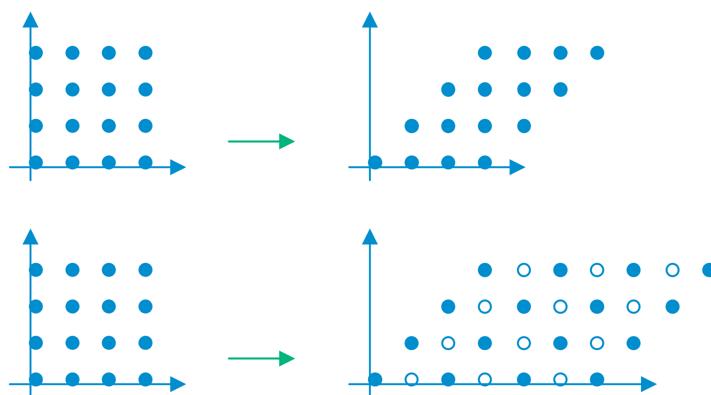
Εθνικό Μετσόβιο Πολυτεχνείο
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ
Εργαστήριο Υπολογιστικών Συστημάτων

Διπλωματική εργασία της:

Μαρίας Γ. Αθανασάκη

Θέμα:

Αλγόριθμοι και Τεχνικές Αυτόματης Παραγωγής Κώδικα για Παράλληλη Εκτέλεση Προγραμμάτων με Φωλιασμένους Βρόχους



Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης

Αθήνα, Ιούνιος 2001

Πρόλογος

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του λέκτορα κυρίου Νεκτάριου Κοζύρη. Για την ολοκλήρωσή της θα ήθελα να εκφράσω τις θερμές ευχαριστίες μου προς τον επιβλέποντα καθηγητή μου, κύριο Νεκτάριο Κοζύρη, και τον υποψήφιο διδάκτορα του ίδιου τμήματος, Γεώργιο Γκούμα για την συνεχή και ακούραστη καθοδήγησή τους σε κάθε βήμα της εκπόνησης της παρούσας διπλωματικής εργασίας μου, καθώς και για τις συμβουλές που με μεγάλη προθυμία μου προσέφεραν σε οποιαδήποτε δυσκολία ή προβληματισμό αντιμετώπισα. Επίσης, θα ήθελα να ευχαριστήσω τα μέλη της τριμελούς επιτροπής κύριο Γεώργιο Παπακωνσταντίνου και κύριο Παναγιώτη Τσανάκα, καθηγητές του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ του Ε.Μ.Π. Τέλος, ευχαριστώ όλα τα μέλη του εργαστηρίου για την συμβολή τους στην εξοικείωσή μου με το χώρο, και ιδιαίτερα τον υποψήφιο διδάκτορα Ιωάννη Δροσίτη για την προθυμία του να απαντήσει σε οποιαδήποτε απορία μου.

Περιεχόμενα

ΕΙΣΑΓΩΓΗ 1

ΚΕΦΑΛΑΙΟ 1: ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΒΡΟΧΩΝ – TILING 5

1.1 Φωλιασμένοι βρόχοι 5
1.2 Διανύσματα εξάρτησης 6
1.3 Γραμμικοί μετασχηματισμοί βρόχων – Unimodular πίνακες 8
1.4 Παραλληλοποίηση προγραμμάτων – Tiling 12
1.5 Σκοπός του tiling 18
1.6 Ιδιότητες tiling 21
1.7 Μετασχηματισμός Υπερκόμβων (Supernode Transformation) 24
1.8 Κόστος υπολογισμού, επικοινωνίας των tiles 25
1.9 Tiling με μηδενικό κόστος επικοινωνίας 29

ΚΕΦΑΛΑΙΟ 2: ΠΑΡΟΥΣΙΑΣΗ ΜΕΘΟΔΩΝ ΕΠΙΛΥΣΗΣ ΑΛΓΕΒΡΙΚΩΝ ΠΡΟΒΛΗΜΑΤΩΝ 31

2.1 Συστήματα ανισοτήτων 31
2.2 Μέθοδος απαλοιγής Fourier-Motzkin 36
2.2.1 Περιγραφή της μεθόδου 36
2.2.2 Απαλοιφή περιττών ανισοτήτων 46
2.2.3 Πολυπλοκότητα της μεθόδου απαλοιφής Fourier-Motzkin 56
2.3 Lattices – Hermite Normal Form 57

ΚΕΦΑΛΑΙΟ 3: ΓΕΝΝΗΣΗ ΚΩΔΙΚΑ SPMD 63

3.1 Εύρεση των ορίων του Tile Space. 64
3.1.1 Ακριβής μέθοδος. 64
3.1.2 Γρήγορη μέθοδος. 67
3.1.3 Σύγκριση των δύο μεθόδων. 75
3.2 Εύρεση ορίων ενός tile. 78
3.2.1 Μέθοδος ορθογώνιας σάρωσης του tile. 78
3.2.2 Μέθοδος πλάγιας σάρωσης του tile. 82
3.3 Λήψη, αποστολή δεδομένων. 91

ΚΕΦΑΛΑΙΟ 4: ΥΛΟΠΟΙΗΣΗ 95

4.1 Παρουσίαση υλοποιημένου κώδικα. 95
4.2 Παράδειγμα. 99
4.3 Συμπεράσματα. 107

ΠΑΡΑΡΤΗΜΑΤΑ 109

Παράρτημα 1: Αρχεία κώδικα που χρησιμοποιήθηκαν κατόπιν τροποποιήσεων . . 109
Παράρτημα 2: Αρχεία κώδικα που κατασκευάστηκαν. 135

ΒΙΒΛΙΟΓΡΑΦΙΑ 165

Εισαγωγή

Οι σύγχρονοι υπερυπολογιστές, οι οποίοι βασίζονται στην ταυτόχρονη λειτουργία και στην επικοινωνία περισσότερων του ενός επεξεργαστών, υπόσχονται μεγάλη υπολογιστική ισχύ και γρήγορη επίλυση ιδιαίτερα πολύπλοκων και απαιτητικών αλγορίθμων. Ιδιαίτερα απαιτητικά θεωρούνται τα προγράμματα που περιέχουν επαναλαμβανόμενες δομές εκτέλεσης, όπως είναι οι φωλιασμένοι βρόχοι, που αποτελούνται από εντολές επανάληψης "for loops" τη μία μέσα στην άλλη. Για παράδειγμα, η εύρεση του δυναμικού σε κάθε σημείο ενός τρισδιάστατου χώρου απαιτεί την χρησιμοποίηση 4 φωλιασμένων βρόχων, κάθε ένας από τους οποίους περιλαμβάνει πολλές επαναλήψεις. Παρόμοια προβλήματα απαντώνται συχνά στον κλάδο της επεξεργασίας εικόνας και της μετεωρολογίας. Επίσης, μεγάλη υπολογιστική ισχύ απαιτούν η μελέτη της μαγνητικής συμπεριφοράς υλικών, με σκοπό τη μείωση του θορύβου σε λεπτά μεταλλικά films, που χρησιμοποιούνται για την επικάλυψη δίσκων υψηλής πυκνότητας, η μεγάλης έκτασης προσομοίωση των φαινομένων των ωκεανών και της ανταλλαγής θερμότητας με ατμοσφαιρικά ρεύματα, η μελέτη των χημικών και δυναμικών μηχανισμών που επηρεάζουν τη διαδικασία μείωσης του όζοντος στην ατμόσφαιρα, καθώς και η σχεδίαση νέων φαρμάκων κατά του AIDS και του καρκίνου, με τον εντοπισμό νέων παραγόντων που αναστέλλουν τη δράση της πρωτεάσης της ανθρώπινης ανοσοανεπάρκειας. Στη συνέχεια, η παράλληλη επεξεργασία βρίσκει εφαρμογή στην ανάπτυξη νέων αποδοτικότερων αεροσκαφών, μέσω της υπολογιστικής μηχανικής ρευστών, στη σχεδίαση νέων χημικών καταλυτών που ελέγχονται από ένζυμα, για την οποία είναι απαραίτητες εκτενείς προσομοιώσεις για τη μείωση του χρόνου σχεδίασης των καταλυτών και για τη βελτιστοποίηση των ιδιοτήτων τους και στην πραγματοποίηση αξονικών και μαγνητικών τομογραφιών σε πραγματικό χρόνο. Τέλος,

χρησιμοποιείται για τη μελέτη μοντέλων ατμοσφαιρικής ποιότητας, για την έγκαιρη λήψη των πλέον πρόσφορων μέτρων, καθώς και για τη μελέτη της δομής του σχηματισμού πρωτεϊνών, με τη χρήση υπολογιστικών προσομοιώσεων.

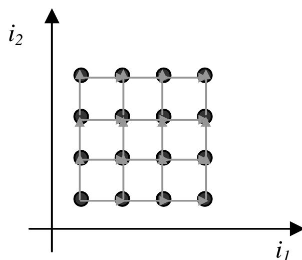
Προκειμένου, όμως, να επιτευχθούν τα προσδοκώμενα αποτελέσματα είναι απαραίτητος ένας πολύ προσεκτικός σχεδιασμός του τρόπου με τον οποίο οι υπερυπολογιστές χρησιμοποιούν τους διαθέσιμους πόρους καθώς και του τρόπου επικοινωνίας μεταξύ των επεξεργαστών, έτσι ώστε η απώλεια χρόνου να είναι όσο το δυνατόν μικρότερη. Επειδή, όμως ένας τέτοιος σχεδιασμός είναι κάθε φορά μια ιδιαίτερα επίπονη διαδικασία και απαιτεί γνώση των χαρακτηριστικών του συγκεκριμένου προγράμματος, αλλά και της αρχιτεκτονικής του συστήματος, κρίνεται σκόπιμη, όπου αυτό είναι δυνατόν η αυτοματοποίηση της παραγωγής του τελικού κώδικα για την παράλληλη εκτέλεση κάθε προγράμματος.

Για το συγκεκριμένο σκοπό, προτείνεται αρχικά στη βιβλιογραφία [KOZ98] η μέθοδος της fine-grained παραλληλοποίησης. Πρόκειται για ένα μετασχηματισμό, ο οποίος με δεδομένο το πρόβλημα που θέλουμε να λύσουμε βρίσκει τη χρονική στιγμή που πρέπει να εκτελεστεί η κάθε επανάληψη, ώστε όσο το δυνατόν περισσότερες να εκτελούνται ταυτόχρονα.

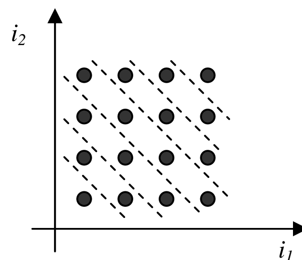
Σαν παράδειγμα θεωρούμε το πρόγραμμα φωλιασμένων βρόχων που περιγράφεται από τον εξής κώδικα:

```
for (i1=0; i1<=3; i1++)
      for (i2=0; i2<=3; i2++)
      {
            A[i1][i2]=(A[i1-1][i2]+A[i1][i2-1])/2
      }
```

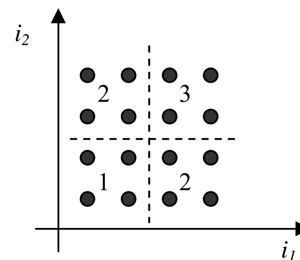
Στο σχήμα 0.1α έχουμε αναπαραστήσει στο επίπεδο τις επαναλήψεις του δισδιάστατου αυτού προβλήματος. Στο ίδιο σχήμα έχουμε σχεδιάσει για κάθε επανάληψη ποιες άλλες πρέπει να έχουν εκτελεστεί πριν από αυτήν ώστε να είναι έτοιμα τα απαραίτητα δεδομένα για την εκτέλεσή της.



Σχήμα 0.1α



Σχήμα 0.1β



Σχήμα 0.1γ

Στο σχήμα 0.1β με διακεκομμένες γραμμές έχουμε χωρίσει τις επαναλήψεις που πρέπει να εκτελεστούν σε διαφορετικές χρονικές στιγμές. Δηλαδή, μεταξύ δύο διαδοχικών

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

διακεκομμένων γραμμών βρίσκονται οι επαναλήψεις που με τη βοήθεια ενός fine-grained παραλληλισμού θα εκτελεστούν ταυτόχρονα.

Επειδή, όμως, το υπολογιστικό κόστος κάθε επανάληψης είναι πολύ μικρό, πολλές φορές είναι προτιμότερη η ομαδοποίηση των γειτονικών επαναλήψεων και ο χειρισμός τους ως ένας κόμβος. Κάθε μία από τις ομάδες που σχηματίζεται με τον τρόπο αυτό ονομάζεται υπερκόμβος, ή tile. Με τον τρόπο αυτό επιτυγχάνουμε coarse-grained παραλληλισμό. Για παράδειγμα, στο σχήμα 0.1γ έχουμε χωρίσει τις επαναλήψεις του προηγούμενου προγράμματος σε τετράγωνα tiles. Σε κάθε ένα έχουμε αναγράψει τη χρονική στιγμή που πρέπει να εκτελεστεί, ώστε όσο το δυνατόν περισσότερα από αυτά να εκτελεστούν ταυτόχρονα.

Γενικά, ο n-διάστατος χώρος των επαναλήψεων μπορεί να έχει οποιοδήποτε κυρτό σχήμα και όχι μόνο ορθογωνικό. Επίσης, τα tiles μπορούν να είναι γενικά υπερπαραλληλεπίπεδα και όχι μόνο ορθογώνια. Προκειμένου να παραχθεί αυτόματα ο κώδικας για την παράλληλη εκτέλεσή τους, ιδιαίτερη δυσκολία παρουσιάζει η εύρεση των ορίων μέσα στα οποία βρίσκονται οι συντεταγμένες των tiles που πρέπει να διατρεχθούν και οι συντεταγμένες των επαναλήψεων που ανήκουν σε κάθε tile.

Η παρούσα εργασία, βασισμένη σε δημοσιεύσεις από περιοδικά και διεθνή συνέδρια του χώρου της παράλληλης επεξεργασίας, υποθέτει ότι είναι δεδομένος ο τρόπος χωρισμού του χώρου των επαναλήψεων σε tiles. Σκοπός της είναι να αναλύσει τις τεχνικές ανεύρεσης των παραμέτρων, που είναι απαραίτητες για την παραγωγή του τελικού κώδικα, ο οποίος μπορεί να χρησιμοποιηθεί για παράλληλη επεξεργασία ενός προβλήματος.

Συγκεκριμένα, μελετάει τις μεθόδους που έχουν ήδη προταθεί στη βιβλιογραφία [ANC91] για τη διάσχιση όλων των tiles και τη σάρωση του εσωτερικού τους. Στη συνέχεια, ερευνάει μία νέα μέθοδο για κάθε ένα από τα δύο προβλήματα και τις συγκρίνει με τις ήδη υπάρχουσες. Τέλος, αναπτύσσει τον κώδικα που είναι απαραίτητος για την αυτοματοποίηση όλων των παραπάνω. Επομένως, γίνεται μια επισκόπηση των μεθόδων της βιβλιογραφίας για την αυτόματη παραγωγή κώδικα SPMD, στη συνέχεια προτείνονται τροποποιήσεις ή βελτιώσεις και υλοποιούνται σε C++.

Τα αποτελέσματα της παρούσας εργασίας έχουν περιληφθεί σε επιστημονικές εργασίες που έχουν υποβληθεί σε ένα διεθνές συνέδριο του χώρου της παράλληλης επεξεργασίας [GOU01a] και σε ένα περιοδικό [GOU01b]. Για την κατανόησή της προϋποτίθενται βασικές γνώσεις προγραμματισμού και κάποιες στοιχειώδεις έννοιες γραμμικής άλγεβρας και γεωμετρίας.

Αναλυτικότερα, στο Κεφάλαιο 1 επεξηγούνται μέσα από παραδείγματα οι βασικές έννοιες του tiling, τις οποίες θα χρησιμοποιήσουμε στη συνέχεια. Επίσης, για την πληρότητα της παρούσας εργασίας, αναφέρονται οι σκοποί που μπορεί να εξυπηρετήσει η τεχνική του tiling, καθώς και οι προϋποθέσεις που πρέπει να ικανοποιούνται ώστε να είναι έγκυρη.

Στο Κεφάλαιο 2 παρουσιάζεται κατ' αρχήν η μέθοδος απαλοιφής Fourier-Motzkin, η οποία έχει χρησιμοποιηθεί ευρύτατα στη βιβλιογραφία για την επίλυση προβλημάτων της παράλληλης επεξεργασίας. Στη συνέχεια, παρουσιάζεται ένας αλγόριθμος για την εύρεση της ερμητιανής μορφής ενός πίνακα.

Στο Κεφάλαιο 3, περιγράφονται όλες οι απαραίτητες διαδικασίες για την παραγωγή του τελικού SPMD κώδικα για την παράλληλη εκτέλεση ενός προγράμματος. Συγκεκριμένα, παρουσιάζονται δύο διαφορετικές μέθοδοι για την εύρεση των ορίων των n εξωτερικότερων από τους βρόχους του παραγόμενου κώδικα για παράλληλη επεξεργασία και στη συνέχεια δύο μέθοδοι για την κατασκευή των n εσωτερικότερων από τους βρόχους του τελικού κώδικα. Επίσης, γίνεται μια αναφορά στον τρόπο επικοινωνίας μεταξύ των μονάδων του συστήματος παράλληλης επεξεργασίας, καθώς και στον τρόπο επιλογής των δεδομένων που πρέπει να σταλούν από τον έναν επεξεργαστή στον άλλο.

Τέλος, στο Κεφάλαιο 4 περιγράφεται ο κώδικας που αναπτύξαμε για την αυτοματοποίηση όλων των παραπάνω και χρησιμοποιείται για την εφαρμογή ενός επεξηγηματικού παραδείγματος. Έπειτα συνοψίζονται σε λίγες γραμμές τα σημαντικότερα συμπεράσματα της παρούσας εργασίας.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Κεφάλαιο 1 : Μετασχηματισμοί βρόχων - Tiling

1.1 Φωλιασμένοι βρόχοι

Τα προγράμματα στα οποία αναφέρεται η παρούσα εργασία περιέχουν *«τέλεια φωλιασμένους βρόχους»*, δηλαδή έχουν την εξής μορφή:

```
for (i1=l1; i1<=u1; i1++)
    for (i2=l2; i2<=u2; i2++)
        … … … … …
        for (in=ln; in<=un; in++)
        {
            Ομάδα εντολών
        }
```

όπου *τα* l_1 *και* u_1 *είναι* *σταθεροί ακέραιοι αριθμοί* και *τα* l_k, u_k $(k=2,\ldots,n)$ *είναι γραμμικές συναρτήσεις των* $i_1,\ldots,i_{k-1}$.

$$l_k = \max(\lceil a_{10}+a_{11}i_1+\ldots+a_{1(k-1)}i_{k-1}\rceil,\lceil a_{20}+a_{21}i_1+\ldots+a_{2(k-1)}i_{k-1}\rceil,\ldots,\lceil a_{p0}+a_{p1}i_1+\ldots+a_{p(k-1)}i_{k-1}\rceil)$$

$$u_k = \max(\lfloor b_{10}+b_{11}i_1+\ldots+b_{1(k-1)}i_{k-1}\rfloor,\lfloor b_{20}+b_{21}i_1+\ldots+b_{2(k-1)}i_{k-1}\rfloor,\ldots,\lfloor b_{q0}+b_{q1}i_1+\ldots+b_{q(k-1)}i_{k-1}\rfloor)$$

όπου τα a_{ij} και b_{ij} είναι σταθεροί ρητοί αριθμοί.

Κάθε επανάληψη του συστήματος αυτού των φωλιασμένων βρόχων αντιπροσωπεύεται από το n-διάστατο διάνυσμα $\mathbf{i}=(i_1,i_2,\ldots,i_n)\in Z^n$, που ονομάζεται διάνυσμα επανάληψης ή ***iteration vector***. Κάθε στοιχείο του διανύσματος επανάληψης αντιστοιχεί σε έναν από τους φωλιασμένους βρόχους. Το στοιχείο i_1 αντιστοιχεί στον εξωτερικότερο βρόχο και το i_n στον εσωτερικότερο.

Ορισμός 1.1

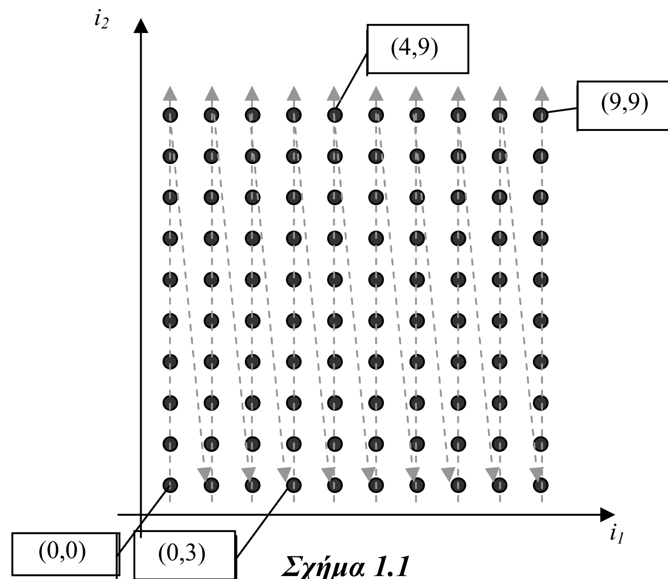
Iteration Space $I^n \subset Z^n$ είναι το σύνολο των iteration vectors που πρόκειται να διατρεχθούν κατά την εκτέλεση του προγράμματος. $\mathbf{I^n} =\{\mathbf{i}=(i_1,i_2,\ldots,i_n) \mid i_j\in Z \wedge l_j\le i_j\le u_j, 1\le j\le n\}$.

Σύμφωνα με τους περιορισμούς που δόθηκαν παραπάνω για τη μορφή που μπορούν να έχουν τα όρια των φωλιασμένων βρόχων, το iteration space I^n είναι ένα κυρτό υποσύνολο του Z^n.

1.2 Διανύσματα εξάρτησης

Θεωρούμε ένα απλό παράδειγμα:

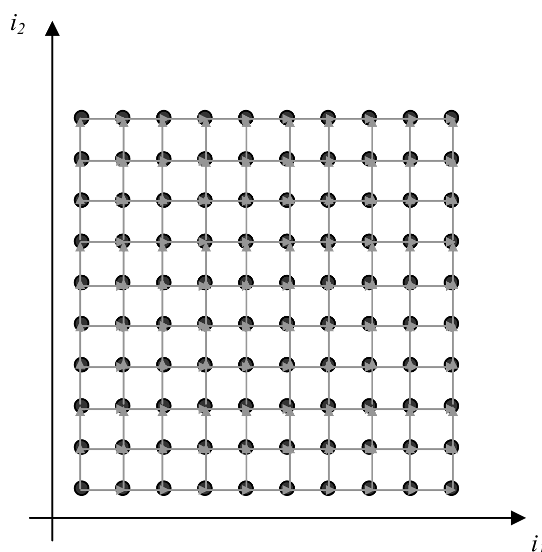
```
for (i1=0; i1<=9; i1++)
      for (i2=0; i2<=9; i2++)
      {
            A[i1][i2]=(A[i1-1][i2]+A[i1][i2-1])/2
      }
```



Σχήμα 1.1

Το πρόγραμμα αυτό ορίζει μια ομάδα επαναλήψεων, η οποία γραφικά μπορεί να παρασταθεί στο επίπεδο όπως φαίνεται στο σχήμα 1.1.

Στο σχήμα 1.1 φαίνεται και η σειρά με την οποία θα διατρεχθούν οι επαναλήψεις κατά την εκτέλεση του παραπάνω προγράμματος. Στην πραγματικότητα, όμως, προκειμένου να αρχίσει η εκτέλεση των σημείων της δεύτερης στήλης του σχήματος, δεν είναι απαραίτητο να ολοκληρωθεί η εκτέλεση της πρώτης στήλης. Συγκεκριμένα, όπως φαίνεται στο σχήμα 1.2, για την εκτέλεση του κάθε σημείου είναι απαραίτητο να έχει ολοκληρωθεί μόνο η εκτέλεση αυτού που βρίσκεται ακριβώς από κάτω του και του σημείου που βρίσκεται στα αριστερά του.



Σχήμα 1.2

Το γεγονός αυτό μπορεί διαφορετικά να εκφραστεί με τη χρήση των διανυσμάτων εξάρτησης, τα οποία ορίζονται ως εξής:

Ορισμός 1.2

Ως ***<u>διάνυσμα εξάρτησης</u>*** μιας επανάληψης ενός προγράμματος με φωλιασμένους βρόχους ορίζουμε τη διαφορά του διανύσματος $\boldsymbol{i}=(i_1,i_2,\dots,i_n)$ που εκφράζει την συγκεκριμένη επανάληψη μείον το διάνυσμα $\boldsymbol{i}'=(i_1',i_2',\dots,i_n')$, το οποίο εκφράζει μια επανάληψη, της οποίας τα αποτελέσματα χρησιμοποιούνται άμεσα κατά τους υπολογισμούς που πραγματοποιούνται από την $\boldsymbol{i}$.

Σύμφωνα με τον ορισμό αυτό, τα διανύσματα εξαρτήσεων του παραπάνω παραδείγματος είναι τα $(i_1,i_2)-(i_1-1,i_2)=(1,0)$ και $(i_1,i_2)-(i_1,i_2-1)=(0,1)$ και είναι σταθερά για όλες τις επαναλήψεις.

Αν στο εσωτερικό των φωλιασμένων βρόχων είχαμε την εντολή

```
A[i₁][i₂]=(A[1][i₂]+A[i₁][1])/2
```

τότε τα διανύσματα εξάρτησης θα ήταν (i_1,i_2)-$(1,i_2)$=$(i_1$-1,0) και (i_1,i_2)-$(i_1$,1)= (0,i_2-1), τα οποία δεν είναι σταθερά.

Στο εξής, στην παρούσα εργασία θα ασχοληθούμε με προγράμματα που έχουν μόνο *σταθερά διανύσματα εξάρτησης* στο εσωτερικό φωλιασμένων βρόχων. Επίσης υποθέτουμε ότι τα διανύσματα εξάρτησης των προγραμμάτων μας είναι *λεξικογραφικά θετικά*. Με τον όρο αυτό εννοούμε ότι θα πρέπει το πρώτο μη μηδενικό στοιχείο τους να είναι θετικό. (Για παράδειγμα είναι λεξικογραφικά θετικά τα διανύσματα (1,2,1), (0,3,1), (1,0,-2), (0,2,-5), αλλά όχι τα διανύσματα (-1,0,0), (0,-2,3).) Το γεγονός αυτό εξασφαλίζει ότι κάθε επανάληψη του αρχικού προγράμματος με φωλιασμένους βρόχους, αν αυτό εκτελεστεί σειριακά, θα εξαρτάται μόνον από επαναλήψεις που έχουν ήδη εκτελεστεί.

Ορισμός 1.3

Πίνακας εξαρτήσεων των επαναλήψεων των φωλιασμένων βρόχων είναι ο πίνακας του οποίου οι στήλες αποτελούν το σύνολο των διανυσμάτων εξάρτησης των επαναλήψεων.

Στο παραπάνω παράδειγμα ο πίνακας εξαρτήσεων είναι ο $D=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$

1.3 Γραμμικοί μετασχηματισμοί βρόχων - Unimodular πίνακες

Η ανταλλαγή (interchange), η αναστροφή (reversal) και η περιστροφή (skewing) βρόχων μπορούν να χαρακτηριστούν ως γραμμικοί μετασχηματισμοί του iteration space ή των iteration vectors. Κάθε ένας από τους γραμμικούς μετασχηματισμούς αυτούς μπορεί να παρασταθεί από έναν $n \times n$ πίνακα μετασχηματισμού. Η σύνθεση περισσότερων του ενός γραμμικών μετασχηματισμών πραγματοποιείται με πολλαπλασιασμό των αντίστοιχων πινάκων. Με κάθε μετασχηματισμό που αντιπροσωπεύεται από έναν πίνακα T, ένα διάνυσμα εξάρτησης $\boldsymbol{d}$ από την επανάληψη $\boldsymbol{i}^s$ στην επανάληψη $\boldsymbol{i}^t$ μετασχηματίζεται στο διάνυσμα εξάρτησης $T\boldsymbol{d}$ από την επανάληψη $T\boldsymbol{i}^s$ στην επανάληψη $T\boldsymbol{i}^t$. Αφού ο πίνακας T έχει διαστάσεις $n \times n$, είναι προφανές ότι από έναν n-διάστατο χώρο, ο μετασχηματισμένος χώρος που προκύπτει είναι και αυτός n-διάστατος. Στη συνέχεια θα περιγράψουμε αναλυτικά κάθε έναν από τους προαναφερθέντες μετασχηματισμούς βρόχων.

I) Η ***ανταλλαγή (interchange) δύο βρόχων*** μετατρέπει το διάνυσμα επανάληψης *(i,j)* στο διάνυσμα (*j*,*i*). Η μετατροπή αυτή παριστάνεται με το γραμμικό μετασχηματισμό:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}.$$

Ο πίνακας μετασχηματισμού είναι ο $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

Επίσης, η κυκλική εναλλαγή τριών βρόχων (i,j,k) έτσι ώστε ο εσωτερικότερος βρόχος k να γίνει ο πιο εξωτερικός μπορεί να πραγματοποιηθεί με ανταλλαγή των βρόχων j και k και στη συνέχεια ανταλλαγή των i και k. Τα δύο αυτά βήματα παριστάνονται αντίστοιχα από τους πίνακες μετασχηματισμού T_1 και T_2 για τους οποίους ισχύει:

$$T_1\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} i \\ k \\ j \end{pmatrix} \text{ και } T_2\begin{pmatrix} i \\ k \\ j \end{pmatrix} = \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

Η σύνθεση των δύο μετασχηματισμών δίνει το μετασχηματισμό που παριστάνεται από τον πίνακα $T=T_2T_1$:

$$T = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

II) Η ***<u>αναστροφή (reversal) ενός βρόχου</u>*** πραγματοποιείται με πολλαπλασιασμό με -1 του αντίστοιχου δείκτη. Για παράδειγμα, η αναστροφή του εσωτερικότερου δύο βρόχων και στη συνέχεια η ανταλλαγή τους παριστάνονται αντίστοιχα με τους πίνακες

$$T_{rev} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \text{ και } T_{\text{int}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Η σύνθεσή τους δίνει τον τελικό πίνακα μετασχηματισμού, ο οποίος είναι ο

$$T_{\text{int}}T_{rev} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

III) Η ***<u>περιστροφή (skewing) ενός βρόχου</u>*** παριστάνεται από τον πίνακα μετασχηματισμού T_{skew}, ο οποίος προσθέτει κάποιο ακέραιο πολλαπλάσιο ενός δείκτη βρόχου σε έναν άλλο δείκτη:

$$T_{skew} = \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}$$

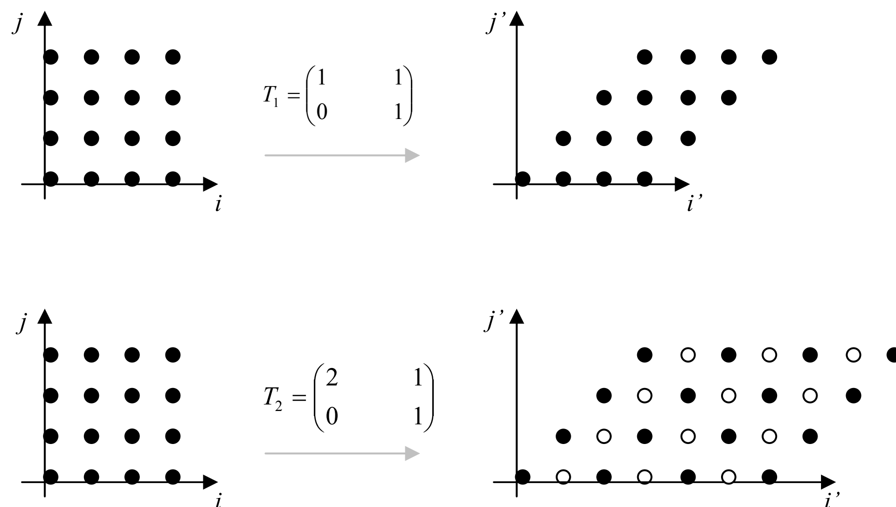
Όλοι οι παραπάνω μετασχηματισμοί βρόχων χαρακτηρίζονται ως "*unimodular*" και παριστάνονται από "*unimodular*" πίνακες. Με τον όρο αυτό εννοούμε:

Ορισμός 1.4

Ένας τετραγωνικός πίνακας *T* χαρακτηρίζεται ***ορθομοναδιαίος*** ή ***unimodular*** αν και μόνο αν όλα τα στοιχεία του είναι ακέραια και η ορίζουσά του ισούται με ± 1.

Οι unimodular πίνακες έχουν την ιδιότητα ότι οι αντίστροφοί τους είναι και αυτοί unimodular. Επομένως κάθε γραμμικός μετασχηματισμός βρόχων που παριστάνεται με έναν unimodular πίνακα, αντιστρέφεται και πάλι με έναν γραμμικό unimodular μετασχηματισμό. Αν εφαρμόσουμε έναν unimodular μετασχηματισμό σε κάποιο iteration space τότε κάθε ακέραιο σημείο του αρχικού χώρου αντιστοιχίζεται σε ακέραιο σημείο του μετασχηματισμένου χώρου (επειδή ο πίνακας μετασχηματισμού έχει μόνο ακέραια στοιχεία). Αντίστροφα, κάθε ακέραιο σημείο του μετασχηματισμένου χώρου αντιστοιχίζεται σε ακέραιο σημείο του αρχικού χώρου. Επίσης, ο όγκος του αρχικού και του μετασχηματισμένου χώρου είναι ίδιος, αφού η ορίζουσα του πίνακα μετασχηματισμού ισούται με ± 1.

Αντίθετα, ο αντίστροφος ενός ακέραιου πίνακα με ορίζουσα διάφορη του ± 1 δεν είναι ακέραιος. Αν εφαρμόσουμε έναν τέτοιο μετασχηματισμό σε κάποιο iteration space τότε κάθε ακέραιο σημείο του αρχικού χώρου αντιστοιχίζεται σε ακέραιο σημείο του μετασχηματισμένου χώρου. Αντίστροφα, όμως, κάθε ακέραιο σημείο του μετασχηματισμένου χώρου δεν αντιστοιχεί σε ακέραιο σημείο του αρχικού χώρου. Τα σημεία του μετασχηματισμένου χώρου, των οποίων η εικόνα στον αρχικό χώρο δεν είναι ακέραια, θα ονομάζονται ***«τρύπες»***. Στο σχήμα 1.3 έχει σχεδιαστεί ένα παράδειγμα



Σχήμα 1.3

unimodular μετασχηματισμού και ένα παράδειγμα non-unimodular μετασχηματισμού. Στην περίπτωση του non-unimodular μετασχηματισμού έχουμε σχεδιάσει με μαύρους κύκλους τα σημεία του μετασχηματισμένου χώρου που αντιστοιχούν σε ακέραια σημεία

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

του αρχικού χώρου και με άσπρους κύκλους τα σημεία του μετασχηματισμένου χώρου που δεν αντιστοιχούν σε ακέραια σημεία του αρχικού χώρου.

Επίσης, παρατηρούμε ότι κάθε $n \times n$ unimodular πίνακας μετασχηματισμού T μπορεί να μετασχηματίσει μια ομάδα φωλιασμένων βρόχων βάθους n. Ο μετασχηματισμός αυτός είναι επιτρεπτός αν τα διανύσματα εξαρτήσεων του μετασχηματισμένου προγράμματος είναι επιτρεπτά, δηλαδή αν είναι λεξικογραφικά θετικά. Κάθε διάνυσμα εξάρτησης $\boldsymbol{d}$ του αρχικού προγράμματος αντιστοιχίζεται στο διάνυσμα εξάρτησης $T\boldsymbol{d}$ για το μετασχηματισμένο πρόγραμμα. Θα πρέπει, λοιπόν, το πρώτο μη μηδενικό στοιχείο του $T\boldsymbol{d}$ να είναι θετικό.

Τέλος, η εικόνα κάθε επανάληψης $\boldsymbol{i}$ του αρχικού προγράμματος θα είναι η επανάληψη $\boldsymbol{j}=T\boldsymbol{i}$ του μετασχηματισμένου. Επειδή ο πίνακας T έχει μη μηδενική ορίζουσα, είναι αντιστρέψιμος και συνεπώς κάθε σημείο $\boldsymbol{j}$ του μετασχηματισμένου χώρου αντιστοιχεί στο σημείο $\boldsymbol{i}=T^{-1}\boldsymbol{j}$ του αρχικού χώρου. Επομένως, αν τα όρια του αρχικού χώρου επαναλήψεων εκφράζονται από το σύστημα ανισοτήτων $B\boldsymbol{i}\le\boldsymbol{b}$, τότε τα όρια του μετασχηματισμένου χώρου δίδονται από το σύστημα $BT^{-1}\boldsymbol{j}\le\boldsymbol{b}$.

Παράδειγμα 1.1

Θεωρούμε το αρχικό πρόγραμμα:

```
for (i1=1; i1<=10; i1++)
     for (i2=1; i2<=i1+1; i2++)
          for (i3=i2; i3<=20; i3++)
          {
               Ομάδα εντολών
          }
```

Σε αυτό εφαρμόζουμε το μετασχηματισμό που ορίζεται από τον πίνακα: $T = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$.

Ο μετασχηματισμένος χώρος αποτελείται από τα διανύσματα $j = \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} = \begin{pmatrix} i_2 + i_3 \\ i_1 + i_2 \\ i_1 \end{pmatrix}$. Ο αντίστροφος μετασχηματισμός δίνει τα διανύσματα του αρχικού χώρου συναρτήσει των διανυσμάτων του μετασχηματισμένου χώρου:

$$\begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} = \begin{pmatrix} j_3 \\ j_2 - j_3 \\ j_1 - j_2 + j_3 \end{pmatrix}.$$

Τα όρια του αρχικού χώρου εκφράζονται από το σύστημα:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} \le \begin{pmatrix} 10 \\ -1 \\ 1 \\ -1 \\ 20 \\ 0 \end{pmatrix}.$$

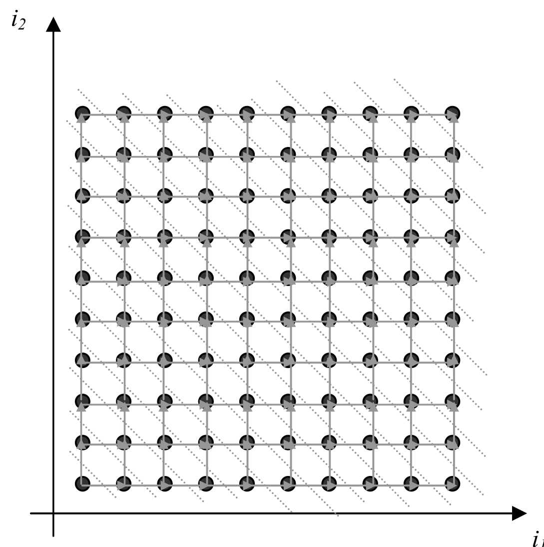
Αντικαθιστώντας με $\boldsymbol{i}=T^{-1}\boldsymbol{j}$ παίρνουμε το σύστημα ανισοτήτων που εκφράζει τα όρια του μετασχηματισμένου χώρου:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -2 \\ 0 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & 2 & -2 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} \le \begin{pmatrix} 10 \\ -1 \\ 1 \\ -1 \\ 20 \\ 0 \end{pmatrix}.$$

♦

1.4 Παραλληλοποίηση προγραμμάτων - Tiling

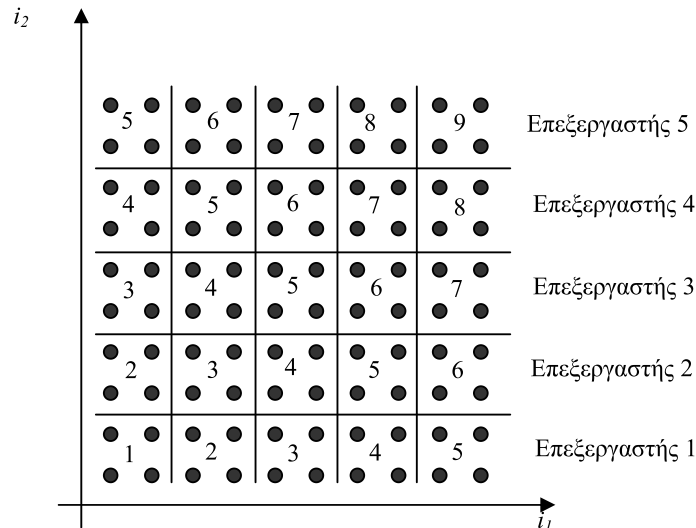
Συνεχίζοντας το παράδειγμα των σχημάτων 1.1 και 1.2, χωρίζουμε το επίπεδο με



Σχήμα 1.4

πλάγιες διακεκομμένες γραμμές όπως φαίνεται στο σχήμα 1.4. Παρατηρούμε ότι είναι δυνατόν οι επαναλήψεις που βρίσκονται μεταξύ δύο διαδοχικών διακεκομμένων γραμμών να εκτελεστούν όλες ταυτόχρονα, από διαφορετικούς επεξεργαστές, προκειμένου να μειώσουμε το συνολικό χρόνο εκτέλεσης του προγράμματος. Μία τέτοια διαμέριση του αρχικού χώρου των επαναλήψεων δίνει την έννοια του *fine-grained παραλληλισμού*, όπως περιγράφεται στη βιβλιογραφία (βλέπε [KOZ98]). Πρόκειται δηλαδή για έναν μετασχηματισμό των αρχικών βρόχων επαναλήψεων, που ως στόχο έχει την παράλληλη εκτέλεση όσο το δυνατόν περισσότερων επαναλήψεων και συνεπώς την κατά το δυνατόν μείωση του συνολικού χρόνου επεξεργασίας.

Επειδή, όμως, το υπολογιστικό κόστος της κάθε επανάληψης είναι πολύ μικρό, κρίνεται προτιμότερο να χωρίσουμε το επίπεδο σε τετράγωνα, όπως στο σχήμα 1.5 και ο κάθε επεξεργαστής να αναλάβει τον υπολογισμό μιας γραμμής τέτοιων τετραγώνων.

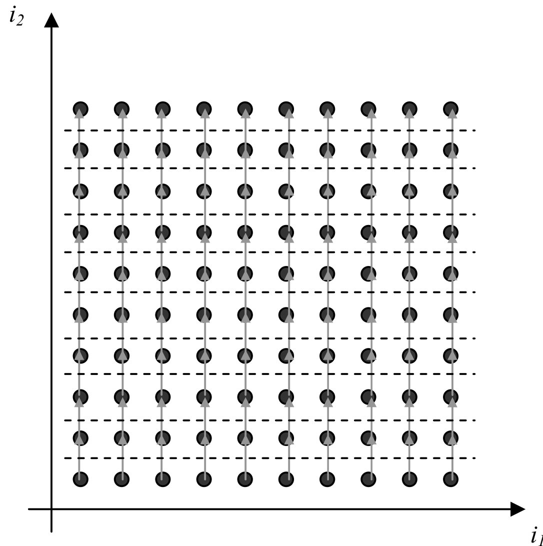


Σχήμα 1.5

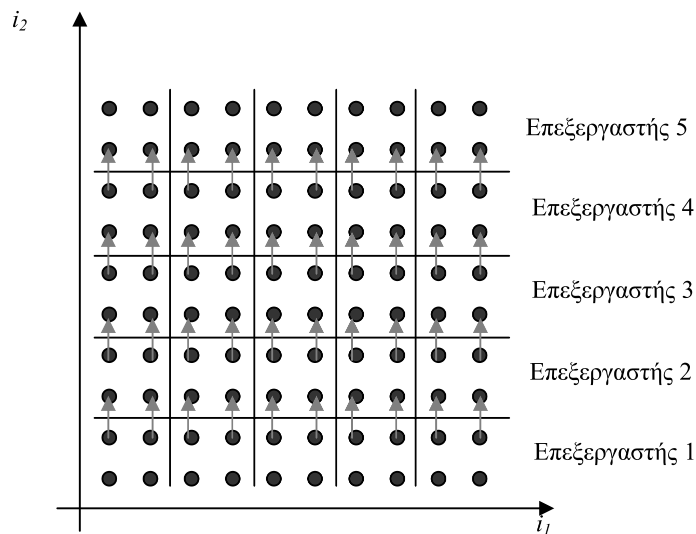
Αν υποθέσουμε ότι η διάρκεια εκτέλεσης κάθε τετραγώνου είναι ο μοναδιαίος χρόνος, τότε η χρονική στιγμή εκτέλεσης του κάθε τετραγώνου θα είναι αυτή που αναγράφεται στο εσωτερικό του. Δηλαδή, ο επεξεργαστής 1 ξεκινά την εκτέλεση του προγράμματος την στιγμή 1, μόλις ολοκληρώσει ένα τετράγωνο, στέλνει στον επεξεργαστή 2 τα απαραίτητα δεδομένα, και έπειτα συνεχίζει με την εκτέλεση του επομένου τετραγώνου που του έχει καταχωρηθεί. Ο επεξεργαστής 2 ξεκινά την εκτέλεση του προγράμματός του τη στιγμή 2, μόλις λάβει τα πρώτα δεδομένα από τον επεξεργαστή 1. Όταν ολοκληρώσει την εκτέλεση ενός τετραγώνου, στέλνει τα απαραίτητα δεδομένα στον επεξεργαστή 3 και συνεχίζει με το επόμενο.

Το παράδειγμα αυτό δίδει την έννοια του *coarse-grained παραλληλισμού*, ή *tiling*. Στόχος του είναι η μείωση του όγκου των δεδομένων που ανταλλάσσονται μεταξύ των επεξεργαστών. Στο σχήμα 1.6α φαίνεται ποια δεδομένα πρέπει να μεταφερθούν από έναν

επεξεργαστή σε άλλον στην περίπτωση που έχουμε πραγματοποιήσει fine-grained παραλληλισμό, οπότε οι επαναλήψεις που ανήκουν στην ίδια γραμμή έχουν ανατεθεί στον ίδιο επεξεργαστή. Στο σχήμα 1.6β φαίνεται ποια δεδομένα πρέπει να σταλούν από έναν επεξεργαστή σε άλλον στην περίπτωση του tiling.



Σχήμα 1.6α

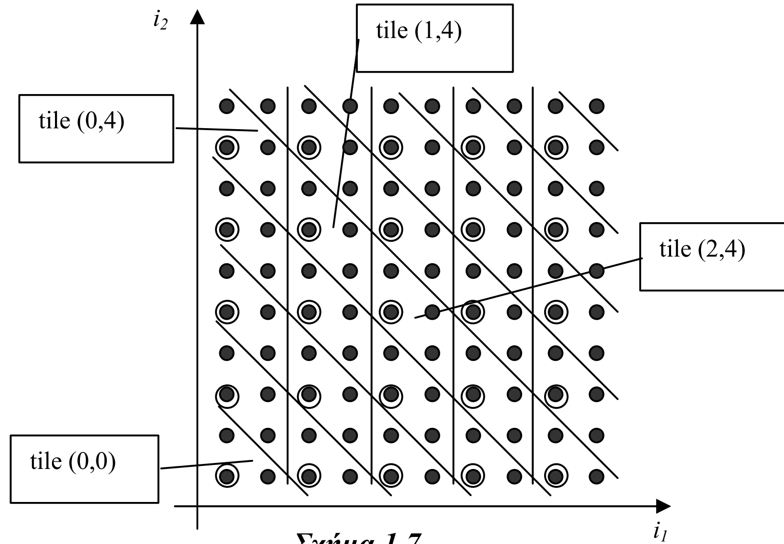


Σχήμα 1.6β

Ο χωρισμός του επιπέδου δεν είναι απαραίτητο να γίνει σε τετράγωνα, γενικότερα μπορούμε να χωρίσουμε σε παραλληλόγραμμα, όπως στο σχήμα 1.7.

Αν οι φωλιασμένοι βρόχοι του προγράμματός μας έχουν βάθος *n*, τότε οι επαναλήψεις μπορούν να παρασταθούν με τα ακέραια σημεία ενός *n*-διάστατου χώρου. Αντίστοιχα ο χωρισμός θα πρέπει να γίνει σε *n-διάστατα υπερπαραλληλεπίπεδα*, τα οποία ονομάζονται ***tiles***.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ



Σχήμα 1.7

Το κάθε tile αντιπροσωπεύεται από ένα n-διάστατο διάνυσμα $\boldsymbol{t}=(t_1,t_2,...,t_n)\in Z^n$, που ονομάζεται ***tile vector***, όπως φαίνεται στο σχήμα 1.7.

Επίσης, σε κάθε tile αντιστοιχεί ένα σημείο εκκίνησης, που ονομάζεται ***tile origin iteration (TOI)***. Για το tile (0,…,0) ενός n-διάστατου χώρου, origin iteration είναι το σημείο (0,…,0). Για οποιοδήποτε άλλο tile, origin iteration είναι το σημείο με το οποίο θα συμπέσει το σημείο (0,…,0) αν μετατοπίσουμε παράλληλα το tile (0,…,0) έτσι ώστε να συμπέσει με το εν λόγω tile.

Στο σχήμα 1.7 τα tile origin iterations του κάθε tile έχουν κυκλωθεί.

Ορισμός 1.5

Ο χωρισμός σε tiles είναι πλήρως καθορισμένος αν γνωρίζουμε τα n διανύσματα των ακμών των υπερπαραλληλεπιπέδων. Επομένως κάθε tiling χαρακτηρίζεται από έναν $n\times n$ πίνακα $\boldsymbol{P}$ που ονομάζεται ***inverse tiling matrix*** του οποίου οι στήλες αποτελούν τα διανύσματα ακμών $\boldsymbol{p_j}$ του υπερπαραλληλεπιπέδου.

Κάθε iteration (=επανάληψη) $\boldsymbol{i}$ ενός tile $\boldsymbol{t}$ με tile origin $\boldsymbol{i_{toi}}$ μπορεί να γραφτεί συναρτήσει των στηλών του πίνακα P στη μορφή $i = i_{toi} + \sum_{j=1}^{n} x_j \cdot p_j,\ \ 0 \le x_j < 1$. Σύμφωνα με τη σχέση αυτή κάθε tile είναι ένα ημιανοιχτό n-διάστατο υπερπαραλληλεπίπεδο.

Παράδειγμα 1.2

Για το σχήμα 1.5 ο αντίστοιχος inverse tiling matrix είναι ο

$$P_1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Το tile (0,0) έχει $\boldsymbol{i_{toi}}$=(0,0). Τα σημεία που ανήκουν στο tile αυτό μπορούν να γραφούν ως εξής:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Για το σχήμα 1.7 ο αντίστοιχος inverse tiling matrix είναι ο

$$P_2 = \begin{bmatrix} 2 & 0 \\ -2 & 2 \end{bmatrix}$$

Το tile (1,4) έχει $\boldsymbol{i_{toi}}$=(2,6). Τα σημεία που ανήκουν στο tile αυτό μπορούν να γραφούν ως εξής:

$$\begin{pmatrix} 2 \\ 6 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix} + 0 \cdot \begin{pmatrix} 2 \\ -2 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 7 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix} + 0 \cdot \begin{pmatrix} 2 \\ -2 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 2 \\ -2 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 3 \\ 6 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 2 \\ -2 \end{pmatrix} + 0{,}5 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

♦

Ορισμός 1.6

Ισοδύναμα, κάθε tiling μπορεί να οριστεί από τον $n \times n$ πίνακα $H=P^{-1}$, ο οποίος ονομάζεται ***tiling matrix***. Γεωμετρικά κάθε γραμμή i του H είναι ένα διάνυσμα κάθετο σε μία από τις πλευρές του υπερπαραλληλεπιπέδου που ορίζει το χωρισμό σε tiles.

Ο πίνακας H έχει την ιδιότητα ότι: Μία επανάληψη $\boldsymbol{i}$ ανήκει στο tile $\boldsymbol{t}$ αν και μόνο αν ισχύει $t = \lfloor H \cdot i \rfloor$. Επίσης, αν $\boldsymbol{i_{toi}}$ είναι το σημείο εκκίνησης του tile $\boldsymbol{t}$ ισχύει: $t = H \cdot i_{toi} \in Z^n$

Ορισμοί 1.7

Tile Iteration Space *TIS(H)=*$\{i \in Z^n \mid 0 \le H \cdot i < 1\}$ είναι το σύνολο όλων των σημείων του Z^n που ανήκουν στο tile *(0,...,0)*.

