Prof. Dr. Thomas Schmidt
HAW Hamburg, Dept. Informatik,
Raum 780, Tel.: 42875 - 8452
Email: schmidt@informatik.haw-hamburg.de
Web: http://inet.cpt.haw-hamburg.de/teaching
/ws-2011-12/verteilte-systeme

Verteilte Systeme

Aufgabe 2: Verteilte Primzahlfaktorisierung im Aktor-Modell

Ziele:

1. Message Passing im Aktor-Modell kennenlernen
2. Verteilungsszenarium für ein nebenläufiges Problem *begründet* Entwerfen
3. Konzipiertes Szenarium mittels asynchroner Nachrichten implementieren
4. Erzieltes Ergebnis mittels Performanzmessung evaluieren

Vorbemerkungen:

In dieser Aufgabe betrachten wir ein lose gekoppeltes Problem verteilter Rechenlast, dass im sog. Aktor-Modell (s. u.) gelöst werden soll. Dabei tauschen Worker (Aktoren) Nachrichten zur Koordinierung aus, um gemeinsam im Wettbewerb das gegebene Problem zu lösen.

Ihre Aufgabe ist es zunächst, ein *durchdachtes Konzep*t zu erstellen, in dem die Aufgabenverteilung und die Kommunikationsschritte passend zum Problem gewählt werden. Messen Sie die Qualität Ihrer Lösungsideen an den Qualitätseigenschaften verteilter Systeme und benutzen Sie diese Kriterien in der *Konzeptbegründung*. Diskutieren Sie insbesondere das *Skalierungsverhalten* und die *Fehlertoleranz*. Evaluieren Sie die tatsächliche Leistungsfähigkeit Ihrer Lösung mithilfe einer verteilten Laufzeitmessung.

Problemstellung:

Die Primfaktorenzerlegung großer Zahlen ist eines der numerisch "harten" Probleme. Public Key Security Verfahren (RSA) leiten z.B. ihre Schlüsselsicherheit davon ab, dass eine öffentlich bekannte große Zahl nicht in der notwendigen Zeit in ihre (unbekannten) Primfaktoren zerlegt werden kann. Aus umgekehrter Sicht ist es von Interesse, Rechenverfahren zu entwerfen, mit welchen die Primfaktorisierung möglichst beschleunigt werden kann.

Wie Sie sich leicht überlegen können, hat das naive Ausprobieren aller infrage kommenden Teiler einen Rechenaufwand von $\mathcal{O}(\sqrt{\mathcal{N}})$, wenn $\mathcal{N}$ die zu faktorisierende Zahl ist. Der nachfolgende Algorithmus, welchen wir verteilt implementieren wollen, geht auf Pollard zurück und findet einen Primfaktor p im Mittel nach $1{,}18\sqrt{p}$ Schritten. Seine zugrundeliegende Idee ist die des 'Geburtstagsproblems': Wie Sie mit einfachen Mitteln nachrechnen können, ist die Wahrscheinlichkeit überraschend groß, auf einer Party zufällig eine Person zu treffen, die am gleichen Tag Geburtstag hat wie Sie. [Randbemerkung: Pikanterweise ist gerade das Nichtbeachten dieses Geburtstagsproblems der Grund für die kryptographische Schwäche der WLAN Verschlüsselung WEP.]

Die Pollard Rho Methode zur Faktorisierung:

Die Rho Methode ist ein stochastischer Algorithmus, welcher nach zufälliger Zeit, aber zuverlässig Faktoren einer gegebenen *ungeraden* Zahl $\mathcal{N}$ aufspürt. Hierzu wird zunächst eine Pseudo-Zufallssequenz von Zahlen $x_i \leq \mathcal{N}$ erzeugt:

$$x_{i+1} = x_i^2 + a \bmod \mathcal{N},\ a \neq 0, -2 \text{ beliebig.}$$

Gesucht werden nun die Perioden der Sequenz x_i, also ein Index p, so dass $x_{i+p} = x_i$. p ist dann ein Teiler von $\mathcal{N}$.

Solche Zyklenlängen p lassen sich leicht mithilfe von Floyd's Zyklenfindungsalgorithmus aufspüren:

Berechne $d = (x_{2i} - x_i) \bmod \mathcal{N}$, dann ist

$p = GGT(d, \mathcal{N})$, wobei GGT der größte gemeinsame Teiler ist.

