Verteilte Systeme

Belegarbeit 03.02.2012

Bearbeitungszeit 3 Wochen

Die professionelle Entwicklung eines Sudoku-Spiels auf Java-Basis ist an einer kritischen Stelle ins Stocken geraten. Nach Entwicklung aller Komponenten und Medien, und des damit verbundenen Verbrauchs fast aller Entwicklungsgelder, hat sich herausgestellt, dass die Performance des Spiele-Kerns bei der Population größerer Rätsel zu wünschen übrig lässt. Da das Entwicklungsbudget keinerlei Spielraum für das Hinzuziehen weiterer Experten erübrigt, müsst Ihr Euch als Teil des Entwicklungsteams des Problems annehmen.

Die grobe technische Analyse hat ergeben daß der Sudoku-Lösungsalgorithmus mit zunehmender Sudoku-Größe (Dimension) kritisch wird, da dieser bei der Reduzierung des Rätsels für alle Zellen einmal aufgerufen wird. Eine Parallelisierung dieser Aufrufe selbst ist nicht möglich, aber innerhalb des Lösungsalgorithmus ergeben sich einige Möglichkeiten für Parallelverarbeitung. Das Design-Team hat daraufhin mehrere Strategien zur Verteilung des Lösungsalgorithmus zur Erprobung vorgeschlagen, welche Ihr nun implementieren und testen sollt.

Sudoku Core-Klassen:

Jedes Sudoku-Rätsel verfügt über eine Dimension, welche die Größe des Rätsels bestimmt: Jedes Sudoku hat Dimension^4 Zellen, seine Elemente (Zeilen, Spalten, Segmente) haben jeweils Dimension^2 Zellen. Ein "normales" Sudoku hat also 81 Zellen, Elementgröße 9, und Dimension 3. Die Zellen werden dabei mit Ziffern besetzt deren Basis der Elementgröße entspricht: Ein Sudoku mit Dimension 3 basiert also auf 9 verschiedenen Ziffern [0, 8], ein Sudoku mit Dimension 4 besitzt hexadezimale Ziffern [0-9, a-f], und ein Sudoku mit der maximalen Dimension 6 benötigt Ziffern zur Basis 36, i.e. [0-9, a-z]. Intern werden die Ziffern als Bytes mit den Werten [-1, 35] abgelegt, der Wert -1 representiert dabei undefinierte Zellen.

Die finale Klasse `Sudoku` stellt das Basisgerüst für das Sudoku-Spiel dar. In dieser sind die Basisalgorithmen zum Erzeugen neuer Rätsel (`void populate()`), zum Reduzieren auf eine minimale Anzahl vorgegebener Zellen (`void reduce()`), und zum Lösen der Rätsel (`Set<Sudoku> resolve()`) definiert. Daneben definiert diese Klasse auch die main()-Methode.

Diejenigen Methoden, die zur Verteilung geeignet sind, sind im Interface `SudokuPlugin` deklariert. Die Methode "`byte getDimension()`" gibt dabei die Dimension eines Sudoku zurück.Die Methode "`short getRadix()`" liefert analog das Quadrat der Dimension, i.e. die Länge der Elemente und die Basis der Ziffern. Die finale Klasse `SudokuPlugin0` implementiert dieses Interface und stellt die bestehende single-threading Lösung dar. Die Idee ist, dass für jede Teilaufgabe eine Kopie dieser Klasse erzeugt werden soll um verschiedene Ideen zur Verteilung zu erproben – ohne dabei den gesamten Sudoku-Algorithmus verstehen oder gar ändern zu müssen.

