Adatszerkezetek és algoritmusok

Horváth Géza

tizenegyedik előadás

Előadások témái

1. Az algoritmusokkal kapcsolatos alapfogalmak bevezetése egyszerű példákon keresztül.
2. Az algoritmusok futási idejének aszimptotikus korlátai.
3. Az adatszerkezetekkel kapcsolatos alapfogalmak. A halmaz, a multihalmaz és a tömb adatszerkezet bemutatása.
4. Az adatszerkezetek folytonos és szétszórt reprezentációja. A verem, a sor és a lista.
5. Táblázatok, önátrendező táblázatok, hash függvények és hash táblák, ütközéskezelés.
6. Fák, bináris fák, bináris keresőfák, bejárás, keresés, beszúrás, törlés.
7. Kiegyensúlyozott bináris keresőfák: AVL fák.
8. Piros-fekete fák.
9. B-fák.
10. Gráfok, bejárás, legrövidebb út megkeresése.
11. Párhuzamos algoritmusok.
12. Eldönthetőség és bonyolultság, a P és az NP problémaosztályok.
13. Lineáris idejű rendezés. Összefoglalás.

Párhuzamos algoritmusok – bevezetés

A **párhuzamos algoritmusok** végrehajtásához szükségesek **párhuzamos számítások,** melyeket olyan számítógépeken tudunk végrehajtani, melyek több processzor maggal rendelkeznek. Ezek a számítógépek manapság már igen elterjedtek, a 4 magos, a 6 magos, a 10 magos, sőt, a 18 magos processzorok is a mindennapok részei. Ezen számítógépek esetén minden processzor mag egy **közös memórián** osztozik.

A jelenlegi **többmagos processzorok** használatának következtében, a jelenleg kapható laptopok és asztali számítógépek mindegyike közös memórián osztozó, de párhuzamos számításokat végző eszközzé vált.

Párhuzamos algoritmusok – bevezetés

Az eddig tanult algoritmusaink mindegyike **szekvenciális algoritmus** volt, melyeket egymagos processzort használó számítógépeken történő futásra dolgoztak ki, így egy adott lépésben egyetlen egy műveletet végeztek.

Ma olyan **párhuzamos algoritmusokat** fogunk használni, melyek többmagos processzorok segítségével valósítanak meg párhuzamos számításokat, ezzel jelentősen csökkentve a futási időt. Megismerkedünk továbbá a dinamikus többszálú algritmusok – dynamic multithreaded algorithms – gyakorlati használatával is.

Párhuzamos algoritmusok – bevezetés

A dinamikus többszálúság segítségével a programozók úgy készíthetnek párhuzamosan működő programokat, hogy nem kell aggódniuk a kommunikációs protokollok miatt. A párhuzamos felület tartalmaz egy ütemezőt, mely automatikusan összehangolja a számításokat, ezzel nagymértékben megkönnyíti a programozói munkát. Annak ellenére, hogy a dinamikus többszálú programozást támogató felületek jelenleg is folyamatosan fejlődnek, szinte mindegyik támogat két megoldást:

1. a **beágyazott párhuzamosságot** (nested parallelism), és
2. a **párhuzamos ciklusokat** (parallel loops).

Párhuzamos algoritmusok – bevezetés

A beágyazott párhuzamosság használata esetén lehetőség van alprogramok – függvények, eljárások – meghívására és futtatására, miközben a főprogram is tovább fut. Így egyidőben hajtódik végre több programrész.

Párhuzamos ciklus esetén a ciklus minden egyes lépése egyszerre hajtódik végre, egyidőben számolódnak ki az értékek a ciklusváltozó minden egyes értékére.

Új műveletek

A dinamikus többszálúság gyakorlati használatakor egyszerűen plusz utasításokat vezetünk be a szekvenciális programunkba. A pszeudokódot kiegészítjük az alábbi utasításokkal:

- **parallel**
- **spawn**
- **sync**

Ha elhagyjuk a fentebbi párhuzamos utasításokat az adott pszeudokódból, akkor minden esetben egy szekvenciális kódot kapunk, mely ugyanazt a problémát beágyazott párhuzamosság használata nélkül oldja meg.

