ΕΡΓΑΣΙΑ ΣΤΑ ΠΛΑΙΣΙΑ ΤΟΥ ΜΑΘΗΜΑΤΟΣ: ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

**Χειμερινό Εξάμηνο**

****

**Εργασiα 2016-17: Conway’s Game of Life**

1115201300065 Κατηφόρης Ελευθέριος 1115201300177 Τουμάσης Άγγελος

**Περιεχόμενα**

* **Εισαγωγή**
* **Σχεδιασμός Διαμοιρασμού Δεδομένων**
* **Σχεδιασμός και υλοποίηση MPI κώδικα**
* **Παρουσίαση αποτελεσμάτων** 
  + **MPI**
    - *Generations = 100*
    - *Generations = 500*
    - *Generations = 1000*
  + **Openmp**
  + **Cuda**
* **Συμπεράσματα**
* **Βιβλιογραφία**

**Εισαγωγή**

Η εργασία καλύπτει όλες τις απαιτήσεις της εκφώνησης και διαθέτει σε καίρια σημεία τον απαραίτητο σχολιασμό.

Αποτελείται από τα παρακάτω αρχεία :

main.c , master.c , worker.c , functions.c, header.h και το ενδεικτικό checker.cpp, με σειριακή υλοποίηση του προβλήματος που χρησιμοποιήθηκε για τη σύγκριση των αποτελεσμάτων του παράλληλου προβλήματος ώστε να ελεγχθούν ότι είναι σωστά τα παραγόμενα αποτελέσματα.

Παρέχεται επιπλέον makefile με τις εντολές για μεταγλώττιση αλλά και τη διαγραφή των αρχείων initial.dat και final.dat που περιέχουν αντίστοιχα το πρόβλημα και τη λύση του, όπως και αρχεία μετρήσεων μετά από κάθε τρέξιμο, για την αποφυγή περιττών στοιχείων.

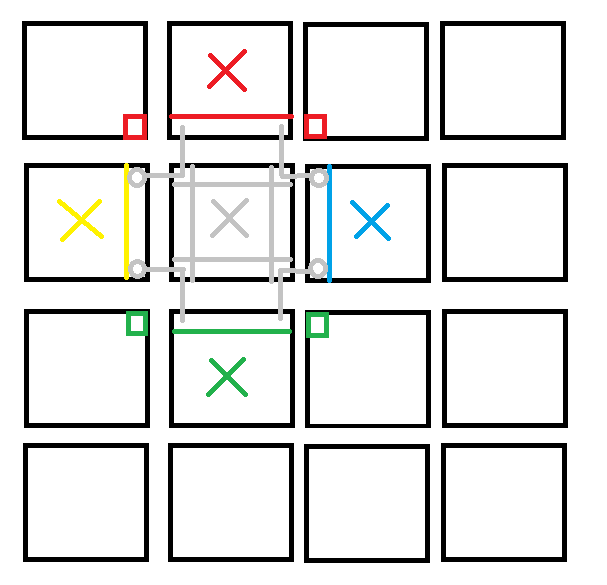
**Σχεδιασμός Διαμοιρασμού Δεδομένων**

Δημιουργούνται όσες διεργασίες όσες επιλεχθούν από τον χρήστη, εκ των οποίων η πρώτη αναλαμβάνει το ρόλο του MASTER . Ο ΜASTER διαμοιράζει και αποστέλλει τις πληροφορίες στις υπόλοιπες διεργασίες που έχουν το ρόλο του WORKER. Πραγματοποιείται διαμοιρασμός σε Βlock και χρήση της καρτεσιανής τοπολογίας (με περιοδικότητα).

Προκειμένου να επιλυθεί το πρόβλημα είναι αναγκαίος ο υπολογισμός των εσωτερικών στοιχείων ενός Block (Independent\_Update) , των περιμετρικών (Dependent\_Update) και φυσικά των σωστών διαγώνιων (UpdateDiag).

Στο σχήμα δίπλα φαίνεται ο τρόπος επικοινωνίας μεταξύ των Block στα οποία έχουμε διαμοιράσει το πρόβλημα.

