[0] = x[0];
  if (n > 1) {
    val_t * t = new val_t[n];
    parScan3Red(x, t, 1, n − 1);
    parScan3Prop(x[0], x, t, y, 1, n − 1);
    delete[] t;
  }
}
val_t parScan3Red(val_t * x, val_t * t, int s, int e) {
  if (s == e) {
    return x[s];
```

```
  } else {
    int m = (s + e) / 2;
    int lv = spawn parScan3Red(x, t, s, m);
    int rv = parScan3Red(x, t, m + 1, e);
    sync;
    t[m] = lv;
    /* Invariant: t[m] == x[s] ⊗ x[s + 1] ⊗ ··· ⊗ x[m] */
    return lv ⊗ rv;
  }
  /* Invariant: returns x[s] ⊗ x[s + 1] ⊗ ··· ⊗ x[e − 1] ⊗ x[e] */
}
void parScan3Prop(val_t v, val_t * x, val_t * t, val_t * y, int s, int e) {
  /* Invariant: v == x[0] ⊗ x[1] ⊗ ··· ⊗ x[s − 1] */
  if (s == e) {
    y[s] = v ⊗ x[s];
  } else {
    int m = (s + e) / 2;
    /* Invariant: t[m] == x[s] ⊗ x[s + 1] ⊗ ··· ⊗ x[m] */
    spawn parScan3Prop(v, x, t, y, s, m);
    parScan3Prop(v ⊗ t[m], x, t, y, m + 1, e);
    sync;
  }
}
```

Do szacowania efektywności algorytmu przyjmijmy, że T_1^R i T_∞^R oznaczają odpowiednio pracę i długość ścieżki krytycznej funkcji parScan3Red zaś T_1^P i T_∞^P te same wartości dla funkcji parScan3Prop.

Work: $T_1^R(N) = 2T_1^R(\frac{N}{2}) + \Theta(1)$, gdzie $T_1^R(1) = \Theta(1)$. Jak dla redukcji zatem $T_1^R(N) = \Theta(N)$. Tak samo, $T_1^P = \Theta(N)$. Ostatecznie $T_1(N) = \Theta(1) + T_1^R(N) + T_1^P(N) = \Theta(N)$, zakładając, że alokacja i zwalnianie pamięci mają złożoność czasową $\Theta(1)$. Praca wykonywana przez powyższy algorytm jest zatem optymalna.

Span: $T_\infty^R(N) = T_\infty^R(\frac{N}{2}) + \Theta(1)$, gdzie $T_\infty^R(1) = \Theta(1)$. Jak dla redukcji zatem $T_\infty^R(N) = \Theta(\log N)$. Tak samo, $T_\infty^P = \Theta(\log N)$. Ostatecznie $T_\infty(N) = \Theta(1) + T_\infty^R(N) + T_\infty^P(N) = \Theta(\log N)$, ponownie zakładając, że alokacja i zwalnianie pamięci mają złożoność czasową $\Theta(1)$. Długość ścieżki krytycznej powyższego algorytmu jest zatem także optymalna.

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

Zadanie 4: Maksymalna podtablica

Dana jest N-elementowa tablica liczb całkowitych. Problem maksymalnej podtablicy polega na znalezieniu ciągłej podtablicy o maksymalnej sumie elementów. Przykładowo, dla tablicy o elementach 1, -4, 3, -7, 5, -3, 2, 2, -6, 4, -1, maksymalna podtablica składa się z elementów 5, -3, 2, 2 i ma sumę 6. Dla tablicy zawierającej jedynie ujemne liczby, maksymalną podtablicą jest podtablica pusta o sumie 0.

Optymalny algorytm sekwencyjny ma złożoność $\Theta(N)$. Napisz efektywny algorytm równoległy i oblicz jego pracę, długość ścieżki krytycznej oraz równoległość. Na nasze potrzeby wystarczy, aby wynikiem algorytmu była suma elementów maksymalnej podtablicy.

```
typedef struct interval_s {
  int sum;
  int left;
  int right;
  int center;
} interval_t;
```

```
int maxSubarray(int const * a, int n)
{
  interval_t iv;
  maxSubarrayRec(&iv, a, 0, n - 1);
  return max3(iv.left, iv.right, iv.center);
}

void maxSubarrayRec(interval_t * iv, int const * a, int l, int r) {
  if (l == r) {
    iv->sum = a[l];
    iv->left = iv->right = iv->center = max2(iv->sum, 0);
  } else {
    interval_t liv;
    interval_t riv;
    int mid = (l + r) / 2;
    spawn maxSubarrayRec(&liv, a, l, mid);
    maxSubarrayRec(&riv, a, mid + 1, r);
    sync;
    iv->sum = liv.sum + riv.sum;
    iv->left = max2(liv.left, liv.sum + riv.left);
    iv->right = max2(riv.right, riv.sum + liv.right);
    iv->center = max3(liv.center, riv.center, liv.right + riv.left);
  }
}
```

Work: $T_1(N) = 2T_1(\frac{N}{2}) + \Theta(1)$, gdzie $T_1(1) = \Theta(1)$. Z Twierdzenia Mistrza otrzymujemy zatem $T_1(N) = \Theta(N)$.

Span: $T_\infty(N) = T_\infty(\frac{N}{2}) + \Theta(1)$, gdzie $T_\infty(1) = \Theta(1)$. Rozwijając rekursję dostajemy $T_\infty(N) = \Theta(\log N)$.

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