

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΝΑΦΟΡΑ 2^{ης} ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: oslab005
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 15.05.2024

▪ Ενότητα 1 – Συγχρονισμός σε υπάρχοντα κώδικα

Εισαγωγικές ερωτήσεις

Χρησιμοποιούμε το διαθέσιμο Makefile, προκειμένου να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα simplesync. Παρατηρούμε, ότι ενώ έχουμε αρχικοποιήσει μία μεταβλητή (int val) στην τιμή 0 και την αυξάνουμε κατά 1 όσες φορές τη μειώνουμε κατά 1 – 10.000.000 φορές – το τελικό αποτέλεσμα που εμφανίζεται στην οθόνη δεν είναι 0, όπως αναμενόταν. Η εξήγηση για αυτό είναι ότι οι διεργασίες που αυξάνουν και μειώνουν την τιμή της μεταβλητής εκτελούνται ταυτόχρονα – και μάλιστα το κρίσιμο τμήμα τους που κάνει (val++) ή (val--) – δεν είναι συγχρονισμένες, με αποτέλεσμα να μοιράζονται κοινή μνήμη και να αντλούν / αποθηκεύουν την νέα τιμή της σε λάθος στιγμές: πριν ολοκληρωθεί και αποθηκευτεί η αύξηση έχει εκτελεστεί αφαίρεση ή αντιστρόφως.

Ακόμα, παρατηρούμε ότι παράγονται δύο διαφορετικά εκτελέσιμα από το simplesync.c : το simplesync-atomic και το simplesync-mutex, μέσω του Makefile. Ο λόγος που συμβαίνει αυτό είναι αφενός μεν τα #defined που έχουμε κάνει μέσα στο simplesync.c και φαίνονται παρακάτω:

```
/* Dots indicate lines where you are free to insert code at will */
/* ... */
pthread_mutex_t lock;
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
# error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif
```

αφετέρου δε, τα flags κατά τη μεταγλώττιση στο Makefile (-DSYNC_MUTEX και -DSYNC_ATOMIC), τα οποία δίνουν τιμές 1 στις mutex και atomic παραλλαγές της simplesync, αντίστοιχα.

Κατόπιν, επεκτείνουμε τον κώδικα της simplesync.c στις δύο παραλλαγές της και το νέο αρχείο .c φαίνεται ακολούθως. Οι αλλαγές εντοπίζονται στην __sync_add_and_fetch και __sync_sub_and_fetch που εξασφαλίζουν το atomicity των προσθέσεων και αφαιρέσεων, αντίστοιχα, και στα mutex lock/unlocks στο κρίσιμο τμήμα της αύξησης/μείωσης της μεταβλητής val.

Η αλλαγμένη συνάρτηση που υλοποιεί την αύξηση φαίνεται ακολούθως, ενώ όμοια λογική υπάρχει και στη συνάρτηση που υλοποιεί τη μείωση της μεταβλητής val.

```
41 void *increase_fn(void *arg)
42 {
43     int i;
44     volatile int *ip = arg;
45
46     fprintf(stderr, "About to increase variable %d times\n", N);
47     for (i = 0; i < N; i++) {
48         if (USE_ATOMIC_OPS) {
49             __sync_add_and_fetch(ip,1);
50         } else {
51             pthread_mutex_lock(&lock);
52             /* You cannot modify the following line */
53             ++(*ip);
54             pthread_mutex_unlock(&lock);
55         }
56     }
57     fprintf(stderr, "Done increasing variable.\n");
58
59     return NULL;
60 }
```

Ο συνολικός – αλλαγμένος κώδικας φαίνεται ακολούθως:

Το αποτέλεσμα, όπως φαίνεται και παρακάτω, είναι 0, όπως πρέπει:

```
peppasmich@Michael-Peppas:/mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/κώδικες Ασκήσεων/askhsh2$ ./simplesync-atomic  
About to increase variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
OK, val = 0.  
peppasmich@Michael-Peppas:/mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/κώδικες Ασκήσεων/askhsh2$ ./simplesync-mutex  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done decreasing variable.  
Done increasing variable.  
OK, val = 0.
```