Στέλνονται εκτός από τις 4 πλευρές (περιμετρικά) κάθε Block και οι 2 πάνω και οι 2 κάτω διαγώνιοι. Το γκρι Block στο παράδειγμα λαμβάνει τα κυκλωμένα στοιχεία από το κίτρινο και το μπλε και τα προωθεί (Send) αντίστοιχα. Στη συνέχεια μέσα από το πράσινο και το κόκκινο θα μπορέσει να γίνει και η σωστή ενημέρωση των διαγώνιων όπως απαιτείται από το πρόβλημα (8 γείτονες).

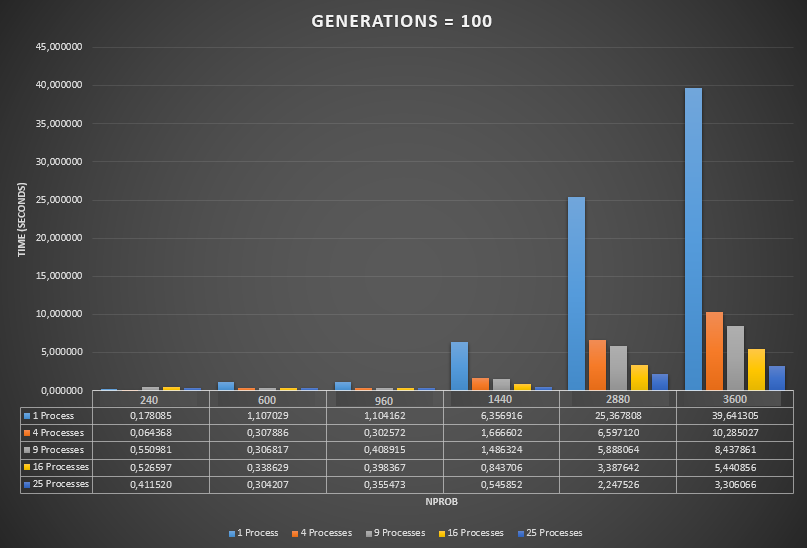
Για να αποφύγουμε την επικοινωνία της κάθε διεργασίας με 8 γείτονες αποφασίσαμε για κάθε διεργασία αρχικά να στέλνει στους πάνω και κάτω γείτονές της την πάνω και κάτω γραμμή της αντίστοιχα έτσι ώστε όταν στείλουμε στους δεξιά και αριστερά γείτονες τις αντίστοιχες στήλες να έχουμε στη διάθεσή μας και τις 2 (πάνω και κάτω δεξιά ή αριστερά αντίστοιχα) γωνίες που χρειάζεται το γειτονικό μπλοκ.Έτσι αποφύγαμε την επικοινωνία της κάθε διεργασίας με 8 γείτονες και στην υλοποίησή μας επικοινωνεί μόνο με 4 (πάνω,κάτω,δεξιά και αριστερά) και δε χρειάζεται επικοινωνία με τους διαγώνιους γείτονες.

