Verteilte Systeme

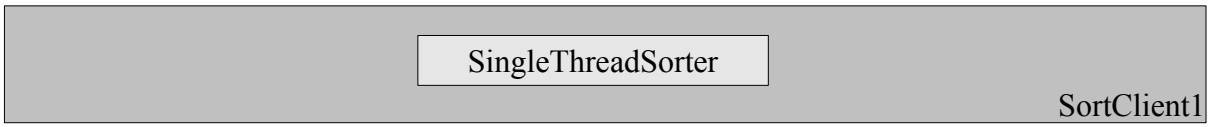
Übung X

Bearbeitungszeit 2 Wochen

Die Übung adressiert die Verteilung von Algorithmen in Clustern und Supercomputern mittels Rekursion über Prozesse und Threads. Zur Demonstration wird ein MergeSort-Algorithmus verwendet und erweitert. Die Messungen müssen auf Mehrkernsystemen ausgeführt werden um sinnvolle Ergebnisse zu liefern, beachtet jedoch dabei ob es sich bei Euren Computern um echte Mehrkernsysteme, oder nur um virtuelle Barrel-Technologie bzw. Hyperthreading handelt! Interessante Quellen sind:

- Merge-Sort Verfahren: http://en.wikipedia.org/wiki/Merge_sort
- EBNF-Syntaxbeschreibungen: http://en.wikipedia.org/wiki/EBNF

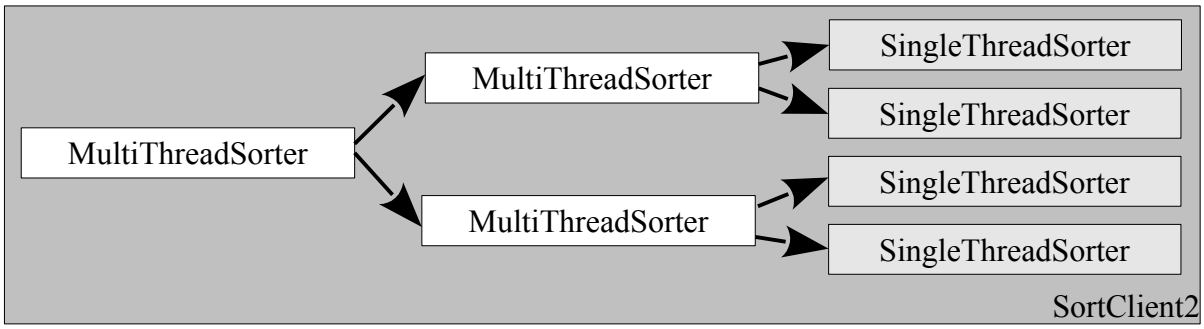
Aufgabe 1: Einfache Rekursion



Vorarbeit: Verwendet einen Archivmanager, einen Text-Editor sowie Copy&Paste um aus der Datei "goethe-faust.zip" eine Text-Datei "goethe-faust-x50.txt" in 50-facher Länge zu replizieren. Eine solche wird benötigt um für die kommenden Sortieroperationen genügend Daten verfügbar zu haben.

Bringt die Applikation **SortClient1** zum Laufen, und messt aus wie lange es dauert diese Datei nach Wörtern zu sortieren. Macht Euch mit den Klassen ***SingleThreadSorter***, ***SortClient1***, sowie deren abstrakter Superklasse ***SortClient*** vertraut.

Aufgabe 2: Rekursion über multiple Threads eines Prozesses



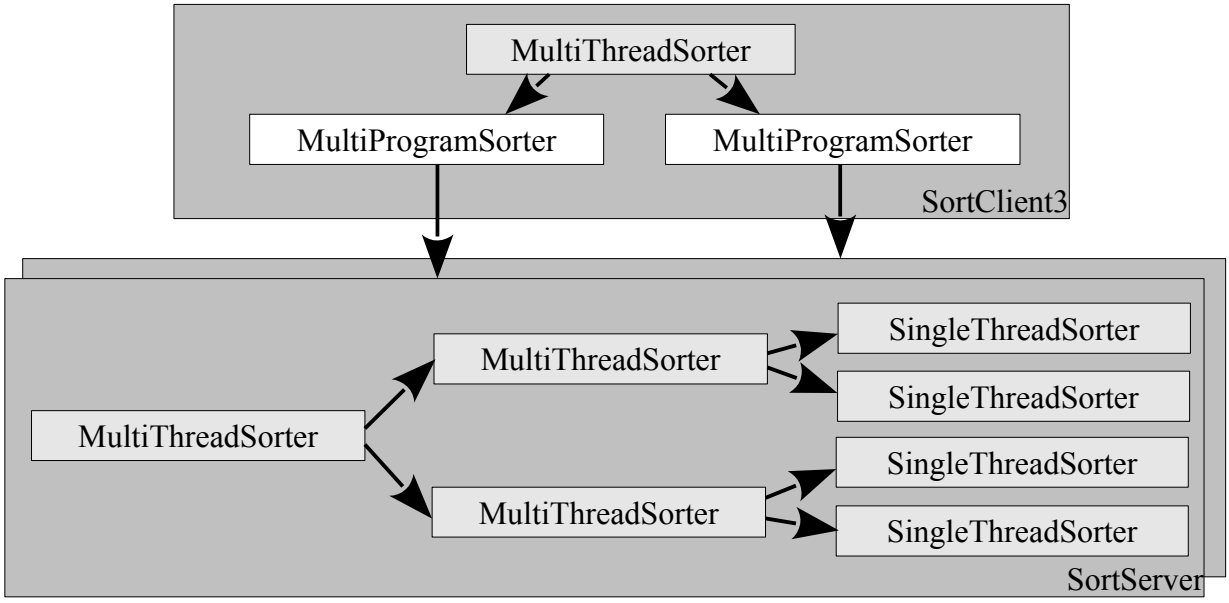
Benennt die Klasse *MultiThreadSorterSkeleton* in ***MultiThreadSorter*** um. Skaliert die Implementierung wie in Methode sort() unter TODO beschrieben vektorprozessorartig auf zwei Threads.