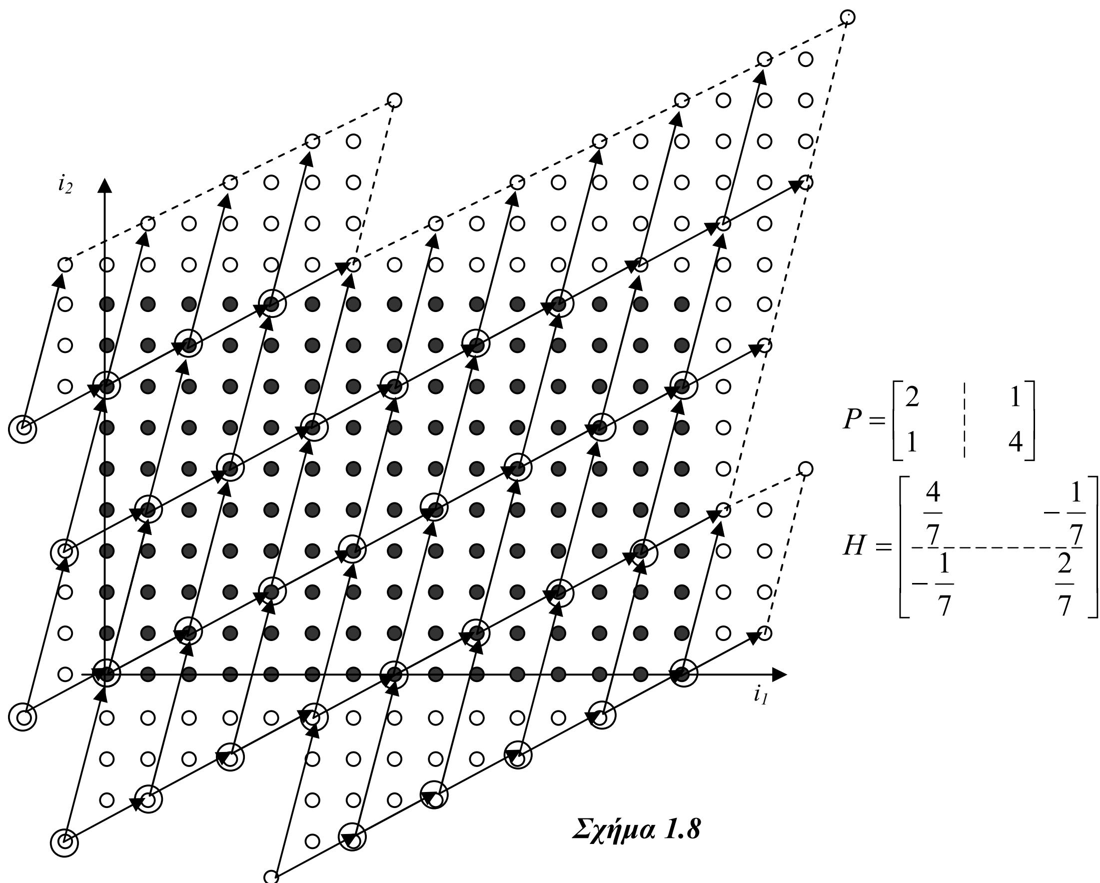
Tile Space $I^S(I^n,H)=\{i^S \in Z^n \mid i^S=\lfloor H \cdot i \rfloor, i \in I^n\}$ είναι το σύνολο των tiles των οποίων τουλάχιστον ένα σημείο ανήκει στο iteration space.

Tile Origin Space $TOS(I^S,H^{-1})=\{i \in Z^n \mid i = H^{-1} i^S, i^S \in I^S\}$ είναι το σύνολο των origins των tiles του tile space.

Στο σημείο αυτό παρατηρούμε ότι το tile origin space δεν είναι πάντα υποσύνολο του iteration space, αφού μπορεί να υπάρχουν tiles, των οποίων τα origins δεν ανήκουν στον αρχικό χώρο I^n, ενώ ανήκουν στον I^n κάποια άλλα σημεία του ίδιου tile.

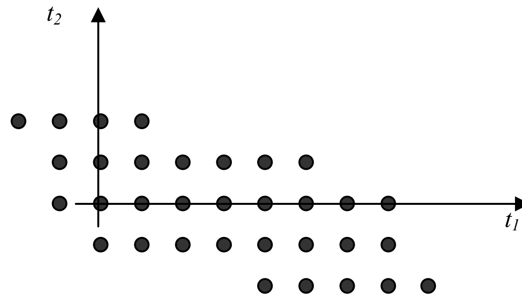
Παράδειγμα 1.3

Στο σχήμα 1.8 απεικονίζεται με μαύρες τελείες το iteration space I^n ενός



Σχήμα 1.8

προγράμματος και με κυκλωμένες τελείες μαύρες ή λευκές το tile origin space για έναν δεδομένο πίνακα tiling *H*. Στο σχήμα 1.9 απεικονίζεται το tile space του ίδιου tiling.



Σχήμα 1.9

♦

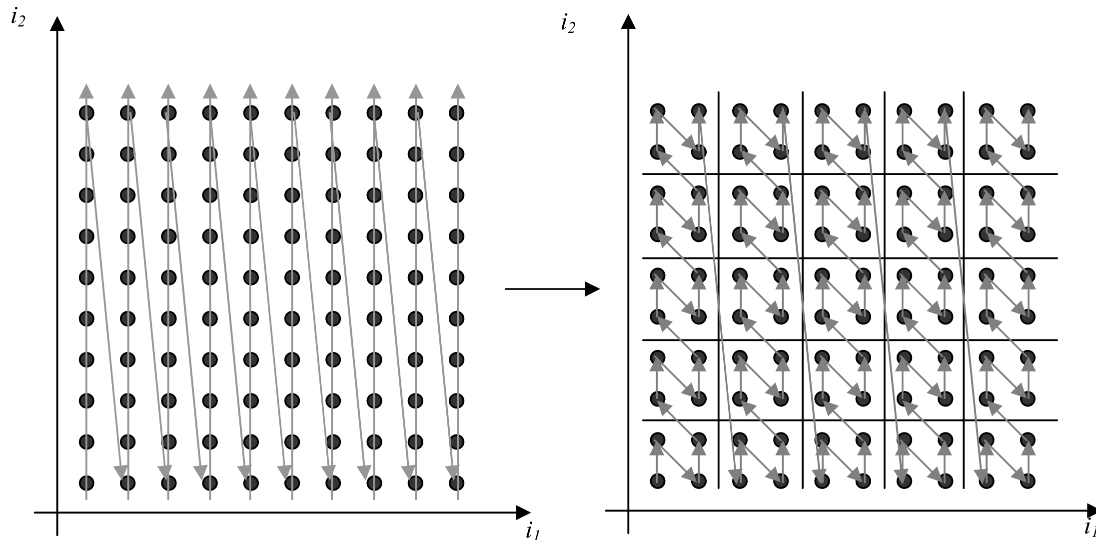
1.5 Σκοπός του tiling

I) Όπως έχει ήδη φανεί από όλα τα προηγούμενα παραδείγματα, με τη βοήθεια του tiling μπορούμε να πετύχουμε το βέλτιστο καταμερισμό των επαναλήψεων ενός προγράμματος με πολλαπλά φωλιασμένους βρόχους μεταξύ των επεξεργαστών ενός συστήματος παράλληλης επεξεργασίας. Ο καταμερισμός αυτός μπορεί να γίνει κατά τρόπο τέτοιο ώστε να επιτρέπεται όσο το δυνατόν μεγαλύτερος ταυτοχρονισμός μεταξύ των τρέχουσων διεργασιών, χωρίς καμία από αυτές να μένει άεργη περιμένοντας να φθάσουν οι υπόλοιπες σε κάποιο σημείο συγχρονισμού. Ταυτόχρονα είναι επιθυμητή η κατά το δυνατόν μικρότερη επικοινωνία μεταξύ των διεργασιών, ώστε να μειωθεί ο χρόνος που απαιτείται για την διεκπεραίωση της επικοινωνίας αυτής. Διαφορετικά, μπορούμε να πούμε ότι επιδιώκουμε τη μείωση του χρόνου επικοινωνίας ως προς το χρόνο υπολογισμού. Αυτό επιτυγχάνεται με την ομαδοποίηση γειτονικών επαναλήψεων, έτσι ώστε η πλειοψηφία των δεδομένων που χρειάζονται να έχουν υπολογιστεί από τον ίδιο επεξεργαστή, στα πλαίσια δηλαδή του ίδιου tile. Επομένως, *μειώνεται ο συνολικός χρόνος επεξεργασίας ιδιαίτερα απαιτητικών προγραμμάτων*.

II) Επίσης, με τις μεθόδους του tiling μπορούμε να αυξήσουμε την *τοπικότητα της αναφοράς* τέτοιων προγραμμάτων *στη μνήμη*. Επομένως αυξάνεται η πιθανότητα να βρίσκεται η ζητούμενη κάθε φορά διεύθυνση στην γρήγορη μνήμη του συστήματος (ή cache memory) και το ποσοστό επαναχρησιμοποίησης της γρήγορης μνήμης. Όλα αυτά συνιστούν μια καλύτερη διαχείριση των παρεχόμενων πόρων από το πρόγραμμα. Ο στόχος αυτός επιτυγχάνεται επειδή

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

ουσιαστικά με την τεχνική του tiling συντελείται μια αναδιάταξη της σειράς εκτέλεσης των επαναλήψεων του αρχικού προγράμματος, όπως έχει σχεδιαστεί στο σχήμα 1.10. Στο σημείο αυτό, για πληρότητα της παρούσας εργασίας δίδεται ένα παράδειγμα χρησιμοποίησης της τεχνικής αυτής.



Σχήμα 1.10

Παράδειγμα 1.4

Θεωρούμε ένα πρόγραμμα πολλαπλασιασμού δύο πινάκων $N \times M$ και $M \times P$. Ο αρχικός κώδικάς του έχει την εξής μορφή:

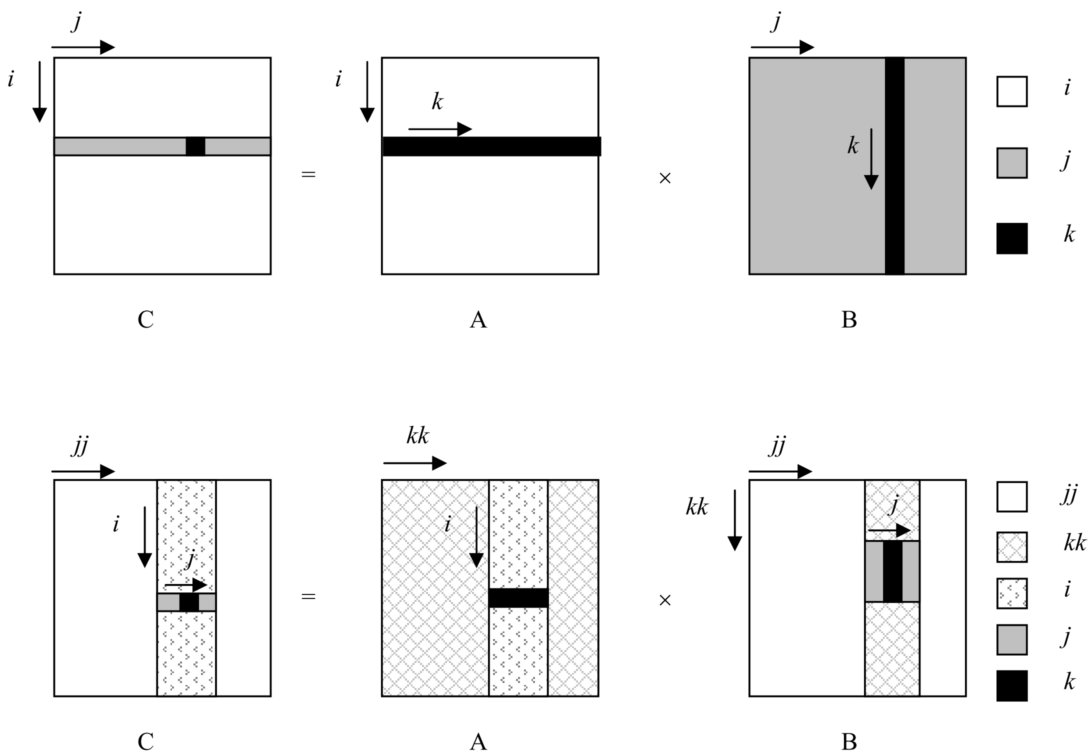
```
for (i=0; i<=N-1; i++)
     for (j=0; j<=P-1; j++)
          for (k=0; k<=M-1; k++)
          {
               C[i][j]=C[i][j]+A[i][k]*B[k][j];
          }
```

Στο παράδειγμα αυτό η ίδια γραμμή του πίνακα A χρησιμοποιείται συνέχεια από τις επαναλήψεις του μεσαίου βρόχου (*j*). Αν το μέγεθος M της γραμμής είναι μεγάλο σε σχέση με το μέγεθος της γρήγορης μνήμης, τότε τα στοιχεία της γραμμής δεν θα βρίσκονται ακόμη στη γρήγορη μνήμη όταν ζητείται να επαναχρησιμοποιηθούν. Για τον πίνακα *B* η επαναχρησιμοποίηση των στοιχείων του γίνεται στον πιο εξωτερικό βρόχο (*i*). Μεταξύ δύο αναγνώσεων του ίδιου στοιχείου του B έχει ήδη ζητηθεί όλος ο πίνακας και επομένως οι αναφορές σε αυτόν δεν θα ικανοποιούνται ποτέ από την γρήγορη μνήμη.

Προκειμένου να αντιμετωπιστούν τα παραπάνω προβλήματα και να παραχθεί πιο αποδοτικός κώδικας είναι δυνατόν να χρησιμοποιήσουμε τις τεχνικές του tiling και να τροποποιήσουμε το παραπάνω πρόγραμμα ως εξής:

```
for (jj=0; jj<=P-1; j+=stepjj)
     for (kk=0; kk<=M-1; kk+=stepkk)
          for (i=0; i<=N-1; i++)
               for (j=jj; j<=min(P-1,jj+stepjj-1); j++)
                    for (k=kk; k<=min(M-1,kk+stepkk-1); k++)
                    {
                         C[i][j]=C[i][j]+A[i][k]*B[k][j];
                    }
```

Με τον τρόπο αυτό αλλάζουμε την σειρά με την οποία θα εκτελεστούν οι διάφορες επαναλήψεις του προγράμματος. Στο σχήμα 1.11 έχουμε αναπαραστήσει γραφικά τον τρόπο προσπέλασης των δεδομένων των τριών πινάκων από τους φωλιασμένους βρόχους του αρχικού και του τελικού προγράμματος. Συγκεκριμένα, όπως φαίνεται στο σχήμα αυτό, οι επαναλήψεις του εξωτερικότερου βρόχου *i* πραγματοποιούνται από το τελικό πρόγραμμα πριν ολοκληρωθούν όλες οι επαναλήψεις των εσωτερικότερων βρόχων. Επομένως μικραίνει η απόσταση μεταξύ διαδοχικών αναφορών στη ίδια θέση μνήμης και αυξάνεται η πιθανότητα εύρεσης των ζητούμενων δεδομένων στη γρήγορη μνήμη.



Σχήμα 1.11

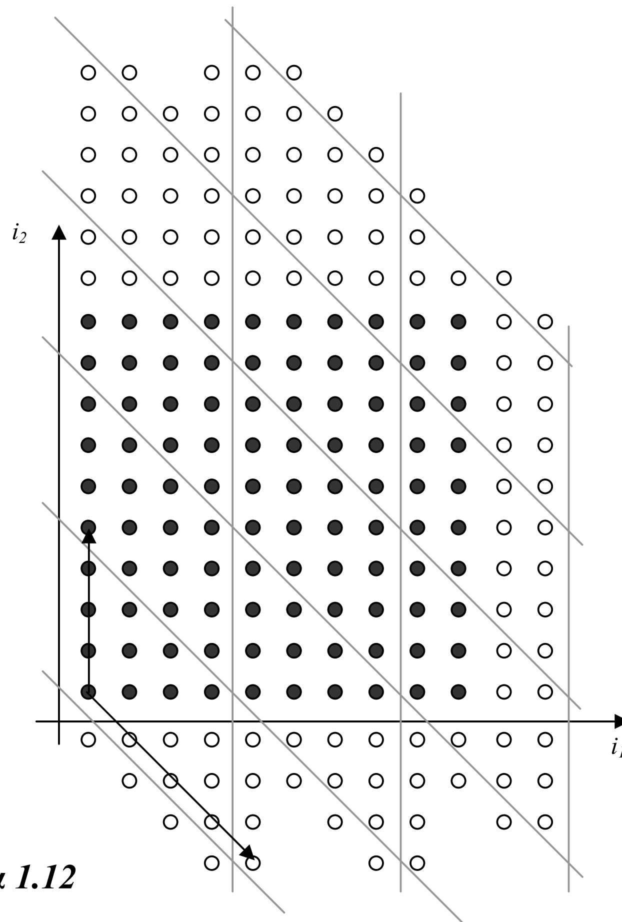
1.6 Ιδιότητες tiling

Τα tiles θα πρέπει να πληρούν ορισμένες προϋποθέσεις, οι οποίες, όπως παρουσιάζονται στο άρθρο των C.Ancourt, F.Irigoin "*Scanning Polyhedra with DO Loops*" [ANC91], είναι:

I) Πρέπει να είναι <u>*σαφώς οριοθετημένα*</u> και <u>*πανομοιότυπα*</u> μεταξύ τους. Για το σκοπό αυτό ορίζονται με τη βοήθεια n οικογενειών παράλληλων μεταξύ τους υπερεπιπέδων.

Δηλαδή πρέπει (γεωμετρικά) ο πίνακας P να αποτελείται από n γραμμικώς ανεξάρτητες μεταξύ τους στήλες, ή (αλγεβρικά) να είναι <u>*αντιστρέψιμος*</u>. Επίσης πρέπει όλα τα στοιχεία του πίνακα P να είναι <u>*ακέραιοι αριθμοί*</u>.

Μόνο τα ακραία tiles μπορεί να είναι διαφορετικά από τα υπόλοιπα. Αυτό οφείλεται στο γεγονός ότι κάποια σημεία του Z^n που ανήκουν σε αυτά πιθανόν να μην ανήκουν στον χώρο I^n (iteration space).



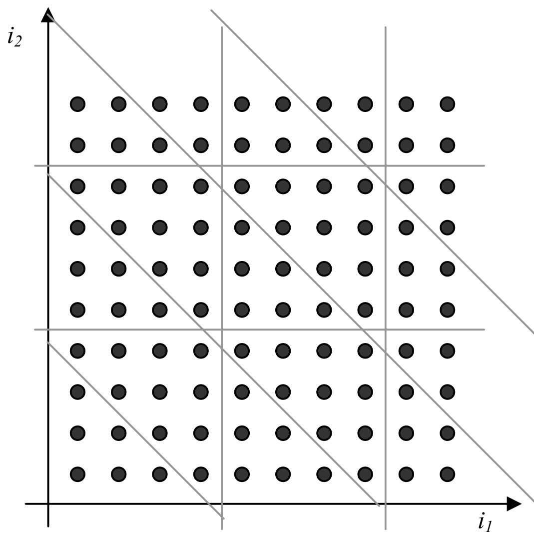
Σχήμα 1.12

Παράδειγμα αποδεκτού tiling είναι αυτό του σχήματος 1.12, όπου

$$P = \begin{bmatrix} 4 & 0 \\ -4 & 4 \end{bmatrix} \text{ και } H = \begin{bmatrix} \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

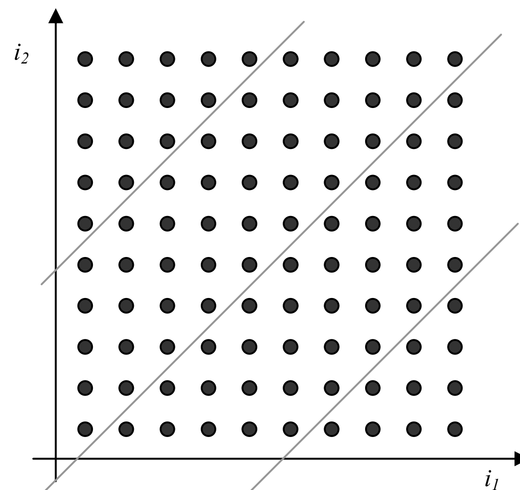
Στο σχήμα 1.12 με μαύρες τελείες έχουμε παραστήσει τα σημεία που ανήκουν στο I^n, ενώ με λευκές τελείες τα σημεία που ανήκουν στο Z^n αλλά δεν ανήκουν στο I^n.

Αντίθετα, δεν είναι αποδεκτά tilings της μορφής των σχημάτων 1.13, 1.14 και 1.15.



Μη πανομοιότυπα μεταξύ τους tiles

Σχήμα 1.13



Μη πεπερασμένα tiles

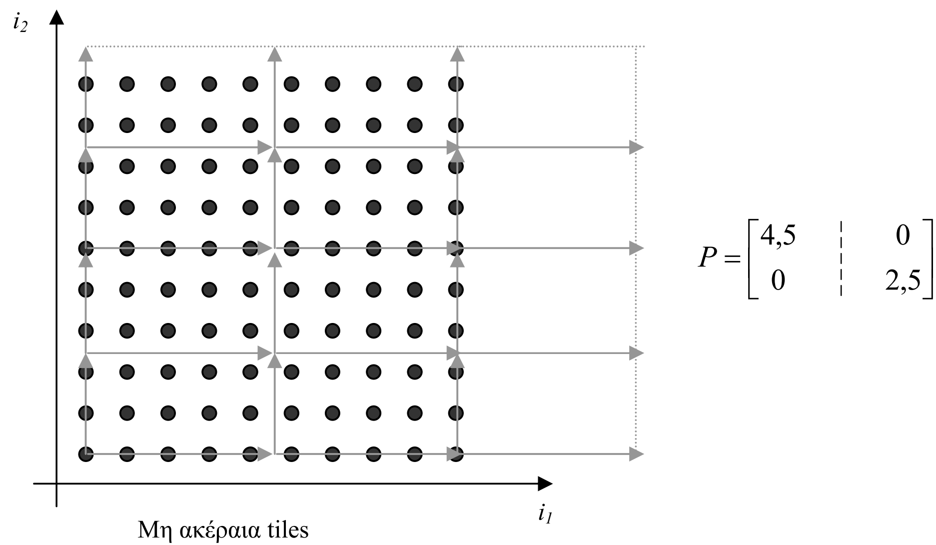
Σχήμα 1.14

Η προϋπόθεση αυτή εξασφαλίζει ότι ο SPMD κώδικας (δηλαδή ο κώδικας για την παράλληλη εκτέλεση του προγράμματος) θα μπορεί να παραχθεί εύκολα, γρήγορα και απλά. Επίσης εξασφαλίζει ότι τα απαραίτητα δεδομένα για την εκτέλεση ενός tile μπορούν να προσαρμοστούν με κάποιο τρόπο στη γρήγορη μνήμη του συστήματος.

II) Πρέπει να καλύπτουν ακριβώς όλο το χώρο I^n (iteration space), έτσι ώστε η κάθε επανάληψη του αρχικού προγράμματος να εκτελεστεί από τον μετασχηματισμένο κώδικα μία και μόνο μία φορά.

III) Πρέπει η εκτέλεση του κάθε tile να είναι *ατομική*.

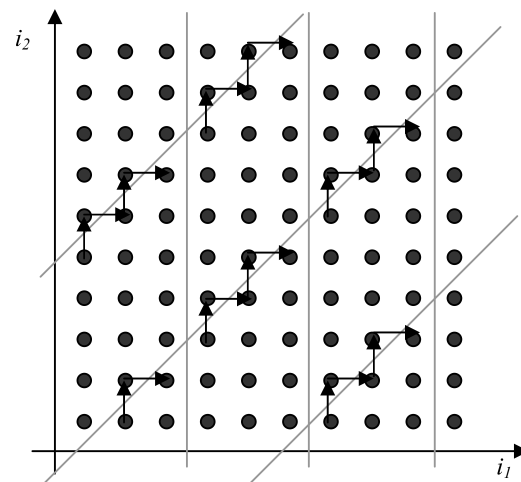
Δηλαδή από τη στιγμή που θα αρχίσει η εκτέλεση ενός tile πρέπει να μπορεί να ολοκληρωθεί χωρίς να απαιτείται κανενός είδους συγχρονισμός ή ανταλλαγή δεδομένων με τα γειτονικά tiles. Όλα τα απαραίτητα δεδομένα πρέπει λαμβάνονται πριν αρχίσει η



Σχήμα 1.15

εκτέλεση του tile και τα αποτελέσματα των υπολογισμών που χρειάζονται σε κάποιο άλλο tile να αποστέλλονται αφού ολοκληρωθεί η εκτέλεση του εν λόγω tile.

Για να είναι αυτό δυνατόν πρέπει να μην υπάρχουν εξαρτήσεις της μορφής του σχήματος 1.16.



Σχήμα 1.16

Για να ισχύει η προϋπόθεση αυτή ικανή και αναγκαία συνθήκη είναι $H \cdot D \geq 0$ (ή $H \cdot D \leq 0$). Δηλαδή πρέπει όλα τα στοιχεία του πίνακα *HD* να είναι μη αρνητικά (ή μη θετικά αντίστοιχα).

Αυτό συμβαίνει επειδή: Κάθε στοιχείο του πίνακα *HD* είναι το εσωτερικό γινόμενο ενός διανύσματος-γραμμής του *H* με ένα διάνυσμα-στήλη του *D*. Κάθε γραμμή

$h_i = (h_{i1}, h_{i2}, ..., h_{in})$ του *H* εκφράζει την κατεύθυνση ενός (*n-1*)-διάστατου υπερεπιπέδου στον *n*-διάστατο χώρο. Κάθε στήλη $d_j = (d_{1j}, d_{2j}, ..., d_{nj})^T$ του *D* εκφράζει την κατεύθυνση ενός διανύσματος εξάρτησης. Το πρόσημο του εσωτερικού γινομένου $h_i \cdot d_j$ αντιπροσωπεύει τη φορά με την οποία το διάνυσμα d_j διασχίζει το υπερεπίπεδο h_i. Για να αποφευχθούν κυκλικές εξαρτήσεις μεταξύ των tiles πρέπει όλα τα οριακά υπερεπίπεδα να διασχίζονται από όλα τα διανύσματα εξάρτησης με την ίδια φορά. Επομένως αρκεί όλα τα στοιχεία του γινομένου *HD* να έχουν το ίδιο πρόσημο.

Παράδειγμα 1.5

Στο σχήμα 1.16 έχουμε:

$$P = \begin{bmatrix} 3 & 0 \\ 3 & 5 \end{bmatrix} \text{ και } H = P^{-1} = \begin{bmatrix} \frac{1}{3} & 0 \\ -\frac{1}{5} & \frac{1}{5} \end{bmatrix}$$

$$D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Επομένως:

$$H \cdot D = \begin{bmatrix} \frac{1}{3} & 0 \\ -\frac{1}{5} & \frac{1}{5} \end{bmatrix}$$

Παρατηρούμε ότι ο πίνακας *HD* περιέχει θετικά και αρνητικά στοιχεία, επομένως το tiling που εκφράζεται από τον πίνακα *H* είναι άκυρο για το συγκεκριμένο πρόγραμμα.

♦

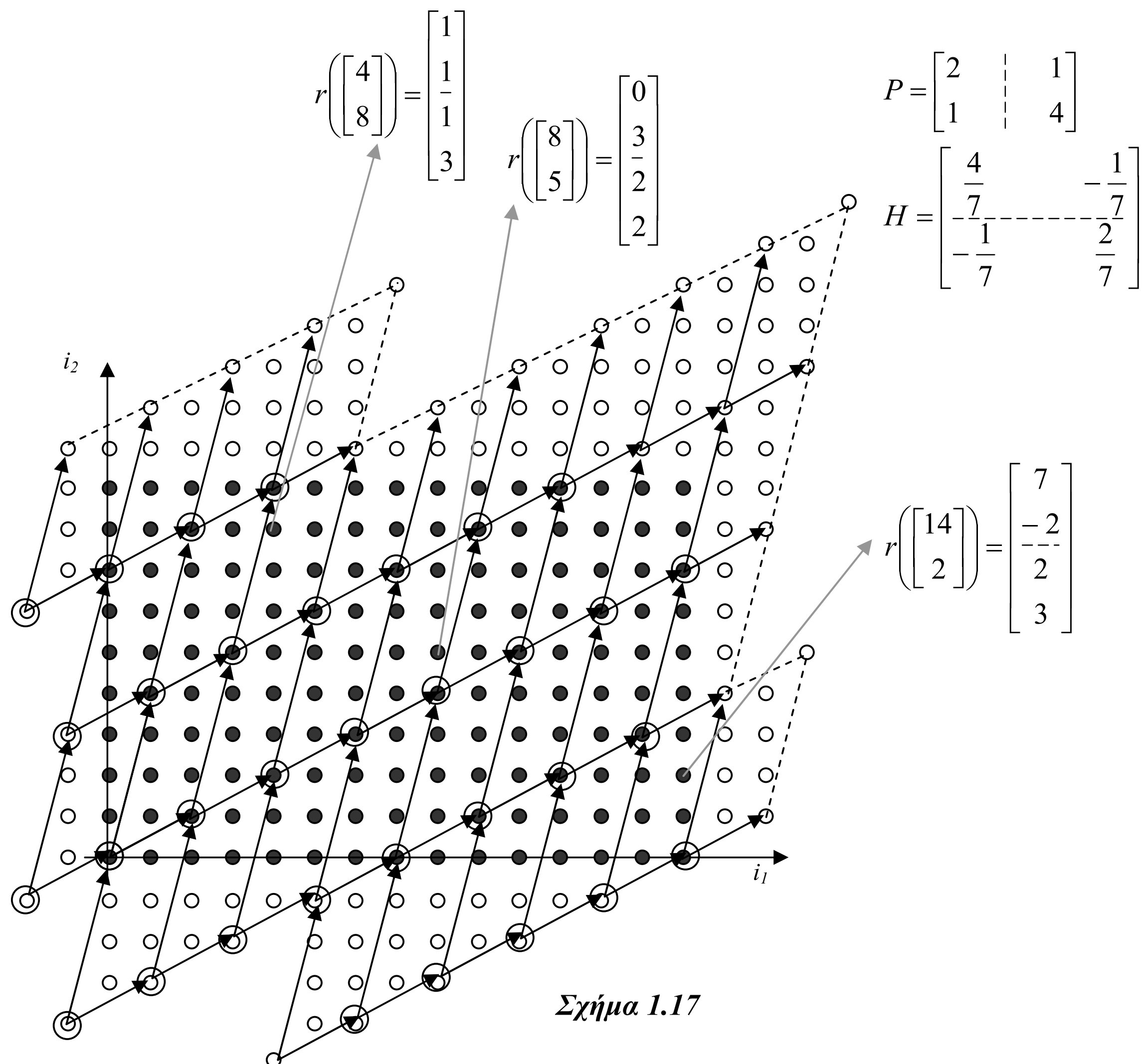
1.7 Μετασχηματισμός Υπερκόμβων (Supernode Transformation)

Η διαδικασία του tiling μπορεί να οριστεί ως ένας μετασχηματισμός

$$r : Z^n \rightarrow Z^{2n}, r(i) = \begin{bmatrix} \lfloor H \cdot i \rfloor \\ i - H^{-1} \cdot \lfloor H \cdot i \rfloor \end{bmatrix}$$

Δηλαδή κάθε *n*-διάστατο iteration vector απεικονίζεται σε ένα *2n*-διάστατο διάνυσμα, το οποίο αποτελείται από το tile vector του tile $t = \lfloor H \cdot i \rfloor$ στο οποίο ανήκει η επανάληψη i και τη σχετική θέση του i ως προς την tile origin iteration $i_{toi} = H^{-1} \cdot t$ του tile.

Στο σχήμα 1.17 δίδεται ένα παράδειγμα του μετασχηματισμού αυτού.



Σχήμα 1.17

1.8 Κόστος υπολογισμού, επικοινωνίας των tiles

Ως ***κόστος υπολογισμού*** (***computation cost***) του tile ορίζεται ο αριθμός των ακέραιων σημείων του n-διάστατου χώρου που περιέχονται σε αυτό. Αν ο inverse tiling matrix $P = H^{-1}$ περιέχει μόνο ακέραια στοιχεία, τότε το κόστος υπολογισμού δίδεται από τον τύπο:

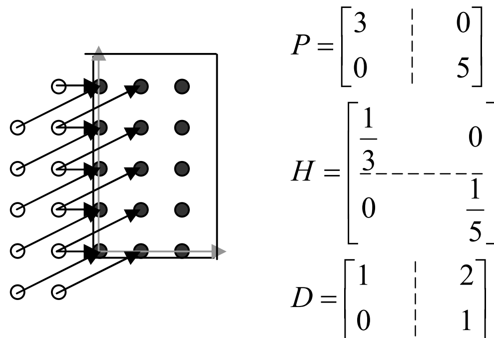
$$V_{comp}(H) = \frac{1}{|\det(H)|} = |\det(P)|$$

Στο παράδειγμα του σχήματος 1.17 είναι $V_{comp} = |\det(P)| = 7$

Ως ***κόστος επικοινωνίας*** (***communication cost***) ενός tile ορίζουμε την ποσότητα των δεδομένων που πρέπει να ληφθούν από κάποια γειτονικά tiles, που έχουν ήδη εκτελεστεί, ώστε να αρχίσει η εκτέλεση του εν λόγω tile.

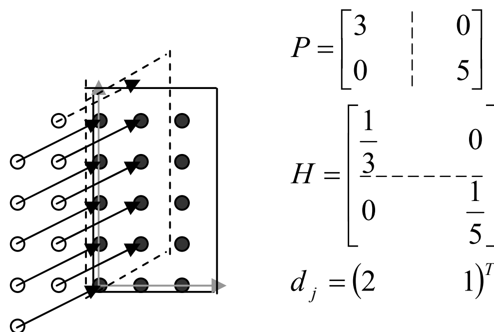
Προκειμένου να μετρήσουμε το κόστος επικοινωνίας, θεωρούμε τις εξής διαδοχικές προσεγγίσεις:

(α) Αρχικά το κόστος επικοινωνίας ενός tile μπορεί να θεωρηθεί ανάλογο του αριθμού των διανυσμάτων εξάρτησης που διασχίζουν τα όρια του tile και καταλήγουν μέσα σε αυτό. Πρόκειται για μία υπερεκτίμηση του κόστους επικοινωνίας, γιατί δύο διανύσματα εξάρτησης που ξεκινούν από το ίδιο σημείο θα έπρεπε σε μία ακριβή προσέγγιση να μετρώνται μία μόνο φορά και όχι δύο, όπως φαίνεται στο σχήμα 1.18.



Σχήμα 1.18

(β) Στη συνέχεια, σε μία ακόμη χαλαρότερη προσέγγιση, το κόστος επικοινωνίας μπορεί να μετρηθεί από τον αριθμό των διανυσμάτων εξάρτησης, τα οποία διασχίζουν τα όρια του tile. Με τον τρόπο αυτό μετράμε και τις εξαρτήσεις που διασχίζουν τα όρια του tile, άλλά δεν καταλήγουν μέσα σε αυτό, όπως το διάνυσμα που στο σχήμα 1.19 παριστάνεται με διακεκομμένο βέλος.



Σχήμα 1.19

(γ) Τέλος, χρησιμοποιώντας την υπόθεση (β), υπολογίζουμε ξεχωριστά το κόστος επικοινωνίας που διέρχεται διαμέσου κάθε πλευράς του tile και έπειτα τα προσθέτουμε.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Με τον τρόπο αυτό, όμως, υπολογίζουμε περισσότερες από μία φορές τις εξαρτήσεις που καταλήγουν στις ακμές του tile. Για παράδειγμα, στο σχήμα 1.19 η εξάρτηση που καταλήγει στο tile origin θα μετρηθεί δύο φορές, αφού κόβει και τις δύο πλευρές του παραλληλογράμμου.

Αν d_j είναι ένα διάνυσμα εξάρτησης, τότε ο αριθμός των εξαρτήσεων d_j που διασχίζουν την πλευρά h_i ή $p_1 \times \ldots \times p_{i-1} \times p_{i+1} \times \ldots \times p_n$ ισούται με τον όγκο του υπερπαραλληλεπιπέδου που ορίζεται από τα διανύσματα d_j, $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n$, δηλαδή με

$$\left|\det\left(\left[d_j, p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\right]\right)\right| = \left|d_j \cdot \left(p_1 \times \ldots \times p_{i-1} \times p_{i+1} \times \ldots \times p_n\right)\right| =$$
$$= \left|\det(P)\left(h_i \cdot d_j\right)\right| = \frac{1}{\left|\det(H)\right|} h_i \cdot d_j$$

όπου έχουμε θεωρήσει ότι σύμφωνα με την προϋπόθεση της ατομικότητας των tiles θα ισχύει $h_i \cdot d_j \geq 0$. Συνεπώς, το κόστος επικοινωνίας που εισάγεται από την πλευρά h_i του tile, θα είναι $\frac{1}{\left|\det(H)\right|} \sum_{j=1}^{m} h_i \cdot d_j$. Τέλος, ολόκληρο το κόστος επικοινωνίας του tile είναι το άθροισμα των επικοινωνιακών φορτίων διαμέσου όλων των πλευρών του, οπότε ισχύει:

$$\boxed{V_{comm}(H) = \frac{1}{\left|\det(H)\right|} \sum_{i=1}^{n} \sum_{j=1}^{m} h_i \cdot d_j = \frac{1}{\left|\det(H)\right|} \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{n} h_{ik} \cdot d_{kj}}$$

Ο τύπος αυτός αποτελεί μια αρκετά καλή προσέγγιση του κόστους επικοινωνίας, επειδή στην πράξη το μέγεθος του tile είναι πολύ μεγαλύτερο από τα μέτρα των διανυσμάτων εξάρτησης.

Αν τα tiles που βρίσκονται κατά μήκος μίας διάστασης του tile space ανατίθενται στον ίδιο επεξεργαστή, τότε τα διανύσματα εξάρτησης που τέμνουν τις αντίστοιχες πλευρές του tile δεν θα έπρεπε να λαμβάνονται υπ' όψη, γιατί δεν συνεπάγονται επικοινωνία μεταξύ των επεξεργαστών. Στην περίπτωση αυτή είναι ορθότερο το κόστος επικοινωνίας να υπολογίζεται από τον τύπο:

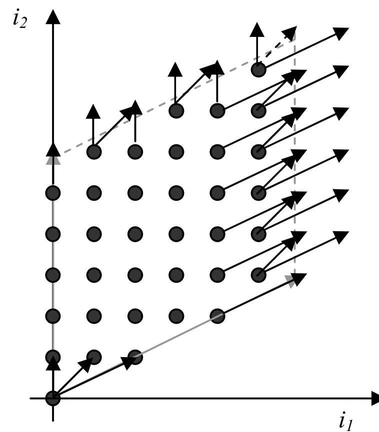
$$\boxed{V_{comm}(H) = \frac{1}{\left|\det(H)\right|} \sum_{i \in \{1, \ldots, x-1, x+1, \ldots, n\}} \sum_{j \in \{1, \ldots, m\}} h_i \cdot d_j}$$

όπου έχουμε θεωρήσει ότι τα tiles που βρίσκονται κατά μήκος της ίδιας ευθείας, που ορίζεται από το διάνυσμα h_x ανατίθενται στον ίδιο επεξεργαστή.

Παράδειγμα 1.6

Αν ένα tile ορίζεται, όπως φαίνεται στο σχήμα 1.20, από τον πίνακα $P = \begin{bmatrix} 6 & 0 \\ 3 & 6 \end{bmatrix}$

ή $H = P^{-1} = \begin{bmatrix} \frac{1}{6} & 0 \\ -\frac{1}{12} & \frac{1}{6} \end{bmatrix}$ και ο πίνακας εξαρτήσεων είναι $D = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$



Σχήμα 1.20

τότε το κόστος επικοινωνίας που διέρχεται από κάθε πλευρά του tile εξ’ αιτίας κάθε διανύσματος εξάρτησης είναι:

h_i \ d_j	1	2	3
1	1/3	1/6	0
2	0	1/12	1/6

Άρα $V_{comm} = \frac{1}{|\det(H)|} \sum_{i=1}^{n} \sum_{j=1}^{m} h_i \cdot d_j = 36 \cdot \left(\frac{1}{3} + \frac{1}{6} + 0 + 0 + \frac{1}{12} + \frac{1}{6} \right) = 27$

Στην πραγματικότητα ισχύει $V_{comm} = 26$. Η διαφορά αυτή οφείλεται στο γεγονός ότι το διάνυσμα που στο σχήμα εικονίζεται με διακεκομμένο βέλος υπολογίστηκε δύο φορές.

♦

Θεωρούμε ότι για δεδομένο κόστος υπολογισμού, ένα tiling είναι βέλτιστο αν αυτό εισάγει το μικρότερο δυνατό κόστος επικοινωνίας. Το πρόβλημα αυτό μαθηματικά εκφράζεται ως εξής:

$$\text{Ελαχιστοποίηση του } V_{comm}(H) = \frac{1}{|\det(H)|}\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{n} h_{ik}\cdot d_{kj}$$

$$\text{Με δεδομένα ότι } \begin{cases} V_{comp}(H) = \dfrac{1}{|\det(H)|} = v \\ H\cdot D \geq 0 \end{cases}$$

Επειδή η λύση του προβλήματος αυτού ξεφεύγει από το σκοπό της παρούσας εργασίας, δεν θα παρουσιαστεί αναλυτικά. Οι αναγνώστες που θα επιθυμούσαν περισσότερες λεπτομέρειες πάνω σε αυτό παραπέμπονται στο άρθρο "*Communication-Minimal Tiling of Uniform Dependence Loops*" του Jingling Xue [XUE97].

Στο εξής θα θεωρούμε ότι ο tiling matrix *H* είναι δεδομένος και θα ασχοληθούμε με το πρόβλημα της εύρεσης των ορίων του Tile Space και του τρόπου διάσχισης του κάθε tile.

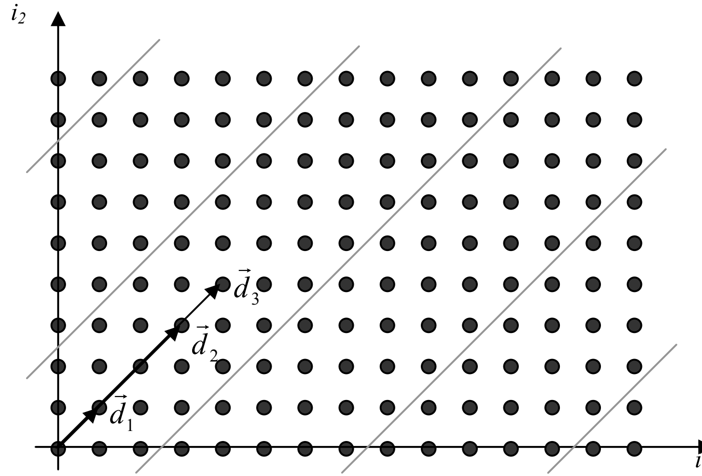
1.9 Tiling με μηδενικό κόστος επικοινωνίας

Στην περίπτωση που στόχος μας είναι η μείωση του χρόνου επεξεργασίας με παραλληλοποίηση του προγράμματος, αν η τάξη του $n \times m$ πίνακα *D* είναι $n_d < n$, τότε είναι δυνατόν ο *n*-διάστατος χώρος των επαναλήψεων του προγράμματος να χωριστεί με τη βοήθεια $n - n_d$ ομάδων παράλληλων μεταξύ τους υπερεπιπέδων σε περιοχές εντελώς ανεξάρτητες μεταξύ τους και σε κάθε επεξεργαστή να ανατεθεί μία από τις περιοχές αυτές. Τότε, ο κάθε επεξεργαστής θα πρέπει να διατρέξει τα σημεία που του αντιστοιχούν και να πραγματοποιήσει τους απαραίτητους υπολογισμούς, χωρίς να παρεμβάλλεται κανενός είδους επικοινωνία ή συγχρονισμός μεταξύ τους. Περισσότερες λεπτομέρειες πάνω στο θέμα αυτό μπορεί κανείς να βρει στα άρθρα [WOL91], [HOL92], [SHA91], [PEI89].

Παράδειγμα 1.7

Αν σε ένα δισδιάστατο χώρο επαναλήψεων ο πίνακας εξαρτήσεων έχει τη μορφή $D = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 3 & 4 \end{bmatrix}$ τάξης 1<2, τότε ο χώρος των επαναλήψεων μπορεί να χωριστεί όπως στο σχήμα 1.21. Ο χωρισμός αυτός δεν αποτελείται από έγκυρα tiles, αλλά στη συγκεκριμένη περίπτωση δίνει τα βέλτιστα αποτελέσματα.

♦



Σχήμα 1.21

Η μέθοδος αυτή δεν μπορεί να εφαρμοστεί σε όλες τις περιπτώσεις και για το λόγο αυτό δεν θα αναφερθούμε ξανά σε αυτήν. Θα μας απασχολήσουν μόνο πιο γενικές μέθοδοι.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Κεφάλαιο 2: Παρουσίαση μεθόδων επίλυσης αλγεβρικών προβλημάτων

2.1 Συστήματα ανισοτήτων

Ένα σύστημα l γραμμικών ανισοτήτων με n αγνώστους $i = (i_1, i_2, ..., i_n)$ μπορεί να παρασταθεί από έναν $l \times n$ πίνακα A και ένα n-διάστατο διάνυσμα $\vec{a}$ έτσι ώστε να ισχύει: $A \cdot i \leq \vec{a}$.

Ο χώρος των επαναλήψεων I^n, όπως ορίστηκε στην παράγραφο 1.1 μπορεί να παρασταθεί από τον πίνακα B και το διάνυσμα $\vec{b}$, που προκύπτουν ως εξής:

Έστω ότι η μεταβλητή i_k κυμαίνεται μεταξύ των ορίων:

$$l_k = \max\left(l_{k1}, ..., l_{kp_k}\right) \text{ και } u_k = \min\left(u_{k1}, ..., u_{kq_k}\right)$$

όπου:

$$l_{kr} = a_{kr0} + a_{kr1} i_1 + ... + a_{kr(k-1)} i_{(k-1)} = \frac{m_{kr0} + m_{kr1} i_1 + ... + m_{kr(k-1)} i_{(k-1)}}{n_{kr}},$$

$$u_{kr} = b_{kr0} + b_{kr1} i_1 + ... + b_{kr(k-1)} i_{(k-1)} = \frac{v_{kr0} + v_{kr1} i_1 + ... + v_{kr(k-1)} i_{(k-1)}}{w_{kr}}$$

και οι m_{krx}, n_{kr}, v_{krx}, w_{kr} είναι ακέραιοι αριθμοί.

Τότε οι ζητούμενοι πίνακες είναι:

$$B = \begin{pmatrix} 1 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 1 & 0 & ... & 0 \\ -1 & 0 & ... & 0 \\ ... & ... & ... & ... \\ -1 & 0 & ... & 0 \\ ... & ... & ... & ... \\ -b_{n11} & -b_{n12} & ... & 1 \\ ... & ... & ... & ... \\ -b_{nq_n1} & -b_{nq_n2} & ... & 1 \\ a_{n11} & a_{n12} & ... & -1 \\ ... & ... & ... & ... \\ a_{np_n1} & a_{np_n2} & ... & -1 \end{pmatrix} \quad \kappa\alpha\iota \quad \vec{b} = \begin{pmatrix} b_{110} \\ ... \\ b_{1q_10} \\ -a_{110} \\ ... \\ -a_{1p_10} \\ ... \\ b_{n10} \\ ... \\ b_{nq_n0} \\ -a_{n10} \\ ... \\ -a_{np_n0} \end{pmatrix}$$

Ισοδύναμα, αν θέλουμε να δουλεύουμε μόνο με ακέραιους πίνακες έχουμε:

$$B = \begin{pmatrix} w_{11} & 0 & ... & 0 \\ ... & ... & ... & ... \\ w_{1q_1} & 0 & ... & 0 \\ -n_{11} & 0 & ... & 0 \\ ... & ... & ... & ... \\ -n_{1p_1} & 0 & ... & 0 \\ ... & ... & ... & ... \\ -v_{n11} & -v_{n12} & ... & w_{n1} \\ ... & ... & ... & ... \\ -v_{nq_n1} & -v_{nq_n2} & ... & w_{nq_n} \\ m_{n11} & m_{n12} & ... & -n_{n1} \\ ... & ... & ... & ... \\ m_{np_n1} & m_{np_n2} & ... & -n_{np_n} \end{pmatrix} \quad \kappa\alpha\iota \quad \vec{b} = \begin{pmatrix} v_{110} \\ ... \\ v_{1q_10} \\ -m_{110} \\ ... \\ -m_{1p_10} \\ ... \\ v_{n10} \\ ... \\ v_{nq_n0} \\ -m_{n10} \\ ... \\ -m_{np_n0} \end{pmatrix}$$

© Μαρια Αθανασακη

Ασφαλώς οποιαδήποτε άλλη μορφή των πινάκων αυτών είναι αποδεκτή, αρκεί να εκφράζονται σωστά τα όρια του χώρου I^n από το σύστημα $B \cdot i \leq \vec{b}$

Επίσης, το σύνολο των σημείων $\boldsymbol{i}$ που ανήκουν στο tile με tile origin $\boldsymbol{i_{toi}}$ δίδονται, σαν συνέπεια του ορισμού 1.6, από το σύστημα ανισώσεων: $\vec{0} \leq H \cdot (i - i_{toi}) < \vec{1}$. Ισοδύναμα, αν g είναι ο ελάχιστος ακέραιος αριθμός με τον οποίο πρέπει να πολλαπλασιαστεί ο H για να προκύψει ακέραιος πίνακας, έχουμε:

$$\vec{0} \leq gH \cdot (i - i_{toi}) < \vec{g}$$

$$\Rightarrow \vec{0} \leq gH \cdot (i - i_{toi}) \leq \overrightarrow{(g-1)}$$

$$\Rightarrow \begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot (i - i_{toi}) \leq \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix}$$

Δηλαδή οι πίνακες που περιγράφουν το σύστημα αυτό είναι οι:

$$S = \begin{pmatrix} gH \\ -gH \end{pmatrix} \text{ και } \vec{s} = \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix}.$$

Παράδειγμα 2.1

Θεωρούμε το ακόλουθο πρόγραμμα φωλιασμένων βρόχων:

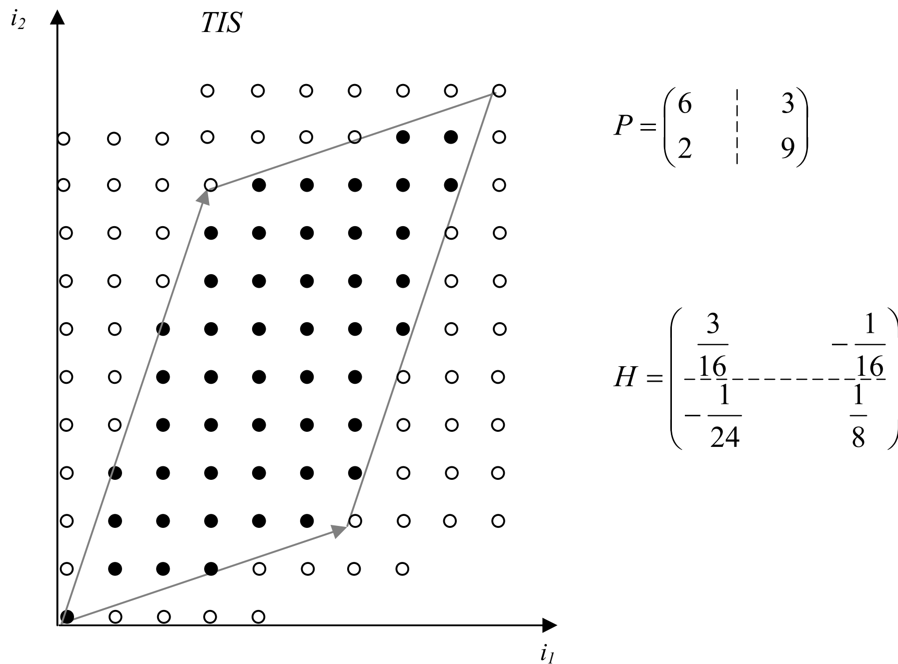
```
for (i1=0; i1<=15; i1++)
     for (i2=0; i2<=31; i2++)
     {
          X[i1][i2]=X[i1-3][i2-1]+X[i1-1][i2-3];
     }
```

Ο χώρος των επαναλήψεων είναι: $I^2 = \{(i_1,i_2)|\ 0 \leq i_1 \leq 15,\ 0 \leq i_2 \leq 31\}$. Το σύστημα ανισοτήτων που εκφράζει το χώρο αυτό είναι το εξής:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 15 \\ 0 \\ 31 \\ 0 \end{pmatrix}$$

Ο πίνακας εξαρτήσεων του προβλήματος είναι ο $D=\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$. Έστω ότι στο πρόβλημα αυτό εφαρμόζουμε μετασχηματισμό tiling με τη βοήθεια του tiling matrix $H=\begin{pmatrix} \frac{3}{16} & -\frac{1}{16} \\ -\frac{1}{24} & \frac{1}{8} \end{pmatrix}$. Ο μετασχηματισμός αυτό είναι έγκυρος, (σύμφωνα με τις προϋποθέσεις της παραγράφου 1.6) επειδή ισχύει $H\cdot D=\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{3} \end{pmatrix}\geq 0$, και όλα τα στοιχεία του πίνακα $P=H^{-1}=\begin{pmatrix} 6 & 3 \\ 2 & 9 \end{pmatrix}$ είναι ακέραια.

