Verteilte Systeme

Belegarbeit 07.02.2014

Bearbeitungszeit 3 Wochen

Die professionelle Entwicklung eines Schach-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 Berechnung längerer Zugsequenzen zu wünschen übrig lässt. Zudem lässt die Performance des Spiele-Kerns bei der Berechnung längerer Zugsequenzen zu wünschen übrig. Da das Entwicklungsbudget keinerlei Spielraum für das Hinzuziehen weiterer Experten erübrigt, müsst Ihr Euch als Teil des Entwicklungsteams des Problems annehmen. Hintergründe zu Schach findet ihr unter http://en.wikipedia.org/wiki/Chess.

Der bestehende Code besteht im Prinzip aus folgenden Klassen:

- **`ChessPieceType`**: Schachfigurentypen (ohne Farbe)
- **`ChessTableBoard`**: Tabellarische Implementierung eines Schachbretts
- **`ChessAnalyzer`**: KI-Engine zur Implementierung eines Computergegners
- **`ChessXfenCodec`**: Encodierung/Decodierung von Schachstellungen in erweiterter Forsyth-Edwards Notation (X-FEN) – siehe http://en.wikipedia.org/wiki/X-FEN
- **ChessClient**: Swing-basierter GUI-Client
- **ChessCommand**: Kommando-Zeilen basierter Client, kann für Laufzeitmessungen abgeändert und verwendet werden

Der GUI-Client zum Schach-Spiel kann ohne Parameter gestartet werden, in diesem Fall erscheint ein 8x8 Spielbrett in Standardaufstellung. Dabei spielt der User immer weiß, während der Computer stets schwarz spielt. Das Spiel unterstützt altertümliche Schachvarianten mit mehr/weniger als 8 Spalten, und zusätzlichen Figuren (Erzbischof = Läufer+Turm, Kanzler = Springer+Turm, Kaiserin = Dame+Springer). Daher kann der Client mit folgenden Parametern gestartet werden:

1. Analyzer-Klasse: `de.htw.ds.board.chess.ChessAnalyzer`
2. Board-Klasse: `de.htw.ds.board.chess.ChessTableBoard`
3. Suchtiefe für die KI: empfohlen für kleinere Bretter 6, für größere 4, sonst 5
4. Anzahl der Zeilen auf dem Schachbrett (für Euch immer 8)
5. Anzahl der Spalten auf dem Schachbrett (für Euch 3-10)

Die Laufzeit-Argumente

```
"de.htw.ds.board.chess.ChessAnalyzer de.htw.ds.board.chess.ChessTableBoard 4 8 10"
```

z.B. starten ein Schachspiel in Standard-Aufstellung für die normalen Schachfiguren plus zwei Erzbischöfe. Bei 9 Zeilen werden die normalen Schachfiguren plus ein Kanzler verwendet, usw.

Statt der Parameter 4 und 5 kann auch die X-FEN Repräsentation einer Schachstellung übergeben werden. So starten folgende Laufzeit-Argumente ein Schach-Spiel in Grundaufstellung:

```
"de.htw.ds.board.chess.ChessAnalyzer de.htw.ds.board.chess.ChessTableBoard
5 "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
```

Aufgabe 1: Vektor-Prozessierung (5 Punkte)

Vorbereitung: Kopiert die Klasse `ChessTableBoard` nach **`ChessTableBoard1`**, und modifiziert die letztere. Die Methode getActiveMoves() iteriert über das gesamte Spielfeld, wobei jede einzelne Iteration distributiv zu jeder anderen Iteration austauschbar ist. Daher soll versucht werden die Leistung des Gesamtsystems durch Vektorprozessor-artige Implementierung dieser Methode zu steigern, wozu die Methode mittels so vieler Threads ausgeführt werden soll wie Prozessorkerne im System verfügbar sind.

Dazu soll ein Runnable im Zusammenspiel mit einer verschuldeten Semaphore zum Einsatz kommen, und die Threads sollen in einem globalen Thread-Pool (sic!) verwaltet werden der Daemon-Treads verwendet; für letztes könnt ihr die Klasse `de.sb.javase.sync.DaemonThreadFactory` verwenden. Dabei soll beim Aufruf von `collectMoves()` der Parameter `mustCaptureKing` immer als `false` übergeben, und die Ergebnisse gesammelt werden. Falls einer der Threads `true` zurück liefert, müssen vor der Rückgabe alle Teilergebnisse (erstes Argument wird befüllt!) die `false` lieferten eliminiert werden (es sind dann nur Züge gültig die den gegnerischen König „schlagen“).

