Παράλληλα και Διανεμημένα Συστήματα
Επιτάχυνση Πολλαπλασιασμού Δυαδικών Αραιών Πινάκων (SpGEMM)

Παυλίδης Χρήστος
AEM: 9480
pavlidic@ece.auth.gr

1 Εισαγωγή

Ο πολλαπλασιασμός δυαδικών πινάκων (BMM) βρίσκει χρήση κυρίως στον τομέα των γράφων και των δικτύων καθώς και στον τομέα των τυπικών γλωσσών.

Στο επιστημονικό πεδίο των γράφων, με την βοήθεια του BMM (Binary Matrix Multiplication) μπορεί εύκολα να υπολογιστεί το transitive closure ενός κατευθυνόμενου γράφου, δηλαδή ένας επίσης δυαδικός πίνακας που το αντιπροσωπεύει δείχνοντας όλες τις κορυφές στις οποίες είναι δυνατό να καταλήξεις ξεκινώντας από μία άλλη κορυφή [5]. Επίσης μπορεί να υπολογιστεί και το transitive reduction ενός γράφου, ένας πίνακας/γράφος που έχει όλα τα μονοπάτια του αρχικού με τον ελάχιστο αριθμό ακμών να συνδέουν τις κορυφές του [1]. Στο πεδίο των τυπικών γλωσσών, έχει αποδειχθεί ότι το πρόβλημα της ανάλυσης μίας πρότασης σε οποιαδήποτε γλώσσα (πχ Ελληνικά ή γλώσσα προγραμματισμού C) με ένα σετ από γραμματικούς κανόνες μπορεί να διαμορφωθεί σε απλό πολλαπλασιασμό δυαδικών πινάκων [4].

2 Υλοποίηση

2.1 Σειριακή

Αρχικά υλοποίησα τον αλγόριθμο του πολλαπλασιασμού έτσι ώστε να παίρνει σαν είσοδο ένα sparse matrix A σε CSR μορφή και έναν B σε CSC έτσι ώστε να μπορεί να διασχίζει τις σειρές του A και τις στήλες το B με σκοπό να ελέγχει αν έχουν έστω ένα κοινό στοιχείο, στην οποία περίπτωση βγαίνει από τον βρόγχο και προσθέτει το στοιχείο στον τελικό πίνακα C=A*B. Έτσι, ο πίνακας C υπολογιζόταν στοιχείο προς στοιχείο, σειρά προς σειρά, με αποτέλεσμα να είναι και αυτός σε CSC μορφή.

Έπειτα, υλοποίησα συνάρτηση μετασχηματισμού των CSR πινάκων εισόδου σε blocked form, όπου χωρίζεται ο όλος πίνακας σε blocks και αποθηκεύονται τα μη μηδενικά από αυτά σε CSR μορφή. Το καθένα από τα μη μηδενικά αποθηκευμένα blocks είναι επίσης αραιά σε CSR μορφή. Έτσι, ο τελικός πίνακας έχει αποθηκευμένα τα μη μηδενικά στοιχεία του κάθε μη μηδενικού block το ένα μετά το άλλο. Ο πολλαπλασιασμός των block πινάκων έγινε πάλι με την ίδια σειρά όπως στον απλό πολλαπλασιασμό πινάκων, με την διαφορά ότι τώρα έκανα OR merging μεταξύ των αποτελεσμάτων της πρόσθεσης $C(i,j) = \sum_k A(i,k) * B(k,j)$ για τον υπολογισμό του κάθε τελικού block. Τέλος, αφού είχαν υπολογιστεί όλα τα block, ανακατασκευαζόταν ο τελικός πίνακας σε απλή CSR μορφή.