Επειδή ο ελάχιστος ακέραιος με τον οποίο πρέπει να πολλαπλασιαστεί ο H για να γίνει ακέραιος είναι g=48, έπεται ότι τα σημεία που ανήκουν στο tile με origin $\boldsymbol{i_{toi}}$ ικανοποιούν το σύστημα ανισοτήτων:

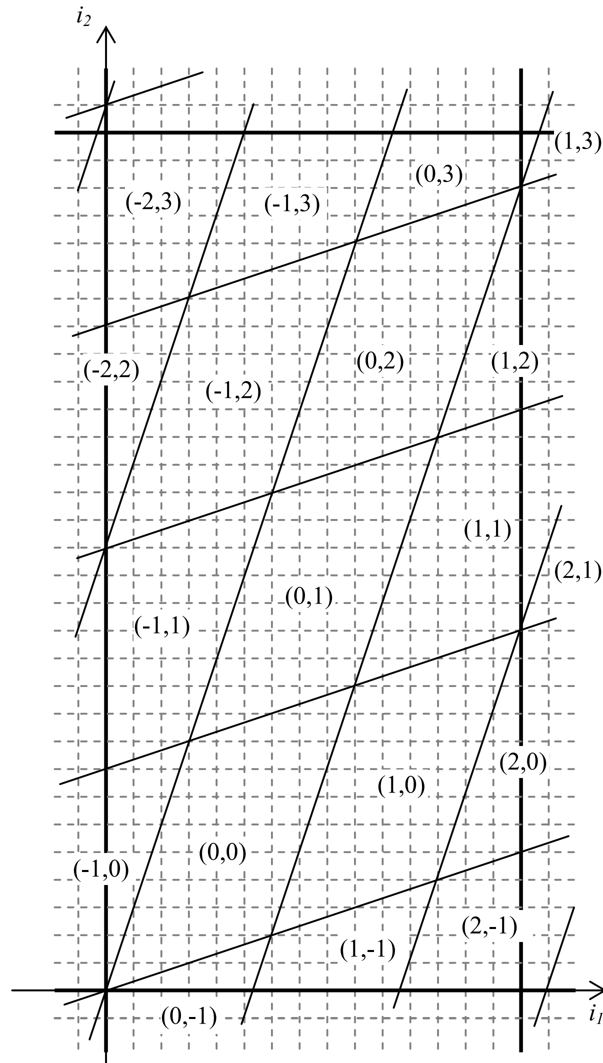


Σχήμα 2.1

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

$$\begin{pmatrix} 9 & -3 \\ -2 & 6 \\ -9 & 3 \\ 2 & -6 \end{pmatrix} \begin{pmatrix} i_1 - i_{toi,1} \\ i_2 - i_{toi,2} \end{pmatrix} \leq \begin{pmatrix} 47 \\ 47 \\ 0 \\ 0 \end{pmatrix}$$

Στο σχήμα 2.1 έχουμε σχεδιάσει το χώρο *TIS* που εκφράζεται από το τελευταίο σύστημα, και στο σχήμα 2.2 τα tiles στα οποία χωρίζεται ο χώρος I^n. Στο εσωτερικό κάθε tile έχουμε σημειώσει τις συντεταγμένες του στο χώρο I^S.



Σχήμα 2.2

♦

2.2 Μέθοδος απαλοιφής Fourier-Motzkin

Στο παράδειγμα 1.1 του προηγούμενου κεφαλαίου συμπεράναμε ότι τα όρια ενός χώρου, ο οποίος προκύπτει κατόπιν ενός μετασχηματισμού εκφράζονται από το εξής σύστημα ανισοτήτων:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -2 \\ 0 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & 2 & -2 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} \leq \begin{pmatrix} 10 \\ -1 \\ 1 \\ -1 \\ 20 \\ 0 \end{pmatrix}$$

Στο σύστημα αυτό παρατηρούμε ότι όλες οι ανισότητες εμπλέκουν τη μεταβλητή j_3, που αντιστοιχεί στον πιο εσωτερικό βρόχο. Επομένως καμία από αυτές δεν μπορεί να χρησιμεύσει ως άνω ή ως κάτω όριο των δεικτών των πιο εξωτερικών βρόχων. Αυτό σημαίνει ότι αν επιχειρήσουμε να γράψουμε το μετασχηματισμένο κώδικα με βάση τις ανισότητες αυτές, οι δείκτες j_1 και j_2 δεν θα είναι φραγμένοι και συνεπώς οι αντίστοιχοι βρόχοι θα είναι ατέρμονοι.

Στην πραγματικότητα, όμως μπορούμε να παράγουμε κάποια όρια για αυτούς ως εξής: Από τις ανισότητες $j_3 \leq 10$ και $j_2 - 2j_3 \leq 1 \Leftrightarrow j_2 \leq 1 + 2j_3$ συμπεραίνουμε ότι $j_2 \leq 1 + 2 \times 10 \Leftrightarrow j_2 \leq 21$. Όμοια από τις ανισότητες $-j_3 \leq -1 \Leftrightarrow j_3 \geq 1$ και $-j_2 + j_3 \leq -1 \Leftrightarrow j_2 \geq 1 + j_3$ συμπεραίνουμε ότι $j_2 \geq 1 + 1 \Leftrightarrow j_2 \geq 2$. Με τον τρόπο αυτό παρήγαμε ένα άνω και ένα κάτω όριο για τη μεταβλητή j_2. Με τον ίδιο τρόπο μπορούν να παραχθούν και τα όρια της μεταβλητής j_1. Αυτό είναι, άλλωστε, το αντικείμενο της παρούσας παραγράφου.

2.2.1 Περιγραφή της μεθόδου

Η μέθοδος απαλοιφής Fourier-Motzkin, όπως παρουσιάζεται στο άρθρο των A.J.C.Bik, H.A.G.Wijshoff "*Implementation of Fourier-Motzkin Elimination*" [BIK95], μπορεί να χρησιμοποιηθεί **(1)** για να ελέγξει αν έχει λύση ένα σύστημα ανισοτήτων $A\vec{x} \leq \vec{b}$ ή **(2)** για να μετατρέψει ένα σύστημα σε μορφή τέτοια ώστε τα κάτω και άνω όρια κάθε μεταβλητής x_m να δίδονται ως συνάρτηση των $x_1, x_2, ..., x_{m-1}$. Η ιδιότητα αυτή της μεθόδου είναι ιδιαίτερα χρήσιμη όταν θέλουμε να χρησιμοποιήσουμε φωλιασμένους βρόχους για να διατρέξουμε ένα σύνολο επαναλήψεων I^n το οποίο ορίζεται από ένα

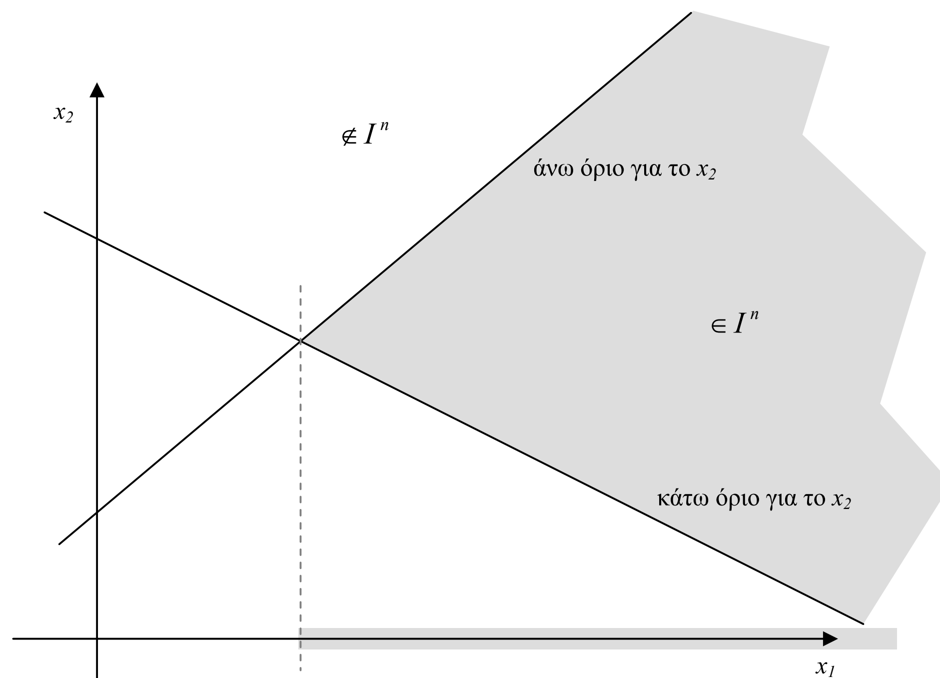
σύστημα ανισοτήτων της μορφής $A\vec{x} \le \vec{b}$. Προφανώς, θα πρέπει τα όρια του δείκτη i_k κάθε ενός από τους φωλιασμένους βρόχους να εκφραστούν συναρτήσει των *k-1* δεικτών των εξωτερικότερων βρόχων. Επομένως η μέθοδος απαλοιφής Fourier-Motzkin μπορεί να μετατρέψει έναν οποιοδήποτε πεπερασμένο κυρτό χώρο επαναλήψεων σε μορφή κατάλληλη για τα όρια φωλιασμένων βρόχων.

Η μέθοδος βασίζεται στην παρατήρηση ότι μια μεταβλητή x_m μπορεί να απαλειφθεί από ένα σύστημα ανισοτήτων αν αντικαταστήσουμε κάθε ζεύγος ανισοτήτων, που εκφράζουν ένα άνω και ένα κάτω όριο, από το όριο που προκύπτει ως εξής:

$$\left.\begin{array}{l} L \le c_1 \cdot x_m \\ c_2 \cdot x_2 \le U \end{array}\right\} \Rightarrow c_2 \cdot L \le c_1 \cdot U$$

όπου $c_1 > 0$ και $c_2 > 0$.

Μετά την απαλοιφή αυτή θα προκύψει ένα νέο σύστημα ανισοτήτων που δεν θα εμπλέκει καθόλου τη μεταβλητή x_m. Γεωμετρικά η εκτέλεση της παραπάνω πράξης για κάθε ζεύγος άνω και κάτω ορίου του x_m ουσιαστικά ισοδυναμεί με την εύρεση της προβολής του χώρου που ορίζεται από τις ανισότητες $L \le c_1 \cdot x_m$ και $c_2 \cdot x_2 \le U$ πάνω σε επίπεδο κάθετο στον άξονα των x_m. Το γεγονός αυτό έχει παρασταθεί γραφικά για ένα δισδιάστατο πρόβλημα στο σχήμα 2.3.



Σχήμα 2.3

Κατά την εφαρμογή της μεθόδου, από το σύστημα $A\vec{x} \le \vec{b}$, όπου A είναι ένας $d \times n$ πίνακας, όχι απαραίτητα τετραγωνικός, απαλείφουμε διαδοχικά τη μεταβλητή x_m αρχίζοντας με *m=n* και μειώνοντας κάθε φορά την τιμή του *m* κατά *1.*

Για κάθε τιμή του *m* γνωρίζουμε τον πίνακα *A*, διαστάσεων $d \times m$ και το διάνυσμα $\vec{b}$ διάστασης *d*. Επομένως το σύστημα $A \cdot \vec{x} \le \vec{b}$ μπορεί να γραφεί με την μορφή των ανισώσεων

$$\sum_{j=1}^{m} a_{ij} \cdot x_j \le b_i \text{ για } 1 \le i \le d \qquad (2.2.1)$$

Αναδιατάσσουμε το παραπάνω σύστημα σύμφωνα με τις τιμές των στοιχείων a_{im}, έτσι ώστε για συγκεκριμένα $p, q \in N$ να ισχύει

$$a_{im} > 0 \text{ για κάθε } 1 \le i \le p$$
$$a_{im} < 0 \text{ για κάθε } p < i \le q$$
$$a_{im} = 0 \text{ για κάθε } q < i \le d$$

και $p \le q \le d$

Από την αναδιάταξη αυτή προκύπτουν τρία σύνολα ανισώσεων, στα οποία οι συντελεστές του αγνώστου x_m είναι όλοι θετικοί ακέραιοι αριθμοί.

$$a_{im} \cdot x_m \le b_i - \sum_{j=1}^{m-1} a_{ij} \cdot x_j \text{ με } 1 \le i \le p$$
$$-b_i + \sum_{j=1}^{m-1} a_{ij} \cdot x_j \le (-a_{im}) x_m \text{ με } p < i \le q \qquad (2.2.2)$$
$$\sum_{j=1}^{m-1} a_{ij} \cdot x_j \le b_i \text{ με } q < i \le d$$

Οι πρώτες *p* ανισότητες ορίζουν το πάνω όριο της συντεταγμένης x_m, το οποίο ισούται με το ελάχιστο (minimum) των $\left\lfloor \frac{b_i - \sum_{j=1}^{m-1} a_{ij} \cdot x_j}{a_{im}} \right\rfloor$, $1 \le i \le p$.

Οι επόμενες $q-p$ ανισότητες ορίζουν το κάτω όριο της συντεταγμένης x_m το οποίο ισούται με το μέγιστο (maximum) των $\left\lceil \dfrac{-b_i + \sum_{j=1}^{m-1} a_{ij} \cdot x_j}{-a_{im}} \right\rceil$, $p < i \le q$.

Στην περίπτωση που κάποιος από τους συντελεστές $a_{im}=1$, τότε στον κώδικα που θα παραχθεί δεν χρειάζεται να τοποθετήσουμε την αντίστοιχη συνάρτηση ακεραίου μέρους προς τα άνω ή προς τα κάτω. Επίσης αν έχουμε μόνο ένα άνω όριο ($p=1$) παραλείπουμε την συνάρτηση υπολογισμού του ελαχίστου. Όμοια, αν έχουμε μόνο ένα κάτω όριο ($q=p+1$), τότε παραλείπουμε την συνάρτηση υπολογισμού του μεγίστου.

Μετά την εύρεση αυτών των ορίων, απαλείφουμε από το σύστημα τον άγνωστο x_m και προχωρούμε στην ανεύρεση των ορίων των εξωτερικότερων βρόχων.

Προκειμένου να απαλείψουμε τον άγνωστο x_m πρέπει να αντικαταστήσουμε κάθε ζευγάρι ανισοτήτων της μορφής $L \le c_1 \cdot x_m$ και $c_2 \cdot x_m \le U$, όπου c_1, c_2>0, από την ανισότητα $\frac{L}{c_1} \le \frac{U}{c_2}$. Αν στο αρχικό σύστημα είχαμε μόνο ακεραίους και θέλουμε να εξακολουθήσουμε να δουλεύουμε μόνο με ακεραίους αριθμούς, πρέπει να αντικαταστήσουμε το παραπάνω ζευγάρι από την ανισότητα $c_2 \cdot L \le c_1 \cdot U$. Για να αποφύγουμε κατά το δυνατόν τους πολύ μεγάλους αριθμούς, μπορούμε να διαιρέσουμε και τα δύο μέλη της ανισότητας που προκύπτει με το μέγιστο κοινό διαιρέτη των c_1 και c_2. Δηλαδή τελικά προκύπτει η ανισότητα $\frac{c_2}{\gcd(c_1, c_2)} \cdot L \le \frac{c_1}{\gcd(c_1, c_2)} \cdot U$.

Επομένως ο δείκτης του πιο εσωτερικού βρόχου απαλείφεται από το σύστημα της μορφής *(2.2.2)* αντικαθιστώντας τις πρώτες q ανισότητες από τις ακόλουθες $p(q-p)$ ανισότητες:

$$\sum_{j=1}^{m-1} \left(c_{i'} \cdot a_{ij} + c_i \cdot a_{i'j} \right) x_j \le c_{i'} \cdot b_i + c_i \cdot b_{i'}$$

όπου $c_i = \dfrac{a_{im}}{\gcd(a_{i'm}, a_{im})}$ και $c_{i'} = -\dfrac{a_{i'm}}{\gcd(a_{i'm}, a_{im})}$

για όλα τα i,i': $1 \le i \le p$ και $p < i' \le q$

Οι $p(q-p)$ αυτές ανισότητες μαζί με τις τελευταίες $d-q$ ανισότητες του συστήματος *(2.2.2)* δίνουν ένα νέο σύστημα της μορφής *(2.2.1)* με κάποια διαφορετική τιμή για το d και μία μικρότερη κατά 1 τιμή για το m.

Αν εφαρμόσουμε τα παραπάνω βήματα *n* φορές, με αρχική τιμή του *m*=*n*, τελικά θα έχουμε βρει τα όρια του περιγραφόμενου χώρου σε μορφή κατάλληλη για την περιγραφή τους σε φωλιασμένους βρόχους, δηλαδή τα όρια κάθε συντεταγμένης x_i του $\vec{x}$ θα εκφράζονται συναρτήσει μόνο των x_j με $j < i$.

Γενικά είναι δυνατόν σε κάποιο στάδιο εφαρμογής της μεθόδου να εμφανιστεί μία ανίσωση της μορφής $0 \le c$ με $c < 0$, ή το maximum των κάτω ορίων της μεταβλητής x_1 να είναι μεγαλύτερο από το minimum των άνω ορίων της. Τότε πρόκειται για έναν κενό υπόχωρο του Z^n, δηλαδή το αρχικό σύστημα $A \cdot \vec{x} \le \vec{b}$ είναι αδύνατο. Επίσης είναι πιθανόν στη γενική περίπτωση κάποιες μεταβλητές να μην έχουν άνω ή κάτω όριο. Στις περιπτώσεις, όμως, που θα εξετάσουμε στην παρούσα εργασία δεν είναι δυνατόν να συμβεί αυτό, επειδή θα ασχοληθούμε μόνο με φραγμένα υποσύνολα του Z^n.

<u>*Ανακεφαλαιώνοντας*</u>, κατά την εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin, σε κάθε βήμα *m*=*n*,...,*1* ένα σύστημα ανισώσεων $A \cdot \vec{x} \le \vec{b}$ με κάποιον $d \times m$ πίνακα A και $\vec{x} = (x_1, \ \ldots, \ x_m)^T$ αναπαρίσταται από τον επαυξημένο κατά μία στήλη πίνακα $(A \quad | \quad \vec{b})$ διαστάσεων $d \times (m+1)$. Οι γραμμές αυτού του πίνακα αναδιατάσσονται σύμφωνα με τις τιμές των στοιχείων της *m*-οστής στήλης. Στη συνέχεια για όλους τους δυνατούς συνδυασμούς των *i,i΄*: $1 \le i \le p$ και $p < i' \le q$ οι γραμμές *i* και *i΄* προστίθενται αφού πολλαπλασιαστούν με τους παράγοντες $-\frac{a_{i'm}}{\gcd(a_{i'm}, a_{im})}$ και $\frac{a_{im}}{\gcd(a_{i'm}, a_{im})}$ αντιστοίχως. Οι γραμμές που προκύπτουν μαζί με τις τελευταίες *d-q* γραμμές του προηγούμενου πίνακα συνιστούν τις γραμμές του καινούριου πίνακα.

Αυτή η διαδικασία επαναλαμβάνεται στον πίνακα που προκύπτει έως ότου απαλειφθούν από το σύστημα όλοι οι άγνωστοι.

Συνεπώς παίρνουμε διαδοχικά μία ακολουθία πινάκων επαυξημένων κατά μία στήλη:

$$\left(A^{(n)} \quad | \quad \vec{b}^{(n)}\right) \to \ldots \to \left(A^{(1)} \quad | \quad \vec{b}^{(1)}\right) \to \vec{b}^{(0)}$$

Κάθε πίνακας αυτής της ακολουθίας $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ με $1 \le m \le n$ έχει τις δικές του τοπικές παραμέτρους $p^{(m)}$, $q^{(m)}$, και $d^{(m)}$ και αναπαριστά τα όρια της συντεταγμένης x_m στην επιθυμητή μορφή σύμφωνα με το ακόλουθο σύστημα ανισοτήτων:

$$A^{(m)} \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{pmatrix} \le \vec{b}^{(m)}$$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Αν ο τελευταίος πίνακας-στήλη $\vec{b}^{(0)}$ της ακολουθίας περιέχει τουλάχιστον 1 αρνητικό στοιχείο, $b_i^{(0)} < 0$, τότε υπονοείται κάποια ανισότητα της μορφής $0 \le b_i^{(0)} < 0$. Επομένως το αρχικό σύστημα ανισοτήτων δεν έχει λύση. Διαφορετικά, το αρχικό σύστημα έχει τουλάχιστον μία πραγματική λύση. Αν όμως μας ενδιαφέρουν μόνο οι ακέραιες τιμές του διανύσματος $\vec{x}$, τότε δεν είναι σίγουρο ότι υπάρχει ακέραια λύση. Συνεπώς η μέθοδος αυτή παρέχει μια αναγκαία, αλλά όχι ικανή συνθήκη για την ύπαρξη ακέραιας λύσης ενός συστήματος ανισοτήτων.

Παράδειγμα 2.2

Έστω ότι έχουμε τρεις φωλιασμένους βρόχους και συνεπώς ο χώρος των επαναλήψεων είναι 3-διάστατος. Έστω ότι από τα διανύσματα των εξαρτήσεων για τη συγκεκριμένη περίπτωση έχουμε με κάποιον αλγόριθμο βρει ότι το βέλτιστο tiling του 3-διάστατου χώρου περιγράφεται από τους πίνακες:

$$P = \begin{pmatrix} 12 & 3 & -8 \\ 6 & -3 & 2 \\ -6 & 3 & 4 \end{pmatrix} \text{ και } H = P^{-1} = \begin{pmatrix} \frac{1}{18} & \frac{1}{9} & \frac{1}{18} \\ \frac{1}{9} & 0 & \frac{2}{9} \\ 0 & \frac{1}{6} & \frac{1}{6} \end{pmatrix}$$

Τότε g=18 και επομένως το σύστημα $\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot \left(\vec{i} - \vec{i}_{toi}\right) \le \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix}$ για τη συγκεκριμένη περίπτωση γράφεται:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 0 & 4 \\ 0 & 3 & 3 \\ -1 & -2 & -1 \\ -2 & 0 & -4 \\ 0 & -3 & -3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \le \begin{pmatrix} 17 \\ 17 \\ 17 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

όπου $\vec{x} = \left(\vec{i} - \vec{i}_{toi}\right)$

Προκειμένου να διατρέξουμε το tile θα πρέπει να εκφράσουμε τα όρια των συντεταγμένων x_i του $\vec{x}$ συναρτήσει των $x_j, j < i$. Για να το επιτύχουμε αυτό μετασχηματίζουμε το τελευταίο σύστημα σύμφωνα με τη μέθοδο απαλοιφής *Fourier-Motzkin*.

Έτσι, παίρνουμε την εξής ακολουθία πινάκων:

$$\begin{pmatrix} gH & \vert & (g-1)\vec{1} \\ -gH & \vert & \vec{0} \end{pmatrix} = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 17 \\ 2 & 0 & 4 & 17 \\ 0 & 3 & 3 & 17 \\ -1 & -2 & -1 & 0 \\ -2 & 0 & -4 & 0 \\ 0 & -3 & -3 & 0 \end{array}\right) \rightarrow \text{με αναδιάταξη} \rightarrow$$

$$\left(A^{(3)} \quad \vert \quad \vec{b}^{(3)}\right) = \left(\begin{array}{ccc|c} 2 & 0 & 4 & 17 \\ 0 & 3 & 3 & 17 \\ 1 & 2 & 1 & 17 \\ \hdashline -1 & -2 & -1 & 0 \\ 0 & -3 & -3 & 0 \\ -2 & 0 & -4 & 0 \end{array}\right)$$

$$\rightarrow \text{με απαλοιφή του } x_3 \rightarrow \left(\begin{array}{cc|c} -2 & -8 & 17 \\ 6 & -12 & 51 \\ 0 & 0 & 17 \\ -3 & -3 & 17 \\ 0 & 0 & 17 \\ -6 & 12 & 68 \\ 0 & 0 & 17 \\ 3 & 3 & 51 \\ 2 & 8 & 68 \end{array}\right) \rightarrow \text{με αναδιάταξη} \rightarrow$$

$$\left(A^{(2)} \quad \vert \quad \vec{b}^{(2)}\right) = \left(\begin{array}{cc|c} -6 & 12 & 68 \\ 2 & 8 & 68 \\ 3 & 3 & 51 \\ \hdashline -3 & -3 & 17 \\ -2 & -8 & 17 \\ 6 & -12 & 51 \\ \hdashline 0 & 0 & 17 \\ 0 & 0 & 17 \\ 0 & 0 & 17 \end{array}\right)$$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

$$\rightarrow \text{με απαλοιφή του } x_2 \rightarrow \left(\begin{array}{c|c} -18 & 136 \\ -18 & 187 \\ 0 & 119 \\ -18 & 340 \\ 0 & 85 \\ 18 & 306 \\ 0 & 68 \\ 18 & 459 \\ 18 & 255 \\ 0 & 17 \\ 0 & 17 \\ 0 & 17 \end{array}\right) \rightarrow \text{με αναδιάταξη} \rightarrow$$

$$\left(A^{(1)} \mid \vec{b}^{(1)}\right) = \left(\begin{array}{c|c} 18 & 306 \\ 18 & 459 \\ 18 & 255 \\ \hdashline -18 & 136 \\ -18 & 187 \\ -18 & 340 \\ \hdashline 0 & 119 \\ 0 & 85 \\ 0 & 68 \\ 0 & 17 \\ 0 & 17 \\ 0 & 17 \end{array}\right) \rightarrow \text{με απαλοιφή ττο } x_1 \rightarrow \vec{b}^{(0)} = \begin{pmatrix} 442 \\ 493 \\ 646 \\ 595 \\ 646 \\ 799 \\ 391 \\ 442 \\ 595 \\ 119 \\ 85 \\ 68 \\ 17 \\ 17 \\ 17 \end{pmatrix}$$

Επειδή το διάνυσμα $\vec{b}^{(0)}$ αποτελείται, όπως αναμέναμε μόνο από θετικά στοιχεία, το αρχικό σύστημα ανισοτήτων είναι επιλύσιμο. Τα όρια των βρόχων όπως προκύπτουν από τους παραπάνω επαυξημένους πίνακες είναι τα εξής:

for $\left(x_1 = \max\left(\left\lceil \frac{-136}{18} \right\rceil, \left\lceil \frac{-187}{18} \right\rceil, \left\lceil \frac{-340}{18} \right\rceil\right);\ x_1 \le \min\left(\left\lfloor \frac{306}{18} \right\rfloor, \left\lfloor \frac{459}{18} \right\rfloor, \left\lfloor \frac{255}{18} \right\rfloor\right);\ x_1{+}{+}\right)$

for $\left(x_2 = \max\left(\left\lceil \frac{-17-3x_1}{3} \right\rceil, \left\lceil \frac{-17-2x_1}{8} \right\rceil, \left\lceil \frac{-51+6x_1}{12} \right\rceil\right);\ x_2 \le \min\left(\left\lfloor \frac{68+6x_1}{12} \right\rfloor, \left\lfloor \frac{68-2x_1}{8} \right\rfloor, \left\lfloor \frac{51-3x_1}{3} \right\rfloor\right);\ x_2{+}{+}\right)$

for $\left(x_3 = \max\left(-x_1-2x_2, \left\lceil \frac{-3x_2}{3} \right\rceil, \left\lceil \frac{-2x_1}{4} \right\rceil\right);\ x_3 \le \min\left(\left\lfloor \frac{17-2x_1}{4} \right\rfloor, \left\lfloor \frac{17-3x_2}{3} \right\rfloor, 17-x_1-2x_2\right);\ x_3{+}{+}\right)$

{ Εντολές σώματος φωλιασμένων βρόχων }

Στο πρόγραμμα που προκύπτει παρατηρούμε ότι σε ορισμένες περιπτώσεις οι συναρτήσεις άνω ή κάτω ακεραίου μέρους δεν είναι απαραίτητες, παρόλο που ο παρονομαστής είναι διάφορος του 1. Δηλαδή ο παρονομαστής μπορεί να είναι διαιρέτης του αριθμητή. Αυτό μπορεί να αποφευχθεί αν πριν την παραγωγή των ορίων απλοποιήσουμε τους επαυξημένους πίνακες από τους οποίους προκύπτουν τα όρια των φωλιασμένων βρόχων διαιρώντας κάθε γραμμή τους με το μέγιστο κοινό διαιρέτη των στοιχείων της γραμμής αυτής. Για παράδειγμα η εντολή

$$x_3 = \max\left(-x_1-2x_2, \left\lceil \frac{-3x_2}{3} \right\rceil, \left\lceil \frac{-2x_1}{4} \right\rceil\right)$$

γράφεται απλούστερα

$$x_3 = \max\left(-x_1-2x_2, -x_2, \left\lceil \frac{-x_1}{2} \right\rceil\right)$$

Τέλος, παρατηρούμε ότι το κάτω όριο του πιο εξωτερικού βρόχου ισούται πάντα με $\left\lceil \frac{-136}{18} \right\rceil = -7$ ενώ το άνω όριο μπορεί να αντικατασταθεί από την απλούστερη έκφραση $\left\lfloor \frac{255}{18} \right\rfloor = 14$. Γενικά, είναι πιθανόν να μπορούν να γίνουν και άλλες απλοποιήσεις, οι οποίες είναι λιγότερο προφανείς.

♦

Παράδειγμα 2.3

Έστω ότι αρχικά έχουμε το εξής σύστημα ανισοτήτων:

$$\begin{pmatrix} 0 & -1 & -3 \\ 0 & 0 & -1 \\ -1 & 0 & 6 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 0 & -6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \le \begin{pmatrix} -10 \\ -1 \\ -1 \\ 15 \\ 3 \\ 50 \end{pmatrix}.$$

Η εφαρμογή της μεθόδου Fourier-Motzkin δίνει την εξής ακολουθία επαυξημένων πινάκων:

$$\left(\begin{array}{ccc|c} -1 & 0 & 6 & -1 \\ 0 & 1 & 3 & 15 \\ 0 & 0 & 1 & 3 \\ \hdashline 0 & 0 & -1 & -1 \\ 0 & -1 & -3 & -10 \\ 1 & 0 & -6 & 50 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 0 & 1 & 12 \\ 1 & 2 & 80 \\ \hdashline -1 & -2 & -21 \\ 0 & -1 & -1 \\ \hdashline 0 & 0 & 5 \\ 0 & 0 & 49 \\ 0 & 0 & 2 \\ -1 & 0 & -7 \\ 1 & 0 & 68 \end{array}\right) \rightarrow$$

$$\left(\begin{array}{c|c} 1 & 78 \\ 1 & 68 \\ \hdashline -1 & 3 \\ -1 & -7 \\ \hdashline 0 & 5 \\ 0 & 49 \\ 0 & 2 \\ 0 & 59 \\ 0 & 11 \end{array}\right) \rightarrow \begin{pmatrix} 81 \\ 71 \\ 7 \\ 61 \\ 5 \\ 49 \\ 2 \\ 59 \\ 11 \end{pmatrix}$$

Επειδή όλα τα στοιχεία του τελικού πίνακα-διανύσματος είναι θετικά, το αρχικό σύστημα έχει λύση. Τα όρια των x_1, x_2 είναι:

$$\max(-3,7) \le x_1 \le \min(78,68)$$

$$\max\left(\left\lceil \frac{21 - x_1}{2} \right\rceil, 1\right) \le x_2 \le \min\left(12, \left\lfloor \frac{80 - x_1}{2} \right\rfloor\right)$$

Προφανώς η έκφραση $\max(-3,7) \le x_1 \le \min(78,68)$ μπορεί απλούστερα να γραφτεί $7 \le x_1 \le 68$. Πιθανόν, όμως, να μπορούν να γίνουν και άλλες απλοποιήσεις που δεν είναι τόσο προφανείς. Αυτό, άλλωστε θα είναι και το θέμα με το οποίο θα ασχοληθούμε στο υπόλοιπο της παρούσας παραγράφου.

♦

2.2.2 Απαλοιφή περιττών ανισοτήτων

Κατ' αρχήν, για να απλοποιήσουμε το σύστημα ανισοτήτων που προκύπτει πρέπει από κάθε επαυξημένο πίνακα $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ να απαλείψουμε τις ανισότητες που δεν αναφέρονται στη μεταβλητή x_m, δηλαδή τις γραμμές του πίνακα που το m-οστό στοιχείο τους ισούται με 0, $a_{im}{}^{(m)} = 0$. Οι γραμμές αυτές χρησίμεψαν για την εύρεση του πίνακα $\left(A^{(m-1)} \quad | \quad \vec{b}^{(m-1)}\right)$, αλλά έπειτα είναι άχρηστες.

Στη συνέχεια, ανισότητες που εμπλέκουν τη μεταβλητή x_m μπορεί να είναι άχρηστες επειδή υπερκαλύπτονται από κάποιες άλλες ανισότητες. Η απαλοιφή αυτών των περιττών ορίων θα έχει ως αποτέλεσμα την παραγωγή πιο αποδοτικού κώδικα, αφού κατά τη διάρκεια εκτέλεσης του παραγόμενου προγράμματος θα πρέπει να αποτιμηθούν λιγότερες εκφράσεις. Επιπλέον αν σε κάποια περίπτωση το άνω ή το κάτω όριο μιας μεταβλητής περιγράφεται από μία μόνο ανισότητα, τότε δεν χρειάζεται η αποτίμηση της συνάρτησης ελαχίστου ή μεγίστου.

Για τον εντοπισμό τους προτείνονται δύο μέθοδοι: η μέθοδος απλοποίησης Ad-Hoc, η οποία έχει μικρό υπολογιστικό κόστος, αλλά δεν καταφέρνει να εντοπίσει όλα τα περιττά όρια, και η μέθοδος ακριβούς απλοποίησης, η οποία έχει μεγάλο υπολογιστικό κόστος και ανιχνεύει όλα τα περιττά όρια των μεταβλητών x_m. Δηλαδή η μέθοδος ακριβούς απλοποίησης βελτιώνει την αποδοτικότητα του παραγόμενου κώδικα για παράλληλη επεξεργασία με κόστος την αύξηση του χρόνου μεταγλώττισης του προγράμματος. Πρακτικά, προκειμένου να απαλείψουμε όλες τις περιττές ανισότητες με όσο το δυνατόν μικρότερο κόστος μπορούμε να εφαρμόσουμε τη μέθοδο απλοποίησης Ad-Hoc και στη συνέχεια στο σύστημα που προκύπτει να εφαρμόσουμε τη μέθοδο ακριβούς απλοποίησης.

Απλοποίηση Ad-Hoc

Για κάθε μεταβλητή βρόχου x_m στο σύστημα φωλιασμένων βρόχων βάθους n έχουμε τέσσερις μεταβλητές $l_m^{\min}$, $l_m^{\max}$ και $u_m^{\min}$, $u_m^{\max}$, οι οποίες μπορούν να πάρουν τιμές από το σύνολο $Z \cup \{-\infty,+\infty\}$. Οι μεταβλητές $l_m^{\min}$ και $l_m^{\max}$ θα καταγράψουν την ελάχιστη και τη

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

μέγιστη τιμή αντίστοιχα του κάτω ορίου αυτού του δείκτη βρόχου. Όμοια, οι μεταβλητές $u_m^{\min}$ και $u_m^{\max}$ θα καταγράψουν την ελάχιστη και τη μέγιστη τιμή αντίστοιχα του άνω ορίου αυτού του δείκτη βρόχου. Αυτό σημαίνει ότι ο δείκτης x_m θα μπορεί να πάρει τιμές στο διάστημα $\left[l_m^{\min}, u_m^{\max}\right]$.

Αρχικοποιούμε τις μεταβλητές μας με τις τιμές $l_m^{\min} = l_m^{\max} = -\infty$ και $u_m^{\min} = u_m^{\max} = +\infty$ για κάθε $1 \le m \le n$. Στη συνέχεια οι μεταβλητές αυτές παίρνουν ακέραιες τιμές κατά τη διάρκεια μιας προς τα πίσω επισκόπησης όλων των επαυξημένων κατά μία στήλη πινάκων $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ που κατασκευάσαμε κατά την εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin. Δηλαδή αυτή τη φορά διατρέχουμε τα όρια των βρόχων κατ' αύξουσα σειρά του δείκτη m.

Για κάθε $d \times (m+1)$ πίνακα $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ εξετάζουμε όλες τις γραμμές με $\alpha_{im} \neq 0$, οι οποίες έχουν απομείνει. Τα αντίστοιχα άνω και κάτω όρια εκφράζονται με την εξής μορφή:

$$\left\lfloor \frac{b_i + \sum_{j=1}^{m-1} (-a_{ij}) x_j}{a_{im}} \right\rfloor \qquad (2.2.3) \qquad \text{ή} \qquad \left\lceil \frac{b_i + \sum_{j=1}^{m-1} (-a_{ij}) x_j}{a_{im}} \right\rceil \qquad (2.2.4)$$

Έστω ότι για όλα τα $j<m$ έχουμε ήδη βρει τα όρια μέσα στα οποία κινείται η μεταβλητή x_j: $x_j \in \left[l_j^{\min}, u_j^{\max}\right]$. Τότε η ελάχιστη τιμή l' και η μέγιστη τιμή u' του αριθμητή των εκφράσεων *(2.2.3)* και *(2.2.4)* μπορεί να αποτιμηθεί ως εξής:

$$l' = b_i + \sum_{j=1}^{m-1} (-a_{ij})^+ \cdot l_j^{\min} - (-a_{ij})^- \cdot u_j^{\max}$$

$$u' = b_i + \sum_{j=1}^{m-1} (-a_{ij})^+ \cdot u_j^{\max} - (-a_{ij})^- \cdot l_j^{\min}$$

όπου $a^+ = \max(a,0)$ και $a^- = \max(-a,0)$

Επομένως αν $\alpha_{im} > 0$ το αντίστοιχο άνω όριο κινείται στο διάστημα $[l,u]$ όπου $l = \left\lfloor \frac{l'}{a_{im}} \right\rfloor$ και $u = \left\lfloor \frac{u'}{a_{im}} \right\rfloor$. Αν ισχύει $u_m^{\max} \le l$ αυτό το άνω όριο είναι περιττό αφού σε κάθε περίπτωση υπερκαλύπτεται από τα άνω όρια που εξετάσαμε προηγουμένως και απαλείφεται. Όμοια αν $u \le u_m^{\min}$, αυτό το άνω όριο μπορεί να αντικαταστήσει όλα όσα

εξετάσαμε προηγουμένως για την συγκεκριμένη μεταβλητή x_m. Αφότου ένα άνω όριο έχει εξετασθεί με τον τρόπο που μόλις περιγράψαμε, εκτελούμε τις ακόλουθες αναθέσεις:

$$u_m^{\min} := \min\left(u_m^{\min}, l\right)$$
$$u_m^{\max} := \min\left(u_m^{\max}, u\right)$$

Όμοια, αν a_{im}<0 το αντίστοιχο κάτω όριο κινείται στο διάστημα $[l,u]$ όπου $l = \left\lceil \frac{u'}{a_{im}} \right\rceil$ και $u = \left\lceil \frac{l'}{a_{im}} \right\rceil$. Αν ισχύει $u \le l_m^{\min}$ αυτό το κάτω όριο είναι περιττό αφού σε κάθε περίπτωση υπερκαλύπτεται από τα κάτω όρια που εξετάσαμε προηγουμένως και απαλείφεται. Αν $l_m^{\max} \le l$, αυτό το κάτω όριο μπορεί να αντικαταστήσει όλα όσα εξετάσαμε προηγουμένως για την συγκεκριμένη μεταβλητή x_m. Αφότου ένα κάτω όριο έχει εξετασθεί με τον τρόπο που μόλις περιγράψαμε, εκτελούμε τις ακόλουθες αναθέσεις:

$$l_m^{\min} := \max\left(l_m^{\min}, l\right)$$
$$l_m^{\max} := \max\left(l_m^{\max}, u\right)$$

Αν συνεχίσουμε με τον ίδιο τρόπο για όλα τα x_m τελικά θα πάρουμε μια νέα ακολουθία πινάκων $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ από την οποία θα έχουν απαλειφθεί κάποιες περιττές γραμμές, δηλαδή κάποια περιττά όρια.

Παράδειγμα 2.4

Έστω ότι τα όρια ενός 2-διάστατου χώρου εκφράζονται από τις ανισότητες

$$\begin{pmatrix} 0 & -1 \\ -1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} -1 \\ -1 \\ 3 \\ 4 \end{pmatrix},$$

που αντιπροσωπεύονται από τον επαυξημένο πίνακα $\left(\begin{array}{cc|c} 0 & -1 & -1 \\ -1 & 1 & -1 \\ 0 & 1 & 3 \\ 1 & 0 & 4 \end{array}\right)$.

Αν εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin παίρνουμε την εξής ακολουθία επαυξημένων πινάκων:

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

$$\left(\begin{array}{cc|c} -1 & 1 & -1 \\ 0 & 1 & 3 \\ \hline 0 & -1 & -1 \\ \hline 1 & 0 & 4 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 4 \\ \hline -1 & -2 \\ \hline 0 & 2 \end{array}\right) \rightarrow \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

Επειδή το τελικό διάνυσμα στήλη δεν περιέχει αρνητικά στοιχεία, ο χώρος που περιγράφεται από το αρχικό σύστημα ανισοτήτων δεν είναι κενός. Επομένως, στην ακολουθία πινάκων που προέκυψε μπορούμε να εφαρμόσουμε τη μέθοδο απλοποίησης Ad-Hoc προκειμένου να εντοπίσουμε κάποιες περιττές ανισότητες. Πράγματι, μετά από εξέταση του 2ου πίνακα της ακολουθίας, $\left(\begin{array}{c|c} 1 & 4 \\ \hline -1 & -2 \\ \hline 0 & 2 \end{array}\right)$ παίρνουμε $u_1^{\min} = u_1^{\max} = 4$ και $l_1^{\min} = l_1^{\max} = 2$. Στη συνέχεια από την 1η γραμμή του πίνακα $\left(\begin{array}{cc|c} -1 & 1 & -1 \\ 0 & 1 & 3 \\ \hline 0 & -1 & -1 \\ \hline 1 & 0 & 4 \end{array}\right)$ παίρνουμε $u_2^{\min} = 1$ και $u_2^{\max} = 3$. Για την επόμενη γραμμή του πίνακα, η οποία εκφράζει άλλο ένα άνω όριο της μεταβλητής x_2, ισχύει $l = u = 3$. Δηλαδή $u_2^{max} \leq l$, οπότε αυτό το άνω όριο μπορεί να απαλειφθεί.

Επομένως, τελικά τα χρήσιμα όρια του αρχικού χώρου στην επιθυμητή μορφή περιγράφονται από τους πίνακες $\left(\begin{array}{cc|c} -1 & 1 & -1 \\ \hline 0 & -1 & -1 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 4 \\ \hline -1 & 2 \end{array}\right)$.

♦

Παράδειγμα 2.5

Έστω ότι τα όρια ενός 2-διάστατου χώρου εκφράζονται από τις ανισότητες

$$\begin{pmatrix} -1 & -2 \\ 1 & -2 \\ 1 & 2 \\ -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} -3 \\ 0 \\ 300 \\ -1 \\ 100 \end{pmatrix},$$

Αν εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin παίρνουμε την εξής ακολουθία επαυξημένων πινάκων:

$$\left(\begin{array}{cc|c} 1 & 2 & 300 \\ 0 & 1 & 100 \\ \hdashline -1 & -2 & -3 \\ 1 & -1 & 0 \\ \hdashline -1 & 0 & -1 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 100 \\ 1 & 100 \\ \hdashline -1 & 197 \\ -1 & -1 \\ \hdashline 0 & 594 \end{array}\right) \rightarrow \begin{pmatrix} 297 \\ 99 \\ 297 \\ 99 \\ 594 \end{pmatrix}$$

Επειδή το τελικό διάνυσμα στήλη δεν περιέχει αρνητικά στοιχεία, ο χώρος που περιγράφεται από το αρχικό σύστημα ανισοτήτων δεν είναι κενός. Επομένως στην ακολουθία πινάκων που προέκυψε μπορούμε να εφαρμόσουμε τη μέθοδο απλοποίησης Ad-Hoc προκειμένου να εντοπίσουμε κάποιες περιττές ανισότητες. Πράγματι, μετά από εξέταση του 2ου πίνακα της ακολουθίας, $\left(\begin{array}{c|c} 1 & 100 \\ 1 & 100 \\ \hdashline -1 & 197 \\ -1 & -1 \\ \hdashline 0 & 594 \end{array}\right)$ παίρνουμε:

$$u_1^{\min} = u_1^{\max} = 100 \text{ και } l_1^{\min} = l_1^{\max} = 1.$$

Επίσης προκύπτει ότι το 1ο ή το 2ο άνω όριο, καθώς και το 1ο κάτω όριο μπορούν να απαλειφθούν. Στη συνέχεια από την 1η γραμμή του πίνακα $\left(\begin{array}{cc|c} 1 & 2 & 300 \\ 0 & 1 & 100 \\ \hdashline -1 & -2 & -3 \\ 1 & -1 & 0 \\ \hdashline -1 & 0 & -1 \end{array}\right)$ παίρνουμε:

$$u_2^{\min} = \left\lfloor \frac{300-100}{2} \right\rfloor = 100 \text{ και } u_2^{\max} = \left\lfloor \frac{300-1}{2} \right\rfloor = 149.$$

Για την επόμενη γραμμή του πίνακα, η οποία εκφράζει άλλο ένα άνω όριο της μεταβλητής x_2, ισχύει $l = u = 100$. Δηλαδή $u \leq u_2^{min}$, οπότε αυτό το άνω όριο μπορεί να αντικαταστήσει το προηγούμενο.

Μετά από όμοια βήματα και για τα κάτω όρια που εκφράζονται από τον πίνακα αυτό, συμπεραίνουμε τελικά ότι τα χρήσιμα όρια του αρχικού χώρου στην επιθυμητή μορφή περιγράφονται από τους πίνακες $\left(\begin{array}{cc|c} 0 & 1 & 100 \\ \hdashline 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 100 \\ \hdashline -1 & -1 \end{array}\right)$.

♦

Ακριβής απλοποίηση

Η μέθοδος ακριβούς απλοποίησης βασίζεται στην παρατήρηση ότι μια συγκεκριμένη ανισότητα υπερκαλύπτεται από άλλες ανισότητες σε ένα σύστημα ανισώσεων αν το σύστημα που προκύπτει με αντιστροφή της ανισότητας αυτής είναι αδύνατο. Για παράδειγμα η αντιστροφή της ανισότητας $x \le 20$ στο ακόλουθο σύστημα ανισώσεων είναι: $x > 20$, το οποίο εφ' όσον πρόκειται για ακέραιες μεταβλητές γράφεται: $21 \le x$.

$$\begin{matrix} 0 \le x \le 13 \\ x \le 20 \end{matrix} \xrightarrow{\text{αντιστροφή}} \begin{matrix} 0 \le x \le 13 \\ 21 \le x \end{matrix}$$

Η εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin στο δεύτερο σύστημα δίνει $0 \le 13$ και $21 \le 13$, από όπου φαίνεται ότι το σύστημα δεν έχει λύση. Επομένως η τρίτη ανισότητα δεν χρειάζεται και μπορεί να απαλειφθεί.

Κατά την εφαρμογή της μεθόδου ακριβούς απλοποίησης, κατασκευάζουμε βηματικά για διαδοχικές τιμές του *m=1,…,n,* έναν πίνακα της μορφής:

$$\left(\begin{array}{ccccc|c} A^{(1)} & \vec{0} & \vec{0} & \ldots & \vec{0} & \vec{b}^{(1)} \\ & A^{(2)} & \vec{0} & \ldots & \vec{0} & \vec{b}^{(2)} \\ & . & & & & . \\ & & . & & & . \\ & & & . & & . \\ & & & & A^{(m)} & \vec{b}^{(m)} \end{array} \right) \qquad (2.2.5)$$

όπου $\vec{0}$ είναι ένα διάνυσμα-στήλη κατάλληλης διάστασης.

Σε κάθε βήμα *m* ο πίνακας αυτός εκφράζει τα όρια των *m* πρώτων μεταβλητών. Ο αριθμός των θετικών στοιχείων της *m*-οστής στήλης του επαυξημένου πίνακα $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ ισούται με τον αριθμό των άνω ορίων της μεταβλητής x_m. Όμοια, ο αριθμός των αρνητικών στοιχείων της *m*-οστής στήλης του $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$ ισούται με τον αριθμό των κάτω ορίων της μεταβλητής x_m. Αν υπάρχουν παραπάνω από ένα άνω όρια, τότε αντιστρέφουμε το πρώτο από αυτά και στο σύστημα που προκύπτει εφαρμόζουμε τη μέθοδο απαλοιφής Fourier-Motzkin προκειμένου να ελέγξουμε αν έχει λύση. Αν το σύστημα έχει λύση, τότε αποκαθιστούμε το όριο στην αρχική του μορφή, και επαναλαμβάνουμε τη διαδικασία για το επόμενο άνω όριο. Αλλιώς το όριο αυτό απαλείφεται, τόσο από τον πίνακα της μορφής *(2.2.5),* όσο και από τον πίνακα $\left(A^{(m)} \quad | \quad \vec{b}^{(m)}\right)$. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να εξετάσουμε όλα τα άνω όρια ή να απομείνει μόνο ένα άνω όριο. Παρόμοια βήματα εφαρμόζουμε αν υπάρχουν

περισσότερα από ένα κάτω όρια. Στη συνέχεια αυξάνουμε την τιμή του m κατά *1* και κατασκευάζουμε τον επόμενο πίνακα της μορφής *(2.2.5)*. Επαναλαμβάνουμε την παραπάνω διαδικασία έως ότου το m πάρει την τιμή n.

Σε έναν πίνακα η αντιστροφή ενός άνω ή κάτω ορίου το οποίο απεικονίζεται στην *ι*-οστή σειρά γίνεται ως εξής:

$$\left(\begin{array}{ccc|c} & . & & . \\ & . & & . \\ & . & & . \\ a_{i1} & \ldots & a_{im} & b_i \\ & . & & . \\ & . & & . \\ & . & & . \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} & . & & . \\ & . & & . \\ & . & & . \\ -a_{i1} & \ldots & -a_{im} & -b_i-1 \\ & . & & . \\ & . & & . \\ & . & & . \end{array}\right)$$

Η εξέταση κάθε πίνακα της μορφής *(2.2.5)* ξεχωριστά για κάθε τιμή του m εγγυάται ότι θα απαλειφθούν μόνο τα όρια που είναι περιττά σε σχέση με τις πιθανές τιμές των δεικτών των εξωτερικότερων βρόχων. Αντίθετα, αν χρησιμοποιούσαμε μόνο τον τελευταίο πίνακα της μορφής (2.2.5) με $m=n$ για να ελέγξουμε αν είναι περιττά τα όρια των πιο εσωτερικών βρόχων, τότε θα απαλείφονταν κάποια όρια τα οποία θα υπερκαλύπτονται από όρια των πιο εσωτερικών μεταβλητών. Αυτό σημαίνει ότι για κάποιες τιμές της μεταβλητής x_m οι πιο εσωτερικές μεταβλητές $x_{m+1}, x_{m+2}, \ldots$ θα εκτελούν άεργα loops. Το γεγονός αυτό, δηλαδή, θα επιβαρύνει το χρόνο επεξεργασίας του παραγόμενου κώδικα.

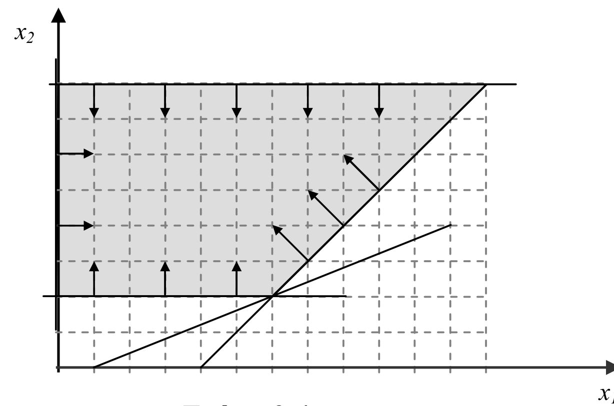
Επίσης αν επιτρέψουμε την απαλοιφή ενός μοναδικού άνω ή κάτω ορίου μιας μεταβλητής, τότε θα προκύψουν ατελείωτα loops.

