

Παράλληλα συστήματα - Χειμερινό '22

Εργασία 2/11-1-2022

Ομάδα:

sdi2000064 Κωνσταντίνος Κανελλάκης

sdi2000212 Μιχάλης Χατζησπύρου

Ο κώδικας αναπτύχθηκε και δοκιμάστηκε στο υπολογιστικό σύστημα linux. Τα scripts της bash επίσης έτρεξαν στα μηχανήματα λινουξ της σχολής ωστόσο τα python scripts έτρεξαν σε προσωπικό υπολογιστή με έκδοση 3.6.13 όπου εγκαταστήθηκαν και οι απαιτούμενες βιβλιοθήκες. Να σημειωθεί ότι οι πίνακες και τα δεδομένα που παρουσιάζονται πιο πάνω μπορεί να παρουσιάσουν μερικές μικρές αλλοιώσεις σε σχέση με την θεωρία (αυτά που δηλαδή θα περιμέναμε να βγουν) καθώς στο μηχανήμα που τρέχαμε τα προγράμματα συχνά ήταν και άλλοι συνδεδεμένοι. Συγκεκριμένα χρησιμοποιήθηκε το μηχανήμα με αριθμό 25 του οποίου τα χαρακτηριστικά είναι τα εξής:

Operating system: VERSION="18.04.6 LTS (Bionic Beaver)"

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 4

Thread(s) per core: 1

Core(s) per socket: 4

Socket(s): 1

Model name: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz

Cache line size: 64

Compiler version: gcc version 7.5.0

Άσκηση 2.1

Για τον υπολογισμό των βελών εντός του κύκλου αναπτύχθηκε μία συνάρτηση η οποία καλείται από την υλοποίηση με threads. Για τον υπολογισμό των τυχαίων μεταβλητών χρησιμοποιήσαμε τον δοσμένο κώδικα εξασφαλίζοντας ότι για κάθε thread το seed είναι διαφορετικό. Επίσης, κάναμε τις απαραίτητες μετατροπές για τον υπολογισμό τυχαίων αριθμών στο διάστημα (-1,1). Παρατηρήσαμε ότι η συνάρτηση my_drand στο αρχείο my_rand.c καθυστερούσε την εκτέλεση του προγράμματος οπότε την επιστρέψαμε στην αρχική της μορφή και πλέον διπλασιάζουμε την τιμή που επιστρέφει και αφαιρώντας 1 παίρνουμε έναν αριθμό στο διάστημα (-1,1).

Για τον παράλληλο αλγόριθμο κάθε νήμα υπολογίζει με την βοήθεια της συνάρτησης υπολογισμού των βελών εντός του κύκλου ένα μέρος του συνολικού αθροίσματος το οποίο και προσθέτει στην global μεταβλητή arrows αν και μόνο αν κανένα άλλο νήμα δεν γράφει σε αυτή. Αφού όλα τα νήματα τερματίσουν η συνάρτηση main κάνει τους υπολογισμούς για το π .