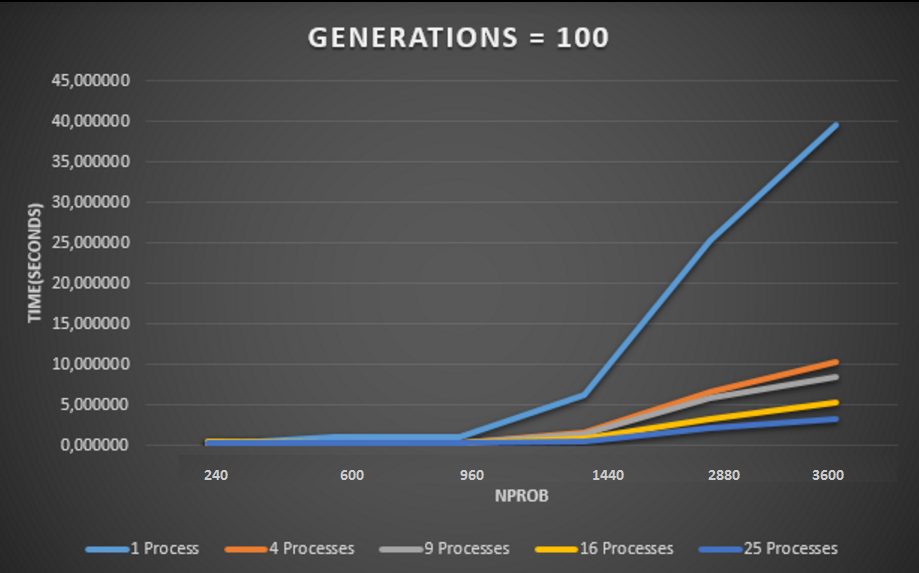
**Σχεδιασμός και υλοποίηση MPI κώδικα**

Ο MASTER λειτουργεί όπως αναφέρθηκε παραπάνω, ενώ οι WORKERS εκτελούν επαναλήψεις όσες το πλήθος των GENERATIONS και επικοινωνούν ασύγχρονα με τις γειτνιάζουσες διεργασίες για τον υπολογισμό του σωστού αποτελέσματος. Ακόμα γίνεται η χρήση datatypes για την αποφυγή πολλαπλών αντιγραφών που θα επέφερε ιδιαίτερα σημαντικό κόστος στην αποστολή των στηλών των υποπινάκων (subarrays). Η λήψη της πληροφορίας στους WORKERS από τον MASTER γίνεται μέσω της MPI\_Irecv, την οποία περιμένουμε να ολοκληρωθεί πριν αρχίσουμε τις πράξεις για κάθε γενιά, καλύπτοντας το συγκεκριμένο χρονικό διάστημα με αρχικοποιήσεις των πινάκων και των request που θα χρησιμοποιήσουμε. Κατά τα GENERATIONS λαμβάνουμε την αριστερή και δεξιά στήλη όσο γίνεται ο υπολογισμός των εσωτερικών στοιχείων (Independent\_Update) και στη συνέχεια ετοιμάζουμε ανάλογα τις προωθήσεις των διαγώνιων και συνεχίζουμε με την αποστολή τους. Κάνουμε τον υπολογισμό των περιμετρικών στοιχείων (Dependent\_Update) και τέλος αφού λάβουμε τις σωστές διαγώνιες με την ενημέρωση τους (UpdateDiag). Σημαντικό είναι να αναφέρουμε ότι έγινε χρήση του Derived Datatype MPI\_CHAR που περιέχει ακέραιους από 0 έως 127 μιας και χρειαζόμαστε μόνο 0 ή 1 για το πρόβλημα μας, οπότε δεσμεύουμε το μικρότερο δυνατό Datatype.

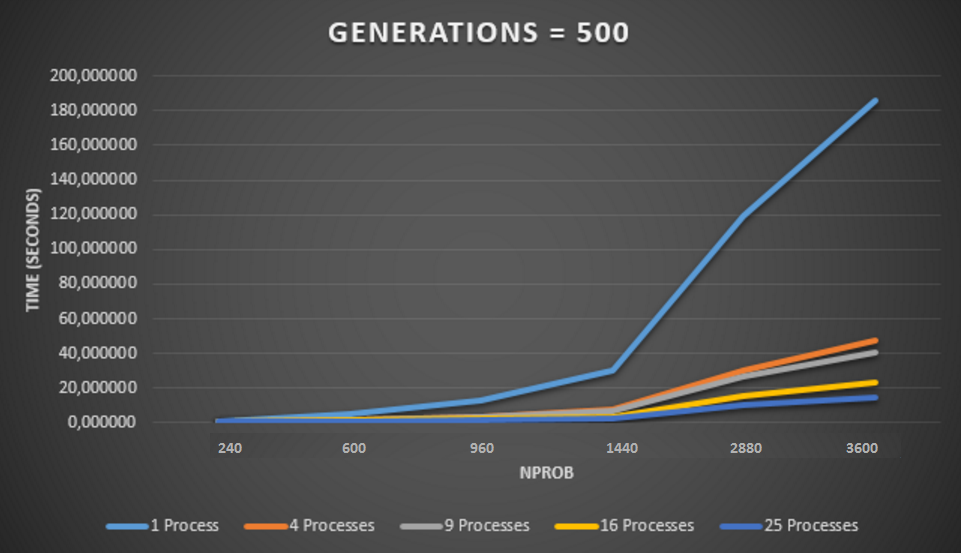
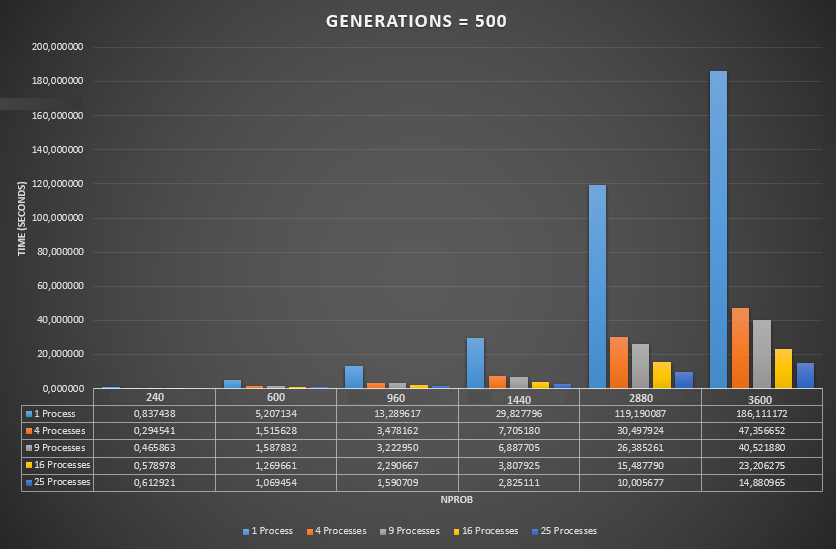
**Παρουσίαση αποτελεσμάτων**

