ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

ΤΜΗΜΑ
ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

Μια προσέγγιση στο MPI, το OpenMP και την Cuda.

Χειμερινό εξάμηνο

Αντώνης Παρώνης: **1115201500126**
Χρήστος Σαμούχος: **1115201600149**
Μυρτώ Πλευράκη: **1115201500132**

Αθήνα 2019

Περιεχόμενα

Εισαγωγή

Σκοπός της εργασίας είναι η εξοικείωση με τον παράλληλο προγραμματισμό και τις διάφορες εκφάνσεις του. Πιο συγκεκριμένα γίνεται χρήση των τεχνικών, διεπαφής μηνυμάτων (Message Passing Interface), της εφαρμογής παράλληλης

προγραμματιστικής διεπαφής (parallel programming application interface), γνωστότερο και ως OpenMP και του προγραμματισμού σε κάρτες γραφικών (GPU programming). Αυτές οι υλοποιήσεις ακολουθούν την λογική του SPMD, δηλαδή της εκτέλεσης μοναδικού κώδικα από "πολλούς", με σκοπό την κατανομή του φορτίου εργασίας στο παράλληλο σύστημα.

Το πρόβλημα

Το πρόβλημα που κληθήκαμε να αντιμετωπίσουμε υλοποιεί την μελέτη ενός απλοποιημένου φυσικού φαινομένου. Συγκεκριμένα έπρεπε να μελετήσουμε την μεταφορά της θερμότητας σε μεταλλική πλάκα με τα εξής χαρακτηριστικά:

- Η πλάκα θερμαίνεται στιγμιαία ακριβώς στο κέντρο.
- Η πλάκα είναι δισδιάστατη επομένως η έννοια του πάχους δεν συμπεριλαμβάνεται στην μελέτη.
- Αξιωματικά θεωρούμε ότι η περίμετρος της πλάκας έχει πάντα μηδενική θερμοκρασία.

Αλγοριθμική Προσέγγιση

Αφού περιγράψαμε την φυσική προσέγγιση του προβλήματος με τα επιμέρους χαρακτηριστικά του, θα αναφέρουμε τα σημεία που αφορούν τον προγραμματιστή:

- Θωρούμε ότι η πλάκα χωρίζεται με την βοήθεια ενός πλέγματος στο καρτεσιανό σύστημα συντεταγμένων σε κελιά με αμελητέες διαστάσεις που περιγράφονται πλήρως από την θερμοκρασία τους. Εικονικά, μπορούμε να θεωρήσουμε την πλάκα ως πίνακα, όπου οι τιμές αντιστοιχίζονται με θερμαινόμενα.
- Ο υπολογισμός της μεταφοράς της θερμότητας προφανώς και εξαρτάται από τον χρόνο, δεδομένου ότι το φυσικό φαινόμενο έχει διάρκεια. Δεν χρησιμοποιούμε κάποια συγκεκριμένη μονάδα χρόνου, αλλά γενικά υποθέτουμε ότι ένα στιγμιότυπο του προγράμματος συμβαίνει στην μονάδα TIMESTEPS. Επομένως η διάρκεια του πειράματος κάθε φορά επηρεάζεται από την μακροεντολή που ορίζεται με #define στην αρχή του προγράμματος. Επομένως θα γίνουν TIMESTEPS φορές οι υπολογισμοί που παράγουν τον καινούργιο πίνακα, την νέα δηλαδή θερμική κατανομή στην πλάκα.

- Για του μοντέλου προγραμματιστικά χρησιμοποιείται ένας τρισδιάστατος πίνακας με πλάτος και ύψος αυτά της πλάκας και πάχος ίσο με δύο. Πρακτικά δεσμεύουμε δυναμικά δύο μεταλλικές πλάκες, όπου κάθε μία αναφέρεται σε διαφορετικό στιγμιότυπο. Αναλυτικότερα, για τον υπολογισμό της "μελλοντικής" κατάστασης απαιτούνται οι τιμές των κελιών από την παροντική κατάσταση.

```
float *current_array, *future_array;
```

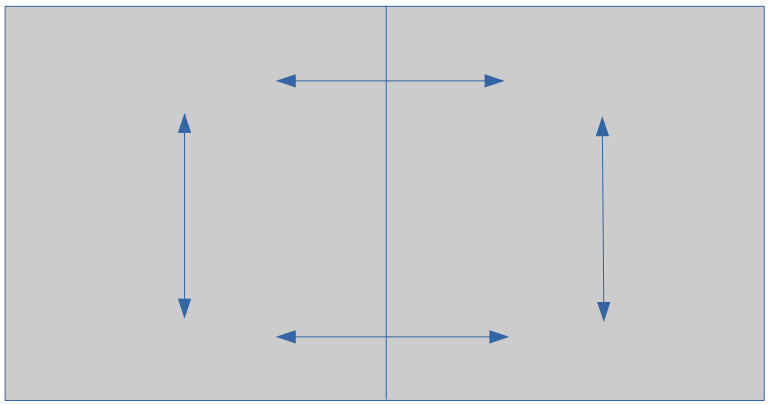
Έτσι οι νέες τιμές παράγονται και αποθηκεύονται στον μελλοντικό πίνακα. Τελικά οδηγούμαστε στην συνεχή εναλλαγή των δύο πινάκων για τον τελικό υπολογισμό του μελλοντικού πίνακα.

- Ο διαχωρισμός των δεδομένων του πίνακα γίνεται σε μικρότερους υποπίνακες (blocks), οι οποίοι διαμοιράζονται προσπαθώντας να τηρήσουν όσο το δυνατόν καλύτερα την αρχή της εξισορρόπησης φόρτου. Αυτό επιτυγχάνεται με την επιλογή των κατάλληλων συναρτήσεων του MPI (θα αναλυθεί περαιτέρω στην αντίστοιχη ενότητα)

- Η επικοινωνία των επεξεργαστικών κόμβων υποβιβάζεται σύμφωνα με την αρχή της τοπικότητας. Με άλλα λόγια μόνο οι κόμβοι που αναλαμβάνουν την επεξεργασία των blocks που εφάπτονται μεταξύ τους μπορούν να επικοινωνούν.

```
MPI_Comm MPI_CART_COMM;     /*create new communicator in order to
                              change the topology*/
```

Χαρακτηριστικά η λογική που χρησιμοποιείται σε παράδειγμα είναι η εξής. Έστω πίνακας που διαχωρίζεται σε τέσσερεις διεργασίες. Τότε η επικοινωνία γίνεται μόνο εκεί που υποδεικνύουν τα βέλη.