Die Wirksamkeit der Änderung ist mittels einer Testreihe abzuschätzen und im Methoden-Kommentar der neuen Implementierung zu dokumentieren. Dazu soll zuerst das Messsystem dokumentiert werden, i.e. Prozessor-Typ und Abschätzung maximaler Skalierungsfaktor (Single-Core/Multi-Core/Hyper-Threading). Sodann ist die Laufzeit der ursprünglichen Single-Threading Implementierung mit der Multi-Threading Implementierung zu vergleichen, und schließlich zu prognostizieren ab wie vielen Prozessoren bzw. ab welcher Suchtiefe sich die Maßnahme lohnt.

Aufgabe 2: Threadbasierter Rekursionsbaum (10 Punkte)

Vorbereitung: Erzeugt eine Subklasse von `ChessAnalyzer` namens **`ChessAnalyzer2`**, und modifiziert die letztere. Die Methode *predictMoves()* führt eine Rekursion aus, wobei jede Iteration distributiv zu jeder anderen Iteration austauschbar ist. Der Algorithmus soll so verändert werden dass auf der ersten (sic!) Rekursionsebene vektorprozessorartig Multi-Threading mittels eines Callables im Zusammenspiel mit einem Future durchgeführt wird, wobei jeder mögliche Zug nebenläufig berechnet werden soll. Die Threads sollen wie in Aufgabe 1 in einem statischen Thread-Pool verwaltet werden.

Die Implementierung soll zudem je nach Suchtiefe die ursprüngliche Single-Threading oder die neue Multi-Threading Implementierung aufrufen, je nachdem welche der beiden als schneller eingestuft wird.

Die Wirksamkeit der Änderung ist mittels einer Testreihe abzuschätzen und im Methoden-Kommentar der neuen Implementierung zu dokumentieren. Dazu soll zuerst das Messsystem dokumentiert werden, i.e. Prozessor-Typ und Abschätzung maximaler Skalierungsfaktor (Single-Core/Multi-Core/Hyper-Threading). Sodann ist die Laufzeit der ursprünglichen Single-Threading Implementierung mit der Multi-Threading Implementierung zu vergleichen, und schließlich zu prognostizieren ab wie vielen Prozessoren bzw. ab welcher Suchtiefe sich die Maßnahme lohnt.

Aufgabe 3: JAX-WS Web-Service (10 Punkte)

Gerade zu Beginn eines Schach-Spiels gibt es sehr viele Zugoptionen, was die Performance des Spiels (und damit die handhabbare Suchtiefe des Zugfindungsalgorithmus) negativ beeinflusst. Daher soll ein Datenbank-gestützter Zugvorhersage-Cache als globaler Web-Service eingerichtet werden der für gegebene Spielbrett-Stellungen gecachte Analyse-Ergebnisse liefert – im Endeffekt etwas ähnliches wie eine Eröffnungsbibliothek; siehe http://en.wikipedia.org/wiki/Chess_opening zur Ableitung von Beispieldaten:

```
CREATE TABLE OpeningMove (
     identity BIGINT AUTO_INCREMENT,
     position VARCHAR(128) NOT NULL,
     source SMALLINT NOT NULL,
     sink SMALLINT NOT NULL,
     rating BIGINT NOT NULL,
     searchDepth TINYINT NOT NULL,
     PRIMARY KEY (identity),
     UNIQUE KEY (position, source, sink)
) ENGINE=InnoDB;
```

Als „position“ sind dabei XFEN-Repräsentationen von Schach-Stellungen zu speichern, allerdings ohne Angabe der Zuguhren – also nur Stellung plus Rochade-Rechte plus aktive Partei. „source“ und „sink“ verstehen sich als eindimensionale Positionsangaben, i.e. rank * fileCount + file.

Darauf basierend soll bottom-up ein JAX-WS basierter Web-Service **`ChessService`** implementiert werden. Achtet darauf dass Datenbank-Verbindungen bei Inaktivität Server-seitig geschlossen werden; daher soll zur Verhinderung dieses Verhaltens eine Instanz von `de.sb.javase.JdbcConnectionMonitor` geeignet eingesetzt werden. Achtet auf kritische Abschnitte bei Verwendung der Datenbank-Verbindung. Folgende beiden Service-Methoden sollen für den Web-Service definiert werden:

MovePrediction[] getMovePredictions (String xfen, short searchDepth);

Falls einer der Parameter *`null`* ist oder sich in einem illegalen Wertebereich befindet, soll die Operation sofort *`null`* zurückgeben. Andernfalls soll diese Operation mittels der gegebenen XFEN-Repräsentation einer Schachstellung alle darauf passenden Zugvorhersagen aus der Datenbank ermitteln und zurück liefern; passende Zugvorhersagen sind dabei solche die mit der gleichen (aber reduzierten) X-FEN Stellung verknüpft sind, und mit einer Suchtiefe assoziiert sind welche gleich oder höher als die gegebene ist. Jeder Zug in einer Zugvorhersage besteht dabei aus 2 short-Werten. Falls keine für die aktive Seite und Suchtiefe geeigneten Eröffnungszüge zur Verfügung stehen, soll eine leere Menge zurück geliefert werden.

