Όνομα: Χρήστος Πατσούρας
Αριθμός Μητρώου: M1612

Προηγμένες Μέθοδοι Προγραμματισμού HW4

- **Γενικές Πληροφορίες**: Το μηχάνημα που χρησιμοποιήσα έχει CPU Intel® Core™ i5-7200U CPU @ 2.50GHz × 4, τρέχει λειτουργικό Ubuntu 16.04 LTS 64 bit με 8GB RAM, εκ της οποίας 1GB είναι μόνιμα δεσμευμένη από το λειτουργικό. Το μέγιστο μέγεθος μνήμης στο JVM το έχω ορίσει στα 7GB και στα δύο προγράμματα Java που παραδίδω. Χρησιμοποίοησα Java 1.8 server VM. Η έκδοση του g++ είναι 5.4.0. Και για τα συνολικά τέσσερα προγράμματα που παραδίδω υπάρχει σχετικό makefile, το οποίο με την εντολή make κάνει compile, με την εντολή make run εκτελείται και με make clean σβήνει τα αρχεία που δημιουργήθηκαν κατά το compilation. Από όσο διάβασα στο Garbage Collection Tuning Guide (https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/) by default χρησιμοποιείται throughput garbage collector, δηλαδή ο garbage collector τρέχει σε ξεχωριστό thread παράλληλα με το πρόγραμμα.

- **Γρήγορη Java**: Ξεκινάω με μια μεγάλη λίστα από 20 εκατομμύρια ακεραίους και κρατάω ζωντανό τυχαία ένα στοιχείο ανά 2500 στοιχεία καταλήγοντας σε μια τελική λίστα με 8000 στοιχεία. Στην περίπτωση της Java, καλώ τον garbage collector, ώστε τα εναπομείναντα στοιχεία να αποκτήσουν καλό locality. Στη συνέχεια, για έναν μεγάλο αριθμό επαναλήψεων (40000), κάνω πράξεις στη λίστα, πηγαίνοντας σε όλους τους κόμβους της. Συγκεκριμένα, υπολογίζω το πλήθος των πρώτων και των περιττών αριθμών, το άθροισμα των στοιχείων της λίστας, τον μέγιστο και των ελάχιστο αριθμό στη λίστα και εκτυπώνω αυτά τα στοιχεία. Τέλος, από κάθε κόμβο προσθέτω ή αφαιρώ έναν μικρό τυχαίο αριθμό. Χρησιμοποιώ την std::list για C++ και την LinkedList στη Java. Στη Java χρησιμοποιώ λίστα από int. Το αντίστοιχο σε C++ είναι λίστα με int *, διότι στη Java τα στοιχεία της λίστας είναι ξεχωριστά αντικείμενα και ο κόμβος στη λίστα δείχνει σε αυτά. Οι χρόνοι στη Java είναι στο συνολικό πρόγραμμα 66 sec, ενώ στο σημαντικό κομμάτι μετά τη διαγραφή των κόμβων που μας ενδιαφέρει είναι 56 sec. Οι χρόνοι σε C++ είναι συνολικά 178 sec και το κρίσιμο κομμάτι 174 sec. Οι χρόνοι της C++ είναι 2.7 φορές και 3.11 φορές πιο αργοί αντίστοιχα. Η διαφορά αυτή μπορεί να μεγαλώσει αυξάνοντας τον αριθμό των επαναλήψεων. Ο λόγος που άφησα το μέγιστο μέγεθος μνήμης στο JVM στα 7 GB είναι για να μη φτάνει η μνήμη που δεσμεύω σε μεγάλο ποσοστό της μέγιστης μνήμης και καλείται συνεχώς ο garbage collector, επιβαρύνοντας το πρόγραμμα.

- **Γρήγορη C++**: Δημιουργώ έναν πίνακα περίπου 4.5GB, τον standard_overhead, τον οποίο γεμίζω και δεν εκμεταλλεύομαι καθόλου, απλά τον αφήνω στη μνήμη. Ο χώρος που καταλαμβάνει είναι 65-70% της μνήμης. Έτσι, οι νέες επαναληπτικές δεσμεύσεις θα αναγκάσουν τον garbage collector να κληθεί, έχοντας να ψάξει έναν πολύ μεγάλο πίνακα χωρίς κανένα νεκρό στοιχείο. Στη συνέχεια δεσμεύω και αποδεσμεύω επαναληπτικά τον πίνακα additional_overhead, ένα πίνακα 50000 θέσεων από A (δική μου κλάση), πάνω στον οποίο κάνω πράξεις για μεγάλο αριθμό επαναλήψεων (10000), αφού πρώτα τον γεμίσω με δεδομένα. Οι χρόνοι στη C++ είναι για το συνολικό πρόγραμμα 36 sec και του κρίσιμου τμήματος που είναι οι πράξεις πάνω στον additional_overhead 26 sec. Οι αντίστοιχοι χρόνοι σε Java είναι για το συνολικό πρόγραμμα 150 sec και για το κρίσιμο τμήμα 90 sec. Οι χρόνοι της Java είναι 4.17 φορές και 3.46 φορές πιο αργοί αντίστοιχα. Η διαφορά αυτή μπορεί να μεγαλώσει αυξάνοντας τον αριθμό των επαναλήψεων.