Foster

Με την μέθοδο Foster έχουμε την δυνατότητα να σχεδιάσουμε την παραλληλοποίηση ενός σειριακού προγράμματος έτσι ώστε κάθε διεργασία να

έχει περίπου το ίδιο φορτίο με τις υπόλοιπες και να ελαχιστοποιήσουμε την επικοινωνία.
Η μέθοδος αυτή μας δίνει τέσσερα στάδια/βήματα στα οποία μπορούμε να χωρίσουμε τον σχεδιασμό αυτό, καθώς και να εφαρμόσουμε βελτιστοποιήσεις:

ΔΙΑΜΕΡΙΣΗ: Χωρίζουμε του υπολογισμούς και τα δεδομένα πάνω στα οποία θα πραγματοποιηθούν αυτοί οι υπολογισμοί σε μικρές εργασίες. Προσοχή στην αναγνώριση εργασιών που μπορούν να εκτελούνται παράλληλα.

ΕΠΙΚΟΙΝΩΝΙΑ: Προσδιορίζουμε τις απαραίτητες επικοινωνίες μεταξύ των εργασιών που καθορίσαμε στο προηγούμενο βήμα.

ΣΥΣΣΩΡΕΥΣΗ: Συνδυάζουμε εργασίες και επικοινωνίες που αναγνωρίσαμε προηγουμένως σε μεγαλύτερες εργασίες.

ΑΝΤΙΣΤΟΙΧΗΣΗ: Αναθέτουμε σε διεργασίες/νήματα τις σύνθετες εργασίες που καθορίστηκαν στο προηγούμενο βήμα. Αυτό πρέπει να γίνεται έτσι ώστε οι επικοινωνίες να ελαχιστοποιούνται και κάθε διεργασία/νήμα να έχει περίπου το ίδιο φορτίο εργασίας.

Το αρχικό πρόγραμμα που μας δόθηκε ακολουθεί μία διαμέριση σε 1-διάσταση. Αυτό στο πρόβλημα μας σημαίνει ότι η πλάκα χωρίστηκε σε επιμέρους λωρίδες και η κάθε λωρίδα επικοινωνούσε με τις αριστερές και δεξιές της για ανταλλαγή halo στοιχείων. Έτσι κάθε processor είχε:

1. Να εφαρμόσει υπολογισμούς σε ένα κομμάτι $N \mathrm{x} \frac{M}{P}$ στοιχείων και
2. Να επικοινωνήσει με 2 γείτονές του και να ανταλλάξει 4 (2 send, 2 receive) κομμάτια N στοιχείων στα άκρα.

Έτσι προκύπτει ο χρόνος που δίνεται από τον τύπο:

$$T_{1d} = T_{comp} + T_{comm} = T \cdot t_c \cdot \frac{M}{P} \cdot N + 4 \cdot T \cdot (t_{latency} + t_w \cdot N)$$

όπου t_c ο χρόνος για τον υπολογισμό ενός στοιχείου.

Η επιτάχυνση του παράλληλου προγράμματος είναι:

$$S = \frac{t_c \cdot N \cdot M \cdot T}{T_{1d}} = \frac{P \cdot t_c \cdot N \cdot M}{t_c \cdot M \cdot N + 4 \cdot P \cdot (t_{latency} + t_w \cdot N)}$$

και η αποδοτικότητα:

$$E = \frac{S}{P} = \frac{t_c \cdot N \cdot M}{t_c \cdot M \cdot N + 4 \cdot P \cdot (t_{latency} + t_w \cdot N)}$$

Ενώ το πρόγραμμα που υλοποιήσαμε ακολουθεί την διαμέριση 2-διαστάσεων. Αυτό στο πρόβλημα μας σημαίνει ότι η πλάκα χωρίστηκε σε επιμέρους πλάκες οι οποίες επικοινωνούσαν με 4 γείτονες τους (βόρειο, νότιο, ανατολικό, δυτικό) για ανταλλαγή halo στοιχείων. Έτσι κάθε processor είχε:

1. Να εφαρμόσει υπολογισμούς σε ένα κομμάτι $\frac{N}{\sqrt{P}}$ x $\frac{M}{\sqrt{P}}$ στοιχείων και
2. Να επικοινωνήσει με 4 γείτονες του και να ανταλλάξει 4 (2 send, 2 receive) κομμάτια $\frac{N}{\sqrt{P}}$στοιχείων (με δυτικό, ανατολικό) και 4 (2 send, 2 receive) κομμάτια $\frac{M}{\sqrt{P}}$ στοιχείων (με βόρειο, νότιο).

Έτσι προκύπτει ο χρόνος που δίνεται από τον τύπο:

$$T_{2d} = T_{comp} + T_{comm} = T \cdot t_c \cdot \frac{M}{\sqrt{P}} \cdot \frac{N}{\sqrt{P}} + 4 \cdot T \cdot (t_{latency} + t_w \cdot \frac{N}{\sqrt{P}}) + 4 \cdot T \cdot (t_{latency} + t_w \cdot \frac{M}{\sqrt{P}}) = T \cdot t_c \cdot \frac{M \cdot N}{P} + 4 \cdot T \cdot (2 \cdot t_{latency} + t_w \cdot \frac{N+M}{\sqrt{P}}) = T \cdot (t_c \cdot \frac{M \cdot N}{P} + 8 \cdot t_{latency} + t_w \cdot \frac{N+M}{\sqrt{P}})$$

όπου t_c ο χρόνος για τον υπολογισμό ενός στοιχείου.

Η επιτάχυνση του παράλληλου προγράμματος είναι:

$$S = \frac{t_c \cdot N \cdot M \cdot T}{T_{2d}} = \frac{t_c \cdot N \cdot M \cdot P}{t_c \cdot M \cdot N + 8 \cdot t_{latency} \cdot P + t_w(\cdot N + M) \cdot \sqrt{P}}$$

και η αποδοτικότητα:

$$E = \frac{S}{P} = \frac{t_c \cdot N \cdot M}{t_c \cdot M \cdot N + 8 \cdot t_{latency} \cdot P + t_w(\cdot N + M) \cdot \sqrt{P}}$$

Η αποτελεσματικότητα (isoefficiency) για ένα αλγόριθμο μια συνάρτηση του N, που υποδεικνύει πόσο κλιμακωτός είναι ένας αλγόριθμος. Δηλαδή, υποδεικνύει πώς ο το μέγεθος του υπολογισμού (στην περιπτωση μας τα M και N, τα οποια στις μετρησεις μας αυξάνονται με τον ίδιο ρυθμό) πρέπει να κλιμακωθεί με P για να παραμείνει το E σταθερό.