Αξίζει να σημειωθεί πως το πρόγραμμά μας, όπως φαίνεται και από τον κώδικα που παραθέσαμε, ελέγχει την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνει το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

Ερωτήσεις

1. Αρχικά, τρέχουμε με χρήση της εντολής `time` τη `simplesync-atomic`, η οποία έχει παραχθεί από τον αρχικό κώδικα, χωρίς παραλλαγές, δηλαδή χωρίς συγχρονισμό. Το αποτέλεσμα φαίνεται ακολούθως:

```
peppasmich@Michael-Peppas: /mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλίο - Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-atomic  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
NOT OK, val = 1229810.  
  
real    0m0.030s  
user    0m0.041s  
sys     0m0.000s
```

Όπως εύκολα γίνεται αντιληπτό, το αποτέλεσμα είναι λανθασμένο, αφού δεν υπάρχει συγχρονισμός. Ακολούθως, τρέχουμε τους σωστούς κώδικες `simplesync-atomic` και `simplesync-mutex`, αντίστοιχα, και το αποτέλεσμα φαίνεται ακολούθως:

```
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-atomic  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done decreasing variable.  
Done increasing variable.  
OK, val = 0.  
  
real    0m0.264s  
user    0m0.501s  
sys     0m0.000s  
peppasmich@Michael-Peppas: /mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-mutex  
About to increase variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
OK, val = 0.  
  
real    0m1.105s  
user    0m1.504s  
sys     0m0.676s
```

Παρατηρούμε, όπως εξάλλου ήταν και αναμενόμενο, ότι ο χρόνος εκτέλεσης του ασυγχρόνιστου προγράμματος είναι πολύ μικρότερος και από τα δύο προγράμματα που εκτελούν συγχρονισμό και εμφανίζουν – φυσικά – το σωστό αποτέλεσμα. Ο λόγος για αυτό είναι ότι στο αρχικό πρόγραμμα εκτελούνται όλες οι προσθέσεις και οι αφαιρέσεις ταυτόχρονα, χωρίς έτσι να παράγεται σωστό αποτέλεσμα, αφού χάνεται η σωστή αλληλουχία τους και χρησιμοποιούν κοινή μνήμη, ενώ στις άλλες περιπτώσεις εκτελούνται μόνο όταν το κρίσιμο τμήμα της προηγούμενης έχει ολοκληρωθεί, υστερώντας έτσι σε χρόνο, εμφανίζοντας όμως το σωστό αποτέλεσμα.

2. Όπως φαίνεται και από την παραπάνω εικόνα, η μέθοδος των `atomic operations` είναι κατά πολύ (περίπου 5 φορές) γρηγορότερη από τη μέθοδο των `mutexes`. Ο λόγος πίσω από αυτό είναι ότι στην περίπτωση των `atomic operations` οι προσθέσεις και οι αφαιρέσεις μπορούν να τρέχουν παράλληλα – ατομικά – και δεν επικαλύπτεται μόνο το κρίσιμο (σε `assembly`) κομμάτι τους ενώ το μη κρίσιμο εκτελείται ταυτοχρόνως, ενώ στην περίπτωση των `mutexes`, έχουμε κλειδώσει όλο το κομμάτι των προσθέσεων και των αφαιρέσεων, με αποτέλεσμα αυτές να εκτελούνται σειριακά, μία κάθε φορά.

3. Αλλάζουμε το Makefile, προσθέτοντας τα flags -S και -g, ώστε να παραγάγουμε τον αναγνώσιμο κώδικα των αρχείων μας σε assembly, ως εξής:

```
Makefile
1  #
2  # Makefile
3  #
4
5  CC = gcc
6
7  # CAUTION: Always use '-pthread' when compiling POSIX threads-based
8  # applications, instead of linking with "-lpthread" directly.
9  CFLAGS = -Wall -O2 -pthread -S -g
```