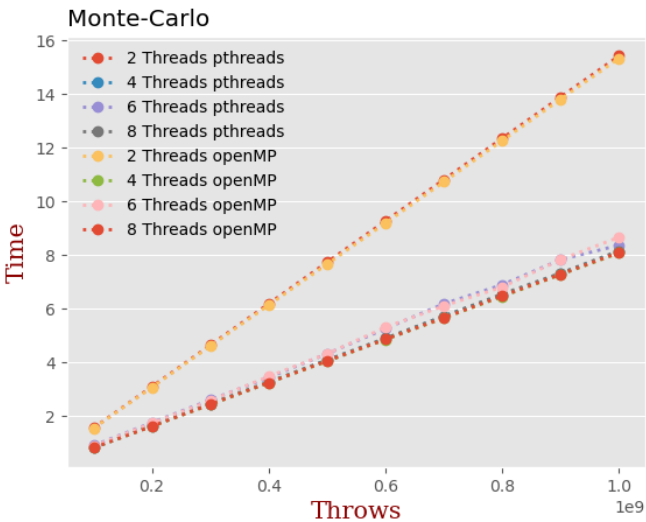
Για την εκτίμηση των αποτελεσμάτων των δύο αλγορίθμων δημιουργήθηκαν δύο επιπλέον αρχεία. Το `script.sh` παίρνει ως ορίσματα τον αριθμό των ρίψεων και τον αριθμό των νημάτων από τον οποίο θα ξεκινήσει. Για κάθε αριθμό ρίψεων εκτελείται το πρόγραμμα 4 φορές και υπολογίζει τον μέσο όρο εκτέλεσης του προγράμματος. Στην συνέχεια αυξάνεται ο αριθμός των ρίψεων κατά τον αριθμό των ρίψεων που δόθηκαν ως όρισμα. Αυτή η διαδικασία επαναλαμβάνεται για 10 φορές και στη συνέχεια αυξάνεται ο αριθμός των νημάτων ανά 2 και εκτελείται όλη η προηγούμενη διαδικασία. Αυτό εκτελείται για 4 διαφορετικές τιμές νημάτων. Στη συνέχεια το παραγόμενο αρχείο `"averages.txt"` (στον φάκελο υπάρχουν τα αρχεία `averages8` και `averages9` τα οποία περιέχουν πειραματικές τιμές με εύρος 10^8 - 10^9 και 10^9 - 10^{10} αντίστοιχα για να εκτελέσετε το `script plots.py` σε περίπτωση που θέλετε) χρησιμοποιείται από το `plots.py` για την δημιουργία γραφικής παράστασης και πίνακα με τα δεδομένα όπως φαίνεται παρακάτω:

0E+08	2.00E+08	3.00E+08	4.00E+08	5.00E+08	6.00E+08	7.00E+08	8.00E+08	9.00E+08	1.00E+09
41474	3.141467	3.141497	3.1415	3.141535	3.141536	3.141526	3.141568	3.141561	3.141587
41882	3.141784	3.141774	3.141691	3.141703	3.141662	3.14162	3.141606	3.141607	3.141597
41732	3.141844	3.141848	3.141726	3.141719	3.141712	3.141676	3.141657	3.14165	3.141667
41684	3.141627	3.141768	3.141789	3.141749	3.141696	3.141696	3.141691	3.141693	3.141651
41733	3.141743	3.141685	3.141768	3.141757	3.141806	3.141763	3.141714	3.141718	3.141702
41855	3.141813	3.141764	3.141712	3.141678	3.141662	3.141649	3.141651	3.141655	3.141631
41771	3.141713	3.141757	3.141736	3.141691	3.141684	3.141658	3.14165	3.141633	3.14165
4167	3.141554	3.141591	3.141635	3.141645	3.141644	3.141648	3.141622	3.141606	3.141584
41748	3.141678	3.14158	3.141635	3.141676	3.141701	3.141686	3.141658	3.141665	3.141639

[illegible]

2 Threads	1.55262000	3.09494375	4.63859175	6.18147875	7.73150050	9.26727900	10.80573650	12.34853050	13.88801250	15.41961000
4 Threads	0.81890225	1.63227200	2.44788825	3.25441850	4.08363850	4.87687650	5.72231150	6.50262500	7.31217150	8.13026400
6 Threads	0.90115425	1.75295875	2.60119075	3.43547975	4.31843500	5.25427400	6.18137100	6.89073875	7.82654450	8.36329475
8 Threads	0.82328050	1.65295750	2.46267600	3.26659450	4.08239550	4.89833050	5.71887275	6.54701700	7.32342500	8.15615200
OpenMP										
2 Threads	1.53981375	3.06887200	4.60704025	6.12944300	7.65908475	9.18818700	10.72172000	12.25912400	13.80420175	15.32571575
4 Threads	0.81618175	1.61858275	2.42268000	3.23972625	4.04019375	4.85045050	5.64992050	6.45441875	7.26902875	8.07489450
6 Threads	0.87669275	1.73669100	2.57367325	3.47453025	4.31921750	5.30356375	6.10363125	6.80331425	7.83171875	8.66060700
8 Threads	0.81548250	1.61836375	2.43334750	3.24373425	4.04966475	4.85869575	5.66073475	6.47379925	7.28095100	8.09275350