Ο αλγόριθμος αυτός όμως ήταν αρκετά αργός και για πίνακες που στην Matlab έπαιρναν κάτω από δευτερόλεπτο, εδώ χρειαζόταν αρκετά λεπτά. Στην προσπάθειά μου να βελτιώσω τον αλγόριθμο, ανακάλυψα τον αλγόριθμο πολλαπλασιασμού του Gustavson [3], όπου με reordering της σειράς με την οποία διανύουμε τα στοιχεία καταλήγουμε σε χαμηλότερη πολυπλοκότητα. $\mathcal{O}(nnz_A \cdot \frac{nnz_B}{n_B})$ σε αντίθεση με $\mathcal{O}(n_A \cdot m_B \cdot (\frac{nnz_A}{n_A} + \frac{nnz_B}{n_B})$ στην χειρότερη πριν, για πίνακες μεγέθους $n \times m$. Ο αλγόριθμος αυτός λειτουργεί με τους πίνακες εισόδου να είναι στην ίδια αραιή μορφή. Για την αποφυγή του OR merge των πινάκων, υλοποίησα συνάρτηση insertion στοιχείου σε CSR πίνακα όπου βρίσκει που πρέπει να μπει το στοιχείο με binary search και έπειτα μετακινεί τα υπόλοιπα για να κάνει χώρο και το τοποθετεί. Ο blocking αλγόριθμος τροποποιήθηκε έτσι ώστε κι εκεί να γίνεται πολλαπλασιασμός των πινάκων με την σειρά του Gustavson κι εκεί φάνηκε το όφελος του blocking. Ο χρόνος του BMM για μεγάλους πίνακες έπεσε από δέκα λεπτά σε μερικά δευτερόλεπτα για βέλτιστη επιλογή block, όμως ήμουν ακόμα μακριά τον χρόνο κάτω του δευτερολέπτου της Matlab.

Τελικά, προσπάθησα να υλοποιήσω τον κώδικα που χρησιμοποιεί το πρόγραμμα Julia [2] για BMM που επίσης κάνει χρήση του Gustavson, αλλά αντί για insert, δημιουργεί ένα binary flag array για κάθε σειρά, προσθέτει τα στοιχεία που πρέπει να προστεθούν εκτός σειράς και έπειτα τα ταξινομεί. Παρακάτω παραθέτεται ο ψευδοκώδικας:

Algorithm 1 Gustavson Binary SpGEMM

Input: $A \in \mathbb{R}^{p \times q}, B \in \mathbb{R}^{q \times r}$
Output: $C \in \mathbb{R}^{p \times r}$

```
xb[:] ← 0                                   ▷ Boolean flag array
for a_{i*} ∈ A do
    for a_{ij} ∈ nz(A) do
        for b_{jk} ∈ nz(B) do
            if xb[k] = 0 then
                c_{ik} ← 1
                xb[k] ← 1
            end if
        end for
    end for
    quickSort(c_{i*})                       ▷ Needed for CSR
    for c_{ij} ∈ nz(C) do
        xb[j] ← 0                           ▷ Reset flag array
    end for
end for
```

όπου $a_{i*} = A(i,:)$, $a_{*j} = A(:,j)$ και $a_{ij} = A(i,j)$ σε

Matlab notation.

Η υλοποίηση αυτή ήταν 2 τάξεις μεγέθους πιο γρήγορη και τελείωνε στον μισό χρόνο από την matlab για μεγάλους πίνακες, όπως για το μέγεθος που δώθηκε στο test Matlab κώδικα στην εκφώνηση.

Η εφαρμογή blocking όμως σε αυτόν τον αλγόριθμο, παρά το ότι ο κάθε block πολλαπλασιασμός χρησιμοποιεί το προηγούμενο σαν mask για να υπολογίσει το επόμενο, χειροτέρευε τους χρόνους όσο μίκρυνε το μέγεθος του block. Αυτό γινόταν μάλλον γιατί έπρεπε να αντιγράφει πρώτα τα στοιχεία της σειράς του προηγούμενου block πριν υπολογίσει τα στοιχεία που πρέπει να προστεθούν στην σειρά. Δοκίμασα διάφορα όπως merge με insertion sort αφού υπολογιστούν ξεχωριστά αντί για αντιγραφή κατά τον υπολογισμό, διαφορετικό είδος ταξινόμησης, να αποθηκεύω τις σειρές out of order και να κάνω ταξινόμηση μόνο στο τέλος αλλά τίποτα δεν έφτανε στον χρόνο του απλού πολλαπλασιασμού. Το binary flag array στον απλό αλγόριθμο δρα ήδη σαν mask για ολόκληρη την υπολογιζόμενη σειρά.

