Για την ανάπτυξη των προγραμμάτων χρησιμοποίησα ένα 64bit Windows συστημα με 16gb μνήμη ram, C++ 14 και java 11. Στα 2 προγράμματα java δεσμευσα 8gb μνημη για το heap, δημιουργώντας 250 εκατομμύρια αντικείμενα(ζευγάρι με αργή java/γρήγορη c++) και 200 εκατομμύρια αντίστοιχα για το άλλο ζευγάρι. Ο default αλγόριθμος που χρησιμοποιεί η java 11 είναι ο **G1**.

**G1**

Ο G1 είναι ένας generational αλγόριθμος.Τα αντικείμενα γίνονται allocate στο heap σε περιοχές(regions) που ονομάζονται eden, survivors (young generation) και old generation. Κάποιες δράσεις του χαρακτηρίζονται ως “stop-the-world”, που σημαίνει ότι όλα τα νήματα του προγράμματος που είναι ενεργά τη προκειμένη στιγμή θα σταματήσουν για χάρη της stop-the-world δράσης.

Ο gc έχει 3 φάσεις: “young gc” και “old gc”.

**young gc**: Είναι παράλληλη διεργασία, που σημαίνει ότι τρέχει σε πολλά νήματα του G1 (προκαθορισμένη τιμη: 10). Κατα τη συγκεκριμένη ενέργεια τα ζωντανά αντικείμενα μετακινούνται από τη περιοχή eden προς την survivor. Αν παλαιώσουν αρκετά πάνε στην old region. Η μνήμη του young generation είναι φτιαγμένη από ένα σετ σε μη συνεχόμενες περιοχές στη μνήμη. Αυτό κάνει πιο εύκολη την αλλαγή του μεγέθους τους όταν χρειαστεί.

**old gc**: Μαρκάρει(marking) ζωντανά αντικείμενα σε όλο το heap, και έπειτα τα κάνει compact.

Ο G1 έχει 5 φάσεις που λαμβάνουν χώρα κατά το old gc. Αυτές είναι:

1) *Initial Mark*(stop-the-world): Η διαδικασία αυτή μπορεί να γίνει σε ενα normal young gc. Μαρκάρισμα των αντικειμένων με δείκτες στο old generation. Tα survivor regions μπορεί να έχουν δείκτες στο old generation(root region scanning).

2)*Root Region Scanning*: πρέπει να έχει ολοκληρωθεί πριν ένα young gc συμβεί ξανά.

3)*Concurrent Marking*: Ψάχνει ζωντανά αντικείμενα σε ΟΛΟ το heap. Αν βρει άδειο χώρο, τον αποκλείει από την διαδικασία αμέσως. Ένα young gc μπορεί να παρέμβει κατά τη διαδικασία αυτή.

4)*Remark*(stop-the-world): Ολοκληρώνει το μαρκάρισμα των ζωντανών αντικειμένων στο heap με το snapshot-at-the-beginning αλγόριθμο.

5)*Cleanup*(stop-the-world & concurrent): Απελευθερώνει τις περιοχές της μνήμης με σκουπίδια.

6)*Copying*(stop-the-world): μετακίνηση ή αντιγραφή αντικειμένων σε περιοχές που δεν έχουν χρησιμοποιηθεί.

*Προβληματισμός: Τα βήματα 1,3,4,5,6 στο πρόγραμμα μας συμβαίνουν κατα το young gc!*

Τα 5 βήματα αυτά παρουσιάζονται στο documentation της oracle σαν κομμάτι του old gc. Παρόλα αυτά στα logs του προγράμματος μου, καταλαβαίνω πως ανήκουν στο young collection.