Κατόπιν, ανοίγουμε το simplesync-atomic.o και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα atomic operations, το οποίο φαίνεται ακολούθως:

```
65  .LBE15:
66      .loc 1 48 3 is_stmt 1 view .LVU16
67      .loc 1 49 4 view .LVU17
68      lock addl    $1, (%rbx)
69      .loc 1 47 22 discriminator 2 view .LVU18
70      .loc 1 47 16 discriminator 1 view .LVU19
71      .loc 1 48 3 view .LVU20
72      .loc 1 49 4 view .LVU21
73      lock addl    $1, (%rbx)
74      .loc 1 47 22 discriminator 2 view .LVU22
75      .loc 1 47 16 discriminator 1 view .LVU23
76      subl    $2, %eax
77      jne .L2
78      .loc 1 57 2 view .LVU24
```

```
164  .LBE21:
165      .loc 1 69 3 is_stmt 1 view .LVU48
166      .loc 1 70 4 view .LVU49
167      lock subl    $1, (%rbx)
168      .loc 1 68 22 discriminator 2 view .LVU50
169      .loc 1 68 16 discriminator 1 view .LVU51
170      .loc 1 69 3 view .LVU52
171      .loc 1 70 4 view .LVU53
172      lock subl    $1, (%rbx)
173      .loc 1 68 22 discriminator 2 view .LVU54
174      .loc 1 68 16 discriminator 1 view .LVU55
175      subl    $2, %eax
176      jne .L8
177      .loc 1 78 2 view .LVU56
```

4. Ομοίως, ανοίγουμε το `simplesync-mutex.o` και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα `pthread_mutex_lock` / `pthread_mutex_unlock`, το οποίο φαίνεται ακολούθως:

```
76  .L2:
77      .loc 1 48 3 view .LVU17
78      .loc 1 51 4 view .LVU18
79      movq    %r12, %rdi
80      call    pthread_mutex_lock@PLT
81  .LVL4:
82      .loc 1 53 4 view .LVU19
83      .loc 1 53 7 is_stmt 0 view .LVU20
84      movl    0(%rbp), %eax
85      .loc 1 54 4 view .LVU21
86      movq    %r12, %rdi
87      .loc 1 53 4 view .LVU22
88      addl    $1, %eax
89      movl    %eax, 0(%rbp)
90      .loc 1 54 4 is_stmt 1 view .LVU23
91      call    pthread_mutex_unlock@PLT
```

▪ Ενότητα 2 – Παράλληλος υπολογισμός του συνόλου Mandelbrot

Εισαγωγή

Σημειώνουμε πως αμφότερα τα προγράμματά μας (mandel_sema.c και mandel_cond.c) ελέγχουν την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνουν το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

1. Αρχικά, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα mandel_sema.c και υλοποιεί το ζητούμενο με τη χρήση σημαφόρων. Το struct του καθενός thread φαίνεται ακολούθως και περιέχει το id του, τον συνολικό αριθμό των νημάτων και έναν πίνακα στον οποίο αποθηκεύονται τα δεδομένα του compute του mandel_line και αργότερα απεικονίζονται. Το ίδιο struct χρησιμοποιείται και για το mandel_cond.c στο επόμενο ερώτημα και για τον λόγο αυτό δεν θα παρατεθεί ξανά.

```
65  struct thread_info_struct {
66      pthread_t tid; /* POSIX thread id, as returned by the library */
67
68      int *color_val; /* Pointer to array to manipulate */
69      int thrid; /* Application-defined thread id */
70      int thrcnt;
71  };
```

Η λογική πίσω από τον κώδικά μας είναι η εξής: εάν έχουμε n threads να τρέχουν το mandel, καθένα από αυτά έχει τον δικό του σημαφόρο (έχουμε δηλαδή έναν πίνακα από σημαφόρους, τον sem) αρχικοποιημένο σε 0, εκτός από το 1^ο thread, του οποίου είναι αρχικοποιημένος σε 1. Έτσι, αφού το 1^ο thread εκτελέσει το κρίσιμο τμήμα του, αυτό θα ξυπνήσει το 2^ο, το 2^ο το 3^ο κ.ο.κ., μέσω της sem_post του αντίστοιχου σημαφόρου. Επίσης, έχουμε χειριστεί το πρόγραμμά μας έτσι ώστε το κρίσιμο τμήμα να περιλαμβάνει μόνο τη φάση του output και όχι και αυτή του compute, για λόγους επίδοσης που θα σχολιαστούν αργότερα, στα αντίστοιχα ζητούμενα ερωτήματα, και άρα τα computes γίνονται ταυτόχρονα και αποθηκεύονται στον ξεχωριστό πίνακα του κάθε thread. Ο κώδικας αυτός φαίνεται ακολούθως:

2. Για το ερώτημα αυτό, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα `mandel_cond.c` και υλοποιεί το ζητούμενο με τη χρήση `condition variables`. Επισημαίνουμε πως το κάθε `thread` χρησιμοποιεί το `struct` που παρατέθηκε προηγουμένως. Η λογική αυτού του προγράμματος είναι η εξής: έχουμε n `threads`, τα οποία πάλι (για λόγους επίδοσης) τρέχουν παράλληλα το `compute` κάθε γραμμής που έχουν αναλάβει και έχουν ως κρίσιμο τμήμα μόνο αυτό της απεικόνισης μιας γραμμής, πριν το οποίο περιμένουν σε μια κοινή κελιδαριά. Έτσι, όντας συνδεδεμένα σε μια κοινή κλειδαριά (`lock`), έχουμε έναν πίνακα από `conditions` (`cond`), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή, αρχικά τρέχει μόνο το υπ' αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει `broadcast` στο επόμενο (στο αντίστοιχο `condition`) για να ξεκινήσει. Έτσι, το `mandel` απεικονίζεται επιτυχώς στην έξοδο, με τη βέλτιστη επίδοση.

Αξίζει να επισημάνουμε ότι τα `threads` μοιράζονται τις `global` μεταβλητές τους. Για τον λόγο αυτό, προτιμήσαμε κάθε `thread` να κάνει `malloc` έναν δικό του πίνακα, ώστε να μπορούν όλα μαζί – παράλληλα – να υπολογίζουν την έξοδο μίας γραμμής, περιορίζοντας έτσι το κρίσιμο τμήμα των `threads` μόνο στη διαδικασία του `output`, επιτυγχάνοντας βέλτιστη επίδοση. Αυτό, γίνεται φανερό στον τρόπο που αρχικοποιούμε το `struct` κάθε `thread` και στο σώμα της συνάρτησης που αυτό καλεί.

Ο κώδικας του `mandel_cond.c` φαίνεται ακολούθως:

Ερωτήσεις

1. Για το σύστημα συγχρονισμού που υλοποιήσαμε, χρειαστήκαμε αριθμό σημαφόρων ίσο με τον αριθμό των threads που τρέχουν συγχρόνως και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ειδικότερα, χρησιμοποιήσαμε έναν πίνακα από σημαφόρους (τον sem), ο οποίος είναι αρχικοποιημένος στο 0, εκτός από την 1^η θέση του, που είναι αρχικοποιημένη στο 1. Έτσι, θα εκτελεστεί μόνο το 1^ο thread (αφού τα άλλα θα περιμένουν) και τα επόμενα θα καλούνται διαδοχικά, καθώς το προηγούμενο θα κάνει sem_post το επόμενο. Αυτό γίνεται φανερό στον κώδικα mandel_sema.c που παραθέσαμε παραπάνω.
2. Αρχικά, χρησιμοποιούμε την εντολή cat /proc/cpuinfo και διαπιστώνουμε ότι το μηχανήμα μας διαθέτει 4 πυρήνες (σε αντίθεση με τον orion που διαπιστώσαμε ότι έχει μόνο 1), καθιστώντας τη δοκιμή πολλών threads εφικτή. Αυτό, φαίνεται ακολούθως:

```
processor       : 7
vendor_id      : GenuineIntel
cpu family     : 6
model          : 158
model name     : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping       : 9
microcode      : 0xffffffff
cpu MHz        : 2807.998
cache size     : 6144 KB
physical id    : 0
siblings       : 8
core id        : 3
cpu cores      : 4
apicid         : 7
initial apicid : 7
fpu            : yes
fpu_exception  : yes
cpuid level    : 21
wp             : yes
```

