ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΑΡΑΛΛΗΛΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Heat 2D (MPI-OpenMP)

Χρήστος Πατσούρας Μ1612

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2019

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

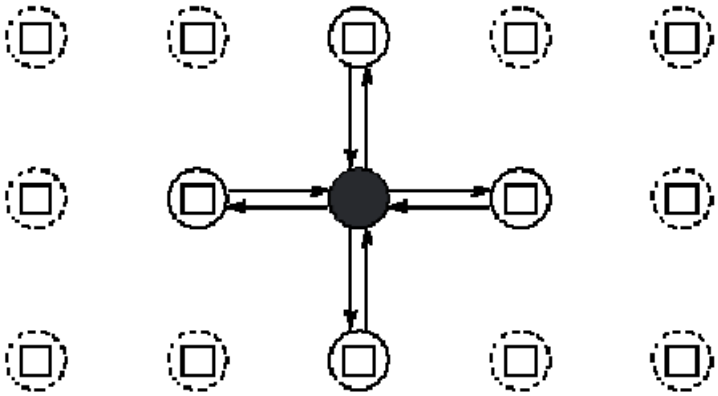
ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

1. ΕΙΣΑΓΩΓΗ

Στόχος της εφαρμογής είναι η προσομοίωση μεταφοράς θερμότητας σε μια επιφάνεια. Στην ουσία έχουμε ένα πίνακα διάστασης $M \times N$, όπου κάθε στοιχείο του πίνακα είναι ένα σημείο του χώρου και έχει μια συγκεκριμένη θερμοκρασία. Η αρχική θερμοκρασία είναι υψηλή στο κέντρο και μηδέν στα ακριανά στοιχεία. Το σύστημα αλλάζει κατάσταση με την πάροδο του χρόνου. Αυτό συμβαίνει διότι κάθε σημείο επηρεάζει και επηρεάζεται από τα γειτονικά του. Γειτονικά στοιχεία θεωρούνται αυτά που βρίσκονται πάνω, κάτω, δεξιά και αριστερά σε ένα στοιχείο, όπως στην Εικόνα 1. Τα στοιχεία που βρίσκονται στα άκρα δεν αλλάζουν τιμές καθώς θεωρούνται στοιχεία που απορροφούν ή εκπέμπουν θερμότητα στο σύστημα.

Το πρόγραμμα που δίνεται ως πρότυπο για τις απαιτήσεις της άσκησης είναι διαθέσιμο εδώ και ζητούμενο είναι η αξιολόγηση, ο επανασχεδιασμός και η βελτίωση του με στόχο την καλύτερη κλιμάκωση και η σύγκριση χρονικά, σε επιτάχυνση και σε αποδοτικότητα του προγράμματος αυτού με αυτά που θα παραχθούν σε MPI, σε MPI+OpenMP (υβριδικό) και σε Cuda.



Εικόνα 1: Γείτονες σημείου

2. ΣΧΕΔΙΑΣΜΟΣ ΔΙΑΜΟΙΡΑΣΜΟΥ ΔΕΔΟΜΕΝΩΝ ΣΤΙΣ ΔΙΕΡΓΑΣΙΕΣ

2.1 Partitioning

Εν γένει το partitioning μπορεί να γίνει ως προς τα δεδομένα και ως προς τους υπολογισμούς ή τη λειτουργία που πρέπει να επιτελεστεί. Σε αυτή την περίπτωση, το partitioning μπορεί να γίνει μόνο ως προς τα δεδομένα διότι υπάρχει ένας μοναδικός υπολογισμός για όλα τα δεδομένα. Συγκεκριμένα μπορούμε να αποφασίσουμε το πως θα διαχωρίσουμε τα δεδομένα του πίνακα με διάφορους τρόπους. Εφόσον ο φόρτος επεξεργασίας είναι ο ίδιος για όλο τον πίνακα, δεν υπάρχει λόγος να γίνει κάποιος κυκλικός διαμοιρασμός. Οπότε τον πίνακα θα τον χωρίσουμε είτε σε γραμμές είτε σε στήλες είτε σε blocks. Στο δοσμένο πρόγραμμα ο διαχωρισμός γίνεται σε γραμμές. Εμείς θα αποδείξουμε ότι ο διαχωρισμός σε blocks είναι ορθότερος ως προς την κλιμάκωση. Αυτόν θα ακολουθήσουμε και στην υλοποίηση του νέου προγράμματος.

2.2 Communication

Όπως αναφέρθηκε και στην Ενότητα 1, η τιμή ενός στοιχείου εξαρτάται από τα τέσσερα γειτονικά του σε διάταξη βοράς-νότος-ανατολή-δύση. Άρα, σε περίπτωση διαχωρισμού του πίνακα, οι διεργασίες πρέπει να ανταλλάσσουν πληροφορίες των στοιχείων που χρειάζονται οι «γειτονικές» διεργασίες για τους υπολογισμούς που πραγματοποιούν. Αυτές θα είναι είτε οι ακραίες γραμμές είτε οι ακραίες στήλες στις περιπτώσεις διαχωρισμού σε οριζόντιες ή κάθετες λωρίδες αντίστοιχα και συγκεκριμένα η πρώτη και η τελευταία κάθε υποπίνακα. Στην περίπτωση διαχωρισμού σε blocks θα είναι όλα τα ακραία στοιχεία, συγκεκριμένα η πρώτη και η τελευταία γραμμή και στήλη κάθε υποπίνακα. Θεωρούμε ότι η επικοινωνία μεταξύ των υπολογιστών είναι ίδιων τεχνικών χαρακτηριστικών και ότι δεν υπάρχει επικοινωνιακός φόρτος από άλλες εφαρμογές.

Η επικοινωνία στο συγκεκριμένο πρόβλημα είναι:

- Local: επικοινωνούμε με ένα μικρό τμήμα των διεργασιών, τις γειτονικές μας, όχι με όλες.
- Structured: Ακολουθούμε καρτεσιανή τοπολογία για να δομίσουμε την επικοινωνία και επικοινωνούμε μόνο με τέσσερις γείτονες στην περίπτωση του διαχωρισμού σε blocks. Στην περίπτωση του διαχωρισμού σε γραμμές ή στήλες έχουμε μόνο δύο γείτονες, όπως στο παράδειγμα που δίνεται.
- Static: οι γείτονες είναι πάντα οι ίδιοι, ορίζονται στην αρχή του προγράμματος και δεν αλλάζουν με την πάροδο του χρόνου.
- Synchronous: οι παραγωγοί και οι καταναλωτές των μηνυμάτων εκτελούνται συντονισμένα και συνεργάζονται για τη μεταφορά δεδομένων.

Στη δική μας υλοποίηση θα ακολουθήσουμε το διαχωρισμό σε blocks και τη διάταξη με τους τέσσερις γείτονες για κάθε διεργασία.

2.3 Agglomeration

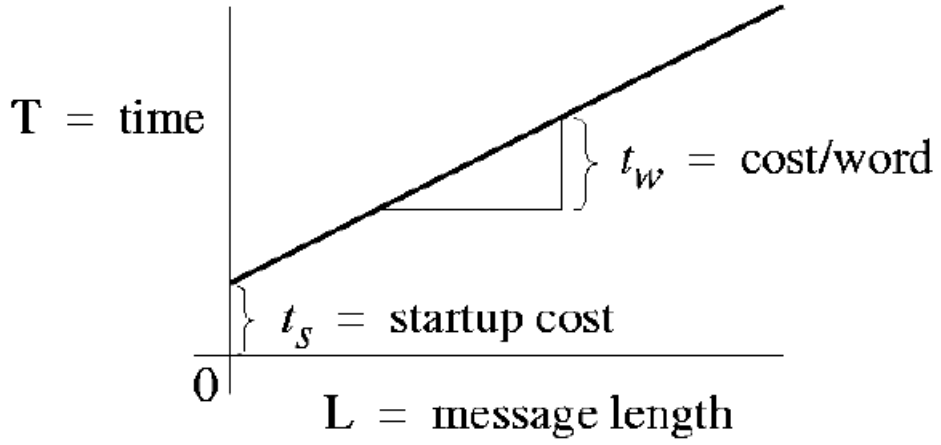
Θα εξετάσουμε και τις δύο προαναφερθείσες περιπτώσεις διαχωρισμού, δηλαδή σε λωρίδες (γραμμές ή στήλες) και blocks, και θα αποδείξουμε ότι ο διαχωρισμός σε blocks κλιμακώνει καλύτερα. Έστω ότι έχουμε ένα πίνακα $M \times N$. Χωρίς βλάβη της γενικότητας θα θεωρήσουμε ότι $M \leq N$. Αφού ισχύει $M \leq N$, για να έχουμε λιγότερα στοιχεία για ανταλλαγή, ο διαχωρισμός σε λωρίδες θα είναι σε στήλες. Δε θα είχε διαφορά ο

διαχωρισμός σε γραμμές. Ο συνολικός χρόνος εκτέλεσης ενός παράλληλου προγράμματος υπολογίζεται ως εξής:

$$T = \frac{1}{p} \cdot \left(\sum_{i=1}^{p-1} T_{comp}^{i} + \sum_{i=1}^{p-1} T_{comm}^{i} + \sum_{i=1}^{p-1} T_{idle}^{i} \right)$$

όπου T_{comp}: χρόνος υπολογισμών, T_{comm}: χρόνος επικοινωνίας και T_{idle}: χρόνος αδράνειας/αναμονής.

Η διάρκεια του χρόνου αναμονής δε μπορεί να προβλεφθεί για αυτό και δε θα υπολογιστεί στο αποτέλεσμα. Αφού έχουμε ομοιογενή συστήματα, το T_{comp} για ένα στοιχείο θα είναι παντού ίδιο και δεν εξαρτάται από τον διαμοιρασμό, αλλά από τη φύση του προβλήματος. Επίσης για κάθε επικοινωνία που θα κάνει κάθε διεργασία το κόστος υπολογίζεται μέσω του τύπου $T_{msg} = t_s + t_w \cdot L$ (βλ. Εικόνα 2).