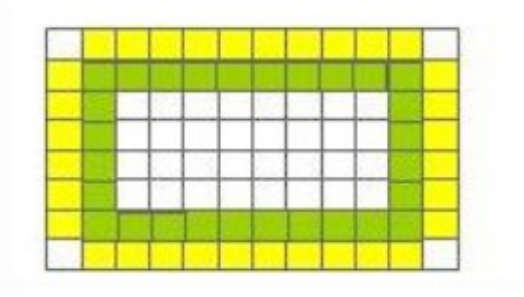
Το isoefficiency του παράλληλου προγράμματος που βασίζεται στη 1-διάσταση είναι O(P^2)
γιατί τα M και N πρέπει να αναπτύσσονται με το ίδιο ρυθμό με το P για να παραμεινει σταθερο το E, οπότε το μέγεθος του υπολογισμού/το μέγεθος του προβλήματος (M επί N) πρέπει να αυξανεται όσο το οπως ο ρυθμός του P^2.

Ενώ, το isoefficiency του παράλληλου προγράμματος που βασίζεται στη 1-διάστασεις είναι O(P), γιατί το μεγεθος του προβλήματος πρέπει να αναπτυχθεί ως $\sqrt{P}$,οπότε πρέπει να αυανεται με ίδιο ρυθμό με το P.

Άρα το παράλληλο πρόγραμμα που βασίζεται στις 2-διαστάσεις κλιμακώνεται καλύτερα από αυτό της 1-διάστασης.

Μια παράλληλη προσέγγιση

Η κύρια υλοποίηση του παράλληλου προγράμματος εκμεταλλεύεται ακριβώς τον διαχωρισμό των δεδομένων σε blocks όπως αναφερθήκαμε. Αναλυτικότερα κάθε επεξεργαστικός κόμβος θα έχει ένα σύνολο δεδομένων που θα έχουν την μορφή που παρουσιάζεται στην εικόνα.



Συνοπτικά, κάθε χρώμα αντιπροσωπεύει και μια μοναδική ιδιότητα του κελιού, η οποία μας ενδιαφέρει στον γενικότερο προγραμματιστικό σχεδιασμό αλλά και τον χειρισμό των δεδομένων.

Τα πράσινα είναι τα περιφερειακά κελιά κάθε block που η τιμή τους εξαρτάται και από τα διπλανά block. Τα λευκά κελιά θεωρούνται ανεξάρτητα των γειτονικών block με αποτέλεσμα ο υπολογισμός τους να γίνεται τοπικά χωρίς την ανάγκη επικοινωνίας. Ο προσδιορισμός της θέσης του block ως προς το

χώρο γίνεται με βάση το σύστημα συντεταγμένων Βορά-Νότου-Ανατολής-Δύσης.

MPI

Για την καλύτερη κατανόηση θα γίνει παράλληλη αναφορά των συναρτήσεων που χρησιμοποιήθηκαν και τον σκοπό που εξυπηρετούν. Η βασική ιδέα πίσω από την βέλτιστη παραλληλοποίηση, δηλαδή την μέγιστη αξιοποίηση των υπολογιστικών δυνατοτήτων με όσο των δυνατόν μείωση του χρόνου αδράνειας (idle), είναι η "άναρχη δομή". Πριν παρεξηγηθεί η ορολογία, αναφερόμαστε σε μία επεξεργαστική δομή η οποία λειτουργεί χωρίς MASTER κόμβο(node). Προσοχή, σε κάθε παράλληλο πρόγραμμα υπάρχει τουλάχιστον ένα σειριακό μέσα. Αυτό που εμείς προσπαθούμε να επιτύχουμε είναι την μείωση του πεδίου δράσης του σειριακού όσο το δυνατόν περισσότερο. Με απλά λόγια δεν θέλουμε για κανένα λόγο οι κόμβοι να περιμένουν να ξεκινήσουν την επεξεργασία μέχρι να εκτελεστεί κάποια άλλη διεργασία από τον MASTER. Έτσι ενώ στον αρχικό πρόγραμμα η αρχικοποίηση των τιμών γινόταν με την ανακατεύθυνση εισόδου με τον πίνακα, στη δική μας βελτιωμένη έκδοση κάθε κόμβος αρχικοποιεί κομμάτι του πίνακα που του αντιστοιχεί στο δικό του "χώρο". Αυτό προσφέρει:

- μεγαλύτερη κλιμάκωση, διότι δεν αποθηκεύεται κάπου ολόκληρος ο πίνακας με κίνδυνο υπερχείλισης μνήμης(overflow). Συνεπώς το πρόγραμμά μας επιτρέπει την μελέτη καταστάσεων με μεγαλύτερες διαστάσεις, δηλαδή περισσότερα δεδομένα.
- Μειώνει τον χρόνο εκτέλεσης καθώς δεν χρειάζεται ο MASTER να μοιράσει τις διαστάσεις της μεταλλικής πλάκας σε ίσα τμήματα και να τα μεταφέρει μέσο επικοινωνητή στους άλλους κόμβους.

Περισσότερα αναφέρονται στην αντίστοιχη παράγραφο περί επεκτασιμότητας, όπου επισημαίνεται και η χρήση του Parallel I/0 ως επιπλέον μέσο βελτίωσης της εισόδου αλλά και της εξόδου του προγράμματος.