Παράδειγμα 2.6

Το ακόλουθο σύστημα γραμμικών ανισοτήτων περιγράφει ένα κυρτό δισδιάστατο πολύγωνο, το οποίο έχει σχεδιαστεί στο σχήμα 2.4.

$$\begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 2 & -5 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 4 \\ 2 \\ -2 \\ 8 \end{pmatrix}$$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ



Σχήμα 2.4

Αν στο σύστημα αυτό εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin και στη συνέχεια απλοποιήσουμε την προκύπτουσα ακολουθία πινάκων με τη μέθοδο Ad-Hoc, θα πάρουμε την ακολουθία:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hdashline 1 & -1 & 4 \\ 2 & -5 & 2 \\ 0 & -1 & -2 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ \hdashline -1 & 0 \end{array}\right)$$

Επειδή η ακολουθία αυτή ορίζει ένα μόνο άνω και ένα κάτω όριο για τη μεταβλητή x_1 δεν μπορούμε να κάνουμε καμιά επιπλέον απλοποίηση στο δεύτερο πίνακα. Για τη μεταβλητή x_2 ορίζονται 3 κάτω όρια. Επομένως πρέπει να ελέγξουμε αν κάποιο από αυτά είναι περιττό. Για το σκοπό αυτό κατασκευάζουμε τον πίνακα της μορφής (2.2.5):

$$\left(\begin{array}{cc|c} 1 & 0 & 12 \\ \hdashline -1 & 0 & 0 \\ \hdashline 0 & 1 & 8 \\ \hdashline 1 & -1 & 4 \\ 2 & -5 & 2 \\ 0 & -1 & -2 \end{array}\right)$$

Στη συνέχεια αντιστρέφουμε διαδοχικά την 4η,την 5η και την 6η σειρά του και ελέγχουμε αν το σύστημα που προκύπτει έχει λύση.

Με αντιστροφή της 4ης γραμμής του πίνακα και εφαρμογή της μεθόδου Fourier-Motzkin παίρνουμε την εξής ακολουθία:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hdashline -1 & 1 & -5 \\ \hdashline 2 & -5 & 2 \\ 0 & -1 & -2 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 21 \\ 1 & 12 \\ -1 & -8 \\ -1 & -7 \\ -1 & 0 \\ 0 & 6 \end{array}\right) \rightarrow \begin{pmatrix} 13 \\ 14 \\ 21 \\ 4 \\ 5 \\ 12 \\ 6 \end{pmatrix}$$

Επειδή το τελικό διάνυσμα-στήλη αποτελείται από θετικά στοιχεία μόνο, το υπό εξέταση όριο πρέπει να αποκατασταθεί όπως ήταν. Στη συνέχεια αντιστρέφουμε την επόμενη γραμμή του πίνακα και εφαρμόζουμε πάλι τη μέθοδο Fourier-Motzkin. Η ακολουθία που προκύπτει είναι η εξής:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hdashline -2 & 5 & -3 \\ \hdashline 1 & -1 & 4 \\ 0 & -1 & -2 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ 1 & 5 \\ 1 & 12 \\ -1 & -7 \\ -1 & 0 \\ 0 & 6 \end{array}\right) \rightarrow \begin{pmatrix} 5 \\ 12 \\ -2 \\ 5 \\ 5 \\ 12 \\ 6 \end{pmatrix}$$

Επειδή το τελικό διάνυσμα-στήλη περιέχει ένα αρνητικό στοιχείο, το υπό εξέταση όριο απαλείφεται. Στη συνέχεια αντιστρέφουμε την επόμενη γραμμή του πίνακα και εφαρμόζουμε πάλι τη μέθοδο Fourier-Motzkin. Η ακολουθία που προκύπτει είναι η εξής:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hdashline 0 & 1 & 1 \\ \hdashline 1 & -1 & 4 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ 1 & 5 \\ 1 & 12 \\ -1 & 0 \end{array}\right) \rightarrow \begin{pmatrix} 12 \\ 5 \\ 12 \end{pmatrix}$$

Αφού το τελικό διάνυσμα δεν περιέχει κανένα αρνητικό στοιχείο, το υπό εξέταση όριο αποκαθίσταται στην αρχική μορφή του. Τελικά τα ακριβή όρια που περιγράφουν τον αρχικό χώρο στην επιθυμητή μορφή περιγράφονται από την ακολουθία πινάκων:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hdashline 1 & -1 & 4 \\ 0 & -1 & -2 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ \hdashline -1 & 0 \end{array}\right)$$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Στο σημείο αυτό παρατηρούμε ότι η ανισότητα $x_1 \leq 12$ είναι περιττή σε σχέση με όλο το υπόλοιπο σύστημα ανισοτήτων, όπως φαίνεται και στο σχήμα 2.4. Παρ' όλα αυτά δεν πρέπει να απαλειφθεί, επειδή είναι το μοναδικό άνω όριο της μεταβλητής x_1.

♦

Παράδειγμα 2.7

Συνεχίζοντας το παράδειγμα 1.1, προκειμένου να παράγουμε τον τελικό κώδικα, πρέπει να εκφράσουμε το σύστημα ανισοτήτων

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -2 \\ 0 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & 2 & -2 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} \leq \begin{pmatrix} 10 \\ -1 \\ 1 \\ -1 \\ 20 \\ 0 \end{pmatrix}$$

σε μορφή κατάλληλη για τη γραφή των ορίων των τελικών βρόχων. Εφαρμόζοντας στο σύστημα αυτό τη μέθοδο απαλοιφής Fourier-Motzkin και στη συνέχεια κάνοντας τις απαραίτητες απλοποιήσεις, καταλήγουμε στο εξής σύστημα για τα όρια του μετασχηματισμένου χώρου:

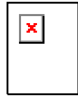
$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 2 & 0 \\ 0 & -1 & 0 \\ 1 & -1 & 0 \\ 2 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -2 \\ -1 & 2 & -2 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} \leq \begin{pmatrix} 29 \\ -2 \\ 21 \\ 20 \\ -2 \\ 19 \\ 41 \\ 10 \\ -1 \\ 20 \\ -1 \\ 1 \\ 0 \end{pmatrix}$$

Επομένως ο μετασχηματισμένος κώδικας είναι ο εξής:

```
for (j1=2; j1<=29; j1++)
     for (j2=max(2,j1-19,2j1-41);
                                j2<=min(21,floor((j1+20)/2)); j2++)
          for (j3=max(1,ceil((j2-1)/2),ceil((2j2-j1)/2));
                                   j3<=min(10,j2-1,j2-j1+20); j3++)
          {
               Ομάδα εντολών
          }
```

♦

2.2.3 Πολυπλοκότητα της μεθόδου απαλοιφής Fourier-Motzkin

Ας υποθέσουμε ότι το αρχικό σύστημα ανισοτήτων αποτελείται από d ανισότητες με n αγνώστους. Στη χειρότερη περίπτωση οι μισές από αυτές *(d/2)* αποτελούν ένα άνω όριο για τη μεταβλητή x_n και οι άλλες μισές *(d/2)* αποτελούν ένα κάτω όριο για τη μεταβλητή x_n. Τότε από το πρώτο βήμα της μεθόδου Fourier-Motzkin θα προκύψουν σύμφωνα με την παραπάνω περιγραφή $\left(\frac{d}{2}\right)\cdot\left(\frac{d}{2}\right)=\left(\frac{d}{2}\right)^2$ νέες ανισότητες, οι οποίες εκφράζονται με τις  γραμμές ενός πίνακα. Και πάλι στη χειρότερη περίπτωση οι μισές από αυτές αποτελούν ένα άνω όριο για τη μεταβλητή x_{n-1} και οι άλλες μισές αποτελούν ένα κάτω όριο για τη μεταβλητή x_{n-1}. Συνεχίζοντας με τον ίδιον τρόπο, συμπεραίνουμε ότι οι πίνακες της προκύπτουσας ακολουθίας έχουν τον εξής αριθμό γραμμών:

$$d \rightarrow \frac{d^2}{4} \rightarrow \frac{d^4}{64} \rightarrow \ldots$$

το πολύ. Δηλαδή από το *i*-οστό βήμα της μεθόδου παράγεται ένας πίνακας με $\frac{d^{2^i}}{2^{2^{(i+1)}-2}}$ γραμμές στη χειρότερη περίπτωση. Για την ολοκλήρωση του αλγορίθμου απαιτούνται n βήματα. Συνεπώς ο τελευταίος πίνακας της ακολουθίας (πίνακας – διάνυσμα) θα αποτελείται από $\frac{d^{2^n}}{2^{2^{(n+1)}-2}}$ γραμμές. Ο αριθμός αυτός εκφράζει και την πολυπλοκότητα της μεθόδου. Δηλαδή έχουμε:

$$\textit{Πολυπλοκότητα} = O\left(\frac{d^{2^n}}{2^{2^{(n+1)}-2}}\right) \approx O\left(\left(\frac{d}{2}\right)^{2^n}\right) \qquad (2.2.6)$$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Επομένως η πολυπλοκότητα του αλγορίθμου εξαρτάται διπλά εκθετικά από τον αριθμό των μεταβλητών του αρχικού συστήματος.

2.3 Lattices – Hermite Normal Form

Ορισμός 2.1

Έστω A ένας πίνακας διαστάσεων $m \times n$. Ονομάζουμε ***lattice*** του πίνακα A το σύνολο των σημείων $\mathcal{L}(A)=\{y: y=Ax \wedge x \in Z^n\}$.

Επομένως, οι τρύπες που δημιουργούνται στο μετασχηματισμένο χώρο αν εφαρμόσουμε έναν non-unimodular μετασχηματισμό T, όπως περιγράψαμε στο πρώτο κεφάλαιο, είναι τα σημεία $j \in Z^n$, τα οποία $j \notin \mathcal{L}(T)$. Αντίθετα, πραγματικά σημεία του μετασχηματισμένου χώρου είναι τα $j \in \mathcal{L}(T)$.

Προφανώς, αν ισχύει $B=AT \Leftrightarrow A=BT^{-1}$, όπου T είναι ένας unimodular πίνακας, τότε κάθε διάνυσμα-στήλη του πίνακα B γράφεται ως ακέραιος γραμμικός συνδυασμός των στηλών του A, δηλαδή κάθε διάνυσμα-στήλη του πίνακα B ανήκει στο $\mathcal{L}(A)$. Αντίστροφα, κάθε διάνυσμα-στήλη του πίνακα A γράφεται ως ακέραιος γραμμικός συνδυασμός των στηλών του B, δηλαδή κάθε διάνυσμα-στήλη του πίνακα A ανήκει στο $\mathcal{L}(B)$. Επομένως οι πίνακες A και B έχουν το ίδιο ακριβώς lattice, $\mathcal{L}(A)=\mathcal{L}(B)$.

Ορισμός 2.2

Ο τετραγωνικός, αντιστρέψιμος πίνακας $H=[\vec{h}_1, \vec{h}_2, ..., \vec{h}_n] \in R^{n \times n}$ είναι σε ***Κανονική Ερμητιανή Μορφή Στηλών*** *(Column Hermite Normal Form, HNF)*, αν και μόνο αν:

- Ο H είναι κάτω τριγωνικός ($h_{ij} \neq 0 \Rightarrow i \geq j$)
- Για κάθε $i>j$, ισχύει $0 \leq h_{ij} < h_{ii}$ (Σε κάθε γραμμή μεγαλύτερο είναι αυτό που βρίσκεται στη διαγώνιο του πίνακα και όλα τα στοιχεία του πίνακα είναι θετικά.)

Διαισθητικά, σύμφωνα με τον ορισμό αυτό, ένας πίνακας H είναι σε ερμητιανή μορφή αν φαίνεται ως εξής:

$$H = \begin{bmatrix} d_{11} & 0 & \cdots & 0 \\ h_{21} & d_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & d_{nn} \end{bmatrix},$$

όπου όλα τα στοιχεία είναι μη αρνητικά και σε κάθε γραμμή ισχύει $d_{ii} > h_{i1}, \ldots, h_{i(i-1)}$.

Ορισμός 2.3

Γενικότερα, ο πίνακας $H=[\vec{h}_1, \vec{h}_2, ..., \vec{h}_n] \in R^{n\times n}$, ο οποίος μπορεί να μην είναι αντιστρέψιμος, ούτε τετραγωνικός, είναι σε ***Κανονική Ερμητιανή Μορφή Στηλών*** *(Column Hermite Normal Form, HNF)*, αν και μόνο αν:

- Υπάρχουν $1\leq i_1<\ldots<i_n$ τέτοιοι ώστε $h_{ij}\neq 0 \Rightarrow i>i_j$. (Δηλαδή το ύψος κάθε στήλης είναι μικρότερο από το ύψος της προηγούμενης.)
- Για κάθε $k>j$, ισχύει $0\leq h_{i_j k} < h_{i_j j}$. (Το κορυφαίο μη μηδενικό στοιχείο κάθε στήλης είναι το μεγαλύτερο της αντίστοιχης γραμμής.)

Για κάθε $m\times n$ πίνακα A, του οποίου οι γραμμές είναι γραμμικώς ανεξάρτητες, υπάρχει ένας $n\times n$ unimodular πίνακας T, τέτοιος ώστε $AT=\begin{bmatrix}\widetilde{A} & 0\end{bmatrix}$ και ο $\widetilde{A}$ να έχει κανονική ερμητιανή μορφή στηλών. Αφού ο πίνακας $\widetilde{A}$ προκύπτει από τον A με unimodular μετασχηματισμό, έπεται σύμφωνα με όσα αναφέραμε παραπάνω ότι $\mathcal{L}(A)=\mathcal{L}(\widetilde{A})$. Επίσης, *η ερμητιανή κανονική μορφή $\widetilde{A}$ κάθε πίνακα A είναι μοναδική*. Όσοι αναγνώστες επιθυμούν περισσότερες λεπτομέρειες πάνω στο θέμα αυτό μπορούν να ανατρέξουν στο βιβλίο του A.Schrijver “*Theory of linear and integer programming*” [SCH86].

Στη συνέχεια θα δώσουμε τον αλγόριθμο υπολογισμού της μορφής HNF $\widetilde{A}$ κάθε ακέραιου πίνακα $A\in Z^{n\times n}$. Προηγουμένως, όμως είναι απαραίτητο να αναφερθούμε στον ευκλείδειο αλγόριθμο για τον υπολογισμό του μέγιστου κοινού διαιρέτη.

Ευκλείδειος αλγόριθμος υπολογισμού του Μέγιστου Κοινού Διαιρέτη

Ο αλγόριθμος του Ευκλείδη για τον υπολογισμό του μέγιστου κοινού διαιρέτη (ΜΚΔ) δύο ακεραίων αριθμών, υπολογίζει μια ακολουθία τιμών $\mu_1>\mu_2>...>\mu_{v-1}>\mu_v=0$ όλες από τις οποίες έχουν τον ίδιο ΜΚΔ. Ο αλγόριθμος σταματάει όταν η ακολουθία φθάσει στο 0 και τότε η τελευταία μη μηδενική τιμή είναι ο ζητούμενος μέγιστος κοινός διαιρέτης. Έστω ότι αναζητάμε το ΜΚΔ των α και β, όπου β>0 και γ= α *mod* β. Ο ευκλείδειος αλγόριθμος στηρίζεται στην παρατήρηση ότι αν γ=0, τότε ΜΚΔ(α,β)=β, ενώ αν γ≠0, τότε ΜΚΔ(α,β)=ΜΚΔ(β,γ). Προφανώς, αν γ= α *mod* β =0, τότε ο β διαιρεί τον α και τον εαυτό του και δεν υπάρχει κανένας μεγαλύτερος ακέραιος με την ιδιότητα αυτή. Αντίθετα, αν γ=α *mod* β≠0, μπορούμε να εκφράσουμε τον α ως α=βδ+γ ⇔ γ=α-βδ, για κάποια τιμή του δ. Επειδή ο ΜΚΔ(α,β) διαιρεί τόσο τον α όσο και τον β, θα διαιρεί και τον γ. Όμοια αποδεικνύεται ότι και ο ΜΚΔ(β,γ) διαιρεί τον α. Επειδή β>γ>0, το προηγούμενο βήμα μπορεί να επαναληφθεί έως ότου ο αλγόριθμος τερματιστεί.

♦

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Αλγόριθμος υπολογισμού της ερμητιανής μορφής ακέραιου πίνακα

Η κανονική ερμητιανή μορφή ενός πίνακα H, ο οποίος αποτελείται μόνο από ακέραια στοιχεία, βρίσκεται με μία σειρά unimodular μετασχηματισμών, δηλαδή ανταλλαγή δύο στηλών του πίνακα, πολλαπλασιασμό των στοιχείων μιας στήλης με –1, πρόσθεση (ή αφαίρεση) ακέραιου πολλαπλάσιου μιας στήλης σε (ή από) κάποια άλλη. Οι μετασχηματισμοί αυτοί αρχικά μηδενίζουν τα στοιχεία του πίνακα, που βρίσκονται πάνω από την κύρια διαγώνιό του, και στη συνέχεια μειώνουν την τιμή των στοιχείων κάτω από τη διαγώνιο, ενώ ταυτόχρονα διατηρούν το lattice των στηλών του πίνακα H σταθερό.

Προκειμένου να μηδενίσουμε την τιμή του στοιχείου h_{ij}, το οποίο βρίσκεται πάνω από την κύρια διαγώνιο, χρησιμοποιούμε τον ευκλείδειο αλγόριθμο εύρεσης του μέγιστου κοινού διαιρέτη g=ΜΚΔ(h_{ij},h_{ii}). Κατά την διαδικασία εφαρμογής του αλγορίθμου αυτού, σε κάθε βήμα εφαρμόζουμε την πράξη $\mu_{\iota+2}=\mu_\iota \textit{ mod } \mu_{\iota+1}= \mu_\iota-\delta\mu_{\iota+1}$ όχι μόνο στους αριθμούς h_{ij} και h_{ii}, αλλά σε ολόκληρες τις στήλες j και i που αυτοί αντιπροσωπεύουν. Δηλαδή αφαιρούμε την στήλη i του πίνακα H, πολλαπλασιασμένη επί δ, από την στήλη j αυτού. Στο τέλος του ευκλείδειου αλγορίθμου, φροντίζουμε ώστε η στήλη της παραχθείσας ακολουθίας που περιέχει το στοιχείο $\mu_{\nu-1}$=g=ΜΚΔ(h_{ij},h_{ii}) να βρίσκεται στη στήλη i του πίνακα, ενώ η στήλη που περιέχει το στοιχείο μ_ν=0, να βρίσκεται στη στήλη j του πίνακα.

Στη συνέχεια πρέπει να προσαρμόσουμε την τιμή αυτών που βρίσκονται κάτω από την διαγώνιο έτσι ώστε να ισχύει $0\le h_{ij}<h_{ii}$ για κάθε $i>j$. Για το σκοπό αυτό βρίσκουμε δύο αριθμούς π, υ τέτοιους ώστε π≥0, $0\le\upsilon<h_{ii}$ και $h_{ij}=\pi h_{ii}+\upsilon$. Έπειτα αφαιρούμε τη στήλη i, πολλαπλασιασμένη επί π από την στήλη j. Μετά την ολοκλήρωση της πράξης αυτής μεταξύ των στηλών i και j, το στοιχείο h_{ij} θα έχει πάρει την τιμή υ.

Όλη η διαδικασία που περιγράψαμε μπορεί να παρασταθεί με ψευδοκώδικα για έναν $m\times n$ πίνακα B ως εξής:

```
HNF(B):
row=1
col=1
repeat
     Multiply columns to make all of the elements
                              B[row,col],…,B[row,n] positive.
     for k=col+1 to n do
          Column-GCD(B,row,k,col)
     if B[row,col]=0 row++
     else
          for k=1 to col-1 do
               c=⌊B[row,k]/B[row,col]⌋
               B[.,k]=B[.,k]-cB[.,col]
               (which means: subtract c times column col from column k)
          row++
```

```
          col++
      endif
until row>m or col>n

Column-GCD(B,row,k,col):
while B[row,k]≠0 do
      swap(B[.,k],B[.,col]) (which means: swap columns k, col)
      c=⌊B[row,k]/B[row,col]⌋
      B[.,k]=B[.,k]-cB[.,col]
loop
```

Στην περίπτωση που η είσοδος του ψευδοκώδικα αυτού είναι ένας μη τετραγωνικός, ή μη αντιστρέψιμος πίνακας, τότε το αποτέλεσμα που προκύπτει είναι σύμφωνο με τον γενικευμένο ορισμό 2.3.

♦

Παράδειγμα 2.8

Έστω ότι θέλουμε να βρούμε την κανονική ερμητιανή μορφή του πίνακα

$$\begin{pmatrix} 2 & 6 & 1 \\ 4 & 7 & 7 \\ 0 & 0 & 1 \end{pmatrix}$$

Η ακολουθία των βημάτων που θα ακολουθηθεί για τον υπολογισμό της είναι η εξής:

Κατ' αρχήν παρατηρούμε ότι όλα τα στοιχεία της πρώτης γραμμής είναι θετικά. Επομένως δεν χρειάζεται να κάνουμε τίποτα για αυτό. Στη συνέχεια, προκειμένου να μηδενίσουμε το στοιχείο h_{12}, βρίσκουμε το ΜΚΔ(h_{12},h_{11})=ΜΚΔ(6,2)=2, οπότε συμπεραίνουμε ότι αρκεί να αφαιρέσουμε την πρώτη στήλη από την δεύτερη 3 φορές. Έτσι προκύπτει ο πίνακας $\begin{pmatrix} 2 & 0 & 1 \\ 4 & -5 & 7 \\ 0 & 0 & 1 \end{pmatrix}$. Όμοια, προκειμένου να μηδενίσουμε το στοιχείο h_{13}, βρίσκουμε το ΜΚΔ(h_{13},h_{11})=ΜΚΔ(1,2)=1, οπότε συμπεραίνουμε ότι πρέπει να αφαιρέσουμε την τρίτη από την πρώτη 2 φορές και έπειτα να τις ανταλλάξουμε μεταξύ τους. Έτσι προκύπτει ο πίνακας $\begin{pmatrix} 1 & 0 & 0 \\ 7 & -5 & -10 \\ 1 & 0 & -2 \end{pmatrix}$.

Έπειτα, για να γίνουν τα στοιχεία h_{22} και h_{23} θετικά, πολλαπλασιάζουμε τη δεύτερη και την τρίτη στήλη με –1. Έτσι παίρνουμε τον πίνακα $\begin{pmatrix} 1 & 0 & 0 \\ 7 & 5 & 10 \\ 1 & 0 & 2 \end{pmatrix}$. Για να μηδενίσουμε το στοιχείο h_{23}, βρίσκουμε το ΜΚΔ(h_{23},h_{22})=ΜΚΔ(10,5)=5, οπότε συμπεραίνουμε ότι αρκεί να αφαιρέσουμε τη δεύτερη στήλη από την τρίτη 2 φορές. Έτσι προκύπτει ο πίνακας $\begin{pmatrix} 1 & 0 & 0 \\ 7 & 5 & 0 \\ 1 & 0 & 2 \end{pmatrix}$. Για να προσαρμόσουμε το στοιχείο h_{21} έτσι ώστε να ισχύει $0 \leq h_{21} < h_{22}$, αφαιρούμε μία φορά την δεύτερη στήλη από την πρώτη, οπότε προκύπτει ο πίνακας $\begin{pmatrix} 1 & 0 & 0 \\ 2 & 5 & 0 \\ 1 & 0 & 2 \end{pmatrix}$.

Στη συνέχεια παρατηρούμε ότι το στοιχείο h_{33} είναι ήδη θετικό και ότι ισχύει $0 \leq h_{31}, h_{32} < h_{33}$. Επομένως δεν χρειάζεται να κάνουμε καμία άλλη ενέργεια στον προηγούμενο πίνακα.

Άρα η ζητούμενη ερμητιανή μορφή του αρχικού πίνακα είναι:

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 5 & 0 \\ 1 & 0 & 2 \end{pmatrix}$$

Κατά την προηγούμενη διαδικασία όλοι οι μετασχηματισμοί πινάκων που εκτελέσαμε ήταν unimodular. Άρα ο αρχικός και ο τελικός πίνακας έχουν το ίδιο lattice.

♦

© Μαρια Αθανασακη

Κεφάλαιο 3 : Γέννηση κώδικα SPMD

Για την παραγωγή του τελικού *SPMD* κώδικα, που θα επιτρέπει στους επεξεργαστές να διατρέχουν παράλληλα τα tiles που ορίζονται από έναν μετασχηματισμό με *tiling matrix* τον *H*, πρέπει κατ' αρχήν να βρούμε των βέλτιστο πίνακα *H* για το συγκεκριμένο πρόβλημα. Κριτήριο για την επιλογή του *H*, όπως προτείνεται στη βιβλιογραφία, μπορεί να είναι η ελαχιστοποίηση της επικοινωνίας μεταξύ των επεξεργαστών [IRI88], [XUE97], [RAM92a], [BOU94], [RAM92b], ή η ελαχιστοποίηση του συνολικού χρόνου εκτέλεσης του προγράμματος [HOD98], [GOU01]. Στη συνέχεια πρέπει να εφαρμόσουμε το μετασχηματισμό υπερκόμβων που προκύπτει για να βρούμε το Tile Space I^S και το Tile Iteration Space *TIS(H)*.

Με το μετασχηματισμό του tiling αναδιατάσσεται η σειρά εκτέλεσης των επαναλήψεων του Iteration Space I^n σύμφωνα με την εξής λογική: για κάθε tile του Tile Space I^S διατρέχονται τα σημεία που ανήκουν σε αυτό. Δηλαδή ο τελικός κώδικας αποτελείται από *2n* φωλιασμένους βρόχους. Οι *n* πιο εξωτερικοί από αυτούς διατρέχουν το χώρο I^S, χρησιμοποιώντας τους δείκτες $t_1, t_2, \ldots, t_n$, ενώ οι *n* πιο εσωτερικοί βρόχοι διατρέχουν τα σημεία του tile $\vec{t} = (t_1, t_2, \ldots, t_n)$ χρησιμοποιώντας τους δείκτες $i_1', i_2', \ldots, i_n'$.

Δηλαδή ο κώδικας που θα προκύψει μετά από έναν μετασχηματισμό tiling θα έχει τη μορφή:

```
for (t1=L1S; t1<=U1S; t1++)
      for (t2=L2S; t2<=U2S; t2++)
      … … … … …
            for (tn=LnS; tn<=UnS; tn++)
                  for (i1'=l1'; i1'<=u1'; i1'++)
                        for (i2'=l2'; i2'<=u2'; i2'++)
                        … … … … …
                              for (in'=ln'; in'<=un'; in'++)
                              {
                                    AS1(t,i');
                                    … … …
                                    ASm(t,i');
                              }
```

Για να γράψουμε κώδικα της μορφής αυτής αρκεί να γνωρίζουμε τα όρια $L_1^S,\ldots,L_n^S$, $U_1^S,\ldots,U_n^S$, $l_1',\ldots,l_n'$, και $u_1',\ldots,u_n'$ σαν συνάρτηση των δεικτών των εξωτερικότερων βρόχων. Η εύρεση των ορίων αυτών θα είναι το αντικείμενο του παρόντος κεφαλαίου.

Στην παρακάτω ανάλυσή μας διακρίνουμε δύο ξεχωριστά προβλήματα:

(α) την εύρεση των ορίων $L_1^S,\ldots,L_n^S$, $U_1^S,\ldots,U_n^S$ τα οποία θα μας επιτρέψουν να διατρέξουμε όλα τα tiles και

(β) την εύρεση των ορίων $l_1',\ldots,l_n'$, και $u_1',\ldots,u_n'$ τα οποία θα μας επιτρέψουν να σαρώσουμε το εσωτερικό κάθε tile.

3.1 Εύρεση των ορίων του Tile Space

3.1.1 Ακριβής μέθοδος

Μία μέθοδος προτεινόμενη από τους C.Ancourt, F.Irigoin στο άρθρο "*Scanning Polyhedra with DO Loops*" [ANC91], η οποία με πολύ μεγάλο υπολογιστικό κόστος καταφέρνει να περιγράψει ακριβώς τα όρια του χώρου I^S, είναι η εξής:

Ένα σημείο $\boldsymbol{i} \in Z^n$ ανήκει στο Iteration Space I^n ενός προβλήματος αν και μόνο αν ικανοποιεί το σύστημα των ανισώσεων $B \cdot i \le \vec{b}$. Επίσης, όπως αναφέρθηκε σε προηγούμενη παράγραφο, το σημείο $\boldsymbol{i}$ ανήκει στο tile με origin $\boldsymbol{i_{toi}}$ αν και μόνο αν επαληθεύεται το σύστημα ανισοτήτων $\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot (i - i_{toi}) \le \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix}$. Αν $\boldsymbol{t}$ είναι οι

συντεταγμένες του tile στο Tile Space, τότε ισχύει $i_{toi} = P \cdot t = H^{-1} \cdot t$. Δηλαδή, η παραπάνω σχέση μπορεί ισοδύναμα να γραφεί: $\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot \left(i - H^{-1} \cdot t\right) \leq \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$ $\Leftrightarrow$ $\begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \leq \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$.

Ένα tile ***t*** ανήκει στο Tile Space I^S αν και μόνο αν υπάρχει ένα τουλάχιστον σημείο ***i*** που να ανήκει στο tile ***t*** και ταυτόχρονα να ανήκει στο Iteration Space I^n. Δηλαδή πρέπει να ικανοποιούνται ταυτόχρονα τα συστήματα ανισοτήτων $\begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \leq \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$ και $B\vec{i} \leq \vec{b}$. Τα δύο αυτά συστήματα μπορούν να γραφούν στη μορφή ενός συστήματος ως εξής:

$$\begin{pmatrix} 0 & B \\ -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \leq \begin{pmatrix} \vec{b} \\ (g-1)\vec{1} \\ \vec{0} \end{pmatrix} \qquad (3.1.1)$$

Προκειμένου να παράγουμε τον τελικό κώδικα θα πρέπει να εκφράσουμε τα όρια κάθε μιας από τις μεταβλητές t_i, $i=1,\ldots,n$ συναρτήσει των $t_1,t_2,\ldots,t_{i-1}$. Αυτό επιτυγχάνεται με εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin στο σύστημα ανισοτήτων *(3.1.1).*

Παράδειγμα 3.1

Συνεχίζοντας το παράδειγμα 2.1, αν θελήσουμε να βρούμε τα όρια του Tile Space με την παρούσα μέθοδο, θα πρέπει να εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin στο σύστημα ανισοτήτων της μορφής *(3.1.1)*:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ -48 & 0 & 9 & -3 \\ 0 & -48 & -2 & 6 \\ 48 & 0 & -9 & 3 \\ 0 & 48 & 2 & -6 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 15 \\ 0 \\ 31 \\ 0 \\ 47 \\ 47 \\ 0 \\ 0 \end{pmatrix}$$

Έτσι παίρνουμε το εξής σύστημα:

$$\begin{pmatrix} 16 & 0 & 0 & 0 \\ -12 & 0 & 0 & 0 \\ 2 & 9 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -96 & -432 & 0 & 0 \\ -32 & -16 & 0 & 0 \\ -48 & 0 & 9 & 0 \\ 0 & 24 & 1 & 0 \\ -96 & -48 & 16 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -48 & -2 & 0 \\ 16 & 0 & -3 & 0 \\ 6 & 3 & -1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -48 & -2 & 6 \\ 16 & 0 & -3 & 1 \\ 0 & 0 & 0 & -1 \\ -48 & 0 & 9 & -3 \\ 0 & 24 & 1 & -3 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 45 \\ 35 \\ 31 \\ 5 \\ 517 \\ 47 \\ 140 \\ 93 \\ 141 \\ 15 \\ 47 \\ 0 \\ 0 \\ 0 \\ 31 \\ 47 \\ 0 \\ 0 \\ 47 \\ 0 \end{pmatrix}$$

Από το σύστημα αυτό, για το πρόβλημα εύρεσης των ορίων του Tile Space είναι χρήσιμες μόνο οι 6 πρώτες ανισότητες, οι οποίες αναφέρονται στους δείκτες t_1 και t_2. Δηλαδή τα όρια των 2 εξωτερικότερων βρόχων του τελικού κώδικα θα δίνονται από το σύστημα των ανισοτήτων:

$$\left(\begin{array}{cc} 16 & 0 \\ \hline -12 & 0 \\ \hline 2 & 9 \\ 2 & 1 \\ \hline -96 & -432 \\ -32 & -16 \end{array}\right) \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \leq \left(\begin{array}{c} 45 \\ \hline 35 \\ \hline 31 \\ 5 \\ \hline 517 \\ 47 \end{array}\right)$$

Ο κώδικας που θα παραχθεί με τη βοήθεια του συστήματος αυτού, λαμβάνοντας υπ' όψη ότι $\lceil -35/12 \rceil = -2$ και $\lfloor 45/16 \rfloor = 2$, είναι ο εξής:

```
for (t1=-2; t1<=2; t1++)
     for (t2=max(⌈(-517-96*t1)/432⌉,⌈(-47-32*t1)/16⌉);
                              t2<=min(⌊(31-2*t1)/9⌋,5-2*t1); t2++)
     … … … … …
```

Ο κώδικας αυτός διατρέχει τα tiles (-2,2), (-2,3), (-1,0), …, (-1,3), (0,-1), …, (0,3), (1,-1), …, (1,3), (2,-1), …, (2,1), τα οποία, σύμφωνα με το σχήμα 2.2, είναι πράγματι αυτά που πρέπει να διατρεχθούν.

♦

3.1.2 Γρήγορη μέθοδος

Η επόμενη μέθοδος για την εύρεση των ορίων του Tile Space στηρίζεται στην παρατήρηση ότι τα tile origins των tiles που ανήκουν στο Tile Space πρέπει να βρίσκονται μέσα στο Iteration Space ή πολύ κοντά σε αυτό. Επομένως, προκειμένου να βρούμε τα όρια μέσα στα οποία κινούνται, αρκεί να μετατοπίσουμε κατάλληλα τα όρια του Iteration Space. Η μετατόπιση που χρειάζεται δίδεται στο επόμενο λήμμα. Στην διατύπωση και στην απόδειξη του λήμματος αυτού χρησιμοποιούμε το συμβολισμό:

$$\alpha^{+}=max(\alpha,0) \text{ και } \alpha^{-}=max(-\alpha,0).$$

Λήμμα 3.1

Αν εφαρμόσουμε τον μετασχηματισμό tiling που ορίζεται από τον inverse tiling matrix *P,* σε έναν Iteration Space I^n τα όρια του οποίου δίδονται από το σύστημα ανισοτήτων $B\vec{i} \le \vec{b}$, τότε όλα τα σημεία του Tile Origin Space *TOS* που προκύπτει ικανοποιούν το σύστημα ανισοτήτων

$$B \cdot i_{toi} \le \vec{b}' \qquad (3.1.2)$$

όπου $\vec{b}'$ είναι ένα *n*-διάστατο διάνυσμα του οποίου κάθε στοιχείο υπολογίζεται από το αντίστοιχο στοιχείο του διανύσματος $\vec{b}$ σύμφωνα με τη σχέση

$$b_i' = b_i + \frac{g-1}{g}\sum_{r=1}^{n}\left(\sum_{j=1}^{n}\beta_{ij}p_{jr}\right)^{-} \qquad (3.1.3)$$

όπου β_{ij} είναι το στοιχείο της *i*-οστής γραμμής και της *j*-οστής στήλης του πίνακα *B*, p_{jr} είναι το στοιχείο της *j*-οστής γραμμής και της *r*-οστής στήλης του πίνακα *P* και *g* είναι ο ελάχιστος ακέραιος αριθμός με τον οποίο πρέπει να πολλαπλασιαστεί ο πίνακας $H=P^{-1}$ ώστε όλα τα στοιχεία του να γίνουν ακέραια.

Απόδειξη

Έστω ότι το σημείο $\boldsymbol{i}$ ανήκει στο tile με origin $\boldsymbol{i_{toi}}$. Τότε, όπως αναφέρθηκε στον ορισμό 1.5, το σημείο $\boldsymbol{i}$ μπορεί να εκφραστεί σαν άθροισμα του $\boldsymbol{i_{toi}}$ και ενός γραμμικού συνδυασμού των διανυσμάτων-στηλών του πίνακα P.

$$i = i_{toi} + \sum_{j=1}^{n} x_j p_j$$

Επίσης, όπως αναφέρθηκε στην παράγραφο 2.1 ισχύει: $\vec{0} \le gH \cdot (i - i_{toi}) \le (g-1)\vec{1}$. Η l-οστή γραμμή αυτού του συστήματος ανισοτήτων γράφεται: $0 \le h_l \cdot (i - i_{toi}) \le \frac{g-1}{g}$, όπου h_l είναι η l-οστή γραμμή-διάνυσμα του πίνακα $H=P^{-1}$. Επομένως ισχύει: $0 \le h_l \cdot \sum_{j=1}^{n} x_j p_j \le \frac{g-1}{g} \Rightarrow$

$$0 \le \sum_{j=1}^{n} x_j (h_l \cdot p_j) \le \frac{g-1}{g}$$

Επειδή $H=P^{-1}$ ισχύει $h_l p_l=1$ και $h_l p_j=0$ για κάθε $j \neq l$. Επομένως η τελευταία σχέση γράφεται:

$$0 \le x_l \le \frac{g-1}{g} \text{ για κάθε } l=1,\dots,n.$$

Για κάθε $i \in I^n$ ισχύει το σύστημα ανισοτήτων $B \cdot i \le \vec{b}$, όπου B είναι ένας πίνακας διαστάσεων $d \times n$. Η k-οστή γραμμή του (για κάποιο $k{:}1 \le k \le d$) γράφεται: $\sum_{j=1}^{n} \beta_{kj} i_j \le b_k$. Συναρτήσει του tile origin $\boldsymbol{i_{toi}}$ έχουμε:

$$\sum_{j=1}^{n} \beta_{kj} \left(i_{toi,j} + \sum_{l=1}^{n} x_l p_{jl} \right) \le b_k \Rightarrow$$

$$\sum_{j=1}^{n} \beta_{kj} i_{toi,j} \le b_k - \sum_{j=1}^{n} \beta_{kj} \sum_{l=1}^{n} x_l p_{jl} \Rightarrow$$

$$\sum_{j=1}^{n} \beta_{kj} i_{toi,j} \le b_k - \sum_{l=1}^{n} x_l \sum_{j=1}^{n} \beta_{kj} p_{jl} .$$

Επιπλέον, όπως αποδείξαμε προηγουμένως, ισχύει $0 \le x_l \le \frac{g-1}{g}$ για κάθε $l=1,\dots,n$. Αν όλα τα μέλη της σχέσης αυτής πολλαπλασιαστούν με $\sum_{j=1}^{n} \beta_{kj} p_{jl}$, παίρνουμε:

α) Αν $\sum_{j=1}^{n} \beta_{kj} p_{jl} > 0$: $0 \le x_l \sum_{j=1}^{n} \beta_{kj} p_{jl} \le \frac{g-1}{g} \sum_{j=1}^{n} \beta_{kj} p_{jl}$

β) Αν $\sum_{j=1}^{n} \beta_{kj} p_{jl} < 0$: $\frac{g-1}{g} \sum_{j=1}^{n} \beta_{kj} p_{jl} \le x_l \sum_{j=1}^{n} \beta_{kj} p_{jl} \le 0$

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Σύμφωνα με τον ορισμό των συμβόλων α^+ και α^- σε κάθε περίπτωση ισχύει:

$$-\frac{g-1}{g}\left(\sum_{j=1}^{n}\beta_{kj}p_{jl}\right)^- \leq x_l\sum_{j=1}^{n}\beta_{kj}p_{jl} \leq \frac{g-1}{g}\left(\sum_{j=1}^{n}\beta_{kj}p_{jl}\right)^+ \Rightarrow$$

$$-x_l\sum_{j=1}^{n}\beta_{kj}p_{jl} \leq \frac{g-1}{g}\left(\sum_{j=1}^{n}\beta_{kj}p_{jl}\right)^-$$

Προσθέτοντας κατά μέλη για όλα τα $l=1,\ldots,n$ παίρνουμε:

$$-\sum_{l=1}^{n}x_l\sum_{j=1}^{n}\beta_{kj}p_{jl} \leq \frac{g-1}{g}\sum_{l=1}^{n}\left(\sum_{j=1}^{n}\beta_{kj}p_{jl}\right)^-$$

Από την τελευταία σχέση και από την ανισότητα $\sum_{j=1}^{n}\beta_{kj}i_{toi,j} \leq b_k - \sum_{l=1}^{n}x_l\sum_{j=1}^{n}\beta_{kj}p_{jl}$, που αποδείξαμε προηγουμένως, συμπεραίνουμε ότι: $\sum_{j=1}^{n}\beta_{kj}i_{toi,j} \leq b_k + \frac{g-1}{g}\sum_{l=1}^{n}\left(\sum_{j=1}^{n}\beta_{kj}p_{jl}\right)^-$.

Η σχέση αυτή ισχύει για κάθε $k=1,\ldots,d$ Επομένως για κάθε tile με origin $\boldsymbol{i_{toi}}$ ισχύει $B\cdot i_{toi} \leq \vec{b}'$, όπου το διάνυσμα $\vec{b}'$ κατασκευάζεται από το $\vec{b}$ σύμφωνα με τον τύπο $b_k' = b_k + \frac{g-1}{g}\sum_{i=1}^{n}\left(\sum_{j=1}^{n}\beta_{kj}p_{ji}\right)^-$

♦

Αν θέλουμε να δουλεύουμε στο χώρο I^S, τότε λαμβάνοντας υπ’ όψη ότι $\boldsymbol{i_{toi}}=P\boldsymbol{t}$, συμπεραίνουμε ότι τα όρια του Tile Space I^S δίδονται ισοδύναμα από το σύστημα ανισοτήτων

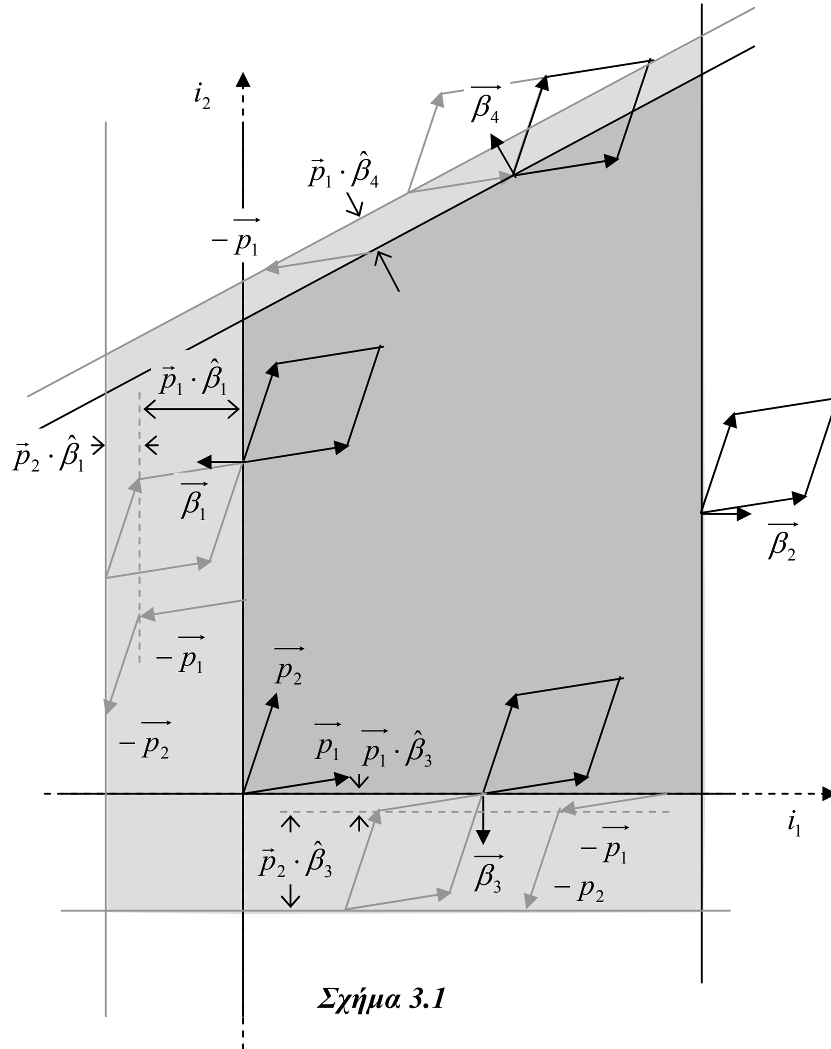
$$BP\cdot t \leq \vec{b}' \qquad (3.1.4)$$

Προκειμένου στη συνέχεια να παράγουμε τον τελικό κώδικα θα πρέπει να εκφράσουμε τα όρια κάθε μιας από τις μεταβλητές t_i, $i=1,\ldots,n$ συναρτήσει των $t_1,t_2,\ldots,t_{i-1}$. Αυτό επιτυγχάνεται με εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin στο σύστημα ανισοτήτων *(3.1.4)*.

Γεωμετρική ερμηνεία

Ο όρος που προστίθεται σε κάθε στοιχείο του διανύσματος $\vec{b}$ εκφράζει μια παράλληλη μετατόπιση του αντίστοιχου ορίου του αρχικού χώρου I^n. Η μετατόπιση αυτή μπορεί να υπολογιστεί όπως περιγράφεται στο σχήμα 3.1. Η περιοχή που είναι σκιασμένη με σκούρο γκρι χρώμα αντιπροσωπεύει τον Iteration Space I^n ενός 2-διάστατου προβλήματος. Αν ο μετασχηματισμός tiling είναι δεδομένος και δίδεται από τον πίνακα P, τότε προκειμένου να βρούμε τα όρια μέσα στα οποία μπορούν να κινούνται τα tile origins

του προβλήματος αυτού πρέπει να επεκτείνουμε το χώρο I^n έτσι ώστε να συμπεριλάβουμε και την περιοχή που έχει χρωματιστεί με ανοιχτό γκρι στο σχήμα.



Σχήμα 3.1

Η επιθυμητή μετατόπιση κάθε ορίου του αρχικού χώρου υπολογίζεται με βάση την εξής λογική: Κάθε γραμμή-διάνυσμα β_i του πίνακα B είναι ένα διάνυσμα του n-διάστατου χώρου, κάθετο στο αντίστοιχο όριο του I^n και με φορά προς τα έξω. Αν ένα από τα διανύσματα-ακμές του tile p_j σχηματίζει γωνία μεγαλύτερη από 90° με το β_i (όπως τα διανύσματα p_1 και β_1, p_2 και β_1, p_1 και β_3, p_2 και β_3, p_1 και β_4, του σχήματος 3.1) τότε πρέπει το αντίστοιχο όριο του I^n να μετατοπιστεί κατά το διάνυσμα $-p_j$. Η γωνία μεταξύ δύο διανυσμάτων είναι μεγαλύτερη των 90° αν και μόνο αν το εσωτερικό γινόμενό τους είναι αρνητικό. Επίσης, η μετατόπιση του ορίου $\beta_i x=b_i$ κατά $-p_j$ δίδει ένα (n-1)-διάστατο υπερεπίπεδο που περιγράφεται από την εξίσωση $\beta_i(x-(-p_j))=b_i \Leftrightarrow \beta_i x=b_i-\beta_i p_j$. Επομένως, αν το εσωτερικό γινόμενο μιας γραμμής του πίνακα B, β_i και μιας στήλης του P, p_j είναι αρνητικό, τότε πρέπει να αφαιρεθεί από τη σταθερά b_i. Στον τύπο *(3.1.3)* προσθέτουμε στα

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

διανύσματα β_i για κάθε διάνυσμα p_j τη σταθερά $\left(\sum_{k=1}^{n}\beta_k p_{kj}\right)^{-} = \left(\vec{\beta}_i \cdot \vec{p}_j\right)^{-}$ η οποία ισούται με *0* αν $\vec{\beta}_i \cdot \vec{p}_j > 0$ και με $-\vec{\beta}_i \cdot \vec{p}_j$ αν $\vec{\beta}_i \cdot \vec{p}_j < 0$. Δηλαδή σε κάθε περίπτωση η σταθερά αυτή εκφράζει την επιθυμητή μετατόπιση.

Ο παράγοντας $\frac{g-1}{g}$ με τον οποίο πολλαπλασιάζουμε τη σταθερά μετατόπισης εκφράζει το γεγονός ότι κάθε tile είναι ένα ημιανοιχτό υπερπαραλληλεπίπεδο και επομένως δεν είναι απαραίτητο στον Tile Space να συμπεριλάβουμε τα tiles που απλώς εφάπτονται στον χώρο I^n.

♦

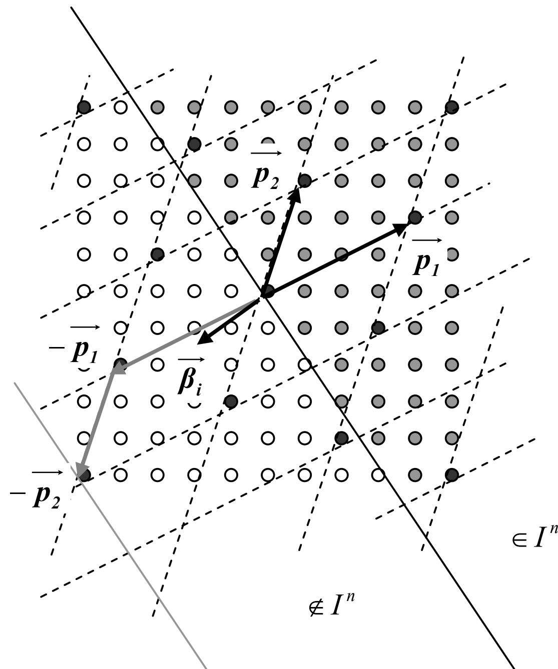
Στο σημείο αυτό οφείλουμε να σημειώσουμε ότι παραπάνω αποδείξαμε μόνον ότι «Αν ένα tile ανήκει στο χώρο I^S, τότε ικανοποιείται το σύστημα ανισοτήτων (*3.1.4*)», και όχι το ανάποδο. Αυτό σημαίνει ότι οι συντεταγμένες ενός tile μπορεί να ικανοποιούν το (*3.1.4*), αλλά στην πραγματικότητα να μην ανήκει στο I^S. Δηλαδή, με τον τρόπο αυτό ο παραγόμενος κώδικας θα πρέπει να σαρώσει και κάποια περιττά tiles. Όπως θα φανεί, όμως, και από τα επόμενα παραδείγματα, τα tiles αυτά περιορίζονται μόνο στις ακμές του αρχικού χώρου I^n και ο αριθμός τους είναι πάντα αμελητέος σε σχέση με τον αριθμό όλων των tiles του I^S. Επίσης, με τις μεθόδους που θα αναλύσουμε στο επόμενο κεφάλαιο το τελικό πρόγραμμα θα μπορεί αμέσως να ανιχνεύσει ότι ένα tile δεν περιλαμβάνει κανένα σημείο του I^n, και συνεπώς δεν θα εκτελέσει πολλούς περιττούς υπολογισμούς.