Εικόνα 2: Απλό μοντέλο επικοινωνίας

2.3.1 Διαχωρισμός σε λωρίδες

Στην περίπτωση που γίνει ο διαχωρισμός σε P λωρίδες, αφού $M \leq N$, ο διαχωρισμός θα γίνει σε στήλες για να έχουμε λιγότερες ανταλλαγές στοιχείων. Άρα κάθε διεργασία θα έχει $M \left\lfloor \frac{N}{P} \right\rfloor$ στοιχεία, αλλά οι πρώτες $Nmodp$ θα έχουν μια επιπλέον στήλη. Επίσης η πρώτη και η τελευταία διεργασία θα έχει μια λιγότερη στήλη. Άρα σύμφωνα με το παραπάνω ο χειρότερος χρόνος που μπορούμε να πετύχουμε είναι:

$$T_{comp,max} = t_c \cdot \left(M \left\lfloor \frac{N}{P} \right\rfloor + 1 \right)$$

Για την επικοινωνία θα πάρουμε τη χειρότερη περίπτωση όπου μια διεργασία έχει κομμάτι του πίνακα που έχει δυο γείτονες. Από την ανταλλαγή των δυο στηλών που θα γίνει μεταξύ των δυο διεργασιών προκύπτει το εξής:

$$T_{msg} = 2(t_s + t_w \cdot (2 \cdot M))$$

Συνεπώς:

- Execution time: $T = T_{comp,max} + T_{msg} = t_c \cdot \left(M \left\lfloor \frac{N}{P} \right\rfloor + 1 \right) + 2 \cdot t_s + 4 \cdot M \cdot t_w$
- Speedup: $S = \frac{t_c \cdot M \cdot N}{t_c \cdot \left(M \left\lfloor \frac{N}{P} \right\rfloor + 1 \right) + 2 \cdot t_s + 4 \cdot M \cdot t_w}$

- Efficiency: $E = \frac{t_c \cdot M \cdot N}{\left(t_c \cdot \left(M\left\lfloor\frac{N}{P}\right\rfloor+1\right)+2 \cdot t_s+4 \cdot M \cdot t_w) \cdot\right) \cdot P}$

2.3.2 Διαχωρισμός σε blocks

Θεωρούμε ότι ο διαχωρισμός γίνεται σε P blocks και χωρίς βλάβη της γενικότητας θα θεωρήσουμε ότι ο P είναι τέλειο τετράγωνο και κάθε διάσταση μοιράζεται σε $\sqrt{P}$ blocks. Σε αυτήν την περίπτωση σε κάθε μπλοκ θα έχουμε $\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor \cdot \left\lfloor\frac{N}{\sqrt{P}}\right\rfloor$ στοιχεία σε κάθε block. Οι πρώτες $Nmod\sqrt{P}$ διεργασίες θα έχουν μια επιπλέον στήλη, ενώ οι πρώτες $Mmod\sqrt{P}$ μια επιπλέον γραμμή. Επίσης όλες οι διεργασίες που θα περιλαμβάνουν ακραία στοιχεία θα έχουν μια λιγότερη γραμμή ή/και στήλη. Άρα σύμφωνα με το παραπάνω ο χειρότερος χρόνος που μπορούμε να πετύχουμε είναι:

$$T_{comp,max} = t_c \cdot \left(\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor + 1\right) \cdot \left(\left\lfloor\frac{N}{\sqrt{P}}\right\rfloor + 1\right)$$

Για την επικοινωνία θα πάρουμε τη χειρότερη περίπτωση όπου μια διεργασία έχει κομμάτι του πίνακα που έχει δυο γείτονες. Από την ανταλλαγή των δυο στηλών που θα γίνει μεταξύ των δυο διεργασιών προκύπτει το εξής:

$$T_{msg} = 2 \cdot \left(t_s + t_w \cdot 2 \cdot \left(\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor + 1\right)\right) + 2 \cdot \left(t_s + t_w \cdot 2 \cdot \left(\left\lfloor\frac{N}{\sqrt{P}}\right\rfloor + 1\right)\right)$$

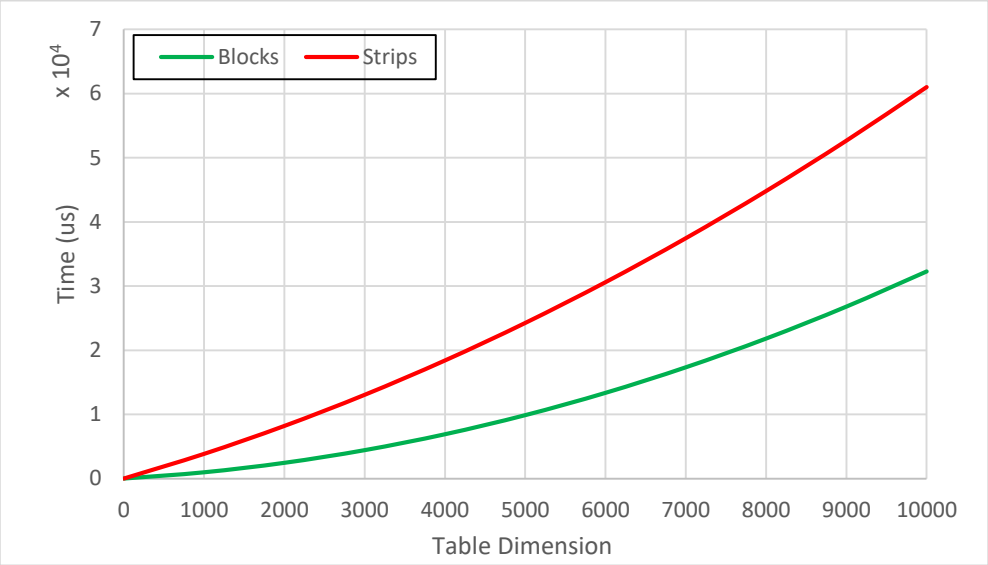
Συνεπώς:

- Execution time: $T = T_{comp,max} + T_{msg} = 4 \cdot t_s + (4 \cdot t_w + t_c) \cdot \left(\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor + 1\right) \cdot \left(\left\lfloor\frac{N}{\sqrt{P}}\right\rfloor + 1\right)$
- Speedup: $S = \frac{t_c \cdot M \cdot N}{4 \cdot t_s+(4 \cdot t_w+t_c) \cdot \left(\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor+1\right) \cdot \left(\left\lfloor\frac{N}{\sqrt{P}}\right\rfloor+1\right)}$
- Efficiency: $E = \frac{t_c \cdot M \cdot N}{\left(4 \cdot t_s+(4 \cdot t_w+t_c) \cdot \left(\left\lfloor\frac{M}{\sqrt{P}}\right\rfloor+1\right) \cdot \left(\left\lfloor\frac{N}{\sqrt{P}}\right\rfloor+1\right)\right) \cdot P}$

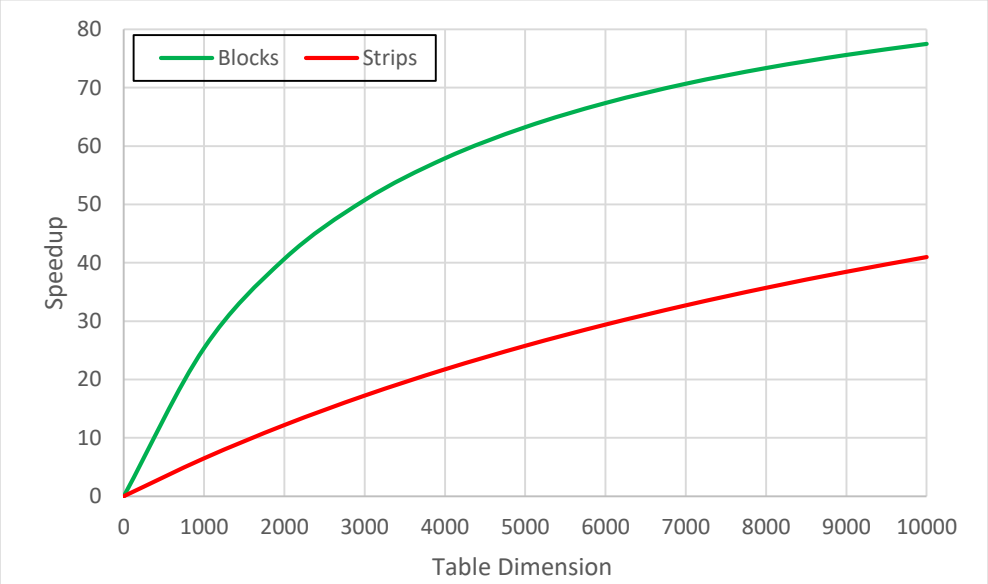
2.3.3 Σύγκριση διαχωρισμών

Από μετρήσεις με το mpptest και από εκτέλεση του ανάλογου κώδικα σε ένα δοκιμαστικό πίνακα βρέθηκαν ότι $t_c = 0.025\ us$, $t_s = 0.6\ us$ και $t_w = 0.9\ us$.

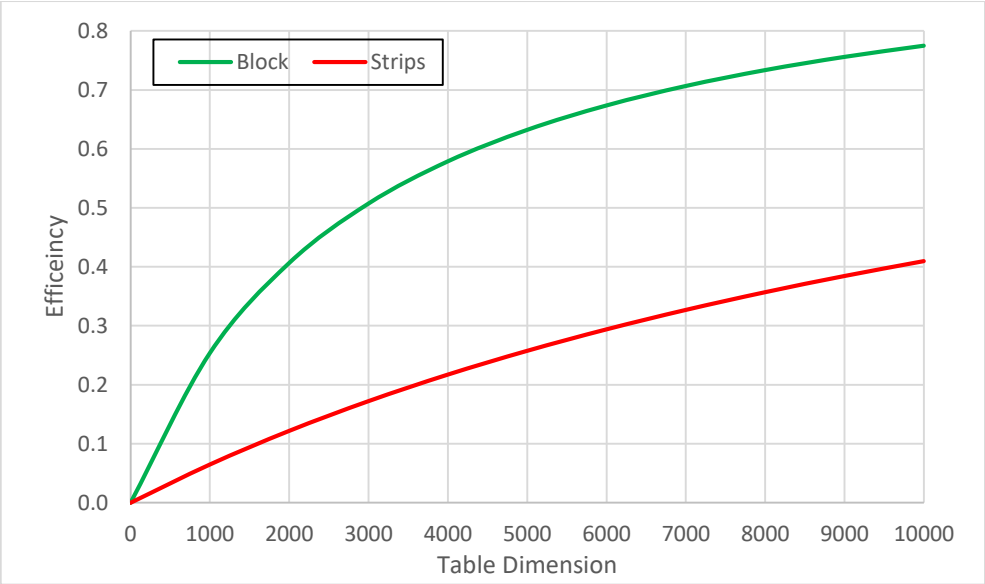
Παρακάτω παρουσιάζονται τα διαγράμματα χρόνου, επιτάχυνσης και αποδοτικότητας των δυο επιλογών υλοποίησης μεταβάλλοντας το μέγεθος του πίνακα. Για την εξαγωγή τους χρησιμοποιήθηκε ένας πίνακας τετράγωνος (M=N) και P=100.