Μια άλλη σημαντική σχεδιαστική επιλογή ήταν η χρήση συναρτήσεων του mpi που αφήνουν τον διαχωρισμό των δεδομένων του πίνακα (ότι διαστάσεις και να

```
MPI_Dims_create(rank_size, 2, processor_scheme);
                                        /*Create the processor scheme, how
                                             they are going to be organized
*/
```

έχει) στην διακριτική ευχέρεια του συστήματος(βλ. Κώδικα απο πάνω).Το πλέγμα(grid) κατασκευάζεται με παρόμοιο τρόπο ακολουθώντας τα πρότυπα της

καρτεσιανής τοπολογίας(εφαρμόζοντας το σύστημα Βορά-Νότου για καλύτερη κατανόηση).

```
 MPI_Cart_create(MPI_COMM_WORLD, 2, processor_scheme, periods, 1,
&MPI_CART_COMM);
 MPI_Cart_coords(MPI_CART_COMM,my_rank,2,my_coords);

 MPI_Cart_shift(MPI_CART_COMM,0,1,&neighbors[NORTH], &neighbors[SOUTH]); /* Y
axis */
 MPI_Cart_shift(MPI_CART_COMM,1,1,&neighbors[WEST], &neighbors[EAST]);/*X axis */
```

Να σημειωθεί για την ονοματολογία των διαφόρων tags που χρησιμοποιήθηκαν θεωρήσαμε διάνυσμα με αφετηρία τον τόπο προέλευσης(βάζουμε NORTH όταν τα δεδομένα έρχονται από τον βορά κτλ.) Ενδιαφέρον παρουσιάζει και η δημιουργία καινούργιων τύπων δεδομένων που εξυπηρετούν στις λήψεις και αποστολές μεταξύ των γειτονικών blocks(στέλνουμε ολόκληρες σειρές και στήλες και όχι μεμονωμένα στοιχεία).

```
 MPI_Type_vector(local_size_y, 1, local_size_x+2 , MPI_FLOAT,
&vertical_vector_temp);
 MPI_Type_create_resized(vertical_vector_temp, 0, sizeof(float),
&vertical_vector);
 MPI_Type_commit(&vertical_vector);

 MPI_Type_contiguous(local_size_x, MPI_FLOAT, &horizontal_vector_temp);

 MPI_Type_create_resized(horizontal_vector_temp, 0, sizeof(float),
&horizontal_vector);
 MPI_Type_commit(&horizontal_vector);
```

Όπως επισημάνθηκε προηγουμένως απαιτείται η επικοινωνια των blocks μεταξύ τους για τον υπολογισμό των στοιχείων της περιμέτρου. Αυτό επιτυγχάνεται με την χρήση ανασταλτικής μόνιμης επικοινωνίας για να αποφεύγονται οι καθυστερήσεις και να μειώνεται ο χρόνος εκτέλεσης. Συγκεκριμένα γίνεται πρώτα χρήση των συναρτήσεων receive ώστε οι κόμβοι να είναι σε αναμονή για τα νέα στοιχεία.

Ένα παράδειγμα φαίνεται στον κώδικα.

```
  MPI_Irecv(current_array + GET_OFFSET(local_size_y+1,1), 1,
horizontal_vector,                                neighbors[SOUTH], SOUTH,
MPI_CART_COMM, &receiving_requests[SOUTH]);

  MPI_Isend(current_array + GET_OFFSET(1,1), 1, horizontal_vector,
neighbors[NORTH], SOUTH,                    MPI_CART_COMM,
&sending_requests[NORTH]);
```

Αφού οι κόμβοι ανταλλάξουν στοιχεία περνάμε στο στάδιο του υπολογισμού.

Εκεί θεωρούμε την συνάρτηση update, η οποία λειτουργεί σε δύο φάσεις. Στον υπολογισμό των εσωτερικών-ανεξάρτητων κελιών και στον υπολογισμό της περιμέτρου. Ο διαχωρισμός γίνεται προφανώς για την αύξησης της ταχύτητας εκτέλεσης, αφού τα εσωτερικά σημεία μπορούν να υπολογιστούν χωρίς απαραίτητα η ανταλλαγή δεδομένων να έχει ολοκληρωθεί μειώνοντας τον χρόνο αδράνειας.

OpenMP

Η διεπαφή openMP μας προσφέρει την δυνατότητα εύκολου και γρήγορου προγραμματισμού σε κοινόχρηστη μνήμη. Αυτό μας δίνει την ευκαιρία να παραλληλοποιήσουμε το πρόγραμμα μας με έναν υβριδικό τρόπο, (MPI+openMP) δηλαδή κατανεμημένη μνήμη με χρήση κοινόχρηστης σε κάθε κόμβο(node). Η διεπαφή openMP είναι, όπως αναφέρθηκε, αρκετά εύκολη, πράγμα που έκανε την προσαρμογή του αρχικού προγράμματος απλή ως διαδικασία.

Σχεδιαστικά, οι μόνες αλλαγές που χρειάστηκε να κάνουμε ήταν η χρήση των νημάτων κατά την εκκίνηση. Επομένως έπρεπε να αλλάξουμε το MPI_Init σε MPI_Init_threads, καθώς και να χρησιμοποιησουμε τα directives του OMP για παραλληλοποίηση των βρόγχων μας. Πιο συγκεκριμένα, χρησιμοποιήθηκαν οι εντολές του openmp για την παραλληλοποίηση των υπολογισμών που κάθε κόμβος έχει αναλάβει να διεκπεραιώσει στα πλαίσια δικαιοδοσίας του. Παραθέτουμε το αντίστοιχο κομμάτι κώδικα για καλύτερη κατανόηση:

```
#include <omp.h>

 MPI_Init_thread(NULL,NULL,MPI_THREAD_MULTIPLE,&provided);
 #pragma omp parallel default(none)
shared(local_size_x,local_size_y,sub_array,
 MPI_CART_COMM,receiving_requests,sending_requests,my_rank)
      private(current_array,future_array)
 firstprivate(iz,neighbors,horizontal_vector,vertical_vector)

{
    for(int time_step = 1, array_size =
(local_size_x+2)*(local_size_y+2); time_step <= TIMESTEPS; time_step+
+){       /* main for that updates the values of the subarrays for
TIMESTEPS times */

      current_array = sub_array + iz * array_size;
      future_array  = sub_array + (1-iz) * array_size;

#pragma omp single
```

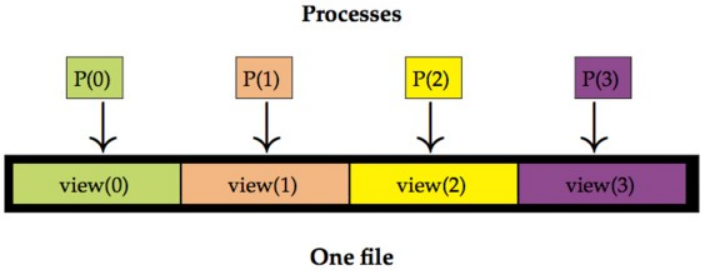
Επιπλέον, στην παραλληλοποίηση των βρόγχων αποφασίσαμε να χρησιμοποιήσουμε χρονοπρογραμματισμό τύπου static, καθώς κάθε επανάληψη έχει από μόνο του ισομοιρασμένο φόρτο. Σε δόκιμες που κάναμε, ο χρονοπρογραμματισμός dynamic αύξανε σημαντικά τον χρόνο εκτέλεσης, λόγω του επιπλέον φόρτου για runtime χρονοπογραμματισμό νημάτων.

