Πανεπιστήμιο Θεσσαλίας
Πρόγραμμα Σπουδών
Μηχανικών Προφορικής

Πτυχιακή Εργασία

Μορφές κατανεμημένου και παράλληλου προγραμματισμού με την γλώσσα C και εφαρμογή αυτών στο πρόβλημα των N-σωμάτων

Σέρρης Φίλιππος

Επιβλέπων Καθηγητής: Σάββας Ηλίας

Λάρισα, 2019

Περίληψη

Στην παρούσα πτυχιακή μελετάται το πρόβλημα των N-σωμάτων σε ένα σύστημα χωρίς συγκρούσεις. Για την επίλυση του προβλήματος χρησιμοποιούνται τρείς μέθοδοι. Αρχικά υλοποιείτε η σειριακή επίλυση, έπειτα υλοποίηση μέσω της MPI, της OpenMP και τον συνδυασμό αυτών. Τέλος παρουσιάζεται η υλοποίηση μέσω της κάρτας γραφικών με την χρήση της CUDA. Ο κώδικας που είναι υπεύθυνος για την επίλυση του προβλήματος είναι γραμμένος στην γλώσσα προγραμματισμού C. Για κάθε μια από τις μεθόδους επίλυσης έχει γίνει πληθώρα εκτελέσεων με σκοπό την χρονομέτρηση και σύγκριση μεταξύ αυτών σε διάφορες καταστάσεις. Σκοπός είναι να γίνουν εμφανείς οι διαφορές μεταξύ αυτών και να ώστε να αποδειχτεί ποια είναι η πιο κατάλληλη για το παρόν πρόβλημα.

Forms of distributed and parallel programming with C and their appliance to N-body problem

Serris Filippos

Abstrac

In the present thesis is being studied the N-body problem in a system without collisions. Foe the solution of this problem is being used tree methods. At first is implemented a serial solution, next an MPI implementation, an OpenMP and the combination of the two. At last the problem is implemented with use of the graphics card and CUDA. The code that is responsible for the solution of this problem is being written in C programming language. For every one of the solution methods there have been multiple executions with the purpose of measuring and comparing the execution times between them in different situations. The purpose of all this is to be clear the differences between them and to prove which is more suitable for the present problems solution.

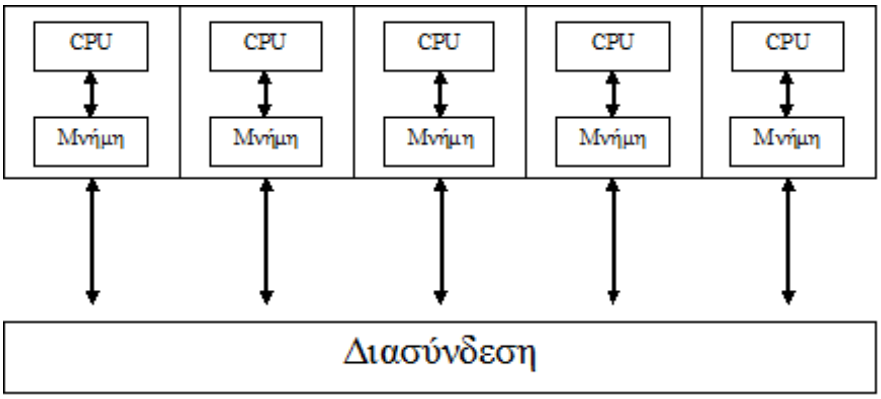
Περιεχόμενα

1 Εισαγωγή

Με την πάροδο του χρόνου τα προβλήματα που πρέπει να αντιμετωπιστούν γίνονται ολοένα και πιο περίπλοκα έτσι υπάρχει η ανάγκη να βρεθούν τρόποι ώστε να αντιμετωπισθούν. Οι υπολογιστές ήρθαν να καλύψουν αυτό το κενό καθώς παρέχουν στον άνθρωπο μεγαλύτερες ταχύτητες και επεξεργαστική ισχύ. Στην παρούσα εργασία θα φανεί πως μέσω της χρήσης των υπολογιστών κάτι τέτοιο μπορεί να επιτευχθεί. Πιο συγκεκριμένα θα παρουσιαστούν τα βασικά στοιχεία του κατανεμημένου προγραμματισμού με την χρήση της MPI, καθώς και σημεία που απαιτούν προσοχή κατά την υλοποίηση κώδικα αυτή. Στην συνέχεια θα γίνει αναφορά στα συστήματα κοινόχρηστης μνήμης και την OpenMP. Έπειτα θα επακολουθήσει ο συνδυασμός των δύο αυτών τεχνικών. Σκοπός αυτού να γίνει κατανοητό ο τρόπος με τον οποίο μπορούν αυτές οι δύο τεχνικές να συνδυαστούν αποδοτικά για την καλύτερη αντιμετώπιση διάφορων προβλημάτων. Τέλος θα παρουσιαστεί πως μπορεί να υλοποιηθεί η λύση ενός προβλήματος με την χρήση της κάρτας γραφικών και την CUDA. Θα παρουσιαστούν τα βασικά τμήματα και σημεία που χρειάζονται προσοχή μέσω διάφορων παραδειγμάτων. Στο κυρίως θέμα της πτυχιακής εργασίας θα χρειαστεί ο συνδυασμός του κλάδου των Μαθηματικών της Φυσικής καθώς και της Πληροφορικής για την για την μελέτη ενός προβλήματος αστροφυσικής που αφορά τις δυνάμει μεταξύ πλανητών ενός ηλιακού συστήματος το οποίο ονομάζεται Πρόβλημα των N-σωμάτων. Αφού γίνει περιγραφή της φύσης αυτού του προβλήματος θα παρουσιαστούν οι βασικές εξισώσεις οι οποίες είναι απαραίτητες για την υλοποίηση του για την επίλυση μέσω της Άμεσης μεθόδου. Πιο συγκεκριμένα θα δειχθούν οι εξισώσεις που καθορίζουν την δύναμη μεταξύ των σωμάτων που αποτελούν το σύστημα που μελετάται καθώς και πως θα υπολογιστούν οι αρχικές θέσεις και ταχύτητες αυτών. Τέλος γνωρίζοντας όλα αυτά θα γίνει η προγραμματιστική προσέγγιση του προβλήματος και επίλυση αυτού με τις μεθόδους που τον κεφαλαίων που προηγούνται. Σκοπός οι χρονομέτρηση των προσομοιώσεων για διάφορο πλήθος επαναλήψεων και σωμάτων και σύγκριση των αποτελεσμάτων.

2 Προγραμματισμός με την MPI

Η MPI δεν αποτελεί μια ακόμη γλώσσα προγραμματισμού, αντιθέτως ορίζεται ως μια βιβλιοθήκη συναρτήσεων που καλούνται από την ανάλογη γλώσσα προγραμματισμού μεταξύ των C, C++ και Fortran που χρησιμοποιείτε για την επίλυση του ενός προβλήματος [3]. Η MPI ανήκει στην οικογένεια των συστημάτων κατανεμημένης μνήμης, πιο συγκεκριμένα λέγοντας σύστημα κατανεμημένης μνήμης γίνεται λόγος για ένα σύστημα που αποτελείται από ένα σύνολο ζευγών επεξεργαστή-μνήμης, αυτά συνδέονται μεταξύ τους μέσω ενός δικτύου για την μεταφορά δεδομένων. Από εδώ και πέρα τέτοιου είδους ζεύγη θα καλούνται διεργασίες. Πιο συγκεκριμένα η σχέση αυτή μπορεί να γίνει πιο κατανοητή βλέποντας την εικόνα που ακολουθεί (Εικόνα 1).



Εικόνα 1. Σύστημα κατανεμημένης μνήμης [3]

Αποτέλεσμα αυτής της σχέσης είναι πως τα δεδομένα ενός προβλήματος θα πρέπει να είναι γνωστά σε όλες τις διεργασίες που συμμετέχουν, αυτό με την σειρά του δημιουργεί την ανάγκη επικοινωνίας μεταξύ αυτών. Η MPI παρέχει λύση στο πρόβλημα αυτό μέσω διαφόρων συναρτήσεων. Οι συναρτήσεις αυτές έχουν ως σκοπό την μεταφορά δεδομένων μεταξύ των διεργασιών την MPI. ΟΙ μεταφορές αυτές ονομάζονται επικοινωνίες και χωρίζονται σε δύο κατηγορίες, επικοινωνία από σημείο σε σημείο και συλλογική επικοινωνία[3,5]. Αυτό που διαχωρίζει αυτές τις δύο κατηγορίες είναι το πλήθος διεργασιών που συμμετέχουν στην μεταφορά των δεδομένων.

Οι επικοινωνία από σημείο σε σημείο αποτελείται από συναρτήσεις που είναι υπεύθυνες για την μεταφορά δεδομένων μεταξύ δύο διεργασιών. Σε αυτή την περίπτωση η μία από αυτές

αναλαμβάνει τον ρόλο του αποστολέα και η επόμενη των ρόλο του παραλήπτη. Αντίθετα η συλλογική επικοινωνία αποτελείται από συναρτήσεις που εμπλέκουν περισσότερες απο δύο διεργασίες είτε στον ρόλο του αποστολέα είτε στον ρόλο του παραλήπτη.

Πέραν της επικοινωνίας μεταξύ των διεργασιών σημαντικό κομμάτι είναι ο τρόπος που θα γίνει ο κατανομή των δεδομένων στις διεργασίες που συμμετέχουν. Κάτι τέτοιο είναι πολύ σημαντικό καθώς διαφορετικά προβλήματα απαιτούν διαφορετικές μεθόδους επίλυση, αποτέλεσμα αυτού πως η κατανομή των δεδομένων θα διαφέρει. Ανάλογα με την περίπτωση που αντιμετωπίζεται απαιτείται διαφορετική προσέγγιση του προβλήματος αλλά και χρήση διαφορετικών συναρτήσεων για να επιτευχθούν τα επιθήματα αποτελέσματα.

2.1 Βασικές έννοιες της MPI

Με την χρήση της MPI τα προβλήματα πλέων δεν αντιμετωπίζονται με τον συμβατικό τρόπο επίλυσης όπου ο κώδικας ενός προβλήματος εκτελούταν σειριακά από μία και μόνο διεργασία. Πλέων στην διάθεση του χρήστη υπάρχει ένα πλήθος διεργασιών για την υλοποίηση των προβλημάτων που με την σωστή διαχείριση τους η επίλυση αυτών μπορεί να βελτιωθεί αισθητά. Στη συνέχεια φαίνεται ένα απλό τμήμα κώδικα (Εικόνα 2) όπου κάθε διεργασία εκτυπώνει ένα μήνυμα στην οθόνη με σκοπό να γίνει πιο κατανοητή η λειτουργία της MPI.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int size,rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  printf("\nProcess no %d of %d processes\n",rank,size);