2.2 Παράλληλη

Για όλους τους παραπάνω λόγους, αποφάσισα να παραλληλοποιήσω τον απλό Gustavson αλγόριθμο που φαίνεται στον ψευδοκώδικα. Αρχικά, καθώς ο αλγόριθμος υπολογίζει τα στοιχεία του τελικού πίνακα σειρά προς σειρά, δημιούργησα μέθοδο που υπολογίζει μόνο το μέρος του πίνακα που καθορίζεται στην είσοδο. Έπειτα, με την βοήθεια της OpenMP, αφού έσπασα τον πίνακα σε ισόποσα κομμάτια, ανέθεσα τον υπολογισμό του κάθε κομματιού σε διαφορετικό thread ώστε να γίνουν παράλληλα, μιας και δεν εξαρτάται το ένα από το άλλο. Το αποτέλεσμα συνδέεται στο τέλος σε έναν πίνακα. Τέλος, με την χρήση MPI, χώρισα τον πίνακα μεγαλύτερα κομμάτια, και ανέθεσα το καθένα από αυτά σε διαφορετικό group από OpenMP threads.

2.3 Ορθότητα

Για την εξασφάλισή της ορθότητας του αλγορίθμου, υλοποίησα σε Matlab συνάρτηση που δημιουργεί δύο τυχαίους αραιούς πίνακες, τους πολλαπλασιάζει μέσω της Matlab αλλά και με χρήση της σειριακής συνάρτησης σε γλώσσα C και ελέγχει αν βγαίνουν τα ίδια αποτελέσματα. Αν ναι, παίρνουμε μήνυμα επιτυχίας καθώς και του χρόνου που πήρε για τον υπολογισμό σε Matlab και σε C. Παρακάτω βλέπουμε παράδειγμα για υπολογισμό 2 random πινάκων με μέγεθος $n = 5 \cdot 10^6$ με $\simeq 2$ μη μηδενικά στοιχεία ανά γραμμή.

```
>> test_SpGEMM(5e6,2)
Matrices are the same!
Matlab time: 3.294767e+00 sec
C      time: 1.769425e+00 sec
>>
```

Figure 1: Έλεγχος ορθότητας σε Matlab

Για μεγαλύτερους πίνακες, ο χρόνος της C μένει σχεδόν σταθερά μισός από της Matlab. Για μικρότερους πίνακες, η C υλοποίηση είναι αρκετές δεκάδες φορές πιο γρήγορη, όπως παρακάτω για $n = 5 \cdot 10^4$

```
>> test_SpGEMM(5e4,2)
Matrices are the same!
Matlab time: 8.590000e-03 sec
C      time: 1.032300e-02 sec
```

Figure 2: Έλεγχος ορθότητας με μικρότερο n

3 Αποτελέσματα

Παρακάτω παρατίθενται γραφήματα με τα αποτελέσματα της παράλληλης υλοποίησης στην συστοιχία του Αριστοτελείου. Οι τιμές που φαίνονται είναι ο μέσος όρος που πάρθηκε από 5 μετρήσεις στον ίδιο πίνακα για συγκεκριμένο μέγεθος (n) και πυκνότητα (d) πολλαπλασιάζοντάς τον με τον εαυτό του, καθώς δεν υπήρχε χρόνος για παραπάνω μετρήσεις ή και χώρος στην συστοιχία για να δοκιμασθεί διαφορετικός random πίνακας κάθε φορά.

Στην εικόνα 3 βλέπουμε τους χρόνους για 1 MPI task σε 1 node με πολλαπλές τιμές για τον αριθμό των OpenMP threads και την πυκνότητα των μη μηδενικών όρων αλλά για σταθερό μέγεθος n. Γενικά, και για τα επόμενα γραφήματα, ισχύει: $d0\_5 \rightarrow 2.5 \cdot 10^6$, $d1 \rightarrow 5 \cdot 10^6$, $d2 \rightarrow 1 \cdot 10^7$ και $d5 \rightarrow 2.5 \cdot 10^7$ μη μηδενικά στοιχεία αντιστοίχως.