Αφού οι επικοινωνίες μας είναι ασύγχρονες, θεωρήσαμε ως βέλτιστο, όλα τα νήματα να συμβάλλουν στους βρόγχους και να μην έχουμε επικάλυψη επικοινωνίας με την εξής εντολή:

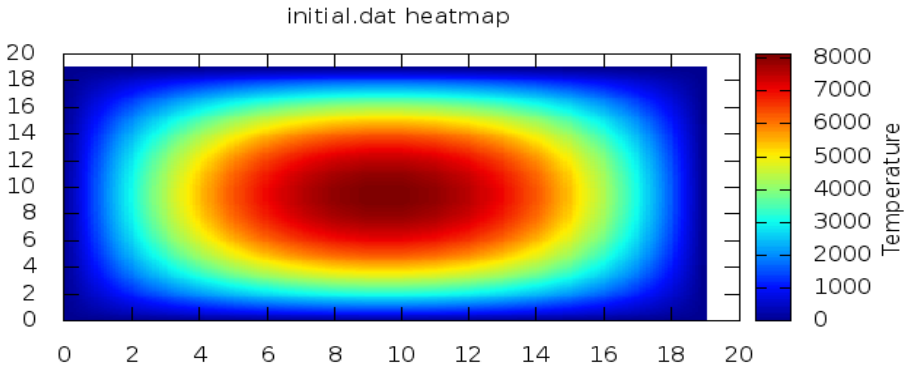
```
#pragma omp single
```

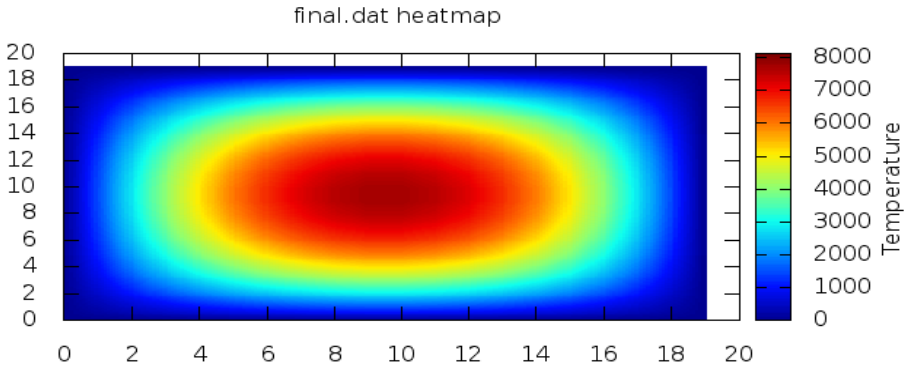
Parallel Input/Output (I/O)

Το σύστημα παράλληλης εισόδου και εξόδου που προσθέσαμε στον πρόγραμμα είναι υπεύθυνο για την δυνατότητα εισόδου και εξόδου σε αρχείο, ενώ αρχικά αυτό γινόταν τοπικά στο πρόγραμμα. Αυτό μας δίνει την δυνατότητα να διατηρήσουμε την αρχή της παραλληλίας(χωρίς σειριακό χρόνο) κερδίζοντας το πλεονέκτημα της αποθήκευσης των δεδομένων σε αρχεία. Προφανώς αυτό προσδίδει και ευελιξία εισόδου δεδομένων χωρίς να χρειάζεται να "πειράξουμε" τον πηγαίο κώδικα. Συνοπτικά κάθε κόμβος έχει περιορισμένη ορατότητα τόσο στο αρχείο εισόδου όσο και στο αρχείο εξόδου του προγράμματος χωρίς να έχει δυνατότητα να επηρεάσει χώρους εκτός του πεδίου δικαιοδοσίας που του αναθέτουμε. Διαγραμματικά συμβαίνει το εξής:



Τελικά για λόγους κατανόησης(και αφού μας δίνεται η δυνατότητα παραγωγής αρχείου εξόδου) με την βοήθεια του gnuplot οπτικοποιήσαμε είσοδο και έξοδο για την τελική αποτίμηση του προγράμματος από φυσικής πλευράς(σε ένα τυχαίο παράδειγμα), όπου παράγεται το εξής:





Τέλος, κάτι που αφορά φυσικά όλα τα κομμάτια που αντιμετωπίσαμε, αλλά πρώτα στο MPI, είναι ο συγχρονισμός. Επιλέξαμε δηλαδή τα βέλτιστα σημεία για

```
MPI_Request sending_requests[4], receiving_requests[4];
MPI_Barrier(MPI_COMM_WORLD);
MPI_Waitall(4, receiving_requests, MPI_STATUSES_IGNORE);
```

τον συγχρονισμό των διεργασιών για να αποφύγουμε το πρόβλημα ότι μπορεί ένας επεξεργαστικός κόμβος να έχει ξεκινήσει τον υπολογισμό των περιφερειακών κελιών χωρίς ακόμα να έχει λάβει τις καινούργιες τιμές.

Για την χρονομέτρηση του προγράμματος τόσο στο MPI, όσο και στο υβριδικό μαζί με τις επεκτάσεις χρησιμοποιήθηκαν οι εξής:

```
double time_start, time_end;
time_start = MPI_Wtime();
time_end=MPI_Wtime();
printf("total time is: %lf\n", time_end - time_start );
```

CUDA

Είναι πλέον γνωστό στις παράλληλες τεχνολογίες ότι η χρήση καρτών γραφικών(GPU) στα παράλληλα συστήματα είναι πολλά υποσχόμενη και πλέον αρκετά διαδεδομένη. Η NVIDIA προσφέρει την διεπαφή CUDA για τον προγραμματισμό των GPU της. Ως τρίτο μέρος της εργασίας μας, προσαρμόσαμε το δοσμένο πρόβλημα στον προγραμματισμό κάρτας γραφικών και δοκιμάσαμε τις επιδόσεις του κώδικα στην κάρτα γραφικών NVIDIA 940 MX.