Παραδείγματα 3.2

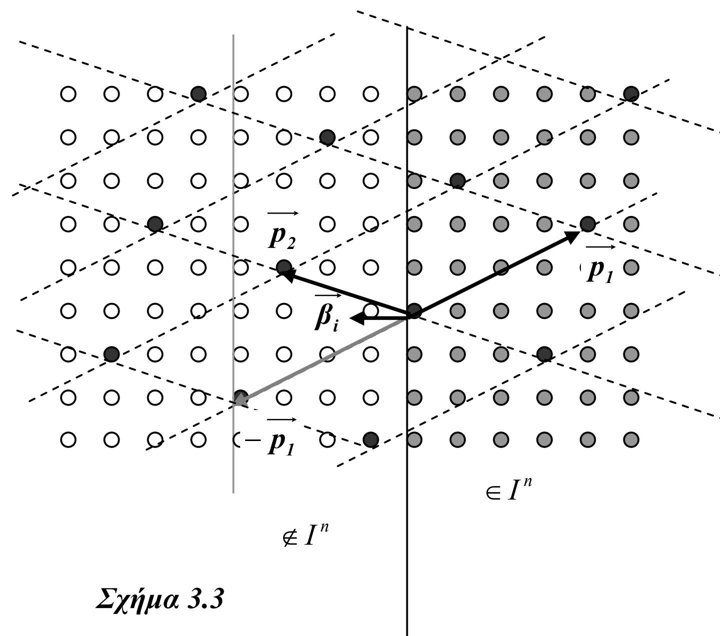
Στο σχήμα 3.2 έχουμε σχεδιάσει ένα από τα όρια του χώρου I^n και τα διανύσματα-στήλες του πίνακα *P*, τα οποία αποτελούν τις πλευρές των tiles. Οι μαύρες κουκίδες είναι τα origins κάποιων tiles τα οποία μπορεί να ανήκουν ή να μην ανήκουν στο Tile Space, οι γκρι κουκίδες είναι τα σημεία του Z^n που ανήκουν στο χώρο I^n, ενώ οι άσπρες είναι τα σημεία του Z^n που δεν ανήκουν στο χώρο I^n. Τα tile origins που βρίσκονται δεξιά της μαύρης γραμμής ανήκουν στο χώρο I^n και συνεπώς πρέπει να περιληφθούν στο χώρο *TOS*. Εκτός από αυτές όμως πρέπει να περιληφθούν στο *TOS* και τα tile origins που βρίσκονται μεταξύ της μαύρης και της γκρι γραμμής, επειδή είναι πιθανόν κάποια άλλα σημεία των αντίστοιχων tiles να ανήκουν στο I^n. Επομένως προκειμένου να παράγουμε το χώρο *TOS* από τον I^n πρέπει να μετατοπίσουμε το όριο του I^n από την μαύρη στη γκρι γραμμή. Αυτό πραγματοποιείται μετακινώντας την ευθεία $\beta_i x=b_i$ κατά το διάνυσμα $-p_1-p_2$.

Όμοιο είναι και το σχήμα 3.3, με τη μόνη διαφορά ότι σε αυτό το όριο που έχει σχεδιαστεί πρέπει να μετακινηθεί μόνο κατά το διάνυσμα $-p_1$ προκειμένου να συμπεριλάβει στο χώρο *TOS* το ποιο απομακρυσμένο από το I^n tile origin. Αυτό συμβαίνει

επειδή όπως φαίνεται στο σχήμα το διάνυσμα p_2 σχηματίζει με το β_i γωνία μικρότερη από 90°. Στα σχήματα αυτά για διευκόλυνση της διαισθητικής παρουσίασης δεν έχουμε λάβει υπ’ όψη ότι στο Tile Space δεν είναι απαραίτητο να περιληφθούν τα tiles που απλώς εφάπτονται στο I^n.



Σχήμα 3.2



Σχήμα 3.3

♦

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Παράδειγμα 3.3

Για να βρούμε τα όρια του Tile Space του παραδείγματος 2.1, θα πρέπει να επεκτείνουμε τα όρια του Iteration Space, έτσι ώστε να προκύψει ένα σύστημα ανισοτήτων της μορφής *(3.1.4)*:

$$\begin{pmatrix} 6 & 3 \\ -6 & -3 \\ 2 & 9 \\ -2 & -9 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \leq \begin{pmatrix} 15 \\ \frac{47}{48}9 \\ 31 \\ \frac{47}{48}11 \end{pmatrix} \cong \begin{pmatrix} 15 \\ 8{,}8125 \\ 31 \\ 10{,}7708 \end{pmatrix}$$

Τα όρια που προκύπτουν έχουν σχεδιαστεί στο σχήμα 3.4 με γκρι γραμμές. Επίσης, με μαύρες ή γκρι κουκίδες έχουμε σχεδιάσει τα origins των tiles που με τη μέθοδο αυτή θα συμπεριληφθούν στο Tile Space, ενώ με άσπρες αυτών που δεν θα συμπεριληφθούν. Ανάμεσα στα tiles που συμπεριλαμβάνονται στο χώρο I^S, υπάρχουν και κάποια που δεν έχουν κανένα σημείο μέσα στο χώρο I^n. Είναι τα 2 tiles των οποίων τα σημεία εκκίνησης έχουν γκρι χρώμα.

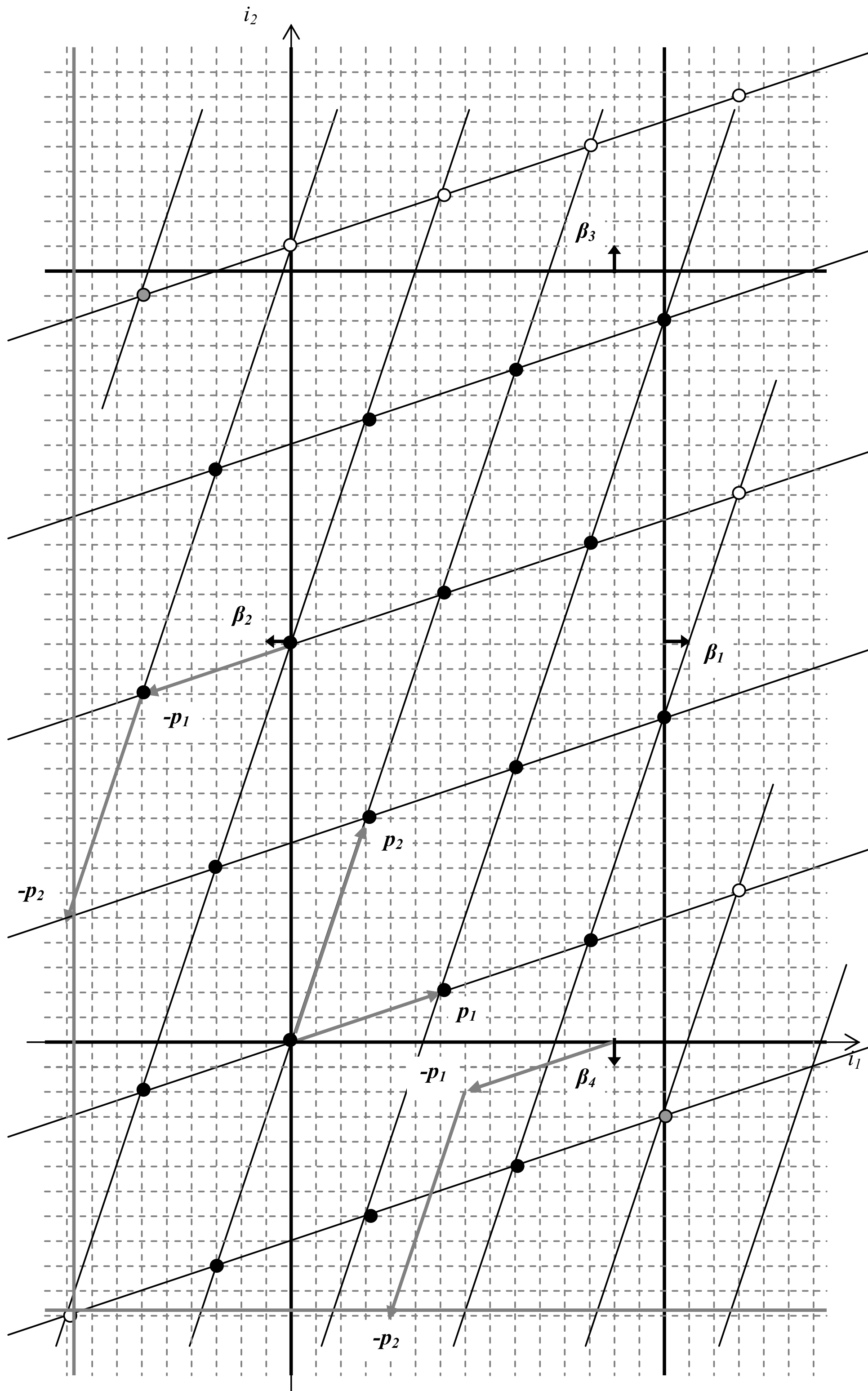
Αν εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin στο παραπάνω σύστημα παίρνουμε τις εξής ανισότητες:

$$\begin{pmatrix} 768 & 0 \\ \hdashline -256 & 0 \\ \hline 2 & 1 \\ 2 & 9 \\ \hdashline -32 & -16 \\ -96 & -432 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \leq \begin{pmatrix} 2677 \\ \hdashline 919 \\ \hline 5 \\ 31 \\ \hdashline 47 \\ 517 \end{pmatrix}$$

Ο κώδικας που θα παραχθεί με τη βοήθεια του συστήματος αυτού, λαμβάνοντας υπ' όψη ότι $\lceil -919/256 \rceil = -3$ και $\lfloor 2677/768 \rfloor = 3$, είναι ο παρακάτω και διατρέχει πράγματι τα tiles που έχουν προβλεφθεί στο σχήμα 3.4.

```
for (t1=-3; t1<=3; t1++)
    for (t2=max(⌈(-47-32*t1)/16⌉,⌈(-517-96*t1)/432⌉);
                         t2<=min(5-2t1,⌊(31-2t1)/9⌋); t2++)
    … … … … …
```

♦



Σχήμα 3.4

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

3.1.3 Σύγκριση των δύο μεθόδων

Αν ο αρχικός κώδικας περιέχει φωλιασμένους βρόχους βάθους n, των οποίων τα όρια εκφράζονται με απλές γραμμικές συναρτήσεις, τότε τα συστήματα *(3.1.1)* και *(3.1.4)* θα αποτελούνται από $4n$ ανισότητες $2n$ μεταβλητών και $2n$ ανισότητες n μεταβλητών αντίστοιχα.

Επομένως, σύμφωνα με τον τύπο *(2.2.6),* η ακριβής μέθοδος υπολογίζει έναν πίνακα με $(2n)^{2^{2^n}}$ γραμμές για τον προσδιορισμό των ορίων του Tile Space, ενώ η γρήγορη μέθοδος υπολογίζει έναν πίνακα με n^{2^n} γραμμές. Επιπλέον, προκειμένου το τελικό σύστημα να μην περιέχει περιττές ανισότητες, οι οποίες θα επιβαρύνουν το τελικό πρόγραμμα με πολλές πράξεις, θα πρέπει να εφαρμόσουμε τις μεθόδους ανίχνευσης των περιττών ανισοτήτων. Στην περίπτωση που θα εφαρμόσουμε την ακριβή μέθοδο απλοποίησης, η οποία ενδείκνυται στις περισσότερες περιπτώσεις προκειμένου να βρεθούν όλες οι περιττές ανισότητες, για κάθε γραμμή των πινάκων που παράχθηκαν κατά την εφαρμογή της μεθόδου Fourier-Motzkin, θα πρέπει να εφαρμόσουμε και πάλι άλλη μια φορά τη μέθοδο Fourier-Motzkin. Το γεγονός αυτό επιβαρύνει πολύ περισσότερο την ήδη χρονοβόρα «ακριβή μέθοδο».

Επίσης, η γρήγορη μέθοδος καταλήγει σε λιγότερες ανισότητες, το οποίο σημαίνει ότι το τελικό πρόγραμμα θα πρέπει να υπολογίζει λιγότερες εκφράσεις κατά την εκτέλεσή του. Από την άλλη πλευρά, όμως, αυτό έχει σαν αποτέλεσμα να διατρέχονται και κάποια περιττά tiles. Ευτυχώς, αυτά περιορίζονται μόνο στις ακμές του αρχικού χώρου I^n. Επομένως αν υποθέσουμε ότι ο χώρος I^n είναι αρκετά μεγάλος, τότε σίγουρα ο αριθμός τους θα είναι αμελητέος σε σχέση με τον αριθμό όλων των tiles που πρέπει να διατρεχθούν. Πάντως κατά τη διάρκεια εκτέλεσης του τελικού προγράμματος, αυτό θα ανιχνεύσει αμέσως, εξετάζοντας τα όρια των μεταβλητών $i_1',\ldots, i_n'$ οι οποίες διατρέχουν το εσωτερικό του κάθε tile, ότι το συγκεκριμένο tile δεν περιέχει κανένα σημείο του I^n, οπότε δεν θα πραγματοποιηθούν περιττοί υπολογισμοί.

Συμπεραίνουμε, λοιπόν, ότι το μεγάλο πλεονέκτημα της «γρήγορης μεθόδου» είναι το γεγονός ότι χρειάζεται πολύ λιγότερο χρόνο και υπολογιστική ισχύ για τον υπολογισμό των ορίων του Tile Space. Προκειμένου να διαπιστώσουμε τη διαφορά, τρέξαμε ορισμένα παραδείγματα με διαφορετικούς πίνακες P, B, b και μετρήσαμε τις γραμμοπράξεις που απαιτεί καθεμιά από τις δυο μεθόδους για την ολοκλήρωσή της. Σε κάθε περίπτωση εφαρμόσαμε μια φορά τη μέθοδο απαλοιφής Fourier-Motzkin με απλοποίηση Ad-Hoc και με ακριβή απλοποίηση και μια φορά χωρίς ακριβή απλοποίηση. Γενικά, η δεύτερη εφαρμογή (χωρίς ακριβή απλοποίηση) δεν θα είναι πολύ χρήσιμη σε πραγματικά προβλήματα, επειδή ο παραγόμενος κώδικας θα πρέπει να υπολογίζει πολλές περιττές

εκφράσεις σε κάθε επανάληψη. Η παράθεσή της εδώ έγινε μόνο για να φανεί η διαφορά των δύο μεθόδων υπολογισμού των ορίων του Tile Space σε κάθε περίπτωση. Στον επόμενο πίνακα έχουμε καταγράψει τα πειραματικά αποτελέσματα. Δυστυχώς δεν ήταν δυνατόν να συγκρίνουμε πιο πολύπλοκα παραδείγματα εξ' αιτίας του μεγάλου υπολογιστικού και χρονικού κόστους της ακριβούς μεθόδου. Πάντως είναι σίγουρο ότι όσο πιο πολύπλοκο είναι το παράδειγμα, τόσο πιο έντονη θα είναι και η διαφορά των απαιτούμενων υπολογισμών για τις δύο μεθόδους.

Πίνακες παραδειγμάτων		Ακριβής μέθοδος (αριθμός γραμμοπράξεων)		Γρήγορη μέθοδος (αριθμός γραμμοπράξεων)	
Πίνακας $[B \mid \vec{b}]$	Πίνακας P	Με ακριβή απλοποίηση	Χωρίς ακριβή απλοποίηση	Με ακριβή απλοποίηση	Χωρίς ακριβή απλοποίηση
$\left[\begin{array}{cc|c} 1 & 0 & 39 \\ 0 & 1 & 29 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 6 & 4 \\ 2 & 8 \end{array}\right]$	56.080	192	58	6
$\left[\begin{array}{cc|c} 1 & 0 & 6 \\ 0 & 1 & 4 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 2 & 1 \\ 1 & 2 \end{array}\right]$	87.504	192	58	6
$\left[\begin{array}{cc|c} 1 & 0 & 49 \\ 0 & 1 & 59 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 10 & -5 \\ -8 & 10 \end{array}\right]$	78.656	192	58	6
$\left[\begin{array}{cc|c} 1 & 0 & 15 \\ -1 & 0 & 0 \\ 0 & 1 & 31 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 6 & 3 \\ 2 & 9 \end{array}\right]$	55.898	192	58	6
$\left[\begin{array}{cc|c} 1 & 0 & 14 \\ -1 & 0 & 0 \\ 0 & 1 & 9 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 2 & 1 \\ 1 & 4 \end{array}\right]$	78.577	192	58	6
$\left[\begin{array}{cc|c} 1 & 0 & 21 \\ -2 & 5 & 155 \\ -1 & 0 & 0 \\ -1 & -3 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 6 & -2 \\ 1 & 5 \end{array}\right]$	96.070	225	63	7
$\left[\begin{array}{cc|c} 1 & 0 & 23 \\ -2 & 5 & 155 \\ -1 & 0 & 0 \\ -1 & -3 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 6 & -2 \\ 1 & 5 \end{array}\right]$	109.307	225	63	7
$\left[\begin{array}{cc|c} 1 & 0 & 20 \\ -1 & 0 & 0 \\ 1 & 1 & 20 \\ 0 & -1 & 0 \end{array}\right]$	$\left[\begin{array}{c|c} 0 & 3 \\ 3 & 1 \end{array}\right]$	2.203	51	63	7

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

<table>
<tr><td>$\begin{bmatrix} 1 & 0 & \vdots & 20 \\ -1 & 0 & \vdots & 0 \\ 1 & 1 & \vdots & 20 \\ 0 & -1 & \vdots & 0 \end{bmatrix}$</td><td>$\begin{bmatrix} 1 & \vdots & 3 \\ 4 & \vdots & 1 \end{bmatrix}$</td><td>90.184</td><td>231</td><td>53</td><td>5</td></tr>
<tr><td>$\begin{bmatrix} 1 & 0 & \vdots & 18 \\ -1 & 0 & \vdots & 0 \\ 1 & 1 & \vdots & 18 \\ 0 & -1 & \vdots & 0 \end{bmatrix}$</td><td>$\begin{bmatrix} 3 & \vdots & 1 \\ 1 & \vdots & 2 \end{bmatrix}$</td><td>712.891</td><td>1,437</td><td>49</td><td>10</td></tr>
<tr><td>$\begin{bmatrix} 1 & 0 & 0 & \vdots & 49 \\ 0 & 1 & 0 & \vdots & 59 \\ 0 & 0 & 1 & \vdots & 39 \\ -1 & 0 & 0 & \vdots & 0 \\ 0 & -1 & 0 & \vdots & 0 \\ 0 & 0 & -1 & \vdots & 0 \end{bmatrix}$</td><td>$\begin{bmatrix} 10 & \vdots & -5 & \vdots & 0 \\ 0 & \vdots & 0 & \vdots & 5 \\ 0 & \vdots & 10 & \vdots & -5 \end{bmatrix}$</td><td>2.975.058.150</td><td>4.976</td><td>218</td><td>14</td></tr>
<tr><td>$\begin{bmatrix} 1 & 0 & 0 & \vdots & 10 \\ 2 & 1 & 0 & \vdots & 30 \\ -2 & 3 & 1 & \vdots & 20 \\ -1 & 0 & 0 & \vdots & 0 \\ 3 & -1 & 0 & \vdots & 0 \\ 0 & -2 & -1 & \vdots & 0 \end{bmatrix}$</td><td>$\begin{bmatrix} 10 & \vdots & 0 & \vdots & 0 \\ 0 & \vdots & 15 & \vdots & 0 \\ 0 & \vdots & 0 & \vdots & 5 \end{bmatrix}$</td><td>2.052.590</td><td>841.988</td><td>21</td><td>21</td></tr>
<tr><td>$\begin{bmatrix} 1 & 0 & 0 & \vdots & 100 \\ 1 & 1 & 0 & \vdots & 300 \\ 1 & 0 & 1 & \vdots & 200 \\ -1 & 0 & 0 & \vdots & 0 \\ -1 & -1 & 0 & \vdots & 0 \\ -1 & 0 & -1 & \vdots & 0 \end{bmatrix}$</td><td>$\begin{bmatrix} 2 & \vdots & 0 & \vdots & 0 \\ 0 & \vdots & 3 & \vdots & 0 \\ 0 & \vdots & 0 & \vdots & 1 \end{bmatrix}$</td><td>335.676.050</td><td>6.165</td><td>670</td><td>26</td></tr>
</table>

3.2 Εύρεση ορίων ενός tile

3.2.1 Μέθοδος ορθογώνιας σάρωσης του tile

Για να βρίσκεται ένα σημείο στο εσωτερικό ενός tile $\boldsymbol{t}$ με origin $\boldsymbol{i_{toi}}=H^{-1}\boldsymbol{t}$ πρέπει, όπως αναφέραμε και σε προηγούμενη παράγραφο να ικανοποιείται το σύστημα ανισοτήτων $\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot (i - i_{toi}) \le \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix} \Leftrightarrow \begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \le \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$.

Ταυτόχρονα, βρίσκεται εντός του Iteration Space αν και μόνο αν ικανοποιείται το σύστημα $B \cdot i \le \vec{b}$.

Επομένως ένα σημείο πρέπει να διατρεχθεί στα πλαίσια του συγκεκριμένου tile $\boldsymbol{t}$ αν και μόνο αν ικανοποιούνται ταυτόχρονα τα συστήματα $B \cdot i \le \vec{b}$ και $\begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \le \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$. Τα συστήματα αυτά μπορούν να γραφούν με τη μορφή ενός συστήματος ως εξής:

$$\begin{pmatrix} 0 & B \\ -gI & gH \\ gI & -gH \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \le \begin{pmatrix} \vec{b} \\ (g-1)\vec{1} \\ \vec{0} \end{pmatrix} \qquad (3.2.1)$$

Αν εφαρμόσουμε στο σύστημα αυτό τη μέθοδο απαλοιφής Fourier-Motzkin, θα πάρουμε τα όρια κάθε δείκτη i_k σαν συνάρτηση των δεικτών των εξωτερικότερων βρόχων $i_1,\ldots,i_{k-1}$ και $t_1,\ldots,t_n$. Παρατηρούμε ότι το σύστημα αυτό είναι πανομοιότυπο με αυτό που χρησιμοποιήσαμε κατά την περιγραφή της ακριβούς μεθόδου εύρεσης των ορίων του Tile Space (σχέση (*3.1.1*)). Η μόνη διαφορά έγκειται στο ότι μετά την ολοκλήρωση της μεθόδου Fourier-Motzkin, αν αναζητάμε τα όρια του Tile Space, θα αξιοποιήσουμε τις ανισότητες που παράχθηκαν και αναφέρονται στις πρώτες n μεταβλητές $t_1,\ldots,t_n$, ενώ αν αναζητάμε τα όρια του κάθε tile, θα αξιοποιήσουμε τις ανισότητες που αναφέρονται στις n τελευταίες μεταβλητές $i_1,\ldots,i_n$. Προφανώς, αν χρησιμοποιήσουμε την ακριβή μέθοδο για την εύρεση των ορίων του Tile Space και την παρούσα μέθοδο για την εύρεση των ορίων του κάθε tile, δεν είναι απαραίτητο να εφαρμόσουμε δύο φορές τη μέθοδο απαλοιφής Fourier-Motzkin.

Παράδειγμα 3.4

Για τον μετασχηματισμό tiling που περιγράψαμε στο παράδειγμα 2.1, λύσαμε στο παράδειγμα 3.3 το σύστημα της μορφής *(3.1.1)*, η οποία ταυτίζεται με τη μορφή *(3.2.1)*.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Προκειμένου να διατρέξουμε το εσωτερικό κάθε tile, θα χρησιμοποιήσουμε τις 14 τελευταίες ανισότητες του συστήματος που παράχθηκε, οι οποίες αναφέρονται στους δείκτες των δύο εσωτερικότερων βρόχων του τελικού κώδικα:

$$\begin{pmatrix} -48 & 0 & 9 & 0 \\ 0 & 24 & 1 & 0 \\ -96 & -48 & 16 & 0 \\ 0 & 0 & 1 & 0 \\ \hdashline 0 & -48 & -2 & 0 \\ 16 & 0 & -3 & 0 \\ 6 & 3 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & -48 & -2 & 6 \\ 16 & 0 & -3 & 1 \\ \hdashline 0 & 0 & 0 & -1 \\ -48 & 0 & 9 & -3 \\ 0 & 24 & 1 & -3 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ i_1 \\ i_2 \end{pmatrix} \le \begin{pmatrix} 140 \\ 93 \\ 141 \\ 15 \\ \hdashline 47 \\ 0 \\ 0 \\ 0 \\ \hline 31 \\ 47 \\ 0 \\ \hdashline 0 \\ 47 \\ 0 \end{pmatrix}$$

Επομένως ο παραγόμενος κώδικας θα έχει την εξής μορφή:

```
… … … … …
      for(i1=max(⌈(-47-48*t2)/2⌉,⌈16*t1/3⌉,6*t1+3*t2,0);
            i1<=min(⌊(140+48*t1)/9⌋,93-24*t2, ⌊(141+96*t1+48*t2)/16⌋,
            15); i1++)
            for(i2=max(0,⌈(-47-48*t1+9*i1)/3⌉,⌈(24*t2+i1)/3⌉);
                  i2<=min(31,⌊(47+48*t2+2*i1)/6⌋,-16*t1+3*i1); i2++)
            {
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

♦

Μία παραλλαγή της μεθόδου αυτής έχει ως εξής:

Μπορούμε να εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin για τα συστήματα:

$$B \cdot i \le \vec{b} \qquad (3.2.2)$$

και

$$\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot (i - i_{toi}) \le \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix} \qquad (3.2.3)$$

ξεχωριστά. Στη συνέχεια κάθε άνω (ή κάτω) όριο των μεταβλητών i_k εκφράζεται ως το minimum (ή maximum αντίστοιχα) των εκφράσεων που προκύπτουν από την επίλυση του *(3.2.3),* μετατοπισμένων κατά $i_{toi,k}$, και των εκφράσεων που προκύπτουν από την επίλυση του *(3.2.2)*.

Με την παραλλαγή αυτή, στην περίπτωση που δεν επιθυμούμε να χρησιμοποιήσουμε την ακριβή μέθοδο εύρεσης των ορίων του Tile Space, απαλλασσόμαστε από την υποχρέωση να εφαρμόσουμε τη μέθοδο Fourier-Motzkin για ένα μεγαλύτερο σύστημα όπως το *(3.2.1)*.

Επίσης, ο αριθμός των ανισοτήτων που προκύπτει από την υπέρθεση των συστημάτων *(3.2.2)* και *(3.2.3)* είναι συνήθως μικρότερος από τον αριθμό των ανισοτήτων που προκύπτουν αν εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin στο σύστημα *(3.2.1).* Το γεγονός αυτό οφείλεται στην αδυναμία της μεθόδου ακριβούς απλοποίησης να ανιχνεύσει ότι μια ανισότητα είναι περιττή στο Z^n αν το σύστημα που προκύπτει με την αντιστροφή της έχει λύση στο χώρο R^n.

Ταυτόχρονα δίδεται η δυνατότητα να ελέγχουμε ακόμη λιγότερες εκφράσεις κατά τη διάρκεια εκτέλεσης του τελικού προγράμματος για τα tiles που δεν είναι τοποθετημένα κοντά σε κάποια άκρη του Iteration Space. Το γεγονός αυτό μπορεί να εξοικονομήσει αρκετή υπολογιστική ισχύ και συνεπώς αρκετό χρόνο, αλλά προϋποθέτει την ύπαρξη κάποιας αποδοτικής μεθόδου διαχωρισμού των tiles σε εξωτερικά και εσωτερικά. Τότε ο τελικός κώδικας θα έχει τη μορφή:

```
if (tile crosses the iteration space bounds) then (run code with
                                expressions derived from (3.2.2) and (3.2.3))
else (run code only with expressions derived from (3.2.3))
```

Για παράδειγμα, σαν μέθοδος διαχωρισμού των tiles σε εξωτερικά και εσωτερικά μπορεί να χρησιμοποιηθεί ο έλεγχος αν όλες οι κορυφές του tile (οι οποίες είναι 2^n το πλήθος) βρίσκονται εντός των ορίων του iteration space. Αν αυτό ισχύει, τότε δεδομένου ότι οι χώροι που μας ενδιαφέρουν στην παρούσα εργασία είναι πάντα κυρτοί, θα ανήκουν στο iteration space και όλα τα άλλα σημεία του tile.

Παράδειγμα 3.5

Για το πρόβλημα του προηγούμενου παραδείγματος, αν εφαρμόσουμε την παραλλαγή αυτή, θα πρέπει να χρησιμοποιήσουμε τα συστήματα ανισοτήτων της μορφής *(3.2.2):*

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 15 \\ 0 \\ 31 \\ 0 \end{pmatrix}$$

και της μορφής *(3.2.3):*

$$\begin{pmatrix} 9 & -3 \\ -2 & 6 \\ -9 & 3 \\ 2 & -6 \end{pmatrix} \begin{pmatrix} i_1 - i_{toi,1} \\ i_2 - i_{toi,2} \end{pmatrix} \leq \begin{pmatrix} 47 \\ 47 \\ 0 \\ 0 \end{pmatrix}$$

Στο πρώτο από αυτά δεν χρειάζεται να εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin, επειδή έχει ήδη την επιθυμητή μορφή. Η εφαρμογή της μεθόδου στο δεύτερο από αυτά δίνει τις παρακάτω ανισότητες:

$$\begin{pmatrix} 16 & 0 \\ \hdashline -1 & 0 \\ \hline -2 & 6 \\ -3 & 1 \\ \hdashline 9 & -3 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} i_1 - i_{toi,1} \\ i_2 - i_{toi,2} \end{pmatrix} \leq \begin{pmatrix} 141 \\ \hdashline 0 \\ \hline 47 \\ 0 \\ \hdashline 47 \\ 0 \end{pmatrix}$$

Επομένως ο παραγόμενος κώδικας, λαμβάνοντας υπ’ όψη ότι $\lfloor 141/16 \rfloor = 8$, θα είναι ο εξής:

```
… … … … …
      i_toi,1=6*t_1+3*t_2;
      i_toi,2=2*t_1+9*t_2;
      for(i_1=max(0,i_toi,1); i_1<=min(15,i_toi,1+8); i_1++)
            for(i_2=max(0,i_toi,2+⌈(-47+9*i_1)/3⌉,i_toi,2+⌈i_1/3⌉);
                        i_2<=min(31,i_toi,2+⌊(47+2*i_1)/6⌋,i_toi,2+3*i_1); i_2++)
            {
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

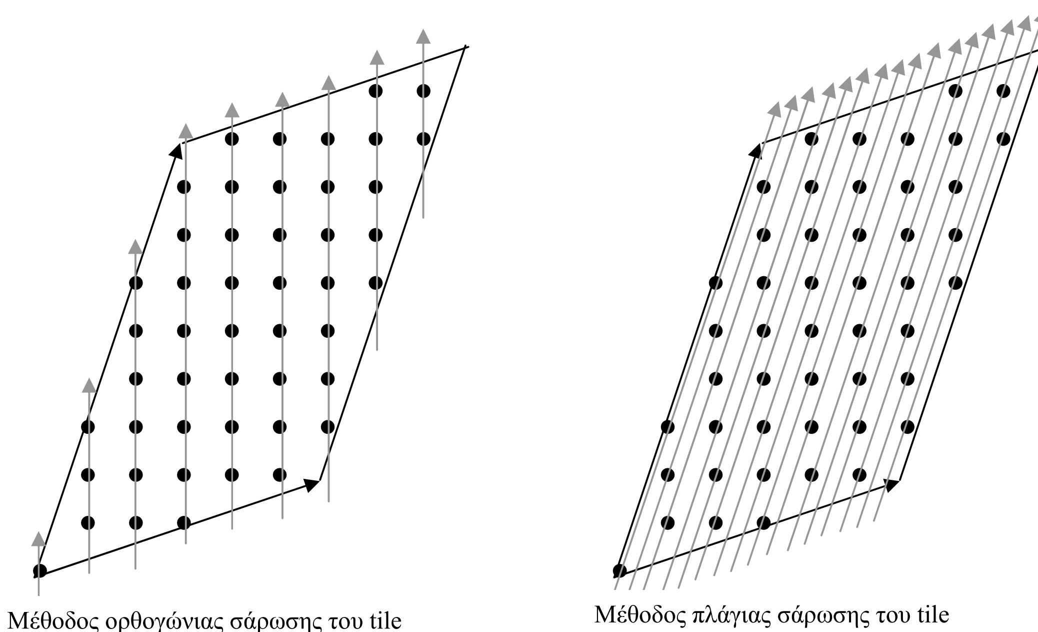
Παρατηρούμε ότι πράγματι ο παραγόμενος κώδικας πρέπει να αποτιμήσει λιγότερες εκφράσεις από αυτόν που παράχθηκε στο προηγούμενο παράδειγμα. Επίσης, για τα εσωτερικά tiles μπορούμε να απλοποιήσουμε ακόμη περισσότερο τον κώδικα ως εξής:

```
… … … … …
      i_toi,1=6*t1+3*t2;
      i_toi,2=2*t1+9*t2;
      for(i1=i_toi,1; i1<=i_toi,1+8; i1++)
            for(i2=max(i_toi,2+⌈(-47+9*i1)/3⌉,i_toi,2+⌈i1/3⌉);
                        i2<=min(i_toi,2+⌊(47+2*i1)/6⌋,i_toi,2+3*i1); i2++)
            {
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

♦

3.2.2 Μέθοδος πλάγιας σάρωσης του tile

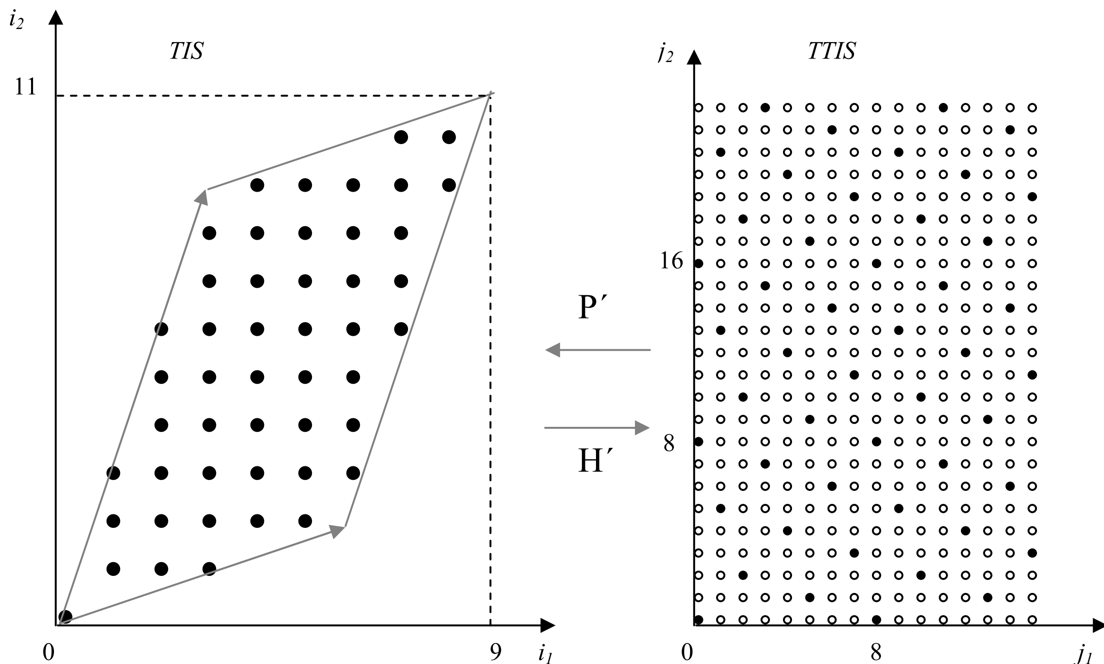
Η μέθοδος αυτή χρησιμοποιεί έναν non-unimodular μετασχηματισμό και διατρέχει το tile κινούμενη παράλληλα προς τις πλευρές του, σε αντίθεση με την προηγούμενη μέθοδο, η οποία κινείται κατά τις κατευθύνσεις των ορθοκανονικών αξόνων του *n*-διάστατου χώρου, όπως φαίνεται στο σχήμα 3.5. Σκοπός μας είναι να διατρέξουμε τον Tile Iteration Space και στη συνέχεια να ολισθήσουμε τα σημεία του κατά μια κατάλληλη σταθερά ώστε να συμπέσουν με τα σημεία ενός άλλου tile. Έτσι μπορούμε να διατρέξουμε όλον το χώρο των επαναλήψεων του αρχικού προγράμματος. Για να το επιτύχουμε μετασχηματίζουμε τον Tile Iteration Space (TIS) σε έναν ορθογώνιο χώρο, ο οποίος ονομάζεται Transformed Tile Iteration Space (TTIS). Διατρέχουμε τον TTIS με τη βοήθεια ενός φωλιασμένου βρόχου βάθους *n* και σε κάθε επανάληψη μετασχηματίζουμε τους δείκτες του βρόχου έτσι



Σχήμα 3.5

ώστε να επανερχόμαστε στον TIS.

Δηλαδή, αναζητούμε ένα ζευγάρι πινάκων μετασχηματισμού (P', H') τέτοιους ώστε $TTIS \xrightarrow{P'} TIS$ και $TIS \xrightarrow{H'} TTIS$, όπως φαίνεται στο σχήμα 3.6. Διαισθητικά, αντιλαμβανόμαστε ότι τα διανύσματα-στήλες του P' πρέπει να είναι παράλληλα προς τις ακμές του tile, δηλαδή παράλληλα προς τα διανύσματα-στήλες του P. Ισοδύναμα, πρέπει τα διανύσματα-γραμμές του H' να είναι παράλληλα προς τα διανύσματα-γραμμές του H. Επίσης, πρέπει το lattice του H' να αποτελείται από ακέραια σημεία, ώστε να μπορεί να διατρεχθεί από τους δείκτες ενός φωλιασμένου βρόχου. Επομένως αναζητάμε ένα n-διάστατο μετασχηματισμό H' τέτοιο ώστε $H'=MH$, όπου ο M είναι ένας $n\times n$ διαγώνιος πίνακας και $\mathcal{L}(H')\subseteq Z^n$. Στη συνέχεια θα αποδείξουμε ότι η δεύτερη προϋπόθεση ικανοποιείται αν και μόνο αν ο H' είναι ακέραιος.



Σχήμα 3.6

Λήμμα 3.2

Έστω ο μετασχηματισμός $\boldsymbol{j}=A\boldsymbol{i}$. Ισχύει $\boldsymbol{j}\in Z^n$ για κάθε $\boldsymbol{i}\in Z^n$ αν και μόνο αν ο πίνακας A είναι ακέραιος.

Απόδειξη

Αν ο πίνακας A αποτελείται μόνο από ακέραια στοιχεία, τότε είναι προφανές ότι θα ισχύει $\boldsymbol{j}\in Z^n$ για κάθε $\boldsymbol{i}\in Z^n$.

Αντιστρόφως, αν ισχύει $\boldsymbol{j}\in Z^n$ για κάθε $\boldsymbol{i}\in Z^n$, θα ισχύει και για $\boldsymbol{i}=\boldsymbol{u_k}$, όπου u_k είναι το μοναδιαίο διάνυσμα κατά την κατεύθυνση του k-οστού άξονα. Δηλαδή $\boldsymbol{u_k}=(u_{k1},u_{k2},\ldots,u_{kn})$ και $u_{kk}=1$, $u_{kj}=0$ αν $k\neq j$. Δηλαδή, $\boldsymbol{j}= A\boldsymbol{u_k}= (\sum_{i=1}^{n} a_{1i}u_{ki}, \sum_{i=1}^{n} a_{2i}u_{ki}, \ldots, \sum_{i=1}^{n} a_{ni}u_{ki})= (a_{1k}, a_{2k}, \ldots, a_{nk}) \in Z^n$. Η σχέση αυτή ισχύει για κάθε $k=1,\ldots,n$. Επομένως όλα τα στοιχεία του πίνακα A είναι ακέραιοι αριθμοί.

♦

Έστω ότι κατασκευάζουμε τον πίνακα M ως εξής: Κάθε διαγώνιο στοιχείο m_{kk} είναι ο ελάχιστος ακέραιος αριθμός που καθιστά το διάνυσμα $m_{kk}h_k$ ακέραιο, όπου h_k είναι η k-οστή γραμμή-διάνυσμα του πίνακα H. Τότε θα ικανοποιούνται και οι δύο προϋποθέσεις που θέσαμε προηγουμένως για τον πίνακα $H'=MH$.

Είναι προφανές ότι πίνακας H' που μόλις κατασκευάσαμε στη γενική περίπτωση θα είναι non-unimodular. Επομένως ο μετασχηματισμένος χώρος των επαναλήψεων μπορεί να περιέχει «τρύπες». Στο σχήμα 3.6 έχουμε παραστήσει με άσπρες κουκίδες τις «*τρύπες*» του μετασχηματισμένου χώρου και με μαύρες κουκίδες τα σημεία που αντιστοιχούν σε ακέραια σημεία του αρχικού χώρου. Άρα, για να διατρέξουμε τον αρχικό χώρο *TIS* θα πρέπει να διατρέξουμε όλα τα πραγματικά σημεία του μετασχηματισμένου χώρου *TTIS* και να τα μετασχηματίζουμε στον αρχικό χώρο χρησιμοποιώντας τον πίνακα μετασχηματισμού P'.

Προκειμένου να διατρέξουμε τα πραγματικά σημεία του μετασχηματισμένου χώρου χρησιμοποιούμε έναν φωλιασμένο βρόχο βάθους n με δείκτες $\boldsymbol{j}=(j_1,j_2,\ldots,j_n)=H'(\boldsymbol{i}-\boldsymbol{i_{toi}})$. Από την σχέση $\vec{0} \le H\cdot(i-i_{toi}) < \vec{1}$, η οποία εκφράζει τα όρια ενός tile παίρνουμε ισοδύναμα:

$$\vec{0} \le MH\cdot(i-i_{toi}) < M\vec{1} \Leftrightarrow \vec{0} \le H'\cdot(i-i_{toi}) < M\vec{1} \Leftrightarrow \begin{pmatrix}0\\0\\\vdots\\0\end{pmatrix} \le \begin{pmatrix}j_1\\j_2\\\vdots\\j_n\end{pmatrix} < \begin{pmatrix}m_{11}\\m_{22}\\\vdots\\m_{nn}\end{pmatrix} \Leftrightarrow$$

$$\begin{pmatrix}0\\0\\\vdots\\0\end{pmatrix} \le \begin{pmatrix}j_1\\j_2\\\vdots\\j_n\end{pmatrix} \le \begin{pmatrix}m_{11}-1\\m_{22}-1\\\vdots\\m_{nn}-1\end{pmatrix}$$

Επομένως εύκολα αντιλαμβάνεται κανείς ότι τα όρια μέσα στα οποία κινείται καθ' ένας από του ς δείκτες $j_1,j_2,\ldots,j_n$ είναι:

$$0 \le j_k \le m_{kk}-1 \qquad (3.2.4)$$

Το βήμα c_k, όμως, κάθε ενός από τους δείκτες $j_1,j_2,\ldots,j_n$ δεν ισούται πάντα με 1. Επίσης, όταν ο δείκτης j_k αυξάνεται κατά c_k, προκειμένου να πέσουμε σε πραγματικό

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

σημείο του μετασχηματισμένου χώρου και όχι σε τρύπα, πρέπει οι δείκτες $j_{k+1},\ldots,j_n$ να πάρουν κάποια αρχική τιμή μετατόπισης $\alpha_{(k+1)k},\ldots,\alpha_{nk}$.

Έστω ότι για κάποια τιμή του διανύσματος $\boldsymbol{j}$ ισχύει $P'j\in Z^n$. Τότε το πρώτο ερώτημα που πρέπει να αντιμετωπίσουμε είναι πόσο πρέπει να αυξήσουμε το δείκτη j_n έτσι ώστε το επόμενο σημείο του μετασχηματισμένου χώρου που θα σαρώσουμε να αντιστοιχεί σε ακέραιο σημείο του αρχικού χώρου. Δηλαδή αναζητούμε τον ελάχιστο ακέραιο $c_n\in Z^n$ για τον οποίο ισχύει $P'(j_1, \ldots, j_{n-1}, j_n+c_n)^T \in Z^n$.

Αφ' ότου βρούμε τη ζητούμενη τιμή του c_n, το επόμενο βήμα είναι να υπολογίσουμε το βήμα του j_{n-1} ώστε το επόμενο σημείο που θα σαρώσουμε να μην είναι τρύπα. Στην περίπτωση αυτή, είναι πιθανόν ότι μαζί με την αύξηση του j_{n-1} πρέπει ταυτόχρονα να δώσουμε μια κατάλληλη αρχική τιμή $\alpha_{n(n-1)}$ στη μεταβλητή j_n για την οποία θα ισχύει $0\le\alpha_{n(n-1)}<c_n$.

Γενικότερα, για κάθε δείκτη j_k πρέπει να υπολογίσουμε το βήμα του c_k και τις μετατοπίσεις $\alpha_{(k+1)k}, \ldots, \alpha_{nk}$ των εσωτερικότερων δεικτών, έτσι ώστε να ισχύει: $P'(j_1, \ldots, j_{k-1}, j_k+c_k, j_{k+1}+\alpha_{(k+1)k}, \ldots, j_n+\alpha_{nk})^T \in Z^n$. Για κάθε δείκτη βρόχου j_k έχουμε $k-1$ διαφορετικές τιμές μετατοπίσεων α_{ki}, οι οποίες αντιστοιχούν στα βήματα c_i των $k-1$ δεικτών των εξωτερικότερων βρόχων j_i ($i=1,\ldots,k-1$). Αυτές οι τιμές μετατοπίσεων είναι οι $\alpha_{k1},\ldots,\alpha_{k(k-1)}$. Στο επόμενο λήμμα αποδεικνύεται ότι τα βήματα c_k και οι τιμές των μετατοπίσεων α_{kl} δίνονται απ' ευθείας από τα στοιχεία της κανονικής ερμητιανής μορφής $\widetilde{H}'$ του πίνακα H'.

Λήμμα 3.3

Αν $\widetilde{H}'$ είναι η κανονική ερμητιανή μορφή στηλών του πίνακα H' και $\boldsymbol{j}=(j_1,\ldots,j_n)$ είναι το διάνυσμα που χρησιμοποιούμε για να διατρέξουμε τα πραγματικά σημεία του $\mathcal{L}(H')$, τότε το βήμα του δείκτη j_k είναι $c_k=\widetilde{h}'_{kk}$ και οι αρχικές μετατοπίσεις του είναι $\alpha_{kl}=\widetilde{h}'_{kl}$, ($k$=1, …, n και l= 1, …, k-1).

Απόδειξη

Ισχύει $\mathcal{L}(H')=\mathcal{L}(\widetilde{H}')$ και $\vec{0}\in\mathcal{L}(H')$. Επίσης, οι στήλες του $\widetilde{H}'$ ανήκουν στο $\mathcal{L}(H')$. Έστω διάνυσμα $\vec{x}\in Z^n/\vec{0}$, τέτοιο ώστε $x_j=0$ για κάθε $j<k$ και $0\le x_j\le \widetilde{h}'_{jk}$ για κάθε $k\le j\le n$ και το διάνυσμα $\vec{x}$ δεν ισούται με καμία από τις στήλες του $\widetilde{H}'$. Αρκεί να αποδείξουμε ότι $\vec{x}\notin\mathcal{L}(H')$.

Ακολουθώντας τη μέθοδο της εις άτοπον απαγωγής, υποθέτουμε ότι $\vec{x}\in\mathcal{L}(H')=\mathcal{L}(\widetilde{H}')$, δηλαδή ότι υπάρχει κάποιο $i\in Z^n$ τέτοιο ώστε $\widetilde{H}'i = \vec{x}$. Ο πίνακας $\widetilde{H}'$ είναι κάτω τριγωνικός και όλα τα στοιχεία του είναι μη αρνητικά, επομένως έχουμε: $x_1=\widetilde{h}'_{11}\, i_1=0 \Rightarrow i_1=0$. Όμοια αποδεικνύεται ότι $i_j=0$ για κάθε $j<k$. Επίσης ισχύει $x_k = \widetilde{h}'_{kk}\, i_k$. Σύμφωνα με τα παραπάνω, πρέπει να ισχύει $0 \le x_k = \widetilde{h}'_{kk}\, i_k \le \widetilde{h}'_{kk} \Rightarrow 0 \le i_k \le 1$. Επίσης, έχουμε $0\le x_{k+1} =$

$\widetilde{h}'_{(k+1)k}\, i_k + \widetilde{h}'_{(k+1)(k+1)}\, i_{k+1} \le \widetilde{h}'_{(k+1)k}$. Επειδή $\widetilde{h}'_{(k+1)(k+1)} \ge \widetilde{h}'_{(k+1)k}$ έπεται ότι $i_{k+1}=0$. Όμοια αποδεικνύεται ότι $i_j=0$ για κάθε $j>k+1$.

Άρα έχουμε $\vec{x}=\vec{0}$, ή το $\vec{x}$ ισούται με την k-οστή στήλη του $\widetilde{H}'$, πράγμα το οποίο έρχεται σε αντίθεση με τις αρχικές υποθέσεις μας. Επομένως $\vec{x} \notin \mathcal{L}(H')$ και δεν υπάρχουν ακέραιοι αριθμοί μικρότεροι από αυτούς που ορίστηκαν στο παραπάνω λήμμα που να μπορούν να χρησιμοποιηθούν ως βήματα ή ως αρχικές μετατοπίσεις των δεικτών του ζητούμενου μετασχηματισμένου βρόχου.

♦

Σύμφωνα με την παραπάνω ανάλυση, το σημείο που πρέπει να σαρωθεί από την επόμενη τιμή των δεικτών των φωλιασμένων βρόχων υπολογίζεται από την παρούσα τιμή τους ,αφού τα βήματα και οι μετατοπίσεις προστίθενται στις παρούσες τιμές των δεικτών.

Στο σημείο αυτό οφείλουμε να παρατηρήσουμε τα εξής: Πρώτον, στον κώδικα που ακολουθεί, πριν τους φωλιασμένους βρόχους, οι δείκτες αρχικοποιούνται με τις τιμές: $j_2= -\alpha_{21}$, …, $j_n= -\alpha_{n1}$. Αυτό γίνεται ώστε τελικά στην πρώτη επανάληψη που θα πραγματοποιηθεί να είναι $\boldsymbol{j}=(j_1,\ldots,j_n)=\vec{0}$. Δεύτερον, κάθε ένας από τους φωλιασμένους βρόχους πρέπει να επιλέγει κάθε φορά την τιμή της αρχικής μετατόπισης του αντίστοιχου δείκτη j_k ανάμεσα στις πιθανές μετατοπίσεις α_{k1}, α_{k2}, …, $\alpha_{k(k-1)}$. Υπενθυμίζουμε ότι α_{kl} είναι η κατάλληλη μετατόπιση του δείκτη j_k όταν ο δείκτης j_l ($l<k$) αυξάνεται κατά το βήμα του c_l. Η ακόλουθη μορφή φωλιασμένων βρόχων δίδει μια λύση στο πρόβλημα αυτό χρησιμοποιώντας μια επιπλέον μεταβλητή *phase*, η οποία δείχνει ποιος δείκτης έχει μόλις αυξηθεί. Η έκφραση του υπολοίπου ακέραιας διαίρεσης (*mod* %) χρησιμοποιείται για να σαρωθεί πρώτο το μικρότερο σημείο σε κάθε κατεύθυνση.

Θεώρημα 3.1

Ο ακόλουθος κώδικας με φωλιασμένους βρόχους βάθους n είναι κατάλληλος για να διατρέξει όλα τα σημεία του μετασχηματισμένου χώρου των επαναλήψεων ενός tile (*TTIS*) με τη λεξικογραφική τους σειρά.

```
j2=-H̃'[2][1]; …; jn=-H̃'[n][1]; phase=1;
for(j1=0; j1<=m11-1; j1+=c1, phase=1)
      for(j2=(j2+H̃'[2][phase])%c2; j2<=m22-1; j2+=c2, phase=2)
      … … … … … …
            for(jn=(jn+H̃'[n][phase])%cn; jn<=mnn-1; jn+=cn,
                                                         phase=n)
            {
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

Απόδειξη

Εξάγεται απ' ευθείας από τα λήμματα 3.2 και 3.3

♦

Μία διαφορετική αντιμετώπιση του προβλήματος αυτού δίδεται από τον παρακάτω κώδικα, ο οποίος είναι πιο πολύπλοκος, αλλά λύνει πιο αποδοτικά κάποια προβλήματα που θα παρουσιαστούν παρακάτω.

