Parallelprogrammierung – Parallele Algorithmen

Programmierparadigmen

Johannes Brauer

21. September 2020

Ziele

- Einstieg in das Thema parallele Algorithmen gewinnen
- Beispiele paralleler Programme kennenlernen
- Beurteilen möglicher Geschwindigkeitsgewinne durch Parallelisierung von Berechnungen

Grundlagen

Parallelität jenseits von Nebenläufigkeit

- Bisher stand die Beherrschung der Probleme, die bei nebenläufigen Prozessen auftreten, im Vordergrund.
- Jetzt wird der Fokus auf die Parallelprogrammierung gelegt.
- Gründe (teilweise schon genannt):
  - Jeder Arbeitsplatzrechner ist heute ein Parallelrechner.
    - Wie können die sich daraus ergebenden Möglichkeiten genutzt werden?
  - Arbeitsplatzrechner verfügen meist auch über Grafikprozessoren (graphics-processing units, GPUs), die mithilfe spezieller Bibliotheken auch für Nicht-Grafik-Anwendungen genutzt werden können.
- Parallele Algorithmen können sowohl gemeinsamen, geteilten Speicher oder auch verteilten Speicher nutzen.
  - Die entscheidende Frage lautet: Wie müssen Programme geschrieben werden, damit sie parallele Algorithmen nutzen können?
  - Unterstützung durch Werkzeuge (Bibliotheken, Compiler etc.) ist dabei hilfreich oder sogar notwendig.

Einstiegsbeispiel: Parallelisierung von `map`

(vgl. u. a. [Wal16])

- bekannte Standardfunktion höherer Ordung, wie sie in jeder funktionalen Programmiersprache zur Verfügung steht
- `map` wendet eine Funktion auf jedes Element einer Sequenz (Collection) an.
- idealer Kandidat für Parallelisierung
- Den Ablauf eines Algorithmus zu parallelisieren verursacht immer auch Verwaltungsaufwand (overhead).
- Deswegen lohnt die Parallelisierung von `map` in der Regel nur, wenn die auf die Elemente der Sequenz anzuwendende Funktion eine zeitintensive Berechnung ist.

Clojure `pmap`

- Mit `(pmap f coll)` wird `pmap` genauso aufgerufen wie `map`:

```
user> (pmap inc [1 2 3 4 5])
;; => (2 3 4 5 6)
```

- „Simulation“ einer zeitaufwändigen Berechnung:

```
(defn square-slowly [x]
  (Thread/sleep 2000)
  (* x x))
```

- Anwendung von `square-slowly` mit `map`

```
user> (time (doall (map square-slowly (repeat 3 10))))
"Elapsed time: 6017.028608 msecs"
;; => (100 100 100)
```

- Der Aufruf von `doall` erzwingt, dass die standardmäßig von `map` erzeugte *lazy sequence* auch materialisiert wird.
- Mit der Verwendung von `pmap` wird die Laufzeit gedrittelt:

```
user> (time (doall (pmap square-slowly (repeat 3 10))))
"Elapsed time: 2004.683838 msecs"
;; => (100 100 100)
```

Implementierung von pmap

```
(defn pmap
  "Like map, except f is applied in parallel. Semi-lazy in that the
  parallel computation stays ahead of the consumption, but doesn't
  realize the entire result unless required. Only useful for
  computationally intensive functions where the time of f dominates
  the coordination overhead."
  {:added "1.0"
   :static true}
  ([f coll]
   (let [n (+ 2 (.. Runtime getRuntime availableProcessors))
         rets (map #(future (f %)) coll)
         step (fn step [[x & xs :as vs] fs]
                (lazy-seq
                 (if-let [s (seq fs)]
                   (cons (deref x) (step xs (rest s)))
                   (map deref vs))))]
     (step rets (drop n rets))))
  ([f coll & colls]
   (let [step (fn step [cs]
                (lazy-seq
                 (let [ss (map seq cs)]
                   (when (every? identity ss)
                     (cons (map first ss) (step (map rest ss)))))))]
     (pmap #(apply f %) (step (cons coll colls))))))
```

- `pmap` verwendet einen Thread-pool der Größe

```
(.. Runtime getRuntime availableProcessors).
```

- Threads werden „auf Vorrat“ erzeugt, bleiben am Leben und können wiederverwendet werden.