Όπως φαίνεται, από το διάγραμμα και τα δεδομένα του πίνακα, όσο αυξάνονται οι ρίψεις της συνάρτησης monte carlo τόσο αυξάνεται και ο χρόνος των υλοποιήσεων μας. Ωστόσο ο σειριακός αλγόριθμος αυξάνεται με πολύ μεγαλύτερο ρυθμό σε σχέση με τις πολυνηματικές υλοποιήσεις όπως και περιμέναμε. Τόσο στην υλοποίηση με pthreads όσο και στην υλοποίηση με OpenMP όσο αυξάνεται ο αριθμός των νημάτων παρατηρούμε όλο και μικρότερη βελτίωση στον χρόνο. Όταν πια ο αριθμός των νημάτων ξεπερνά τα 4 είναι φανερό ότι η βελτίωση στον χρόνο είναι σχεδόν μηδαμινή. Το γεγονός αυτό οφείλεται στο ότι ο επεξεργαστής μας είναι 4 πυρήνων. Παρόλα αυτά η υλοποίηση με Pthreads είναι πιο αργή από την υλοποίηση με OpenMP, γεγονός το



οποίο αναμέναμε. Το OpenMP κατανέμει καλύτερα το φόρτο εργασίας μεταξύ των νημάτων σε σχέση με τα pthreads. Αυτό γίνεται αντιληπτό αφού βάσει των στοιχείων του πίνακα, η διαφορά στον χρόνο εκτέλεσης των δυο αλγορίθμων (με OpenMP και με Pthreads) αυξάνεται όσο αυξάνονται οι ρίψεις των βελών. Επίσης, αξίζει να σημειωθεί ότι, σύμφωνα με τα δεδομένα του πίνακα, η αύξηση στον αριθμό των ρίψεων παρέχει καλύτερη εκτίμηση του π(ακρίβεια 7 σημαντικών ψηφίων). Ωστόσο, ο σειριακός αλγόριθμος προσεγγίζει καλύτερα την τιμή του π μέχρι και 10⁹ ρίψεις σε σχέσει με τις πολυνηματικές υλοποιήσεις. Τέλος, με εξαίρεση τα 2 νήματα η υλοποίηση με OpenMP φαίνεται να προσεγγίζει καλύτερα το π από

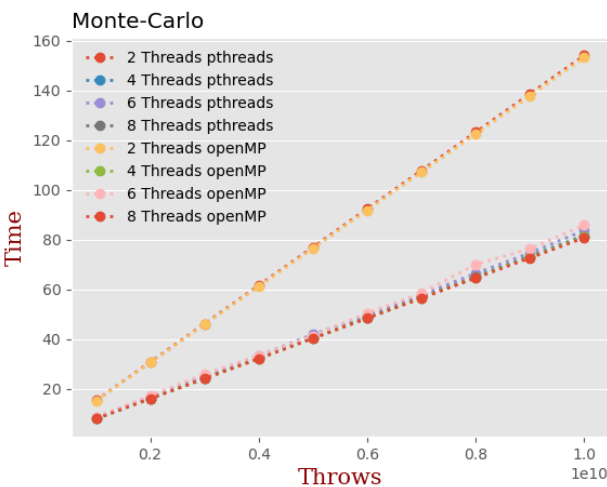
	1.00E+09	2.00E+09	3.00E+09	4.00E+09	5.00E+09	6.00E+09	7.00E+09	8.00E+09	9.00E+09	1.00E+10
Serial	3.141587	3.141554	3.141575	3.141588	3.141584	3.141589	3.141588	3.141583	3.14159	3.141592
Pthreads										
2 Threads	3.141597	3.141582	3.141593	3.141591	3.141596	3.141594	3.141591	3.141592	3.141597	3.141594
4 Threads	3.141667	3.141577	3.141591	3.141593	3.141597	3.141592	3.14159	3.141587	3.141599	3.141595