Σχήμα 1: Σύγκριση χρόνου εκτέλεσης με βάση το μέγεθος δεδομένων μεταξύ μπλοκ και λωρίδων

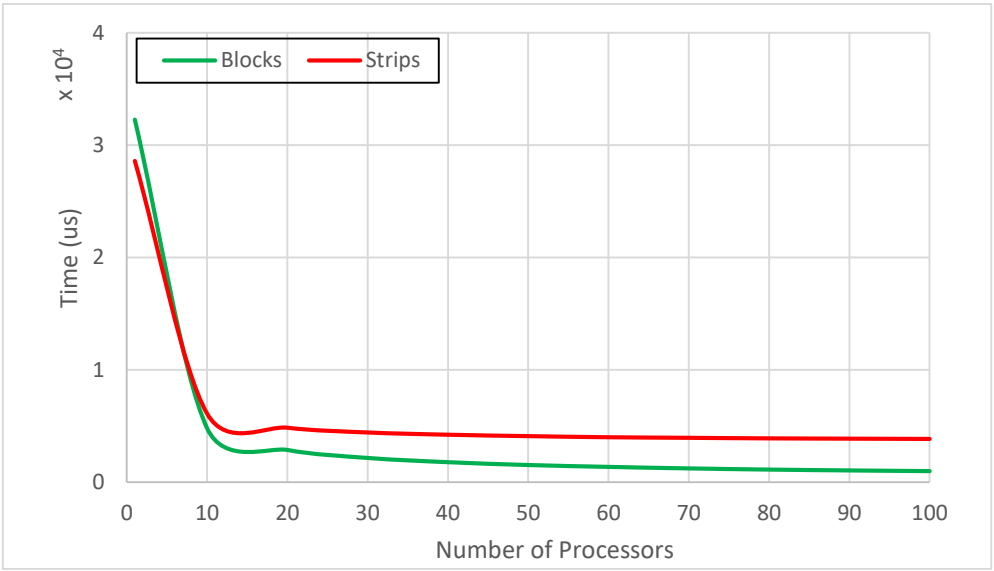


Σχήμα 2: Σύγκριση επιτάχυνσης με βάση το μέγεθος δεδομένων μεταξύ μπλοκ και λωρίδων

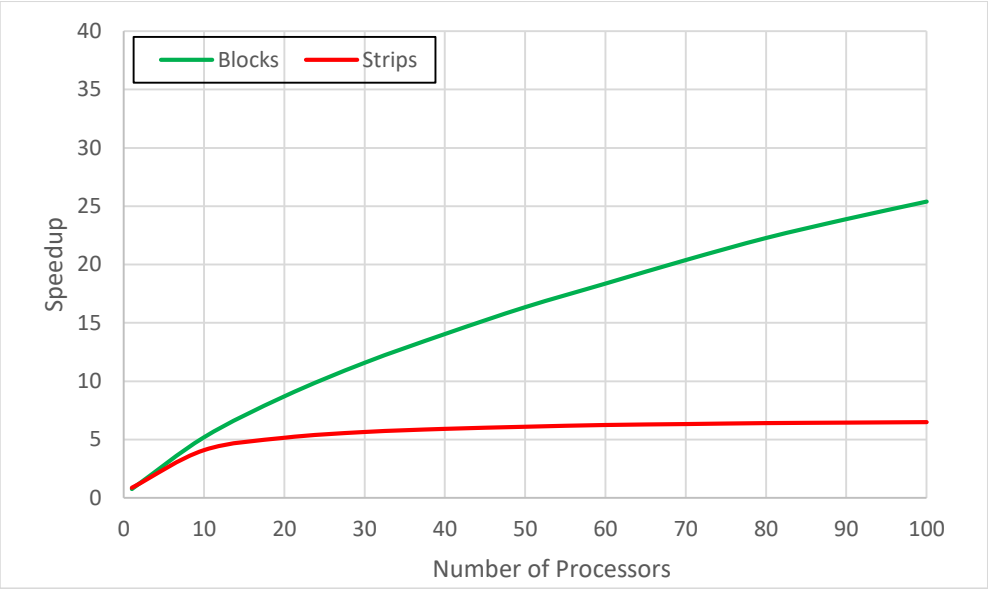


Σχήμα 3: Σύγκριση αποδοτικότητας με βάση το μέγεθος δεδομένων μεταξύ μπλοκ και λωρίδων

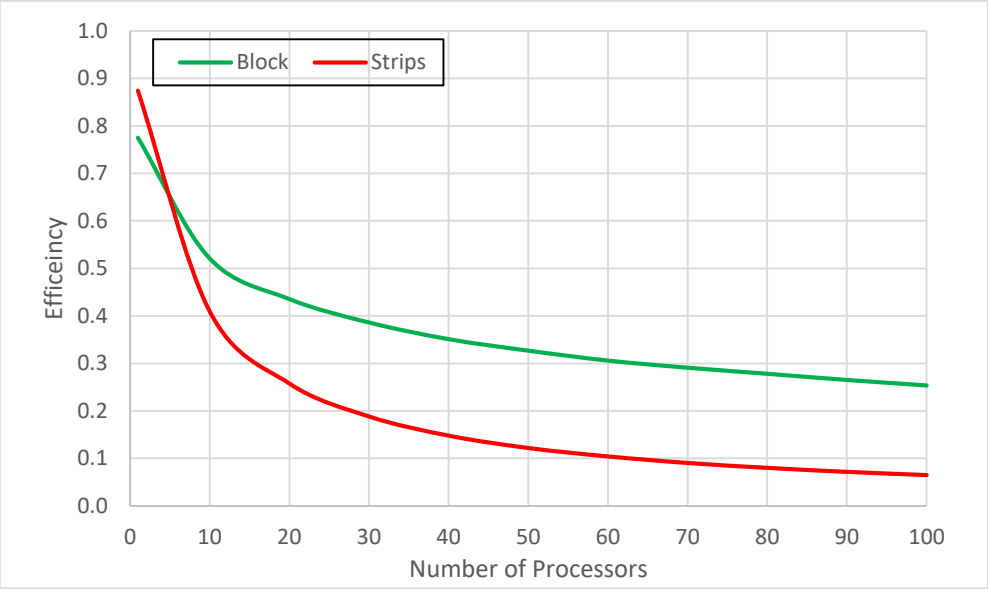
Ακολουθούν αντίστοιχα διαγράμματα στα οποία διατηρείται σταθερό το μέγεθος του πίνακα 1000×1000 και μεταβάλλεται ο αριθμός των επεξεργαστών.



Σχήμα 4: Σύγκριση χρόνου εκτέλεσης με βάση το πλήθος των επεξεργαστών μεταξύ μπλοκ και λωρίδων



Σχήμα 5: Σύγκριση επιτάχυνσης με βάση το πλήθος των επεξεργαστών μεταξύ μπλοκ και λωρίδων



Σχήμα 6: Σύγκριση αποδοτικότητας με βάση το πλήθος των επεξεργαστών μεταξύ μπλοκ και λωρίδων

Όλα τα διαγράμματα συγκλίνουν στο συμπέρασμα ότι καλύτερη επιλογή είναι ο διαχωρισμός του προβλήματος σε blocks.

2.4 Mapping

Υπάρχουν δύο γενικές στρατηγικές στο mapping:

- Οι λειτουργίες που μπορούν να εκτελεστούν ταυτόχρονα τοποθετούνται σε διαφορετικούς επεξεργαστές, ώστε να μπορούμε να εκμεταλλευτούμε αυτή τη δυνατότητα

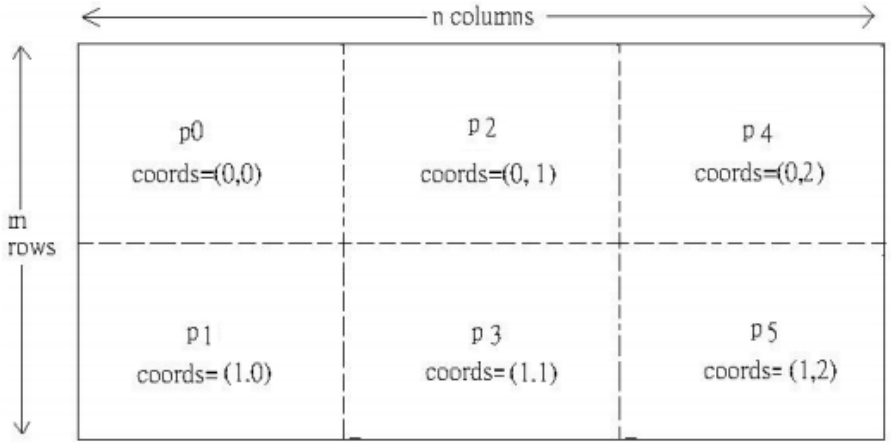
- Οι διεργασίες που επικοινωνούν συχνά μεταξύ τους, τοποθετούνται στον ίδιο κόμβο για να αποφεύγεται όσο είναι δυνατό η interprocessor επικοινωνία

Στην υλοποίησή μας κάθε διεργασία παίρνει το δικό της επεξεργαστή γιατί λαμβάνει ένα τμήμα του πίνακα και μπορεί να εκτελεστεί ταυτόχρονα από τις άλλες. Παράλληλα, στόχος μας είναι να φέρουμε όσο το δυνατόν πιο κοντά (ιδανικά στον ίδιο κόμβο) τις διεργασίες που επικοινωνούν, γνωρίζοντας ότι κάθε μία έχει τέσσερις σταθερούς γείτονες. Για να το πετύχουμε αυτό, θα εφαρμόσουμε καρτεσιανή τοπολογία.

3. ΣΧΕΔΙΑΣΜΟΣ ΚΩΔΙΚΑ

3.1 Σχεδιασμός MPI προγράμματος

Σύμφωνα και με όσα αποδείχτηκαν στην Ενότητα 2.3.3, ο ορθότερος ως προς την κλιμάκωση διαχωρισμός είναι σε blocks δύο διαστάσεων. Κάθε διεργασία θα αναλάβει έναν υποπίνακα. Για την επικοινωνία θα δημιουργήσουμε ένα δισδιάστατο καρτεσιανό πλέγμα διεργασιών στη μορφή που φαίνεται στην Εικόνα 3. Συγκεκριμένα, η πρώτη διεργασία θα πάρει το πάνω αριστερά κομμάτι και η ανάθεση θα συνεχιστεί πρώτα προς τα κάτω και μετα προς τα δεξιά. Η πρώτη βασική διαφοροποίηση με το δοσμένο πρόγραμμα είναι ότι θα χρησιμοποιήσουμε και τη master διεργασία για υπολογισμούς. Για αυτό η αρίθμιση στην Εικόνα 3 ξεκινάει απο το μηδέν. Κάθε διεργασία θα επικοινωνεί με τους τέσσερις γείτονες της (βοράς-νότος-ανατολή-δύση), όπως ορίζονται από το πλέγμα. Αν κάποιος γείτονας δεν υπάρχει (π.χ. ο p0 δεν έχει βορά και δύση στο παράδειγμά μας), θα τίθεται ίσος με MPI_PROC_NULL.