**Ζευγάρι προγραμμάτων: αργή java/ γρηγορη c++.**

Τα 2 προγράμματα είναι ισοδύναμα: Δημιουργούν 250εκ αντικείμενα και τα προσθέτουν σε μια custom doubly linked list. Στο πρόγραμμα της java έγινε δέσμευση 8gb για το heap με την εντολή ‘-Xmx’. Ο χρόνος ολοκλήρωσης της java ήταν 41 sec, ενώ της c++ ειναι 11.8 sec. Η java εκτιμώ ότι είναι πιο αργή από την c++ διότι ο G1 garbage collector καλείται αρκετά συχνα και δουλεύει concurrent με το πρόγραμμα. Αυτό σημαίνει ότι διακόπτεται η εκτέλεση του προγράμματος όταν καλείται ο garbage collector, ώστε να κάνει marking, evacuate και compact αντικειμένων μέσα στο heap.  
Αυτό που παρατήρησα στην java είναι ότι γίνεται συνεχώς normal young gc(**minor collection**) και κανένα full collection.

**Ζευγάρι προγραμμάτων: γρήγορη java/ αργη c++.**

Σε αυτό το ζευγάρι δημιουργούνται 200εκ. αντικείμενα τα οποία προστίθενται σε μία custom doubly linked list. Στη συνέχεια διαγράφουμε το 90% των αντικειμένων της λίστας, και ανακατεύουμε τους δείκτες τους.Τέλος διατρέχουμε τη λίστα 5 φορές. Η μία φορά ισοδυναμεί με το ξεκίνημα από την αρχή, να πηγαίνει στο τέλος και να καταλήγει πάλι στην αρχή. Στόχος είναι να δημιουργήσουμε κακό locality στην c++, ωστε η διάσχιση των κόμβων της λίστας να καθυστερεί. Στην java για την οργάνωση της μνήμης θα φροντίσει ο garbage collector και έτσι θα δούμε ότι θα είναι σημαντικά γρηγορότερη.

Η java χρειαζεται 40 sec για να ολοκληρωθεί, ενώ της c++ περίπου 27 sec. Γνωρίζω ότι δεν ήταν το επιθυμητό αποτέλεσμα αυτό, αλλά μόνο αν τα συγκρίνουμε με βάση την εκτέλεση σαν σύνολο, από την αρχή μέχρι το τέλος. Η διαδικασία προσθήκης των κόμβων στη λίστα της java είναι 3 φορές πιο αργή από τη c++, λόγω των συνεχών collection του garbage collector. Το σημείο όμως που έχει σημασία, δηλαδή κατα την διάσχιση της λίστας είναι σημαντικά πιο γρήγορο στη java (2.2 sec) σε σχέση με την c++ (12sec), το οποίο θεωρώ καλύπτει το ζητούμενο.

**Δεύτερο σκέλος εργασίας:**

Δοκιμάζοντας τις ίδιες εντολές και στα 2 προγράμματα σε java παρατήρησα ότι αντιδρούν με όμοιο τρόπο στις παραμετροποιήσεις του garbage collector. Οπότε θα αναφέρω τα αποτελέσματα της αργής java.

* Μέγεθος παλιάς και νέας γενιάς

Young generation: –XX:NewRatio=n, όπου n <integer>.

Για n=3 π.χ, έχουμε το ¼ του heap size για eden και survivor περιοχές, δηλαδή το young generation. Με τα default settings του G1 θα δούμε ότι στα 40 sec που διήρκησε το πρόγραμμα καλέστηκε μία φορά ένας **concurrent cycle**. Αυτό εμπεριέχει τις φάσεις του remark και clean-up και αφορά τη μετανάστευση των αντικειμένων στο old generation.

Την καλύτερη απόδοση την παρατήρησα για n=2, κατα την οποία το προγραμμα χρειάστηκε 28.5 sec για να ολοκληρωθει, από τα 40 sec με τα default settings. Στο documentation της oracle για τον G1, αναφέρεται πως όσο μεγαλύτερη είναι η young generation τόσο σπανιότερα γίνονται και τα minor collections. Στη περίπτωση μας βλέπουμε ότι μεγαλώνοντας την young generation έγιναν λιγότερα garbage collections, αποτελώντας το λόγο που μειώθηκε ο συνολικός χρόνος εκτέλεσης του προγράμματος.

Στα logs του G1 ενα minor collection αποτυπώνεται με την εξής μορφή: GC(22) Pause Young (Normal) (G1 Evacuation Pause) 3224M->3229M(6922M) 2027.024ms. 3224M->3229M.

