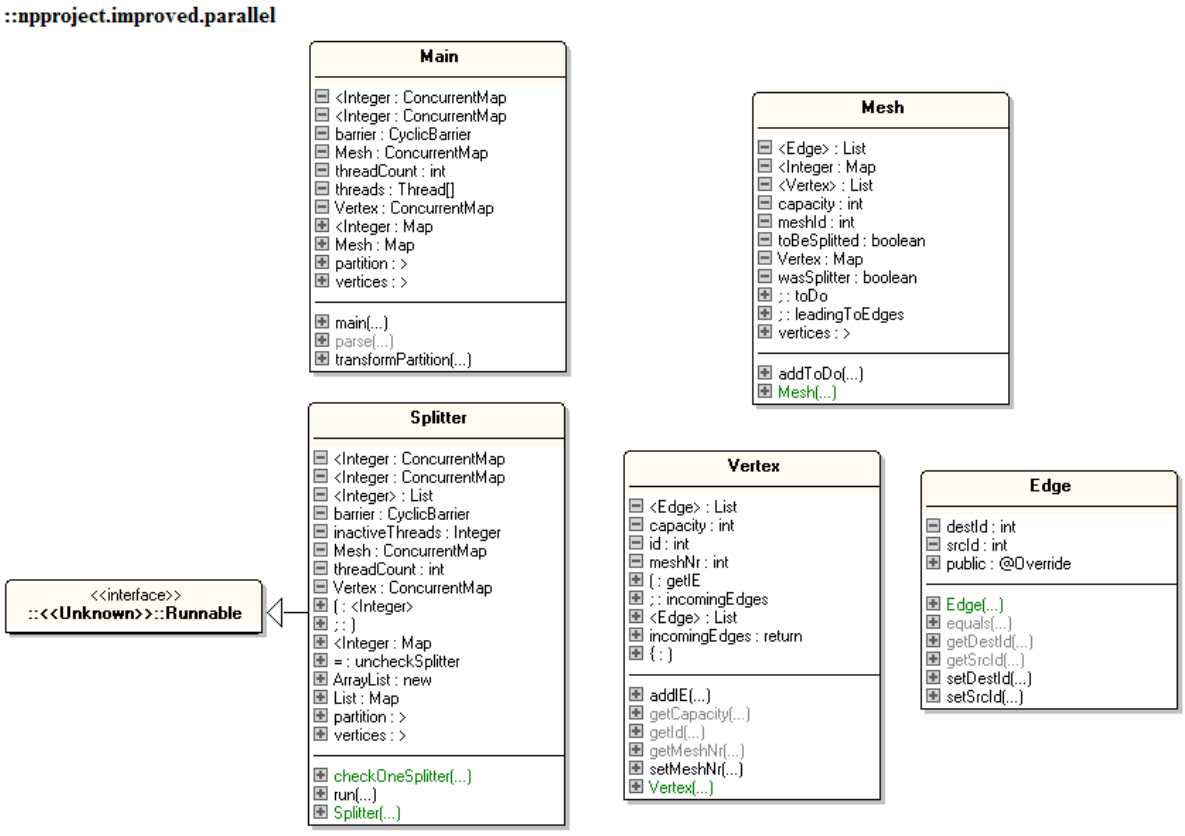
Praktisches Projekt
Nebenläufige Programmierung 2011

Nader A. Max F.

11. Juli 2011

Struktur



Jeder Knoten wird von einem Vertex Objekt dargestellt, enstprechend jede Kante von einem Edge Objekt. Diese speichern ihre jeweiligen gegebenen Attribute. Ein Vertex besitzt ausserdem neben einem Feld, in dem er die ID der ihm zugehörigen Masche speichert, auch eine Liste der auf ihn eingehenden Edges.

Ein Mesh Objekt speichert die ihr aktuell zugehörigen Knoten sowie alle eingehenden Edges aller enthaltenen Knoten. Diese Liste wird mittels `updateLE()` nach jeder Änderung aktualisiert. In der Main Klasse wird das Parsen und Bilden der ersten Partition erledigt. Die als Map von Mesh Objekten realisierte Partition wird als Referenz an die jeweilige Anzahl von Splitter Objekten übergeben. Nachdem diese möglichst parallel mittels wiederholter Aufrufe von `checkOneSplitter()` und `split()` die Partition maximal verfeinert haben und die Abbruchbedingung eintritt, wird in `main()`, nach dem Warten auf die Terminierung der gestarteten Threads, die Größe der modifizierten Map `partition` ausgegeben.