  MPI_Finalize();

  return 0;
}
```

Εικόνα 2. Εκτύπωση μηνύματος με την MPI

Με την χρήση της MPI δεν αλλάζει κάτι στην δομή του προγράμματος σε σχέση με την είδη γνωστή. Ξεκινώντας συμπεριλαμβάνονται οι αναγκαίες βιβλιοθήκες, ακολουθεί ο κύριο κώδικας με την δήλωση μεταβλητών και του ανάλογους υπολογισμούς, τέλος η εντολή return. Για την χρήση της MPI απαιτείται να συμπεριληφθεί η βιβλιοθήκη mpi.h όπως φαίνεται και πιο πάνω στην εικόνα του κώδικα. Με αυτό τον τρόπο παραχωρείται η πρόσβαση στις συναρτήσεις που διαθέτει η MPI, στους τύπους των μεταβλητών που χρησιμοποιεί και σε άλλα βασικά τμήματα που είναι απαραίτητα για την επίλυση του προβλήματος. Στο κυρίως μέρος του κώδικα το πρώτο πράγμα παρατηρείται είναι η εντολή MPI_Init(). Αρμοδιότητα της εντολής αυτής είναι να εκκινεί την παραλληλία της MPI διευθετώντας ζητήματα που πρέπει να τακτοποιηθούν πριν την οποιαδήποτε κλήση των συναρτήσεων της [3,5]. Παράδειγμα μία λειτουργία της εντολής αυτής είναι η δέσμευση των απαραίτητων πόρων του συστήματος ώστε να διασφαλιστεί η ομαλή λειτουργία του προγράμματος. Η εντολή MPI_Finalize() στο τέλος του κώδικα κάνει ακριβώς την αντίθετη εργασία, είναι υπεύθυνη για την ενημέρωση του συστήματος πως η MPI δεν είναι πλέων απαραίτητη και έτσι να γίνουν οι ανάλογες ενέργειες. Αποτέλεσμα της εντολής αυτής είναι η αποδέσμευση των οποιονδήποτε πόρων είχαν δεσμευτεί με την χρήση της εντολής MPI_Init() στην αρχή του προγράμματος[3,5]. Καλό θα ήταν καμία συνάρτηση που έχει σχέση με την MPI να μην κληθεί έξω από την εμβέλεια των δύο αυτών εντολών έτσι ώστε να διασφαλιστεί η ομαλότητα στην εκτέλεση του προγράμματος.

Δύο βασικές συναρτήσεις της MPI που δίνουν χρήσιμες πληροφορίες για τις διεργασίες του συστήματος είναι οι MPI_Comm_size() και MPI_Comm_rank() [3,5,9]. Η συνάρτηση MPI_Comm_size() είναι αυτή όπου εκχωρεί σε μια μεταβλητή όπου στην συγκεκριμένη περίπτωση είναι η μεταβλητή size το σύνολο των διεργασιών που βρίσκονται στην διάθεση της MPI. Με την σειρά της η συνάρτηση MPI_Comm_rank() εκχωρεί στην ανάλογη μεταβλητή rank μία μοναδική τιμή που διαχωρίζει την κάθε διεργασία από τις υπόλοιπες του συνόλου size. Κάτι τέτοιο βοηθά στον διαχωρισμό των διεργασιών κατά την υλοποίηση διαφόρων πράξεων όπως θα φανεί και στα παραδείγματα που ακολουθούν. Για να γίνει ποιο κατανοητή η σημασία των εντολών αυτών ακολουθεί ένα παράδειγμα. Σε μία υπολογιστική συστοιχία που αποτελείται από δέκα διεργασίες, τότε η μεταβλητή size είναι ίση με δέκα (size=10) και κάθε διεργασία έχει μια ξεχωριστή μεταβλητή rank με τιμή από μηδέν έως και εννέα ανάλογα με την θέση που έχει στην υπολογιστική συστοιχία. Οι τιμές αυτές είναι

χρήσιμες για την επικοινωνία μεταξύ των διεργασιών καθώς μέσω αυτών μπορούν να προσδιοριστούν οι αποστολείς και παραλήπτες των ανάλογων μηνυμάτων που ορίζονται στις επικοινωνίες όπως θα φανεί στην συνέχεια. Στις δύο συναρτήσεις που περιγράφθηκαν υπάρχει το όρισμα MPI_COMM_WORLD, αυτό που αντιπροσωπεύει το εν λόγο τμήμα της εντολής είναι ο επικοινωνιτής της MPI [3,5], δηλαδή το σύνολο των διεργασιών που επικοινωνούν. Έχοντας όλε αυτές της πληροφορίες η έξοδος του πιο πάνω κώδικα μπορεί να γίνει εύκολα κατανοητή όπως παρουσιάζεται στην εικόνα που ακολουθεί (Εικόνα 3).

```
felix@felix-desktop:~/Desktop/thesis/code/examples/mpi$ mpicc message.c -o message
felix@felix-desktop:~/Desktop/thesis/code/examples/mpi$ mpiexec -np 4 ./message

Process no 0 of 4 processes

Process no 2 of 4 processes

Process no 3 of 4 processes

Process no 1 of 4 processes
```

Εικόνα 3. Έξοδος κώδικα Εικόνας 2

2.2 Επικοινωνία μεταξύ διεργασιών

Βασικό κομμάτι της MPI είναι η επικοινωνία μεταξύ των διεργασιών που απαρτίζουν το σύνολο size καθώς αυτό αποτελεί βασικό τμήμα της επίλυσης ενός προβλήματος. Οι επικοινωνίες που πραγματοποιούνται μέσω της MPI χωρίζονται σε δύο κατηγορίες όπου βασικό ρόλο στον διαχωρισμό τους αποτελεί το πλήθος των διεργασιών που συμμετέχουν σε αυτές. Η πρώτη κατηγορία αφορά την επικοινωνία μεταξύ δύο και μόνο διεργασιών όπου η μία αποστέλλει δεδομένα και η δεύτερη παραλαμβάνει. Αντίστοιχα στη δεύτερη κατηγορία απαιτείται η συμμετοχή περισσότερων από δύο διεργασιών. Πιο συγκεκριμένα με αυτόν τον τρόπο είτε μία διεργασία αποστέλλει δεδομένα προς όλες τις υπόλοιπες και αυτές με την σειρά τους παραλαμβάνουν, είτε όλες οι διεργασίες αποστέλλουν δεδομένα προς τις υπόλοιπους και αντίστοιχα αυτές παραλαμβάνουν.

2.2.1 Επικοινωνίες από σημείο προς σημείο

Η πιο απλή επικοινωνία μεταξύ διεργασιών είναι αυτή που συμμετάσχουν μόνο δύο από αυτές καθώς η μία αναλαμβάνει τον ρόλο του αποστολέα και η δεύτερη τον ρόλο του παραλήπτη. Σε αυτή την εκδοχή τον ρόλο του αποστολέα αναλαμβάνει η διεργασία με τιμή rank ίση με μηδέν (rank=0), συνήθως είναι αυτή καθώς αυτή αναλαμβάνει την αλληλεπίδραση με τον χρήστη και έτσι έχει γνώση όλων των δεδομένων του προβλήματος. Η εντολή που είναι υπεύθυνη για την αποστολή των δεδομένων από την μια διεργασία στην άλλη ονομάζεται MPI_Send() όπως φαίνεται και στο παράδειγμα που ακολουθεί.

MPI_Send(&x, 1, MPI_INT, i, 0, MPI_COMM_WORLD); [3,5,9]

- &x: η διεύθυνση στη μνήμη του στοιχείου που αποστέλλεται
- 1: το πλήθος των στοιχείων που αποστέλλονται από την μια διεργασία στην άλλη
- MPI_INT: ο τύπος των δεδομένων που αποστέλλονται μεταξύ των διεργασιών
- i: τιμή rank κάθε διεργασίας για τον καθορισμό της διεργασίας που παραλαμβάνει
- 0: ετικέτα μηνύματος
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Με την χρήση της παραπάνω εντολής γίνεται μεταφορά του στοιχείο x από μία διεργασία που θα οριστεί να εκτελέσει την εντολή προς την διεργασία i που αναμένει να παραλάβει την μεταβλητή αυτή. Όπως η διεργασία αποστολές πρέπει να εκτελέσει την εντολή MPI_Send() με σκοπό την αποστολή κάπου δεδομένου, έτσι και η διεργασία με ον ρόλο του παραλήπτη πρέπει να εκτελέσει την ανάλογη εντολή παραλαβής. Η εντολή που αυτή ονομάζεται MPI_Recv() όπως φαίνεται και στο παράδειγμα που ακολουθεί.

MPI_Recv(&y, 1, MPI_INT, i, 0, MPI_COMM_WORLD); [3,5,9]

- &y: διεύθυνση στην μνήμη που θα αποθηκευτεί το στοιχείο x
- 1: ποσότητα δεδομένων προς παραλαβή
- MPI_INT: τύπος δεδομένων που θα λάβει η διεργασία
- i: διεργασία η οποία αποστέλλει τα δεδομένα
- 0: ετικέτα μηνύματος
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Όπως γίνεται κατανοητό οι δύο αυτές εντολές είναι άρρηκτα συνδεδεμένες για να επιτευχθεί σωστά η μεταβίβαση δεομένων μεταξύ των διεργασιών. Κάθε πλευρά αναλαμβάνει ένα ρόλο με τις αντίστοιχες οδηγίες ώστε τα δεδομένα να διαμοιραστούν με επιτυχία. Κάτι το οποίο συναντάτε στις συναρτήσεις της MPI είναι οι τύποι των δεδομένων που χρησιμοποιεί. Όπως φαίνεται και από τα παραδείγματα που προηγηθήκαν υπάρχουν διαφορές σε σχέση με τον τρόπο που ίδια η C συμβολίζει τους ανάλογους τύπους. Μια γρήγορη αντιστοιχία σε σχέση με τους τύπους μεταβλητών της γλώσσας C φαίνεται στον πίνακα που ακολουθεί (Πίνακας 1).

Τύποι μεταβλητών C	**Τύποι μεταβλητών MPI**
Char	MPI_CHAR
int	MPI_INT
long int	MPI_LONG
Float	MPI_FLOAT
Double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE

Πίνακας 1. Αντιστοιχία τύπων μεταβλητών [3,8]

Μέχρι στιγμής με την χρήση των εντολών MPI_Send() και MPI_Recv() φαίνεται πως γίνεται η μεταφορά μίας και μόνο μεταβλητής μεταξύ των διεργασιών. Παρ' όλα αυτά υπάρχει η δυνατότητα μεταφοράς ενός συνόλου δεδομένων μεταξύ διεργασιών, όπως για παράδειγμα

ένας πίνακας δεδομένων. Ένας πίνακας μπορεί να μεταφερθεί μεταξύ των διεργασιών με μικρές αλλαγές στην σύνταξη των εντολών που προηγήθηκαν. Όταν μεταφέρονται πίνακες έναντι μεταβλητών δεν είναι πλέων αναγκαία η χρήση του συμβόλου & για την διεύθυνση της μεταβλητής στην μνήμη, αρκεί μόνο το όνομα του πίνακα. Μεταφέροντας πίνακες το σημείο όπου αντιστοιχεί στο πλήθος των δεδομένων που αποστέλλονται μεταξύ των διεργασιών πρέπει να αλλάξει. Αντί να γίνεται αναφορά μίας μεταβλητής γίνεται αναφορά του πλήθος των κελιών του πίνακα που αποστέλλεται μεταξύ των διεργασιών. Έστω για παράδειγμα πως αντί της μεταβλητής x πρέπει να μεταφερθεί ο πίνακας A μεγέθους δέκα στοιχείων από κάποια διεργασία στον αντίστοιχου μεγέθους πίνακα B κάποιας άλλης. Τότε η εντολές αποστολής και παραλαβής θα έχουν την ακόλουθη μορφή.

```
MPI_Send(A, 10,  MPI_INT, i, 0, MPI_COMM_WORLD);
MPI_Recv(B, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

Σε περίπτωση που το σύνολο των διεργασιών που συμμετάσχουν και είναι απαραίτητο να λάβουν τα δεδομένα είναι μεγαλύτερο της μίας τότε απλώς θα πρέπει από την πλευρά του αποστολέα η εντολή αποστολής να εκτελεστεί αρκετές φορές ώστε να καλύψει το πλήθος αυτών. Η μόνη διαφορά που θα υπάρξει είναι πως κάθε φορά θα πρέπει να αλλάζει η τιμή που καθορίζει του παραλήπτη.

2.2.2 Συναρτήσεις συλλογικής επικοινωνίας

Όπως και με την επικοινωνία από σημείο σε σημείο έτσι και εδώ με την χρήση της συλλογικής επικοινωνίας σκοπός είναι η μεταφορά δεδομένων μεταξύ των διεργασιών. Αυτό που αλλάζει είναι πως υπάρχει η δυνατότητα να αποσταλούν δεδομένα σε πολλές διεργασίας μαζικά με την χρήση μόνο μιας εντολής. Κάτι τέτοιο είναι αρκετά αποδοτικό όσον αφορά μεγάλο πλήθος διεργασιών σε σχέση με τον τρόπο που αναφέρθηκε στην προηγούμενη ενότητα. Κάτι τέτοιο είναι εφικτό με την χρήση της συνάρτησης MPI_Bcast() που δίνει την δυνατότητα μεταφοράς δεδομένα από μια διεργασία προς όλες τις υπόλοιπες. Ένα ακόμη θετικό με την χρήση της συνάρτησης αυτής είναι πως η διεργασία που παραλαμβάνει δεν πρέπει να εκτελέσει κάποια εντολή για την παραλαβή των δεδομένων αυτών. Πιο αναλυτικά η εντολή φαίνεται στο παράδειγμα που ακολουθεί.

MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD); [9]

- &x: διεύθυνση της μεταβλητής προς αποστολή στην μνήμη
- 1: πλήθος των δεδομένων που αποστέλλονται
- MPI_INT: τύπος δεδομένων της MPI
- 0: τιμή rank της διεργασία που αποστέλλει τα δεδομένα
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Όπως και με την MPI_Send() έτσι και εδώ υπάρχει η δυνατότητα μεταφοράς ολόκληρων πινάκων μεταξύ των διεργασιών με την εφαρμογή μικρών αλλαγών. Όπως υπάρχει η δυνατότητα αποστολής δεδομένων από μία διεργασία προς όλες τις υπόλοιπες μπορεί να συμβεί και το αντίστροφο. Από πλευράς τους κάθε διεργασία μπορεί να συλλέξει δεδομένα από όλες τις υπόλοιπες διεργασίες με την χρήση της εντολής MPI_Reduce(). Στο παράδειγμα που ακολουθεί φαίνεται η σύνταξη της εντολής.

MPI_Reduce(&x, &y, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); [3,5,9]

- &x: διεύθυνση τοπικής μεταβλητής
- &y: διεύθυνση μεταβλητής συσσώρευσης αποτελέσματος
- 1: ποσότητα δεδομένων που θα αποσταλούν από κάθε διεργασία
- MPI_INT: τύπος δεδομένων της MPI
- MPI_SUM: τελεστής πράξης
- 0: διεργασία η οποία συλλέγει τα αποτελέσματα
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Όπως φαίνεται και από το παράδειγμα πιο πάνω πέραν από την συλλογή δεδομένων από τις διεργασίες υπάρχει η δυνατότητα να τελεστεί και μια πράξει μεταξύ των δεδομένων που συλλέγονται. Έστω για παράδειγμα πως ζητείται το άθροισμα των στοιχείων ενός πίνακα και αφού η κάθε διεργασία υπολογίσει το άθροισμα που της αναλογεί, τότε αυτά τα

αποτελέσματα είναι ανάγκη να αθροιστούν ξανά για να προκύψει το τελικό αποτέλεσμα. Όπως είναι φανερό πέραν από την άθροιση η MPI δίνει την δυνατότητα να τελεστούν διάφορες πράξεις μεταξύ των επιμέρους αποτελεσμάτων. Στον πίνακα που ακολουθεί (Πίνακας 2) φαίνονται οι πράξεις που μπορούν να τελεστούν μέσω της εντολής MPI_Reduce().

Τελεστής	**Λειτουργία**
MPI_SUM	Άθροιση
MPI_PROD	Γινόμενο
MPI_MAX	Μέγιστη τιμή
MPI_MIN	Ελάχιστη τιμή
MPI_LAND	Λογικό ΚΑΙ – σύζευξη
MPI_LOR	Λογικό Η – διάζευξη

Πίνακας 2. Τελεστές πράξης της εντολής Reduce [9]

Σε διάφορα προβλήματα μπορεί να μην είναι αναγκαίο όλες οι διεργασίες να έχουν γνώση όλων των δεδομένων του προβλήματος. Το να αποσταλούν τα απαραίτητα δεδομένα ένα προς ένα θα ήταν αρκετά χρονοβόρο. Αντιθέτως το να αποσταλούν όλα τα δεδομένα προς όλες τις διεργασίες ενώ δεν είναι απαραίτητο θα ήταν σοβαρή σπατάλη μνήμης. Σε τέτοια προβλήματα αρκεί η χρήση της εντολής MPI_Scatter(). Δουλειά αυτής είναι να τεμαχίσει τα δεδομένα σε τμήματα μήκους ανάλογου των διεργασιών που συμμετάσχουν. Έπειτα να αποστέλλει αυτά τα δεδομένα στις υπόλοιπες διεργασίες. Σκοπός της εντολής πως κάθε διεργασία θα λάβει ίδιο πλήθος δεδομένων διαφορετικού περιεχομένου. Μετά τον διαχωρισμό το πρώτο τμήμα δεδομένων αποστέλλεται στην πρώτη διεργασία σύμφωνα με την τιμή rank, ακολουθεί το δεύτερο τμήμα το οποίο απευθύνετε στην δεύτερη σε σειρά διεργασία και ούτω καθεξής. Με αυτό τον τρόπο κάθε διεργασία έχει ένα διαφορετικό σύνολο δεδομένων να εργαστεί, ταυτόχρονα υπάρχει μια τάξη στα δεδομένα ώστε να είναι γνωστό ποια διεργασία είναι υπεύθυνη για ποιο τμήμα δεδομένων. Η δομή της εντολής φαίνεται αναλυτικά στο παρακάτω παράδειγμα.

MPI_Scatter(A, N/size, MPI_INT, Aw, N/size, MPI_INT, 0, MPI_COMM_WORLD); [5,9]

- A: πίνακας του οποίου το περιεχόμενα θα αποσταλεί στις υπόλοιπες διεργασίες
- N/size: το πλήθος των δεδομένων που θα αποσταλούν σε κάθε διεργασία
- MPI_INT: τύπος δεδομένων της MPI (εμφανίζεται μια φορά για τα δεδομένα αποστολής και μια δεύτερη για τα δεδομένα που παραλαμβάνονται)
- Aw: πίνακας υποδοχής των N/size πλήθους δεδομένων του πίνακα A
- 0: διεργασία αποστολής
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Με αυτό τον τρόπο με την χρήση μίας και μόνο εντολής διαφορετικά δεδομένα μεταφέρονται σε όλους τους συμμετέχοντες κόμβους και έτσι υπάρχει μια πιο αποδοτική λύση. Η MPI διαθέτει μια αντίστοιχη εντολή όσων αναφορά την συλλογή δεδομένων από τους διάφορες διεργασίες η οποία είναι η MPI_Gather(). Όπως και με την MPI_Scater() έτσι κι εδώ μία διεργασία έχει αναλάβει να συλλέγει δεδομένα από τις υπόλοιπες. Η σειρά με την οποία γίνεται η συλλογή των δεδομένων αυτών είναι αντίστοιχη με αυτής της εντολής MPI_Scatter(), βασισμένη στην τιμή rank της εκάστοτε διεργασίας. Πιο αναλυτικά η MPI_Gather() φαίνεται παρακάτω στο παράδειγμα που ακολουθεί.

MPI_Gather(Cw, N/size, MPI_INT, C, N/size, MPI_INT, 0, MPI_COMM_WORLD); [5,9]

- Cw: πίνακας από όπου αποστέλλονται τα δεδομένα
- N/size: πλήθος δεδομένων
- MPI_INT: τύπος δεδομένων της MPI
- C: πίνακας συλλογής αποτελεσμάτων
- 0: διεργασία παραλαβής
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Μέχρι τώρα με τις συλλογικές επικοινωνίες περιγράφηκαν καταστάσεις όπου μία διεργασία αναλαμβάνει τον ρόλο είτε του αποστολέας είτε του παραλήπτη σε μία επικοινωνία. Σε προβλήματα που υπάρχει η ανάγκη δεδομένα που έχουν μεταβληθεί από άλλες διεργασίες να

είναι γνωστά σε όλες τις υπόλοιπες πρέπει να γίνει η χρήση διαφορετικών συναρτήσεων. Η MPI δίνει την δυνατότητα σε μερικές από τις συναρτήσεις που περιγράφηκαν πιο πάνω να έχουν μια επικοινωνία όπου όλες οι διεργασίες μπορούν να συλλέγουν δεδομένα από όλες τα υπόλοιπες διεργασίες. Μια τέτοια εντολή είναι η MPI_Allreduce(), λειτουργεί ακριβώς όπως και η MPI_Reduce() που περιγράφηκε πιο πάνω με την μόνη διαφορά πως πλέων η συλλογή δεν γίνετε από μία διεργασία, αντιθέτως γίνετε από το σύνολο αυτών που απαρτίζουν την υπολογιστική συστοιχία. Οι ομοιότητες και οι διαφορές μεταξύ των δύο εντολών μπορούν να γίνουν πιο εμφανείς μέσω του παρακάτω παραδείγματος.

MPI_Allreduce(&x, &y, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD); [3,9]

- &x: διεύθυνση τοπικής μεταβλητής
- &y: διεύθυνση μεταβλητής συσσώρευσης αποτελέσματος
- 1: ποσότητα δεδομένων που θα αποσταλούν από κάθε διεργασία
- MPI_INT: τύπος δεδομένων της MPI
- MPI_SUM: τελεστής πράξης
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Οι δύο εντολές είναι αρκετά όμοιες μεταξύ τους με την διαφορά πως από την MPI_Allreduce() απουσιάζει η μεταβλητή που καθορίζει τον προορισμός τον δεδομένων που συλλέγονται. Αυτό είναι φυσικό αφού πλέων τα αποτελέσματα δεν έχουν μία συγκεκριμένη διεργασία ως προορισμό αλλά όλο το σύνολο αυτών. Σε συνέχεια αυτής της ακολουθίας για την εντολή MPI_Gather() υπάρχει η αντίστοιχη MPI_Allgather() έτσι ώστε να μπορεί να εξυπηρετήσει όλες τις διεργασίες. Παρακάτω φαίνεται η μορφή η οποία ακολουθεί η εντολή MPI_Allgather().

MPI_Allgather(Cw, N/size, MPI_INT, C, N/size, MPI_INT, MPI_COMM_WORLD); [5,9]

- Cw: πίνακας από όπου αποστέλλονται τα δεδομένα
- N/size: πλήθος δεδομένων
- MPI_INT: τύπος δεδομένων της MPI
- C: πίνακας συλλογής αποτελεσμάτων
- MPI_COMM_WORLD: το σύνολο των διεργασιών που απαρτίζουν των κόσμο επικοινωνίας της MPI

Όπως και πριν έτσι και εδώ παρατηρείται πως η μόνη διαφορά μεταξύ των δύο εντολών είναι η έλλειψη αναφοράς της διεργασίας παραλαβής, το γεγονός αυτό συμβαίνει για τον ίδιο λόγο όπως και στην MPI_Gather().

2.3 Διαμοιρασμός δεδομένων μεταξύ κόμβων

Όπως παρουσιάστηκε και στην αρχή του κεφαλαίου εκτός από τις επικοινωνίες μεταξύ των διεργασιών υπάρχουν και όλοι παράγοντες που καθορίζουν την αποδοτικότητα ενός κώδικα. Για την αποδοτική επίλυση ενός προβλήματος θα πρέπει κάθε διεργασία να αναλαμβάνει τον ίδιο φόρτο εργασίας σε σχέση με τις υπόλοιπες. Σε περιπτώσεις που δεν είναι απαραίτητη η μεταφορά μεταξύ των διεργασιών τότε τα δεδομένα να μην αποστέλλονται για την εξοικονόμηση μνήμης.

2.3.1 Αποστολή τμημάτων δεδομένων προς εξοικονόμηση μνήμης

Έχοντας ένα πρόβλημα και γνωρίζοντας πως αυτό αποτελείται από τυχαία δεδομένα. Ο μόνος τρόπος για τη διασφάλιση της σωστής κατανομής αυτών μεταξύ των διεργασιών είναι πως κάθε διεργασία θα αναλάβει το ίδιο πλήθος δεδομένων σε σχέση με τις υπόλοιπες. Η MPI διαθέτει τις ανάλογες συναρτήσεις που έχουν την δυνατότητα να διαμοιράζουν τα δεδομένα με τέτοιο τρόπο όπως φάνηκε και στις προηγούμενες ενότητες. Βασική προϋπόθεση αποτελεί να οριστεί σωστά η συνθήκη με την οποία θα πραγματοποιηθεί ο τεμαχισμός των δεδομένων που θα αναλάβει η κάθε διεργασία.

Έστω πως ζητείται να βρεθεί το άθροισμα δύο διανυσμάτων Α και Β όπου οι πληροφορίες τους είναι αποθηκευμένες στου ανάλογους πίνακες οι οποίοι απαρτίζονται από τυχαίες τιμές.