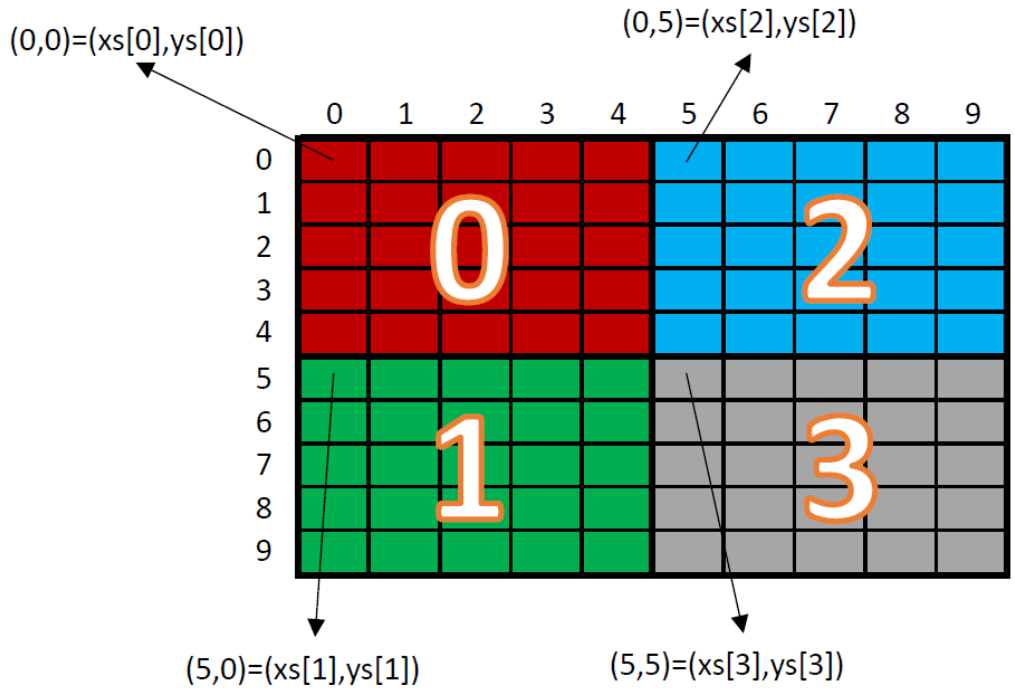


Εικόνα 3: Δισδιάστατο καρτεσιανό πλέγμα διεργασιών

Ο πίνακας πάνω στον οποίο δουλεύουμε είναι ο u, όπως και στο δοσμένο πρόγραμμα. Ο u είναι ένας τρισδιάστατος πίνακας. Ουσιαστικά είναι δύο δισδιάστατοι πίνακες ενωμένοι σε έναν τρισδιάστατο για να μπορούμε να ανταλλάσουμε τους δείκτες τους. Το μέγεθος καθενός από τους δύο δισδιάστατους πίνακες ισούται με το κομμάτι που του αντιστοιχεί απο τον ολικό πίνακα του προβλήματος που είναι ίσος με $\mathrm{NXPROB} \times \mathrm{NYPROB}$. Το πλήθος των blocks στα οποία θα χωρίσουμε τον πίνακα ισούται με $GRIDX \times GRIDY$. Σε κάθε υποπίνακα προσθέτουμε 2 γραμμές και 2 στήλες για τα halo points. Συνεπώς, το μέγεθος του καθενός από τους δύο δισδιάστατους πίνακες του u ειναι $(\mathrm{NXPROB}/\ \mathrm{GRIDX}\ + 2) \times (\mathrm{NYPROB}/\mathrm{GRIDY}\ + 2)$.

Παρόλο που ο μεγάλος πίνακας $NXPROB \times NYPROB$, δεν εμφανίζεται πουθενά στο πρόγραμμα, θα πρέπει να προσδιορίσουμε τη θέση κάθε υποπίνακα που χειρίζεται κάθε διεργασία στον ολικό πίνακα, ώστε αφενός να μπορούμε να αρχικοποιήσουμε τους υποπίνακες και αφετέρου να μπορέσουμε να παράξουμε τα dat αρχεία initial και final. Για αυτόν τον προσδιορισμό θα υπολογίσουμε και θα μοιράσουμε σε όλες τις διεργασίες τους πίνακες xs και ys, στη θέση i των οποίων βρίσκεται η συνεταγμένη x και y αντίστοιχα του πάνω αριστερού σημείου της i-οστής διεργασίας, όπως αυτή καθορίζεται από τη συνάρτηση MPI_Comm_rank.

Η Εικόνα 4 ξεδιαλύνει όσα αναλυτικά εξηγήθηκαν στις προηγούμενες δύο παραγράφους. Έστω παράδειγμα με πίνακα 10×10 που χωρίζεται σε 4 blocks, 2 σε κάθε διάσταση, καθένα εκ των οποίων έχει διάσταση 5×5. Έτσι, κάθε πίνακας u που αναλαμβάνει μια διεργασία είναι μεγέθους $(10/2 + 2) \times (10/2 + 2) = 7 \times 7$.



Εικόνα 4: Πίνακας u και διαχωρισμός του σε blocks

Για την ανταλλαγή γραμμών και στηλών μεταξύ των διεργασιών δημιουργούμε τους δύο MPI τύπους row και column, ώστε να στέλνουμε γραμμές και στήλες αντίστοιχα. Για τον τύπο row χρησιμοποιούμε τη συνάρτηση MPI_Type_contiguous που επιτρέπει την αντιγραφή συνεχόμενων δεδομένων ενός τύπου. Για τον τύπο column χρησιμοποιούμε τη συνάρτηση MPI_Type_vector που είναι ένας πιο γενικός datatype constructor που επιτρέπει την αντιγραφή blocks δεδομένων που ισαπέχουν μεταξύ τους. Τα στοιχεία μιας στήλης ενός υποπίνακα απέχουν μεταξύ τους κατά το μέγεθος μίας γραμμής του πίνακα u που είναι $(NYPROB/GRIDY + 2)$, όσες δηλαδή και οι στήλες του. Το μέγεθος του block δεδομένων είναι 1, αφού χρειαζόμαστε ένα στοιχείο από κάθε γραμμή, ενώ χρειαζόμαστε $(NXPROB/GRIDX)$ τέτοια blocks, όσες δηλαδή και οι γραμμές του κάθε υποπίνακα.

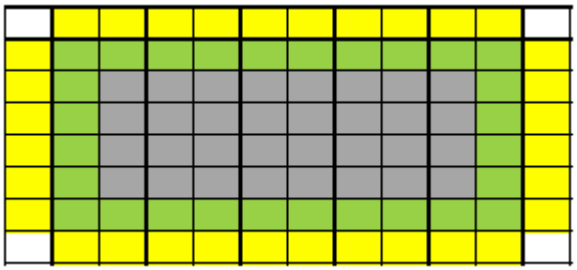
Κάθε διεργασία αρχικοποιεί το δικό της πίνακα σαν να είναι τμήμα του μεγαλύτερου πίνακα σύμφωνα με τον κώδικα της initdat από το αρχικό πρόγραμμα. Όπως προαναφέρθηκε, η κάθε διεργασία μπορεί να προσδιορίζει τα όρια της μέσω των πινάκων xs και ys.

Η κεντρική επανάληψη που χρονομετρούμε και συγκρίνουμε στα πλαίσια της άσκησης βασίζεται στη λογική του ψευδοκώδικα που δόθηκε στις οδηγίες σχεδιασμού, ο οποίος ακολουθεί:

```
MPI_Barrier (συγχρονισμός διεργασιών πριν τις μετρήσεις)
Start MPI_Wtime
For #επαναλήψεων
        Isend (SRequest) X 4 (B,N,Δ,A)
        Irecv (RRequest) X 4 (B,N,Δ,A)
```

```
            Υπολογισμός εσωτερικών στοιχείων «μετά» (γκρι στοιχεία στο σχήμα)
        Wait (RRequest) X 4 (B,N,Δ,A) ή WaitAll (array of RRequests)
            Υπολογισμός εξωτερικών στοιχείων «μετά> (πράσινα στο σχήμα)
            Πριν Πίνακας = Μετά Πίνακας
            (reduce για σύγκλιση εδώ)
        Wait(SRequest) X 4 (B,N,Δ,A) ή WaitAll (array of SRequests)
End for
End MPI_Wtime
```

Στην Εικόνα 5 φαίνονται τα χρωματιστά σημεία που αναφέρονται στον ψευδοκώδικα και αντιστοιχούν σε έναν υποπίνακα που αναλαμβάνει μια διεργασία. Τα κίτρινα σημεία είναι τα halo points όπου αποθηκεύονται οι γραμμές και οι στήλες που στέλνονται από τους γείτονες. Τα πράσινα σημεία είναι τα εξωτερικά στοιχεία του υποπίνακα που έχουν κάποιον γείτονα που ανήκει σε άλλη διεργασία, οπότε ο υπολογισμός της νέας τους τιμής πρέπει να γίνει μετά τη λήψη στοιχείων από τους γείτονες. Τα γκρι σημεία είναι τα εσωτερικά στοιχεία του υποπίνακα που όλοι τους οι γείτονες ανήκουν στην ίδια διεργασία και ο υπολογισμός της νέας τους τιμής είναι ανεξάρτητες από ανταλλαγές με γείτονες. Τα λευκά σημεία δεν παίζουν κανέναν ρόλο γιατί δεν είναι γείτονες κανενός σημείου που ανήκει στον υποπίνακα.



Εικόνα 5: Εσωτερικά, εξωτερικά και αλώ σημεία υποπίνακα

Εν γένει, η επικοινωνία είναι σχετικά αργή εξαιτίας του δικτύου επικοινωνίας και της εξάρτησης από τη διεργασία-παραλήπτη. Με την κλασική blocking επικοινωνία (MPI_Send, MPI_Recv), η διεργασία πρέπει να περιμένει αδρανής όσο διαρκεί η επικοινωνία. Με την non-blocking επικοινωνία, το MPI παρέχει τη δυνατότητα επικάλυψης της επικοινωνίας με πράξεις, αν και απαιτείται ο επιπρόσθετος έλεγχος για το αν έχει ολοκληρωθεί η επικοινωνία. Κάθε φορά που διενεργείται μια επικοινωνία δημιουργείται ένα αντικείμενο επικοινωνίας που εκτός από το ίδιο το μήνυμα περιέχει και άλλες πληροφορίες, όπως ο παραλήπτης, ο τύπος του μηνύματος, ο παραλήπτης, το tag κλπ. Όταν μια διεργασία έχει σταθερούς γείτονες, τα πεδία αυτά είναι τα ίδια και το μόνο που αλλάζει είναι το μήνυμα καθαυτό. Δεν είναι αναγκαίο να δημιουργούμε συνέχεια το ίδιο αντικείμενο με τα ίδια πεδία εκτός από το μήνυμα. Καλύτερη πρακτική θα ήταν να δημιουργούμε ένα ανά γείτονα και να το επαναχρησιμοποιούμε όσες φορές χρειαστεί. Αυτό το πλεονέκτημα μας το δίνει η persistent επικοινωνία, η λογική της είναι σαν να δημιουργούμε ένα μόνιμο μικρό «μισό» κανάλι επικοινωνίας στο οποίο μπορούμε να στέλνουμε πάντα τον ίδιο τύπο μηνύματος στο ίδιο παραλήπτη. Το κανάλι αυτό χαρακτηρίστηκε ως μισό, διότι ούτε η διεγασία δε λαμβάνει από το γείτονα της στο ίδιο κανάλι ούτε είναι απαραίτητο ότι ο γείτονας της θα χρησιμοποιεί persistent επικοινωνία.