- **Ρυθμίσεις garbage collector**: Από τις επιλογές της εκφώνησης, ασχολήθηκα με το σχετικό μέγεθος της παλιάς και της νέας γενιάς και με το αν ο GC είναι παράλληλος ή όχι, καθώς και το πόσα threads έχει. Πλέον, αφού δε συγκρίνουμε με τη C++, δε θα εξετάσουμε 2 διαφορετικούς χρόνους (κρίσιμο τμήμα και σύνολο), αλλά μόνο το συνολικό χρόνο των προγραμμάτων.
  1. Σχετικό μέγεθος παλιάς και νέας γενιάς: Όπως αναφέρεται στο GC Tuning Guide της Oracle, όσο μεγαλύτερο είναι το μέγεθος της νέας γενιάς, τόσο λιγότερο συχνά γίνονται minor collections. Ωστόσο, για περιορισμένο μέγεθος heap, μεγαλύτερη νέα γενιά σημαίνει αυτομάτως μικρότερη παλιά γενιά, οπότε θα αυξηθεί η συχνότητα των major collections. Η βέλτιστη επιλογή εξαρτάται από τη διάρκεια ζωής των αντικειμένων που δεσμεύονται από την εφαρμογή. Με την παράμετρο -XX:NewRatio=N θέτω την αναλογία μεταξύ νέας και παλιάς γενιάς ίση με 1:N. Οι τιμές που δοκίμασα ήταν από 1 μέχρι 10, αφού μετά το 8 μεγάλωνε αρκετά ο χρόνος.
     Στην περίπτωση της γρήγορης Java, ο χρόνος του προγράμματος με default ρυθμίσεις είναι 66 sec. Για N=2 παρατήρησα την καλύτερη επίδοση με 63 sec. Βελτίωση παρατήρησα και για N=3,5,6 με 64 sec και για N=1 με 65 sec. Από N=7 μέχρι N=10 παρατήρησα επιβράδυνση με 67 sec, 71 sec, 70 sec και 70 sec αντίστοιχα.
     Στην περίπτωση της γρήγορης C++, ο χρόνος του προγράμματος με default ρυθμίσεις είναι 143 sec. Για N=5 παρατήρησα την καλύτερη επίδοση με 137 sec. Βελτίωση παρατήρησα και για N=3,4 με 138 sec. Για N=6 παρατήρησα τον ίδιο χρόνο με τις default ρυθμίσεις. Για N=2, 7, 9, 10 παρατήρησα επιβράδυνση με 148 sec, 146 sec, 146 sec και 145 sec αντίστοιχα. Για N=1 το πρόγραμμα έβγαλε OutOfMemoryError: Java heap space.

  2. GC παράλληλος ή όχι και πλήθος threads: Για τη συγκεκριμένη επιλογή, δοκίμασα σειριακό garbage collector (-XX:+UseSerialGC), παράλληλο garbage collector (-XX: +UseParallelGC) και διαφορετικό πλήθος threads (-XX:ParallelGCThreads=N για N=1,2,4,8,16).
     Στην περίπτωση της γρήγορης Java, ο χρόνος του προγράμματος με default ρυθμίσεις είναι 69 sec. Με σειριακό GC ο χρόνος του προγράμματος είναι 71 sec, ενώ με παράλληλο GC είναι 76 sec. Οι χρόνοι για διαφορετικό πλήθος thread είναι για N=1 thread 71 sec, για N=2 threads 73 sec, για N=4 threads 76 sec, για N=8 threads είναι 76 sec και για N=16 threds 93 sec.
     Στην περίπτωση της γρήγορης C++, ο χρόνος του προγράμματος με default ρυθμίσεις είναι 147 sec. Με σειριακό GC ο χρόνος του προγράμματος είναι 137 sec, ενώ με παράλληλο GC είναι 150 sec. Οι χρόνοι για διαφορετικό πλήθος thread είναι για N=1 thread 151 sec, για N=2 threads 153 sec, για N=4 threads 148 sec, για N=8 threads είναι 158 sec και για N=16 threds 163 sec. Για την περίπτωση του σειριακού GC, βλέποντας το system monitor, παρατήρησα ότι το πρόγραμμα καταλαμβάνει πολύ περισσότερη μνήμη, σχεδόν 1.5GB παραπάνω, σε σχέση με τη default περίπτωση και τον παράλληλο GC.