

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 Primzahlfaktorierung 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 Primfaktorierung 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:

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

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 = zu faktorisierende Zahl, a = worker-basierter Zufall):  
  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 Primalitystest 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 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.

Algorithmische Lösungskonzeption:

Beachten Sie folgendes beim Konzipieren des Programmablaufs:

1. Ein Client verteilt Arbeitsaufgaben an die verfügbaren Worker, welche die Faktorisierung durchführen und den gefundenen Teiler an den Client zurückmelden.
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. Die Wahl eines passenden L ist Ihnen überlassen.
3. Sobald eine Lösung gefunden wurde, verteilt der Client 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. Alle Aktoren sind lose gekoppelt und kommunizieren über Publish/Subscribe-Gruppen. Entscheiden Sie, wie Sie ohne zentrale Kontrollinstanz mit dynamisch ändernden Workern am besten umgehen.
 - a. Wie erhalten neu hinzukommende Worker Aufgaben? Verhindern Sie, dass Worker lange Zeit auf (vorhandene) Arbeit warten und somit Rechenzeit verschwendet wird.
 - b. In der Gruppe reden N Clients mit M Workern. Wie entscheiden Worker für welchen Client sie arbeiten?
 - c. Wie verhindern Sie, dass Clients sich gegenseitig die Worker „wegnehmen“ und letztlich kein Client Fortschritt machen kann? Vorsicht: Es ist *nicht* gefordert, dass Worker fair auf Clients verteilt werden. Lediglich soll verhindert werden, dass Clients ihre Arbeit unterbrechen, um von Client zu Client zu wechseln, ohne je eine Aufgabe zu beenden.

Das Aktormodell

Aktoren sind nebenläufige, unabhängige Softwarekomponenten, die keine gemeinsame Sicht auf 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. Hier nutzen wir für die Kommunikation zwischen Aktoren Gruppen, die ein lose verteiltes Publish-Subscribe Modell darstellen.

Ein einzelner CAF-Knoten agiert als Anker für die Gruppenkommunikation. Er ruft zu Beginn seiner Laufzeit `publish_local_groups()` auf und hat keine andere Funktion. Dann können sich andere Knoten mit `remote_group()` ein Gruppen-Handle zum Nachrichtenaustausch holen. Aktoren, die Nachrichten aus einer Gruppe empfangen wollen, müssen `join()` mit dem Handle aufrufen.

Beim Bearbeiten einer Nachricht kann die Adresse des Senders über `self->current_sender()` ausgelesen werden. Diese Adresse lässt sich mit anderen Aktor Handles vergleichen.

Kurzanleitung und Grundgerüst für CAF

Für CAF benötigen Sie einen C++ 17 kompatiblen C++ Compiler. Unter Windows benötigen Sie mindestens Microsoft Visual Studio 2019. Auf allen Systemen benötigen Sie zusätzlich CMake.

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/inetrg/vslab-caf.git
$ cd vslab-caf
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Das Projekt ist so konfiguriert, dass CAF im Hintergrund geladen und gebaut wird. Im Ordner `vslab-caf/src` finden Sie die Quellcodedatei `main.cpp` mit einem Grundgerüst für die Praktikumaufgabe: alle notwendigen Includes, ein beispielhaftes Command-Line-Interface, Code für das Serialisieren des 512-bit Integer 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 allen Fragen rund um C++ und die Standardbibliothek: <http://en.cppreference.com/w/>

Aufgabenstellung

Teilaufgabe 1:

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

Hierzu benötigen Sie:

1. Einen Client, welcher die zu faktorisierende Zahl entgegennimmt, die Teilaufgaben erstellt, an die Worker (Aktoren) schickt 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. Der Client bezieht zur Laufzeit hinzukommende oder ausfallende Worker dynamisch in die Arbeitsverteilung ein. Dabei kommunizieren Worker und Clients nie direkt, sondern immer über die Gruppe.
2. Kommunizierende Worker (Aktoren), die
 - a. Aufträge aus der Gruppe entgegennehmen,
 - 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 Clients 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 Client, 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.

Ein Server, der als Einstiegspunkt in die Gruppenkommunikation dient, ist bereits im Grundgerüst implementiert.

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, vergleichen Sie mit anderen Lösungen.