Γιατί όμως GPU;

Αν και η GPU έχει πολύ περιορισμένες δυνατότητες έναντι μιας CPU, σε θέμα παραλληλίας προσφέρει μεγάλη υπολογιστική ισχύ. Αυτό, με απλά λόγια, συμβαίνει επειδή η GPU περιέχει έναν πολύ μεγάλο αριθμό "επεξεργαστών" που μπορούν να κάνουν πολλές πράξεις παράλληλα. Εκμεταλλευόμενοι αυτή ακριβώς την ιδιότητα μεταφέρουμε το το υπολογιστικά βαρύ κομμάτι του προγράμματος στην GPU και το αποτέλεσμα ξανά στην CPU για να το εμφανίσουμε. Στην δική μας περίπτωση δηλαδή το όλο σύνολο των υπολογισμών για την ανανέωση των τιμών που έχει περιγραφεί και πιο πάνω είναι στην διακριτική ευχέρεια της GPU. Αναλυτικά:

```
    gpuErrchk(cudaMemcpy(heatmap, dev_next_map, sizeof(float)*ARRAY_SIZE,
    cudaMemcpyDeviceToHost));
    gpuErrchk(cudaEventRecord(endEvent));
    cudaDeviceSynchronize();

    prtdat(X_SIZE, Y_SIZE, heatmap, output_file_name);
    gpuErrchk(cudaEventElapsedTime(&duration, startEvent, endEvent));
    printf("GPU elapsed time: %f\n", duration);
```

Σύγκλιση

Η σύκλιση είναι ένας τρόπος να μειώσουμε τον χρόνο εκτέλεσης για μεγάλα δεδομένα. Η λογική είναι η εξής. Εάν κατά την εκτέλεση του προγράμματος παρατηρήσουμε – μετρήσουμε αλλαγή στις τιμές των επιμέρους κελιών μικρότερη του 5% τότε θεωρούμε αυθαίρετα ότι το πρόγραμμα πρέπει να σταματήσει. Με άλλα λόγια υποθέτουμε ότι το τελικό αποτέλεσμα θα απέχει

ελάχιστα σε σχέση με αυτό που παράξαμε και έτσι γλιτώνουμε υπολογιστικούς πόρους αλλά και χρόνο. Παρακάτω φαίνεται η σύγκλιση σε μορφή κώδικα:

```
#ifdef REDUCE_PROGRAM
        convergence_condition[0] = 1;
        for(iy=1; iy<=local_size_y; iy++){
          for(ix=1 ; ix < local_size_x+1; ix++){
              if((*(future_array + GET_OFFSET(iy,ix)) - *(current_array +
GET_OFFSET(iy,ix)) / *(current_array + GET_OFFSET(iy,ix)) ) > CONV_ERROR){
                convergence_condition[0] = 0;
                iy = local_size_y + 1;
                break;
            }
          }
          if(convergence_condition[0] == 0){
            break;
          }
        }
        MPI_Allreduce(&convergence_condition[0],
&convergence_condition[1] , 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD); /* first
position has the sending condition and the second has receiving condition
*/

        if(convergence_condition[1])
          break;
#endif
```

Συμπεράσματα – Πίνακες

80x64

Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	0.117834	0.123675	0.251828	0.472953	-	-
MPI+reduce	0.108176	0.250775	-	0.811404	-	-
Hybrid (2 threads per process)	none	0.045776	0.03264	0.173826	0.320277	0.244829
Hybrid(4 threads per process)	none	0.031481	0.020667	2.730618	0.181092	0.205706
Hybrid(8 threads per process)	none	-	0.023065	0.169726	0.109521	0.220185

160x128

Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	0.577654	0.396939	0.418815	0.465565	-	-
MPI+reduce	0.01633	0.209345	0.505324	-	-	-
Hybrid (2 threads per process)	none	0.051944	0.76977	0.180434	1.380832	1.49408
Hybrid(4 threads per process)	none	0.056824	0.925817	2.985597	2.301648	0.249838
Hybrid(8 threads per process)	none	-	0.060789	0.157516	0.127914	0.143516

320x256

Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	1.858629	0.611642	0.250797	0.34743	-	-
MPI+reduce	0.014581	0.86817	0.485801	1.221267	-	-
Hybrid (2 threads per process)	none	0.132427	0.965918	0.32489	0.435818	1.534831
Hybrid(4 threads per process)	none	0.143306	1.14172	0.127059	0.19733	3.117966
Hybrid(8 threads per process)	none			0.152974	0.177764	

640x512

Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	7.448187	3.397568	0.733955	1.365794	-	-
MPI+reduce	0.020498	4.544736	1.20206	-	-	1.92662
Hybrid (2 threads per process)	none	1.063358	0.945146	1.438764	1.167888	1.380056
Hybrid(4 threads per process)	none	1.05942	0.358795	1.918252	1.55461	1.637306
Hybrid(8 threads per process)	none			0.71198	0.412207	0.295673