Explizite Parallelisierung

- Neben der Funktion `pmap`, die die Parallelisierung implizit vornimmt, kennt Clojure noch zwei weitere Funktionen:

  pvalues wertet die Ausdrücke, die als Argumente übergeben werden, parallel aus

  pcall erwartet Thunks (parameterlose Funktionen) als Argumente und führt sie parallel aus.

```
user> (time (doall (pvalues
                     (square-slowly 10)
                     (square-slowly 10)
                     (square-slowly 10))))
"Elapsed time: 2006.196172 msecs"
;; => (100 100 100)
user> (time (doall (pcalls #(square-slowly 10)
                           #(square-slowly 10)
                           #(square-slowly 10))))
"Elapsed time: 2002.614955 msecs"
;; => (100 100 100)
```

Parallelität und Performance

- Die bisherigen Beispiele paralleler Berechnungen waren sehr einfach bzw. naheliegend (mapping).
- Soll z. B. die Summe einer Liste von Zahlen parallelisiert werden, braucht man eine weiter gehende Strategie.
- Will man eine Berechnung parallelisieren, müssen zwei Fragen beantwortet werden:
  1. Wie sieht eine geeignete Parallelisierungsstragtegie aus bzw. gibt es überhaupt eine?
  2. Welche Geschwindigkeitsgewinne können erzielt werden?

Beispiel: Das Java-Programm `ParallelSort`

Quellcode von ParallelSort.java

```
/** A Java program to illustrate the potential speedup of parallelism, in this
    case for sorting: the program uses the new (as of Java 8) Arrays.parallelSort
    method, and also the Arrays.parallelSetAll method for parallel initialization
    of the roughly 4M array of integers.

    As noted, Java 8 or greater is requied.
*/
import java.util.Random;
import java.util.Arrays;

public class ParallelSort {
    private int[ ] array2Sort;

    public static void main(String[ ] args) {
        new ParallelSort().doIt();
    }
    private void doIt() {
        final int arraySize = 1024 * 1024 * 4;  //about 4M: 4,194,304  /** line 1 **/

        //** serial sort
        init(arraySize);
```

```
        long serial = sortArray(array2Sort, false);                    /** line 2 **/

        //** parallel sort
        init(arraySize);
        long parallel = sortArray(array2Sort, true);                   /** line 3 **/

        //** report
        log("Serial sort of array of size:   " + arraySize);
        log("Elapsed time: " + serial);
        log("");

        log("Parallel sort of array of size: " + arraySize);
        log("Elapsed time: " + parallel);

        log("");
        double ratio = (double) serial / (double) parallel;
        if (parallel < serial)
            log("Speedup:  " + ratio);
        else
            log("Slowdown: " + ratio);
    }
    private void init(int size) {
        array2Sort = new int[size];
        /** parallel initialization of array elements **/
        Arrays.parallelSetAll(array2Sort,                              /** line 4 **/
                              i->new Random().nextInt());              /** line 5 **/
    }
    private long sortArray(int[ ] array, boolean parallel) {
        long start = System.currentTimeMillis();
        /** parallel version of merge-sort **/
        if (parallel) Arrays.parallelSort(array);                      /** line 6 **/
        /** variant of quicksort ('dual pivot') **/
        else Arrays.sort(array);                                       /** line 7 **/
        long stop = System.currentTimeMillis();
        return stop - start;
    }
    private void log(String msg) {
        System.out.println(msg);
    }
}
```

Benutzung von der Kommandozeile

```
bash-3.2$ java ParallelSort
Serial sort of array of size:   4194304
Elapsed time: 824

Parallel sort of array of size: 4194304
Elapsed time: 358

Speedup:  2.3016759776536313
bash-3.2$ java ParallelSort
```

```
Serial sort of array of size:   4194304
Elapsed time: 710

Parallel sort of array of size: 4194304
Elapsed time: 258

Speedup:  2.751937984496124
bash-3.2$ java ParallelSort
Serial sort of array of size:   4194304
Elapsed time: 807

Parallel sort of array of size: 4194304
Elapsed time: 295

Speedup:  2.7355932203389832
```

Amdahls Gesetz (erläutert an einem Beispiel)

- Das Gesetz, benannt nach einem früheren IBM-Ingenieur, der später eine eigene Großrechnerfirma gründete, beschreibt die theoretische Grenze für die mögliche Geschwindigkeitsgewinnen durch parallele Berechnungen.
  - Das Gesetz kann die Intuition über Geschwindigkeitssteigerung schärfen.
- Beispielproblem:
  - Berechne das arithmetische Mittel von 4M Gleitkommazahlen auf einer Einprozessormaschine *M1*; das Problem heiße *P*.
  - Berechne die Geschwindigkeitssteigerung, die für P mit einer Vierprozessormaschine *M4* erzielt werden kann.
  - Annahme: *M1* und *M4* unterscheiden sich nur in der Anzahl der Prozessoren.