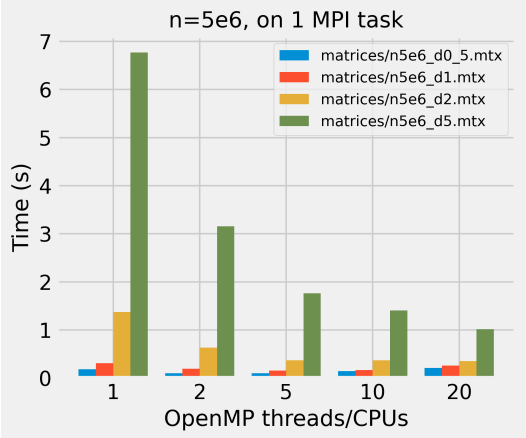

Figure 3: $n = 5 \cdot 10^6$ σε 1 node της συστοιχίας

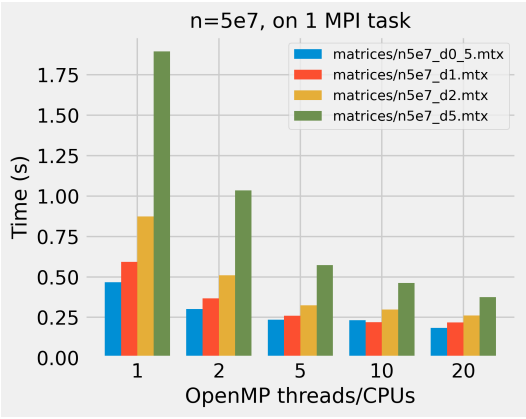

Figure 4: $n = 5 \cdot 10^7$ σε 1 node της συστοιχίας

Παρατηρούμε ότι ο αλγόριθμος φαίνεται να έχει σχετικά καλό scaling το οποίο μοιάζει να φτάνει στον κορεσμό μετά τα 5 10 threads. Επίσης, ο χρόνος φαίνεται να αυξάνει τετραγωνικά με την πυκνότητα d, πράγμα που συνάδει με

την υπόθεσή μας για την πολυπλοκότητά του, μιας και πολλαπλασιάσουμε έναν πίνακα τον εαυτό του.

Στην εικόνα 4, κάνουμε παρόμοιες παρατηρήσεις αλλά για μεγαλύτερο n και σχεδόν ίδιο αριθμό μη μηδενικών στοιχείων. Είναι φανερό ότι η αύξηση του n μείωσε αρκετά τον χρόνο ολοκλήρωσης.

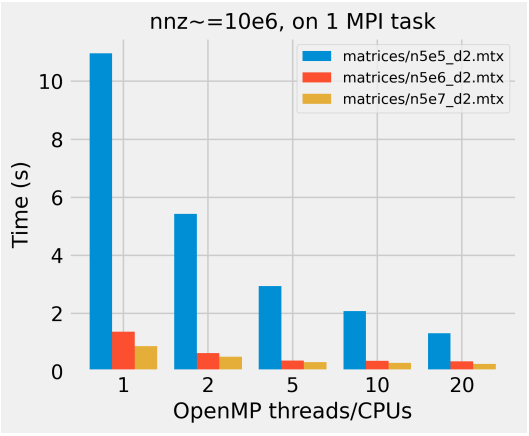

Figure 5: $nnz = 10 \cdot 10^6$ σε 1 node της συστοιχίας

Στην εικόνα 5 οι υποψίες μας επιβεβαιώνονται, καθώς βλέπουμε μεγάλη μείωση του χρόνου για όλο και μεγαλύτερο n, για διαφορετικό αριθμό από CPUs. Στην εικόνα 6 βλέπουμε τους χρόνους ολοκλήρωσης για μεγαλύτερη πυκνότητα μη μηδενικών στοιχείων και σε λογαριθμική κλίμακα. Παρατηρούμε ότι χρησιμοποιώντας 20 threads έχουμε μείωση του χρόνου κατά μία τάξη μεγέθους για όλους τους πίνακες.