1280x1024

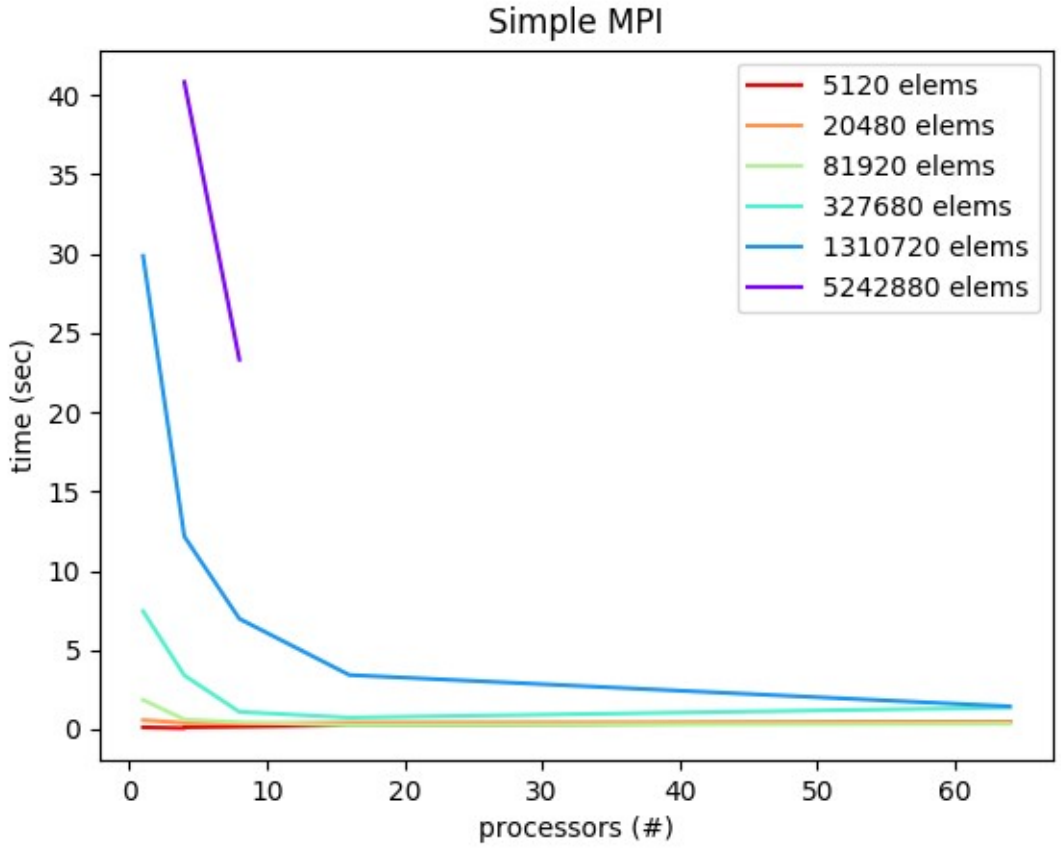
Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	29.869207	12.162705	3.419614	1.4436554	-	-

MPI+reduce	0.127601	16.224221	4.651058	-	-	-
Hybrid (2 threads per process)	none	1.710512	1.860733	0.508988	0.395254	1.48069
Hybrid(4 threads per process)	none	3.03457	1.982774	2.824294	0.966209	2.323134
Hybrid(8 threads per process)	none	-	1.784635	0.751315	0.662492	0.366983

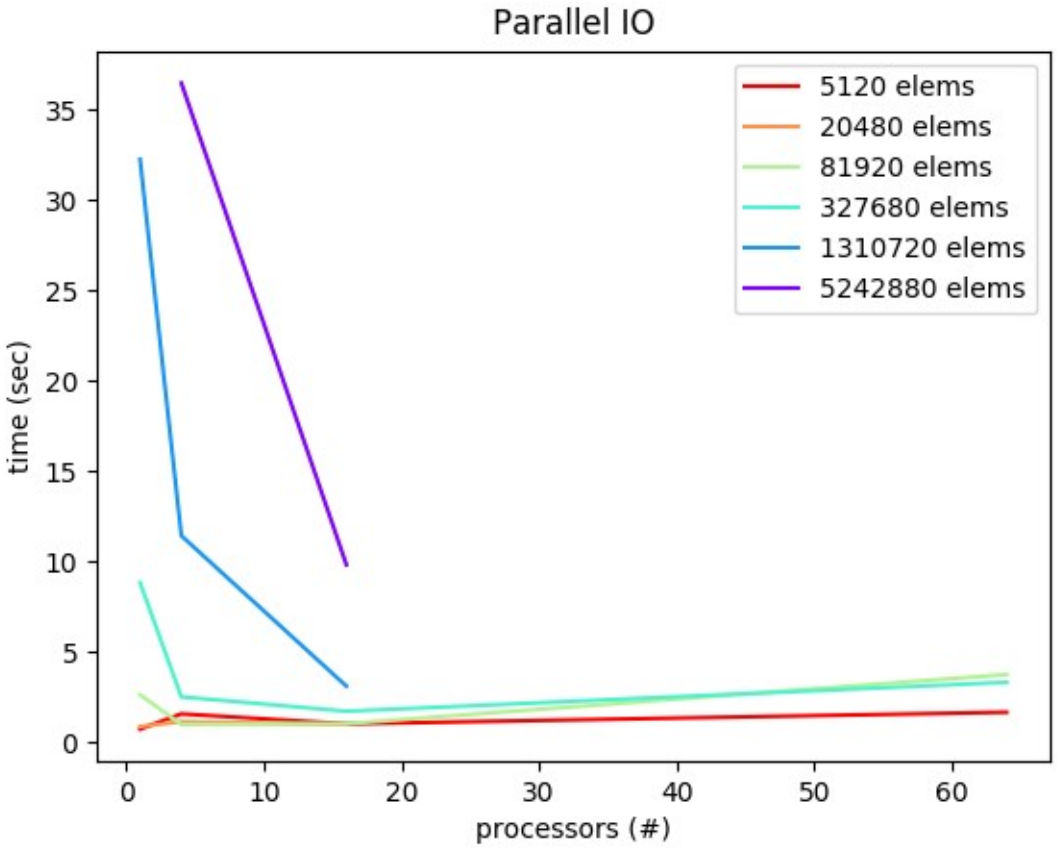
2560x2048

Υπολογιστικοί Κόμβοι	1	1	2	8	16	20
Tasks	1	4	16	64	128	160
MPI	-	40.832371	23.321913	-	-	-
MPI+reduce	0.240486	55.392949	16.843816	-	2.287471	-
Hybrid (2 threads per process)	none	14.337766	2.096967	0.514833	1.465128	1.212631
Hybrid(4 threads per process)	none	14.49125	2.976385	2.21376	1.507082	1.473423
Hybrid(8 threads per process)	none	-	7.95305	1.14677	0.54079	0.645348

Παρακάτω παρουσιαζόνται τα παραπάνω στοιχεία γραφικά, για καλύτερες συγκρίσεις:

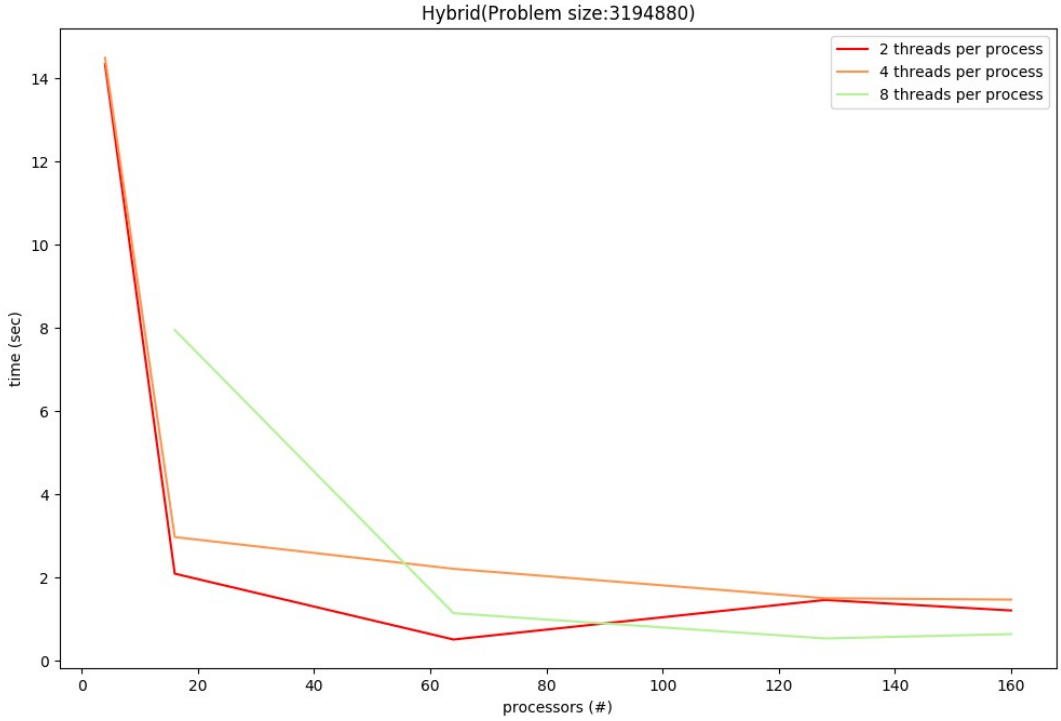


Στην εικόνα του απλού MPI παρατηρουμε οτι στα μικρότερα μεγέθη, όσο αυξάνονταν τα tasks, επιβαρυνόταν το πρόγραμμα. Αυτό είναι λογικό, καθώς η επιπρόσθετη επικοινωνία επιβαρύνει επιπλέον τις επιδόσεις, όταν υπάρχει μικρό περιθώριο βελτ'ιωσης. Παρ᾽ολα αυτά στα μεγαλύτερα μεγέθη προβλήματος, η αύξηση των tasks προκαλεί επιτάχυνση του προγράμματος.



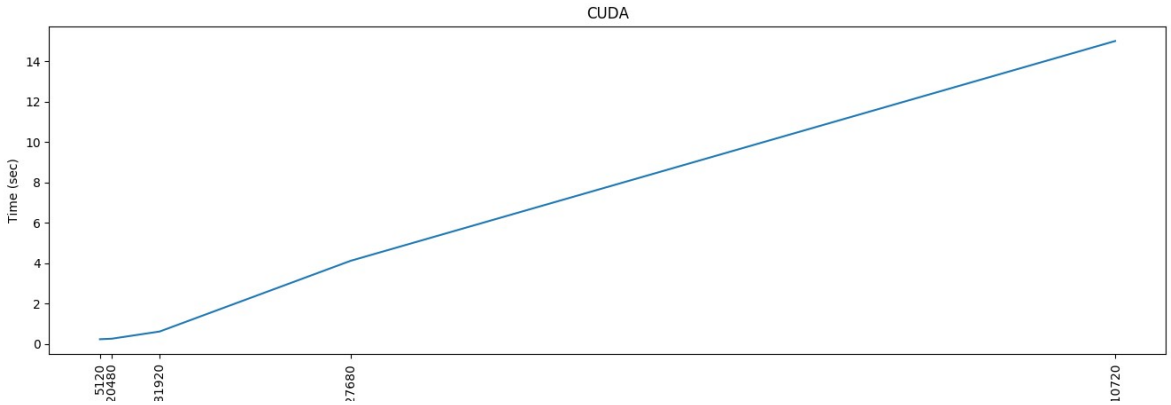
Με parallel IO το speedup του προγράμματος φαίνεται να παραμένει σταθερό με την αύξηση των πόρων, το οποίο είναι πολύ θετικό, καθώς μπορούμε να έχουμε πιο πρακτική είσοδο και έξοδο στο πρόγραμμα μας και να μην επιβαρύνομαστε επιπλέον απο αυτό. Συμπερασματικά η κλιμάκωση φόρτου γίνεται ομαλά.

Στην εικόνα του υβριδικού MPI+openMP προγραμμάτος, παρατηρουμε το προγραμμα μας έχει μια ομαλότερη συμπεριφορά με την αύξηση των tasks στα 8 threads ανά διεργασία. Σε αυτό πιθανολογούμε ότι παίζει ρόλο το γεγονός της μικρότερης επικοινωνίας άρα και συμφόρησης όταν χρησιμοποιούμε το μοντέλο της κοινόχρηστης μνήμης ανά κόμβο.



CUDA

Problem size	Time (ms)
80x64	233.35
160x128	260.08
320x256	615.17
640x512	4118.07
1280x1024	151520
2560x2048	62722687
10240x16384	236653854



Εν κατακλείδι να προσθέσουμε ότι ο χρόνος εκτέλεσης της cuda δεν είναι άμεσα συγγκρίσιμος με τα υπόλοιπα λόγω διαφοράς υπολογιστικής ισχύως τον συστημάτων που χρησιμοποιήθηκαν.

Χρησιμοποιώντας τους χρόνους από τους πάνω πίνακες και τους τύπους που αναφέραμε πιο πάνω βλέπουμε ενδεικτικά οτι:

# processors	Average Speedup	Average Efficiency
4	2.5622	0.6405639
16	8.7645	0.5477828
64	10.4976	0.1640260

Υποσημείωση

- *Τα προγράμματα εκτελέστηκαν στο υπολογιστικό σύστημα του Δημόκριτου εκτός απο την cuda με την χρήση του μεταγλωττιστή mpicc*

- *Η cuda εκτελέστηκε στο περιβάλλον του visual studio κατεβάζοντας την αντίστοιχη βιβλιοθήκη.*

- *Λόγω τεχνικών προβλημάτων τύπου συμφόρησης στο rbs.marie του Δημόκριτου τα παραπάνω αποτελέσματα είναι τα βέλτιστα που μπορούσαμε να πετύχουμε.*