6 Threads	3.141651	3.141636	3.14159	3.141597	3.141598	3.141592	3.141591	3.141596	3.141592	3.141591
8 Threads	3.141702	3.141675	3.14163	3.14161	3.141606	3.141601	3.141602	3.141597	3.141594	3.141597
OpenMP										
2 Threads	3.141631	3.141626	3.141614	3.141605	3.1416	3.141601	3.141605	3.1416	3.141598	3.141598
4 Threads	3.14165	3.141588	3.141589	3.141598	3.14161	3.141597	3.141598	3.141597	3.1416	3.141595
6 Threads	3.141584	3.14161	3.141587	3.141594	3.141591	3.141595	3.141606	3.14161	3.141602	3.141595
8 Threads	3.141639	3.141617	3.14159	3.141589	3.141585	3.141581	3.141585	3.141591	3.141599	3.141605

ότι η υλοποίηση με pthreads.

Για αριθμό ρίψεων $10^9 - 10^{10}$ και 2,4,6,8 νήματα έχουμε:

	1.00E+09	2.00E+09	3.00E+09	4.00E+09	5.00E+09	6.00E+09	7.00E+09	8.00E+09	9.00E+09	1.00E+10
Serial	42.42681	84.68907	126.9811	170.5107	215.0225	253.8154	296.2896	338.6576	381.0079	423.5081
Pthreads										
2 Threads	15.44005850	30.85268450	46.23434750	61.64317050	77.06407500	92.48458550	107.86077300	123.24553700	138.68368880	154.08678230
4 Threads	8.14777200	16.25384675	24.37318975	32.47435925	40.66248325	48.81389300	56.89924550	65.18816550	73.53764325	81.66997225
6 Threads	8.65606225	16.88844250	25.36775150	33.43418625	41.99583075	49.73987175	57.91930875	66.53064075	74.77851400	83.77593300
8 Threads	8.13594050	16.26251575	24.38102500	32.46940425	40.61286675	48.72151025	56.89325875	65.05362350	73.19058275	81.29863950
OpenMP										
2 Threads	15.33139425	30.64310450	45.94201950	61.23870725	76.55097375	91.89124350	107.20493500	122.50940280	137.81717930	153.25875800
4 Threads	8.10396900	16.17493900	24.21073975	32.31070550	40.43228050	48.56362275	56.59202975	64.87730075	73.10220925	81.17459700
6 Threads	8.62439875	17.41982850	26.04391700	33.51160975	41.45797925	50.59232450	58.61390400	70.14029875	76.41143650	85.94926950
8 Threads	8.08559275	16.17042425	24.20437450	32.26629100	40.36071425	48.46862275	56.56086025	64.65235975	72.70858450	80.80542375



Σύμφωνα με τα δεδομένα για ρίψεις 10^9 - 10^{10} παρατηρούμε αρκετά μεγαλύτερη της διαφοράς στον χρόνο εκτέλεσης της κάθε υλοποίησης με την υλοποίηση σε OpenMP να γίνεται όλο και πιο αποδοτική, χρονικά, καθώς αυξάνονται οι ρίψεις. Τέλος, παρατηρούμε ακόμα καλύτερες εκτιμήσεις για το π συγκρίνοντας και τον πίνακα για το εύρος τιμών 10^8 - 10^9 με τον σειριακό αλγόριθμο να προσεγγίζει περισσότερο την τιμή του π (3.14159265359...) σε

σχέση με τις πολυνηματικές υλοποιήσεις, οι οποίες βρίσκουν επίσης αρκετά καλές προσεγγίσεις σε σχέση με την σειριακή.

Άσκηση 2.2

Για την υλοποίηση των προσεγγίσεων μας χρησιμοποιήσαμε το δοσμένο αρχείο `omp_mat_vect_rand_split.c` και το τροποποιήσαμε καταλλήλως προκειμένου να ελέγχονται οι εισοδοί για τέλεια διαίρεση μεταξύ m και αριθμού νημάτων.

Η μοναδική παραμετροποίηση που έγινε στο αρχείο που μας δόθηκε, είναι η αλλαγή της συνάρτησης `Gen_matrix` για να παράγει άνω τριγωνικό πίνακα.