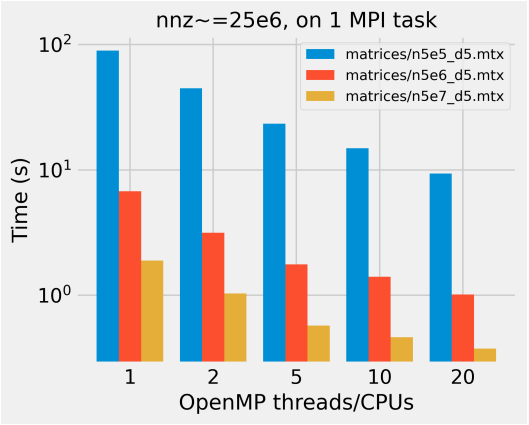

Figure 6: $nnz = 25 \cdot 10^6$ σε 1 node της συστοιχίας

Στην εικόνα 7, βλέπουμε τους Hybrid χρόνους για σταθερό αριθμό από OpenMP threads και μεταβαλλόμενο αριθμό από MPI tasks, σε ένα μόνο node της συστοιχίας. Παρατηρούμε παρόμοιο scaling με πριν, το οποίο φαίνεται να είναι αρκετά καλό ειδικά για μεγαλύτερο φόρτο εργασίας.

Στην 8, έχουμε τα αποτελέσματα για τον ίδιο συνολικό αριθμό από CPUs αλλά διαφορετικές αναλογίες μεταξύ MPI task και OpenMP threads. Εύκολα θα μπορούσε κανείς να βγάλει το συμπέρασμα ότι η χρήση MPI συμφέρει περισσότερο σε σχέση με OpenMP παρά την ευκολία χρήσης της τελευταίας αν αυτό που θέλει την μέγιστη απόδοση.

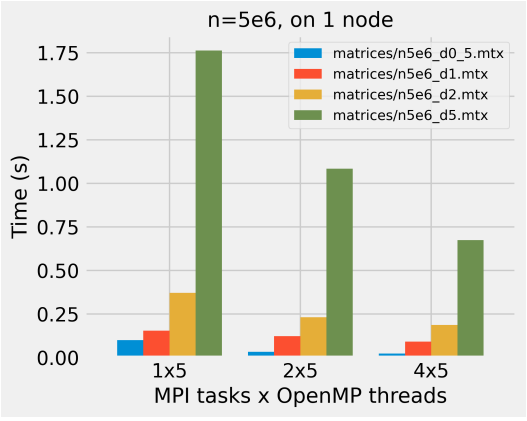

Figure 7: MPI+OpenMP Hybrid για διαφορετικό αριθμό από MPI task

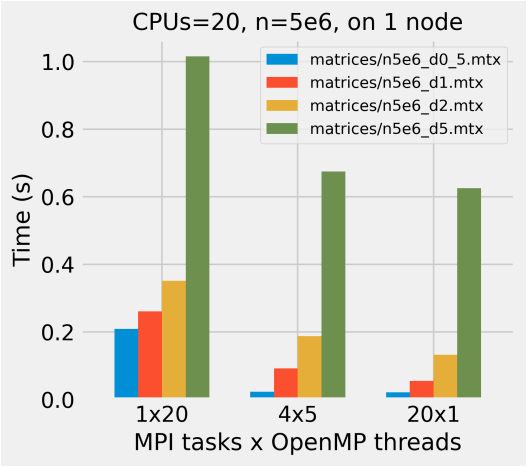

Figure 8: 20 CPUs σε διαφορετικές αναλογίες

Στην συνέχεια κάνουμε μία μικρή ανάλυση για την χρήση του προγράμματος μεταξύ διαφορετικών nodes της συστοιχίας.