Σημειώνεται ότι οι παρακάτω μετρήσεις έγιναν στο μηχάνημα linux01 βραδινές ώρες για να υπάρχει περιορισμένη κινητικότητα χρηστών. Επίσης έγιναν το ίδιο βράδυ ώστε να μην υπάρξουν αλλαγές στο αρχείο machines και τα 15 εν λειτουργία μηχανήματα που διέθετε.

****

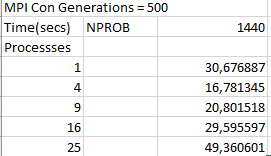
****Εύκολα συμπεραίνουμε ότι ειδικά για τα μεγάλα προβλήματα όσο ανεβαίνει ο αριθμός των διεργασιών τόσο πέφτει ο χρόνος εκτέλεσης. Στα πιο μικρά προβλήματα όπως είναι το 240, 600 επειδή και ο αριθμός των Generations είναι μόνο 100 βλέπουμε ότι δεν υπάρχει μεγάλη διαφορά όταν αλλάζουμε τον αριθμό των διεργασιών κάτι που είναι λογικό.

Στο δεύτερο διάγραμμα βλέπουμε τη μεγάλη κλίση της γραμμής της μιας διεργασίας η οποία ανεβαίνει με μεγάλο ρυθμό όσο ανεβαίνει το μέγεθος του προβλήματος. Ενώ για μεγαλύτερο αριθμό διεργασιών η κλίση της γραμμής πέφτει όλο και περισσότερο!

**** ****

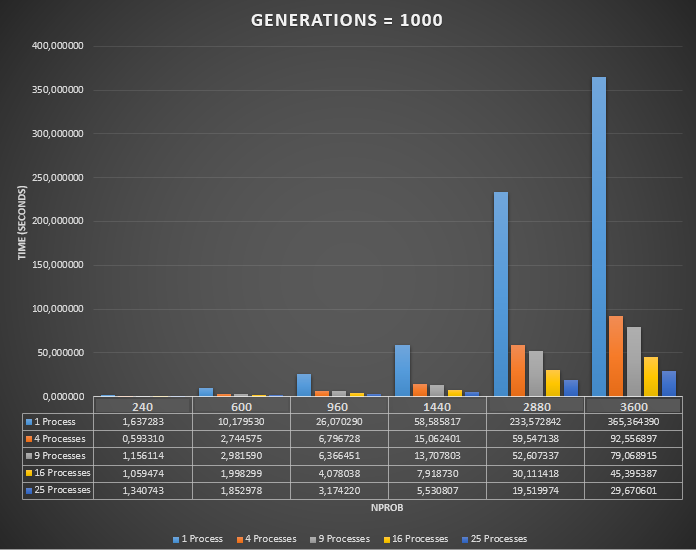
Συνεχίζοντας τις μετρήσεις για 500 Generations έχουμε τα αποτελέσματα που φαίνονται παραπάνω. Βλέπουμε ότι όλες οι τιμές κυμαίνονται σε σχεδόν 5πλάσια μεγέθη με τα αντίστοιχα στα 100 Generations. Τα αποτελέσματα που παίρνουμε είναι τα ίδια μιας και σε ίδιο NPROB όσο αυξάνουμε τις διεργασίες ο χρόνος μειώνεται.

