Technologies for Big Data Analytics

Εργασία 1 - Threads and Processes - Υποερώτημα 1

Antonis Prodromou

2025-11-21

Πίνακας περιεχομένων

1 Περιγραφή Λύσης - Υποερώτημα 1

Σε αυτή την εφαρμογή δημιουργούμε ένα κατανεμημένο σύστημα, όπου επιμέρους εργασίες (tasks) εκτελούνται από διαφορετικά νήματα (threads). Συγκεκριμένα, κάθε νήμα αναλαμβάνει τον πολλαπλασιασμό μιας ομάδας (block) γραμμών ενός πίνακα ($n \times m$), με ένα διάνυσμα ($m \times 1$). Η εκτέλεση γίνεται παράλληλα με πολλαπλά νήματα, ώστε να διαπιστώσουμε εάν ο συνολικός χρόνος εκτέλεσης του πολλαπλασιασμού βελτιώνεται με την αύξηση των νημάτων που συμμετέχουν σε αυτόν. Η επιλογή της εκτέλεσης ενός block και όχι γραμμής - γραμμής (interleaving) από κάθε νήμα γίνεται για να αποφύγουμε το κόστος μεταγωγής (overhead) από νήμα σε νήμα.

1.1 Κύρια Κλάση

Για τη λύση του προβλήματος δημιουργώ μια κλάση `MatrixMultiplication`. Είναι η βασική κλάση του προγράμματος και είναι public, δηλαδή ορατή από παντού και προσπελάσιμη από όλες τις κλάσεις του προγράμματος. Ο constructor αυτής της κλάσης δημιουργεί τα αντικείμενα του πολλαπλασιασμού, δηλαδή «γεμίζει» τον πίνακα array και το διάνυσμα vector με τυχαίους αριθμούς μεταξύ 0 και 10.

1.2 Κλάση Worker

Υλοποιεί (implements) τη διεπαφή Runnable, που βάση ορισμού είναι ένα κοινό πρωτόκολλο για αντικείμενα που θέλουμε να εκτελούν κώδικα ενώ είναι «ενεργά». Τα χαρακτηριστικά της Runnable είναι:

- Μπορεί να εκτελεστεί από πολλά νήματα (threads).
- Πρέπει να υλοποιείται από μια κλάση της οποίας τα στιγμιότυπα (instances) θα εκτελούνται από threads.
- Αυτή η κλάση θα διαθέτει μια μόνο μέθοδο, την run(). Είναι μέθοδος χωρίς παραμέτρους, και σε αυτήν περιέχεται ο κώδικας που θέλουμε να τρέξουν τα threads. Εν προκειμένω, είναι ο πολλαπλασιασμός `result[i] += array[i][j] * vector[j]`
- Δεν τρέχει η ίδια τα threads. Για αυτό χρειαζόμαστε μια συνάρτηση - executor (εν προκειμένω η `multiplier()`, η οποία επεξηγείται παρακάτω).

Εναλλακτικά, θα μπορούσα να κάνω extend την κλάση Thread. Επιλέγω όμως αυτόν τον τρόπο δημιουργίας της κλάσης γιατί η Java επιτρέπει πολλαπλές υλοποιήσεις (implements) αλλά μόνο μια επέκταση (extend) ανά κλάση, προστατεύοντας έτσι από το ενδεχόμενο κοινής ονομασίας των μεθόδων των κλάσεων που γίνονται extend (το πρόγραμμα δεν θα ήξερε π.χ. ποιά από δύο αντικρουόμενες μεθόδους να τρέξει). Κάνοντας implement έχω επομένως μεγαλύτερη ευελιξεία.

1.3 Ροή εκτέλεσης

Το πρόγραμμα έχει την παρακάτω ροή εκτέλεσης:

`main() -> MatrixMultiplication() initialize -> multiplier() -> κάθε νήμα τρέχει το Worker.run()`

Όπου οι ρόλοι είναι:

- `main()` είναι ο orchestrator. Είναι το σημείο εκκίνησης του προγράμματος (entry point). Εδώ ορίζουμε τις διαστάσεις του πίνακα και του διανύσματος, καθώς και τον αριθμό των threads `{1, 2, 4, 8}` με τα οποία θα τρέξουν τα for loop. Για κάθε k threads τρέχει τον multiplier(). Παίρνει τους χρόνους που προκύπτουν από τον multiplier και τους εκτυπώνει.
- `MatrixMultiplication()` είναι ο constructor που δημιουργεί τον πίνακα και το διάνυσμα βάση των διαστάσεων που του δίνουμε στο main().
- `multiplier()` οργανώνει και εκτελεί το πείραμα. Δημιουργεί τα threads και με το `start()` δίνει εντολή στον εργάτη (Worker) για να τρέξει, περνώντας ως arguments τις παραμέτρους που παίρνει από το main(). Περιμένει να ολοκληρωθούν όλα τα νήματα με το `join()`.
- `Worker.run()` είναι ο worker. Υλοποιεί τη μέθοδο της διεπαφής `Runnable` κάνει τους πολλαπλασιασμούς του γινομένου των γραμμών του πίνακα με το διάνυσμα μέσω της μεθόδου του run(). Επιπλέον, μετρά τον χρόνο εκτέλεσης του thread και αποθηκεύει το αποτέλεσμα στον πίνακα `threadTimes`.