Στο αρχείο της πρώτης προσέγγισης τροποποιήθηκε ο δείκτης εκκίνησης της εμφωλευμένης `for` προκειμένου να αγνοεί τα μηδενικά στοιχεία κάτω από την κύρια διαγώνιο.

Στο αρχείο της δεύτερης προσέγγισης χρησιμοποιούμε 2 επιπλέον `command line arguments` εκ των οποίων το πρώτο είναι το `schedule type` και το δεύτερο είναι το `chunk size` για τα οποία θα εκτελεστεί η πολυνηματική υλοποίηση. Έχουν επίσης αλλαχθεί οι συναρτήσεις οι οποίες δέχονται τα υπόλοιπα `command line arguments` για έλεγχο των εισόδων. Είναι απαραίτητο να σημειωθεί ότι η αντιστοιχία μεταξύ αριθμών και `schedule type` γίνεται ως εξής:

```
typedef enum omp_sched_t {  
    // schedule kinds  
    omp_sched_static = 0x1,  
    omp_sched_dynamic = 0x2,  
    omp_sched_guided = 0x3,  
    omp_sched_auto = 0x4,  
} omp_sched_t;
```

Για την παρουσίαση των αποτελεσμάτων χρησιμοποιήθηκαν 2 επιπλέον αρχεία `script.sh` και `table.py`. Το `script.sh` παίρνει ως όρισμα τον αριθμό των νημάτων από το οποίο θα ξεκινήσει να εκτελεί και για τις τρεις υλοποιήσεις (αρχική, προσέγγιση1 και προσέγγιση2) τα προγράμματα για τις διαστάσεις πινάκων που δίνονται στην εκφώνηση. Προκειμένου να προκύψει ο μέσος χρόνος εκτέλεσης της κάθε υλοποίησης για κάθε συνδυασμό διαστάσεων πίνακα όπου δίνεται στην εκφώνηση, τρέχουμε 4 φορές το κάθε πρόγραμμα και λαμβάνουμε τον μέσο όρο του χρόνου εκτέλεσης. Συγκεκριμένα για την δεύτερη υλοποίηση τρέχει πειράματα για κάθε αριθμό νημάτων και κάθε συνδυασμό διαστάσεων πίνακα, διαφορετικά `schedule type`(static,dynamic,auto,guided) και `chunk size`(1,4,8,100). Ύστερα, εξάγει τα αποτελέσματα σε αρχείο `txt` το οποίο το λαμβάνει ως είσοδο το `script table.py` και τα αποθηκεύει σε έναν πίνακα 3 διαστάσεων. Τέλος, επεξεργάζεται τα αποτελέσματα, υπολογίζει ποιος συνδυασμός `schedule type` και `chunk size` είναι ο πιο αποδοτικός και τα αποθηκεύει σε ένα αρχείο `excel` με τη μορφή πινάκων(παραδείγματα περιλαμβάνονται στον φάκελο της εργασίας).

Για την παρουσίαση των αποτελεσμάτων μορφοποιήσαμε το `excel` που λαμβάνουμε από το `script table.py`, προκειμένου να γίνεται πιο εύκολα η σύγκριση των αποτελεσμάτων. Τα μορφοποιημένα αποτελέσματα παραθέτονται παρακάτω:

Παρατηρώντας τα δεδομένα του πίνακα συμπεραίνουμε ότι τόσο η πρώτη όσο και η δεύτερη προσέγγιση για μεγάλες τιμές διάστασης του διανύσματος y ($8000000 \times 8,8000 \times 8000$) είναι πιο αποδοτικές από την αρχική. Αυτό οφείλεται στο γεγονός ότι παραλείπονται αρκετοί περιττοί υπολογισμοί. Συγκεκριμένα για την δεύτερη προσέγγιση παρατηρούμε ότι ανάλογα το schedule type και το chunk size για το οποίο επιλέγουμε να την εκτελέσουμε λαμβάνουμε πληθώρα αποτελεσμάτων.

Αρχικά, είναι αναγκαίο να αναφέρουμε ότι το default schedule type της OpenMP είναι το static ενώ για το chunk size η προεπιλεγμένη τιμή είναι 1. Συνεπώς, οι τιμές αυτές χρησιμοποιούνται στην εκτέλεση της πρώτης προσέγγισης.