Δεν υπάρχει τρόπος να γνωρίζουμε τι δεδομένα περιέχει ο κάθε πίνακας εκ των προτέρων. Για να είναι σίγουρο πως κάθε διεργασία θα αναλάβει ανάλογο μέρος το προβλήματος πρέπει κάθε μία από αυτές να αναλάβει τμήματα ίδιου μεγέθους. Σε περίπτωση που το μέγεθος των πινάκων N διαιρείται ακέραια με το πλήθος των διεργασιών size που συμμετέχουν τότε δεν υπάρχει κάποιο πρόβλημα. Διαμοιράζοντας τα δεδομένα βάσει του μεγέθους N/size οι πίνακες A και B χωρίζονται σε ίσα μεγέθη σύμφωνα με το πλήθος των διεργασιών. Έπειτα κάθε τμήμα πληροφοριών αποστέλλεται σε διαφορετικές διεργασίες. Η υλοποίηση του πιο πάνω προβλήματος φαίνεται στην εικόνα που ακολουθεί (Εικόνα 4). Οι αρχικοί πίνακες συμβολίζονται με τα σύμβολα A,B και C, ενώ οι πίνακες υποδοχής με τα σύμβολα Aw,Bw και Cw.

```
40    MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
41
42    Aw=malloc((N/size)*sizeof(int));
43    Bw=malloc((N/size)*sizeof(int));
44    Cw=malloc((N/size)*sizeof(int));
45
46    MPI_Scatter(A,N/size,MPI_INT,Aw,N/size,MPI_INT,0,MPI_COMM_WORLD);
47    MPI_Scatter(B,N/size,MPI_INT,Bw,N/size,MPI_INT,0,MPI_COMM_WORLD);
48
49    ts=MPI_Wtime();
50
51    for(i=0;i<N/size;i++)
52    {
53      Cw[i]=Aw[i]+Bw[i];
54    }
55
56    MPI_Gather(Cw,N/size,MPI_INT,C,N/size,MPI_INT,0,MPI_COMM_WORLD);
```

Εικόνα 4. Πρόγραμμα άθροισης διανυσμάτων

Όπως φαίνεται στην εικόνα το μέγεθος N αποστέλλεται προς όλους τις διεργασίες με την εντολή MPI_Bcast(), τα υπόλοιπα δεδομένα αποστέλλονται προς όλες τις διεργασίες μέσω της εντολής MPI_Scatter() σε τμήματα N/size από την διεργασία μηδέν. Κάθε διεργασία εκ των προτέρων έχει προετοιμάσει τους πίνακες υποδοχής δεδομένων του ανάλογου μεγέθους (N/size) και αφού όλοι οι κόμβοι υπολογίσουν το τμήμα του πίνακα C που τους αναλογεί, το αποθηκεύουν στον πίνακα Cw και έπειτα γίνεται συλλογή αυτών των αποτελεσμάτων με την MPI_Gather() στον τελικό πίνακα C της διεργασίας μηδέν. Στο παρόν παράδειγμα υπάρχει το εξής πρόβλημα, σε περίπτωση που το πλήθος των δεδομένων δεν διαιρείται τέλεια με το πλήθος των διεργασιών τότε τα δεδομένα που υπολείπονται δεν αποστέλλονται και δεν υπολογίζονται από καμία από τις συμμετέχουσες διεργασίες. Ο πιο κοινός τρόπος για την αντιμετώπιση ενός τέτοιου προβλήματος φαίνεται στην εικόνα (Εικόνα 5) που ακολουθεί.

```
58    if(rank==0)
59    {
60
61      // Σε περίπτωση που η διαίρεση N/size έχει υπόλοιπο το επιπλέων φορτίο υπολογίζεται απο τον κόμβο rank=0
62      if(N%size!=0)
63      {
64        for(i=size*(N/size);i<N;i++)
65        {
66          C[i]=A[i]+B[i];
67        }
68      }
69
70      te=MPI_Wtime();
71
72      printf("\n\nTime required is: %.3f\n",te-ts);
73
74      free(A);
75      free(B);
76      free(C);
77    }
```

Εικόνα 5. Υπολογισμός δεδομένων ατελούς διαίρεσης

Αυτό που κάνει το παραπάνω τμήμα κώδικα είναι να φροντίζει πως σε περίπτωση που υπάρχει ατελείς διαίρεση μεταξύ δεδομένων και διεργασιών τότε αυτά υπολογίζονται από την διεργασία μηδέν και αποθηκεύονται απευθείας στον πίνακα C. Αυτός ο τρόπος είναι ενδεικτικός και διαφορετικά προβλήματα μπορεί να απαιτούν διαφορετικούς τρόπους διαχείρισης των δεδομένων όπως θα φανεί στην συνέχεια. Η επιλογή της διεργασίας μηδέν έγινε λόγου ότι στο συγκεκριμένο πρόβλημα είναι αυτή που έχει γνώση των υπολειπόμενων δεδομένων, σε περίπτωση που η διεργασία αυτή διαφέρει τότε θα πρέπει η τιμή αυτή να διαφέρει στον κώδικα που προηγήθηκε.

2.3.2 Αποστολή όλων των δεδομένων

Υπάρχουν προβλήματα που για την υλοποίηση τους απαιτούν από όλες τις διεργασίες να έχουν πλήρη γνώση των δεδομένων του προβλήματος, σε αντίθετη περίπτωση δεν μπορούν να αποφέρουν τα επιθυμητά αποτελέσματα. Αποτέλεσμα αυτού τα δεδομένα να μην μπορούν να διαχωριστούν σε τμήματα και να πρέπει το σύνολο αυτών να σταλούν σε κάθε διεργασία ξεχωριστά. Για παράδειγμα έστω πως υπάρχει ο πίνακας Α και πως για κάθε κελί αυτού πρέπει να υπολογιστεί το άθροισμα του προηγούμενου και επόμενου κελιού από αυτό. Δεν υπάρχει κάποια μέθοδος που θα μπορέσει να διαχωρίσει τον πίνακα Α έτσι είναι απαραίτητα όλες οι διεργασίες να τον γνωρίζουν ολόκληρο. Στον κώδικα που ακολουθεί (Εικόνα 6) φαίνεται πως μπορούν να αντιμετωπιστούν προβλήματα τέτοιων απαιτήσεων.

```
37    for(i=rank;i<N;i+=size)
38    {
39      if(i-1>=0 && i+1<=N-1)
40      {
41        B[i]=A[i-1]+A[i]+A[i+1];
42      }
43      else if(i-1<0)
44      {
45        B[i]=A[N-1]+A[i]+A[i+1];
46      }
47      else if(i+1>N-1)
48      {
49        B[i]=A[i-1]+A[i]+A[0];
50      }
51      else
52      {
53        B[i]=0;
54      }
55    }
```

Εικόνα 6. Διαχείριση δεδομένων με την χρήση της εντολής for

Με την χρήση της εντολής for όπως φαίνεται πιο πάνω κάθε διεργασία σύμφωνα με την τιμή rank που της αντιστοιχεί αναλαμβάνει να υπολογίσει συγκεκριμένα κελί του πίνακα. Για παράδειγμα εφαρμόζοντας τον πιο πάνω κώδικα σε μια υπολογιστική συστοιχία που αποτελείται από πέντε διεργασίες (size=5) και ο πίνακας C που θα αποθηκευτούν τα αποτελέσματα είναι μεγέθους ίσο με 10 (N=10), τότε κάθε διεργασία είναι υπεύθυνη για το τμήμα του πίνακα C όπως αυτό φαίνεται στον πίνακα που ακολουθεί άλλα έχει την ανάγκη να γνωρίζει επιπλέων δεδομένα. Πιο αναλυτικά όλα αυτά φαίνονται στον πίνακα που ακολουθεί (Πίνακας 3).

Διεργασία	Τιμή i	Απαραίτητα κελιά
0	0, 4, 8	(9,1), (3,5), (7,9)
1	1, 5, 9	(0,2), (4,6), (8,0)
2	2, 6	(1,3), (5,7)
4	3, 7	(2,4), (6,8)

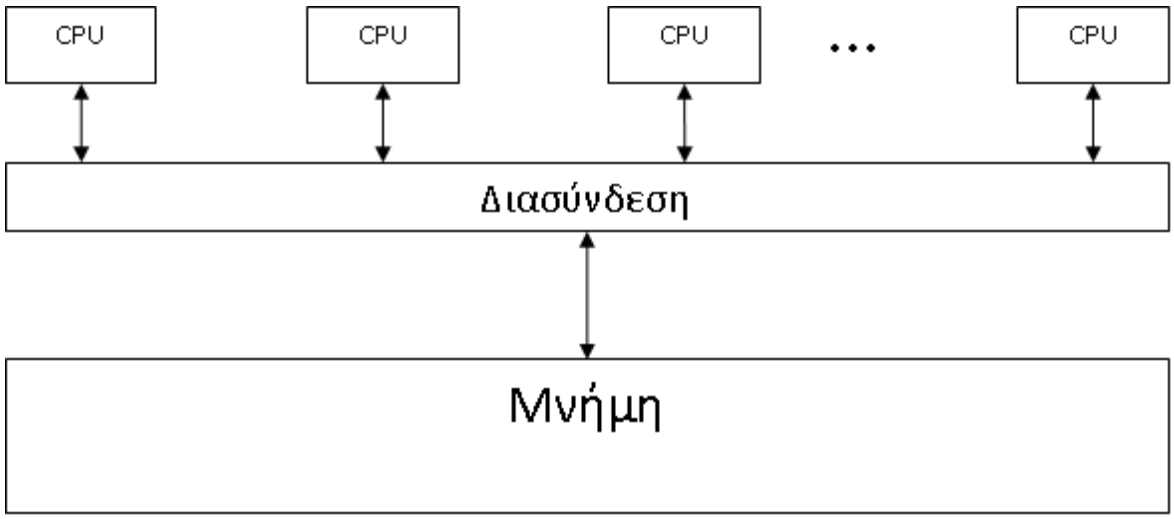
Πίνακας 3. Φορτίο διεργασιών

Ο τρόπος με τον οποίο λειτουργεί η συγκεκριμένη for είναι πως η αφετηρία κάθε διεργασίας στον υπολογισμό του B μέσω του A είναι η ίδια η τιμή rank αυτής. Έτσι κάθε διεργασία

υπολογίζει διαφορετικά δεδομένα από όλες τις υπόλοιπες. Έπειτα προσθέτοντας στην τιμή i το μέγεθος size που είναι ίδιο για όλες τις διεργασίες, κάθε διεργασία προσανατολίζεται σε διαφορετική θέση στον πίνακα A και B. Με την χρήση της εντολής αυτής δεν υπάρχει η ανάγκη για να γίνει έλεγχος στο αν έχει απομείνει κάποιος υπολογισμός καθώς η εντολή for συνεχίζει όπως και μια απλή for έως και την τιμή N-1 των πινάκων.

3 Προγραμματισμός με την OpenMP

Όπως και η MPI έτσι και η OpenMP είναι μία μορφή παράλληλου προγραμματισμού. Τα γράμματα MP αντιπροσωπεύουν των όρο πολυεπεξεργασία καθώς πλέων γίνεται αναφορά σε ένα σύνολο πυρήνων που διαθέτει ένας επεξεργαστής [3]. Πιο συγκεκριμένα η OpenMP δεν χρησιμοποιεί ζεύγη πυρήνων-μνήμης όπως η MPI, αλλά μία συλλογή πυρήνων που μπορούν να προσπελάσουν ολόκληρη την μνήμη η οποία είναι κοινή για το σύνολο αυτών, από εδώ και πέρα αναφορά σε αυτούς θα γίνεται με τον όρο κόμβοι κάτι αντίστοιχο των διεργασιών που χρησιμοποιούταν στο προηγούμενο κεφάλαιο. Η OpenMP ανήκει στα συστήματα κοινόχρηστης μνήμης καθώς η μνήμη που χρησιμοποιείται για την επίλυση ενός προβλήματος είναι κοινή μεταξύ των κόμβων. Ο όρος αυτός μπορεί να γίνει πιο κατανοητός μέσω της εικόνας που ακολουθεί (Εικόνα 7).



Εικόνα 7. Σύστημα κοινόχρηστης μνήμης [3]

Αφού πλέων οι κόμβοι που καλούνται να λύσουν το πρόβλημα πρέπει να προσπελάσουν την ίδια μνήμη η ανάγκη για επικοινωνία μεταξύ αυτών και αποστολή δεδομένων καταργείται. Ένα πρόβλημα που δημιουργείτε λόγο της κατάστασης αυτής είναι πως μεταβλητές που κάθε κόμβος είναι ανάγκη να διαχειρίζεται εξ ολοκλήρου θα πρέπει να δοθούν με διαφορετικό τρόπο ώστε να μπορέσει να γίνει ο κατάλληλος διαχωρισμός για την αποφυγή σφαλμάτων. Η OpenMP έχει φροντίσει για την δημιουργία κατάλληλων μέσων για την διαχείριση των μεταβλητών αυτών αλλά και για την διαμοίραση του όγκου των υπολογισμών μεταξύ των διεργασιών. Ευθύνη του προγραμματιστή είναι η βελτιστοποίηση των υπόλοιπων τμημάτων

ενός κώδικα που καθορίζουν την παραλληλία με σκοπό την γρήγορη και ορθή επίλυση ενός προβλήματος. Για τον σκοπό αυτό υπάρχουν διάφορες συναρτήσεις που παρέχονται από την OpenMP για την κάθε ανάγκη του προβλήματος.