Fibonacci számok

Fibonacci számok:

$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,.
\end{aligned}
$$

Fibonacci számok – szekvenciális rekurzív algoritmus

```
FIB(n)
1  if n ≤ 1
2        return n
3  else x = FIB(n − 1)
4       y = FIB(n − 2)
5       return x + y
```

Fibonacci számok:

$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,.
\end{aligned}
$$

Fibonacci számok – lépésszám

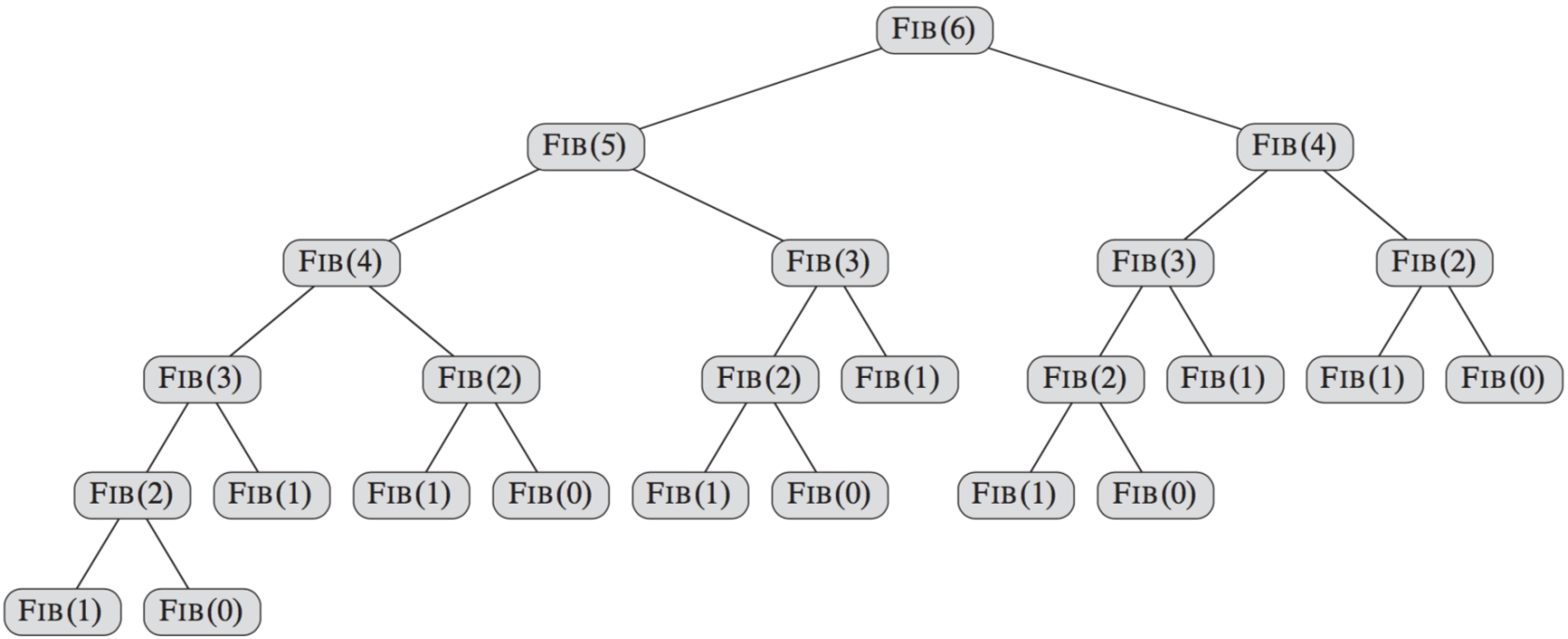


Figure 27.1 The tree of recursive procedure instances when computing FIB(6). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