Kopiert die Klasse *SortClient1* nun nach ***SortClient2***, und schreibt sie so um dass die Rekursion auf eine Verknüpfung von Objekten abgebildet wird, i.e. einen Objektbaum wie oben gezeigt. Realisiert dabei zuerst die oben gezeigte Struktur, unabhängig von der Zahl der Prozessorkerne im System. Testet nun diesen Zwischenstand auf korrekte Funktion.

Ein solcher Setup wäre für Vierkern-Systeme optimal, aber dummerweise soll Eure Software für eine beliebige Zahl von Kernen optimiert sein. Überlegt Euch wie mit Hilfe einer Queue (*java.util.ArrayDeque* implementiert *java.util.Queue*) von Sorter-Instanzen sehr elegant ein balancierter Sortier-Baum wie oben gezeigt aufgebaut werden kann, egal wie viele Kerne im System vorhanden sind. Tip: Wenn man jeweils zwei Elemente aus einer Queue entfernt, diese zu einem neuen Knoten kombiniert, und den entstandenen Knoten wieder in die Queue einfügt, dann enthält die Queue irgendwann nur noch einen einzelnen Knoten. Falls eine Umsetzung mittels Queue nicht gelingt, unterstützt im Code stattdessen die Spezialfälle 1, 2, 4, und 8 Prozessorkerne durch manuell instantiierte Sortierer-Bäume. Verwendet dabei immer die Wurzel des Baumes (links im Bild) zum Sortieren in Eurem Client.

Messt aus wie lange es dauert die Datei "goethe-faust-x50.txt" mittels *SortClient2* zu sortieren. Vergleicht die Laufzeit mit Aufgabe 1.

Aufgabe 3: Rekursion über multiple Prozesse und Threads



Analysiert die Klasse ***MultiProgramSorter***, vor allem die darin beschriebene EBNF-Protokollsyntax und deren Client-seitige Implementierung. Benennt die Klasse *SortServerSkeleton* nach ***SortServer*** um, und implementiert an der mit TODO markierten Stelle das Server-Gegenstück zu o.g. Sortierer. Übernehmt dabei aus *SortClient2* die Erzeugung des internen Sorter-Baums zur Nutzung von Multi-Threading. Beachtet dass der SortServer an sich NICHT das Anfrage-basierte Multi-Threading-Modell implementiert (da nur eine Verbindung pro Zeitpunkt zugelassen ist), sondern Multi-Threading per Rekursion!

Kopiert die Klasse *SortClient1* nach ***SortClient3***, und schreibt sie so um dass eine beliebige Zahl von Socket-Adressen (z.B. „host:8002“) externer *SortServer* als zusätzliche Parameter übernommen werden – ihr könnt zum Parsen einzelner Socket-Adressen den Ausdruck *„**new** de.htw.ds.SocketAddress(text). toInetSocketAddress()“* verwenden. Realisiert zuerst den Spezialfall eines einzelnen externen Servers, indem ihr in *SortClient3* eine einzelne *MultiProgramSorter*-Instanz als Sortierer verwendet. Verwendet diese Zusammenstellung um die Funktion Eure Implementierung ausgiebig zu testen.

Sobald Eure *SortClient3*-Implementierung fehlerfrei läuft, versucht dort (ähnlich wie in Aufgabe 2) die gegebenen Socket-Adressen in einen möglichst balancierten Baum aus *MultiThreadSorter- sowie MultiProgramSorter*-Instanzen umzusetzen, wobei letztere immer die Blätter des Baumes bilden. Beachtet dass es auch hier wichtig ist dass die vorgeschalteten *MultiThreadSorter* des Baumes die Last auf mehrere Threads verteilen, denn andernfalls würden die *MultiProgramSorter* sequentiell statt parallel abgearbeitet, und damit keine zusätzliche Skalierung erreicht!

Startet nun je einen *SortServer* auf bis zu 4 separaten Maschinen (i.e. 4 Knoten eines weiten Clusters), und konfiguriert entsprechend auf einer weiteren Maschine eine Instanz von *SortClient3* - für die meisten von Euch sind so viele Computer nur im Übungsraum verfügbar. Messt wie lange es nun dauert die Datei "goethe-faust-x50.txt" mittels dieses Clients zu sortieren. Vergleicht die Laufzeit mit Aufgabe 1 und 2.

Vergrößert nun die Testdatei um herauszufinden um welchen Faktor sich die maximal sortierbare Datengröße (unter Beibehaltung der Default-Einstellungen der JVMs) beim Client3 gegenüber den Clients 1+2 erhöhen lässt bevor *OutOfMemory*-Fehler auftreten.