Η πρώτη σημαντική διαφοροποίηση σε σχέση με τον ψευδοκώδικα είναι ότι αντί για non-blocking επικοινωνία (MPI_Isend, MPI_Irecv) χρησιμοποιούμε persistent επικοινωνία (MPI_Send_init, MPI_Recv_init, MPI_Startall). Στον κώδικα δημιουργούμε ένα κανάλι επικοινωνίας για send και ένα για receive για κάθε έναν από τους τέσσερις γείτονες και

με τις εντολές MPI_Startall και MPI_Waitall χειριζόμαστε πλέον την persistent επικοινωνία. Επειδή, όμως, στην κεντρική επανάληψη χειριζόμαστε εναλλάξ τις u[0] και u[1], χρειαζόμαστε ακριβώς τα διπλάσια κανάλια.

Όσον αφορά το κομμάτι της σύγκλισης, η σχεδιαστική επιλογή ήταν να γίνει ο υπολογισμός των τετραγωνικών διαφορών μέσα σε διπλό for γιατί πειραματικά αποδείχτηκε καλύτερη λύση από τον υπολογισμό στα ήδη υπάρχοντα for. Για τον υπολογισμό της ολικής τετραγωνικής διαφοράς γίνεται χρήση της MPI_Allreduce γιατί όλες οι διεργασίες χρειάζονται το αποτέλεσμά της.

Ο χρόνος εκτέλεσης του προγράμματος δίνεται από το μέγιστο χρόνο που θα κάνει μια διεργασία να ολοκληρώσει τους υπολογισμούς της. Η ίδια λογική έχει χρησιμοποιηθεί και στη μέτρηση του αρχικού προγράμματος, ώστε η σύγκριση που θα γίνει να είναι ισότιμη.

Τέλος, έχει χρησιμοποιηθεί parallel I/O για την εκτύπωση των dat αρχείων initial και final, που περιέχουν τα δεδομένα στην αρχή και μετά το πέρας του προγράμματος. Το αρχείο που παράγεται γράφοντας ταυτόχρονα όλες οι διεργασίες είναι σε binary μοφή, οπότε ο master το ξαναδιαβάζει και το γράφει με float αριθμούς που καταλαβαίνουμε.

3.2 Σχεδιασμός υβριδικού προγράμματος

Για την υλοποίηση του υβριδικού κώδικα MPI και OpenMP χρησιμοποιήθηκε ως βάση ο ίδιος κώδικας με το MPI. Κάθε MPI διεργασία για τον υπολογισμό του δικού της μέρους του πίνακα, θα χρησιμοποιεί OpenMP. Η προσέγγιση που χρησιμοποιούμε είναι η παραλληλοποίηση στα δύο ακολουθιακά τμήματα στην κεντρική επανάληψη, αυτό του υπολογισμού των νέων εσωτερικών σημείων και αυτό του υπολογισμού των νέων εξωτερικών σημείων.

Στην περίπτωση του ελέγχου σύγκλισης, χρησιμοποιούμε openmp-reduce μεταξύ των νημάτων για τον υπολογισμό τον τοπικών τετραγωνικών διαφορών και στη συνέχεια MPI_Allreduce για να υπολογίσουμε την ολική τετραγωνική διαφορά.

Για να γλιτώσουμε τη δημιουργία και καταστροφή των threads σε κάθε επανάληψη, επιλέξαμε να τα δημιουργήσουμε μια φορά έξω από την επανάληψη και να τρέχουμε παράλληλα τα for με τα ήδη δημιουργημένα threads. Στα διπλά for χρησιμοποιούμε το collapse(2), που συγχωνεύει τις δύο επαναλήψεις σε μία μεγάλη και τη διαχωρίζει ανάλογα με το schedule που έχει δηλωθεί.

Σημαντικό είναι να αναφερθεί ότι ενώ οι επεξεργαστές έχουν κοινή μνήμη δεν έχουν κοινή cache. Αυτό σημαίνει ότι πρέπει να υπάρξει κατάλληλος χειρισμός ώστε κάθε μπλοκ μνήμης που μεταβαίνει στην cache και εγγράφεται, δεν πρέπει να βρίσκεται σε άλλο επεξεργαστή καθώς θα υπάρχει η ανάγκη ενημέρωσης όλων των cache με κάθε αλλαγή το οποίο προκαλεί καθυστέρηση στην εκτέλεση. Άρα, ενώ για τον πίνακα από τον οποίο διαβάζονται τα δεδομένα δεν υπάρχει πρόβλημα, για τον δεύτερο όπου εγγράφονται τα αποτελέσματα μπορεί να υπάρξει πρόβλημα. Οπότε, το τρέχον στοιχείο του πίνακα στο οποίο γίνεται επεξεργασία από έναν επεξεργαστή, πρέπει να μην βρίσκεται σε μπλοκ μνήμης όπου εκεί βρίσκεται το τρέχον στοιχείο ενός άλλου πίνακα. Συνεπώς, αρκεί ο διαμοιρασμός να έχει ελάχιστη απόσταση ενός block μνήμης.

3.3 Σχεδιασμός CUDA προγράμματος

Σε αυτήν την περίπτωση τα πράγματα είναι αρκετά απλούστερα από την άποψη ότι δεν έχουμε να συγχρονίσουμε τη συνεργασία και την επικοινωνία μεταξύ πολλαπλών διεργασιών, αλλά τρέχουμε μόνο μία διεργασία. Η δυσκολία που υπάρχει εδώ είναι να ελέγξουμε τα blocks και τα threads, ώστε να μην έχουν πρόσβαση στο ίδιο σημείο

ταυτόχρονα, ενώ πρέπει να προσέξουμε να μην έχουμε πρόσβαση σε μνήμη που δεν ανήκει στον πίνακα που θέλουμε να προσπελάσουμε, καθώς δεν είναι εφικτό να ανοίξουμε ακριβώς όσα blocks και threads χρειάζονται.

Στις προηγούμενες περιπτώσεις του MPI και του OpenMP, ο τρισδιάστατος πίνακας u είναι ουσιαστικά δύο δισδιάστατοι $\mathrm{NXPROB} \times \mathrm{NYPROB}$. Στην CUDA συνηθίζεται να μη χρησιμοποιούμε πολυδιάστατους πίνακες, αλλά μονοδιάστατους. Ο τρόπος που χρησιμοποιούμε τον u είναι να τον χωρίσουμε σε u0 και u1, οι οποίοι, αντί για δισδιάστατοι, είναι μονοδιάστατοι με ίδιο μήκος $\mathrm{NXPROB} \times \mathrm{NYPROB}$. Αφού πλέον δεν είναι ένας πίνακας ο u, αλλά είναι δύο, δε μπορούμε να ανταλλάζουμε τους δείκτες, όπως κάναμε στις προηγούμενες υλοποιήσεις. Στην Εικόνα 6 φαίνεται η προσέγγιση που έχουμε ακολουθήσει ώστε να λύσουμε αυτό το πρόβλημα. Ουσιαστικά εκτελούμε τις μισές επαναλήψεις (αυξάνουμε κατά το 2 το δείκτη του for σε κάθε επανάληψη) και σε αυτές κάνουμε update πρώτα το u1 και μετά το u0. Έτσι, αν έχουμε ζυγό αριθμό βημάτων, η απάντηση είναι στο u0, ενώ αν έχουμε μονό αριθμό απαντήσεων η απάντηση είναι στο u1 και έχουμε εκτελέσει ένα παραπάνω αχρείαστο update που δεν επηρεάζει το αποτέλεσμα.

```
for (k=0; k<STEPS; k=k+2) {
    update<<<dimGrid, dimBlock>>>(u1, u0);
    update<<<dimGrid, dimBlock>>>(u0, u1);
}
```

Εικόνα 6: Υπολογισμός μεταφοράς θερμότητας σε CUDA

Δοκιμάστηκε ακόμα να έχουμε έναν πίνακα που θα ενσωματώνει τους u0 και u1, είτε δισδιάστατο δεσμευμένο με cudaMallocPitch είτε στη σειρά τον έναν μετά τον άλλο και προσθήκη στο indexing τού πίνακα το $iz \times (\mathrm{NXPROB} \times \mathrm{NYPROB})$, ώστε να προσπελαύνουμε όποιο από τα u0 ή u1 θέλουμε. Σε αυτήν την περίπτωση θα θέλαμε την kernel συνάρτηση update την οποία θα ακολουθούσε μια εντολή cudaDeviceSynchronize και μετά θα κάναμε την αλλαγή του δείκτη $iz = 1 - \mathrm{iz}$, καθώς οι host και οι device πράξεις δε συγχρονίζονται μεταξύ τους. Αντίθετα, στη λογική που ακολουθήσαμε, η επανάληψη περιέχει μόνο device συναρτήσεις που εκτελούνται η μια μετά την άλλη συγχρονισμένα, οπότε αποφεύγουμε το cudaDeviceSynchronize. Πειραματικά, αποδείχτηκε ότι το κόστος τού cudaDeviceSynchronize είναι σημαντικό και ότι η λύση που παρουσιάστηκε στην προηγούμενη παράγραφο είναι η καλύτερη από άποψη χρόνου.

Για την υλοποίηση μας χρησιμοποιήσαμε δισδιάστατο block. Πειραματικά βρήκαμε ότι το 8x8 δίνει τους γρηγορότερους χρόνους. Παράλληλα, έγινε προσπάθεια ένταξης shared memory στη συνάρτηση update, γνωρίζοντας ότι η πρόσβαση σε αυτήν είναι σημαντικά πιο γρήγορη από την πρόσβαση στην global memory. Το πρόβλημα σε αυτήν περίπτωση είναι ότι η shared memory αφορά τα threads που αντιστοιχούν στο ίδιο block και δεν υπάρχει τρόπος να έχουμε πρόσβαση σε αυτήν από άλλο block. Εμείς δεν έχουμε τρόπο να εξασφαλίσουμε ότι και οι 4 γείτονες ενός σημείου θα είναι στο ίδιο block, οπότε δεν καταφέραμε να την αξιοποιήσουμε.