Θεώρημα 3.2

Ο ακόλουθος κώδικας με φωλιασμένους βρόχους βάθους *n* είναι κατάλληλος για να διατρέξει όλα τα σημεία του μετασχηματισμένου χώρου των επαναλήψεων ενός tile (*TTIS*) με τη λεξικογραφική τους σειρά.

```
lb1=lb2=…=lbn=0;                    //lower bounds of indexes
ub1=m11-1; ub2=m22-1; …; ubn=mnn-1; //upper bounds of indexes
for(j1=⌈lb1/H~'[1][1]⌉*H~'[1][1], j2=⌈lb1/H~'[1][1]⌉*H~'[2][1], …,
      jn=⌈lb1/H~'[1][1]⌉*H~'[n][1]; j1<=ub1; j1+=H~'[1][1],
      j2+=H~'[2][1], …, jn+=H~'[n][1])
      for(j2+=⌈(lb2-j2)/H~'[2][2]⌉*H~'[2][2],
            j3+=⌈(lb2-j2)/H~'[2][2]⌉*H~'[3][2], …,
            jn+=⌈(lb2-j2)/H~'[2][2]⌉*H~'[3][2]; j2<=ub2; j2+=H~'[2][2],
            j3+=H~'[3][2], …, jn+=H~'[n][2])
      … … … … … …
            for(jn+=⌈(lbn-jn)/H~'[n][n]⌉*H~'[n][n]; jn<=ubn;
                  jn+=H~'[n][n])
            {
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

Απόδειξη

Στον κώδικα αυτό εξασφαλίζουμε ότι βρισκόμαστε πάντα σε κάποιο πραγματικό σημείο του μετασχηματισμένου χώρου, φροντίζοντας όλες οι μετατοπίσεις των διανυσμάτων-δεικτών των φωλιασμένων βρόχων να είναι ακέραιο πολλαπλάσιο κάποιας στήλης του πίνακα $\widetilde{H}'$.

♦

Στη συνέχεια, πρέπει να προσαρμόσουμε τους παραπάνω φωλιασμένους βρόχους, οι οποίοι σαρώνουν όλα τα σημεία του μετασχηματισμένου χώρου *TTIS*, έτσι ώστε να σαρώνουν τα εσωτερικά σημεία κάθε tile του Tile Space I^S, τα οποία αντιστοιχούν σε κάποιο σημείο του χώρου I^n. Έστω ότι $\boldsymbol{j} \in TTIS$ είναι το διάνυσμα που προκύπτει από τους δείκτες των προηγούμενων φωλιασμένων βρόχων, $\boldsymbol{t} \in I^S$ είναι το tile του οποίου τα

εσωτερικά σημεία $\boldsymbol{i} \in I^n$ θέλουμε να διατρέξουμε και $\boldsymbol{i_{toi}} \in TOS$ είναι το σημείο εκκίνησης του tile $\boldsymbol{t}$. Τότε ισχύει $\boldsymbol{i} = \boldsymbol{i_{toi}} + P'\boldsymbol{j} = P\boldsymbol{t} + P'\boldsymbol{j}$. Επίσης ισχύει $P = P'M$, οπότε από την προηγούμενη σχέση παίρνουμε:

$$\boldsymbol{i} = P'(M\boldsymbol{t} + \boldsymbol{j}) \qquad (3.2.5)$$

Επειδή ο πίνακας M είναι διαγώνιος το διάνυσμα $M\boldsymbol{t}$ υπολογίζεται από το $\boldsymbol{t}$ με την εκτέλεση n πολλαπλασιασμών. Δηλαδή αρκεί να πολλαπλασιάσουμε κάθε στοιχείο t_k του διανύσματος $\boldsymbol{t}$ με το αντίστοιχο στοιχείο m_{kk} του πίνακα M.

Όπως αναφέραμε και προηγουμένως πρέπει να προσέξουμε ιδιαίτερα ώστε τα σημεία $\boldsymbol{j}$ του μετασχηματισμένου χώρου τα οποία θα διατρέξουμε να μην αντιστοιχούν σε σημεία $\boldsymbol{i}$ του αρχικού χώρου τα οποία βρίσκονται έξω από τα όρια του I^n. Ένα σημείο $i \in I^S$ αν και μόνο αν ικανοποιείται το σύστημα ανισοτήτων $B \cdot i \leq \vec{b}$. Ισοδύναμα, αν στη σχέση αυτή αντικαταστήσουμε το διάνυσμα $\boldsymbol{i}$ σύμφωνα με την *(3.2.5)* παίρνουμε:

$$BP' \cdot (j + M \cdot t) \leq \vec{b} \qquad (3.2.6)$$

ή $BP' \cdot x \leq \vec{b}$ όπου $x_k = j_k + m_{kk} t_k$. Επομένως, προκειμένου να βρούμε τα όρια μέσα στα οποία πρέπει να κινείται κάθε ένας από τους δείκτες j_k για να μην ξεφύγουμε από το χώρο I^n, αρκεί εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin στο σύστημα $BP' \cdot x \leq \vec{b}$ και στη συνέχεια από κάθε όριο που θα προκύψει και θα αντιστοιχεί στη μεταβλητή x_k να αφαιρούμε την τιμή $m_{kk} t_k$, για να προκύψει το αντίστοιχο όριο της j_k. Για παράδειγμα, αν από την επίλυση του $BP' \cdot x \leq \vec{b}$ προκύψει κάποιο όριο της μορφής

$$l_k(x_1, \ldots, x_{k-1}) \leq x_k \leq u_k(x_1, \ldots, x_{k-1}),$$

τότε αυτό μετασχηματίζεται σε:

$$l_k(j_1 + m_{11} t_1, \ldots, j_{k-1} + m_{(k-1)(k-1)} t_{k-1}) - m_{kk} t_k \leq x_i \leq u_k(j_1 + m_{11} t_1, \ldots, j_{k-1} + m_{(k-1)(k-1)} t_{k-1}) - m_{kk} t_k.$$

Η μορφή αυτή είναι κατάλληλη για να εκφράσει τα όρια των n εσωτερικότερων φωλιασμένων βρόχων του κώδικά μας.

Με τον τρόπο αυτό, μεταξύ των άλλων, λύνεται και το πρόβλημα των περιττών tiles που πιθανόν να έχουν προκύψει αν έχουμε προηγουμένως χρησιμοποιήσει την γρήγορη μέθοδο για την εύρεση των ορίων του Tile Space. Δηλαδή, διασφαλίζεται ότι αν ένα tile δεν περιέχει καμία επανάληψη του αρχικού χώρου I^n, αυτό θα ανιχνευθεί κατά την διάρκεια εκτέλεσης του προγράμματος.

Επίσης, παρατηρούμε στους πίνακες BP και BP', που εκφράζουν τα πρώτα μέλη των συστημάτων *(3.1.4)* και *(3.2.6),* ότι ο δεύτερος προκύπτει από τον πρώτο με διαίρεση κάθε στήλης k με μία θετική σταθερά m_{kk}. Αφού τα βήματα της μεθόδου απαλοιφής Fourier-Motzkin εξαρτώνται μόνο από τη μορφή των πινάκων αυτών, μπορούμε να εφαρμόσουμε

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

τη μέθοδο και στα δύο συστήματα εκτελώντας τα βήματά της μόνο μια φορά. Δηλαδή, αν εφαρμόσουμε τη μέθοδο Fourier-Motzkin στον πίνακα $[BP|\vec{b}'|BP'|\vec{b}\,]$ φροντίζοντας να φέρουμε τον πίνακα *BP* στην κατάλληλη μορφή, αυτόματα θα πάρουμε και τον *BP΄* στην κατάλληλη μορφή.

Επομένως, ενδείκνυται η γρήγορη μέθοδος για την εύρεση των ορίων του Tile Space να χρησιμοποιείται σε συνδυασμό με την παρούσα, αφού αυτό θα μας επιτρέψει με μία μόνο εφαρμογή της ιδιαίτερα πολύπλοκης μεθόδου Fourier-Motzkin να λύνουμε και τα δύο προβλήματα.

Κατά την παραγωγή του τελικού κώδικα SPMD, θα πρέπει να αναθέτουμε στις μεταβλητές lb_k του κώδικα του θεωρήματος 3.2 το maximum του 0 και των ορίων που προκύπτουν από το σύστημα *(3.2.6)* με τον τρόπο που περιγράψαμε. Όμοια, θα πρέπει να αναθέτουμε στις μεταβλητές ub_k το minimum του m_{kk}-1 και των ορίων που προκύπτουν από το σύστημα *(3.2.6)*.

Παράδειγμα 3.6

Έστω ότι θέλουμε να εφαρμόσουμε τη μέθοδο πλάγιας σάρωσης των επαναλήψεων ενός tile στους μετασχηματισμούς tiling των παραδειγμάτων 3.4 και 3.5:

Από τον πίνακα $H=\begin{pmatrix}\frac{3}{16} & -\frac{1}{16}\\ -\frac{1}{24} & \frac{1}{8}\end{pmatrix}$, κατασκευάζουμε τον πίνακα μετασχηματισμού $H'=MH=\begin{pmatrix}3 & -1\\ -1 & 3\end{pmatrix}$, όπου $M=\begin{pmatrix}16 & 0\\ 0 & 24\end{pmatrix}$. Ο πίνακας αντίστροφου μετασχηματισμού είναι ο $P'=\begin{pmatrix}\frac{3}{8} & \frac{1}{8}\\ \frac{1}{8} & \frac{3}{8}\end{pmatrix}$.

Η κανονική ερμητιανή μορφή του *H΄* είναι: $\widetilde{H}'=\begin{pmatrix}1 & 0\\ 5 & 8\end{pmatrix}$. Επομένως, τα βήματα των δεικτών j_1 και j_2 είναι αντίστοιχα c_1=1 και c_2=8. Η μετατόπιση του j_2 όταν ο j_1 αυξάνεται κατά c_1=1 είναι α_{21}=5. Πράγματι, από το σχήμα 3.6 μπορεί κανείς να επαληθεύσει ότι με τις τιμές αυτές θα κινούμαστε πάντα πάνω σε πραγματικά σημεία του μετασχηματισμένου χώρου TTIS.

Το σύστημα (*3.2.6*) στη συγκεκριμένη περίπτωση γράφεται ως εξής:

$$\begin{pmatrix} \frac{3}{8} & \frac{1}{8} \\ -\frac{3}{8} & -\frac{1}{8} \\ \frac{1}{8} & \frac{3}{8} \\ -\frac{1}{8} & -\frac{3}{8} \end{pmatrix} \begin{pmatrix} j_1 + 16t_1 \\ j_2 + 24t_2 \end{pmatrix} \leq \begin{pmatrix} 15 \\ 0 \\ 31 \\ 0 \end{pmatrix}$$

Αν σε αυτό εφαρμόσουμε τη μέθοδο απαλοιφής Fourier-Motzkin παίρνουμε:

$$\begin{pmatrix} 1 & 0 \\ \hdashline -1 & 0 \\ \hline 3 & 1 \\ 1 & 3 \\ \hdashline -3 & -1 \\ -1 & -3 \end{pmatrix} \begin{pmatrix} j_1 + 16t_1 \\ j_2 + 24t_2 \end{pmatrix} \leq \begin{pmatrix} 45 \\ \hdashline 31 \\ \hline 120 \\ 248 \\ \hdashline 0 \\ 0 \end{pmatrix}$$

Επομένως ο παραγόμενος κώδικας θα είναι ο εξής:

```
… … … … …
      j_toi,1=16*t_1; j_toi,2=24*t_2;
      lb_1=max(0,-31-j_toi,1); ub_1=min(15,45-j_toi,1);
      for(j_1=lb_1, j_2=lb_1*5; j_1<=ub_1; j_1+=1, j_2+=5)
      {
            lb_2=max(0,-3*(j_1+j_toi,1)-j_toi,2,⌈-(j_1+j_toi,1)/3⌉-j_toi,2);
            ub_2=min(23,120-3*(j_1+j_toi,1)-j_toi,2,
                                          ⌊(248-(j_1+j_toi,1))/3⌋-j_toi,2);
            for(j_2+=⌈(lb_2-j_2)/8⌉*8; j_2<=ub_2; j_2+=8)
            {
                  i_1=3/8*(j_toi,1+j_1)+1/8*(j_toi,2+j_2);
                  i_2=1/8*(j_toi,1+j_1)+3/8*(j_toi,2+j_2);
                  Εντολές σώματος φωλιασμένων βρόχων
            }
      }
```

♦

Μπορούμε, όπως και στην προηγούμενη μέθοδο, να διαχωρίσουμε τα tiles σε εσωτερικά (αυτά δηλαδή που δεν τέμνονται από κανένα όριο του Iteration Space) και σε εξωτερικά (αυτά που πιθανόν κάποια σημεία τους να μην ανήκουν στο Iteration Space). Τότε για τα πρώτα θα χρησιμοποιούμε μόνο τα όρια των εκφράσεων *(3.2.4),* ενώ για τα δεύτερα θα χρησιμοποιούμε συνδυασμό των εκφράσεων *(3.2.4)* και *(3.2.6)*. Ο παραγόμενος κώδικας θα έχει τη μορφή:

```
if (tile crosses the iteration space bounds) then (run code with
                          expressions derived from (3.2.4) and (3.2.6))
else (run code only with expressions (3.2.4))
```

Στην περίπτωση αυτή ο κώδικας που αφορά τα εσωτερικά tiles, τα οποία θα είναι η πλειοψηφία αυτών που πρέπει να διατρεχθούν, θα είναι ιδιαίτερα απλός. Θα περιέχει μία μόνο σταθερά ως κάτω όριο και μία σταθερά ως άνω όριο της κάθε μεταβλητής. Αντίθετα, η προηγούμενη μέθοδος για τα εσωτερικά tiles χρησιμοποιεί τις εκφράσεις που προκύπτουν από την επίλυση του συστήματος *(3.2.3)* και οι οποίες στη γενική περίπτωση έχουν αρκετά πιο πολύπλοκη μορφή.

Παράδειγμα 3.7

Για το πρόβλημα του προηγούμενου παραδείγματος, τα εσωτερικά tiles μπορούν να σαρωθούν, σύμφωνα με το θεώρημα 3.2 από τον εξής κώδικα:

```
… … … … …
      j_toi,1=16*t_1; j_toi,2=24*t_2;
      for(j_1=0, j_2=0; j_1<=15; j_1+=1, j_2+=5)
            for(j_2+=⌈(-j_2)/8⌉*8; j_2<=23; j_2+=8)
            {
                  i_1=3/8*(j_toi,1+j_1)+1/8*(j_toi,2+j_2);
                  i_2=1/8*(j_toi,1+j_1)+3/8*(j_toi,2+j_2);
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

ή, ακόμη απλούστερα, σύμφωνα με το θεώρημα 3.1, από τον κώδικα:

```
… … … … …
      j_toi,1=16*t_1; j_toi,2=24*t_2;
      j_2=-5;
      for(j_1=0; j_1<=15; j_1+=1)
            for(j_2=(j_2+5)%8; j_2<=23; j_2+=8)
            {
                  i_1=3/8*(j_toi,1+j_1)+1/8*(j_toi,2+j_2);
                  i_2=1/8*(j_toi,1+j_1)+3/8*(j_toi,2+j_2);
                  Εντολές σώματος φωλιασμένων βρόχων
            }
```

♦

3.3 Λήψη, αποστολή δεδομένων

Όπως αναφέραμε και στο πρώτο κεφάλαιο, τα tiles είναι πάντα ατομικά. Δηλαδή ο κάθε επεξεργαστής λαμβάνει και στέλνει δεδομένα από και προς τους γειτονικούς του πριν και μετά την εκτέλεση των υπολογισμών που αφορούν ένα tile. Στην παράγραφο αυτή θα ασχοληθούμε με το πρόβλημα της αναγνώρισης σε κάθε tile των επαναλήψεων που δίνουν

δεδομένα απαραίτητα για τους υπολογισμούς των γειτονικών tiles. Τις επαναλήψεις αυτές τις καλούμε ***σημεία επικοινωνίας (communication points)***.

Αν γνωρίζουμε τα σημεία επικοινωνίας σε κάθε tile, μπορούμε εύκολα να κατασκευάσουμε τις παραμέτρους των εντολών αποστολής και λήψης δεδομένων στην αρχή και στο τέλος της εκτέλεσης κάθε tile. Χρειαζόμαστε $n\text{-}1$ αποστολές δεδομένων στο τέλος κάθε tile (και όμοια $n\text{-}1$ εντολές λήψης δεδομένων στην αρχή κάθε tile), μία για κάθε έναν από τους γειτονικούς επεξεργαστές, δεδομένου ότι σε κάθε επεξεργαστή έχουμε αναθέσει τα tiles που βρίσκονται κατά μήκος μίας διάστασης του Tile Space (όπως στο σχήμα 1.6β). Κάθε εντολή αποστολής πρέπει να περιλαμβάνει ένα πακέτο με όλα τα δεδομένα επικοινωνίας που είναι απαραίτητα στο συγκεκριμένο επεξεργαστή. Είναι προφανές ότι ο τρόπος εύρεσης των σημείων επικοινωνίας εξαρτάται από τη μέθοδο που χρησιμοποιούμε για την σάρωση των εσωτερικών σημείων του κάθε tile.

Κάθε σημείο επικοινωνίας έχει την εξής ιδιότητα: Αν σε αυτό προσθέσουμε ένα διάνυσμα εξάρτησης, το σημείο που θα προκύψει δεν θα ανήκει στο ίδιο tile. Δηλαδή, αν ένα σημείο $\boldsymbol{i}$ είναι σημείο επικοινωνίας, τότε για κάποιο από τα διανύσματα εξάρτησης $\boldsymbol{d_k}$ δεν ισχύει:

$$\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot \left((i + d_k) - i_{toi} \right) \leq \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix} \qquad (3.3.1)$$

Επομένως αν διατρέχουμε το εσωτερικό κάθε tile χρησιμοποιώντας τη μέθοδο ορθογώνιας σάρωσης, πρέπει να προσθέσουμε όλα τα διανύσματα εξάρτησης ξεχωριστά σε κάθε σημείο του *TIS* και για καθ' ένα να εξετάσουμε αν ικανοποιείται το παραπάνω σύστημα ανισοτήτων. Αν το σύστημα δεν ικανοποιείται, είναι εύκολο να αποφανθούμε σε ποιον από τους γειτονικούς επεξεργαστές πρέπει να αποσταλούν τα αντίστοιχα δεδομένα, εξετάζοντας απλώς ποια ή ποιες από τις ανισότητες του συστήματος δεν επαληθεύονται.

Αν διατρέχουμε τα εσωτερικά σημεία κάθε tile με τη μέθοδο πλάγιας διάσχισης, με τη βοήθεια ενός non-unimodular μετασχηματισμού, η εύρεση των σημείων επικοινωνίας είναι πολύ ευκολότερη. Μετασχηματίζουμε τον πίνακα εξαρτήσεων D, με τη βοήθεια του πίνακα μετασχηματισμού H', έτσι ώστε να βρούμε το σύνολο των διανυσμάτων εξάρτησης στο μετασχηματισμένο χώρο. Αν συμβολίσουμε με D' το μετασχηματισμένο πίνακα εξαρτήσεων, ισχύει $D'=H'D$. Αν στη σχέση $\vec{0} \leq H((\boldsymbol{i}+\boldsymbol{d_k})-\boldsymbol{i_{toi}}) < \vec{1}$ αντικαταστήσουμε $\boldsymbol{i}-\boldsymbol{i_{toi}}=(H')^{-1}\boldsymbol{j}=(MH)^{-1}\boldsymbol{j}=H^{-1}M^{-1}\boldsymbol{j}$ και $\boldsymbol{d_k}=(H')^{-1}\boldsymbol{d_k}'=(MH)^{-1}\boldsymbol{d_k}'=H^{-1}M^{-1}\boldsymbol{d_k}'$, προκύπτει ισοδύναμα: $\vec{0} \leq HH^{-1}M^{-1}(\boldsymbol{j}+\boldsymbol{d_k}') < \vec{1} \Leftrightarrow \vec{0} \leq (\boldsymbol{j}+\boldsymbol{d_k}') < M\vec{1}$. Αν λάβουμε υπ' όψη ότι για έναν έγκυρο μετασχηματισμό όλα τα μετασχηματισμένα διανύσματα εξάρτησης έχουν μόνο μη αρνητικές συντεταγμένες, οι ανισότητες $\vec{0} \leq (\boldsymbol{j}+\boldsymbol{d_k}')$ ικανοποιούνται πάντα. Επομένως η απόφαση αν ένα σημείο είναι σημείο επικοινωνίας θα βασίζεται στην εξέταση των

σχέσεων $(\boldsymbol{j}+\boldsymbol{d_k}')<M\vec{1}$ για κάθε $k=1,\ldots,\rho$, όπου ρ είναι ο αριθμός των διανυσμάτων εξάρτησης του συγκεκριμένου προβλήματος. Δηλαδή, ένα σημείο $\boldsymbol{j}$ του μετασχηματισμένου χώρου είναι σημείο επικοινωνίας αν <u>*δεν ισχύει*</u> $j_l+max(d'_{lk})\leq m_{kk}-1$, ή ισοδύναμα αν ισχύει

$$j_l \geq m_{ll}-max(d'_{lk}) \qquad (3.3.2)$$

όπου $k=1,\ldots,\rho$ για κάποιο $l=1,\ldots,n$. Τα δεδομένα του σημείου αυτού θα πρέπει να σταλούν διαμέσου της επιφάνειας που είναι κάθετη στο l-οστό μοναδιαίο διάνυσμα του μετασχηματισμένου χώρου. Αν η επιφάνεια αυτή χωρίζει μεταξύ τους tiles που έχουν ανατεθεί στον ίδιο επεξεργαστή, δεν χρειάζεται να σταλεί μήνυμα δεδομένων σε κανέναν γειτονικό επεξεργαστή.

Παράδειγμα 3.8

Στο πρόβλημα των προηγούμενων παραδειγμάτων ο πίνακας εξαρτήσεων είναι ο

$$D=\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

Ο μετασχηματισμένος πίνακας εξαρτήσεων είναι ο $D'=H'D=\begin{pmatrix} 8 & 0 \\ 0 & 8 \end{pmatrix}$.

Επειδή $max(d'_{11}, d'_{12})=8$ και $max(d'_{21}, d'_{22})=8$, το σημείο $\boldsymbol{j}=(j_1, j_2)$ είναι σημείο επικοινωνίας αν ισχύει $j_1\geq 16-8=8$ ή $j_2\geq 24-8=16$. Αν όμως τα tiles που έχουν όμοια συντεταγμένη t_1 (όπως τα (-1,0), (0,0), (1,0), (2,0)) ανατεθούν στον ίδιο επεξεργαστή, τότε μόνο τα δεδομένα των σημείων με $j_2\geq 16$ πρέπει να σταλούν στο γειτονικό επεξεργαστή.

♦

© Μαρια Αθανασακη

Κεφάλαιο 4: Υλοποίηση

4.1 Παρουσίαση υλοποιημένου κώδικα

Ο κώδικας της παρούσας διπλωματικής εργασίας είναι γραμμένος σε *C++*. Βασίζεται σε κλάσεις γραμμένες από τον *Γιώργο Γκούμα* μετά από κάποιες τροποποιήσεις και προσθήκες. Συγκεκριμένα, χρησιμοποιήσαμε την κλάση "*rational*", η οποία απεικονίζει έναν ρητό αριθμό σε μορφή ανάγωγου κλάσματος, καθώς και την κλάση "*matrix*", η οποία απεικονίζει έναν δισδιάστατο πίνακα ρητών αριθμών. Ο κώδικας των κλάσεων αυτών, καθώς και τα αρχεία που είναι απαραίτητα για τη μεταγλώττισή τους παρατίθενται στο Παράρτημα 1, στο τέλος της παρούσας εργασίας.

Στη συνέχεια, υλοποιήσαμε στο αρχείο "*Fourier.cc*" τη μέθοδο απαλοιφής Fourier-Motzkin, όπως περιγράφηκε στην παράγραφο 2.2. Αυτή μπορεί να χρησιμοποιηθεί με την κλήση της συνάρτησης

***matrix** *Fourier_Motzkin_Elimination(**const matrix**& Ab, **bool**& consistent,*
***const bool** Ad_Hoc, **const bool** exact)*

Η συνάρτηση αυτή δέχεται ως ορίσματα τον πίνακα *Ab*, ο οποίος είναι ο επαυξημένος πίνακας που περιγράφει το σύστημα ανισοτήτων που θέλουμε να επιλύσουμε, και τις boolean δομές *Ad_Hoc* και *exact*, οι οποίες εκφράζουν αν επιθυμούμε στο σύστημα που θα προκύψει να εφαρμοστεί απλοποίηση Ad-Hoc ή ακριβής απλοποίηση, αντίστοιχα. Αν και τα δύο ορίσματα έχουν την τιμή true, τότε στο προκύπτον σύστημα εφαρμόζεται πρώτα

απλοποίηση Ad-Hoc και έπειτα ακριβής απλοποίηση. Στην παράμετρο *consistent* αποθηκεύεται η τιμή true αν το αρχικό σύστημα έχει λύση, ενώ η τιμή false αν δεν έχει λύση. Στην περίπτωση που κατά την εκτέλεση της συνάρτησης διαπιστωθεί ότι το αρχικό σύστημα δεν έχει λύση, τότε δεν εκτελούνται οι απλοποιήσεις Ad-Hoc και ακριβής, ανεξάρτητα ποιες είναι οι τιμές των ορισμάτων *Ad_Hoc* και *exact*. Τέλος, η ρουτίνα επιστρέφει μια ακολουθία πινάκων, κάθε ένας από τους οποίους εκφράζει τα όρια μιας από τις μεταβλητές του αρχικού συστήματος. Αν ο αρχικός πίνακας *Ab* έχει *n+1* στήλες, τότε επιστρέφει μια ακολουθία *n* πινάκων με *n+1, n, ..., 2* στήλες αντίστοιχα. Ο πρώτος από αυτούς εκφράζει τα όρια της *n*-οστής μεταβλητής του αρχικού συστήματος και ο τελευταίος τα όρια της πρώτης μεταβλητής. Σε κάθε πίνακα είναι τοποθετημένα πρώτα τα άνω όρια και έπειτα τα κάτω όρια. Δεν υπάρχουν γραμμές που να μην αναφέρονται στη συγκεκριμένη μεταβλητή, ανεξάρτητα αν έχει ζητηθεί η εφαρμογή απλοποίησης στους τελικούς πίνακες.

Επίσης, μπορούμε ισοδύναμα να καλέσουμε τη ρουτίνα

matrix *Fourier_Motzkin_Elimination(**const matrix**& A, **const matrix**& b,*
***bool**& consistent, **const bool** Ad_Hoc, **const bool** exact)*

Η διαφορά της από την προηγούμενη έγκειται στο ότι το αρχικό σύστημα δίδεται στη συνάρτηση με τη βοήθεια δύο πινάκων *A* και *b*. Ο πρώτος από αυτούς πρέπει να έχει *n* στήλες με τους συντελεστές των *n* αγνώστων του αρχικού συστήματος, ενώ ο δεύτερος πρέπει να έχει *1* στήλη με τους σταθερούς όρους του αρχικού συστήματος. Ο αριθμός των γραμμών των πινάκων ισούται με τον αριθμό των ανισοτήτων του συστήματος. Το σύστημα που περιγράφεται από τους *A* και *b* είναι το $A\vec{x} \le \vec{b}$. Επίσης, η συνάρτηση αυτή επιστρέφει όλα τα όρια των μεταβλητών σε έναν μόνο πίνακα, στον οποίο είναι τοποθετημένα πρώτα τα όρια των εξωτερικότερων και έπειτα τα όρια των εσωτερικότερων μεταβλητών. Για κάθε μεταβλητή είναι τοποθετημένα πρώτα τα άνω και έπειτα τα κάτω όρια.

Επίσης, στο αρχείο "*tiling.cc*" υλοποιήσαμε τη ρουτίνα

void *Column_Hermite_Normal_Form(**matrix**& H)*

Η ρουτίνα αυτή υπολογίζει την Κανονική Ερμητιανή Μορφή Στηλών του αρχικού πίνακα *H* και την αποθηκεύει στον πίνακα *H*. Ο αλγόριθμος που χρησιμοποιείται για τον υπολογισμό είναι αυτός που περιγράψαμε με ψευδοκώδικα στην παράγραφο 2.3. Αν ο αρχικός πίνακας *H* είναι μη τετραγωνικός ή μη αντιστρέψιμος, τότε το αποτέλεσμα θα είναι σύμφωνο με το γενικευμένο ορισμό 2.3.

Στο ίδιο αρχείο, κατασκευάσαμε τη ρουτίνα

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

void *tiling_inequalities(**const matrix**& H, **const matrix**& B, **const matrix**& b,*
const int *select,* ***const bool*** *simplify,*
matrix*& tiles,* ***matrix****& interior,* ***matrix****& transform,* ***matrix****& hermite,* ***int*** ***m)*

Η ρουτίνα αυτή είναι χρήσιμη για την παραγωγή του τελικού SPMD κώδικα.

Συγκεκριμένα, δέχεται ως παραμέτρους τους πίνακες B και b, διαστάσεων $d \times n$ και $d \times 1$, αντίστοιχα, οι οποίοι περιγράφουν τα όρια του αρχικού Iteration Space με τον τρόπο που περιγράψαμε στην παράγραφο 2.1, καθώς και τον πίνακα H, διαστάσεων $n \times n$, οποίος είναι ο tiling matrix του επιθυμητού μετασχηματισμού.

Επίσης, μέσω του ορίσματος *select* μπορούμε να επιλέξουμε τον τρόπο με τον οποίο θα υπολογιστούν τα όρια του Tile Space και τον τρόπο με τον οποίο θα διατρεχθεί το εσωτερικό κάθε tile. Μπορεί να πάρει τις τιμές:

OUT_EXACT_IN_ORTH:	Χρησιμοποιείται η ακριβής μέθοδος για την εύρεση των ορίων του Tile Space, σύμφωνα με την παράγραφο 3.1.1, και η τροποποιημένη μέθοδος ορθογώνιας σάρωσης, σύμφωνα με την παράγραφο 3.2.1, για το εσωτερικό κάθε tile.
OUT_EXACT_IN_HERM:	Χρησιμοποιείται η ακριβής μέθοδος για την εύρεση των ορίων του Tile Space, σύμφωνα με την παράγραφο 3.1.1, και η μέθοδος πλάγιας σάρωσης, σύμφωνα με την παράγραφο 3.2.2, για το εσωτερικό κάθε tile.
OUT_FAST_IN_ORTH:	Χρησιμοποιείται η γρήγορη μέθοδος για την εύρεση των ορίων του Tile Space, σύμφωνα με την παράγραφο 3.1.2, και η τροποποιημένη μέθοδος ορθογώνιας σάρωσης, σύμφωνα με την παράγραφο 3.2.1, για το εσωτερικό κάθε tile.
OUT_FAST_IN_HERM:	Χρησιμοποιείται η γρήγορη μέθοδος για την εύρεση των ορίων του Tile Space, σύμφωνα με την παράγραφο 3.1.2, και η μέθοδος πλάγιας σάρωσης, σύμφωνα με την παράγραφο 3.2.2, για το εσωτερικό κάθε tile.

Επίσης, το όρισμα *simplify* ορίζει αν θέλουμε να εφαρμοστεί η ακριβής απλοποίηση στους πίνακες που προκύπτουν με την εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin. Η απλοποίηση Ad-Hoc εφαρμόζεται πάντα.

Στον πίνακα *tiles* αποθηκεύονται οι ανισότητες που αφορούν τα όρια του Tile Space, όπως προκύπτουν με την εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin στο σύστημα ανισοτήτων *(3.1.1)* ή *(3.1.4),* ανάλογα με τη μέθοδο που επιλέχθηκε μέσω της μεταβλητής *select*. Στον πίνακα *interior* αποθηκεύονται τα όρια που προκύπτουν με την εφαρμογή της μεθόδου απαλοιφής Fourier-Motzkin στα συστήματα *(3.2.3)* ή *(3.2.6),* ανάλογα με τη μέθοδο σάρωσης των tiles που επιλέχθηκε.

Οι υπόλοιπες παράμετροι της ρουτίνας έχουν νόημα μόνο αν έχει επιλεγεί η μέθοδος πλάγιας σάρωσης των tiles. Συγκεκριμένα, στον πίνακα *transform* αποθηκεύεται ο πίνακας μετασχηματισμού P' από τον μετασχηματισμένο χώρο στον αρχικό. Στον πίνακα *hermite* αποθηκεύεται η ερμητιανή μορφή $\widetilde{H}'$ του πίνακα H', η οποία περιέχει τα βήματα και τις μετατοπίσεις των μεταβλητών j_k, που διατρέχουν το μετασχηματισμένο χώρο. Ο δείκτης *m* δείχνει σε ένα array ακεραίων, κάθε ένας από τους οποίους ισούται με το αντίστοιχο στοιχείο της διαγωνίου του πίνακα *M*. Οι ακέραιοι αυτοί είναι απαραίτητοι για τη γραφή των άνω ορίων των μεταβλητών j_k στον τελικό κώδικα SPMD.

Τέλος, στο ίδιο αρχείο τοποθετήσαμε τη ρουτίνα

void *print_tiling(**const matrix**& H**, const matrix**& B**, const matrix**& b,*
const int *mode**, const bool*** *simplify**, FILE** *outfp)*

Η ρουτίνα αυτή δέχεται ως παραμέτρους τον tiling matrix *H* και τους πίνακες *B* και *b*, οι οποίοι εκφράζουν τα όρια του αρχικού χώρου. Υποθέτουμε ότι τα όρια που εκφράζονται από τους πίνακες *B* και *b* είναι τοποθετημένα με τη σειρά που τοποθετούνται στον πίνακα που επιστρέφεται από τη ρουτίνα *Fourier_Motzkin_Elimination*. Δηλαδή, σε αυτούς είναι τοποθετημένα πρώτα τα όρια των πιο εξωτερικών δεικτών των βρόχων και έπειτα των πιο εσωτερικών δεικτών. Επίσης, είναι τοποθετημένα πρώτα τα άνω όρια κάθε δείκτη και έπειτα τα κάτω όρια αυτού. Η παράμετρος *mode* εκφράζει τον τρόπο με τον οποίο επιθυμούμε να γίνει η εύρεση των ορίων του Tile Space και η σάρωση του εσωτερικού των tiles, ακριβώς όπως η παράμετρος *select* της προηγούμενης ρουτίνας. Η παράμετρος *simplify* ορίζει αν επιθυμούμε να εφαρμοστεί η μέθοδος ακριβούς απλοποίησης από τη μέθοδο απαλοιφής Fourier-Motzkin, όπου αυτή καλείται. Η ρουτίνα γράφει τον ψευδοκώδικα που παράγεται με βάση τα δεδομένα εισόδου στο αρχείο *outfp*, το οποίο υποθέτουμε ότι έχει ανοιχθεί για γράψιμο πριν την κλήση της ρουτίνας.

Η ρουτίνα *print_tiling* στην αρχή της εκτέλεσής της καλεί τη ρουτίνα *tiling_inequalities* και στα αποτελέσματα αυτής στηρίζεται για τη γραφή του ψευδοκώδικα. Τόσο για τη μέθοδο ορθογώνιας σάρωσης των tiles, όσο και για τη μέθοδο της πλάγιας σάρωσης χρησιμοποιεί τις τροποποιημένες μεθόδους παραγωγής των τελικών ορίων, όπως παρουσιάστηκαν στις παραγράφους 3.2.1 και 3.2.2, διαχωρίζοντας τα tiles σε εξωτερικά και εσωτερικά. Στην περίπτωση της μεθόδου πλάγιας σάρωσης των tiles, ο κώδικας που παράγεται είναι σύμφωνος με το Θεώρημα 3.2.

Τα αρχεία με όλες τις παραπάνω συναρτήσεις έχουν παρατεθεί στο παράρτημα 2, στο τέλος της παρούσας εργασίας.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

4.2 Παράδειγμα

Στη συνέχεια, εκτελέσαμε την τελευταία συνάρτηση για διάφορες τιμές της παραμέτρου *mode* και με *simplify=true*, μετρώντας ταυτόχρονα τον αριθμό των γραμμοπράξεων που εκτελούνται από τη συνάρτηση *Fourier_Motzkin_Elimination*, και καταγράφοντας τους ενδιάμεσους πίνακες που παράγονται.

Ο πίνακας του μετασχηματισμού tiling που χρησιμοποιήσαμε και τον οποίο περάσαμε ως παράμετρο στη συνάρτηση *print_tiling* και μέσω αυτής στην *tiling_inequalities* ήταν ο:

$$H = \frac{1}{32}\begin{bmatrix} 5 & 2 \\ \hline -1 & 6 \end{bmatrix}.$$

Ισοδύναμα, οι πλευρές των tiles στα οποία χωρίσαμε το επίπεδο των επαναλήψεων εκφράζονται από τις στήλες του πίνακα:

$$P = \left[\begin{array}{c|c} 6 & -2 \\ 1 & 5 \end{array}\right].$$

Επίσης, ως παραμέτρους στη συνάρτηση *print_tiling* περάσαμε τους πίνακες:

$$B = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -2 & 5 \\ -1 & -3 \end{bmatrix} \text{ και } b = \begin{bmatrix} 21 \\ 0 \\ 155 \\ 0 \end{bmatrix},$$

οι οποίοι εκφράζουν τα όρια του χώρου των επαναλήψεων. Επομένως, τα όρια και ο χωρισμός του χώρου I^2 σε tiles μπορεί να αναπαρασταθεί όπως στο σχήμα 4.1. Στο ίδιο σχήμα έχουμε σημειώσει τις συντεταγμένες του κάθε tile στο Tile Space.

Στην περίπτωση που το όρισμα *mode* έχει την τιμή *OUT_EXACT_IN_ORTH,* για την εύρεση των ορίων του Tile Space, η ρουτίνα *tiling_inequalities* εφαρμόζει τη μέθοδο Fourier-Motzkin στο σύστημα ανισοτήτων:

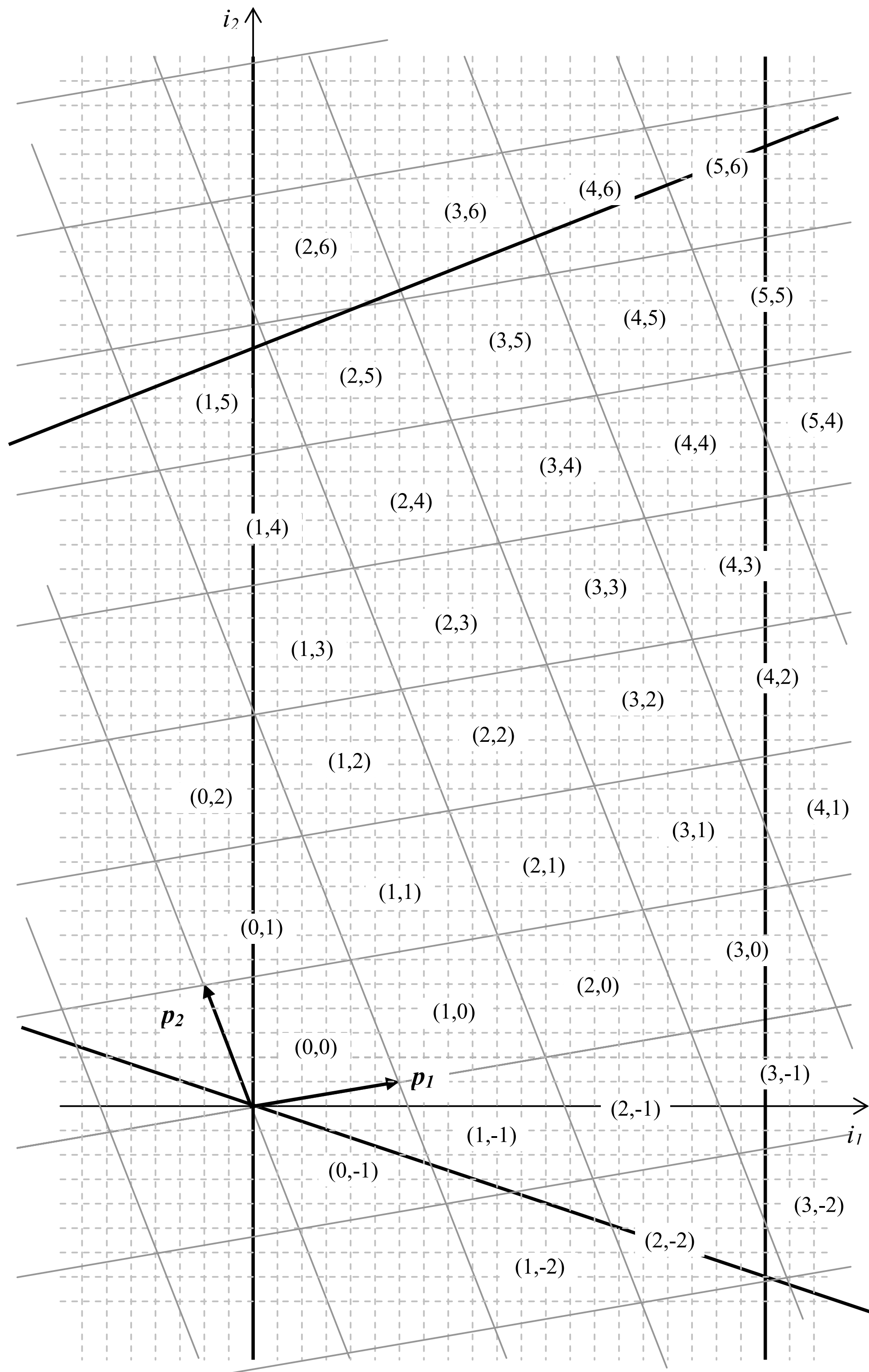
$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -2 & 5 \\ 0 & 0 & -1 & -3 \\ -32 & 0 & 5 & 2 \\ 0 & -32 & -1 & 6 \\ 32 & 0 & -5 & -2 \\ 0 & 32 & 1 & -6 \end{pmatrix} \cdot \vec{x} \le \begin{pmatrix} 21 \\ 0 \\ 155 \\ 0 \\ 31 \\ 31 \\ 0 \\ 0 \end{pmatrix}$$

σύμφωνα με τον τύπο *(3.1.1)*. Ο επαυξημένος πίνακας που προκύπτει από την επίλυση του προηγούμενου συστήματος, έπειτα από 96.070 γραμμοπράξεις, είναι ο εξής:

$$\left(\begin{array}{cccc|c}
160 & 0 & 0 & 0 & 919 \\
-32 & 0 & 0 & 0 & 31 \\
-224 & 928 & 0 & 0 & 5177 \\
-96 & 32 & 0 & 0 & 93 \\
-144 & -208 & 0 & 0 & 341 \\
0 & -16 & 0 & 0 & 47 \\
96 & -32 & 0 & 0 & 367 \\
-96 & 0 & 13 & 0 & 93 \\
-96 & 32 & 16 & 0 & 93 \\
0 & 0 & 1 & 0 & 21 \\
160 & 0 & -29 & 0 & 310 \\
0 & 160 & -7 & 0 & 930 \\
0 & -32 & -3 & 0 & 31 \\
96 & -32 & -16 & 0 & 31 \\
0 & 0 & -1 & 0 & 0 \\
0 & 0 & -2 & 5 & 155 \\
-32 & 0 & 5 & 2 & 31 \\
0 & -32 & -1 & 6 & 31 \\
0 & 0 & -1 & -3 & 0 \\
32 & 0 & -5 & -2 & 0 \\
0 & 32 & 1 & -6 & 0
\end{array}\right)$$

Από αυτόν επιλέγονται και τοποθετούνται στον πίνακα *tiles* οι 7 πρώτες γραμμές που εκφράζουν τα όρια των 2 εξωτερικότερων μεταβλητών t_1, t_2. Επομένως, ο πίνακας που παίρνουμε τελικά μέσω της παραμέτρου *tiles* της ρουτίνας *tiling_inequalities* είναι ο εξής:

$$\left(\begin{array}{cc|c}
160 & 0 & 919 \\
-32 & 0 & 31 \\
-224 & 928 & 5177 \\
-96 & 32 & 93 \\
-144 & -208 & 341 \\
0 & -16 & 47 \\
96 & -32 & 367
\end{array}\right)$$



Σχήμα 4.1

Σύμφωνα με τις ανισότητες που εκφράζονται από τον πίνακα αυτό, οι μεταβλητές t_1, t_2 κινούνται μεταξύ των ορίων:

$$0 \le t_1 \le 5$$
$$\max(\lceil(-341-144t_1)/208\rceil, -2, \lceil(-367+96t_1)/32\rceil) \le t_2 \le \min(\lfloor(5177+224t_1)/928\rfloor, \lfloor(93+96t_1)/32\rfloor)$$

Επομένως από τον τελικό κώδικα θα διατρεχθούν τα tiles: (0,-1),…,(0,2), (1,-2),…,(1,5), (2,-2),…,(2,6), (3,-2),…,(3,6), (4,1),…,(4,6), (5,4),…,(5,6). Από το σχήμα 4.1 μπορεί κανείς εύκολα να επαληθεύσει την ορθότητα των παραπάνω ορίων.

Όσον αφορά τη σάρωση του εσωτερικού των tiles, χρησιμοποιείται η τροποποιημένη μέθοδος της παραγράφου 3.2.1. Για τη γραφή των εσωτερικών ορίων χρησιμοποιούνται οι σχέσεις $B \cdot i \le \vec{b}$ *(3.2.2)* και $\begin{pmatrix} gH \\ -gH \end{pmatrix} \cdot (i - i_{toi}) \le \begin{pmatrix} \overrightarrow{(g-1)} \\ \vec{0} \end{pmatrix}$ *(3.2.3).* Οι πίνακες $B = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -2 & 5 \\ -1 & -3 \end{bmatrix}$ και $b = \begin{bmatrix} 21 \\ 0 \\ 155 \\ 0 \end{bmatrix}$, της σχέσης *(3.2.2)* έχουν ήδη δοθεί στην κατάλληλη μορφή, επομένως δεν χρειάζεται να εφαρμοστεί σε αυτούς η μέθοδος Fourier-Motzkin. Όσον αφορά το σύστημα *(3.2.3),* στο παράδειγμά μας καλείται από τη ρουτίνα *tiling_inequalities* η μέθοδος απαλοιφής Fourier-Motzkin για το σύστημα ανισοτήτων:

$$\begin{pmatrix} 5 & 2 \\ -1 & 6 \\ -5 & -2 \\ 1 & -6 \end{pmatrix} \cdot \vec{x} \le \begin{pmatrix} 31 \\ 31 \\ 0 \\ 0 \end{pmatrix}$$

Ο επαυξημένος πίνακας που προκύπτει μετά από 58 γραμμοπράξεις και ο οποίος επιστρέφεται από τη ρουτίνα *tiling_inequalities* μέσω της παραμέτρου *interior* είναι ο εξής:

$$\left(\begin{array}{cc|c} 16 & 0 & 93 \\ -16 & 0 & 31 \\ \hline 5 & 2 & 31 \\ -1 & 6 & 31 \\ -5 & -2 & 0 \\ 1 & -6 & 0 \end{array}\right)$$

Με βάση τα παραπάνω, η ρουτίνα *print_tiling* τυπώνει στο αρχείο εξόδου τον εξής κώδικα:

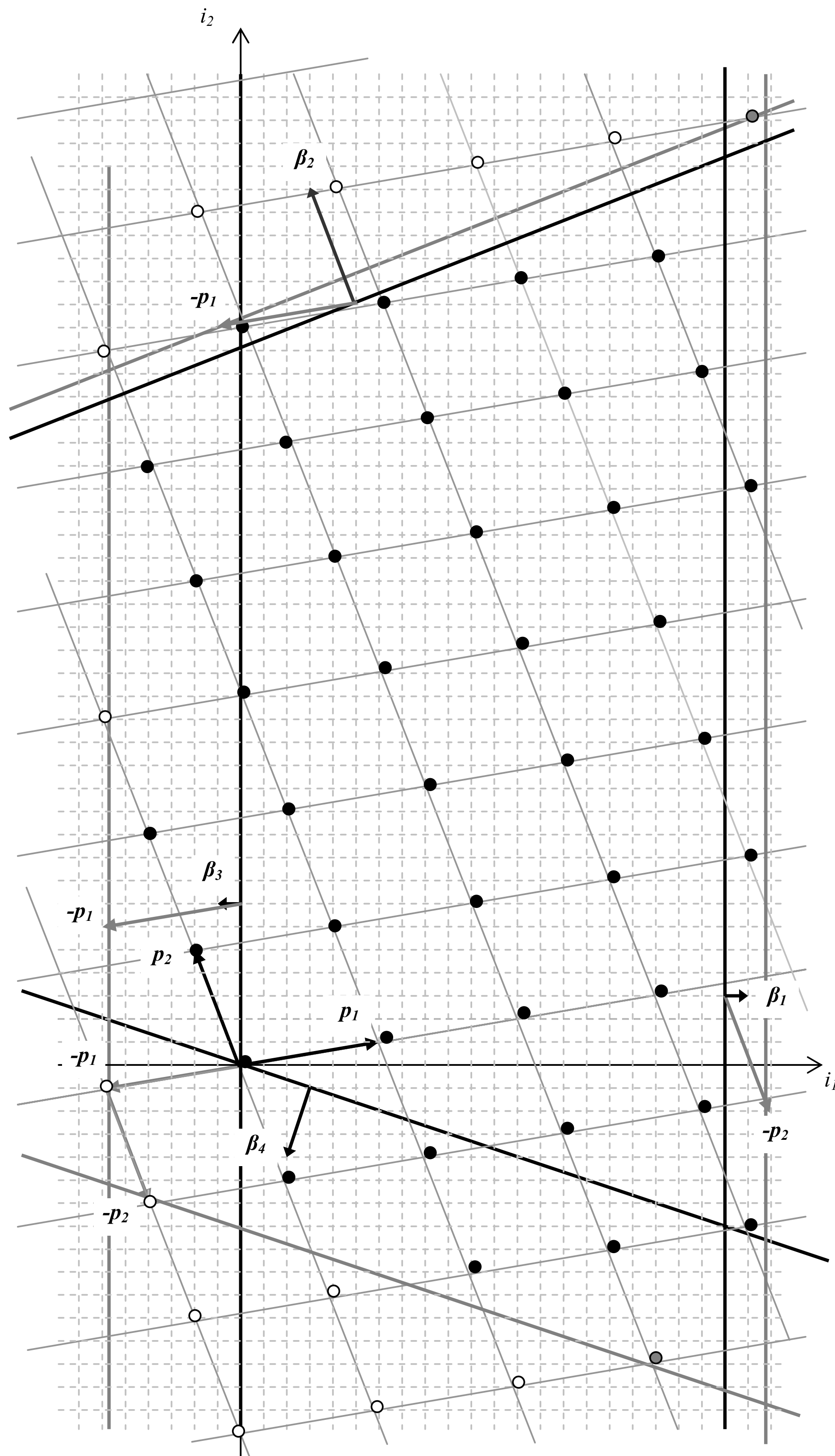
```
For tiles that may cross the Iteration Space bounds:

for(t1=max(ceil((-31)/32)); t1<=min(floor((919)/160)); t1++)
  for(t2=max(ceil((-341-144*t1)/208),ceil((-47+0*t1)/16),ceil((-367+96*t1)/32));
t2<=min(floor((5177+224*t1)/928),floor((93+96*t1)/32)); t2++)
  {
    toi1=6*t1-2*t2;
    toi2=1*t1+5*t2;
    for(i1=max(ceil((0)/1),toi1+ceil((-31)/16));
i1<=min(floor((21)/1),toi1+floor((93)/16)); i1++)
      for(i2=max(ceil((0-1*i1)/3),toi2+ceil((0-5*i1)/2),toi2+ceil((0+1*i1)/6));
i2<=min(floor((155+2*i1)/5),toi2+floor((31-5*i1)/2),toi2+floor((31+1*i1)/6)); i2++)
      {
        ... ... ...
      }
  }

For tiles that do not cross the Iteration Space bounds:

for(t1=max(ceil((-31)/32)); t1<=min(floor((919)/160)); t1++)
  for(t2=max(ceil((-341-144*t1)/208),ceil((-47+0*t1)/16),ceil((-367+96*t1)/32));
t2<=min(floor((5177+224*t1)/928),floor((93+96*t1)/32)); t2++)
  {
    toi1=6*t1-2*t2;
    toi2=1*t1+5*t2;
    for(i1=toi1+max(ceil((-31)/16)); i1<=toi1+min(floor((93)/16)); i1++)
      for(i2=toi2+max(ceil((0-5*i1)/2),ceil((0+1*i1)/6)); i2<=toi2+min(floor((31-
5*i1)/2),floor((31+1*i1)/6)); i2++)
      {
        ... ... ...
      }
  }
```

Στην περίπτωση που το όρισμα *mode* έχει την τιμή *OUT_FAST_IN_HERM,* τότε για την εύρεση των ορίων του Tile Space, πρέπει σύμφωνα με τη σχέση *(3.1.4)* να εφαρμοστεί η μέθοδος απαλοιφής Fourier-Motzkin στο σύστημα ανισοτήτων:

$$\left(\begin{array}{cc|c} 6 & -2 & \frac{367}{16} \\ -6 & 2 & \frac{93}{16} \\ -7 & 29 & \frac{5177}{32} \\ -9 & -13 & \frac{341}{16} \end{array}\right)$$

Μετά από 63 γραμμοπράξεις, παράγεται από τη ρουτίνα *tiling_inequalities* και επιστρέφεται στη *print_tiling* μέσω του πίνακα *tiles* ο εξής επαυξημένος πίνακας:

$$\left(\begin{array}{cc|c} 128 & 0 & 791 \\ -1536 & 0 & 1891 \\ \hline -96 & 32 & 93 \\ -224 & 928 & 5177 \\ 96 & -32 & 367 \\ -144 & -208 & 341 \end{array}\right)$$



Σχήμα 4.2

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

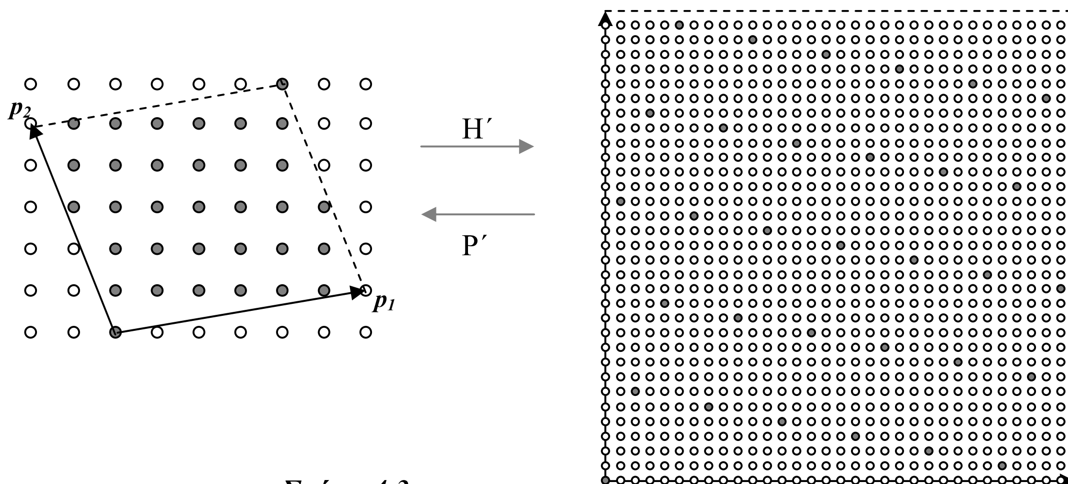
Σύμφωνα με τις ανισότητες που εκφράζονται από τον πίνακα αυτό, οι μεταβλητές t_1, t_2 κινούνται μεταξύ των ορίων:

$$-1 \leq t_1 \leq 6$$
$$\max(\lceil(-367+96t_1)/32\rceil, \lceil(-341-144t_1)/208\rceil) \leq t_2 \leq \min(\lfloor(93+96t_1)/32\rfloor, \lfloor(5177+224t_1)/928\rfloor)$$

Επομένως από τον τελικό κώδικα θα διατρεχθούν τα tiles: : (0,-1),…,(0,2), (1,-2),…,(1,5), (2,-3),…,(2,6), (3,-2),…,(3,6), (4,1),…,(4,6), (5,4),…,(5,6), (6,7). Στο σχήμα 4.2 έχουμε χρωματίσει με μαύρο τα σημεία εκκίνησης των tiles που ανήκουν πράγματι στο Tile Space και με γκρι τα σημεία εκκίνησης των tiles που κανονικά δεν ανήκουν στο Tile Space, αλλά η γρήγορη μέθοδος τα περιλαμβάνει.

