Κωδικός Εργασίας (2.1)

**Ομάδα** [13]

**Μέλος A**: [Γεώργιος,Σιδέρης,1622 ,sideris@uth.gr]

**Μέλος B**: [Φώτιος, Τσώκος, 1679, tsokos@inf.uth.gr]

1. Περιγραφή σημείων συγχρονισμού με ψευτοκώδικα

|  |
| --- |
| **1\_1Ενεργή αναμονή και καθυστερηση/επίλυση με mutexes**  Όπως δείξαμε και στην εργασία 1.2, η επίλυση του προγράμματος με τη χρήση ενεργής αναμονής, παρά την αποτελεσματικότητα της, παρουσίαζε ένα μεγάλο μειονέκτημα: αρκετά μεγαλύτερο χρόνο εκτέλεσης από το πρωτότυπο πρόγραμμα. Όπως εξηγήσαμε το γεγονός αυτό ήταν αναμενόμενο από την “σπατάλη” της επεξεργαστικής ισχύος σε ατέρμονα loops. Μια προσπάθεια επίλυσης με Mutexes σαφώς θα επιδείκνυε καλύτερο χρόνο εκτέλεσης και ευκολότερη διαχείριση των νημάτων, καθώς διασφαλίζουμε την αποκλειστική εκτέλεση του κρίσιμου τμήματος, οποιαδήποτε χρονική στιγμή, από ένα μόνο νήμα και την εξάλειψη της ανάγκης για ενεργή αναμονή.  **1\_2 Σημεία συντονισμού και διαχείριση τους**  Στη προηγούμενη εργασία αναφέραμε δύο “κρίσιμα” σημεία του κώδικα το  Cs1, όπου γίνεται η αρχικοποίηση των μεταβλητών μας για κάθε slice και το Cs2, όπου υπολογίζονται οι παράμετροι σχεδιασμού του fractal. Απαραίτητη προϋπόθεση για τον υπολογισμό του Cs2 είναι κάθε thread να έχει ολοκληρώσει το Cs1. Βασισμένοι στο παραπάνω υλοποιήσαμε την εξής δομή επίλυσης :  1) Κάθε thread εκτελεί το Cs1   1. Αναμονή μέχρι να ολοκληρώσουν όλα την εκτέλεση τους 2. Υπολογισμός του Cs2 και σχεδίαση του fractal   Παρακάτω παραθέτουμε μια ενδεικτική υλοποίηση της λύσης μας σε ψευτοκώδικα |
| Σημ.: Το mutex mtxCS2 αρχικοποιείται ως locked, αντίθετα το mtxCS1 (ώστε το πρώτο νήμα να μπει στο CS1 και να σταματήσει στον CS2). Η χρήση της μεταβλητής progress παίζει το σημαντικότερο ρόλο στον συντονισμό των threads, θα αναλυθεί παρακάτω  Void \* CalcMandel(){  initialize variables;  while (1){ //in order to use thread again after zoom  lock (mtxCS1);  if (level>1){initialize variables} //after first use initialize again  CS1  progress=(++progress)%numberOfSlices;  if (progress!=0){ unlock (mtxCS1); }//let next thread execute CS1  else { unlock (mtxCS2);}  // if all threads are ready let them execute CS2  lock (mtxCS2);  progress=(++progress)%TOOL->mand->slices;  CS2  if (progress!=0){unlock(mtxCS2);}//if not all are finished unlock next  else {unlock (mtxReady);}  // if all are finished it unlocks mtxReady, letting main draw the results  }  }  int main{  readJobParameters();  while (1){  unlock (mtxCS1);  //which executes calcMandel();  lock (mtxReady);  draw Results  readJobParameters();  }  } |