3.1 Βασικές έννοιες της OpenMP

Σε πρώτη φάση η OpenMP ακολουθεί την ίδια λογική όπως και η MPI, διασπά το κυρίως πρόβλημα σε μικρότερα υποπροβλήματα τα οποία αναλαμβάνουν διαφορετικοί κόμβοι. Μια απλή εκδοχή ενός παραλλήλου προγράμματος που εκτυπώνει ένα μήνυμα από όλες τις διεργασίες φαίνεται στην εικόνα που ακολουθεί (Εικόνα 8).

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int *argc, char **argv)
5  {
6    #pragma omp parallel
7    {
8      printf("\nNode no %d of %d nodes\n",omp_get_thread_num() ,omp_get_num_threads());
9    }
10
11   return 0;
12 }
```

Εικόνα 8. Εκτύπωση μηνύματος με την OpenMP

Το πρώτο πράγμα που είναι απαραίτητο κομμάτι για την χρήση της OpenMP όπως φάνηκε και με την MPI είναι να συμπεριληφθεί η βιβλιοθήκη που δίνει πρόσβαση σε αυτή και τις λειτουργίες της όπως φαίνεται στον πιο πάνω κώδικα. Στην συνέχεια ακολουθεί η συνάρτηση omp_get_thread_num() [3,5]η οποία είναι υπεύθυνη για να αντιστοιχήσει τον κάθε κόμβο με μια συγκεκριμένη τιμή ώστε να μπορεί να αναγνωριστεί από τους υπολοίπους και τον χρήστη όταν αυτό είναι αναγκαίο. Έπειτα ακολουθεί η εντολή omp_get_num_threads() [3,5] η οποία εμφανίζει το πλήθος των πυρήνων που είναι διαθέσιμοι στο εκάστοτε σύστημα. Για την εκκίνηση της παραλληλίας αρκεί η χρήση της εντολής #pragma omp parallel. Κάθε κομμάτι κώδικα που εκτελείται παράλληλα πρέπει να είμαι στην εμβέλεια ενός #parallel block εντολών για να λειτουργήσει σωστά, τα υπόλοιπα αναλαμβάνονται από την ίδια την OpenMP. Στην εικόνα που ακολουθεί (Εικόνα 9) φαίνεται το αποτέλεσμα του κώδικα που παρουσιάστηκε στην εικόνα που προηγήθηκε (Εικόνα 8).

```
felix@felix-desktop:~/Desktop/thesis/code/examples/openMP$ gcc message.c -o message -fopenmp
felix@felix-desktop:~/Desktop/thesis/code/examples/openMP$ ./message

Node no 4 of 8 nodes

Node no 3 of 8 nodes

Node no 5 of 8 nodes

Node no 1 of 8 nodes

Node no 6 of 8 nodes

Node no 0 of 8 nodes

Node no 7 of 8 nodes

Node no 2 of 8 nodes
```

Εικόνα 9. Έξοδος κώδικα Εικόνας 8

3.2 Διαμοιρασμός φόρτου με την χρήση της εντολής for

Σε αντίθεση με την MPI η OpenMP διαχειρίζεται με διαφορετικό τρόπο τον παραλληλισμό των προβλημάτων, περικλείοντας το τμήμα του κώδικα που πρέπει να παραλληλιστεί μέσα στα άγκιστρα μίας εντολής for αυτομάτως το πλήθος τον επαναλήψεων διαμοιράζεται μεταξύ των κόμβων του συστήματος και εκτελείται ταυτόχρονα με τα υπόλοιπα [3,4,5]. Αυτό καθιστά τον παραλληλισμό οποιουδήποτε προβλήματος αρκετά απλοϊκό σε σχέση με αυτόν της MPI. Η μέθοδος αυτή καλύπτει και το ενδεχόμενο μίας ατελούς διαίρεσης μεταξύ των επαναλήψεων και των κόμβων του συστήματος, κάτι που δεν ήταν διαθέσιμο με την MPI. Στην εικόνα που ακολουθεί (Εικόνα 10) φαίνεται με πιο τρόπο υλοποιείται ένα τέτοιο τμήμα κώδικα.

```
36
37    #pragma omp parallel for
38    {
39      C[i]=A[i]+B[i];
40    }
41
```

Εικόνα 10. Η εντολή for

Το τμήμα του πιο πάνω κώδικα είναι υπεύθυνο για την παράλληλη άθροιση δύο διανυσμάτων A και B. Ο τρόπος με τον οποίο η εντολή for λειτουργεί για τον παραλληλισμό ενός προβλήματος είναι αρκετά απλώς, διαιρώντας το σύνολο των επαναλήψεων που πρέπει να τελεστούν με τους κόμβους που συμμετέχουν δημιουργούνται πακέτα δεδομένων που με

την σειρά τους κάθε κόμβος αναλαμβάνει προς υπολογισμό. Για παράδειγμα στο πρόβλημα της άθροισης διανυσμάτων όπου οι πίνακες διανυσμάτων A και B είναι μεγέθους δώδεκα στοιχείων (N=12) και το πλήθος των κόμβων που συμμετείχαν είναι ίσο με τρία, η εντολή for θα είχε διαμοιράσει τις πράξεις που θα πρέπει να εκτελεστούν για τον υπολογισμό του C όπως φαίνεται και στον πίνακα που ακολουθεί (Πίνακα 4).

Κόμβος	**Τιμή i**
0	0, 1, 2, 3
1	4, 5, 6, 7
2	8, 9, 10, 11

Πίνακας 4. Διαμοιρασμός εντολής for τέλειας διαίρεσης

Σε συνέχεια του παραδείγματος μπορεί να υπάρχουν προβλήματα όπου οι επαναλήψεις αφήνουν υπόλοιπο όταν διαιρούνται με το σύνολο των κόμβων. Έστω πως σε μία τέτοια περίπτωση το πλήθος των στοιχείων είναι ίσο με δέκα (N=10) και το πλήθος των κόμβων είναι ίσο με τέσσερα, τότε ο φόρτος εργασίας διαμοιράζεται όπως και πιο πάνω με την διαφορά πως οι πρώτοι κόμβοι αναλαμβάνουν τα πλεονάζοντα δεδομένα όπως φαίνεται στον πίνακα πιο κάτω (Πίνακας 5).

Κόμβος	**Τιμή i**
0	0, 1, 2
1	3, 4, 5
2	6, 7
3	8, 9

Πίνακας 5. Διαμοιρασμός εντολής for ατελείς διαίρεσης

Αυτή είναι μία αρκετά χρήσιμη λειτουργία καθώς προλαμβάνει το οποιοδήποτε σφάλμα μπορεί να συμβεί λόγο μίας ατελούς διαίρεσης.

3.3 Οι εντολές shared και private

Το ότι η OpenMP λειτουργεί με κοινή μνήμη μεταξύ των κόμβων δημιουργεί ένα σοβαρό πρόβλημα στην υλοποίηση κάθε προβλήματος. Αν μια διεργασία αλλάξει το περιεχόμενο κάποιας μεταβλητής τότε αυτό δεν μπορεί να ανακτηθεί από τις υπόλοιπες διεργασίες που μπορεί να χρειαζόταν αυτή την πληροφορία για την διεξαγωγή κάποιου υπολογισμού. Για την λύση αυτού του προβλήματος υπάρχουν οι εντολές private() και shared() [4,5] που μπορούν να διαχωρίσουν τέτοιου είδους δεδομένα. Κάνοντας χρήση της εντολής shared() τότε δηλώνουμε πως οποιαδήποτε μεταβλητή υπάρχει εντός των παρενθέσεων είναι κοινή για όλους τους κόμβους του συστήματος, αποτέλεσμα αυτού πως όποια αλλαγή συμβεί σε αυτή θα επηρεάσει κάθε κόμβο αυτού. Αντιθέτως με την χρήση της εντολής private() η OpenMP δημιουργεί μια ξεχωριστή μεταβλητή στην μνήμη για την κάθε κόμβο η οποία είναι δεν είναι προσβάσιμη από τους υπόλοιπους. Στην εικόνα που ακολουθεί (Εικόνα 11) φαίνεται ένα τμήμα κώδικα που υλοποιεί μια τέτοια λειτουργία.

```
28    #pragma omp parallel for private(i) shared(N)
29    {
30      for(i=0;i<N;++)
31      {
32        printf("Node: %d, value of i: %d\n",omp_get_thread_num(),i);
33      }
34    }
```

Εικόνα 11. Χρήση των private & shared

Στον κώδικα αυτό υπολογίζεται το άθροισμα δύο διανυσμάτων όπου κάθε κόμβος αναλαμβάνει να υπολογίσει ένα τμήμα του πίνακα αποτελεσμάτων C. Η μεταβλητή N που αντιπροσωπεύει το μέγεθος των διανυσμάτων είναι κοινή για όλους τους κόμβους, αντίθετα η μεταβλητή i είναι μια ιδιωτική μεταβλητή για κάθε κόμβο του συστήματος, Αυτό είναι που επιτρέπει την διεξαγωγή της παραλληλίας καθώς ένας κόμβος μπορεί να υπολογίζει το κελί του πίνακα C που η τιμή i είναι ίση με πέντε σε (i=5) , ενώ ταυτόχρονα κάποιος άλλος τελεί τον υπολογισμό για την τιμή i ίση με τρία (i=3). Η OpenMP έχει φροντίσει τα ορίσματα N και i μίας εντολής for να έχουν διαχωριστεί κατάλληλα στην μνήμη έτσι ώστε να μην δημιουργηθεί κάποιο πρόβλημα χωρίς την χρήση των εντολών private και shared. Η χρήση των εντολών στο συγκεκριμένο παράδειγμα είναι καθαρά για την επίδειξη της λειτουργίας τους χωρίς όμως αυτό να επηρεάζει την λειτουργία του κώδικα.

3.4 Οι εντολή schedule

Επέκταση της εντολής for για τον τρόπο που θα διαμοιραστούν οι πράξεις ανάμεσα στους κόμβους αποτελεί η εντολή schedule(). Μέσω της χρήσης της εντολής αυτής υπάρχει η δυνατότητα να μετασχηματιστεί ο τρόπος με τον οποίο επιταχύνεται ο παραλληλισμός. Ο τρόπος με τον οποίο αυτό μπορεί να συμβεί είναι πως μετά την for πρέπει να ακολουθήσει η εντολή schedule() με δύο ορίσματα, το πρώτο αντιπροσωπεύει τον τρόπο που θα ακολουθεί για τον διαμοιρασμό και το δεύτερο αφορά το μέγεθος των πακέτων που θα δημιουργηθούν [3,4,5]. Με την χρήση της εντολής αυτής το σύστημα αναθέτει εκ των προτέρων σε κάθε κόμβο πακέτα επαναλήψεων μεγέθους ανάλογο αυτού που θα οριστεί κατά την χρήση της.

3.4.1 Η εντολή static

Θέτοντας το ως πρώτο όρισμα στην εντολή schedule() το λέξη static ακολουθείται η ίδια διαδικασία με την υλοποίηση της παράλληλης εντολής for όπως στην προηγούμενη ενότητα. Η μόνη διαφορά είναι πως πλέων τα δεδομένα δεν αποστέλλονται απευθείας αλλά σε τμήματα συγκεκριμένου μήκους [3,4,5]. Για παράδειγμα ορίζοντας την δεύτερη τιμή ίση με δύο (schedule(static,2)) τότε οι επαναλήψεις του βρόχου for σπάν σε κομμάτια μεγέθους δύο στοιχείων και αντιτίθενται στις διεργασίες, σε περίπτωση που υπολείπονται και άλλες επαναλήψεις η διαδικασία ξεκινά από την αρχή έως ότου αυτές εξαντληθούν. Στην εικόνα που ακολουθεί (Εικόνα 12) φαίνεται πως υλοποιείται η εντολή αυτή.