Επομένως, τρέχουμε το mandel πρώτα και το mandel_sema μετά, με 2 threads και με κατάλληλη χρήση της time, και το αποτέλεσμα φαίνεται ακολούθως:

real	0m0.549s	real	0m0.307s
user	0m0.477s	user	0m0.547s
sys	0m0.070s	sys	0m0.051s

Όπως ήταν αναμενόμενο, η εκτέλεση του mandel διήρκησε πολύ περισσότερο (σχεδόν τον διπλάσιο πραγματικό χρόνο) από αυτή του mandel_sema με 2 threads, αφού η διαδικασία των computes γίνεται πλέον παράλληλα από τα 2 threads και έχουμε απομονώσει μόνο το πραγματικό κρίσιμο τμήμα τους. Παρατηρούμε, έτσι, μια μεγάλη βελτίωση στην επίδοση του προγράμματός μας.

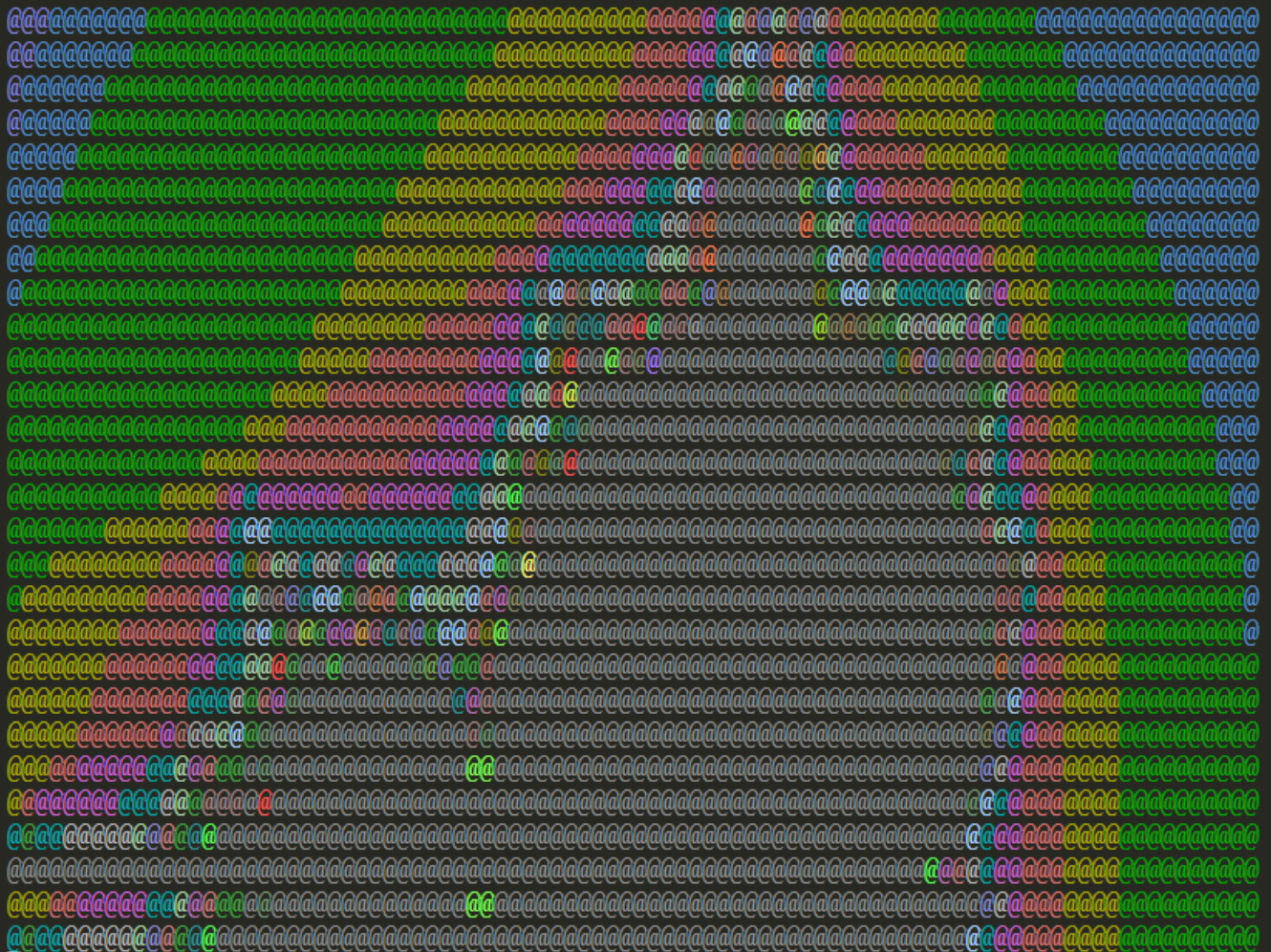
3. Όπως φαίνεται και στον κώδικα `mandel_cond.c` που παραθέσαμε παραπάνω, έχουμε χρησιμοποιήσει αριθμό `condition variables` ίσο με τον αριθμό των `threads` που χρησιμοποιούμε και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ο λόγος για αυτό είναι ο εξής: τα `threads` είναι συνδεδεμένα σε μια κοινή κλειδαριά (`lock`) και έχουμε έναν πίνακα από `conditions` (`cond`), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή (`counter`), αρχικά τρέχει μόνο το υπ' αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει `broadcast` στο επόμενο (στο αντίστοιχο δηλαδή `condition`) για να ξεκινήσει. Έτσι, τα `computes` γίνονται παράλληλα και τα `outputs` με τη σειρά που πρέπει, επιτυγχάνοντας βέλτιστη επίδοση.