void putMovePrediction (String xfen, short searchDepth, MovePrediction movePrediction);

Falls einer der Parameter `null` ist oder sich in einem illegalen Wertebereich befindet, soll die Operation sofort „normal“ verlassen werden. Andernfalls soll diese Operation die gegebene Zugvorhersage in der Datenbank speichern, allerdings nur falls noch keine Zugvorhersagen für die gegebene X-FEN Stellung mit höherer Suchtiefe gecached sind. Daher sollen zuerst alle Zugvorhersagen für die gegebene X-FEN Stellung mit einer Suchtiefe, die kleiner (sic!) als die gegebene ist, gelöscht werden. Dann soll versucht werden eine Zugvorhersage mit den übermittelten Daten in der Datenbank zu speichern. Misslingt dies, soll nichts gespeichert werden. Die Service-Methode selbst soll als asynchron (i.e. one-way) deklariert sein werden, und die beiden Datenbank-Statements sind in einer lokalen ACID-Transaktion zu vereinigen.

Basierend auf dem JAX-WS Webservice wiederum soll eine Subklasse von `ChessAnalyzer` namens **`ChessAnalyzer3`** erzeugt, und deren Methode **`predictMoves()`** folgendermaßen angepasst werden: Falls die Zuguhr des übergebenen Boards größer oder gleich als **20** ist, soll wie vorher sofort `predictMovesSingleThreaded()` aufrufen, und deren Ergebnis zurück gegeben werden. Andernfalls soll zuerst versucht werden mittels Web-Service Aufruf von *getMovePredictions()* gecachte Zugvorhersagen zu erhalten; sind mehrere solche vorhanden, dann soll per Zufall eine davon ausgewählt und zurück gegeben werden. Falls keine vorhanden sind, soll per Aufruf von `predictMovesSingleThreaded()` eine neue berechnet, mittels WebService Aufruf von *putMovePrediction()* im Zugvorhersage-Cache gespeichert, und zurück gegeben werden.

Die zur Erzeugung des Service-Proxies notwendige Web-Service URI soll in einer static-Variable der Klasse `ChessAnalyzer2` als Konstante vorgegeben werden.

Aufgabe 4: Custom-Protokoll für Web-Service Stop (5 Punkte)

Erweitert die Main-Methode Eurer Web-Service Implementierung so dass ein Control-Port und ein Stop-Passwort übergeben werden kann. Ändert sie zudem so ab dass nicht mehr auf eine Tastatureingabe („quit“) gehorcht wird, sondern über den Control-Port auf ein Text-basiertes Custom-Protokoll mit folgender EBNF-Definition:

`request` `:= „CSP“ password`

`response` `„CSP“ „ok“ | „fail“`

Es reicht dazu aus das Service-Only Entwurfsmuster innerhalb des Main-Threads zu realisieren. Die Texte sollen dabei als UTF-Strings mittels `DataInputStream` bzw. `DataOutputStream` gelesen und geschrieben werden. Falls das per Custom-Protokoll übertragene Passwort mit dem in der Main-Methode als Parameter vorhandenen übereinstimmt soll der WebService-Prozess geordnet beendet werden.

Implementiert zudem eine Client-Klasse `ServiceStopper` mit deren Hilfe sich der WebService stoppen lässt. Diese soll als Argumente neben der URL des WebService auch das Password übernehmen, an den Web-Service übertragen, und anhand einer Statusmeldung erkennen lassen ob der Service erfolgreich gestoppt wurde oder nicht.

Gruppenstärke: 2er oder 3er-Gruppen, Ausnahmen nur in genemigten Ausnahmefällen

Bearbeitungszeit: 3 Wochen bis Sonntag 02.03.2014, 24:00 Uhr

Hilfsmittel: Alle außer Code anderer Gruppen

Kommentare: Keine außer wenn explizit verlangt

Hotline: Bei Verständnis-fragen täglich außer Freitags ab 12:00 Uhr unter *0174/3345975*. Bitte keine Fragen per email!

Abgabeformat: 1 jar-File mit dem Quellcode (sic!) 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 2 Punkte Abzug für jeden Fehler, jede Auslassung und für jeden grob umständlichen Lösungsansatz. Die geforderten Schlussfolgerungen und Messergebnisse zählen 1-2 Punkte.