```
28    #pragma omp parallel for schedule(static,2)
29    {
30      for(i=0;i<N;++)
31      {
32        printf("Node: %d, value of i: %d\n",omp_get_thread_num(),i);
33      }
34    }
```

Εικόνα 12. Η εντολή static

Στον πίνακα που ακολουθεί (Πίνακας 6) φαίνεται πως θα γινόταν η κατανομή σε μια περίπτωση με τρις κόμβους και δώδεκα επαναλήψεις.

Κόμβος	Τιμή i
0	0, 1, 6, 7
1	2, 3, 8, 9
2	4, 5, 10, 11

Πίνακας 6. Διαμοιρασμός με την εντολή schedule(static, 2)

Σε περίπτωση που μετά την εντολή static δεν ακολουθεί κάποια μεταβλητή τότε η τιμή αυτή αντικαθιστάτε αυτομάτως με μία περίπου ίση με το πλήθος των επαναλήψεων δια το σύνολο των κόμβων.

3.4.2 Η εντολή dynamic

Με την ίδια λογική όπως παρουσιάστηκε και πιο πάνω έτσι και εδώ συνδυάζοντας με την χρήση της εντολής dynamic παρέχεται η δυνατότητα να αλλαχθεί ο τρόπος διαμοιρασμού των πράξεων, χωρίζοντας τα σε ομάδες ανεξαρτήτου μήκους ανάλογα με τις ανάγκες του προβλήματος. Το μήκος των ομάδων είναι ανάλογο της τιμής που θα οριστεί στην εντολή dynamic [3,4,5]. Στην εικόνα που ακολουθεί (Εικόνα 13) υπολογίζεται το άθροισμα δύο διανυσμάτων που κάθε κόμβος αναλαμβάνει πακέτα των τριών ως εναρκτήρια εργασία και με το πέρας αναλαμβάνει την επομένη σε σειρά τριάδα προς υπολογισμό.

```
28    #pragma omp parallel for schedule(dynamic,2)
29    {
30      for(i=0;i<N;++)
31      {
32        printf("Node: %d, value of i: %d\n",omp_get_thread_num(),i);
33      }
34    }
```

Εικόνα 13. Η εντολή dynamic

Όπως και πριν σε ένα σύστημα που έχει στην διάθεση του οκτώ κόμβους εφαρμόζοντας την εντολής dynamic ορίζοντας ως μέγεθος των τμημάτων ίσο με δύο και εφαρμόζοντας το πάνω στην άθροιση δύο διανύσματα μεγέθους 30 στοιχείων οι υπολογισμοί ανατέθηκαν στις διεργασίες όπως φαίνεται στον πινάκα που ακολουθεί (Πίνακας 7).

Κόμβος	Τιμή i
0	12, 13
1	0 ,1
2	2, 3
3	6, 7, 16, 17
4	8, 9
5	14, 15
6	4, 5
7	10, 11, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29

Πίνακας 7. Διαμοιρασμός με την εντολή schedule(dynamic, 2)

Ο τρόπος με τον οποίο ανατέθηκαν οι υπολογισμοί στους κόμβους του συστήματος θα διαφέρει σε κάθε εκτέλεση καθώς δεν λαμβάνεται υπόψη η τιμή των διεργασιών όπως σε άλλες περιπτώσεις αλλά το ποιος από αυτούς είναι διαθέσιμος να αναλάβει τους υπολογισμούς. Στον πίνακα που ακολουθεί (Πίνακας 8) φαίνεται πως θα γινόταν οι διαμοιρασμός αν αλλάξει την τιμή που καθορίζει το μήκος τον τμημάτων του προηγούμενου παραδείγματος από 2 σε 3.

Κόμβος	Τιμή i
0	9, 10 ,11
1	15, 16, 17
2	12, 13, 14
3	3, 4, 5, 24, 25, 26, 27, 28, 29
4	6, 7, 8
5	18, 19, 20
6	21, 22, 23
7	0, 1, 2

Πίνακας 8. Διαμοιρασμός με την εντολή schedule(dynamic, 3)

3.5 Οι εντολές critical και single

Ανάλογα με τις ανάγκες κάθε προβλήματος υπάρχει η πιθανότητα μέσα σε ένα παράλληλο τμήμα κώδικα να χρειαστεί η χρήση σειριακών τμημάτων για τον υπολογισμό τον αποτελεσμάτων. Σε περιπτώσεις σαν και αυτές απαραίτητη είναι η χρήση των εντολών critical ή single αναλόγως τι επιδιώκεται. Με την χρήση της οδηγίας critical κάθε κόμβος εκτελεί τον κώδικα που ακολουθεί με την διαφορά πως η εκτέλεση δεν γίνεται παράλληλα αλλά σειριακά, ο κόμβος n ξεκινά την υλοποίηση του κώδικα μόλις ο κόμβος n-1 τελειώσει, με την σειρά τους ακολουθούν και οι υπόλοιποι κόμβοι [3,4,5]. Για παράδειγμα σε ένα πρόγραμμα που αυξάνει την τιμή μίας μεταβλητής a ο κόμβος με την τιμή μηδέν ξεκινά να εκτελεί το τμήμα κώδικα, στην συνέχεια ακολουθούν διαδοχικά και όλοι οι υπόλοιποι κόμβοι. Στην εικόνα που ακολουθεί (Εικόνα 14) φαίνεται υλοποίηση του κώδικα που περιγράφηκε.

```
24    int a=3;
25    int b=a;
26
27    #pragma omp parallel
28    {
29      #progama omp critical
30        a++;
31    }
32
33    printf("\nOriginal a= %d, final b= %d");
```

Εικόνα 14. Η εντολή critical

Αντίθετα η εντολή single απομονώνει το τμήμα κώδικα που την ακολουθεί από ο σύνολο των κόμβων και επιτρέπει μόνο σε έναν την εκτέλεση αυτού [4,5]. Σε σύγκριση με το παράδειγμα πιο πάνω έτσι κι εδώ σε ένα πρόβλημα που αυξάνει την τιμή της μεταβλητής a, η OpenMP θα αναθέσει σε ένα και μόνο κόμβο την εργασία αυτή. Η υλοποίηση του παραδείγματος αυτού με την εντολή single φαίνεται στην εικόνα που ακολουθεί (Εικόνα 15).

```
12    int a=3;
13    int b=a;
14
15    #pragma omp parallel
16    {
17      #pragma omp single
18        a++;
19    }
20
21    printf("\nOriginal a= %d, final b= %d");
```

Εικόνα 15. Η εντολή single

3.6 Η εντολή collapse

Σε διάφορα προβλήματα η επίλυση τους απαιτεί την υλοποίηση ενός εμφωλευμένου for εντός ενός υπάρχοντος for, σε τέτοιες περιπτώσεις εφόσον δεν υπάρχει καμία εξάρτηση δεδομένων που υπολογίζονται το ιδανικό θα ήταν να παραλληλιστούν και οι δύο for εντολές. Ο πιο απλός τρόπος να για να επιτευχθεί κάτι τέτοιο θα ήταν αφού κληθεί η αρχική for εντολή με την ανάλογη σύνταξη για την έναρξη της παραλληλίας το ίδιο ακριβώς να συμβεί και στην δεύτερη εντολή for. Παρόλο που αυτό μοιάζει αρκετά λογικό δεν υπάρχει η δυνατότητα να υλοποιηθεί κάτι τέτοιο με αυτή την μορφή καθώς δεν θα επιστρέψει τις αναμενόμενες τιμές ως αποτελέσματα.

Για την επίλυση αυτού το προβλήματος έχει δημιουργηθεί η εντολή collapse. Αυτό που κάνει η εντολή είναι πως στην περίπτωση που υπάρχουν εμφωλευμένες εντολές for τότε δίνει την δυνατότητα να παραλληλιστούν και οι εσωτερικοί βρόχοι. Σημαντικό προτέρημα της εντολής αυτής είναι πως με την χρήση μίας τιμής εντός των παρενθέσεων της εντολής η παραλληλία εμβαθύνει έως ότου χρειαστεί. Η τιμή που ακολουθεί την εντολή collapse είναι και αυτή που καθορίζει το βάθος στο οποίο θα εφαρμοστεί η ο παραλληλισμός από την OpenMP [3]. Στην εικόνα που ακολουθεί φαίνεται ένα παράδειγμα της εντολής collapse (Εικόνα 16).

```
14  #pragma omp parallel for private(j) collapse(x)
15  {
16    for(i=0;i<N;i++)
17    {
18      for(j=0;j<N;j++)
19      {
20        C[i][j]= z*x;
21      }
22    }
23  }
```

Εικόνα 16. Η εντολή collapse

Στον κώδικα της εικόνας που προηγείται παρουσιάζονται τρεις εμφωλευμένοι βρόχοι , ένα αντικαταστήσουμε τη μεταβλητή x με την τιμή 2 τότε ισχύουν τα εξής, οι επαναλήψεις i θα ανατεθούν στις υπάρχουσες διεργασίες, στην συνέχεια οι επαναλήψεις που αφορούν τον μετρητή j θα ακολουθήσουν την ίδια λογική και διαμοιραστούν μεταξύ των διεργασιών. Παρόλα αυτά οι επαναλήψεις που αφορούν τον μετρητή k θα εκτελεστούν από όλες τις διεργασίες. Σε αντίθετη περίπτωση εάν η μεταβλητή x ήταν ίση με την τιμή 3 τότε και οι τρεις βρόχοι θα αντιμετωπιζόταν παράλληλα.

4 Συνδυασμός MPI και OpenMP

Όπως φαίνεται από τα κεφάλαια που προηγήθηκαν για την λύση ενός προβλήματος μπορεί να χρησιμοποιηθεί ένα σύνολο υπολογιστών ή ένα σύνολο επεξεργαστών ενός υπολογιστή. Σε αυτό το κεφάλαιο θα περιγραφεί πως με τον συνδυασμό των μεθόδων αυτών μπορούν να γίνουν βελτιώσεις στην επίλυση ενός προβλήματος. Κάτι τέτοιο είναι εφικτό καθώς η MPI λειτουργεί ε κόμβους και κάθε κόμβος από αυτούς απαρτίζεται από ένα σύνολο επεξεργαστών. Για να γίνει πιο κατανοητό πως μπορεί να επιτευχθεί αυτός ο συνδυασμός ακολουθεί ένα τμήμα κώδικα (Εικόνα 17) που εκτυπώνει ένα μήνυμα.

```c
#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char **argv)
{
  int size,rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  #pragma omp parallel
  {
    printf("\nProsecc %d, node no %d out of %d processes and %d nodes\n",rank,omp_get_thread_num(),size,omp_get_num_threads());
  }

  MPI_Finalize();

  return 0;
}
```

Εικόνα 17. Εκτύπωση μηνύματος με συνδυασμό MPI και OpenMP

Όπως φαίνεται υπάρχουν όλα τα χαρακτηριστικά που υπήρχαν στην υλοποίηση της MPI και OpenMP. Έτσι και εδώ ακολουθείται η λογική σειρά πως ένα πρόβλημα διασπάτε σε υποπροβλήματα που διανέμονται στις διεργασίες της MPI. Από εκεί και μετά ακολουθείται η ίδια διαδικασία για την διανομή αυτών στους κόμβους της OpenMP. Αυτό που κάνει ο συγκεκριμένος κώδικας είναι να αξιοποιεί αυτή την δυνατότητα συνδυασμού των δύο αυτών τεχνικών. Η έξοδος αυτού παρουσιάζεται στην εικόνα που ακολουθεί (Εικόνα 18).