**“3224M->3229M”**: Αυτό δείχνει τη κατειλλημένη μνήμη του heap πριν και μετά το collection. Είναι λογικό να μην μειώνεται αφού τα αντικείμενα δεν πεθαίνουν στο πρόγραμμα.

* πότε ένα αντικείμενο πηγαίνει στην παλαιά γενιά (tenuring)

Ένα αντικείμενο μετακινείται στη παλαιά γενιά οταν το survivor region είναι γεμάτο.

Επίσης με τη παράμετρο -XX:MaxTenuringThreshold =n (default: 15), μπορώ να ορίσω τον μέγιστο αριθμό των iterations πάνω στο αντικείμενο από τα gc collections μέχρι αυτό να μετακινηθεί στην παλαιά γενιά. Αν ένα αντικείμενο δεν χρειάζεται να μείνει στη young generation, τότε αξίζει να μικρύνεις το threshold.

Για n=2, 5 και 15 και αλλάζοντας και το young generation ratio δεν παρατήρησα κάποια διαφορά στον αριθμό των φορών που κλήθηκε ο garbage collector, καθώς ούτε και στον συνολικό χρόνο εκτέλεσης του προγράμματος. Για n=0 όμως, ο συνολικός χρόνος έπεσε στα 34.8 sec και οι συνολικός αριθμός κλήσεων των gc collections στις 29 (απο 41 sec και 31 αντίστοιχα με default settings) . Αυτό το αποδίδω στο ότι τα αντικείμενα παραμένουν ζωντανά για αρκετή ώρα επειδή ανήκουν στη λίστα, οπότε η μετανάστευση στην old generation νωρίς επιτρέπει στον gc να μην ασχολείται με αυτά σε κάθε minor collection.

* αν ο GC είναι παράλληλος ή όχι, καθώς και το πόσα threads έχει

Ο G1 είναι παράλληλος και χρησιμοποιεί 10 threads (default). Αυτό φαίνεται βάζοντας τη παράμετρο -XX:+PrintGCDetails, η οποία δείχνει τα logs του G1 με περισσότερη πληροφορία σε σχέση με το -verbose:gc.

Η παράμετρος : -XX:ParallelGCThreads=n, (n: int). Δίνει το μέγιστο αριθμό νημάτων (threads) για παράλληλη επεξεργασία κατά τη διάρκεια των παύσεων για garbage collection.

Για n=1, με τα υπόλοιπα settings default, ο χρόνος μειώθηκε στα 24 sec.

Για n=3, κατέβηκε περισσότερο, στα 19sec. Αυτό δείχνει πως για το μέγεθος της δουλειάς που έχει να διαχειριστεί ο G1, η παραλληλία από ένα σημείο και μετά προσθέτει overhead χωρίς να προσφέρει κάποιο όφελος.

Ο χρόνος εκτέλεσης πέφτει στα 14.5 sec αν βαλουμε 3 threads και θέσουμε και το -ΧΧ:ΝewRatio = 3.

* αν ο gc τρέχει σε ξεχωριστό thread παράλληλα με το πρόγραμμα.

O G1 έχει και concurrent phases(τρέχει παράλληλα με τα νήματα του προγράμματος - εναλλάξ το πρόγραμμα και ο gc), αλλά και parallel phases (αφορά το gc μόνο, ανεξάρτητα αν είναι concurrent με το πρόγραμμα ή όχι).

Το global marking phase εκτελείται παράλληλα και ταυτόχρονα(parallel and concurrent) με το πρόγραμμα.

To young generation collection γίνεται παράλληλα με το πρόγραμμα(concurrent) και με πολλά νήματα (parallel), αλλά σταματώντας ολα τα νήματα του προγράμματος.

To old region collection γίνεται σε ένα νήμα αποκλειστικά, και σταματάει όλα τα νήματα του προγράμματος.

* αν ο gc(παλαιάς γενιάς) είναι mark-sweep ή compacting.

O G1 είναι compacting garbage collector. Και κάνει whole-heap compactions.

* αν ο gc(παλαιάς γενιάς) είναι concurrent.

Είναι concurrent, σύμφωνα με το documentation της oracle.