4. ΜΕΤΡΗΣΕΙΣ

Οι μετρήσεις αφορούν 1000 επαναλήψεις του προγράμματος. Στις περιπτώσεις που τρέχουμε με έλεγχο σύγκλισης, αυτός γίνεται κάθε 20 επαναλήψεις. Τα μεγέθη των δεδομένων και ο συνδυασμός υπολογιστικών κόμβων και των διεργασιών είναι προκαθορισμένα από την εκφώνηση της άσκησης. Συγκεκριμένα, όσον αφορά τα μεγέθη των δεδομένων, ξεκινάμε από 80x64 και κάθε φορά διπλασιάζουμε και τις δύο πλευρές σε κάθε επόμενο πείραμα, σταματώντας εκεί που μας επιτρέπει το marie.hellasgrid.gr. Το ανώτατο μέγεθος δεδομένων που μπορέσαμε να τρέξουμε επιτυχώς το πρόγραμμα ήταν 2560x2048. Όσον αφορά το συνδυασμό υπολογιστικών κόμβων και διεργασιών, χρησιμοποιήσαμε αυτούς που δίνονται στην εκφώνηση και διαιρούν ακριβώς τις πλευρές. Ούτως ή άλλως ήταν ήδη πολλά σε πλήθος τα πειράματα για να χρησιμοποιήσουμε και τα μεγέθη που δε διαιρούν ακριβώς την πλευρά. Όλοι οι χρόνοι μετρώνται σε δευτερόλεπτα και οι μετρήσεις έχουν γίνει στο marie.hellasgrid.gr.

4.1 MPI πρόγραμμα χωρίς έλεγχο σύγκλισης

Στους Πίνακες 1, 2 και 3 ακολουθούν οι μετρήσεις για χρόνο, επιτάχυνση και αποδοτικότητα του MPI προγράμματος που σχεδιάστηκε στα πλαίσια της άσκησης χωρίς έλεγχο σύγκλισης.

Πίνακας 1: Μετρήσεις χρόνου MPI προγράμματος χωρίς έλεγχο σύγκλισης

Nodes / Tasks (sec)						
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	2.53E-02	9.30E-03	1.24E-01	1.19E-01	2.82E-01	2.12E-01
160x128	9.87E-02	2.91E-02	2.19E-01	2.18E-01	2.04E-01	2.63E-01
320x256	7.52E-01	1.04E-01	2.33E-01	3.33E-01	3.14E-01	2.18E-01
640x512	3.01E+00	7.63E-01	2.57E-01	3.26E-01	2.52E-01	2.13E-01
1280x1024	1.27E+01	3.02E+00	9.05E-01	5.80E-01	4.12E-01	2.52E-01
2560x2048	5.09E+01	1.20E+01	3.19E+00	9.65E-01	1.54E+00	5.18E-01

Πίνακας 2: Επιτάχυνση MPI προγράμματος χωρίς έλεγχο σύγκλισης

Nodes / Tasks					
	1/4	2/16	8/64	16/128	20/160
80x64	2.72	0.20	0.21	0.09	0.12
160x128	3.39	0.45	0.45	0.48	0.38
320x256	7.23	3.23	2.26	2.39	3.45
640x512	3.94	11.71	9.23	11.94	14.13
1280x1024	4.21	14.03	21.90	30.83	50.40
2560x2048	4.24	15.96	52.75	33.05	98.26

Πίνακας 3: Αποδοτικότητα MPI προγράμματος χωρίς έλεγχο σύγκλισης

Nodes / Tasks					
	1/4	2/16	8/64	16/128	20/160
80x64	0.680	0.013	0.003	0.001	0.001
160x128	0.848	0.028	0.007	0.004	0.002
320x256	1.808	0.202	0.035	0.019	0.022
640x512	0.986	0.732	0.144	0.093	0.088
1280x1024	1.051	0.877	0.342	0.241	0.315
2560x2048	1.060	0.997	0.824	0.258	0.614

4.2 MPI πρόγραμμα με έλεγχο σύγκλισης κάθε ν επαναλήψεις

Στους Πίνακες 4, 5 και 6 ακολουθούν οι μετρήσεις για χρόνο, επιτάχυνση και αποδοτικότητα του MPI προγράμματος που σχεδιάστηκε στα πλαίσια της άσκησης με έλεγχο σύγκλισης κάθε 20 επαναλήψεις.

Πίνακας 4: Μετρήσεις χρόνου MPI προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks (sec)						
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	3.33E-02	1.30E-02	4.26E-01	2.54E-01	3.79E-01	2.06E-01
160x128	1.24E-01	3.97E-02	2.92E-01	6.44E-01	2.31E-01	2.49E-01
320x256	8.51E-01	1.33E-01	3.13E-01	5.65E-01	1.76E+00	2.29E-01
640x512	3.39E+00	8.66E-01	7.29E-01	6.72E-01	8.68E-01	2.42E-01
1280x1024	1.58E+01	3.43E+00	1.24E+00	6.61E-01	7.30E-01	2.63E-01
2560x2048	6.29E+01	1.95E+01	4.54E+00	1.53E+00	1.12E+00	4.80E-01

Πίνακας 5: Επιτάχυνση MPI προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks					
	1/4	2/16	8/64	16/128	20/160
80x64	2.56	0.08	0.13	0.09	0.16
160x128	3.12	0.42	0.19	0.54	0.50
320x256	6.40	2.72	1.51	0.48	3.72
640x512	3.91	4.65	5.04	3.91	14.01
1280x1024	4.61	12.74	23.90	21.64	60.08
2560x2048	3.23	13.85	41.11	56.16	131.04

Πίνακας 6: Αποδοτικότητα MPI προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks					
	1/4	2/16	8/64	16/128	20/160
80x64	0.640	0.005	0.002	0.001	0.001
160x128	0.781	0.027	0.003	0.004	0.003
320x256	1.600	0.170	0.024	0.004	0.023
640x512	0.979	0.291	0.079	0.031	0.088
1280x1024	1.152	0.796	0.373	0.169	0.375
2560x2048	0.806	0.866	0.642	0.439	0.819

4.3 Υβριδικό MPI με OpenMP

Στους Πίνακες 7, 8 και 9 ακολουθούν οι μετρήσεις για χρόνο, επιτάχυνση και αποδοτικότητα του υβριδικού MPI με OpenMP προγράμματος που σχεδιάστηκε στα πλαίσια της άσκησης με έλεγχο σύγκλισης κάθε 20 επαναλήψεις.

Πίνακας 7: Μετρήσεις χρόνου υβριδικού προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks / Threads (sec)															
	1/2/2	1/1/4	2/8/2	2/8/4	2/4/4	2/2/8	8/32/2	8/16/4	8/8/8	16/64/2	16/32/4	16/16/8	20/80/2	20/40/4	20/20/8
80x64	0.05	0.17	0.09	3.48	0.09	0.21	0.27	0.13	0.17	0.2	0.14	0.19	0.16	0.13	0.2
160x128	0.15	0.57	0.12	3.29	0.1	0.61	0.13	0.12	0.24	0.71	0.13	0.25	0.18	0.14	0.26
320x256	1.16	4.6	0.15	3.5	0.22	4.63	0.15	0.13	0.64	0.16	0.12	0.46	0.32	0.15	0.46
640x512	4.58	18.3	0.41	4.44	1.33	18.3	0.2	0.25	4.66	0.18	0.19	1.43	0.21	0.17	1.14
1280x1024	18.6	73.5	2.63	8.16	5.09	73.8	0.72	1.38	18.4	0.27	0.73	9.3	0.32	0.61	4.67
2560x2048	74.2	294	9.77	22.8	19.97	294.5	2.61	5.19	-	1.34	2.72	-	1.1	2.11	17.2

Πίνακας 8: Επιτάχυνση υβριδικού προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks / Threads															
	1/2/2	1/1/4	2/8/2	2/8/4	2/4/4	2/2/8	8/32/2	8/16/4	8/8/8	16/64/2	16/32/4	16/16/8	20/80/2	20/40/4	20/20/8
80x64	0.62	0.19	0.35	0.01	0.35	0.16	0.12	0.26	0.20	0.17	0.25	0.18	0.21	0.25	0.16
160x128	0.81	0.22	1.03	0.04	1.22	0.20	0.98	1.02	0.51	0.18	0.93	0.51	0.71	0.91	0.48
320x256	0.73	0.19	5.75	0.24	3.94	0.18	5.83	6.45	1.33	5.39	6.75	1.86	2.66	5.87	1.87
640x512	0.74	0.19	8.25	0.76	2.55	0.19	17.21	13.84	0.73	19.26	18.23	2.37	16.54	20.18	2.97
1280x1024	0.85	0.21	6.01	1.94	3.10	0.21	21.94	11.45	0.86	58.30	21.56	1.70	50.16	26.07	3.38
2560x2048	0.85	0.21	6.44	2.76	3.15	0.21	24.10	12.12	-	46.94	23.13	-	57.18	29.81	3.65

Πίνακας 9: Αποδοτικότητα υβριδικού προγράμματος με έλεγχο σύγκλισης κάθε v=20 επαναλήψεις

Nodes / Tasks / Threads															
	1/2/2	1/1/4	2/8/2	2/8/4	2/4/4	2/2/8	8/32/2	8/16/4	8/8/8	16/64/2	16/32/4	16/16/8	20/80/2	20/40/4	20/20/8
80x64	0.16	0.05	0.02	0.0003	0.02	0.01	0.002	0.004	0.003	0.001	0.002	0.001	0.001	0.002	0.001
160x128	0.20	0.06	0.07	0.001	0.08	0.01	0.02	0.02	0.008	0.001	0.007	0.004	0.004	0.006	0.003
320x256	0.18	0.05	0.36	0.008	0.25	0.01	0.09	0.10	0.021	0.042	0.053	0.015	0.02	0.04	0.012
640x512	0.19	0.05	0.52	0.02	0.16	0.01	0.27	0.22	0.011	0.150	0.142	0.019	0.10	0.13	0.019
1280x1024	0.21	0.05	0.38	0.06	0.19	0.01	0.34	0.18	0.013	0.455	0.168	0.013	0.31	0.16	0.021
2560x2048	0.22	0.05	0.40	0.09	0.20	0.01	0.38	0.19	-	0.367	0.181	-	0.36	0.19	0.023

4.4 CUDA

Σε αυτήν την περίπτωση είναι λίγο διαφορετικά τα πειράματα που κάναμε, αφού εδώ δεν έχουμε πολλαπλές διεργασίες που μοιράζονται τη δουλειά. Οπότε, επιλέξαμε να κάνουμε τα πειράματά μας με βάση το πλήθος των επαναλήψεων, ξεκινώντας από 10 και δεκαπλασιάζοντας κάθε φορά έως τις 100000. Τα μεγέθη των πινάκων είναι τα ίδια που

χρησιμοποιήθηκαν στα πειράματα για MPI και υβριδικό MPI με OpenMP. Στον Πίνακα 10 ακολουθούν οι μετρήσεις σε σχέση με τις επαναλήψεις που τρέξαμε.