Fibonacci számok – lépésszám

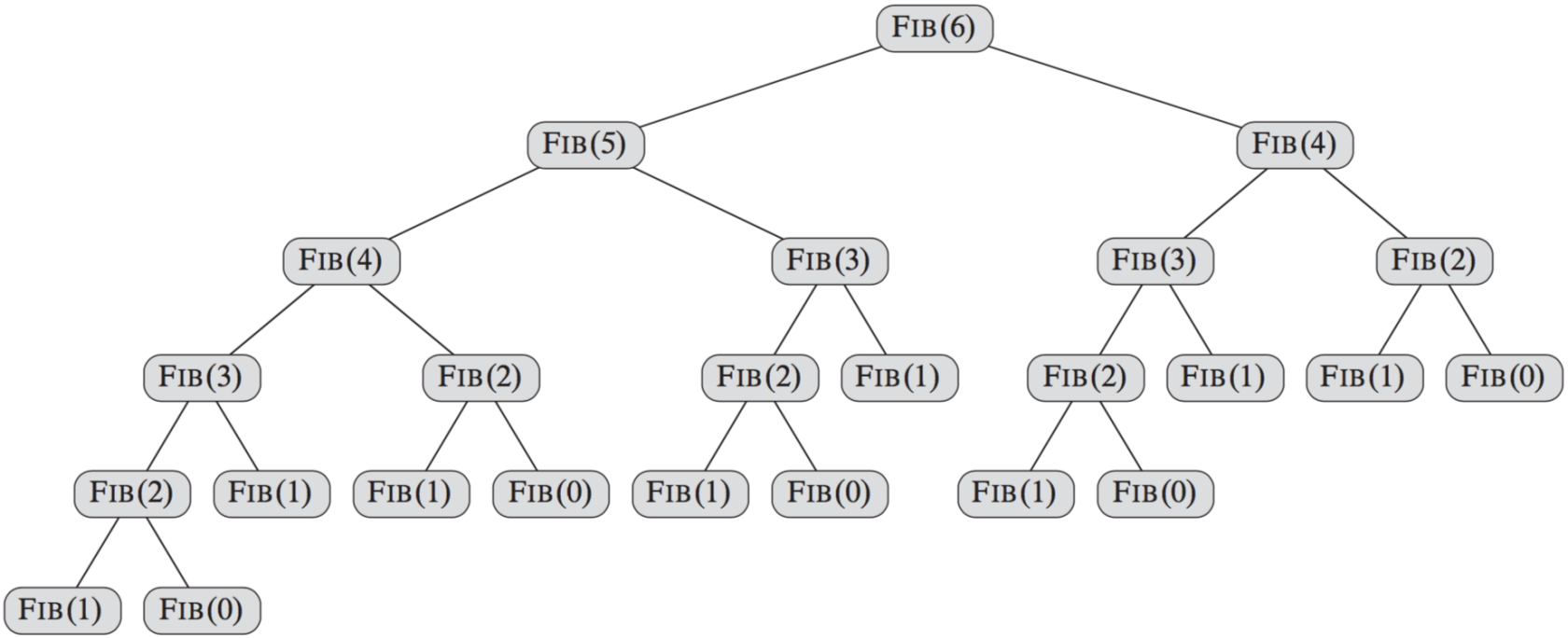


Figure 27.1 The tree of recursive procedure instances when computing FIB(6). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

$$T(n) = \Theta(\phi^n) ,$$

where $\phi = (1 + \sqrt{5})/2$ is the golden ratio

Fibonacci számok – beágyazott párhuzamos algoritmus

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4       y = P-FIB(n − 2)
5       sync
6       return x + y
```

Fibonacci számok:

$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,.
\end{aligned}
$$

A beágyazott párhuzamosság a 3. pontban kezdődik. A spawn utasítást követő függvényhívás egy külön szálon hajtódik végre, míg az eredeti (szülő) program tovább fut, ellentétben a szekvenciális esettel, amikor a (szülő) program futása megáll arra az időre, amíg meg nem kapja a (gyerek) függvény visszatérési értékét.
Jelen esetben tehát a program harmadik és negyedik sora párhuzamosan fut, egy időben számolja ki a P-FIB(n-1) és a P-FIB(n-2) értékeket.

Fibonacci numbers – beágyazott párhuzamos algoritmus

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4       y = P-FIB(n − 2)
5       sync
6       return x + y
```

Fibonacci számok:

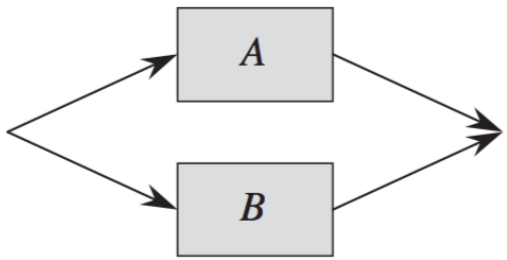
$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,.
\end{aligned}
$$

A főprogram addig nem használhatja a párhuzamosan meghívott függvény visszatérési értékét (x) biztonsággal, amíg el nem jut az 5. sorban a sync utasításhoz.
A sync utasítás hatására a szülő program "összevárja" amíg az összes olyan gyerek program futása befejeződik, melyeket a szülő program indított el, és csak ezután folytatja a futását. Ekkor már biztonságosan folytatható a program, mivel minden párhuzamos számítás befejeződött, minden visszatérési érték kiszámításra került.