Änderungen sind daher bei Verteilung in Aufgabe 1&2 nur an diesen Kopien erlaubt. Die zur Abänderung anstehenden Methoden sind dabei:

- `Sudoku create()`: Gibt eine Sudoku.Instanz der gleichen Klasse wie die des Empfängers zurück. Achtet darauf dass hier die richtige Sudoku Subklasse zum Instantiieren verwendet wird!
- `void main(String[])`: Diese Test-Methode muss ebenfalls die richtige Sudoku-Klasse zum Instantiieren verwenden!
- `Set<Byte> getAntiSolutions(ElementType, int)`: Berechnet die möglichen Ziffern, die im gesamten gegebenen Sudoku-Element auftreten können – ausser an der gegebenen Position
- `Set<Sudoku> resolve(int, int, Set<Byte>)`: Implementiert das Lösungs-Backtracking für die gegebene Menge an Ziffern-Alternativen an der gegebenen Position. Die gegebene Rekursionstiefe dient dabei analytischen Zwecken.

(Aufgabe 1: 5 Punkte): Vektor-Prozessierung

Kopiert die Klasse `SudokuPlugin0` nach `SudokuPlugin1` und modifiziert Letztere bei dieser Aufgabe. In der Methode `getAntiSolutions(ElementType, int)` sind die drei Aufrufe der Methode

`getSolutions(int)` sowie das Hinzufügen des Ergebnisses zur Resultat-Menge distributiv. Daher soll versucht werden diese Tätigkeiten vektorprozessorartig auf verschiedene Threads zu verteilen.

Für das Starten der Threads an mehreren Stellen der Methode soll eine (innere statische oder externe) Runnable-Klasse `SudokuSolver` definiert werden. Es soll eine Semaphore zum Resynchronisieren dieser Threads genutzt werden, die Gesamtzahl der erzeugten Threads soll hingegen unbeschränkt bleiben. Die Anzahl der Threads soll aber minimal sein, daher soll für bereits definierte (festgelegte) Zellen kein Thread gestartet werden! Wichtig: Achtet auf das korrekte Freigeben der Semaphore-Tickets!

Messt den Erfolg dieser Maßname mit verschiedenen Sudoku-Dimensionen (NICHT im Debug-Modus!), und dokumentiert das Ergebnis im Klassenkommentar der Klasse SudokuPlugin1. Achtet darauf eventuelle kritische Abschnitte in Mengenoperationen geeignet zu synchronisieren.

(Aufgabe 2: 10 Punkte): Threadbasierter Rekursionsbaum

Kopiert die Klasse `SudokuPlugin0` nach `SudokuPlugin2` und modifiziert Letztere bei dieser Aufgabe. In der Methode `resolve(int, int, Set<Byte>)` sind die rekursiven Aufrufe gegen `resolve(int)` distributiv. Diese erfolgen immer dann wenn der Auflösungsalgorithmus keine Lösung mehr direkt erschließen kann und daher zwischen mehreren Alternativen raten muss (Backtracking). Verteilt die Berechnung der Alternativen auf mehrere Threads so dass die CPUs einer Maschine sinnvoll genutzt werden können. Achtet aber dabei darauf innerhalb des Rekursionsbaumes nicht wesentlich zuviele Threads zu starten; die Verwendung von Multithreading soll daher auf die oberen Rekursionsebenen beschränkt bleiben was mit Hilfe der übergebenen Rekursionstiefe sinnvoll abgeschätzt werden kann wenn ihr pro Rekursion im Mittel von 2-3 rekursiven Aufrufen ausgeht.

Messt den Erfolg dieser Maßname mit verschiedenen Sudoku-Dimensionen (NICHT im Debug-Modus!), und dokumentiert das Ergebnis im Klassenkommentar der Klasse SudokuPlugin2. Achtet darauf eventuelle kritische Abschnitte in Mengenoperationen geeignet zu synchronisieren.

(Aufgabe 3: 10 Punkte): Web-Service

Es soll eine zentrale Internet-Datenbank für Sudokus und deren Lösungen aufgebaut werden, welche mit Hilfe eines RPC-basierten WebService angesprochen werden kann. Der WebService dazu soll Bottom-Up entwickelt werden und folgendes Interface implementieren:

```
public interface SudokuService {
   void storeSolution(byte[] digitsToSolve, byte[] digitsSolved);
   byte[] getSolution(byte[] digitsToSolve)
}
```

Die Persistierung der Digits in der WebService-Implementierung `SudokuServer` soll dazu der Einfachheit halber mittels einer MySQL Datenbank erfolgen, definiert dafür ein geeignetes Schema mit einer Tabelle und gebt das Scribt zur Datenbankerzeugung mit ab. Zur Vereinfachung der Suche kann dabei entweder auf Indizierung oder ein geeignetes Hash-Verfahren (Quasi-IDs) zurückgegriffen werden.