Παρατηρούμε ότι στο συγκεκριμένο παράδειγμα περιλαμβάνονται δύο περιττά tiles, τα (2,-3) και (6,7). Επίσης παρατηρούμε ότι ενώ ο δείκτης t_1, σύμφωνα με τις τελικές ανισότητες μπορεί να πάρει την τιμή –1, τελικά δεν διατρέχεται κανένα tile με τέτοια συντεταγμένη. Αυτό συμβαίνει επειδή για την τιμή αυτή του t_1=-1 υπάρχουν πραγματικές τιμές του t_2 που ικανοποιούν τις 4 τελευταίες ανισότητες, αλλά δεν υπάρχουν ακέραιες τιμές του t_2 που να τις ικανοποιούν.

Για τη σάρωση του εσωτερικού των tiles χρησιμοποιείται η πλάγια μέθοδος, όπως περιγράφεται στην παράγραφο 3.2.2.



Σχήμα 4.3

Επομένως, η ρουτίνα *tiling_inequalities*, που καλείται από την *print_tiling* υπολογίζει τους ακέραιους αριθμούς με τους οποίους πρέπει να πολλαπλασιαστούν οι γραμμές του πίνακα *H* ώστε να προκύψει ακέραιος πίνακας. Οι αριθμοί αυτοί είναι οι 32, 32, επιστρέφονται από την *tiling_inequalities* στην *print_tiling* μέσω του array *m* και αργότερα θα χρησιμεύσουν για τη γραφή των άνω ορίων των 2 πιο εσωτερικών βρόχων του τελικού κώδικα, σύμφωνα με τη σχέση *(3.2.4)*.

Με βάση τις τιμές αυτές υπολογίζεται από την *tiling_inequalities* ο πίνακας μετασχηματισμού $H' = \begin{pmatrix} 5 & 2 \\ -1 & 6 \end{pmatrix}$ και ο πίνακας αντίστροφου μετασχηματισμού $P' = \begin{pmatrix} \frac{3}{16} & -\frac{1}{16} \\ \frac{1}{32} & \frac{5}{32} \end{pmatrix}$, ο οποίος επιστρέφεται στην καλούσα συνάρτηση μέσω της παραμέτρου *transform*. Στο σχήμα 4.3 απεικονίζεται γραφικά ο μετασχηματισμός ενός tile με βάση τους πίνακες αυτούς.

Η ερμητιανή μορφή του πίνακα H' υπολογίζεται από την *tiling_inequalities* και αποθηκεύεται στον πίνακα *hermite*: $\widetilde{H}' = \begin{pmatrix} 1 & 0 \\ 19 & 32 \end{pmatrix}$. Στο σχήμα 4.3 παρατηρούμε ότι πράγματι οι τιμές 1 και 32 είναι κατάλληλες για να χρησιμοποιηθούν ως βήματα των δεικτών των μετασχηματισμένων βρόχων, ενώ η τιμή 19 είναι η κατάλληλη μετατόπιση του εσωτερικού δείκτη j_2 όταν ο j_1 αυξάνεται κατά 1.

Τέλος, προκειμένου οι εσωτερικοί δείκτες j_1, j_2 να μη βγαίνουν έξω από τα αρχικά όρια του Iteration Space, η ρουτίνα *tiling_inequalities* επιλύει το σύστημα της μορφής *(3.2.6):*

$$\begin{pmatrix} \frac{3}{16} & -\frac{1}{16} \\ -\frac{3}{16} & \frac{1}{16} \\ -\frac{7}{32} & \frac{29}{32} \\ -\frac{9}{32} & -\frac{13}{32} \end{pmatrix} \cdot \vec{x} \le \begin{pmatrix} 21 \\ 0 \\ 155 \\ 0 \end{pmatrix}$$

Ο επαυξημένος πίνακας που προκύπτει μετά από 63 γραμμοπράξεις και ο οποίος αποθηκεύεται στην παράμετρο *interior* είναι ο εξής:

$$\left(\begin{array}{cc|c} 5 & 0 & 919 \\ -1 & 0 & 0 \\ \hline -3 & 1 & 0 \\ -7 & 29 & 4960 \\ 3 & -1 & 336 \\ -9 & -13 & 0 \end{array}\right)$$

Με βάση τα παραπάνω, η ρουτίνα *print_tiling* τυπώνει στο αρχείο εξόδου τον εξής κώδικα:

```
For tiles that may cross the Iteration Space bounds:

for(t1=max(ceil((-1891)/1536)); t1<=min(floor((791)/128)); t1++)
  for(t2=max(ceil((-367+96*t1)/32),ceil((-341-144*t1)/208));
t2<=min(floor((93+96*t1)/32),floor((5177+224*t1)/928)); t2++)
  {
    toj1=32*t1;
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
    toj2=32*t2;
    lb1=max(0,ceil((0)/1)-toj1);
    ub1=min(31,floor((919)/5)-toj1);
    for(j1=ceil(lb1/1)*1,j2=ceil(lb1/1)*19;j1<=ub1;j1+=1,j2+=19)
    {
      lb2=max(0,ceil((-336+3*(j1+toj1))/1)-toj2,ceil((0-9*(j1+toj1))/13)-toj2);
      ub2=min(31,floor((0+3*(j1+toj1))/1)-toj2,floor((4960+7*(j1+toj1))/29)-toj2);
      for(j2+=ceil((lb2-j2)/32)*32;j2<=ub2;j2+=32)
      {
        i1=3/16*(toj1+j1)-1/16*(toj2+j2);
        i2=1/32*(toj1+j1)+5/32*(toj2+j2);
        ... ... ...
      }
    }
  }

For tiles that do not cross the Iteration Space bounds:

for(t1=max(ceil((-1891)/1536)); t1<=min(floor((791)/128)); t1++)
  for(t2=max(ceil((-367+96*t1)/32),ceil((-341-144*t1)/208));
t2<=min(floor((93+96*t1)/32),floor((5177+224*t1)/928)); t2++)
  {
    toj1=32*t1;
    toj2=32*t2;
    for(j1=0,j2=0;j1<=31;j1+=1,j2+=19)
      for(j2+=ceil((-j2)/32)*32;j2<=31;j2+=32)
      {
        i1=3/16*(toj1+j1)-1/16*(toj2+j2);
        i2=1/32*(toj1+j1)+5/32*(toj2+j2);
        ... ... ...
      }
  }
```

4.3 Συμπεράσματα

Στην παρούσα διπλωματική εργασία παρουσιάσαμε μία επισκόπηση των μεθόδων που έχουν προταθεί στη βιβλιογραφία για την παραγωγή SPMD κώδικα με τη βοήθεια ενός μετασχηματισμού tiling και στη συνέχεια προτείναμε μία νέα μέθοδο, η οποία μπορεί να εφαρμοστεί σε γενικά παραλληλεπίπεδα tiles και σε οποιοδήποτε κυρτό χώρο επαναλήψεων.

Προκειμένου να επιτύχουμε το σκοπό μας χωρίσαμε το αρχικό πρόβλημα σε δύο υπο-προβλήματα: την απαρίθμηση των tiles και τη σάρωση του εσωτερικού κάθε tile. Στην πρώτη περίπτωση προτείναμε την εύρεση των ορίων του Tile Space με τη βοήθεια μιας κατάλληλης επέκτασης του αρχικού χώρου επαναλήψεων. Στη δεύτερη περίπτωση προτείναμε έναν non-unimodular μετασχηματισμό ώστε το εσωτερικό κάθε tile να μετατραπεί σε ορθογώνιο χώρο.

Έπειτα υλοποιήσαμε τις παραπάνω μεθόδους και με τη βοήθεια αρκετών παραδειγμάτων συγκρίναμε τα αποτελέσματα των προϋπάρχουσων και των προτεινόμενων στην παρούσα εργασία μεθόδων.

Από τα παραδείγματα αυτά συμπεράναμε ότι το σημαντικότερο πλεονέκτημα της μεθόδου που προτάθηκε για την απαρίθμηση των tiles είναι το πολύ μικρό κόστος υπολογισμού σε σχέση με το κόστος υπολογισμού της προϋπάρχουσας, και συνεπώς η ταχύτερη εκτέλεση. Όσον αφορά τη μέθοδο που προτάθηκε για τη σάρωση του

εσωτερικού των tiles, δίδει πολύ πιο απλή μορφή για τα όρια των tiles, ιδιαίτερα αυτών που δεν τέμνονται από κάποιο όριο του χώρου επαναλήψεων. Επίσης, είναι πολύ πιο εύκολη η εύρεση των δεδομένων που πρέπει να σταλούν από τον έναν επεξεργαστή στον άλλο.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Παράρτημα 1: Τροποποιημένος κώδικας κλάσεων που χρησιμοποιήθηκαν

Αρχείο "rational.h"

```
/* definition of class rational which represents rational numbers*/

#include <stdio.h>

class rational{

 protected:
  int numer;
  int denomer;

 public:
  rational();
  rational(const int i);
  rational(const int i,const int j);
  ~rational();
  int gcd(const int i,const int j);
  void printr() const;
  void fprintr(FILE *fp) const;
  double rational2double();
  bool isint();
  rational operator - ();
  rational operator + (rational ob1);
  rational operator - (rational ob1);
  rational operator * (rational ob1);
  rational operator / (rational ob1);
  rational operator = (rational ob1);
  rational operator = (int i);
  bool operator == (int i);
  bool operator == (rational ob1);
  bool operator < (int i);
  bool operator < (rational ob1);
  bool operator > (int i);
  bool operator > (rational ob1);
  bool operator <= (int i);
  bool operator <= (rational ob1);
  bool operator >= (int i);
```

```
  bool operator >= (rational ob1);
  bool operator != (int i);
  bool operator != (rational ob1);

  void floorint();
  void ceilingint();
  int get_numer() const;
  int get_denomer() const;
};
```

Αρχείο "rational.cc"

```
#include "rational.h"
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>

#define RATIONAL_SAPEIRON = rational(1,0)
#define RATIONAL_MAPEIRON = rational(-1,0)
#define RATIONAL_UNDEF = rational(0,0)


rational::rational()
{
}
rational::rational(const int i)
{
 numer=i;
 denomer=1;
}

rational::rational(const int i,const int j)
{
 numer=i/gcd(i,j);
 denomer=j/gcd(i,j);
 if(denomer<0){numer=-numer;denomer=-denomer;}
}

rational::~rational()
{
}

int rational::gcd(const int i,const int j)
{
 if((i==0)&&(j==0))return 1;
 if (i==0) return abs(j);
 if (j==0) return abs(i);
 int x=abs(i);
 int y=abs(j);
 while(x!=y)
  if(x<y)y=y-x;
  else x=x-y;
 return x;
}

void rational::printr() const
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
{
 if(denomer==0)
 {
      if (numer==0) std::cout<<"UNDF";
      else if (numer>0) std::cout<<" +OO";
      else std::cout<<" -OO";
 }
 else if (numer==0)printf("%4d",numer);
 else if(denomer!=1) printf("%2d/%1d",numer,denomer);
 else printf("%4d",numer);
}

void rational::fprintr(FILE *fp) const
{
 if(denomer==0)
 {
      if (numer==0) fprintf(fp,"UNDF");
      else if (numer>0) fprintf(fp,"+OO");
      else fprintf(fp,"-OO");
 }
 else if (numer==0) fprintf(fp,"%d",numer);
 else if(denomer!=1) fprintf(fp,"%d/%d",numer,denomer);
 else fprintf(fp,"%d",numer);
}

double rational::rational2double()
{
 if (denomer==0)
 {
 //Warning!! An error may have occurred.
      return 0;
 }
 else
 {
      double x;
      x=(double) numer/denomer;
      return x;
 }
}

bool rational::isint()
{
 if ((denomer==1)||(numer==0))return true;
 else return false;
}

rational rational::operator-()
{
      rational temp=rational(-numer, denomer);
      return temp;
}

rational rational::operator+(rational ob1)
{
 rational temp;
 int h;

 temp.numer=numer*ob1.denomer+ob1.numer*denomer;
 temp.denomer=denomer*ob1.denomer;
```

```
 h=gcd(temp.numer,temp.denomer);
 temp.numer/=h;
 temp.denomer/=h;

// if(temp.numer==0)temp.denomer=1;

 return temp;
}

rational rational::operator-(rational ob1)
{
 rational temp;
 int h;

 temp.numer=numer*ob1.denomer-ob1.numer*denomer;
 temp.denomer=denomer*ob1.denomer;

 h=gcd(temp.numer,temp.denomer);
 temp.numer/=h;
 temp.denomer/=h;

 return temp;
}

rational rational::operator*(rational ob1)
{
 rational temp;
 int h;

 temp.numer=numer*ob1.numer;
 temp.denomer=denomer*ob1.denomer;

 h=gcd(temp.numer,temp.denomer);
 temp.numer/=h;
 temp.denomer/=h;

 return temp;
}

rational rational::operator/(rational ob1)
{
      rational temp;
      int h;

      temp.numer=numer*ob1.denomer;
      temp.denomer=denomer*ob1.numer;

      h=gcd(temp.numer,temp.denomer);
      temp.numer/=h;
      temp.denomer/=h;

      if(temp.denomer<0){temp.numer=-temp.numer;temp.denomer=-
temp.denomer;}
      return temp;
}

rational rational::operator=(rational ob1)
{
 numer=ob1.numer;
 denomer=ob1.denomer;
```

```
 return *this;

}

rational rational::operator=(int i)
{
 numer=i;
 denomer=1;

 return *this;
}

bool rational::operator==(int i)
{
// int h=i*denomer;
// if (numer==h) return true;
// else return false;
  return (numer==(i*denomer));
}

/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator==(rational ob1)
{
 //if((numer==ob1.numer)&&(denomer==ob1.denomer))return true;
 //else return false;
 return ((numer*ob1.denomer)==(denomer*ob1.numer));
}

bool rational::operator<(int i)
{
 //int h=i*denomer;
 //if (numer<h) return true;
 //else return false;
 return (numer<(i*denomer));
}

/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator<(rational ob1)
{
 //float x1=(float)numer/denomer;
 //float x2=(float)ob1.numer/denomer;
 //if(x1>x2)return true;
 //else return false;
 return ((numer*ob1.denomer)<(denomer*ob1.numer));
}

bool rational::operator>(int i)
{
 //int h=i*denomer;
 //if (numer>h) return true;
 //else return false;
 return (numer>(i*denomer));
}
```

```
/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator>(rational ob1)
{
 //float x1=(float)numer/denomer;
 //float x2=(float)ob1.numer/denomer;
 //if(x1>x2)return true;
 //else return false;
 return ((numer*ob1.denomer)>(denomer*ob1.numer));
}

bool rational::operator<=(int i)
{
 //int h=i*denomer;
 //if (numer<=h) return true;
 //else return false;
 return (numer<=(i*denomer));
}

/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator<=(rational ob1)
{
 //float x1=(float)numer/denomer;
 //float x2=(float)ob1.numer/denomer;
 //if(x1<=x2)return true;
 //else return false;
 return ((numer*ob1.denomer)<=(denomer*ob1.numer));
}

bool rational::operator!=(int i)
{
 //int h=i*denomer;
 //if (numer!=h) return true;
 //else return false;
 return (numer!=(i*denomer));
}

/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator!=(rational ob1)
{
 //if((numer==ob1.numer)&&(numer==0))return false;
 //if((numer!=ob1.numer)||(denomer!=ob1.denomer))return true;
 //else return false;
 return ((numer*ob1.denomer)!=(denomer*ob1.numer));
}

bool rational::operator>=(int i)
{
 //int h=i*denomer;
 //if (numer>=h) return true;
 //else return false;
```

```
 return (numer>=(i*denomer));
}

/*
      This function works correctly if at most one
      of the compared numbers is non-rational (apeiro)
*/
bool rational::operator>=(rational ob1)
{
 //float x1=(float)numer/denomer;
 //float x2=(float)ob1.numer/denomer;
 //if(x1>=x2)return true;
 //else return false;
 return ((numer*ob1.denomer)>=(denomer*ob1.numer));
}

//
// It returns a rational number which is equal
// to the maximum integer
// which is less or equal to the given rational.
//
void rational::floorint()
{
      int tempnumer=abs(numer)/denomer;
      bool ypol=abs(numer)%denomer;

      denomer=1;
      if (numer>=0)
      {
            numer=tempnumer;
      }
      else
      {
            if(ypol) numer=-tempnumer-1;
            else numer=-tempnumer;
      }
}

//
// It returns a rational number which is equal
// to the minimum integer
// which is greater or equal to the given rational.
//
void rational::ceilingint()
{
      int tempnumer=abs(numer)/denomer;
      bool ypol=abs(numer)%denomer;

      denomer=1;
      if (numer>=0)
      {
            if(ypol) numer=1+ tempnumer;
            else numer=tempnumer;
      }
      else
      {
            numer=-tempnumer;
      }
}
```

```
/*
      It returns the numerator of the rational
*/
int rational::get_numer() const
{
      return numer;
}

/*
      It returns the denumerator of the rational
*/
int rational::get_denomer() const
{
      return denomer;
}
```

Αρχείο “matrix.h”

```
/*
  definition of class matrix which represents a 2-dimensional
  matrix containing rational numbers
*/

//#include <stdio.h>
#include "rational.h"
#include "definitions.h"

#define IDENTITY 1
#define SQUARE 2
#define CONTDIAG 3
#define TWODIMENSIONAL 4
#define FULLROWRANK 5

class matrix{

 protected:
  int rows;
  int columns;
  rational **Matrix;

 public:

  matrix();
  matrix(int r,int c);
  matrix(FILE *fp);
  void newdimensions(const int r, const int c);
  matrix(const matrix& m);
  matrix(const matrix& m,const matrix& n);
  matrix(int r,int c,int **A);
  ~matrix();
  int get_rows() const;
  int get_cols() const;
  void matrix_print() const;
  void printresult(double x);
  bool linearindependence();
  bool isint();
  matrix findnormalvector();
```

```
int diagonizematrix(rational r);
rational det();
matrix inverse() const;
matrix diagonalsum();
void optimaltile(int grain);
int multiply(const int n);
matrix operator + (const matrix ob1) const;
matrix operator - (const matrix ob1) const;
matrix operator * (const matrix ob1) const;
matrix operator = (const matrix ob1);

friend int addcheck(matrix m1,matrix m2);
friend int multcheck(matrix m1,matrix m2);
friend int matrixscan(matrix m1);

friend void exchange_rows(matrix& Ab,
                                  const int r1, const int r2);
friend void fourier_sort(matrix& Ab, int& p, int& q);
friend matrix fourier_new(const matrix& prev,
                                  const int p, const int q);
friend void fourier_next(const matrix& prev, matrix& next,
                                  const int p, const int q);
friend bool air_next(const matrix& prev,
                                  const int p, const int q);
friend matrix* Fourier_Motzkin_Elimination(const matrix& Ab,
                      bool& consistent,
                      const bool Ad_Hoc, const bool exact);
friend void row_simplify(matrix & Q);
friend void row_elimination(matrix& A, const int r);
friend void row_elimination(matrix& A, const int r,
                                  int& p, int& q);
friend void Ad_Hoc_Simplification (matrix *Ab,
                                  int *p, int *q);
friend void remove_zero_rows(matrix *A, const int *q);
friend void row_negate(matrix& N, const int r);
friend matrix first_tested(const matrix& A, const int q);
friend matrix add_test_matrix(const matrix& prev,
                            const matrix& next, const int q);
friend void exact_simplification(matrix* A,
                            int *p, int *q);
friend matrix array_to_1matrix(const matrix *arr);
friend matrix Fourier_Motzkin_Elimination(const matrix& A,
                const matrix& b, bool& consistent,
                const bool Ad_Hoc, const bool exact);
friend int row_denomer_lcm(const matrix& A,const int row);
friend int matrix_denomer_lcm(const matrix& A);
friend void upgrade_H(const matrix& H,
                                        matrix& A, matrix& b);
friend void upgrade_H(const matrix& H,const int g,
                                        matrix& A, matrix& b);

friend void Column_Hermite_Normal_Form(matrix& H);
friend void Column_GCD(matrix& A,int row,int x, int col);
friend void easy_extern_bounds(const matrix& B,
          const matrix& P,const matrix& b,matrix& result,
          const bool simplify,const int g);
friend void exact_extern_bounds(const matrix& B,
          const matrix& b, const matrix& H,const int g,
          matrix& result,const bool simplify);
friend void space_transform(const matrix& H, matrix& trans,
```

```
                                                   int **diag);
  friend void print_tiling(const matrix& H,
            const matrix& B, const matrix& b,
            const int mode, const bool simplify, FILE * fp);
};
```

Αρχείο "matrix.cc"

```
#include <iostream.h>
#include <new.h>
//#include <stdio.h>
#include "matrix.h"
#include "combinations.h"
#include <errno.h>
#include <math.h>
//#include <stdlib.h>
//#include <unistd.h>

extern int comb[MAXDEPNUM][MAXDIM];
extern int count;

matrix::matrix()
{
      rows=0;
      columns=0;
}

/*
  Dhmiourgia antikeimenou matrix me r grammes kai c sthles
  Ta stoixeia ths diagwniou arxikopoiountai me 1, ola ta
  upoloipa me 0
*/
matrix::matrix(int r,int c)
{
      int i,j;
      rows=r;
      columns=c;

      if(!(Matrix=new (rational *)[rows]))
      {
            printf("Matrix could not be constructed\n");
            fflush(stdout);
      }
      for(i=0;i<rows;i++)if(!(Matrix[i]= new rational [columns]))
      {
            printf("Row could not be constructed\n");
            fflush(stdout);
      }

      for(i=0;i<rows;i++)
            for(j=0;j<columns;j++)
            {
                  if(i==j) {rational num(1); Matrix[i][j]=num;}
                  else {rational num(0); Matrix[i][j]=num; }
            }
}
```

```
matrix::matrix(FILE *fp)
{
      int a,b;

      fscanf(fp,"%d%d",&a,&b);

      rows=a;
      columns=b;

      if(!(Matrix=new (rational *)[rows]))
      {
            printf("Matrix could not be constructed by matrix(FILE
*)!!\n");
            fflush(stdout);
      }
      for(int i=0; i<rows; i++)
            if(!(Matrix[i]=new (rational)[columns]))
            {
                  printf("Row could not be constructed by matrix(FILE
*)!!\n");
                  fflush(stdout);
            }

      for(int i=0;i<rows; i++)
            for(int j=0; j<columns; j++)
            {
                  fscanf(fp,"%d%d",&a,&b);
                  Matrix[i][j]=rational(a,b);
            }

}

void matrix::newdimensions(const int r,const int c)
{
      if((rows!=r)||(columns!=c))
      {
            for(int i=0; i<rows; i++) delete[] Matrix[i];
            if(rows!=0) delete[] Matrix;
            rows=r;
            columns=c;

            if(!(Matrix=new (rational *)[rows])) printf("Matrix could not
be constructed by function newdimensions\n");fflush(stdout);
            for(int i=0; i<rows; i++) if(!(Matrix[i]=new
rational[columns])) printf("Row could not be constructed by function
newdimensions\n");fflush(stdout);
      }
      return ;
}


/*
  Dhmiourgia antikeimenou apo ena allo pou hdh uparxei
*/
matrix::matrix(const matrix& m)
{
 int i,j;
 rows=m.rows;
 columns=m.columns;
```

```
 Matrix=new (rational *)[rows];
 for(i=0;i<rows;i++)Matrix[i]= new rational [columns];

 for(i=0;i<rows;i++)
  for(j=0;j<columns;j++)
       Matrix[i][j]=m.Matrix[i][j];
}

/*
  Dhmiourgia antikeimenou me th sunenwsh duo allwn pou hdh uparxoun
*/
matrix::matrix(const matrix& m,const matrix& n)
{
 int i,j;
 if(m.rows!=n.rows){
      cout<<"Cannot Create Matrix :Row numbers differ";
      exit(-1);
 }

 rows=m.rows;
 columns=m.columns+n.columns;

 Matrix=new (rational *)[rows];
 for(i=0;i<rows;i++)Matrix[i]= new rational [columns];

 for(i=0;i<m.rows;i++)
  for(j=0;j<m.columns;j++)
        Matrix[i][j]=m.Matrix[i][j];

 for(i=0;i<m.rows;i++)
      for(j=m.columns;j<columns;j++)
        Matrix[i][j]=n.Matrix[i][j-m.columns];

}

/*
  Dhmiourgia antikeimenou apo pinaka akeraiwn
*/
matrix::matrix(int r,int c, int **A)
{
 int i,j;

 rows=r;
 columns=c;

 Matrix=new (rational *)[rows];
 for(i=0;i<rows;i++)Matrix[i]= new rational [columns];
 for(i=0;i<rows;i++)
  for(j=0;j<columns;j++){
        rational num(A[i][j]);
        Matrix[i][j]=num;
  }

}

matrix::~matrix()
{
 int i;

 for(i=0;i<rows;i++)delete [] Matrix[i];
```

```
 if(rows) delete[] Matrix;
}

/*
  It returns the number of rows of the matrix
*/
int matrix::get_rows() const
{
      return rows;
}

/*
  It returns the number of columns of the matrix
*/
int matrix::get_cols() const
{
      return columns;
}

/*Finds if the matrix A (n-1)*n which is in row echelon form
  contains linearly independent vectors */
bool matrix::linearindependence()
{
 int i,j,k;
 rational r;
 int flag=1;
 matrix P(1,columns);

 if(Matrix[0][0]!=0)return true;

 matrix W(rows,rows);

 for(i=0;i<W.rows;i++)
  for(j=0;j<W.columns;j++)W.Matrix[i][j]=Matrix[i][j+1];

 if(W.det()==0)return false;

 return true;
}

bool matrix::isint()
{
 int i,j;
 bool flag=true;
 for(i=0;i<rows;i++)
   for(j=0;j<columns;j++)
    if(Matrix[i][j].isint()==false){
        flag=false;
        return flag;

    }
 return flag;


}
/*
  Euresh Ka8etou dianusmatos se pinaka
*/
matrix matrix::findnormalvector()
{
```

```
 int i,j,k;
 rational r;
 matrix Vector(1,columns);

 for(i=0;i<rows;i++)if(Matrix[i][i]==0)break;

 if(i<rows){

   Vector.Matrix[0][i]=1;
   for(j=i+1;j<Vector.columns;j++)Vector.Matrix[0][j]=0;
   for(j=i-1;j>=0;j--){
    r=0;
    for(k=j+1;k<=i;k++)r=r-Vector.Matrix[0][k]*Matrix[j][k];
    Vector.Matrix[0][j]=r/Matrix[j][j];
   }
 }
 else{

   //printf("Epilush susthmatos\n");
   Vector.Matrix[0][columns-1]=1;
   for(k=rows-1;k>=0;k--){
     r=0;
     for(j=k+1;j<columns;j++)r=r-Matrix[k][j]*Vector.Matrix[0][j];
     Vector.Matrix[0][k]=r/Matrix[k][k];
   }

 }

 return Vector;
}



/*
  Euresh antistrofou
*/
matrix matrix::inverse() const
{
 int k,i,j;

 if (rows!=columns){
   cout<<"Cannot find inverse matrix:Matrix is not square";
   exit(-1);
 }

 matrix I(rows,columns);
 matrix W(*this,I);
 //W.matrix_print();
 matrix Inv(rows,columns);
 printf("\n");
 rational *P = new rational [W.columns];
 rational m,x,y;

 /*Step 1 Diagonize W*/

 for(k=0;k<rows-1;k++){
   i=k;
   while(W.Matrix[i][k]==0)i++;
   if(i==rows){
     cout<<"Matrix is is not invertible\n";
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
      exit(-1);
    }
    /*Exchange rows i and k of W*/

    for(j=0;j<W.columns;j++){
       P[j]=W.Matrix[i][j];
       W.Matrix[i][j]=W.Matrix[k][j];
       W.Matrix[k][j]=P[j];
    }

    for(i=k+1;i<W.rows;i++){
       m=W.Matrix[i][k]/W.Matrix[k][k];
       for(j=0;j<W.columns;j++)
           W.Matrix[i][j]=W.Matrix[i][j]-(m*W.Matrix[k][j]);
    }
   /*
    printf("\n");
    W.matrix_print();
    printf("\n");
   */
  }

  for(i=0;i<rows;i++){
   for(k=rows-1;k>=0;k--){
     m=0;
     for(j=k+1;j<rows;j++)m=m+W.Matrix[k][j]*P[j];
     P[k]=(W.Matrix[k][columns+i]-m)/W.Matrix[k][k];
     fflush(0);
     Inv.Matrix[k][i]=P[k];
   }
  }

//Inv.matrix_print();
return Inv;

}

/*
  Metatrepei enan pinaka se diagwnio topo8etwntas
  sthn kuria diagwnio to r
*/

int matrix::diagonizematrix(rational r)
{
 int i,j;

 if(rows!=columns){
      cout<<"matrix is not square \n";
      return -1;
 }

 for(i=0;i<rows;i++)
      for(j=0;j<columns;j++){
        if(i==j)Matrix[i][j]=r;
        else Matrix[i][j]=0;
 }
}

/*
  Briskei to beltisto tiling sth genikh periptwsh
```

```
*/
void matrix::optimaltile(int grain)
{
 int i,j,k,l;
 rational m;
 rational *P = new rational [columns];
 matrix U(1,rows);
 matrix W(1,columns);
 matrix B(0,rows);
 int Blines=0;
 int flag;
 int FLAG;
 double max=-1;
 int optilenum=0;

 combinations(columns,rows-1);

 for(l=0;l<count;l++){
      matrix A(rows-1,rows);
      for(j=0;j<A.rows;j++)
        for(k=0;k<A.columns;k++)A.Matrix[j][k]=Matrix[k][comb[l][j]];
      //A.matrix_print();
      //printf("\n\n");


      for(k=0;k<A.rows-1;k++){
        i=k;
        while((i<A.rows)&&(A.Matrix[i][k]==0))i++;
        if(i<A.rows){

             /*Exchange rows i and k of A*/

             for(j=0;j<A.columns;j++){
                   P[j]=A.Matrix[i][j];
                   A.Matrix[i][j]=A.Matrix[k][j];
                   A.Matrix[k][j]=P[j];
             }

             for(i=k+1;i<A.rows;i++){
                   m=A.Matrix[i][k]/A.Matrix[k][k];
                   for(j=0;j<A.columns;j++)
                        A.Matrix[i][j]=A.Matrix[i][j]-(m*A.Matrix[k][j]);
             }
        }
      }
//   A.matrix_print();
//   printf("\n");

      /*Elegxos Grammikhs Anejarthsias*/
      bool b= A.linearindependence();
      if(b==true){
              //cout<<"Einai grammika anejarthta\n";

              /*Euresh Ka8etou Dianusmatos*/
              //A.matrix_print();
              U=A.findnormalvector();
              //U.matrix_print();

              /*Elegxos an r*D>0 h rD<0*/
```

```
                        /*Prosoxh mhpws bgainoun ola mhdenika*/
                W=U*(*this);
                //W.matrix_print();
                int pos=0;
                int neg=0;
                for(i=0;i<W.columns;i++){
                           if(W.Matrix[0][i]>0)pos=1;
                           if(W.Matrix[0][i]<0)neg=1;
                }
                if(pos*neg==0){
                          //printf("r*D>=0 or r*D<=0\n");
                          /*Elegxos an exei hdh xrhsimopoih8ei to dianusma
*/
                          FLAG=0;
                          for(i=0;i<Blines;i++){
                            flag=1;
                            for(j=0;j<B.columns;j++)
                                  if(B.Matrix[i][j]!=U.Matrix[0][j])flag=0;
                            if(flag==0) FLAG++;

                          }
                          //printf("FLAG=%d Blines=%d\n",FLAG,Blines);
                          if(FLAG==Blines){
                                /*Eisagwgh dianusmatos ston pinaka B*/
                                B.rows++;
                                B.Matrix[Blines]= new rational [B.columns];

        for(i=0;i<B.columns;i++)B.Matrix[Blines][i]=U.Matrix[0][i];
                                Blines++;
                          }
                }
                /*printf("\n");
                B.matrix_print();
                printf("\n");*/

        }
        //cout<<"--------------------------\n\n\n";
 }

 printf("Sunolo B\n");
 B.matrix_print();
 printf("\n");

 combinations(B.rows,B.columns);

 //printf("count=%d\n",count);

 //matrix H[count];
 //double solutions[count];

 matrix H[MAXCOMB];
 double solutions[MAXCOMB];
 for(l=0;l<count;l++)solutions[i]=-1;

 for(l=0;l<count;l++){
        H[l].rows=B.columns;
        H[l].columns=B.columns;

        H[l],Matrix=new (rational *) [H[l].rows];
```

```
      for(i=0;i<H[l].rows;i++)H[l].Matrix[i]= new rational
[H[l].columns];

      //printf("l=%d\n",l);
      rational det;
      double temp;
      for(j=0;j<H[l].rows;j++)
        for(k=0;k<H[l].columns;k++){
                   H[l].Matrix[j][k]=B.Matrix[comb[l][j]][k];
        }
      //H[l].matrix_print();
      //printf("\n\n");
      for(i=0;i<H[l].rows;i++){ //gia oles tis grammes
                        matrix line(1,H[l].columns);

for(j=0;j<line.columns;j++)line.Matrix[0][j]=H[l].Matrix[i][j];
                          //line.matrix_print();
                          //printf("\n");
                          matrix c(1,(*this).columns);
                          c=line*(*this);
                          //c.matrix_print();
                          //printf("\n");
                          rational C(0);
                          for(j=0;j<c.columns;j++)C=C+c.Matrix[0][j];
                          //C.printr();printf("\n");

for(j=0;j<H[l].columns;j++)H[l].Matrix[i][j]=H[l].Matrix[i][j]/C;
                          //H[l].matrix_print();
      }

      //H[l].matrix_print();
      //printf("\n");
      det=H[l].det();
      //det.printr();
      //printf("\n");
      temp=(double)fabs(det.rational2double());
      //printf("temp=%f\n",temp);
      if(temp>max){
            max=temp;
            solutions[l]=temp;
            for(i=0;i<l;i++)solutions[l]=-1;
      }
      else if(temp==max)solutions[l]=temp;
      else solutions[l]=-1;

 }
      for(l=0;l<count;l++){
             //printf("%f\n",solutions[l]);
             if(solutions[l]>0){
                  double z=(double)1/grain;
                  //printf("z=%f\n",z);
                  double x=(double)z/solutions[l];
                   //printf("x=%f\n",x);
                  double y=(double)1/rows;
                   //printf("y=%f\n",y);
                  double w=pow(x,y);
                   //printf("w=%f\n",w);
                  H[l].printresult(w);
             }
      }
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
}

matrix matrix::diagonalsum()
{
 int i,j;
 matrix d (rows,rows);
 rational r;

 for(i=0;i<rows;i++){
  r=0;
  for(j=0;j<columns;j++)r=r+Matrix[i][j];
  d.Matrix[i][i]=r;
 }
//d.matrix_print();

return d;

}

/*
  Briskei thn orizousa tou pinaka
*/
rational matrix::det()
{
 int k,i,j;

 if (rows!=columns){
      cout<<"Cannot find determinant :Matrix is not square";
      exit(-1);
 }

 rational *P = new rational [columns];
 rational m;
 matrix W(*this);

 /*Step 1 Diagonize W*/

 for(k=0;k<W.rows-1;k++){
      i=k;
      while((W.Matrix[i][k]==0)&&(i<W.rows-1))i++;
      if((i==W.rows-1)&&(W.Matrix[i][k]==0))return 0;


      /*Exchange rows i and k of W*/


      for(j=0;j<W.columns;j++){
            P[j]=W.Matrix[i][j];
            W.Matrix[i][j]=W.Matrix[k][j];
            W.Matrix[k][j]=P[j];
      }

      for(i=k+1;i<W.rows;i++){
            m=W.Matrix[i][k]/W.Matrix[k][k];
            for(j=0;j<W.columns;j++)
                  W.Matrix[i][j]=W.Matrix[i][j]-(m*W.Matrix[k][j]);
      }

  /*
      printf("\n");
```

```
      W.matrix_print();
      printf("\n");
  */
 }

 //W.matrix_print();
 m=1;
 for(i=0;i<rows;i++)m=m*W.Matrix[i][i];
 return m;

}

 void matrix::matrix_print() const
{
//printf("Enter matrix print\n"); fflush(stdout);
 int i,j;

 for(i=0;i<rows;i++){
  for(j=0;j<columns;j++){cout<<" ";Matrix[i][j].printr();}
  cout<<"\n";
 }
}

 void matrix::printresult(double x)
{
 int i,j;
 for(i=0;i<rows;i++){
      if(i==(rows-1)/2){
            printf("H = %5f*",x);
            for(j=0;j<columns;j++){Matrix[i][j].printr();cout<<" ";}
            cout<<"\n";
      }
      else{
            printf("              ");
            for(j=0;j<columns;j++){Matrix[i][j].printr();cout<<" ";}
            cout<<"\n";
      }
 }
}

matrix matrix::operator+(const matrix ob1) const
{
 int i,j;

 if((rows!=ob1.rows)||(columns!=ob1.columns)){
  cout<<"Add Matrix Error";
  exit(-1);
 }
 else{
  matrix temp(rows,columns);
  for(i=0;i<rows;i++)
       for(j=0;j<columns;j++) temp.Matrix[i][j]=Matrix[i][j] +
ob1.Matrix[i][j];
 return temp;
 }
}

matrix matrix::operator-(const matrix ob1) const
{
 int i,j;
```

```
 if((rows!=ob1.rows)||(columns!=ob1.columns)){
  cout<<"Add Matrix Error";
  exit(-1);
 }
 else{
  matrix temp(rows,columns);
  for(i=0;i<rows;i++)
       for(j=0;j<columns;j++)temp.Matrix[i][j]=Matrix[i][j]-
ob1.Matrix[i][j];
 return temp;
 }
}

matrix matrix::operator*(const matrix ob1) const
{
 int i,j,k;

 if(columns!=ob1.rows){
  cout<<"Multiply  Matrix Error";
  exit(-1);
 }
 else{
  matrix temp(rows,ob1.columns);
  for(i=0;i<temp.rows;i++)
       for(j=0;j<temp.columns;j++)temp.Matrix[i][j]=0;

  for(i=0;i<temp.rows;i++)
       for(j=0;j<temp.columns;j++)
             for(k=0;k<columns;k++)
temp.Matrix[i][j]=temp.Matrix[i][j]+Matrix[i][k]*ob1.Matrix[k][j];

 return temp;
 }
}

matrix matrix::operator=(const matrix ob1)
{
//printf("Operator = entered\n");fflush(stdout);
      if((rows==ob1.rows)&&(columns==ob1.columns))
      {
             for(int i=0;i<rows;i++)
                    for(int j=0;j<columns;j++)
                    {
                          Matrix[i][j]=ob1.Matrix[i][j];
                    }
      }
      else
      {
             for(int i=0; i<rows; i++) delete[] Matrix[i];
             if(rows!=0) delete[] Matrix;
             rows=ob1.rows;
             columns=ob1.columns;

             if(!(Matrix=new (rational *)[rows])) printf("Matrix could not
be constructed by function operator=\n");fflush(stdout);
             for(int i=0; i<rows; i++)
                    if(!(Matrix[i]=new rational[columns]))
                          printf("Row could not be constructed by matrix
operator =\n");
```

```
            fflush(stdout);
            for(int i=0; i<rows; i++)
                  for(int j=0; j<columns; j++)
                        Matrix[i][j]=ob1.Matrix[i][j];
      }
//printf("Operator = is about to exit\n");fflush(stdout);
      return *this;
}

int addcheck(matrix m1,matrix m2)
{
 if ((m1.rows==m2.rows)&&(m1.columns==m2.columns))return 1;
 else return -1;
}

int multcheck(matrix m1,matrix m2)
{
 if ((m1.rows==m2.columns)&&(m1.columns==m2.rows))return 1;
 else return -1;
}

int matrixscan(matrix m1)
{
 int i,j,k;
 int flag;
 int FLAG=0;

 if(m1.rows>m1.columns)return -1;
 /*Check if m1 is the identity matrix*/
 flag=0;
 if(m1.rows==m1.columns){

              for(k=0;k<m1.rows;k++){
                   flag=0;
                   i=0;
                   while((i<m1.rows)&&(m1.Matrix[k][i]==0))i++;
                         if((i<m1.columns)&&(m1.Matrix[k][i]==1)){
                               //printf("Brhka upopto asso sth 8esh %d
%d\n",k,i);
                               flag=1;
                               for(j=1;j<m1.rows;j++){
                                     //printf("shmeio %d
%d\n",(k+j)%m1.rows,i);

if(m1.Matrix[(k+j)%m1.rows][i]!=0)flag=0;
                               }

                         }
                   if(flag==1)FLAG++; /*Gia beltistopoihsh mhpws prepei
na mpei ena break?*/

              }
              if (FLAG==m1.rows)return IDENTITY;
 }

/*----end check for identity matrix----*/

/* check if m1 is a square matrix */
if(m1.rows==m1.columns)return SQUARE;
/*----end check for square matrix----*/
```

```
/*Check if m1 is non-negative and contains a diagonal  matrix*/

 flag=1;
 for(i=0;i<m1.rows;i++)
      for(j=0;j<m1.columns;j++)if(m1.Matrix[i][j]<0)flag=0;
 if(flag==1){

       FLAG=0;

              for(k=0;k<m1.rows;k++){
                   flag=0;
                   for(i=0;i<m1.columns;i++){
                         if(m1.Matrix[k][i]>=1){
                               //printf("Brhka upopto mh mhdeniko sth
8esh %d %d\n",k,i);
                               flag=1;
                               for(j=1;j<m1.rows;j++){
                                //  printf("shmeio %d
%d\n",(k+j)%m1.rows,i);

if(m1.Matrix[(k+j)%m1.rows][i]!=0)flag=0;
                               }
                               if(flag==1){FLAG++;break;}
                         }
                   }

              }
              //printf("FLAG=%d\n",FLAG);
              if (FLAG==m1.rows)return CONTDIAG;

}
/*----end check for non-negative that contains diagonal*/

/* Check if m1 is 2*m matrix */
if(m1.rows==2)return TWODIMENSIONAL;
/* End check for 2-dimensional*/

return FULLROWRANK;
}

/*
      It multiplies all the elements of a matrix
      with the given integer n
*/
int matrix::multiply(const int n)
{
      for(int i=0; i<rows; i++)
            for(int j=0; j<columns; j++)
                  Matrix[i][j]=Matrix[i][j]*n;
      return n;
}
```

Αρχείο "definitions.h"

```
#define MAXDIM 10
#define MAXDEPNUM 15
```

```
#define MAXCOMB 100
//#define MAXROWS 10000
//#define MAXCOLUMNS 8
```

Αρχείο "combinations.h"

```
int combinations(int m,int n);
```

Αρχείο "combinations.cc"

```
#include <stdio.h>
#include <stdlib.h>
#include "definitions.h"


  int count;
  int comb[MAXCOMB][MAXDIM];


int combinations(int m,int n)
{
 int i,j;
 int *point=(int *)malloc(10*sizeof(int));
 int *A=(int *)malloc(10*sizeof(int));
 void find_comb(int num,int level,int step,int tot_depen,int *point,int
*A);

 count=0;
 point=A;
 find_comb(0,n-1,0,m,point,A);

 /*for(i=0;i<count;i++){
      for(j=0;j<n;j++)printf("%d",comb[i][j]);
      printf("\n");
 }
 printf("count=%d\n",count);*/

 return count;
}


 void find_comb(int num,int level,int step,int tot_depen,int *point,int
*A)
{
  int i;

  if(level==step)
                  while(num<tot_depen) {
                             for(i=0;i<level;i++){
                                   comb[count][i]=A[i];
                             }
                             comb[count][i]=num;
                             num++;
                             count++;
```

```
                  }
  else
                  while(num<tot_depen){
                   *point=num;point++;
                   find_comb(num+1,level,step+1,tot_depen,point,A);
                   point--;num++;
                   }
}
```

Παράρτημα 2: Κώδικας συναρτήσεων που υλοποιήθηκαν

Αρχείο "Fourier.h"

```
#include "matrix.h"

int abs(int x);
int gcd(const int x, const int y);
rational gcd(const rational x, const rational y);
void exchange_rows(matrix& Ab, const int r1, const int r2);
void fourier_sort(matrix& Ab, int& p, int& q);
void fourier_next(const matrix& prev, matrix& next,
                                const int p, const int q);
bool air_next(const matrix& prev, const int p, const int q);
matrix *Fourier_Motzkin_Elimination(const matrix& Ab, bool& consistent,
                  const bool Ad_Hoc, const bool exact);
void row_simplify(matrix & Q);
void row_elimination(matrix& A, const int r);
void row_elimination(matrix& A, const int r, int& p, int& q);
void Ad_Hoc_Simplification (matrix *Ab, int *p, int *q);
void remove_zero_rows(matrix *A, const int *q);
void row_negate(matrix& N, const int r);
matrix first_tested(const matrix& A, const int q);
matrix add_test_matrix(const matrix& prev, const matrix& next, const int
q);
void exact_simplification(matrix* A, int *p, int *q);
matrix array_to_1matrix(const matrix *arr);
matrix Fourier_Motzkin_Elimination(const matrix& A, const matrix& b,
                   bool& consistent, const bool Ad_Hoc, const bool
exact);
int row_denomer_lcm(const matrix& Q, const int row);
int matrix_denomer_lcm(const matrix& A);
void upgrade_H(const matrix& H, matrix& A, matrix& b);
void upgrade_H(const matrix& H,const int g, matrix& A, matrix& b);
```