Schrittweise Anwendung des Gesetzes:

1. Berechne die Antwortzeit für P auf *M1*: $RT1$
2. Berechne die Antwortzeit für P auf *M4*: $RT4$
3. Berechne das Verhältnis: $\frac{RT1}{RT4}$
   - Wenn $\frac{RT1}{RT4} = 1$, keine Geschwindigkeitssteigerung
   - Wenn $\frac{RT1}{RT4} < 1$, Verlangsamung
   - Wenn $\frac{RT1}{RT4} > 1$, Geschwindigkeitssteigerung.
4. Normalisiere der Einfachheit halber $RT1$ zu 1 Zeiteinheit.
5. Spalte $RT4$ in zwei Teile auf:
   - *optimiert*: hier parallelisiert
   - *unoptimiert*: hier seriell
   - Der Teil *unoptimiert* von $RT4$ umfasst Aktivitäten, die seriell ablaufen müssen.
     - Die Datenmenge muss partitioniert und an Arbeiter (Threads oder Prozesse) verteilt werden; die Ergebnisse der Arbeiter müssen zusammen gefasst werden ...
   - Der Teil *optimiert* von $RT4$ profitiert von der Parallelisierung.

- –

$$Speedup = \frac{RT1}{RT4} = \frac{1}{unoptimiert + optimiert}$$

  - Die Größe *optimiert* hängt von zwei Faktoren ab:

1. Welchen Anteil der Zeit arbeitet *M4* im Parallelmodus bei der Lösung von P?
2. Wieviel schneller ist *M4* im Parallelmodus als *M1* im Seriellmodus?

– Annahmen:

1. *M4* ist im Parallelmodus viermal schneller als *M1*.
2. *M4* arbeitet 80% der Zeit im Parallelmodus bei der Lösung von P.

–

$$Zeitgewinn = \frac{RT1}{RT4} = \frac{1}{unoptimiert + optimiert} = \frac{1}{0.2 + \frac{0,8}{4,0}} = 2,5$$

– Unter den gegebenen Bedingungen ist *M4* um den Faktor $2,5$ schneller als *M1*.

Fazit

- Amdahls Gesetz verdeutlicht, dass naive Annahmen über mögliche Geschwindigkeitssteigerungen bei der Parallelisierung von Berechnungen verfehlt sind.
- Zahlenbeispiele: (aus [Kal15])

Machine	Fraction in local speedup	Local speedup	Global speedup
M4	0.8	4.0	2.5
M8	0.7	8.0	2.6
M12	0.6	12.0	**2.2**
M16	0.5	16.0	1.9
M20	0.4	20.0	1.6
M24	0.3	24.0	1.4

Empfehlungen

Parallelprogrammierung mit Scala

- Coursera MOOC: Parallel programming (Scala) (kostenlos)
- Themen (u. a.):
  - Das `parallel`-Konstrukt von Scala
  - Das `task`-Konstrukt von Scala
  - Mess-Methodik für die Laufzeit parallelisierter Berechnungen
  - ScalaMeter
  - Datenparallelität
  - Parallelisierung von fold/reduce
  - Datenstrukturen für parallele Berechnungen

Clojure

Parallelität mit Reducers

Zum Beispiel in

- Reducers - A Library and Model for Collection Processing
- PARALLEL PROGRAMMING in CLOJURE with REDUCERS

Komunikation mit core.async

Zum Beispiel in

- Clojure core.async Channels
- Mastering Concurrent Processes with core.async

Literaturverzeichnis

Literatur

[Kal15] Martin Kalin. Concurrent and parallel programming concepts, 2015. zuletzt aufgerufen am 10.10.2017.

[Wal16] Akhil Wali. *Mastering Clojure : understand the philosophy of the Clojure language and dive into its inner workings to unlock its advanced features, methodologies, and constructs*. Packt Publishing, Birmingham, UK, 2016.