Többszálú algoritmusok – munka és idő

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

Figure 27.3 The work and span of composed subcomputations. **(a)** When two subcomputations are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. **(b)** When two subcomputations are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

Többszálú algoritmusok – futási idő

The actual running time of a multithreaded computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to processors. To denote the running time of a multithreaded computation on P processors, we shall subscript by P. For example, we might denote the running time of an algorithm on P processors by T_P. The work is the running time on a single processor, or T_1. The span is the running time if we could run each strand on its own processor—in other words, if we had an unlimited number of processors—and so we denote the span by T_∞.

express the span of P-FIB(n) as the recurrence

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) , \end{aligned}$$

which has solution $T_\infty(n) = \Theta(n)$

Párhuzamos ciklusok

As an example, consider the problem of multiplying an $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. The resulting n-vector $y = (y_i)$ is given by the equation

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \ ,$$

for $i = 1, 2, \ldots, n$. We can perform matrix-vector multiplication by computing all the entries of y in parallel as follows:

MAT-VEC(A, x)

```
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4       y_i = 0
5  parallel for i = 1 to n
6       for j = 1 to n
7            y_i = y_i + a_ij x_j
8  return y
```

Hozzáférési verseny

Definíció

Hozzáférési verseny (determinacy race) alakul ki abban az esetben, amikor több, párhuzamosan futó szálon ugyanahhoz a memóriarészhez próbálunk meg hozzáférni úgy, hogy legalább egy szál írni próbálja az adott memóriarészt.

Példa:

```
RACE-EXAMPLE()
1  x = 0
2  parallel for i = 1 to 2
3       x = x + 1
4  print x
```

Hozzáférési verseny

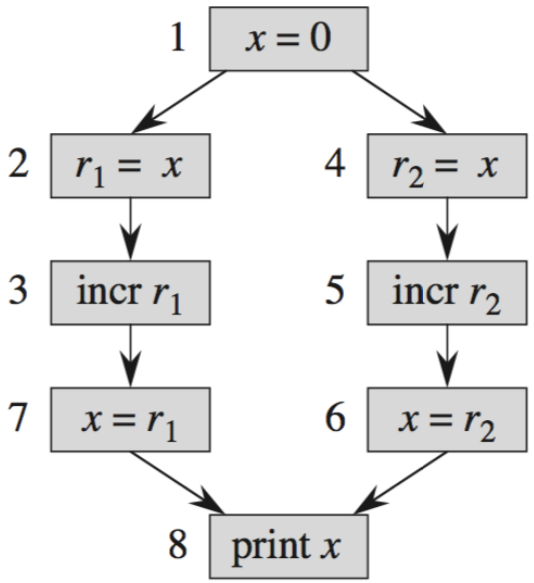
```
RACE-EXAMPLE( )
1   x = 0
2   parallel for i = 1 to 2
3       x = x + 1
4   print x
```

After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that RACE-EXAMPLE should always print the value 2 (its serialization certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

When a processor increments x, the operation is not indivisible, but is composed of a sequence of instructions:

1. Read x from memory into one of the processor's registers.
2. Increment the value in the register.
3. Write the value in the register back into x in memory.

Hozzáférési verseny



(a)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(b)

Figure 27.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A computation dag showing the dependencies among individual instructions. The processor registers are r_1 and r_2. Instructions unrelated to the race, such as the implementation of loop control, are omitted. **(b)** An execution sequence that elicits the bug, showing the values of x in memory and registers r_1 and r_2 for each step in the execution sequence.

Hozzáférési verseny

MAT-VEC-WRONG(A, x)

```
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4        y_i = 0
5  parallel for i = 1 to n
6        parallel for j = 1 to n
7              y_i = y_i + a_ij x_j
8  return y
```

This procedure is, unfortunately, incorrect due to races on updating y_i in line 7, which executes concurrently for all n values of j.

Többszálú mátrix szorzás

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  parallel for i = 1 to n
4      parallel for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Eredeti összefésüléses rendezés

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Többszálú összefésüléses rendezés

```
MERGE-SORT'(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Többszálú MERGE használatával további gyorsulás érhető el.

Irodalomjegyzék