Έστω, τώρα, ότι χρησιμοποιούμε μόνο ένα `condition variable`, έστω `cond` και ότι – χωρίς βλάβη της γενικότητας – έχουμε 2 `threads`. Στην περίπτωση αυτή, θα δημιουργηθεί το εξής πρόβλημα συγχρονισμού: αρχικά, θα τρέξει το 1^ο `thread` τη διαδικασία του `compute` του, όπως και το 2^ο, παράλληλα, ενώ στην συνέχεια, το 1^ο θα τρέξει και το κρίσιμο τμήμα του (`output`), ενώ το 2^ο θα κάνει `wait`, περιμένοντας το 1^ο να τελειώσει και να κάνει `broadcast` στο `cond`, ώστε να ξεκινήσει κι αυτό. Όμως, μπορεί το 1^ο `thread` να τελειώσει το κρίσιμο τμήμα του και να κάνει `broadcast` πριν το 2^ο κάνει `wait` και αυτό θα έχει ως αποτέλεσμα το 2^ο να κάνει `wait` για πάντα, αφού το 1^ο δεν θα ξανακάνει `broadcast` ποτέ. Άρα, το 2^ο `thread` δεν θα κάνει ποτέ `output` και το πρόγραμμά μας θα κολλήσει στο σημείο αυτό. Αντίστοιχο πρόβλημα θα παρατηρηθεί και εάν έχουμε παραπάνω από 2 `threads` σε κοινό `cond`.

Το πρόβλημα αυτό, αποτυπώνεται κάτωθι, όπου παραθέτουμε τον αλλαγμένο κώδικα στο σημείο της κλήσης της συνάρτησης από το κάθε `thread` και το αντίστοιχο `output`, το οποίο παραμένει παγωμένο για πάντα:

```
168 void *compute_and_output_mandel_line(void *arg)
169 {
170     struct thread_info_struct *thr = arg;
171
172     for(int line = thr->thrid; line < y_chars; line += thr->thrcnt) {
173         compute_mandel_line(line, thr->color_val);
174         pthread_mutex_lock(&lock);
175         if(line != counter) pthread_cond_wait(&cond[0], &lock);
176         counter++;
177         output_mandel_line(1, thr->color_val);
178         pthread_cond_broadcast(&cond[0]);
179         pthread_mutex_unlock(&lock);
180     }
181     return NULL;
182 }
```

κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2\$./mandel_cond 6



4. Για τους λόγους που ήδη έχουμε αναφέρει, αμφότερα τα προγράμματα που υλοποιήσαμε για το mandel με πολλαπλά threads, εμφανίζουν σημαντική επιτάχυνση, η οποία (έπειτα από δοκιμές) είναι ολοένα και μεγαλύτερη αυξανομένου του αριθμού των νημάτων, αν και όχι με τον ίδιο ρυθμό, δηλαδή από ένα σημείο threads και πάνω (που φυσικά εξαρτάται από τον αριθμό των πυρήνων του μηχανήματος) η μείωση στον χρόνο είναι μικρή, έως και αμελητέα. Ο λόγος πίσω από αυτό είναι ο εξής: στην περίπτωση που ορίσουμε ως κρίσιμο τμήμα σε κάθε thread τις φάσεις τόσο του υπολογισμού όσο και της εμφάνισης μια γραμμής, αυτό θα έχει ως αποτέλεσμα να τρέχει μόνο ένα νήμα κάθε φορά και η όλη διαδικασία να εκτελείται σειριακά, όπως στο αρχικό mandel! Ένα όμως φροντίσουμε να έχουμε ξεχωριστές δομές δεδομένων για κάθε νήμα (όπως και κάναμε), τότε η φάση του compute – που όπως τελικά διαπιστώνουμε είναι αρκετά χρονοβόρα – δύναται να βγει εκτός του κρίσιμου τμήματος και σε αυτό να παραμείνει μόνο η φάση της αποτύπωσης. Έτσι, ο χρόνος εκτέλεσης μειώνεται αισθητά και γίνεται πραγματική αξιοποίηση του μεγάλου αριθμού νημάτων.

Για να γίνει αυτό φανερό, αλλάζουμε το κρίσιμο τμήμα (έστω του `mandel_sema.c`), ώστε αυτό να περιλαμβάνει αμφότερες τις φάσεις που περιγράψαμε, ως εξής:

```
166 void *compute_and_output_mandel_line(void *arg)
167 {
168     struct thread_info_struct *thr = arg;
169
170     for (int i = thr->thrid; i < y_chars; i += thr->thrcnt) {
171         sem_wait(&sem[i % thr->thrcnt]);
172         compute_mandel_line(i, thr->color_val);
173         output_mandel_line(1, thr->color_val);
174         sem_post(&sem[(i+1) % thr->thrcnt]);
175     }
176     return NULL;
177 }
```

Ο χρόνος εκτέλεσης (έστω για 4 νήματα) φαίνεται ακολούθως και παρατηρούμε ότι είναι ο ίδιος με αυτόν του mandel:

```
real    0m0.554s
user    0m0.509s
sys     0m0.040s
```

5. Παρατηρούμε ότι στην περίπτωση που πατήσουμε Ctrl+C πριν το πρόγραμμά μας τερματίσει, το χρώμα των γραμμών του τερματικού αλλάζει: από άσπρο γίνεται το χρώμα που είχε ο χαρακτήρας του `mandel` που ζωγραφιζόταν όταν το πρόγραμμα σταμάτησε. Αυτό, φαίνεται ακολούθως:

[illegible]

Στο σημείο αυτό, επισημαίνουμε ότι τα δύο προγράμματα που παραθέσαμε στην αρχή έχουν την τελική τους μορφή και για τον λόγο αυτό περιέχουν από την αρχή τον κατάλληλο χειρισμό του σήματος Ctrl+C που θα εξηγήσουμε τώρα.

Προκειμένου, επομένως, το χρώμα του τερματικού να επαναφέρεται στο λευκό (default) πριν το πρόγραμμά μας τερματίσει, προσθέτουμε κατάλληλο χειρισμό του σήματος Ctrl+C, όπως φαίνεται ακολούθως:

```
55 void sig_handler(int signum)
56 {
57     reset_xterm_color(1);
58     exit(1);
59 }
```

```
184 // Signal Handler
185 struct sigaction sa;
186 sa.sa_flags = SA_RESTART;
187 sa.sa_handler = sig_handler;
188 if (sigaction(SIGINT, &sa, NULL) < 0)
189 {
190     perror("sigaction");
191     exit(1);
192 }
```

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Μάϊος 2024