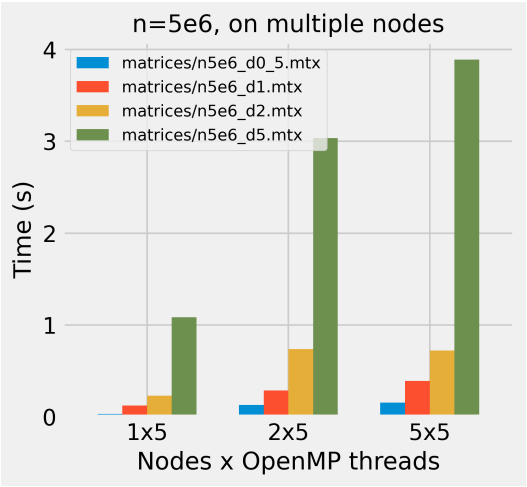

Figure 9: 5 OpenMP threads και διαφορετικός αριθμός από nodes

Δυστυχώς, όπως είναι ξεκάθαρο από την εικόνα 9, η χρήση πολλαπλών nodes δεν προσφέρει μείωση χρόνου, αλλά μάλλον το αντίθετο.

Βέβαια, αν παρατηρήσουμε και την εικόνα 10, βλέπουμε ότι, αν και όχι τόσο μεγάλη όσο όταν χρησιμοποιούμε

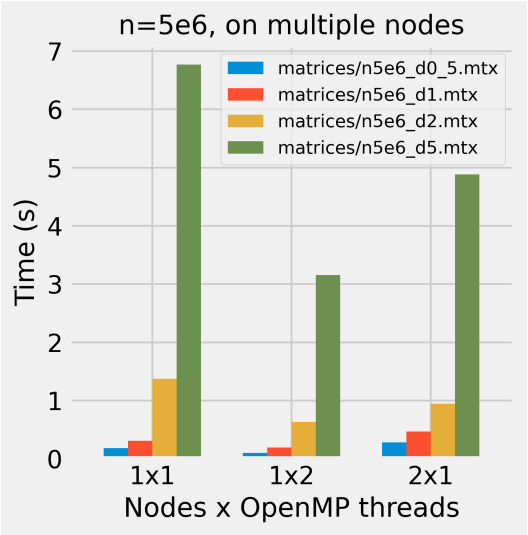

Figure 10: Έλεγχος ορθότητας με μικρότερο n

CPUs στο ίδιο node, η χρήση περισσότερων MPI task σε διαφορετικά nodes προσφέρει κάποια βελτίωση στους χρόνους ολοκλήρωσης. Δυστυχώς πάλι, δεν υπήρχε αρκετός χρόνος για την συλλογή περισσότερων δεδομένων έτσι ώστε να καταλήξουμε σε κάποιο περισσότερο συγκεκριμένο συμπέρασμα καθώς και η συστοιχία ήταν κρατημένη από άλλες εργασίες.

Το μόνο που θα μπορούσαμε ίσως να υποθέσουμε είναι ότι, είτε η MPI δεν διαχειρίζεται καλά την OpenMP όταν πρέπει και να επικοινωνήσει μεταξύ nodes, είτε η δουλειά στο τεστ της εικόνας 9, αν και αρκετών δευτερολέπτων για d=5, δεν ήταν αρκετή ώστε να δικαιολογήσει το overhead της μεταφοράς των αποτελεσμάτων μεταξύ των nodes. Το τελευταίο συνάδει βέβαια και με το ότι στην εικόνα 10, ο χρόνος αυξάνεται για τα μικρά d των 0.5 και 1 από 1 node 1 cpu σε 2 nodes από μία CPU το καθένα. Δεν μπορούμε βέβαια να είμαστε σίγουροι, καθώς ακόμη χρησιμοποιήσαμε τον ίδιο πίνακα κάθε φορά, όπου μπορεί να παίξει σημαντικό ρόλο και η τοπολογία του.

Github link: https://github.com/pavlidic/Binary-SpGEMM

References

[1] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[4] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1):1–15, 2002.

[5] Patrick E O'Neil and Elizabeth J O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Information and Control*, 22(2):132–138, 1973.