Parallelität

Parsen

Das Parsen des gegebenen XML Files sowie das Erstellen der initialen Partition erfolgt sequentiell.

Algorithmus

Wir verwenden eine spezifische Anzahl von Threads die in `main()` gestartet werden und auf deren Terminierung vor Ausgabe des Ergebnisses gewartet wird. Die Methoden der Klasse `Splitter` sollen nebenläufig ausgeführt werden, dafür implementiert diese Klasse das Interface `Runnable` und bietet folglich die Methode `run()` an, in der die Methoden `checkOneSplitter()` und `split()` bis zur Abbruchbedingung ausgeführt werden. Um redundante Arbeit zu vermeiden wird eine vollständig als Splitter bearbeitete Masche als solche markiert.
Es kommen zwei verschachtelte while Schleifen in der Methode `run()` zum Einsatz, um einerseits bei Aktivität die Abbruchbedingung immer wieder zu überprüfen, d.h. ob ein Thread eine Splitter-Masche in `partition` findet, und andererseits sämtliche in `main()` gestartete Threads nicht terminieren zu lassen solange es zumindest einen aktiven Thread gibt.
Weiterhin müssen die inaktiven Threads ebenfalls an den entsprechenden Punkten einen `barrier.await()` Aufruf ausführen, damit die aktiven Threads weiter fortschreiten können und es keine Threads gibt, die sich in verschiedenen Phasen befinden (siehe unten).

Phase eins:
Die Partition wird mittels einer `ConcurrentMap` dargestellt, dies erlaubt vollständig paralleles Arbeiten in der Methode `checkOneSplitter()`, welche ausschließlich Lesezugriff benötigt. So können die Threads für jeweils eine Masche M die zu verfeinernden Maschen mit $Pre(M) \neq M' \wedge Pre(M) \neq \emptyset$ herausfinden. Hierfür müssen zwei Fälle ausgeschlossen

werden, der erste wäre, dass zwei Threads $Pre(M)$ für die gleiche Masche M suchen. Dieser Fall wird durch ein synchronisiertes Setzen einer Flag des jeweiligen Mesh Objekts durch den ersten zugreifenden Thread verhindert. Ein ggf. notwendiges Zurücksetzen der Flag findet garantiert erst nach dem Eintritt sämtlicher Threads in Phase zwei statt.
Der zweite Fall besteht in einem überlappenden Schreibwunsch: eine Masche soll von mehreren Splittern aus innerhalb einer Iteration verfeinert werden. Hier wird wieder synchronisiert eine Flag durch den ersten überprüfenden Thread gesetzt, alle anderen Threads müssen folglich die Bearbeitung dieser Masche $M' \in Pre(M)$ auf eine folgende Iteration verschieben. Die entsprechenden vertagenden Threads dürfen nicht als vollständig bearbeitet markiert werden.

Um gleichzeitige Lese- und Schreibeaktionen zu verhindern wird nach dem Aufruf von `checkOneSplitter()` mittels einer `CyclicBarrier` eine Synchronisierung der Threads erzwungen.

Phase zwei:
Im Laufe des Methodenaufrufs von `split()` besteht Schreibzugriff auf die Liste der bestehenden Maschen. Da sämtliche Konfliktfälle aussortiert/verhindert worden sind, muss hier nur darauf geachtet werden, dass die neu entstehenden Maschen eine eindeutige ID zugewiesen bekommen sowie synchronisiert an das Ende von `partition` hinzugefügt werden. Dies wird durch einen kurzen `synchronized` Block gewährleistet, in dem sowohl die ID zugewiesen wird als auch das Objekt in die Map eingefügt wird. Da die ID als `partition.size()` implementiert ist, ist Eindeutigkeit garantiert.
Ferner werden in dieser Phase Splitter der aktuellen Iteration als nicht vollständig bearbeitet gesetzt, falls durch einen Konflikt nicht alle ihre Vorläufermaschen bearbeitet werden konnnten. Sämtliche modifizierten Maschen setzen ebenfalls ihre Flags auf „undone“.

Nach dem erfolgten `split()` Aufruf wird wiederum mittels einer `CyclicBarrier` synchronisiert.

Wenigstens ein Thread kann so bei jeder Iteration in `run()` anhand eines Splitters die Maschen durchsuchen und anschliessend verfeinern. Somit ist Lebendigkeit garantiert, Sicherheit wird durch Ausschliessen der Konflikte bei den zu verfeinernden Maschenobjekten und durch implizites Locking der Flags in Phase eins sowie von `partition` in Phase zwei erreicht.