Αρχείο "Fourier.cc"

```
//#include <stdio.h>
#include "Fourier.h"

#define MIN(x,y) ((x)>(y))?(y):(x)
#define MAX(x,y) ((x)>(y))?(x):(y)

/*
      Returns the absolute value of integer x
*/
int abs(int x)
{
  if (x<0) return (-x);
  return x;
}

/*
      It returns the greatest common divider
      of two non negative integer numbers
*/
int gcd(const int x, const int y)
{
  if ((x==0)&&(y==0)) return 1;
  if (x==0) return y;
  if (y==0) return x;
  if (x>=y) return gcd((x%y), y);
  return gcd(x, (y%x));
}

rational gcd(const rational x, const rational y)
{
  rational temp(gcd(abs(x.get_numer()),abs(y.get_numer())),
                gcd(abs(x.get_denomer()),abs(y.get_denomer())));
  return temp;
}
/*
      It exchanges the rows r1 and r2 of matrix Ab
*/
void exchange_rows(matrix& Ab, const int r1, const int r2)
{
      rational *temp=Ab.Matrix[r1];
      Ab.Matrix[r1]=Ab.Matrix[r2];
      Ab.Matrix[r2]=temp;
      return;
}

/*
      It sorts the matrix Ab according to the Fourier-Motzkin
      method so that the first p rows express an upper
      bound and the next q rows express a lower bound
*/
void fourier_sort(matrix& Ab, int& p, int& q)
{
      bool ex=true;
      int k=0;

      while (ex)
        {
```

```
              ex=false;
              for (int m=1; m<Ab.rows-k; m++)
              {
                   rational x1=Ab.Matrix[m-1][Ab.columns-2];
                   rational x2=Ab.Matrix[m][Ab.columns-2];

                   if (((x1==0)&&(x2!=0))||((x1<0)&&(x2>0)))
                   {
                     ex=true;
                     exchange_rows(Ab, m-1, m);
                   }
              }
              k++;
         }

      p=0;
      while ((p<Ab.rows)&&(Ab.Matrix[p][Ab.columns-2]>0)) p++;
      q=p;
      while ((q<Ab.rows)&&(Ab.Matrix[q][Ab.columns-2]<0)) q++;

      return;
}

/*
  It finds the next maxtrix of the Fourier-Motzkin sequence.
  Before this function is called the matrix with appropriate
  dimensions must have been constructed
  in order to be filled.
*/

void fourier_next(const matrix& prev, matrix& next,
                                     const int p, const int q)
{
  int d=0;

  for (int i=0; i<p; i++)
  {
       for(int j=p; j<q; j++)
       {
            rational ci=prev.Matrix[i][(prev.columns)-2];
            rational cj=-prev.Matrix[j][(prev.columns)-2];
            rational g=gcd(ci,cj);
            ci=ci/g;
            cj=cj/g;

            int k=0;
            while (k<((prev.columns)-2))
            {
                  next.Matrix[d][k]= prev.Matrix[i][k]*cj+
prev.Matrix[j][k]*ci;
                  k++;
            }
            next.Matrix[d][k]= prev.Matrix[i][k+1]*cj+
prev.Matrix[j][k+1]*ci;

            d++;
       }
  }

  for (int i=q; i<prev.rows;i++)
```

```
  {
        int k=0;
        while (k<((prev.columns)-2))
            {
                  next.Matrix[d][k]=prev.Matrix[i][k];
                  k++;
            }

            next.Matrix[d][k]=prev.Matrix[i][k+1];

            d++;
  }

  return;
}

/*
  It finds the final maxtrix of the Fourier-Motzkin sequence
  and without storing it tests for consistency.
*/
bool air_next(const matrix& prev, const int p, const int q)
{
  rational temp;
  if(prev.columns!=2) printf("air_next function has been called for an
invalid matrix!\n");

  for (int i=0; i<p; i++)
  {
       for(int j=p; j<q; j++)
       {
            rational ci=prev.Matrix[i][0];
            rational cj=-prev.Matrix[j][0];
            rational g=gcd(ci,cj);
            ci=ci/g;
            cj=cj/g;

            temp=prev.Matrix[i][1]*cj+prev.Matrix[j][1]*ci;
            if(temp<=0) return false;
       }
  }

  for (int i=q; i<prev.rows;i++)
  {
      temp=prev.Matrix[i][1];
      if(temp<0) return false;
  }

  return true;
}


void Ad_Hoc_Simplification (matrix *Ab, int *p, int *q);
void remove_zero_rows(matrix *A, const int *q);
void exact_simplification(matrix* A, int *p, int *q);

//
// It applies Fourier-Motzkin elimination method
// to the matrix Ab and returns the sequence of matrixes
// which express the indexes bounds.
// The inner loop bounds are expressed by the first matrix.
```

```
// If Ad_Hoc=true it applies Ad_Hoc simplification.
// If exact=true it applies exact simplification.
// If consistent=false the given matrix express
// an inconsistent system of inequalities.
//
matrix *Fourier_Motzkin_Elimination(const matrix& Ab,
                  bool& consistent,
                  const bool Ad_Hoc, const bool exact)
{
      const int matrixes_num=(Ab.columns-1);
      matrix *temp=new matrix [matrixes_num];
      int* p=new int[matrixes_num];
      int* q=new int[matrixes_num];

      temp[0]=Ab;
      static int total=0;

      for (int i=0; i<matrixes_num; i++)
      {
            fourier_sort(temp[i], p[i], q[i]);
            if((i+1)<matrixes_num)
            {
                  temp[i+1].newdimensions((p[i]*(q[i]-
p[i])+(temp[i].rows-q[i])),(temp[i].columns-1));
                  fourier_next(temp[i], temp[i+1], p[i], q[i]);
                  total+=temp[i+1].get_rows();
            }
      }

      consistent=air_next(temp[matrixes_num-1],p[matrixes_num-
1],q[matrixes_num-1]);
      total+=p[matrixes_num-1]*q[matrixes_num-1];

      remove_zero_rows(temp, q);
      for(int i=0; i<matrixes_num; i++) row_simplify(temp[i]);

      if (consistent)
      {
            if (Ad_Hoc) Ad_Hoc_Simplification(temp, p, q);
            if (exact)  exact_simplification(temp, p, q);
      }

printf("****row - operations executed = %d******\n",total);
      return temp;
}

/*
      It simplifies each row of a matrix
      by dividing it with the gcd of its elements
*/
void row_simplify(matrix& Q)
{
      for(int i=0; i<Q.rows; i++)
      {
            rational g=Q.Matrix[i][0];
            for(int j=1; j<Q.columns; j++) g=gcd(g,Q.Matrix[i][j]);
            if((g!=1)&&(g!=0))
                  for(int j=0; j<Q.columns; j++)
                        Q.Matrix[i][j]=Q.Matrix[i][j]/g;
            int h=row_denomer_lcm(Q,i);
```

```
            if(h!=1)
                  for(int j=0; j<Q.get_cols(); j++)
                        Q.Matrix[i][j]=Q.Matrix[i][j]*h;
      }
      return;
}

//
//It elininates the r row of the matrix A
//where r is 0 to A.rows-1
//
void row_elimination(matrix& A, const int r)
{
      delete[] A.Matrix[r];
      for (int i=r; i<(A.rows-1); i++)
            A.Matrix[i]=A.Matrix[i+1];
      (A.rows)--;
      A.Matrix[A.rows]=NULL;
      return;
}

//
//It elininates the r row of the matrix A
//where r is 0 to A.rows-1
//and at the same time it gives the right values
//to integers p,q,
//which express the number of upper and upper+lower bounds.
//
void row_elimination(matrix& A, const int r, int& p, int& q)
{
      delete[] A.Matrix[r];
      for (int i=r; i<(A.rows-1); i++)
            A.Matrix[i]=A.Matrix[i+1];
      (A.rows)--;
      A.Matrix[A.rows]=NULL;
      if (r<p) p--;
      if (r<q) q--;
      return;
}

//
// It applies Ad_Hoc simplification method to
// the sequence of matrixes Ab.
//
void Ad_Hoc_Simplification (matrix *Ab, int *p, int *q)
{
      const int v=(Ab[0].columns)-1;
      rational *lmin=new rational[v];
      rational *lmax=new rational[v];
      rational *umin=new rational[v];
      rational *umax=new rational[v];
      for (int i=0; i<v; i++)
      {
            lmin[i]=rational(-1,0);  //=-oo
            lmax[i]=rational(-1,0);  //=-oo
            umin[i]=rational(1,0);   //=+oo
            umax[i]=rational(1,0);   //=+oo
      }

      for (int m=v-1; m>=0; m--)
```

```
      {
            if ((p[m]==0)||(q[m]==p[m]))
            {
                  printf("Warning!! Curious matrix form for matrix
%d.\n",m);
                  printf("An index is unbounded.\n");
                  printf("Cannot apply Ad-Hoc simplification to the
rest\n");
                  return;
            }

            for (int i=0; i<p[m]; i++)
            {
                  rational li=0;
                  for (int j=0; j<v-m-1; j++)
                  {
                        if (Ab[m].Matrix[i][j]>0) li=li-
Ab[m].Matrix[i][j]*umax[v-1-j];
                        else li=li-Ab[m].Matrix[i][j]*lmin[v-1-j];
                  }
                  li=li+(Ab[m].Matrix[i][v-m]);
                  li=li/(Ab[m].Matrix[i][v-m-1]);
                  li.floorint();

                  rational ui=0;
                  for (int j=0; j<v-m-1; j++)
                  {
                        if (Ab[m].Matrix[i][j]>0)
                             ui=ui-Ab[m].Matrix[i][j]*lmin[v-1-j];
                        else ui=ui-Ab[m].Matrix[i][j]*umax[v-1-j];
                  }
                  ui=ui+(Ab[m].Matrix[i][v-m]);
                  ui=ui/(Ab[m].Matrix[i][v-m-1]);
                  ui.floorint();

                  if (umax[m]<=li)
                  {
                        row_elimination(Ab[m],i, p[m], q[m]);
                        i--;
                  }
                  else if (ui<=umin[m])
                  {
                        while(i>0)
                        {
                              row_elimination(Ab[m],0, p[m], q[m]);
                              i--;

                        }
                  }

                  umin[m]=MIN(umin[m],li);
                  umax[m]=MIN(umax[m],ui);
            }

            for (int i=p[m]; i<q[m]; i++)
            {
                  rational ui=0;
                  for (int j=0; j<v-m-1; j++)
                  {
                        if (Ab[m].Matrix[i][j]>0)
```

```
                        ui=ui-Ab[m].Matrix[i][j]*umax[v-1-j];
                    else ui=ui-Ab[m].Matrix[i][j]*lmin[v-1-j];
                }
                ui=ui+Ab[m].Matrix[i][v-m];
                ui=ui/Ab[m].Matrix[i][v-m-1];
                ui.ceilingint();

                rational li=0;
                for (int j=0; j<v-m-1; j++)
                {
                    if (Ab[m].Matrix[i][j]>0)
                        li=li-Ab[m].Matrix[i][j]*lmin[v-1-j];
                    else li=li-Ab[m].Matrix[i][j]*umax[v-1-j];
                }
                li=li+Ab[m].Matrix[i][v-m];
                li=li/Ab[m].Matrix[i][v-m-1];
                li.ceilingint();

                if (ui<=lmin[m])
                {
                    row_elimination(Ab[m],i, p[m], q[m]);
                    i--;
                }
                else if (lmax[m]<=li)
                {
                    while(i>p[m])
                    {
                      row_elimination(Ab[m],p[m],p[m],q[m]);
                      i--;

                    }
                }

                lmin[m]=MAX(lmin[m],li);
                lmax[m]=MAX(lmax[m],ui);
            }

            for (int i=q[m]; i<Ab[m].rows; )
            {
                row_elimination(Ab[m],i);
            }
        }
        return;
}
//
// It removes the rows of the matrixes of the sequence A
// which express no loop bounds for the same loop index
// was the first rows of each matrix.
//
void remove_zero_rows(matrix *A, const int *q)
{
        for(int m=0; m<A[0].columns-1; m++)
        {
            for(int i=q[m]; i<A[m].rows; i++)
                delete[] (A[m]).Matrix[i];
            A[m].rows=q[m];
        }
        return;
}
```

```
//
// It negates the r row of matrix N.
// That means the resulting row expresses the opposite inequality.
//
void row_negate(matrix& N, const int r)
{
	int j=0;
	while (j<N.columns-1)
	{
		N.Matrix[r][j]=-N.Matrix[r][j];
		j++;
	}
	N.Matrix[r][j]=-N.Matrix[r][j]-1;
}

//
// It returns a matrix which has q rows
// identical with the first q rows of A
//

matrix first_tested(const matrix& A, const int q)
{
	matrix temp=matrix(q, A.columns);
	for (int i=0; i<q; i++)
		for (int j=0; j<A.columns; j++)
			temp.Matrix[i][j]=A.Matrix[i][j];
	return temp;
}

//
// It returns the next matrix that must be tested after prev
//
matrix add_test_matrix(const matrix& prev, const matrix& next, const int
q)
{
	if (prev.columns+1!=next.columns)
	{
		printf("Error!! add_test_matrix cannot make the matrix
demanded\n");
		return prev;
	}

	matrix temp=matrix(prev.rows+q, next.columns);
	for (int i=0; i<prev.rows; i++)
	{
		for (int j=0; j<prev.columns-1; j++)
			temp.Matrix[i][j]=prev.Matrix[i][j];
		temp.Matrix[i][prev.columns-1]=0;
		temp.Matrix[i][prev.columns]=prev.Matrix[i][prev.columns-1];
	}
	for (int i=prev.rows; i<temp.rows; i++)
		for (int j=0; j<next.columns; j++)
			temp.Matrix[i][j]=next.Matrix[i-prev.rows][j];

	return temp;
}

//
// It applies the exact simlification method
// to the sequence of matrices A
```

```
//
void exact_simplification(matrix* A, int *p, int *q)
{
	int k=(A[0].columns)-2;
	matrix D=first_tested(A[k], q[k]);
	bool test;
	int startsearch=0;
	int endsearch=0;

	while (k>=0)
	{
		startsearch=endsearch;
		endsearch+=q[k];
		int arow=0;
		for (int i=startsearch ; i<endsearch ; i++)
		{
			if(((arow<p[k])&&(p[k]>1))||((arow>=p[k])&&(q[k]-
p[k]>1)))
			{
				row_negate(D,i);
				matrix* store=Fourier_Motzkin_Elimination(D,
test, false, false);
				delete[] store;
				row_negate(D,i);

				if (!test)
				{
					row_elimination(D, i);
					i--;
					endsearch--;
					row_elimination(A[k],arow, p[k], q[k]);
					arow--;
				}
			}
			arow++;
		}

		if(k>0) D=add_test_matrix(D, A[k-1], q[k-1]);
		k--;
	}

	return;
}

//
// It units the first arr[0].columns-1 elements of arr
// into one matrix.
// So the array arr and the result matrix
// express the same loop bounds.
//
// Attention!! Array arr must contain
// at least arr[0].columns-1 elements
// Each element of arr must contain one column less
// than the previous matrix.
//
matrix array_to_1matrix(const matrix *arr)
{
	int r=0;
	for (int i=0; i<(arr[0].columns-1); i++) r+=arr[i].rows;
	int c=arr[0].columns;
```

```
	matrix res=matrix(r,c);

	int resrow=0;
	for (int k=arr[0].columns-2; k>=0; k--)
	{
		for (int i=0; i<arr[k].rows; i++)
		{
			for (int j=0; j<arr[k].columns-1; j++)
					res.Matrix[resrow][j]= arr[k].Matrix[i][j];
			for (int j=arr[k].columns-1; j<res.columns-1; j++)
					res.Matrix[resrow][j]=0;
			res.Matrix[resrow][res.columns-1]=
arr[k].Matrix[i][arr[k].columns-1];

			resrow++;
		}
	}

	return res;
}

//
// It applies Fourier-Motzkin elimination method
// to the pair of matrices A, b and returns a matrix
// which expresses all the indexes bounds.
// The outer loop bounds are expressed by the first rows.
// If Ad_Hoc=true it applies Ad_Hoc simplification.
// If exact=true it applies exact simplification.
// If consistent=false the given matrices express
// an inconsistent system of inequalities.
//
matrix Fourier_Motzkin_Elimination(const matrix& A,
			const matrix& b, bool& consistent,
			const bool Ad_Hoc, const bool exact)
{
	if (A.rows!=b.rows)
	{
		printf("\nError!! Fourier-Motzkin elimination method cannot
be applied \n");
		printf("for a pair of matrices A,b, which have not the same
number of rows.\n");
		return matrix();
	}
	if (b.columns!=1)
	{
		printf("\nError!! Fourier-Motzkin elimination method cannot
be applied\n");
		printf("for a matrix b which has more than one or none
rows.\n");
		return matrix();
	}

	matrix Ab=matrix(A,b);
	matrix* f=Fourier_Motzkin_Elimination(Ab, consistent, Ad_Hoc,
exact);
	matrix res=array_to_1matrix(f);
	delete[] f;
	return res;
}
```

```
/*
        This function computes the Least Common Multiple
        of the denomers of the elements of a row
        of a rational matrix
*/
int row_denomer_lcm(const matrix& A,const int row)
{
        int temp=1;
        int help;

        for (int j=0; j<A.columns; j++)
        {
                help=gcd(temp,(A.Matrix[row][j].get_denomer()));
                temp*=((A.Matrix[row][j].get_denomer())/help);
        }
        return temp;
}

/*
        This function computes the Least Common Multiple
        of the denomers of the elements of a rational matrix
*/
int matrix_denomer_lcm(const matrix& A)
{
      int temp=1;
      int help;

      for (int i=0; i<A.rows; i++)
            for (int j=0; j<A.columns; j++)
            {
                  help=gcd(temp,(A.Matrix[i][j].get_denomer()));
                  temp*=((A.Matrix[i][j].get_denomer())/help);
            }
      return temp;
}

/*
      This function computes the appropriate matrices
      for the Fourier-Motzkin
      elimination to the interior of the tile (0,0)
      defined by matrix H.
*/
void upgrade_H(const matrix& H, matrix& A, matrix& b)
{
      int x=matrix_denomer_lcm(H);
      matrix H1=H;
      H1.multiply(x);

      b.newdimensions(2*H.rows, 1);
      for(int i=0; i<H.rows; i++) b.Matrix[i][0]=x-1;
      for(int i=H.rows; i<(2*H.rows); i++) b.Matrix[i][0]=0;

      A.newdimensions(2*H.rows,H.columns);
      for(int i=0; i<H.rows; i++)
            for(int j=0; j<H.columns; j++)
                  A.Matrix[i][j]=H1.Matrix[i][j];
      for (int i=H.rows; i<2*H.rows; i++)
            for(int j=0; j<H.columns; j++)
                  A.Matrix[i][j]=-H1.Matrix[i-H.rows][j];
      return;
```

```
}

/*
      This function computes the appropriate matrices
      for the Fourier-Motzkin elimination
      to the interior of the tile (0,0) defined by matrix H,
      when the minimum common product
      of all the denumerators of H is given.
*/
void upgrade_H(const matrix& H,const int g, matrix& A, matrix& b)
{
      matrix H1=H;
      H1.multiply(g);

      b.newdimensions(2*H.rows, 1);
      for(int i=0; i<H.rows; i++) b.Matrix[i][0]=g-1;
      for(int i=H.rows; i<(2*H.rows); i++) b.Matrix[i][0]=0;

      A.newdimensions(2*H.rows,H.columns);
      for(int i=0; i<H.rows; i++)
            for(int j=0; j<H.columns; j++)
                  A.Matrix[i][j]=H1.Matrix[i][j];
      for (int i=H.rows; i<2*H.rows; i++)
            for(int j=0; j<H.columns; j++)
                  A.Matrix[i][j]=-H1.Matrix[i-H.rows][j];
      return;
}
```

Αρχείο “tiling.h”

```
#include "Fourier.h"

#define OUT_EXACT_IN_ORTH 1
#define OUT_EXACT_IN_HERM 2
#define OUT_FAST_IN_ORTH 3
#define OUT_FAST_IN_HERM 4

void Column_Hermite_Normal_Form(matrix& H);
void Column_GCD(matrix& A, int row, int x, int col);
void easy_extern_bounds(const matrix& B,const matrix& P,
                  const matrix& b,matrix& result,
                  const bool simplify,const int g);
void exact_extern_bounds(const matrix& B,const matrix& b,
                  const matrix& H, const int g, matrix& result,
                  const bool simplify);
void tiling_inequalities(const matrix& H, const matrix& B,
                  const matrix& b,
                  const int select,const bool simplify,
                  matrix& tiles, matrix& interior,
                  matrix& transform, matrix& hermite,int **m);
void print_tiling(const matrix& H,const matrix& B,const matrix& b,
                  const int mode, const bool simplify, FILE *outfp);
```

Αρχείο "tiling.cc"

```
#include "tiling.h"

/*
	This function computes the corresponding
	column hermitian matrix for the given one.
	Attention! It overwrites the given matrix.
	Attention! It works only for integer matrices.
*/
void Column_Hermite_Normal_Form(matrix& H)
{
	int row=0, col=0;
	while((row<H.rows)&&(col<H.columns))
	{
		for(int j=col; j<H.columns; j++)
		{
			if(H.Matrix[row][j]<0)
			{	//We start from i=row assuming that
				//all upper elements are already equal to 0
				for(int i=row; i<H.rows; i++)
					H.Matrix[i][j]=-H.Matrix[i][j];
			}
		}

		for(int j=col+1; j<H.columns; j++)
			Column_GCD(H,row,j,col);

		if (H.Matrix[row][col]==0) row++;
		else {
			for(int j=0; j<col; j++)
			{
				rational temp= H.Matrix[row][j]/
H.Matrix[row][col];
				temp.floorint();
				for(int i=row; i<H.rows; i++)
					H.Matrix[i][j]=H.Matrix[i][j]-
(temp*H.Matrix[i][col]);
			}
			row++;
			col++;
		}
	}
	return;
}

/*
	This function computes the gcd of the elements
	[row][col] and [row][x] applying
	the same computations or swaps
	to all the correspondind columns of matrix A.
	It assumes that all elements of columns
	col and x located upper
	from row row are already equal to 0.
*/
void Column_GCD(matrix& A, int row, int x, int col)
{
```

```
	while(A.Matrix[row][x]!=0)
	{
		for(int i=row; i<A.rows; i++) //Swap columns col, x
		{
			rational temp=A.Matrix[i][col];
			A.Matrix[i][col]=A.Matrix[i][x];
			A.Matrix[i][x]=temp;
		}
		rational r=A.Matrix[row][x]/A.Matrix[row][col];
		r.floorint();
		for (int i=row; i<A.rows; i++)
			A.Matrix[i][x]=A.Matrix[i][x]-r*A.Matrix[i][col];
	}
	return;
}

/*
	It applies Fourier-Motzkin Elimination for the product of B*P
	and a matrix which we obtain by modifying b properly
	It can be used for an approximate calculation of the bounds
	of tile indexes.
	We assume that B.rows=b.rows, b.columns=1,
	B.columns=P.rows=P.columns
	and that matrix P is integer.
*/
void easy_extern_bounds(const matrix& B, const matrix& P, const matrix&
b,matrix& result,const bool simplify,const int g)
{
	matrix btemp(b);
	for (int i=0; i<B.rows; i++)
	{
		rational c=0;
		for(int r=0; r<P.get_cols(); r++)
		{
			rational t=0;
			for(int j=0; j<P.get_rows(); j++)
			{
				t=t+B.Matrix[i][j]*P.Matrix[j][r];
			}
			if (t<0) c=c-t;
		}
		c=c*rational(g-1,g);
		btemp.Matrix[i][0]=b.Matrix[i][0]+c;
	}

	matrix Btemp=B*P;
	bool test;
	result=Fourier_Motzkin_Elimination(Btemp, btemp,
test,true,simplify);
	if (!test) printf("Error!! No tile has to be traversed!!\n");

	return;
}

/*
	It computes exactly the bounds of tile indexes.
	H is the matrix which describes the tiling
	and g the mcm of its denumerators.
	The matrices B and b express the bounds
	of the total iteration space.
```

```
      We assume that B.rows=b.rows, b.columns=1,
      B.columns=H.columns=H.rows.
*/
void exact_extern_bounds(const matrix& B, const matrix& b,
            const matrix& H, const int g, matrix& result,
            const bool simplify)
{
      matrix S(B.rows+2*H.rows,2*H.columns);
      for(int i=0; i<B.rows; i++)
      {
            for(int j=0;j<H.columns; j++) S.Matrix[i][j]=0;
            for(int j=H.columns; j<2*H.columns; j++)
                  S.Matrix[i][j]=B.Matrix[i][j-H.columns];
      }
      for(int i=B.rows; i<B.rows+H.rows; i++)
      {
            for(int j=0; j<H.columns; j++)
                  if((i-B.rows)==j) S.Matrix[i][j]=-g;
                  else S.Matrix[i][j]=0;
            for(int j=H.columns; j<2*H.columns; j++)
                  S.Matrix[i][j]=H.Matrix[i-B.rows][j-H.columns]*g;
      }
      for(int i=B.rows+H.rows; i<B.rows+2*H.rows; i++)
      {
            for(int j=0; j<H.columns; j++)
                  if((i-B.rows-H.rows)==j) S.Matrix[i][j]=g;
                  else S.Matrix[i][j]=0;
            for(int j=H.columns; j<2*H.columns; j++)
                  S.Matrix[i][j]=H.Matrix[i-B.rows-H.rows][j-
H.columns]*(-g);
      }

      matrix x(b.rows+2*H.rows, 1);
      for(int i=0; i<b.rows; i++) x.Matrix[i][0]=b.Matrix[i][0];
      for(int i=b.rows; i<b.rows+H.rows; i++) x.Matrix[i][0]=(g-1);
      for(int i=b.rows+H.rows; i<b.rows+2*H.rows; i++)
            x.Matrix[i][0]=0;

      bool test;
      matrix temp= Fourier_Motzkin_Elimination(S, x, test, true,
simplify);
      if (!test) printf("Error!! No tile has to be traversed!!\n");

      // It finally returns only the lines of the final matrix
      //which refer to the n outer loops of the desired code.
      // It searches to find how many lines it will return.
      int last=0;
      while(last<temp.rows)
      {
            bool flag=false;
            for(int j=H.columns; j<2*H.columns; j++)
                  if(temp.Matrix[last][j]!=0)
                  {
                        flag=true;
                        break;
                  }
            if(flag) break;
            last++;
      }
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
	// It makes the matrix to be returned.
	result.newdimensions(last,H.columns+1);
	for(int i=0; i<last; i++)
	{
		for(int j=0; j<H.columns; j++)
			result.Matrix[i][j]=temp.Matrix[i][j];
		result.Matrix[i][H.columns]=temp.Matrix[i][2*H.columns];
	}

	return;
}

/*
	This function computes the transformation matrix
	for the new iteration space.
	We assume that iniatially diag*=NULL
	and thus no space has to be freed.
*/
void space_transform(const matrix& H, matrix& trans, int **diag)
{
	(*diag)=new (int)[H.rows];
	for (int i=0;i<H.rows;i++) (*diag)[i]=row_denomer_lcm(H,i);
	trans.newdimensions(H.rows,H.columns);
	for(int i=0;i<H.rows;i++)
		for(int j=0;j<H.columns; j++)
			trans.Matrix[i][j]=H.Matrix[i][j]*(*diag)[i];
	return;
}

/*
	This function computes the transformed external bounds
	of the iteration space according to
	the system of inequalities BP'<=b.
*/
void transformed_iteration_bounds(const matrix& B,
			const matrix& transform, const matrix& b,
			bool simplify, matrix& result)
{
	if( (B.get_rows()!=b.get_rows()) || (b.get_cols()!=1) ||
(B.get_cols()!=transform.get_rows()) )
	{
		printf("Error! Improper matrices given to
transformed_iteration_bounds function\n");
		return;
	}

	matrix temp=B*transform;
	bool test;
	result=Fourier_Motzkin_Elimination(temp, b, test, true, simplify);
	if(!test) printf("Error! No iteration has to be traversed in the
transformed space.\n");
	return;
}

/*
	This function computes all the necessary matrices
	for the rewriting of the loops.
	The matrix H is the tiling matrix,
	B and b express the bounds of the initial total
	iteration space.
```

```
      The integer select selects the mode in which
      the tiles and their interior will be traversed.
      The matrix tiles express the bounds of the tile space.
      The matrix interior expresses the bounds of tile (0,0)
      if the orthogonal mode of traversing is selected
      or the bounds of the transformed iteration space
      if the transformed mode
      of traversing the interior of each tile is selected.
      If one selects to transform the iteration space,
      the inverse transformation matrix (P')
      is stored in transform,
      the upper bounds of new indexes in m[]
      and the steps and offsets of the new indexes
      in the matrix hermite.
      We assume that the given matrix H is square,invertible
      and its inverse matrix is integer.
      We assume that the number of rows of B is equal
      to the number of rows of b,
      b has 1 column
      and that the equality H.rows=H.columns=B.columns holds.
      We assume that *m is initially NULL.
*/
void tiling_inequalities(const matrix& H, const matrix& B,
                  const matrix& b,
                  const int select,const bool simplify,
                  matrix& tiles, matrix& interior,
                  matrix& transform, matrix& hermite,int **m)
{
      int g=matrix_denomer_lcm(H);
      matrix P=H.inverse();
      if (!(P.isint()))
      {
            printf("Error!! The inverse of the given tiling matrix H is
not an integer matrix!\n");
            return;
      }

      if((select==OUT_EXACT_IN_ORTH)||(select==OUT_EXACT_IN_HERM))
      {
            exact_extern_bounds(B,b,H,g,tiles,simplify);
      }
      else if( (select==OUT_FAST_IN_ORTH) || (select==OUT_FAST_IN_HERM) )
      {
            easy_extern_bounds(B,P,b,tiles,simplify,g);
      }

      if((select==OUT_EXACT_IN_ORTH)||(select==OUT_FAST_IN_ORTH))
      {
            matrix S,x;
            upgrade_H(H,g,S,x);
            bool test;
            interior=Fourier_Motzkin_Elimination(S,x,test,true,simplify);
            if (!test) printf("Error!! There are no internal points in
the tile to be traversed!!\n");
      }
      else if( (select==OUT_EXACT_IN_HERM) || (select==OUT_FAST_IN_HERM)
)
      {
            space_transform(H,hermite,m);
            transform=hermite.inverse();
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

```
            Column_Hermite_Normal_Form(hermite);
            transformed_iteration_bounds(B, transform, b, simplify,
interior);
      }

      return;
}

/*
      This function produces the pseudocode
      of the tiling transformation.
      The matrix H is the tiling matrix,
      B and b express the bounds of the initial total
      iteration space.
      The integer mode selects the mode in which
      the tiles and their interior will be traversed.
      We assume that the given matrix H is square,invertible and
      its inverse matrix is integer.
      We assume that the number of rows of B is equal to
      the number of rows of b,
      b has 1 column and
      that the equality H.rows=H.columns=B.columns holds.
      We assume that the form of B is same
      as the matrices returned by
      the Fourier-Motzkin Elimination function.
*/
void print_tiling(const matrix& H, const matrix& B,
                   const matrix& b,
                   const int mode, const bool simplify, FILE *outfp)
{
      matrix tiles, interior, transf, herm;
      int * m;
      tiling_inequalities(H,B,b,mode,simplify,tiles,interior,transf,herm,
&m);
      int n=H.get_cols();

//For tiles that may cross the Iteration Space bounds:\n\n");
      fprintf(outfp,"\nFor tiles that may cross the Iteration Space
bounds:\n\n");
      int prev=0,pos=0,neg=0;
      for(int i=0; i<n; i++)
      {
            prev=neg; pos=neg;
            while((tiles.Matrix[pos][i]>0)&&(pos<tiles.rows))
                  pos++;
            if(i==(n-1)) neg=tiles.get_rows();
            else
            {
                  neg=pos;
                  while((tiles.Matrix[neg][i+1]==0)&&(neg<tiles.rows))
neg++;
            }

            for (int j=0; j<i; j++) fprintf(outfp, "  ");
            fprintf(outfp,"for(t%d=max(",i+1);

            for (int r=pos; r<neg; r++)
            {
                  if (r>pos) fprintf(outfp,",");
                  fprintf(outfp,"ceil((");
```

```
                (-tiles.Matrix[r][tiles.get_cols()-1]).fprintr(outfp);
                for(int k=0;k<i; k++)
                {
                      if(tiles.Matrix[r][k]>=0)
                            fprintf(outfp,"+");
                      tiles.Matrix[r][k].fprintr(outfp);
                      fprintf(outfp,"*t%d",k+1);
                }
                fprintf(outfp,")/");
                (-tiles.Matrix[r][i]).fprintr(outfp);
                fprintf(outfp,")");
          }

          fprintf(outfp,"); t%d<=min(",i+1);

          for (int r=prev; r<pos; r++)
          {
                if (r>prev) fprintf(outfp,",");
                fprintf(outfp,"floor((");
                tiles.Matrix[r][tiles.get_cols()-1].fprintr(outfp);
                for(int k=0;k<i; k++)
                {
                      if(tiles.Matrix[r][k]<=0)
                            fprintf(outfp,"+");
                      (-tiles.Matrix[r][k]).fprintr(outfp);
                      fprintf(outfp,"*t%d",k+1);
                }
                fprintf(outfp,")/");
                tiles.Matrix[r][i].fprintr(outfp);
                fprintf(outfp,")");
          }

          fprintf(outfp,"); t%d++)\n",i+1);
    }

//For tiles that may cross the Iteration Space bounds:\n\n");
    if((mode==OUT_EXACT_IN_ORTH)||(mode==OUT_FAST_IN_ORTH))
    {
          for(int j=1;j<n;j++) fprintf(outfp,"  ");
          fprintf(outfp,"{\n");
          matrix P=H.inverse();
          for(int i=0;i<n;i++)
          {
                for(int j=0;j<n;j++) fprintf(outfp,"  ");
                fprintf(outfp,"toi%d=",i+1);
                P.Matrix[i][0].fprintr(outfp);
                fprintf(outfp,"*t1");
                for(int j=1;j<n;j++)
                {
                      if(P.Matrix[i][j]>=0) fprintf(outfp,"+");
                      P.Matrix[i][j].fprintr(outfp);
                      fprintf(outfp,"*t%d",j+1);
                }
                fprintf(outfp,";\n");
          }

          prev=0; pos=0; neg=0;
          int prevb=0,posb=0,negb=0;
          for(int i=0; i<n; i++)
          {
```

```
                  prev=neg; pos=neg;
                  while( (interior.Matrix[pos][i]>0) &&
(pos<interior.rows) )
                        pos++;
                  if(i==(n-1)) neg=interior.get_rows();
                  else
                  {
                        neg=pos;
                        while( (interior.Matrix[neg][i+1]==0) &&
(neg<interior.rows) ) neg++;
                  }

                  prevb=negb; posb=negb;
                  while((B.Matrix[posb][i]>0)&&(posb<B.rows))
                        posb++;
                  if(i==(n-1)) negb=B.get_rows();
                  else
                  {
                        negb=posb;
                        while( (B.Matrix[negb][i+1]==0) && (negb<B.rows)
) negb++;
                  }

                  for (int j=-n; j<i; j++) fprintf(outfp, "  ");
                  fprintf(outfp,"for(i%d=max(",i+1);

                  for (int r=posb; r<negb; r++)
                  {
                        if (r>posb) fprintf(outfp,",");
                        fprintf(outfp,"ceil((");
                        (-b.Matrix[r][0]).fprintr(outfp);
                        for(int k=0;k<i; k++)
                        {
                              if(B.Matrix[r][k]>=0)
                                    fprintf(outfp,"+");
                              B.Matrix[r][k].fprintr(outfp);
                              fprintf(outfp,"*i%d",k+1);
                        }
                        fprintf(outfp,")/");
                        (-B.Matrix[r][i]).fprintr(outfp);
                        fprintf(outfp,")");
                  }


                  for (int r=pos; r<neg; r++)
                  {
                        fprintf(outfp,",toi%d+ceil((",i+1);
                        (-interior.Matrix[r][interior.get_cols()-
1]).fprintr(outfp);
                        for(int k=0;k<i; k++)
                        {
                              if(interior.Matrix[r][k]>=0)
                                    fprintf(outfp,"+");
                              interior.Matrix[r][k].fprintr(outfp);
                              fprintf(outfp,"*i%d",k+1);
                        }
                        fprintf(outfp,")/");
                        (-interior.Matrix[r][i]).fprintr(outfp);
                        fprintf(outfp,")");
                  }
```

```
                fprintf(outfp,"); i%d<=min(",i+1);

                for (int r=prevb; r<posb; r++)
                {
                      if (r>prevb) fprintf(outfp,",");
                      fprintf(outfp,"floor((");
                      b.Matrix[r][0].fprintr(outfp);
                      for(int k=0;k<i; k++)
                      {
                            if(B.Matrix[r][k]<=0)
                                  fprintf(outfp,"+");
                            (-B.Matrix[r][k]).fprintr(outfp);
                            fprintf(outfp,"*i%d",k+1);
                      }
                      fprintf(outfp,")/");
                      B.Matrix[r][i].fprintr(outfp);
                      fprintf(outfp,")");
                }

                for (int r=prev; r<pos; r++)
                {
                      fprintf(outfp,",toi%d+floor((",i+1);
                      interior.Matrix[r][interior.get_cols()-
1].fprintr(outfp);
                      for(int k=0;k<i; k++)
                      {
                            if(interior.Matrix[r][k]<=0)
                                  fprintf(outfp,"+");
                            ( -interior.Matrix[r][k] ).fprintr(outfp);
                            fprintf(outfp,"*i%d",k+1);
                      }
                      fprintf(outfp,")/");
                      interior.Matrix[r][i].fprintr(outfp);
                      fprintf(outfp,")");
                }

                fprintf(outfp,"); i%d++)\n",i+1);
          }

          for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
          fprintf(outfp,"{\n");
          for(int j=0; j<2*n;j++) fprintf(outfp,"  ");
          fprintf(outfp,"... ... ...\n");
          for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
          fprintf(outfp,"}\n");
          for(int j=1;j<n;j++) fprintf(outfp,"  ");
          fprintf(outfp,"}\n");
    }
//For tiles that may cross the Iteration Space bounds:\n\n");
    else if((mode==OUT_EXACT_IN_HERM)||(mode==OUT_FAST_IN_HERM))
    {
          for(int j=0; j<n-1;j++) fprintf(outfp,"  ");
          fprintf(outfp,"{\n");
          for(int i=0; i<n;i++)
          {
                for(int j=0; j<n;j++) fprintf(outfp,"  ");
                fprintf(outfp,"toj%d=%d*t%d;\n",i+1,m[i],i+1);
          }
```

```
		prev=0; pos=0; neg=0;

		while((interior.Matrix[pos][0]>0)&&(pos<interior.rows))
			pos++;
		if(n==1) neg=interior.get_rows();
		else
		{
			neg=pos;
			while( (interior.Matrix[neg][1]==0) &&
(neg<interior.rows) ) neg++;
		}

		for(int j=0;j<n;j++) fprintf(outfp,"  ");
		fprintf(outfp,"lb1=max(0");
		for (int r=pos; r<neg; r++)
		{
			fprintf(outfp,",ceil((");
			(-interior.Matrix[r][interior.get_cols()-
1]).fprintr(outfp);
			fprintf(outfp,")/");
			(-interior.Matrix[r][0]).fprintr(outfp);
			fprintf(outfp,")-toj1");
		}
		fprintf(outfp,");\n");

		for(int j=0;j<n;j++) fprintf(outfp,"  ");
		fprintf(outfp,"ub1=min(%d",m[0]-1);
		for (int r=prev; r<pos; r++)
		{
			fprintf(outfp,",floor((");
			interior.Matrix[r][interior.get_cols()-
1].fprintr(outfp);
			fprintf(outfp,")/");
			interior.Matrix[r][0].fprintr(outfp);
			fprintf(outfp,")-toj1");
		}
		fprintf(outfp,");\n");

		for(int j=0; j<n; j++) fprintf(outfp,"  ");
		fprintf(outfp,"for(j1=ceil(lb1/%d)*%d",
herm.Matrix[0][0].get_numer(), herm.Matrix[0][0].get_numer());
		for(int j=1;j<n;j++) fprintf(outfp, ",j%d=ceil(lb1/%d)*%d",
j+1,herm.Matrix[0][0].get_numer(), herm.Matrix[j][0].get_numer());
		fprintf(outfp,";j1<=ub1;");
		fprintf(outfp, "j1+=%d", (herm.Matrix[0][0]).get_numer());
		for(int j=1;j<n;j++) fprintf(outfp, ",j%d+=%d", j+1,
(herm.Matrix[j][0]).get_numer());
		fprintf(outfp,")\n");

		for(int i=1; i<n; i++)
		{
			prev=neg; pos=neg;
			while( (interior.Matrix[pos][i]>0) &&
(pos<interior.rows) ) pos++;
			if(i==(n-1)) neg=interior.get_rows();
			else
			{
				neg=pos;
				while( (interior.Matrix[neg][i+1]==0) &&
(neg<interior.rows) ) neg++;
```

```
                  }

                  for(int j=1;j<n+i;j++)fprintf(outfp,"  ");
                  fprintf(outfp,"{\n");

                  for(int j=0;j<n+i;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"lb%d=max(0",i+1);
                  for (int r=pos; r<neg; r++)
                  {
                        fprintf(outfp,",ceil((");
                        (-interior.Matrix[r][interior.get_cols()-
1]).fprintr(outfp);
                        for(int k=0;k<i; k++)
                        {
                              if(interior.Matrix[r][k]>=0)
                                    fprintf(outfp,"+");
                              interior.Matrix[r][k].fprintr(outfp);
                              fprintf(outfp,"*(j%d+toj%d)",k+1,k+1);
                        }
                        fprintf(outfp,")/");
                        (-interior.Matrix[r][i]).fprintr(outfp);
                        fprintf(outfp,")-toj%d",i+1);
                  }
                  fprintf(outfp,");\n");

                  for(int j=0;j<n+i;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"ub%d=min(%d",i+1,m[i]-1);
                  for (int r=prev; r<pos; r++)
                  {
                        fprintf(outfp,",floor((");
                        interior.Matrix[r][interior.get_cols()-
1].fprintr(outfp);
                        for(int k=0;k<i; k++)
                        {
                              if(interior.Matrix[r][k]<=0)
                                    fprintf(outfp,"+");
                              ( -interior.Matrix[r][k] ).fprintr( outfp
);

                              fprintf(outfp,"*(j%d+toj%d)",k+1,k+1);
                        }
                        fprintf(outfp,")/");
                        interior.Matrix[r][i].fprintr(outfp);
                        fprintf(outfp,")-toj%d",i+1);
                  }
                  fprintf(outfp,");\n");

                  for(int j=0; j<n+i; j++) fprintf(outfp,"  ");
                  fprintf(outfp, "for(j%d+=ceil((lb%d-j%d)/%d)*%d", i+1,
i+1, i+1, herm.Matrix[i][i].get_numer(), herm.Matrix[i][i].get_numer());
                  for(int j=i+1; j<n; j++)
                        fprintf(outfp, ",j%d+=ceil((lb%d-j%d)/%d)*%d",
j+1, i+1, i+1, herm.Matrix[i][i].get_numer(),
herm.Matrix[j][i].get_numer());
                  fprintf(outfp,";j%d<=ub%d;",i+1,i+1);
                  fprintf(outfp, "j%d+=%d", i+1,
herm.Matrix[i][i].get_numer());
                  for(int j=i+1;j<n;j++)
                        fprintf(outfp, ",j%d+=%d", j+1,
herm.Matrix[j][i].get_numer());
                  fprintf(outfp,")\n");
```

```
            }

            for(int j=0; j<2*n-1;j++)
                  fprintf(outfp,"  ");fprintf(outfp,"{\n");
            for(int i=0; i<n;i++)
            {
                  for(int j=0; j<2*n;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"i%d=",i+1);
                  transf.Matrix[i][0].fprintr(outfp);
                  fprintf(outfp,"*(toj1+j1)");
                  for(int j=1;j<n;j++)
                  {
                        if(transf.Matrix[i][j]>=0)
                              fprintf(outfp,"+");
                        transf.Matrix[i][j].fprintr(outfp);
                        fprintf(outfp,"*(toj%d+j%d)",j+1,j+1);
                  }
                  fprintf(outfp,";\n");
            }
            for(int j=0; j<2*n;j++)
                  fprintf(outfp,"  ");
            fprintf(outfp,"... ... ...\n");
            for(int i=2*n;i>=n;i--)
            {
                  for(int j=1; j<i;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"}\n");
            }
      }

//For tiles that do not cross the Iteration Space bounds:\n\n");

      fprintf(outfp,"\nFor tiles that do not cross the Iteration Space
bounds:\n\n");
      prev=0;pos=0;neg=0;
      for(int i=0; i<n; i++)
      {
            prev=neg; pos=neg;
            while((tiles.Matrix[pos][i]>0)&&(pos<tiles.rows))
                  pos++;
            if(i==(n-1)) neg=tiles.get_rows();
            else
            {
                  neg=pos;
                  while( (tiles.Matrix[neg][i+1]==0) && (neg<tiles.rows)
) neg++;
            }

            for (int j=0; j<i; j++) fprintf(outfp, "  ");
            fprintf(outfp,"for(t%d=max(",i+1);

            for (int r=pos; r<neg; r++)
            {
                  if (r>pos) fprintf(outfp,",");
                  fprintf(outfp,"ceil((");
                  ( -tiles.Matrix[r][tiles.get_cols()-1] ).fprintr( outfp
);
                  for(int k=0;k<i; k++)
                  {
                        if(tiles.Matrix[r][k]>=0)
                              fprintf(outfp,"+");
```

```
                        tiles.Matrix[r][k].fprintr(outfp);
                        fprintf(outfp,"*t%d",k+1);
                  }
                  fprintf(outfp,")/");
                  (-tiles.Matrix[r][i]).fprintr(outfp);
                  fprintf(outfp,")");
            }

            fprintf(outfp,"); t%d<=min(",i+1);

            for (int r=prev; r<pos; r++)
            {
                  if (r>prev) fprintf(outfp,",");
                  fprintf(outfp,"floor((");
                  tiles.Matrix[r][tiles.get_cols()-1].fprintr( outfp );
                  for(int k=0;k<i; k++)
                  {
                        if(tiles.Matrix[r][k]<=0)
                              fprintf(outfp,"+");
                        (-tiles.Matrix[r][k]).fprintr(outfp);
                        fprintf(outfp,"*t%d",k+1);
                  }
                  fprintf(outfp,")/");
                  tiles.Matrix[r][i].fprintr(outfp);
                  fprintf(outfp,")");
            }

            fprintf(outfp,"); t%d++)\n",i+1);
      }

//For tiles that do not cross the Iteration Space bounds:\n\n");
      if((mode==OUT_EXACT_IN_ORTH)||(mode==OUT_FAST_IN_ORTH))
      {
            for(int j=1;j<n;j++) fprintf(outfp,"  ");
            fprintf(outfp,"{\n");
            matrix P=H.inverse();
            for(int i=0;i<n;i++)
            {
                  for(int j=0;j<n;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"toi%d=",i+1);
                  P.Matrix[i][0].fprintr(outfp);
                  fprintf(outfp,"*t1");
                  for(int j=1;j<n;j++)
                  {
                        if(P.Matrix[i][j]>=0) fprintf(outfp,"+");
                        P.Matrix[i][j].fprintr(outfp);
                        fprintf(outfp,"*t%d",j+1);
                  }
                  fprintf(outfp,";\n");
            }

            prev=0; pos=0; neg=0;
            for(int i=0; i<n; i++)
            {
                  prev=neg; pos=neg;
                  while( (interior.Matrix[pos][i]>0) &&
(pos<interior.rows) ) pos++;
                  if(i==(n-1)) neg=interior.get_rows();
                  else
                  {
```

```
                        neg=pos;
                        while( (interior.Matrix[neg][i+1]==0) &&
(neg<interior.rows) ) neg++;
                  }

                  for (int j=-n; j<i; j++) fprintf(outfp, "  ");
                  fprintf(outfp,"for(i%d=toi%d+max(",i+1,i+1);

                  for (int r=pos; r<neg; r++)
                  {
                        if (r>pos) fprintf(outfp,",");
                        fprintf(outfp,"ceil((");
                        ( -interior.Matrix[r][interior.get_cols()-1]
).fprintr( outfp );
                        for(int k=0;k<i; k++)
                        {
                              if(interior.Matrix[r][k]>=0)
                                    fprintf(outfp,"+");
                              interior.Matrix[r][k].fprintr(outfp);
                              fprintf(outfp,"*i%d",k+1);
                        }
                        fprintf(outfp,")/");
                        (-interior.Matrix[r][i]).fprintr(outfp);
                        fprintf(outfp,")");
                  }

                  fprintf(outfp,"); i%d<=toi%d+min(",i+1,i+1);

                  for (int r=prev; r<pos; r++)
                  {
                        if (r>prev) fprintf(outfp,",");
                        fprintf(outfp,"floor((");
                        interior.Matrix[r][ interior.get_cols()-1
].fprintr( outfp );
                        for(int k=0;k<i; k++)
                        {
                              if(interior.Matrix[r][k]<=0)
                                    fprintf(outfp,"+");
                              ( -interior.Matrix[r][k] ).fprintr( outfp
);

                              fprintf(outfp,"*i%d",k+1);
                        }
                        fprintf(outfp,")/");
                        interior.Matrix[r][i].fprintr(outfp);
                        fprintf(outfp,")");
                  }

                  fprintf(outfp,"); i%d++)\n",i+1);
            }

            for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"{\n");
            for(int j=0; j<2*n;j++) fprintf(outfp,"  ");
            fprintf(outfp,"... ... ...\n");
            for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"}\n");
            for(int j=1;j<n;j++) fprintf(outfp,"  ");
            fprintf(outfp,"}\n");
      }
//For tiles that do not cross the Iteration Space bounds:\n\n");
```

```
      else if((mode==OUT_EXACT_IN_HERM)||(mode==OUT_FAST_IN_HERM))
      {
            for(int j=0; j<n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"{\n");
            for(int i=0; i<n;i++)
            {
                  for(int j=0; j<n;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"toj%d=%d*t%d;\n",i+1,m[i],i+1);
            }

            for(int j=0; j<n; j++) fprintf(outfp,"  ");
            fprintf(outfp,"for(j1=0");
            for(int j=1;j<n;j++) fprintf(outfp,",j%d=0",j+1);
            fprintf(outfp,";j1<=%d;",m[0]-1);
            fprintf(outfp, "j1+=%d", (herm.Matrix[0][0]).get_numer());
            for(int j=1;j<n;j++) fprintf(outfp, ",j%d+=%d", j+1,
(herm.Matrix[j][0]).get_numer());
            fprintf(outfp,")\n");

            for(int i=1; i<n; i++)
            {
                  for(int j=0; j<n+i; j++) fprintf(outfp,"  ");
                  fprintf(outfp, "for(j%d+=ceil((-j%d)/%d)*%d", i+1, i+1,
herm.Matrix[i][i].get_numer(),herm.Matrix[i][i].get_numer());
                  for(int j=i+1; j<n; j++)
                        fprintf(outfp, ",j%d+=ceil((-j%d)/%d)*%d", j+1,
i+1, herm.Matrix[i][i].get_numer(), herm.Matrix[j][i].get_numer());
                  fprintf(outfp,";j%d<=%d;",i+1,m[i]-1);
                  fprintf(outfp, "j%d+=%d", i+1,
herm.Matrix[i][i].get_numer());
                  for(int j=i+1;j<n;j++) fprintf(outfp, ",j%d+=%d", j+1,
herm.Matrix[j][i].get_numer());
                  fprintf(outfp,")\n");
            }

            for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"{\n");
            for(int i=0; i<n;i++)
            {
                  for(int j=0; j<2*n;j++) fprintf(outfp,"  ");
                  fprintf(outfp,"i%d=",i+1);
                  transf.Matrix[i][0].fprintr(outfp);
                  fprintf(outfp,"*(toj1+j1)");
                  for(int j=1;j<n;j++)
                  {
                        if(transf.Matrix[i][j]>=0)
                              fprintf(outfp,"+");
                        transf.Matrix[i][j].fprintr(outfp);
                        fprintf(outfp,"*(toj%d+j%d)",j+1,j+1);
                  }
                  fprintf(outfp,";\n");
            }
            for(int j=0; j<2*n;j++) fprintf(outfp,"  ");
            fprintf(outfp,"... ... ...\n");
            for(int j=0; j<2*n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"}\n");

            for(int j=0; j<n-1;j++) fprintf(outfp,"  ");
            fprintf(outfp,"}\n");
      }
```

```
      return;
}
```

Αρχείο "test.cc"

```
#include <stdio.h>
//#include "matrix.h"
#include "definitions.h"
#include <stdlib.h>
#include <iostream>
#include "tiling.h"

int main( int argc, char ** argv)
{
  if (argc<5) printf ("Error!!4 file names needed\n");
  FILE *inf=fopen(argv[1],"r");
  matrix P(inf);
  fclose(inf);
  P.matrix_print();
  printf("\n");
  matrix H=P.inverse();
  H.matrix_print();
  printf("\n");

  inf=fopen(argv[2],"r");
  matrix B(inf);
  fclose(inf);
  B.matrix_print();
  printf("\n");

  inf=fopen(argv[3],"r");
  matrix b(inf);
  fclose(inf);
  b.matrix_print();
  printf("\n");

  FILE *fp=fopen(argv[4],"w");
  print_tiling(H,B,b,OUT_EXACT_IN_ORTH,true,fp);
  //print_tiling(H,B,b,OUT_EXACT_IN_HERM,true,fp);
  //print_tiling(H,B,b,OUT_FAST_IN_ORTH,true,fp);
  print_tiling(H,B,b,OUT_FAST_IN_HERM,true,fp);

  fclose(fp);

  printf("\nProgram finished!! \n");fflush(stdout);
}
```

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ

Βιβλιογραφία

[KOZ98] Γ.Κ.Παπακωνσταντίνου, Π.Δ.Τσανάκας. Ν.Γ.Κοζύρης «***Απεικόνιση αλγορίθμων σε αρχιτεκτονικές παράλληλης επεξεργασίας***» Ε.Π.Ι.Σ.Ε.Υ./Ε.Μ.Π. 1998.

[SCH86] A.Schriver, "***Theory of linear and integer programming***," Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons, 1986

[BIK95] A.J.C.Bik and H.A.G.Wijshoff "***Implementation of Fourier-Motzkin Elimination***" Proceedings of the first annual Conference of the ASCI, The Netherlands, pp377-386, 1995.

[ANC91] C.Ancourt, F.Irigoin, "***Scanning Polyhedra with DO Loops***", Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP), pp39-50, April 1991.

[IRI88] F.Irigoin, R.Triolet, "***Supernode Partitioning***", Proc. 15th Annual ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages, pp319-329, San Diego, California, Jan 1988.

[XUE97] J.Xue, "***Communication-Minimal Tiling of Uniform Dependence Loops***", Journal of Parallel and Distributed Computing, vol.42, no.1, p.42-59, 1997.

[WOL91] M.E.Wolf, M.S.Lam, "***A Loop Transformation Theory and an Algorithm to Maximize Parallelism***", IEEE Trans. on Parallel and Distributed Systems, vol.42, no.1, pp.42-59, 1997.

[HOL92] E.H.Hollander, "***Partitioning and Labeling Loops by Unimodular Transformations***", IEEE Trans. on Parallel and Distributed Systems, vol.3, no.4, pp.465-476, Jul. 1992.

[SHA91] W.Shang, J.A.B.Fortes, "***Independent Partitioning of Algorithms with Uniform Dependencies***", IEEE Trans. Comput., vol.41, no.2, pp.190-206, Feb. 1992.

[PEI89] J.Peir, R.Cytron, "***Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors***", IEEE Trans. on Computers vol.38, no.8, Aug. 1989.

[RAM92a] J.Ramanujam, P.Sadayappan, "***Tiling Multidimensional Iteration Spaces for Multicomputers***", Journal on Parallel and Distributed Computing, vol.16, pp.108-120, 1992.

[BOU94] P.Boulet, A.Darte, T.Risset, Y.Robert, ***"(Pen)-ultimate tiling?***", INTEGRATION, The VLSI Journal, volume 17, pp.33-51, 1994

[RAM92b] J.Ramanujam, "***Non-Unimodular Transformations of Nested Loops***", Proceedings of Supercomputing 92, (November 92), pp.214-223, 1992.

[HOD98] E.Hodzic, W.Shang, "***On Supernode Transformation with Minimized Total Running Time***", IEEE Trans. on Parallel and Distributed Systems, vol.9, no.5, pp.417-428, May1998.

[GOU01] G.Goumas, A.Sotiropoulos, N.Koziris, "***Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping***", Int'l Parallel and Distributed Processing Symposium 2001 (IPDPS-2001), San Francisco, California, April 2001.

[GOU01a] G.Goumas, M.Athanasaki, N.Koziris***, "Automatic Code Generation for Tiled Iteration Spaces"***, Submitted to 8th International Conference on High Performance Computing, Hyderabad, India, December 2001.

[GOU01b] G.Goumas, M.Athanasaki, N.Koziris***, "An Efficient Code Generation Technique for Tiled Iteration Spaces"***, Submitted to IEEE Transactions on Parallel and Distributed Systems, June 2001.

© ΜΑΡΙΑ ΑΘΑΝΑΣΑΚΗ