

Verteilte Systeme

Prof. Dr. Thomas Schmidt
HAW Hamburg, Dept. Informatik,
Raum 480a, Tel.: 42875 - 8452
Email: t.schmidt@haw-hamburg.de
Web: <http://inet.haw-hamburg.de/teaching>

Aufgabe 2: Verteilte Primzahlfaktorisation im Aktormodell

Ziele:

1. Message Passing im Aktormodell 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, das im sog. Aktormodell (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 Konzept* 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 Primfaktorisation möglichst beschleunigt werden kann.

Wie Sie sich leicht überlegen können, hat das naive Ausprobieren aller infrage kommenden Teiler einen Rechenaufwand von $\mathcal{O}(\sqrt{N})$, wenn 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 Zyklusfindungsalgorithmus 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 Pseudocode sieht der Algorithmus von Pollard wie folgt aus:

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

```
x = rand(1 ... N);
y = x;
p = 1;
Repeat
    x = (x2 + a) mod N;
    y = (y2 + a) mod N;
    y = (y2 + 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. Es ist deshalb günstig, verschiedene Worker gegeneinander im Wettbewerb rechnen zu lassen und regelmäßig Ergebnisse auszuwerten.

Konzipieren Sie den Programmablauf deshalb nach folgenden Regeln:

1. Ein Koordinator verteilt Arbeitsaufgaben an k verteilte Worker.
2. Eine Faktorisierung kann im Mittel bis zu $M = 1,18 \sqrt{\sqrt{N}}$ Schritte dauern, was für große Zahlen N sehr lange ist. Unterteilen Sie die Aufgaben deshalb in L Teile mit Iterationszahl M/L . Wenn ein Worker nach einem Teillauf keinen Faktor gefunden hat, soll er die aktuelle Berechnung unterbrechen und mit neuen Zufallszahlen fortfahren.
3. Sobald eine Lösung gefunden wurde verteilt der Koordinator das nächste Problem. Worker müssen daher regelmäßig ihre Mailbox prüfen und gegebenenfalls laufende Berechnungen unterbrechen. Hinweis: Sie können in CAF mit ``self->mailbox().empty()`` erfragen ob die Mailbox derzeit leer ist.

4. Entscheiden Sie wie viel Autonomie die Worker haben und begründen Sie ihren Entwurf. Beispielsweise kann ein Worker Zufallszahlen selbstständig generieren und autonom einen neuen Teillauf beginnen sobald sie eine bestimmte Anzahl an Rechenschritten erfolglos gerechnet haben. Alternativ kann der Koordinator sämtliche Entscheidungen zentral steuern und Worker erhalten Zufallszahlen und das aktuelle Problem, indem sie bei Überschreitung der maximalen Laufzeit pro Versuch eine Anfrage an den Koordinator schicken (z.B. mit dem gesamten Zustand bestehend aus x , y , a , Iterationszahl und Rechenzeit).

Das Aktormodell

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 Aktormodell auch zur Programmierung von im Netzwerk verteilter Anwendungen.

Aktoren in C++

Das „C++ Actor Framework“ (*CAF*) implementiert ein Aktorsystem, in dem ein Aktor sowohl funktions- als auch klassenbasiert implementiert werden kann. In *CAF* können Aktoren mithilfe der Funktion *publish* an einen Port gebunden werden. Andere Knoten im verteilten System können diesen Aktor dann mit der Funktion *remote_actor* erreichen.

Kurzanleitung und Grundgerüst für CAF

Für CAF benötigen Sie unter BSD, Linux und macOS GCC > 4.7 oder Clang > 3.5. Unter Windows benötigen Sie mindestens Microsoft Visual Studio 2015, Update 3. Auf allen Systemen benötigen Sie zusätzlich CMake.

Den Quellcode zu CAF können Sie entweder direkt per Git zugreifen oder Sie laden den letzten Release von der offiziellen Homepage als .zip Datei runter:

<https://github.com/actor-framework/actor-framework/releases>

Unter BSD, Linux und macOS können Sie mit den folgenden Schritten CAF sowie das Grundgerüst für die Aufgabe runterladen und kompilieren. Idealerweise führen Sie diese Schritte in einem Ordner für das Praktikum aus:

```
git clone https://github.com/actor-framework/actor-framework.git
cd actor-framework
./configure && make
cd ..
git clone https://github.com/inetrg/vslab-caf.git
cd vslab
mkdir build
cd build
cmake ..
make
```

Im Ordner vslab/src finden Sie die Quellcodedatei main.cpp mit einem Grundgerüst für die Praktikumsaufgabe: alle notwendigen Includes, ein beispielhaftes Command-Line-Interface, fertiger Setup für das Serialisieren von 512-bit Integern aus Boost, etc.

Bitte konsultieren Sie bei offenen Fragen:

- das Manual <https://actor-framework.readthedocs.io/en/latest/>
- die Beispielprogramme aus der Vorlesung: <https://github.com/inetrg/vs-cpp>
- die Beispielprogramme im Ordner actor-framework/examples
- und bei Unklarheiten rund um C++ <http://en.cppreference.com/w/>

Aufgabenstellung

Teilaufgabe 1:

Konzipieren Sie ein Verteilungs- und Kommunikationsszenario (s. Hinweise auf Seiten 3 u.4) im Ak-tormodell (es dürfen keine Threads gestartet werden!), in welchem die Rho-Methode auf nebenläufi-gen Workern 'im Wettbewerb' abgearbeitet wird (mit unterschiedlichen Inkrementen a).

Hierzu benötigen Sie:

1. Einen Koordinator, welcher die zu faktorisierte Zahl entgegennimmt, die Teilaufgaben er-stellt, an die Worker (Aktoren) verteilt und das Ergebnis (= die vollständige Primfaktorzerle-gung sowie (a) die *tatsächlich aufgewendete CPU-Zeit*, (b) die *Summe der Rho Zyklendurch-läufe* und (c) die *verstrichene Zeit* vom ersten Versenden bis zum Erhalt des letzten Faktors) ausgibt.
Der Koordinator bezieht zur Laufzeit hinzukommende oder ausfallende Worker dynamisch in die Arbeitsverteilung ein.
2. Einen Manager pro Rechnerinstanz, der eine konfigurierbare Anzahl von Workern startet, überwacht und dem Koordinator vermittelt.
3. Kommunizierende Worker (Aktoren), die
 - a. auf entfernten Rechnern durch den Koordinator gesteuert werden,
 - b. die Pollardmethode auf ihnen übergebene Zahlen anwenden,
 - c. selbst gefundene Faktoren bzw. Zwischenstände zusammen mit der aufgewendeten CPU-Zeit mitteilen,
 - d. auf einen Ausfall des Masters fehlertolerant reagieren.

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

Teilaufgabe 2:

Implementieren Sie nun Ihre konzipierte Lösung in C++ mit

- > Worker-Aktoren, die die Rho-Methode mit 512 bit Integer-Arithmetik (Boost) realisieren
- > einem Starter, der Aufgaben entgegennimmt, delegiert 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:

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

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

$$Z3 = 1137047281562824484226171575219374004320812483047$$

$$Z4 = 1000602106143806596478722974273666950903906112131794745457338659266842446985022076792112309173975243506969710503$$

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