Πίνακας 10: Μετρήσεις χρόνου CUDA προγράμματος βάσει του πλήθους των επαναλήψεων

ITERATIONS					
	10	**100**	**1000**	**10000**	**100000**
80x64	9.86E-05	9.41E-04	9.15E-03	9.10E-02	9.20E-01
160x128	2.11E-04	2.12E-03	2.19E-02	2.24E-01	2.12E+00
320x256	7.10E-04	7.10E-03	7.14E-02	6.80E-01	6.74E+00
640x512	3.34E-03	3.33E-02	3.27E-01	3.28E+00	3.29E+01
1280x1024	1.85E-02	1.83E-01	1.86E+00	1.86E+01	1.86E+02
2560x2048	7.85E-02	7.84E-01	7.84E+00	7.84E+01	7.84E+02

Στον Πίνακα 11 βλέπουμε την επιβράδυνση όσο μεγαλώνει το πλήθος των επαναλήψεων, συγκρίνοντας με τους χρόνους που βλέπουμε στην στήλη για τις 10 επαναλήψεις στον Πίνακα 10. Στην πρώτη στήλη με τις 100 επαναλήψεις, δηλαδή 10 φορές πιο πολλές από τις 10 που συγκρίνουμε, βλέπουμε ότι η επιβράδυνση είναι σε όλες τις περιπτώσεις σχεδόν 10. Αντίστοιχα, κάθε επόμενη στήλη είναι σχεδόν 10 φορές μεγαλύτερη από την προηγούμενη. Από όλα αυτά αντιλαμβανόμαστε ότι ο κυρίως αλγόριθμος που μετράμε δεν έχει θέματα συγχρονισμού και πολλαπλασιασμός του πλήθους των επαναλήψεων σημαίνει και αντίστοιχος πολλαπλασιασμός του χρόνου εκτέλεσης.

Πίνακας 11: Κλιμάκωση χρόνου σε σχέση με την αύξηση χρόνου προγράμματος CUDA

ITERATIONS				
	100 (x10)	1000 (x100)	10000 (x1000)	100000 (x10000)
80x64	9.55	92.84	922.86	9334.42
160x128	10.04	104.04	1060.49	10050.25
320x256	10.00	100.54	956.97	9491.09
640x512	9.97	97.87	981.66	9836.65
1280x1024	9.88	100.77	1008.56	10084.55
2560x2048	9.99	99.91	998.84	9991.25

5. ΣΥΓΚΡΙΣΗ ΜΕ ΑΡΧΙΚΟ ΠΡΟΓΡΑΜΜΑ

Στην ενότητα αυτή θα συγκρίνουμε την MPI υλοποίησή μας με εκείνη που μας δόθηκε σε σχέση με το χρόνο, την επιτάχυνση και την αποδοτικότητα τους.

5.1 Σύγκριση πραγματικών υπολογισμών

Στον Πίνακα 10 βρίσκονται οι χρόνοι που μετρήθηκαν εκτελώντας το δοσμένο πρόγραμμα. Η σύγκριση γίνεται με βάση τα περιεχόμενα του Πίνακα 1, ο οποίος έχει το χρόνους του δικού μας MPI προγράμματος χωρίς έλεγχο σύγκλισης.

Πίνακας 12: Μετρήσεις χρόνου αρχικού προγράμματος προγράμματος χωρίς έλεγχο σύγκλισης

Processes / Threads (sec)						
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	0.07	0.02	0.15	0.94	1.25	1.49
160x128	0.29	0.09	0.25	0.62	1.07	1.36
320x256	1.16	0.30	0.23	0.54	1.05	1.24
640x512	4.70	1.18	0.62	0.74	1.03	1.30

Στον Πίνακα 11 ακολουθεί η επιτάχυνση του νέου προγράμματος έναντι του αρχικού. Υπολογίστηκε με τον τύπο $\frac{t_{old}}{t_{new}}$, όπου t_{old}: ο χρόνος εκτέλεσης του αρχικού προγράμματος που δόθηκε και t_{new}: ο χρόνος εκτέλεσης του δικού μας προγράμματος.

Πίνακας 13: Επιτάχυνση υλοποίησή μας σε σχέση με το αρχικό πρόγραμμα

Processes / Threads						
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	2.76	2.26	1.22	7.89	4.43	7.03
160x128	2.93	3.00	1.15	2.84	5.25	5.17
320x256	1.54	2.89	1.00	1.63	3.34	5.69
640x512	1.56	1.55	2.41	2.26	4.09	6.10

Παρατηρούμε σημαντική επιτάχυνση του δικού μας προγράμματος έναντι του αρχικού σε κάθε περίπτωση. Αξιοσημείωτο είναι ότι η επιτάχυνση κλιμακώνει στις πολλές διεργασίες που είναι ένα από τους στόχους της υλοποίησής μας.

5.2 Σύγκριση αναλυτικών μετρήσεων

Στους Πίνακες 14, 15 και 16 ακολουθούν οι θεωρητικές μετρήσεις για το χρόνο, την επιτάχυνση και την αποδοτικότητα για τη λογική του δοσμένου προγράμματος που είναι διαχωρισμός σε λωρίδες. Βλέπουμε ότι οι μετρήσεις είναι σημαντικά πιο γρήγορες σε σχέση με τις πραγματικές, γεγονός που σχετίζεται με τις παραδοχές και τους χρόνους που θεωρήσαμε αμελητέους.

Πίνακας 14: Θεωρητική μέτρηση χρόνου για διαχωρισμό σε λωρίδες

	Processes / Threads (sec)					
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	4.17E-04	3.21E-04	2.97E-04	2.91E-04	2.89E-04	2.89E-04
160x128	1.09E-03	7.05E-04	6.09E-04	5.85E-04	5.81E-04	5.77E-04
320x256	3.20E-03	1.67E-03	1.28E-03	1.19E-03	1.17E-03	1.16E-03
640x512	1.05E-02	4.35E-03	2.82E-03	2.43E-03	2.37E-03	2.35E-03
1280x1024	3.74E-02	1.28E-02	6.66E-03	5.12E-03	4.87E-03	4.80E-03
2560x2048	1.40E-01	4.20E-02	1.74E-02	1.13E-02	1.02E-02	9.99E-03

Πίνακας 15: Θεωρητική μέτρηση επιτάχυνσης για διαχωρισμό σε λωρίδες

Processes / Threads					
	1/4	2/16	8/64	16/128	20/160
80x64	1.30	1.40	1.43	1.44	1.44
160x128	1.54	1.79	1.86	1.87	1.89
320x256	1.92	2.50	2.70	2.74	2.76
640x512	2.41	3.73	4.31	4.43	4.46
1280x1024	2.92	5.61	7.30	7.68	7.78
2560x2048	3.34	8.06	12.45	13.70	14.05

Πίνακας 16: Θεωρητική μέτρηση αποδοτικότητας για διαχωρισμό σε λωρίδες

Processes / Threads					
	1/4	2/16	8/64	16/128	20/160
80x64	0.325	0.088	0.022	0.011	0.009
160x128	0.386	0.112	0.029	0.015	0.012
320x256	0.481	0.156	0.042	0.021	0.017
640x512	0.603	0.233	0.067	0.035	0.028
1280x1024	0.730	0.351	0.114	0.060	0.049
2560x2048	0.835	0.504	0.195	0.107	0.088

Στους Πίνακες 17, 18 και 19 ακολουθούν οι θεωρητικές μετρήσεις για το χρόνο, την επιτάχυνση και την αποδοτικότητα για τη λογική του δικού μας προγράμματος που είναι διαχωρισμός σε μπλοκ. Σε αυτήν την περίπτωση παρατηρούμε πιο λογικές διαφορές.

Είναι αναμενόμενο να έχουμε διαφορές μεταξύ ενός θεωρητικού μοντέλου και της πρακτικής του υλοποίησης.

Πίνακας 17: Θεωρητική μέτρηση χρόνου για διαχωρισμό σε μπλοκ

Processes / Threads (sec)						
	1/1	1/4	2/16	8/64	16/128	20/160
80x64	1.91E-02	4.91E-03	1.30E-03	3.61E-04	1.76E-04	1.55E-04
160x128	7.53E-02	1.91E-02	4.91E-03	1.30E-03	6.55E-04	5.21E-04
320x256	2.99E-01	7.53E-02	1.91E-02	4.91E-03	2.42E-03	1.98E-03
640x512	1.19E+00	2.99E-01	7.53E-02	1.91E-02	9.51E-03	7.58E-03
1280x1024	4.76E+00	1.19E+00	2.99E-01	7.53E-02	3.76E-02	3.00E-02
2560x2048	1.90E+01	4.76E+00	1.19E+00	2.99E-01	1.50E-01	1.19E-01

Πίνακας 18: Θεωρητική μέτρηση επιτάχυνσης για διαχωρισμό σε μπλοκ

Processes / Threads					
	1/4	2/16	8/64	16/128	20/160
80x64	3.89	14.72	52.84	108.21	123.43
160x128	3.94	15.34	58.07	114.96	144.57
320x256	3.97	15.67	60.94	123.56	150.91
640x512	3.99	15.83	62.45	125.38	157.21
1280x1024	3.99	15.92	63.22	126.56	158.91
2560x2048	4.00	15.96	63.61	127.01	159.56

Πίνακας 19: Θεωρητική μέτρηση αποδοτικότητας για διαχωρισμό σε μπλοκ

Processes / Threads					
	1/4	2/16	8/64	16/128	20/160
80x64	0.972	0.920	0.826	0.845	0.771
160x128	0.986	0.959	0.907	0.898	0.904
320x256	0.993	0.979	0.952	0.965	0.943
640x512	0.996	0.990	0.976	0.980	0.983
1280x1024	0.998	0.995	0.988	0.989	0.993
2560x2048	0.999	0.997	0.994	0.992	0.997

Η σύγκριση των θεωρητικών μετρήσεων δείχνει ότι σε αυτά τα μεγέθη δεδομένων ίσως είναι λίγο ταχύτερη από άποψη χρόνου η προσέγγιση σε λωρίδες, αλλά η προσέγγιση σε μπλοκ κλιμακώνει εμφατικά καλύτερα, πράγμα που φαίνεται και από τους πίνακες για επιτάχυνση και αποδοτικότητα. Τα μεγέθη είναι μικρά γιατί μας περιορίζει το σύστημα στο οποίο μετρήσαμε. Είναι βέβαιο ότι αν μπορούσαμε να δοκιμάσουμε μεγαλύτερα μεγέθη δεδομένων, η εικόνα του πίνακα χρόνων θα ήταν αντίστοιχη εκείνων της επιτάχυνσης και της αποδοτικότητας.