```
felix@felix-desktop:~/Desktop/thesis/code/examples/mpi-openMP$ mpicc message.c -o message -fopenmp
felix@felix-desktop:~/Desktop/thesis/code/examples/mpi-openMP$ mpiexec -np 2 ./message

Prosecc 1, node no 7 out of 2 processes and 8 nodes

Prosecc 1, node no 1 out of 2 processes and 8 nodes

Prosecc 0, node no 0 out of 2 processes and 8 nodes

Prosecc 0, node no 3 out of 2 processes and 8 nodes

Prosecc 0, node no 5 out of 2 processes and 8 nodes

Prosecc 0, node no 7 out of 2 processes and 8 nodes

Prosecc 0, node no 1 out of 2 processes and 8 nodes

Prosecc 0, node no 2 out of 2 processes and 8 nodes

Prosecc 0, node no 4 out of 2 processes and 8 nodes

Prosecc 0, node no 6 out of 2 processes and 8 nodes

Prosecc 1, node no 6 out of 2 processes and 8 nodes

Prosecc 1, node no 2 out of 2 processes and 8 nodes

Prosecc 1, node no 4 out of 2 processes and 8 nodes

Prosecc 1, node no 3 out of 2 processes and 8 nodes

Prosecc 1, node no 5 out of 2 processes and 8 nodes

Prosecc 1, node no 0 out of 2 processes and 8 nodes
```

Εικόνα 18. Έξοδος κώδικα εικόνας 15

Με τον συνδυασμό των δύο αυτών τεχνικών μπορεί να επιτευχθεί μεγάλη επιτάχυνση στην επίλυση ενός προβλήματος. Προβλήματα μεγάλης πολυπλοκότητας διασπόνται σε ένα ευρύτερο σύνολο υπολογιστών και από εκεί περνά στους πυρήνες που διατίθενται από τον υπολογιστή αυτό, κάτι τέτοιο επιτρέπει την τέλεση μεγάλου πλήθους υπολογισμός ταυτόχρονα και έτσι υπάρχει σοβαρή εξοικονόμηση χρόνου.

4.1 Επίλυση του προβλήματος της άθροισης διανυσμάτων

Σε αυτή την ενότητα θα φανεί πως μπορεί ένα πρόβλημα να διασπαστεί σε αρκετά υποπροβλήματα και να επιλυθεί με τον συνδυασμό της MPI και της OpenMP. Θα παρουσιαστεί το πρόβλημα της άθροισης διανυσμάτων για το οποίο έχει γίνει λόγος στα προηγούμενα κεφάλαια και έτσι θα υπάρχει κάποιο κοινό σημείο αναφοράς.

Όπως και με την MPI στο Κεφάλαιο 2 βασικό κομμάτι για την υλοποίηση ενός προβλήματος είναι πως σε κάθε κόμβο πρέπει να δεσμευτεί η κατάλληλη μνήμη για τα δεδομένα του

προβλήματος. Η βασική διεργασία συλλέγει τα απαραίτητα δεδομένα από τον χρήστη, στην συνέχεια αυτά αποστέλλονται στις υπόλοιπες διεργασίες. Έως αυτό το σημείο δεν υπάρχει κάποια διαφορά σε σχέση με την υλοποίηση της MPI. Στην εικόνα που ακολουθεί (Εικόνα 19) φαίνονται οι δηλώσεις των μεταβλητών και η δέσμευση μνήμης για τις συμμετέχουσες διεργασίες.

```
11      int i,N;
12
13      int *A,*B,*C;
14      int *Aw,*Bw,*Cw;
15
16      int size,rank;
```

Εικόνα 19. Δήλωση μεταβλητών άθροισης διανυσμάτων

Το σημείο που διαφοροποιείται ο κώδικας της άθροισης είναι το σημείο στο οποίο γίνονται οι υπολογισμοί των δυνάμεων και της νέας θέσεις των σωμάτων. Οι υπολογισμοί αυτού κάθε αυτού δεν έχουν κάποια διαφορά. Ο τρόπος με τον οποίο εμπλέκεται η OpenMP στην υλοποίηση του προβλήματος είναι όπως και στο βασικό κώδικα του Κεφαλαίου 3 με την εντολή #pragma omp parallel for κάθε τμήμα κώδικα αυτομάτως παραλληλίζεται μεταξύ των κόμβων της OpenMP. Στην εικόνα που ακολουθεί (Εικόνα 20) φαίνονται οι υπολογισμοί που αναφέρθηκαν με τον συνδυασμό της MPI και της OpenMP.

```
48      #pragma omp parallel for schedule(dynamic)
49        for(i=0;i<N/size;i++)
50        {
51          Cw[i]=Aw[i]+Bw[i];
52        }
```

Εικόνα 20. Πράξεις άθροισης διανυσμάτων

5 Προγραμματισμός με CUDA

Μια ακόμη μορφή παράλληλου προγραμματισμού είναι ο προγραμματισμός με την χρήση της κάρτας γραφικών. Οι κάρτες γραφικών έχουν στην διάθεση τους ένα μεγάλο πλήθος πυρήνων που μπορούν να αξιοποιηθούν από τον προγραμματιστή με ανάλογο τρόπο όπως και με την OpenMP. Μιλώντας για ένα επεξεργαστή γίνεται αναφορά σε ένα πλήθος πυρήνων της τάξης των τεσσάρων, έξι ή οκτώ, αντίθετα όταν γίνεται αναφορά σε μία κάρτα γραφικών τότε οι πυρήνες που την απαρτίζουν είναι της τάξης των εκατοντάδων. Με μία τόσο μεγάλη υπολογιστική ισχύ προβλήματα τα οποία μπορούν να παραλληλιστούν θα έχουν μεγάλο όφελος.

5.1 Βασικά στοιχεία προγραμματισμού με CUDA

Αρχικά ακολουθούν κάποιοι όροι που θα συμβάλουν στην κατανόηση της υπόλοιπης ενότητας, όταν γίνεται αναφορά του όρου host αυτό αντιστοιχεί στον επεξεργαστή του συστήματος και την μνήμη που αυτός διαθέτει, όταν αναφέρεται ο όρος device με την ίδια λογική αντιστοιχεί στην κάρτα γραφικών και την μνήμη που αυτή διαθέτει. Αφού πλέων η παραλληλία έχει μεταφερθεί στην κάρτα γραφικών είναι φυσιολογικό να γίνεται αναφορά σε αυτή, ο λόγος που εμπλέκεται και ο επεξεργαστής στην αναφερόμενη διαδικασία είναι πως δεν υπάρχει η δυνατότητα να υλοποιηθεί κώδικας απευθείας στην κάρτα γραφικών έτσι χρειάζεται η διαμεσολάβηση αυτού. Ένα παράδειγμα που φαίνεται η μορφή του κώδικα που πρέπει να εκτελεστεί παρουσιάζεται στην εικόνα που ακολουθεί (Εικόνα 21).

```
3  __global__ void myKernel()
4  {
5
6  }
7
8  int main()
9  {
10   myKernel<<<1,1>>>();
11
12   printf("Hellow world");
13   return 0;
14 }
```

Εικόνα 21. Βασικός κώδικας CUDA

Κάτι διαφορετικό σε σχέση με ότι είχε παρουσιαστεί μέχρι τώρα αποτελεί το γεγονός πως εάν κάποιος κώδικας είναι σχεδιασμένος να εκτελεστεί πέραν από τον επεξεργαστή και στην κάρτα γραφικών η κατάληξη του αρχείου στο οποίο συντάσσεται θα πρέπει να αλλάξει. Πιο αναλυτικά ενώ ακόμη ο κώδικας συντάσσεται σε γλώσσα C και ακολουθεί τους ίδιους κανόνες η κατάληξη του αρχείου θα πρέπει να αλλάξει από sample.c σε sample.cu. Κατά τα άλλα όπως συνέβη και στις ενότητες που προηγηθήκαν απαραίτητη προϋπόθεση για την χρήση των δυνατοτήτων της κάρτας γραφικών είναι να συμπεριληφθεί στον κώδικα η ανάλογη βιβλιοθήκη. Στην συγκεκριμένη περίπτωση η βιβλιοθήκη αυτή είναι η cuda.h [10,11].

Η διαδικασία που ακολουθείτε σε όλα τα προβλήματα που εμπλέκουν την κάρτα γραφικών είναι συγκεκριμένη. Πρώτα γίνεται συλλογή των απαραίτητων δεδομένων, έπειτα τα δεδομένα αυτά αντιγράφονται από την μνήμη του επεξεργαστή και αποθηκεύονται στην μνήμη της κάρτας γραφικών, στην συνέχεια το τμήμα του κώδικα που αναφέρεται στην κάρτα γραφικών φορτώνεται και εκτελείται. Τέλος μετά τον υπολογισμό των κατάλληλων δεδομένων για το πρόβλημα που αντιμετωπίζεται τα αποτελέσματα αντιγράφονται από την κάρτα γραφικών πίσω στον επεξεργαστή.

Ένα βασικό στοιχείο που πρέπει να ληφθεί υπόψη είναι πως για να είναι δυνατή η αντιγραφή των δεδομένων μεταξύ των host και device θα πρέπει να γίνει και η ανάλογη δέσμευση μνήμης στην κάρτα γραφικών [10,11]. Στον κώδικα που θα εκτελέσει ο host δημιουργούνται οι μεταβλητές οι οποίες αφορούν και τους δύο συμμετέχοντες, όμως για τις μεταβλητές οι οποίες αναφέρονται στο device είναι απαραίτητο να γίνει και η ανάλογη δέσμευση μνήμης και στο τέλος χρήσης αυτών και η ανάλογη απελευθέρωση αυτής. Για να μπορεί να γίνει πιο εύκολα ο διαχωρισμός μεταξύ των μεταβλητών που αναφέρονται στο host και στο device ακολουθείται η εξής συνθήκη, κάθε μεταβλητή που αντιστοιχεί στο host θα συμβολίζεται με την πρόθεση h_x ενώ αυτές που αναφέρονται στο device θα συμβολίζονται με d_x.

Για την δέσμευση μνήμης χρησιμοποιείται μια παραλλαγή της εντολής malloc της C που καλείται cudaMalloc(), όπως φαίνεται και από το όνομα της είναι σχεδιασμένη ακριβώς για την περίπτωση που γίνεται χρήση CUDA [10,11]. Με το ίδιο σκεπτικό για την αποδέσμευση της μνήμης αυτής υπάρχει η εντολή cudaFree() [10,11]. Για την επίτευξη της επίλυσης ενός

προβλήματος όπως προαναφέρθηκε απαραίτητη είναι η αντιγραφή δεδομένων και αποτέλεσμα αυτού είναι η ανάγκη μίας ακόμη συνάρτησης υπεύθυνη για αυτή την εργασία. Η συνάρτηση αυτή καλείται cudaMemcpy(). Στην εικόνα που ακολουθεί (Εικόνα 22) φαίνεται ο τρόπος χρήσης των όσων περιγράφθηκαν έως τώρα.