Im **Metacode** sieht der Algorithmus von Pollard wie folgt aus:

rho (N,a) { N = zu faktorisierende Zahl; a = (worker-basiertes) Inkrement der Zufallssequenz;}

```
x = rand(1 … N);
y = x;
p = 1;
Repeat
        x = (x^2 + a) mod N;
        y = (y^2 + a) mod N;
        y = (y^2 + a) mod N;
        d = (y  -  x) mod N;
        p = ggt (d, N);
until (p != 1);

if (p != N) then factor found: p
```

Hinweise: Die Rho-Methode findet nicht nur Primfaktoren, sondern manchmal auch das Produkt von mehreren Primfaktoren - **deshalb muss ein einmal gefundener Faktor noch 'weiterbearbeitet' werden**. Gefundene Faktoren können N zudem auch mehrfach teilen! Wenn die Rho-Methode terminiert, ohne einen echten Faktor gefunden zu haben (p = N), dann ist das untersuchte N entweder unteilbar, oder N wurde als Produkt seiner Primfaktoren entdeckt. Den erstgenannten Fall können Sie über einen Primalitätstest ausschließen, im letztgenannten Fall muss die Faktorisierung mit einer veränderten Zufallssequenz (Startwert und a) erneut durchgeführt werden.
Da es sich um ein zufallsgesteuertes Verfahren mit zufälliger Laufzeit handelt, können zu dem ungewöhnlich hohe Laufzeiten auftreten, ohne dass ein Faktor gefunden wird. Implementieren Sie ggf. eine Abbruchbedingung für die obige Schleife, nach welcher Sie den Algorithmus neu starten.

Das Aktor-Modell

Aktoren sind nebenläufige, unabhängige Softwarekomponenten, die keine gemeinsame Sicht einen Speicherbereich haben. Sie kommunizieren durch asynchronen Nachrichtenaustausch miteinander und können zu ihrer Laufzeit weitere Aktoren erschaffen. Da das Programmiermodell keine gemeinsame Sicht auf einen Speicherbereich vorsieht, werden zum einen Race Conditions ausgeschlossen und zum anderen eignet sich das Aktor-Modell auch zur Programmierung von im Netzwerk verteilter Anwendungen.

In der *Java-Bibliothek Akka*, die sich grob am Aktor-Modell orientiert, sind Aktoren eventbasierte Software-Komponenten, die von der Klassen UntypedActor erben. Empfängt ein Aktor eine Nachricht, wird die onReceive Methode des Aktors aufgerufen. Um netzwerktransparent programmieren zu können, müssen alle Aktoren explizit über einen Remote Server gestartet werden und eindeutige IDs vergeben werden. Das folgende Programm ist ein Beispiel für eine im Netzwerk verteilte Anwendung, basierend auf Akka:

Programmierbeispiel: Aktoren mit Akka

Der Master vergibt eine Berechnungsaufgabe (CalculateMessage) an einen Worker weiter und empfängt dessen berechnetes Ergebnis. Um den Master zu starten, wird in der Main-Methode des Masters eine CalculateMessage erstellt und an eine neue Instanz des Master-Aktors geschickt. Weitere Erklärungen finden Sie in den Quellcode-Kommentaren.

<u>Der Master:</u>

```java
package de.haw.inet.vs.lab2;
import static akka.actor.Actors.poisonPill;
import static akka.actor.Actors.remote;
import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.remoteinterface.RemoteServerModule;

public class Master extends UntypedActor {
	static int resultsReceived = 0;
	static int numberOfWorker;
	static RemoteServerModule remoteSupport;

	@Override
	public void onReceive(Object message) throws Exception {
		if (message instanceof CalculateMessage) {
			CalculateMessage calculate = (CalculateMessage) message;
			// Worker auf dem Remote-Host erstellen
			ActorRef worker = remote().actorFor(Worker.class.getName(),
					"workerserver", 2552);
			// getContext() gibt eine Referenz auf diesen Aktor zurück
			ActorRef me = getContext();
			// tell verschickt eine Message an einen Aktor. Zusätzlich kann
			// man eine Referenz auf einen anderen Aktor übergeben
			worker.tell(calculate, me);

		} else if (message instanceof ResultMessage) {
			System.out.println(((ResultMessage) message).getResult());
			getContext().tell(poisonPill());
		} else {
			throw new IllegalArgumentException("Unknown message [" +
					message + "]");
		}
	}

	public static void main(String[] args) {
		// Der "Client" muss auch als Remote-Aktor gestartet werden um
		// später Nachrichten vom Server empfangen zu können.
		remoteSupport = remote().start("localhost", 2553);
		ActorRef client = remote().actorFor(Master.class.getName(),
				"localhost", 2553);
		CalculateMessage calculate = new CalculateMessage(10, 10);
		client.tell(calculate);
	}
}
```