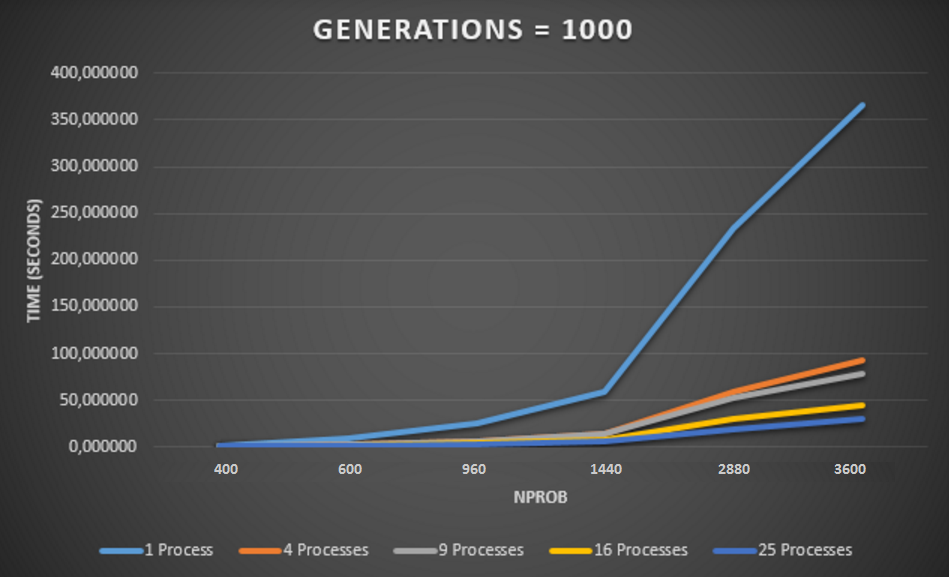
Με σύγκλιση στα 500 Generations με μέγεθος προβλήματος 1440 έχουμε



Οι χρόνοι είναι αρκετά αυξημένοι σε αντιστοιχία με τους χρόνους χωρίς σύγκλιση, λόγω των επιπλέον συγκρίσων.

Όσον αφορά τις μετρήσεις για 1000 Generations παρατηρούμε διπλασιασμό των τιμών σε σύγκριση με τα αντίστοιχα αποτελέσματα στα 500 Generations, όπως φαίνεται και παρακάτω:

****

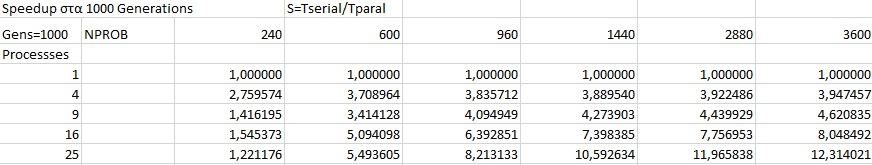
****

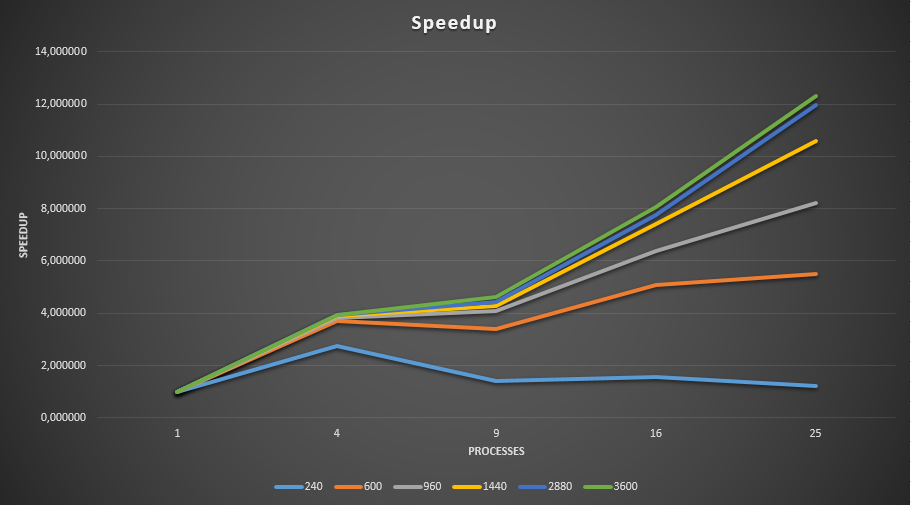
Για να γίνουν πιο ορατά τα αποτελέσματα υπολογίζονται παρακάτω το Speedup και το Efficiency για 1000 Generations

**Speedup**

Ως επιτάχυνση Speedup ορίζουμε το λόγο του χρόνου του σειριακού προγράμματος προς το χρόνο του παράλληλου. Με άλλα λόγια Speedup = Tserial / Tparallel , με Tserial ο σειριακός χρόνος και Tparallel ο παράλληλος χρόνος.