Die Verwendung des Services in den Sudoku-Cients soll in der Klasse `Sudoku` folgendermaßen eingebaut werden: Die URL des Service soll als Quasi-Konfigurationsdatum in einer Static-Variable “SERVICE_URI” vom Typ URI gespeichert werden. Beim Lösen eines gegebenen Sudoku's soll zuerst per Service-Aufruf überprüft werden ob eine Sudoku-Lösung bereits bekannt ist, und wenn ja die existierende Lösung ausgegeben werden; ansonsten soll das Ergebnis neu berechnet werden. Immer wenn ein Sudoku neu berechnet wurde (kann in mehreren Modi der Main-Methode passieren), soll das Ergebnis an den zentralen Sudoku-Service kommuniziert werden.

Messt den Erfolg dieser Maßname mit verschiedenen Sudoku-Dimensionen (NICHT im Debug-Modus!) im Sudoku-Client, und dokumentiert das Ergebnis im Klassenkommentar der Klasse SudokuServer. Entwerft Euren Testcase dabei so dass ca 10% der Sudokus dem Service bereits vorab bekannt sind, und 90% der Lösungen berechnet werden müssen.

(Aufgabe 4: 5 Punkte): Web-Service Lifecycle

Um den Sudoku-Service geziehlt und sicher von Außerhalb des Server-Raumes beenden zu können soll in der SudokuServer.main()-Methode ein Custom-Protokoll Verwendung finden. Übergebt dazu in besagter main-Methode zusätzlich einen Control-Port und ein Passwort. Horcht innerhalb des Main-Threads (ein Thread genügt hier, lagert Eure Implementiertung daher einfach in eine static-Methode *waitForShutdown()* aus) auf eingehende Verbindungen auf diesem Port und erwartet folgendes Custom-Protokoll:

```
request: <password>
response: ”ok” | “fail”
```

Entwickelt dazu einen passenden Client in einer Klasse `ServiceStopper`, der in seiner main-Methode eine Verbindung zum Control-Port des Servers aufbaut und das Passwort übergibt. Der Server soll dann überprüfen ob das gegebene Passwort dem entspricht welches er beim Aufruf übergeben bekommen hat, und wenn ja den Server geordnet beenden. Zuvor soll dem Client in der Response noch mitgeteilt werden ob das Passwort korrekt war oder nicht.

Ihr sollt zur Implementierung dieser Aufgabe auf die Klassen `DataInputStream` und `DataOutputStream` zurückgreifen. Die Strings sollen dabei UTF-encodiert übertragen werden.

Gruppenstärke:	2-3, Einzelarbeiten bitte nur in Ausnahmefällen
Bearbeitungszeit:	3 Wochen bis Freitag 24.02.2012, 24:00 Uhr
Hilfsmittel:	Alle außer Code anderer Gruppen
Ports:	8001 - 8010 im Rechnerraum, sonst frei wählbar
Kommentare:	Keine außer wenn explizit verlangt
Hotline:	Bei Verständnisfragen täglich außer Freitags ab 13:00 Uhr unter 0174/3345975 oder 030/859 729 44. Bitte keine Fragen per Email.
Abgabeformat:	1 jar-File mit dem Quellcode (!) und dem File /META_INF/authors.txt welches Eure Matrikelnummern, Email-Adressen und Namen beinhaltet. Bitte beides unter keinen Umständen vergessen!
Abgabemodalitäten:	Übergabe per Email an sascha.baumeister@gmail.com
Bewertung:	30 Punkte maximal. Pro Teilaufgabe 0.5 bis 1 Punkte Abzug für jeden Fehler, jede Auslassung und für jeden grob umständlichen Lösungsansatz. Die geforderten Schluss-folgerungen und Messergebnisse zählen je 2 Punkte