Der Worker:

```java
package de.haw.inet.vs.lab2;

import static akka.actor.Actors.poisonPill;
import static akka.actor.Actors.remote;
import akka.actor.ActorRef;
import akka.actor.UntypedActor;

public class Worker extends UntypedActor {
	private static int idGenerator = 0;
	private int actorId;

	public Worker() {
```

```
            // Wichtig: Wenn die ID nicht gesetzt wird, wird immer dieselbe In-
            // stanz des Aktors für alle Remote-Aufrufe eines Clients verwendet!
            getContext().setId(idGenerator + "");
            actorId = idGenerator;
            System.out.println("Aktor wurde erstellt: " + idGenerator);
            idGenerator++;
      }

      private ActorRef master;

      private Integer calculate(int a, int b) {
            return new Integer(a + b);
      }

      // message handler
      public void onReceive(Object message) {
            if (message instanceof CalculateMessage) {
                  // Beim ersten Aufruf wird der Sender ermittel
                  this.master = getContext().getSender().get();
                  CalculateMessage calculateMessage = (CalculateMessage) message;

                  Integer result = calculate(calculateMessage.getA(),
                                             calculateMessage.getB());
                  ResultMessage resultMessage = new ResultMessage(result);

                  // Ergebnis an den Master senden
                  master.tell(resultMessage);
                  // Durch this.getContext().tell([Nachricht]) kann der Aktor
                  // sich selbst eine Nachricht schicken. In diesem Fall schickt
                  // sich der Aktor eine "poisonPill". Empfängt ein Aktor diese,
                  // beendet er sich und postStop() wird aufgerufen.
                  getContext().tell(poisonPill());

            } else {
                  throw new IllegalArgumentException("Unknown message [" +
                              message + "]");
            }
      }

      @Override
      public void postStop() {
            System.out.println("Aktor wurde beendet: " + this.actorId);
            super.postStop();
      }

      public static void main(String[] args) throws Exception {
            remote().start("workerserver", 2552);
      }

}
```

Die Calculate-Message:

```
package de.haw.inet.vs.lab2;
```

```
import java.io.Serializable;

public class CalculateMessage implements Serializable {

	private static final long serialVersionUID = 840244832287440949L;
	private int b;
	private int a;

	public CalculateMessage( int a, int b) {
		this.b = b;
		this.a = a;
	}

	public int getB() {
		return b;
	}

	public int getA() {
		return a;
	}

}
```

Die Result-Message:

```
package de.haw.inet.vs.lab2;

import java.io.Serializable;

public class ResultMessage implements Serializable {


	private static final long serialVersionUID = -6065578273626197783L;
	public ResultMessage(Integer result){
		this.result = result;
	}

	public Integer getResult(){
		return this.result;
	}
	private Integer result;

}
```

Um das Beispiel kompilieren und ausführen zu können, benötigen Sie die Akka-Library in der Version 1.2 (http://akka.io/downloads/akka-microkernel-1.2.zip). Sie müssen für Master und Worker jeweils ein Eclipse Projekt anlegen. Das Master-Projekt enthält nur die Master-Klasse und referenziert das Worker-Projekt. Das Worker-Projekt enthält die Worker-Klasse und die Messages. Weiterhin müssen in beiden Projekten die scala-library.jar aus dem „lib/“-Verzeichnisses und alle Jars aus dem „lib/akka“ –Verzeichnis von Akka zum Projekt hinzugefügt werden.

Aufgabenstellung

Teilaufgabe 1:

Konzipieren Sie ein Verteilungs- und Kommunikationsszenario im Aktor-Modell, in welchem die Rho-Methode auf nebenläufigen Workern 'im Wettbewerb' abgearbeitet wird (mit unterschiedlichen Inkrementen a).

Hierzu benötigen Sie:

1. Einen Master mit User-Interface für Start und Auswertung, welcher die zu faktorisierende Zahl entgegennimmt, an die Worker (Aktoren) verteilt und das Ergebnis (= die vollständige Primfaktorzerlegung sowie (a) die *tatsächlich aufgewendete CPU-Zeit, (b) die Summe der Rho Zyklendurchläufe* und
   (c) die *verstrichene Zeit* vom ersten Versenden bis zum Erhalt des letzten Faktors) ausgibt.
2. Kommunizierende Worker (Aktoren), die
   a. auf entfernten Rechnern ebenfalls durch einen Master gestartet werden,
   b. die Pollardmethode auf ihnen übergebene Zahlen anwenden,
   c. selbst gefundene Faktoren zusammen mit der aufgewendeten CPU-Zeit mitteilen
   d. und *asynchron* auf durch andere gefundene Faktoren lauschen.

Legen Sie Ihr Vorgehen *begründet* **in einem kurzen Konzeptpapier dar.**

Teilaufgabe 2:

Implementieren Sie nun Ihre konzipierte Lösung mit

- Worker-Prozessen, die die Rho-Methode in BigInteger Arithmetik realisieren.
- einem Starter, der mit ggf. vorgegebenen Workern kommuniziert und am Ende das Ergebnis gemeinsam mit einer Leistungsstatistik (CPU-Zeiten, verstrichene Wall-Clock Zeiten) ausgibt;
- ggf. weiteren Komponenten aus Ihrem Konzept sowie dem Kommunikationsablauf.

Teilaufgabe 3:

Testen Sie Ihr Programm unter Verteilung auf unterschiedliche Rechner mit den Zahlen:

Z1 = 8806715679 = 3 * 29 * 29 * 71 * 211 * 233

Z2 = 9398726230209357241 = 443 * 503 * 997 * 1511 * 3541 * 7907

Z3 = 113704728156282448422617157521937400432081248304 7

Z4 = 100060210614380659647872297427366695090390611213179474545733865926684244698 50220767921123091739752435069697105 03

Analysieren Sie das Laufzeitverhalten Ihres Programmes: CPU-Zeit versus Wall-Clock Zeit, vergleichen Sie mit anderen Lösungen.