Λαμβάνοντας υπόψη την θεωρία πριν την εκτέλεση των πειραμάτων με την χρήση των script περιμέναμε να δούμε μικρή βελτίωση με την χρήση dynamic, guided και auto όταν το διάνυσμα y έχει μικρό μέγεθος ($8 \times 8000000, 8 \times 80000000$). Αντίστοιχα όταν το διάνυσμα y έχει μεγάλο μέγεθος ($8000000 \times 8,8000 \times 8000$) περιμένουμε να δούμε αισθητή διαφορά χρησιμοποιώντας schedule type dynamic, guided και auto. Οι δύο αυτοί ισχυρισμοί βασίζονται στο γεγονός ότι εφόσον η εμφωλευμένη for ξεκινάει από i , δηλαδή από διαφορετική τιμή για κάθε νήμα, τότε το φόρτο εργασίας δεν θα κατανέμεται ομοιόμορφα μετεξύ των νημάτων. Συνεπώς, για μικρή διάσταση πίνακα y δεν θα υπάρχει σημαντική διαφορά στην απόδοση χρησιμοποιώντας διαφορετικό schedule type, ενώ για μεγάλη διάσταση πίνακα θα είναι σημαντική. Τόσο ο πρώτος όσο και ο δεύτερος ισχυρισμός επιβεβαιώνονται από τα πειραματικά δεδομένα που παραθέτονται παραπάνω.

Παρατηρώντας τον τελευταίο πίνακα, ο οποίος έχει τους αποδοτικότερους schedule type και chunk size, είναι φανερό ότι για διαστάσεις πίνακα 8×8000000 και 8×80000000 υπάρχουν αντίστοιχοι συνδυασμοί static και chunk size που είναι αρκετά κοντά στις αποδοτικότερες αυτές τιμές. Για παράδειγμα για 8 νήματα και διαστάσεις 8×8000000 με static 100 το πρόγραμμα τρέχει σε 0.2069988 (ανατρέξτε στον αντίστοιχο πίνακα για να δείτε την τιμή) το οποίο βρίσκεται αρκετά κοντά στο 0.20698825 για guided 100 το οποίο (από τον πίνακα «Fastest schedule type and chunk size») είναι το αποδοτικότερο από όλους τους άλλους συνδυασμούς.

Αντίστοιχα παίρνοντας την τιμή για 8 threads και διαστάσεις πίνακα 8000000×8 παρατηρούμε ότι κάθε χρονική πειραματική τιμή του static με οποιοδήποτε από τα chunk size του πειράματος είναι σημαντικά λιγότερο αποδοτική. Το ίδιο ισχύει και για τους υπόλοιπους συνδυασμούς νημάτων και διαστάσεων $8000000 \times 8,8000 \times 8000$.

Επιπρόσθετα περιμέναμε το dynamic ή το guided να είναι πιο αποδοτικό καθώς ευνοεί τα προγράμματα στα οποία οι επαναλήψεις δεν λαμβάνουν τον ίδιο χρόνο για να εκτελεστούν λόγω της τροποποίησης που έχει γίνει για την παράλειψη των μηδενικών στοιχείων κάτω από την κύρια διαγώνιο του πίνακα. Όντως τόσο το dynamic όσο και το guided κυριαρχούν στον πίνακα των πιο αποδοτικών χρονικών πειραματικών τιμών («Fastest schedule type and chunk size»). Το guided είναι λογικό να είναι πάντα αρκετά κοντά με το dynamic και κάποιες φορές καλύτερο καθώς κάνει πιο «δυναμικά» την χρήση του chunk size με αποτέλεσμα κάποιες φορές να είναι αρκετά πιο αποδοτικό.

Το auto επίσης φαίνεται να είναι αρκετά κοντά στις πιο αποδοτικές πειραματικές τιμές καθώς επιλέγεται από τον μεταγλωτιστή ή το σύστημα χρόνου εκτέλεσης με διάφορα κριτήρια που στόχο έχουν την αποδοτική εκτέλεση του αλγορίθμου.