Έτσι για τα αντιστοιχα μεγέθη προβλήματος που χρησιμοποιήσαμε έχουμε τα εξής :

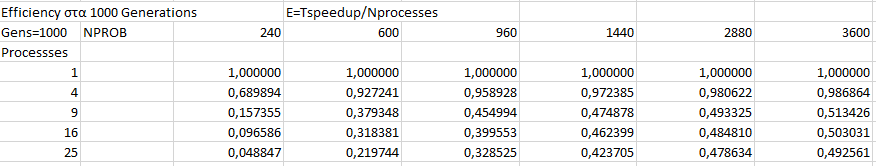




*Παρατηρούμε ότι με εξαίρεση το μικρό πρόβλημα 240x240 όπου η επικοινωνία μεταξύ των διεργασιών φαίνεται να επηρεάζει την απόδοση αφού για 4 διεργασίες βλέπουμε πολύ καλύτερη απόδοση απ ότι με 9 , 16 και 25 , κατα τα άλλα στα μεγαλύτερα προβήματα η συμπεριφορά του προγράμματος όσο αυξάνουν οι διεργασίες είναι όλο και καλύτερη.*

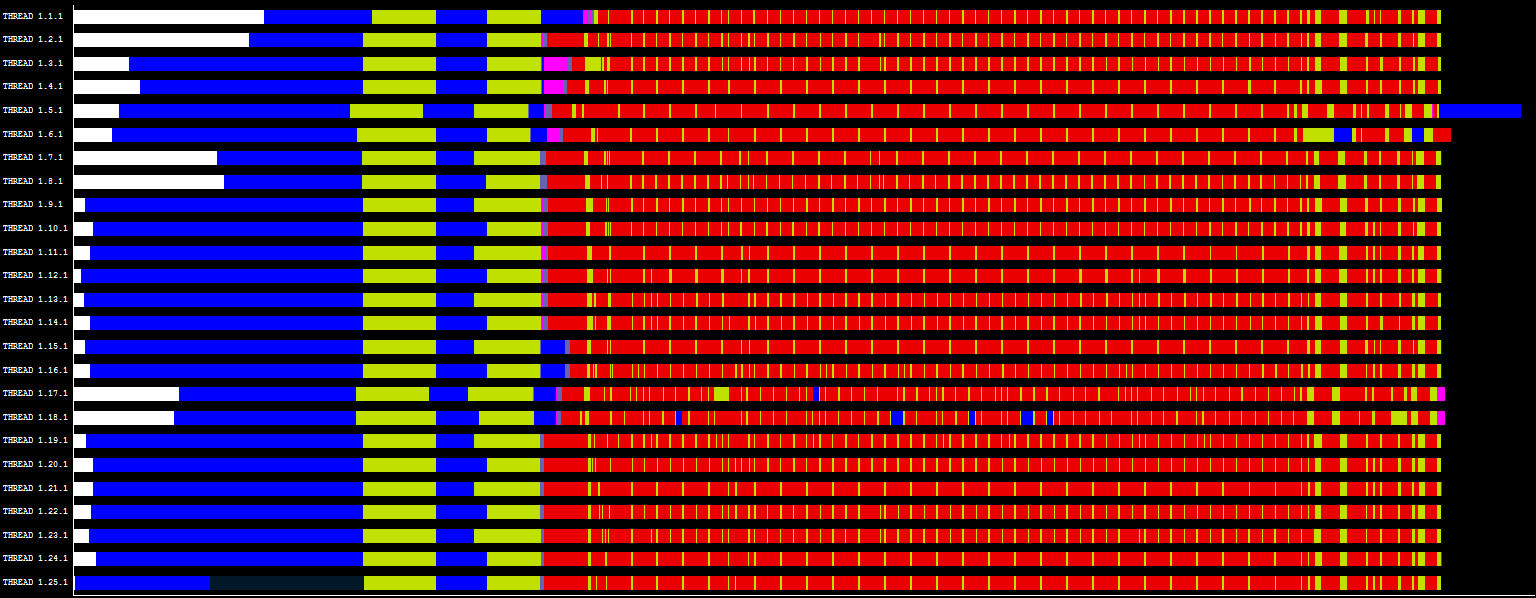
**Efficiency**

Επειδή το Toverhead (έστω ο παράπλευρος χρόνος), αυξάνεται ανάλογα με το πλήθος των πυρήνων p, περιμένουμε μείωση της επιτάχυνσης S όσο αυξάνεται το πλήθος των πυρήνων p, άρα και ο λόγος S / p μειώνεται. Ως αποτελεσματικότητα ορίζεται ο παραπάνω λόγος της επιτάχυνσης προς το πλήθος των πηρύνων. Συνεπώς E = S / p = (Tserial / Tparallel) / p = Tserial / (p \* Tparallel) . Τα άντιστοιχα αποτελέσματα παρουσιάζονται παρακάτω:



*Παραπάνω βλέπουμε το efficiency να μειώνεται με την αύξηση των διεργασιών, και αυτό οφείλεται στο overhead.*