1.4 Συγχρονισμός

Στο συγκεκριμένο πρόγραμμα, δεν χρησιμοποιώ κάποια μέθοδο συγχρονισμού (π.χ. με ορισμό συναρτήσεων με τον τύπο synchronized), γιατί διαχωρίζω εξαρχής τις γραμμές που θα πολλαπλασιάσει κάθε νήμα:

```
int rowsPerThread = n / k;
int startRow = i * rowsPerThread;
int endRow = (i + 1) * rowsPerThread;
```

Αυτές οι παράμετροι συνιστούν και την χωριστή στοίβα για κάθε νήμα, εν αντιθέσει με τα κοινά δεδομένα που είναι π.χ. ο πίνακας και το διάνυσμα. Με άλλα λόγια, δεν δημιουργείται race condition, γιατί τα κοινά δεδομένα μόνο διαβάζονται από τα νήματα και δεν τροποποιούνται από αυτά.

1.5 Παράμετροι του προβλήματος

- `int n, m, k`: διαστάσεις του πίνακα και αριθμός νημάτων.
- `int[][] array`: ο πίνακας προς πολλαπλασιασμό.
- `int[] vector`: το διάνυσμα εισόδου (v).
- `int[] result`: το διάνυσμα εξόδου (r).
- `Random rand`: αντικείμενο της βιβλιοθήκης `java.util.Random` για την παραγωγή τυχαίων αριθμών.
- `long[] threadTimes`: πίνακας που αποθηκεύει τον χρόνο εκτέλεσης κάθε thread.
- `int startRow, endRow`: καθορίζουν το εύρος γραμμών που θα υπολογίσει κάθε thread.

1.6 Καταγραφές

Χρησιμοποιώ τις παρακάτω μεταβλητές για την καταγραφή των χρόνων εκτέλεσης:

```
long startTime = System.nanoTime();  long endTime = System.nanoTime();
long executionTime = (endTime – startTime) / 1000;
```

Επιλέγω να είναι τύπου long γιατί οι ακέραιοι (int) στην Java είναι 32-bit, δηλαδή έχουν τιμές που φτάνουν έως 2^32. Καθώς μετράμε τον χρόνο σε nanoseconds εν προκειμένω θα υπήρχε ο κίνδυνος να φτάσουμε αυτό το όριο σχετικά γρήγορα, μιας και κάθε δευτερόλεπτο έχει 1,000,000,000 ns.

---

2 Αποτελέσματα

Τρέξαμε για 1, 2, 4 και 8 νήματα τον πολλαπλασιασμό για τρία διαφορετικά μεγέθη πίνακα, και πήραμε τα παρακάτω αποτελέσματα:

```
For a (16 x 10) matrix:

1 threads: 330 microseconds
2 threads: 378 microseconds
4 threads: 786 microseconds
8 threads: 1491 microseconds
```

```
For a (64 x 50) matrix:

1 threads: 446 microseconds
2 threads: 378 microseconds
4 threads: 881 microseconds
8 threads: 1903 microseconds

For a (4096 x 1000) matrix:

1 threads: 16430 microseconds
2 threads: 17911 microseconds
4 threads: 2382 microseconds
8 threads: 2972 microseconds
```

Παρατηρούμε τα παρακάτω πως περισσότερα νήματα δεν συνεπάγονται απαραίτητα και γρηγορότερο χρόνο εκτέλεσης, καθώς αυτός προκύπτει από 1, 2 και 4 νήματα, αντίστοιχα ανά περίπτωση. Οι αιτίες είναι:

- Οσο ο υπολογιστικός φόρτος είναι μικρός (δηλαδή για τους μικρότερους πίνακες) ο χρόνος που απαιτείται για τη δημιουργία και την εναλλαγή των νημάτων (π.χ. αποθήκευση και φόρτωση καταχωρητών, initiate τις μεταβλητές, start() και join() των νημάτων, τη διαχείριση των εσωτερικών μεταβλητών τους) είναι μεγαλύτερος από αυτόν που γλυτώνουμε εκτελώντας παράλληλα τις εργασίες χρησιμοποιώντας πολλά νήματα.
- Όταν το υπολογιστικό κόστος μεγαλώνει (στους μεγαλύτερους πίνακες), το κόστος μεταγωγής από νήμα σε νήμα μέσω κοινής μνήμης είναι πλέον συγκριτικά μικρό και προκαλεί πολύ μικρότερη επιβάρυνση σε σχέση με τις διεργασίες.
- Από τη θεωρία γνωρίζουμε πως κάθε Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ) μπορεί να τρέχει μόνο ένα νήμα τη φορά. Εφόσον ο υπολογιστής στον οποίο τρέχουμε το πείραμα έχει 4 ΚΜΕ, κάθε επιπλέον νήμα (όπως π.χ. στην περίπτωση των 8) μπαίνει σε αναμονή. Για αυτόν τον λόγο βλέπουμε πως όταν ο πίνακας και το διάνυσμα είναι μεγάλα (περίπτωση 3), έχω ταχύτερο χρόνο χρησιμοποιώντας 4 νήματα παρά 8.