| **1\_3 Μεταβλητή Progress και συγχρονισμός**  Η βασική λειτουργία της progress είναι να καταγράφει πόσα νήματα έχουν ολοκληρώσει την εργασία τους σε κάθε στιγμιότυπο εκτέλεσης του προγράμματος. Όταν όλα τα νήματα τελειώσουν μια εργασία τότε η progress θα ξαναγίνει 0(Για οποιονδήποτε θετικό ακέραιο Ν ισχύει (kN)modN=0, για k θετικό ακέραιο αριθμό). Έτσι μπορούμε να παρακολουθούμε την πρόοδο της εκτέλεσης του προγράμματος μας, μέσω της progress, και να προβαίνουμε στις κατάλληλες ενέργειες όταν είναι απαραίτητο (πχ όταν progress=0 από την CS1 πάμε στην CS2, ενώ όταν μέσα στη CS2 progress=0 από την CS2 πάμε στον σχεδιασμό του fractal). Με άλλα λόγια η progress διασφαλίζει την σωστή εναλλαγή των διεργασιών, δηλαδή “συγχρονίζει” το πρόγραμμα μας. |
| **1\_4 Γιατί δουλεύει;**  **1)Αμοιβαίος αποκλεισμός:** Τα mutexes διασφαλίζουν ότι όταν νήμα εκτελεί το κρίσιμο τμήμα (KT για συντομία) κανένα άλλο δεν θα εισέλθει μέσα σε αυτό.  **2)Απουσία αδιεξόδου:** Τα νήματα εκτελούν μια διεργασία και στη συνέχεια περιμένουν τα άλλα νήματα να τελειώσουν. Ουσιαστικά κανένα νήμα δεν εμποδίζει άλλα νήματα να μπουν στο KT και αυτό επιτυγχάνεται πάλι με την χρήση των mutexes.  **3)Απουσία λιμοκτονίας:** Όταν ένα νήμα επιθυμεί να μπει στο ΚΤ κάποια στιγμή θα μπει. Αυτό ισχύει διότι (1) ο αριθμός των νημάτων είναι ίσος με τον αριθμό των slices, (2) το πρόγραμμα περιμένει κάθε φορα #slices εκτέλεσης του κάθε κρίσιμου σημείου και (3)κάθε νήμα εκτελεί μία διεργασία τη φορά. Από τον συνδυασμό των (1),(2),(3) προκύπτει ότι κάθε νήμα θα εκτελεστεί εγγυημένα όποτε χρειαστεί. Για παράδειγμα, ας υποθέσουμε ότι είμαστε στην εκτέλεση του n-οστού νήματος και το n+1 νήμα αναμένει να εκτελεστεί τότε: από (1) και (3) ξέρουμε ότι θα είναι αυτό το νήμα που θα εκτελέσει τη διεργασία που του αναλογεί και όχι κάποιο άλλο, εφόσων ο αριθμός των slices είναι ίσος με τον αριθμό των νημάτων και κανένα άλλο νήμα δεν θα εκτελέσει τη εργασία του (απο (3)), και από το (2) γνωρίζουμε ότι τα άλλα νήματα περιμένουν το n+1 νήμα να τερματίσει κάποια στιγμή για να εκτελεστούν. Από τα παραπάνω προκύπτει ότι κάθε νήμα θα εκτελεστεί και εξασφαλίζεται η απουσία λιμοκτονίας.  **Πρόοδος:** Από τα παραπάνω διασφαλίζουμε τη πρόοδο του προγράμματος και πιο συγκεκριμένα γνωρίζουμε ότι από μία κατάσταση Φ(i) θα φτάσει εγγυημένα στην επιθυμητή κατάσταση Φ(j)(progress properties) χωρίς να εισέλθει ποτέ σε κάποια ανεπιθύμητη κατάσταση (safety properties). |

1. Σχολιασμός φαινομένων / αποτελεσμάτων

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2\_1 Αποτελέσματα Μέτρησης Απόδοσης**  Παρακάτω παραθέτω τον χρόνο εκτέλεσης του προγράμματος 1.2 καθώς και του 2.1 για τις ίδιες ακριβώς τιμές (max Iteractions=30)   |  |  |  |  | | --- | --- | --- | --- | | Program | NUMBER | OF | SLICES | |  | 1 | 10 | 100 | | Project 1.2 | 12.467sec | 18.424sec | 38.945sec | | Project 1.2(with thread\_yield) | 10.488sec | 18.592sec | 40.536sec | | Project 2.1 (with mutexes) | 8.384sec | 8.736sec | 10.257sec | |
| **2\_2 Αιτιολόγηση αποτελέσματος**  Το πρόγραμμα εκτελείται χωρίς ενεργή αναμονή, επομένως η μεγάλη διαφορά στη σύγκριση με τις δύο εκδοχές του project 1.2 ήταν αναμενόμενη για τους λόγους που εξηγήσαμε στο 1\_1. |