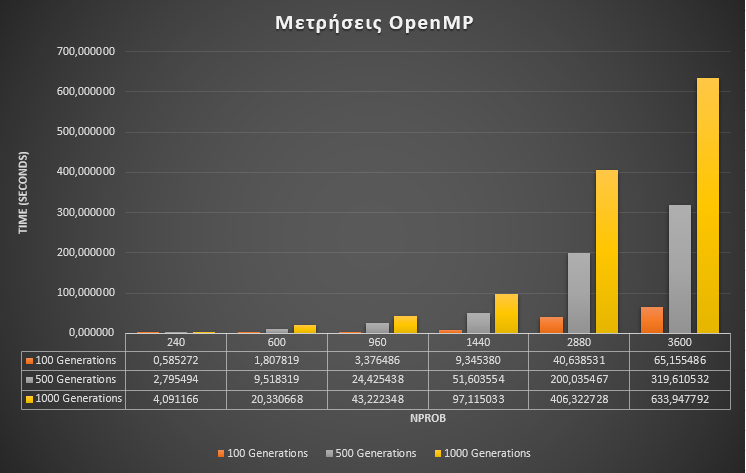
Η χρήση του paraver μας οδηγεί στο παρακάτω διάγραμμα

****

όπου οι χρωματισμοί αντιστοιχίζονται ως εξής:

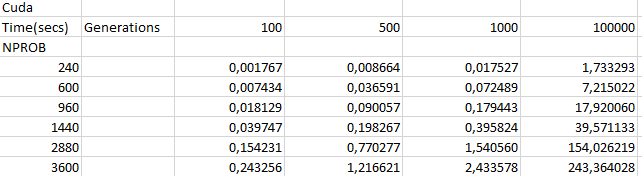


**Openmp**

Συμπεριλήφθηκαν στο πρόγραμμα του MPI, στο αρχείο functions.c τα #pragma omp για να επιτύχουμε τη ζητούμενη παραλληλία. Έτσι διαθέτουμε τα δεδομένα του grid μας που διαχειρίζονται τα thread.****

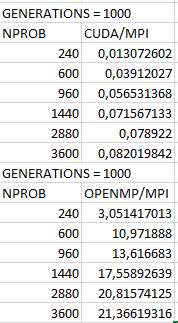
**Cuda**

Όπως κανείς περιμένει η δουλεία γίνεται πιο αποτελεσματικά με τη χρήση της υπολογιστικής δύναμης της GPU. Έτσι έχουμε τα ακόλουθα αποτελέσματα:



**Συμπεράσματα**

Αισθητές γίνονται οι διαφορές ανάμεσα στο MPI σε σχέση με Openmp και Cuda αντίστοιχα μέσα από τα παρακάτω διαγράμματα για τα εκαστοτε μεγέθη προβλήματος στα 1000 Generations:



**Βιβλιογραφία**

**Διαφάνειες και Ηλεκτρονικές Διαλέξεις του μαθήματος**

<http://mpi.deino.net/mpi_functions>

http://www.mpich.org/static/docs/v3.2/www3

<http://beige.ucs.indiana.edu/I590/node100.html>