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-2009-10/verteilte-systeme

Verteilte Systeme

Aufgabe 2: Verteilte Primzahlfaktorisierung

Ziele:

1. Verteilungsszenarium für ein nebenläufiges Problem *begründet* Entwerfen
2. Konzipiertes Szenarium mittels asynchroner Kommunikation implementieren
3. Erzieltes Ergebnis mittels Performanzmessung evaluieren

Vorbemerkungen:

In dieser Aufgabe sind Sie weitgehend frei, für ein gegebenes Problem eine verteilte Lösung zu finden. Das Problem ist vielfältig lösbar – es kommt vor allem auf ein *durchdachtes Konzep*t an. Aufgabenverteilung und Kommunikationsanforderungen sollten zueinander passend gewählt werden, wobei Sie eine Abstimmung der Prozesse untereinander sowohl über Unicast oder Multicast Socket-Kommunikation, als auch über RMIs realisieren können. Messen Sie die Qualität Ihrer Lösungsideen an den Qualitätseigenschaften verteilter Systeme und benutzen Sie diese Kriterien in der *Konzeptbegründung*. 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

$p = \text{GGT}(d, \mathcal{N})$, 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.

Teilaufgabe 1:

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

Hierzu benötigen Sie:

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

Bestimmen Sie ein geeignetes verteiltes Programmiermodell für die Realisierung Ihres Konzepts. **Legen Sie dieses** *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 5022076792112309173975243506969710503

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