6. ΣΥΜΠΕΡΑΣΜΑΤΑ – ΠΑΡΑΤΗΡΗΣΕΙΣ – ΜΕΛΛΟΝΤΙΚΕΣ ΕΠΕΚΤΑΣΕΙΣ

Το γενικό συμπέρασμα είναι πως η ενσωμάτωση μεθόδων παραλληλίας στις σημερινές εφαρμογές κρίνεται αναγκαία, δεδομένης της συνεχούς αύξησης του μεγέθους των δεδομένων προς επεξεργασία, και της πολυπλοκότητας των λειτουργιών που χρειάζεται να πραγματοποιηθούν πάνω σε αυτά.

Παρατηρούμε ότι οι διαφορές μεταξύ των χρόνων εκτέλεσης των δυο υλοποιήσεων (MPI και υβριδικό MPI με OpenMP) είναι πολύ μικρές. Άλλες φορές εμφανίζεται να υπερέχει για λίγο η MPI υλοποίηση και άλλες η υβριδική. Άρα αυτό που μπορούμε να συμπεράνουμε είναι ότι η επιπρόσθετη προσπάθεια για την εισαγωγή του OpenMP κώδικα δεν προσέφερε κάτι ουσιαστικό.

Σχετικά με τις μετρήσεις σε CUDA, στα πρώτα τρία μεγέθη πινάκων (80x64, 160x128, 320x256), η CUDA είνα γρηγορότερη από την καλύτερη επίδοση που πετύχαμε με MPI και υβριδικό MPI με OpenMP ανέξαρτητα από nodes, processes και threads. Στο μέγεθος 640x512 βλέπουμε ότι CUDA με MPI και υβριδικό MPI με OpenMP είναι περίπου στο ίδιο επίπεδο για τις περιπτώσεις των γρήγορων εκτελέσεων του MPI με ή χωρίς OpenMP, ενώ στις αργές υπερτερεί σημαντικά η CUDA. Στις περιπτώσεις των δύο τελευταίων μεγεθών (1280x1024, 2560x2048) η CUDA είναι γενικά καλύτερη από το MPI στις λίγες διεργασίες/tasks (μέχρι 2/16), ενώ το MPI υπερτερεί σημαντικά στις πολλές (8/64 και πάνω). Παρόμοια συμπεριφορά παρουσιάζει και το υβριδικό πρόγραμμα, που είναι πιο γρήγορο σε σχέση με την CUDA στις περισσότερες εκδοχές του για πάνω από 8 nodes, ενώ είναι πιο αργό από την CUDA σε όλους τους συνδυασμούς που χρησιμοποιούν 1 ή 2 nodes. Το συμπέρασμα εδώ είναι ότι η CUDA είναι ένας καλός τρόπος να επιταχύνει κάποιος σημαντικά ένα πρόγραμμα αν έχεις ένα μόνο μηχάνημα, αλλά όσο μεγαλώνει η διάσταση του προβλήματος και τα διαθέσιμα μηχανήματα, το MPI κλιμακώνει καλύτερα και πετυχαίνει γρηγορότερα αποτελέσματα.

Θα είχε ενδιαφέρον να υπήρχαν να μπορούσαμε να τεστάρουμε το πρόγραμμά μας με μεγαλύτερα δεδομένα. Τα 2Gb ως όριο μνήμης δεν είναι επαρκές, αν υπολογίσουμε ότι χρειαζόμαστε επιπρόσθετες γραμμές και στήλες ως halo points, πίνακα που να κρατάει τα όρια του κάθε block, δεσμεύσεις τύπων row και column για επικοινωνία μεταξυ των blocks, δεσμεύσεις τύπου subarray για parallel I/O, πίνακες για να κρατάμε persistent communication κ.α., γεγονός που μας περιορίζει το μέγεθος των δεδομένων που μπορούμε να δεσμεύσουμε. Παράλληλα, όσο μεγαλύτερα είναι τα δεδομένα, τόσο καλύτερα κλιμακώνει το πρόγραμά μας και τόσο καλύτερα μπορούν να μας βγουν τα αποτελέσματα αποδοτικότητας και επιτάχυνσης. Εδώ δεν είναι ιδανικά, κάτι που οφείλεται σε μεγάλο βαθμό στο μέγεθος των δεδομένων. Εξίσου σημαντικό θα ήταν να μπορούσαμε να τεστάρουμε και την CUDA υλοποίηση μας σε κοινά μηχανήματα γιατί οι χρόνοι μπορεί να ποικίλουν ανάλογα με την κάρτα γραφικών και το compute capability της GPU.

7. ΕΞΟΔΟΣ mpiP

Η έξοδος που ακολουθεί αφορά το MPI πρόγραμμα με είσοδο μεγέθους 640×512 χωρίς έλεγχο σύγκλισης για 1000 επαναλήψεις. Το πρόγραμμα τρέχει σε 2 κόμβους με 8 διεργασίες έκαστος. Η εντολή για compile είναι mpicc -O3 -g -Wall mpi_heat.c -L$MPIPROOT/lib -lmpiP -lbfd -liberty -lunwind -o mpi_heat.x. Επειδή η έξοδος είναι πολύ μεγάλη, παραθέτω ένα τμήμα της. Ολόκληρο το αρχείο με όνομα mpi_heat.mpiP υπάρχει στον φάκελο που παραδίδεται.

```
@ mpiP
@ Command : grad1612_mpi_heat.x -c
@ Version                  : 3.4.1
@ MPIP Build date          : Feb  6 2018, 18:24:04
@ Start time               : 2019 02 20 12:46:02
@ Stop time                : 2019 02 20 12:46:03
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 15678
@ Final Output Dir         : .
@ Report generation        : Collective
@ MPI Task Assignment      : 0 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 1 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 2 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 3 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 4 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 5 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 6 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 7 wn006.marie.hellasgrid.gr
@ MPI Task Assignment      : 8 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 9 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 10 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 11 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 12 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 13 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 14 wn007.marie.hellasgrid.gr
@ MPI Task Assignment      : 15 wn007.marie.hellasgrid.gr

---------------------------------------------------------------------------
```

@--- MPI Time (seconds) ----------

Task	AppTime	MPITime	MPI%
0	0.994	0.892	89.82
1	0.994	0.892	89.82
2	0.994	0.892	89.82
3	0.993	0.853	85.82
4	0.994	0.892	89.78
5	0.994	0.892	89.81
6	0.993	0.892	89.79
7	1.01	0.904	89.90
8	0.993	0.887	89.28
9	0.993	0.889	89.52
10	0.993	0.889	89.51
11	0.993	0.889	89.50
12	0.993	0.887	89.32
13	0.993	0.888	89.44
14	0.993	0.89	89.60
15	1	0.899	89.60
*	15.9	14.2	89.40

@--- Callsites: 40 ----------

ID	Lev	File/Address	Line	Parent_Funct	MPI_Call
1	0	grad1612_mpi_heat.c	76	main	Cart_shift
2	0	grad1612_mpi_heat.c	181	main	File_set_view
3	0	grad1612_mpi_heat.c	280	main	File_open
4	0	grad1612_mpi_heat.c	222	main	Recv_init
5	0	grad1612_mpi_heat.c	221	main	Recv_init
6	0	grad1612_mpi_heat.c	203	main	Barrier
7	0	grad1612_mpi_heat.c	217	main	Recv_init
8	0	grad1612_mpi_heat.c	211	main	Send_init
9	0	grad1612_mpi_heat.c	207	main	Send_init
10	0	grad1612_mpi_heat.c	78	main	Cart_shift
11	0	grad1612_mpi_heat.c	180	main	Type_commit
12	0	grad1612_mpi_heat.c	175	main	File_open
13	0	grad1612_mpi_heat.c	224	main	Recv_init

```
 14   0 grad1612_mpi_heat.c    73 main                Cart_create
 15   0 grad1612_mpi_heat.c   271 main                Waitall
 16   0 grad1612_mpi_heat.c   220 main                Recv_init
 17   0 grad1612_mpi_heat.c   277 main                Reduce
 18   0 grad1612_mpi_heat.c   214 main                Send_init
 19   0 grad1612_mpi_heat.c   210 main                Send_init
 20   0 grad1612_mpi_heat.c   282 main                File_close
 21   0 grad1612_mpi_heat.c   308 main                Type_free
 22   0 grad1612_mpi_heat.c   309 main                Type_free
 23   0 grad1612_mpi_heat.c   187 main                File_close
 24   0 grad1612_mpi_heat.c   219 main                Recv_init
 25   0 grad1612_mpi_heat.c   141 main                Type_commit
 26   0 grad1612_mpi_heat.c   213 main                Send_init
 27   0 grad1612_mpi_heat.c   209 main                Send_init
 28   0 grad1612_mpi_heat.c   232 main                Startall
 29   0 grad1612_mpi_heat.c   230 main                Startall
 30   0 grad1612_mpi_heat.c   185 main                Type_commit
 31   0 grad1612_mpi_heat.c   223 main                Recv_init
 32   0 grad1612_mpi_heat.c   143 main                Bcast
 33   0 grad1612_mpi_heat.c   218 main                Recv_init
 34   0 grad1612_mpi_heat.c   306 main                Type_free
 35   0 grad1612_mpi_heat.c   208 main                Send_init
 36   0 grad1612_mpi_heat.c   138 main                Type_commit
 37   0 grad1612_mpi_heat.c   241 main                Waitall
 38   0 grad1612_mpi_heat.c   144 main                Bcast
 39   0 grad1612_mpi_heat.c   307 main                Type_free
 40   0 grad1612_mpi_heat.c   212 main                Send_init
---------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------
---------------------------------------------------------------------
Call                Site       Time    App%    MPI%     COV
File_open             12   4.65e+03   29.21   32.68    0.03
Waitall               37   3.29e+03   20.64   23.09    0.06
File_open              3   2.27e+03   14.23   15.92    0.07
File_close            20   1.17e+03    7.37    8.24    0.15
File_close            23        998    6.27    7.01    0.09
Cart_create           14        645    4.05    4.53    0.00
```

File_set_view 2 515 3.24 3.62 0.27
Barrier 6 263 1.65 1.85 0.36
Startall 29 229 1.44 1.61 0.48
Startall 28 94.8 0.60 0.67 0.16
Waitall 15 74.4 0.47 0.52 0.03
Send_init 9 10.9 0.07 0.08 3.87
Bcast 32 9.19 0.06 0.06 0.25
Send_init 8 8.83 0.06 0.06 3.91
Reduce 17 2.55 0.02 0.02 1.00
Send_init 26 0.644 0.00 0.00 1.88
Send_init 19 0.564 0.00 0.00 0.92
Type_commit 36 0.451 0.00 0.00 0.15
Send_init 40 0.435 0.00 0.00 1.14
Recv_init 5 0.37 0.00 0.00 1.02

@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------

---

Call	Site	Count	Total	Avrg	Sent%
Bcast	32	16	1.02e+03	64	47.06
Bcast	38	16	1.02e+03	64	47.06
Reduce	17	16	128	8	5.88

…….

@--- End of Report ----------