```
22    int *h_A,*h_B,*h_C;
23    int *d_A,*d_B,*d_C;
24
25    printf("\nGve the size of the tables (>10.000): ");
26    scanf("%d",&N);
27
28    h_A=(int*)malloc(N*sizeof(int));
29    h_B=(int*)malloc(N*sizeof(int));
30    h_C=(int*)malloc(N*sizeof(int));
31
32    for(i=0;i<N;i++)
33    {
34      h_A[i]=rand()%(high-low+1)+low;
35      h_B[i]=rand()%(high-low+1)+low;
36    }
37
38    cudaMalloc((void**)&d_A, N*sizeof(int));
39    cudaMalloc((void**)&d_B, N*sizeof(int));
40    cudaMalloc((void**)&d_C, N*sizeof(int));
41
42    cudaMemcpy(d_A,h_A,N*sizeof(int),cudaMemcpyHostToDevice);
43    cudaMemcpy(d_B,h_B,N*sizeof(int),cudaMemcpyHostToDevice);
44
45    int threadsPerBlock=256;
46    int blocksPerGrid=(N+threadsPerBlock-1)/threadsPerBlock;
47
48
49    vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A,d_B,d_C,N);
50
51    cudaMemcpy(h_C,d_C,N*sizeof(int),cudaMemcpyDeviceToHost);
52
53    for(i=0;i<N;i++)
54    {
55      printf("\n%d + %d= %d",h_A[i],h_B[i],h_C[i]);
56    }
57
58    free(h_A);
59    free(h_B);
60    free(h_C);
61
62    cudaFree(d_A);
63    cudaFree(d_B);
64    cudaFree(d_C);
```

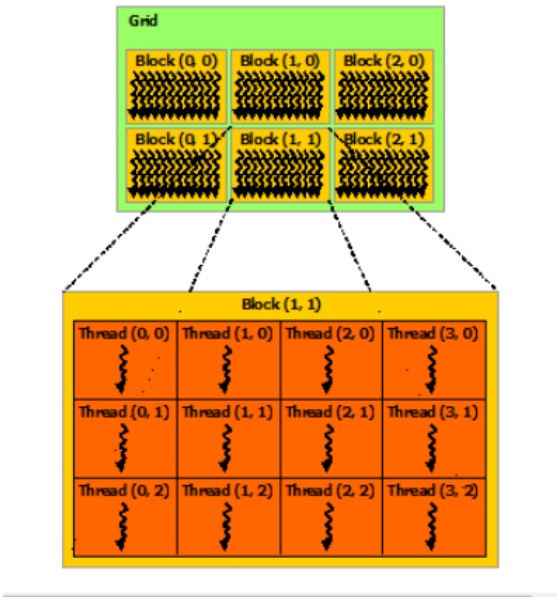
Εικόνα 22. Βασικά στοιχεία CUDA

Σε αντίθεση με ότι έχει αναφερθεί έως τώρα είναι πως συναρτήσεις οι οποίες είναι σχεδιασμένες να εκτελεστούν στην κάρτα γραφικών χαρακτηρίζονται διαφορετικά κατά την δήλωση τους, με την χρήση της λέξης κλειδί __global__ πριν από το όνομα τους διαχωρίζονται από τις συναρτήσεις που έχουν σχεδιαστεί για να εκτελεστούν στον επεξεργαστή [10,11]. Παρόλο που τέτοιου είδους συναρτήσεις αφορούν την κάρτα γραφικών είναι αναγκαστικό να κληθούν από τον επεξεργαστή. Όταν μία συνάρτηση που αναφέρεται στην κάρτα γραφικών καλείται από τον επεξεργαστή τότε συμβολίζεται με την χρήση τριών εισαγωγικών όπως φαίνεται και εδώ myKernel<<<x,y>>>().

Οι μεταβλητές x και y αντιπροσωπεύουν τα μεγέθη που θα καθορίσουν τον τρόπο με τον οποίο θα παραλληλιστεί ο κώδικας που θα υπάρχει μέσα στην συνάρτηση myKernel(). Η μεταβλητή x αναπαριστά τα block που υπάρχουν μέσα σε ένα grid τα οποία θα χρησιμοποιηθούν και η μεταβλητή y τα threads που διαθέτει κάθε ένα block από αυτά.

5.2 Παραλληλισμός με την χρήση CUDA

Βασικό κομμάτι της παραλληλίας όταν εμπλέκεται η κάρτα γραφικών αποτελούν οι δύο μεταβλητές που αναφέρθηκαν πιο πάνω. Για να γίνουν κατανοητές οι ερμηνείες των δύο αυτών θα βοηθήσει η εικόνα που ακολουθεί (Εικόνα 23).



Εικόνα 23. Λογική διάσπασης κάρτας γραφικών [10]

Όπως φαίνεται από την εικόνα (Εικόνα 20) για την υλοποίηση ενός παράλληλου κώδικα μέσω CUDA γίνεται χρήση ενός grid. Ένα grid αποτελείται από ένα σύνολο από blocks που με την σειρά τους αυτά αποτελούνται από έναν αριθμό από threads. Αυτό που ορίζουν οι μεταβλητές εντός τον εισαγωγικών κατά την κλήση μίας συνάρτησης είναι τον αριθμό από blocks και threads που θα εκτελέσουν των ανάλογο κώδικα αντίστοιχα. Λόγο αυτού του σχεδιασμού υπάρχει η δυνατότητα διαφόρων συνδυασμός ως προς το πώς θα εκτελεστεί μια συνάρτηση.

5.2.1 Παραλληλισμός με την χρήση blocks

Με την πρώτη τιμή εντός των εισαγωγικών καθορίζεται ο αριθμός των block που θα εκτελέσουν των κώδικα. Θέτοντας για παράδειγμα την πρώτη τιμή ίση με N και κρατώντας την δεύτερη τιμή ίση με ένα τότε αυτό που συμβαίνει είναι πως ο κώδικας εκτελείται ταυτόχρονα από N blocks και ένα μόνο thread ανά block [11]. Ο κώδικας που αντιστοιχεί στην λειτουργία αυτή φαίνεται στην εικόνα που ακολουθεί (Εικόνα 24).

```
__global__ voidadd(int *A, int *B, int *C,int N)
{
  if(blockIdx.x<N)
  {
    C[blockIdx.x] = A[blockIdx.x] + B[blockIdx.x];
  }
}

int main()
{
  //...

  vectorAdd<<<N,1>>>(d_A,d_B,d_C,N);

  //...
}
```

Εικόνα 24. Παράλληλη εκτέλεση με blocks

Όπως φαίνεται στον κώδικα που αφορά την κάρτα γραφικών υπάρχει μια νέα μεταβλητή blockIdx.x. Η μεταβλητή αυτή αντιστοιχεί στην τιμή που έχει κάθε block μέσα σε ένα grid ώστε να διαχωρίζει τον εαυτό του από τα υπόλοιπα. Καθώς κάθε block τρέχει τον κώδικα ταυτόχρονα με την χρήση του αντιστοιχίζεται σε κάθε ένα από αυτά ένα συγκεκριμένο κελί του τελικού πίνακα προς υπολογισμό.

5.2.2 Παραλληλισμός μέσω threads

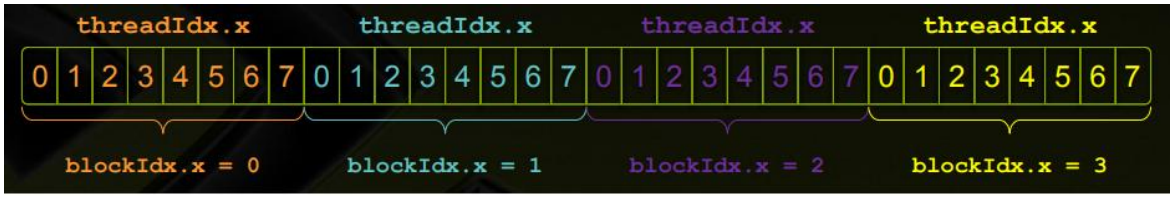
Αντίστροφα υπάρχει η δυνατότητα να αντιστραφεί η συνάρτηση αυτή και να μεταφερθεί η παραλληλία από τα block μίας κάρτας γραφικών στα threads. Με την αλλαγή της δεύτερης μεταβλητής που αντιστοιχεί στα threads που περιέχονται μέσα σε ένα block ίση με N και την τιμή των block ίση με 1 θα υπάρχει μια διαφορετική αντιμετώπιση. Πιο συγκριμένα με αυτό τον τρόπο ενεργοποιείται μόνο ένα block που περιέχει N threads εντός του [11]. Όπως και με τα blocks έτσι κι εδώ κάθε thread έχει μία δική του μοναδική τιμή που το διαχωρίζει από τα υπόλοιπα threads του block και αντιστοιχεί στο threadIdx.x. Ο ανάλογος κώδικας σε σχέση με τον κώδικα των blocks φαίνεται στην εικόνα που ακολουθεί (Εικόνα 25).

```
4   __global__ voidadd(int *A, int *B, int *C,int N)
5   {
6     if(threadIdx.x<N)
7     {
8       C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
9     }
10  }
11
12  int main()
13  {
14    //...
15
16    vectorAdd<<<1,N>>>(d_A,d_B,d_C,N);
17
18    //...
19  }
```

Εικόνα 25. Παραλληλισμός με χρήση threads

5.2.3 Παραλληλισμός με την χρήση blocks και threads

Όπως φάνηκε στις ενότητες που προηγήθηκαν ένα πρόβλημα μπορεί να παραλληλιστεί με την χρήση ενός πλήθους block ή thread. Στην συγκεκριμένη ενότητα θα παρουσιαστεί πως μπορούν να συνδυαστούν αυτές οι δύο μέθοδοι. Για να γίνουν πιο κατανοητές οι μέθοδοι αυτοί πρέπει πρώτα να αναλυθεί με ποιόν τρόπο τα δεδομένα ενός προβλήματος κατανέμονται στην μνήμη που διαθέτει η κάρτας γραφικών. Πλέον δεν γίνεται αναφορά στα δεδομένα τόσο απλά όπως και πριν με την χρήση των blockIdx.x και threadIdx.x αλλά με τον συνδυασμό αυτών τον δύο [11]. Η εικόνα που ακολουθεί (Εικόνα 26) αποσκοπεί στο να γίνουν όλα αυτά πιο κατανοητά.



Εικόνα 26. Συνδυασμός blocks και threads [11]

Αναφερόμενοι πάλι στο πρόβλημα της άθροισης διανυσμάτων, το μήκος των πινάκων είναι ίσο με 32 και για την λύση του προβλήματος χρησιμοποιούνται τέσσερα block που τα κάθε ένα περιέχει οκτώ thread. Αυτό που επιδιώκεται σε αυτό το σημείο είναι πως οι υπολογισμοί θα εκτελεστούν σε όλα τα blocks ταυτόχρονα μέσω τον thread που συμπεριλαμβάνονται μέσα σε αυτά. Όπως πριν χρησιμοποιούταν οι τιμές των blockIdx.x και threadIdx.x για να

μπορέσουν όλοι οι υπολογισμοί να εκτελεστούν παράλληλα εδώ θα πρέπει να γίνουν κάποιες μετατροπές. Στο παράδειγμα που ακολουθεί φαίνεται πως ακριβώς θα μπορέσει να επιτευχθεί κάτι τέτοιο.

int index = threadIdx.x + bloxkIdx.x * M [11]

Η μεταβλητή M αντιστοιχεί στο πλήθος των thread που περιλαμβάνονται μέσα σε ένα block. Έτσι κάθε κελί του πίνακα C της άθροισης αντιστοιχεί στην τιμή index. Για παράδειγμα αν ζητείται να υπολογιστεί το εικοστό πρώτο κελί του πίνακα C τότε σύμφωνα με τον κώδικα που υπάρχει πιο πάνω θα υπολογιστεί από το έκτο thread του τρίτου block. Αυτό προκύπτει από τους υπολογισμούς στον πίνακα που ακολουθεί (Πίνακας 9).

block * threads	**Εύρος block**
0 * 8	0 – 7
1 * 8	8 – 16
2 * 8	17 – 23
3 * 8	24 – 35

Πίνακας 9. Εύρος block βάση των threads που περιέχονται σε αυτό

Αυτό που περιγράφει ο πιο πάνω πίνακας είναι το πλήθος των κελιών που είναι υπεύθυνο να υπολογίσει το κάθε block. Με αυτό το σκεπτικό και σύμφωνα με τον πίνακα το στοιχείο 21 του πίνακα C βρίσκεται υπό την ευθύνη του 3^{ου} block και πιο συγκεκριμένα από το 6^{ο} thread αυτού το block καθώς

index = threadIdx.x + bloxkIdx.x * M

5 + 2 * 8

Σε σύγκριση με την επίλυση του προβλήματος με χρήση πολλών block και ενός thread ή ενός block με πλήθος thread στην εικόνα που ακολουθεί (Εικόνα 27) φαίνεται ο κώδικας που θα πρέπει να υλοποιηθεί συνδυάζοντας αυτά τα δύο.

```
__global__ voidadd(int *A, int *B, int *C,int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;

  if(index<N)
  {
    C[index]=A[index]+B[index];
  }
}

int main()
{
  //...

  vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A,d_B,d_C,N);

  //...
}
```

Εικόνα 27. Παραλληλισμός με συνδυασμό των blocks και threads

6 Περιγραφή προβλήματος των Ν-σωμάτων

Το πρόβλημα των Ν-σωμάτων είναι ένα πρόβλημα αστροφυσικής το οποίο αναφέρεται στις βαρυτικές δυνάμεις που ασκούνται μεταξύ των πλανητών, καθώς και το αποτέλεσμα αυτών που είναι η μετακίνηση των πλανητών του συστήματος. Ο υπολογισμός των δυνάμεων που ασκούνται μέσα στο σύστημα που μελετάτε ακολουθεί την αρχή του Νεύτωνα για δύναμη που ασκείται μεταξύ των ουράνιων σωμάτων [7]. Για τον σκοπό της μελέτης και της προσομοίωσης τέτοιων δυνάμεων οι πλανήτες θα αναπαρασταθούν ως σημειακές μάζες. Για την σωστή υλοποίηση του προβλήματος και έτσι ώστε να προβλεφθεί η νέα θέση καθενός από τα σώματα που θα συμμετάσχουν στο πείραμα απαιτείται είναι η γνώση κάποιο βασικών στοιχείων. Τα δεδομένα που είναι απαραίτητα για την υλοποίηση του προβλήματος είναι η μάζα κάθε σημειακή μάζας, η θέση αυτού καθώς και η ταχύτητα του σε κάποια χρονική στιγμή t. Με αυτά μπορεί να γίνει ο υπολογισμός της δύναμης που ασκείται σε ένα σώμα από τα υπόλοιπα που απαρτίζουν το σύστημα που μελετάται, κατ επέκταση μπορεί να υπολογιστεί η νέα επιτάχυνση, ταχύτητα και θέση του σώματος αυτού.

Αρχικά θα μελετηθεί πως αντιμετωπίζεται το πρόβλημα των Ν-σωμάτων χωρίς συγκρούσεις όταν το σύστημα που μελετάται αποτελείται από δύο μόνο σώματα. Κάτι τέτοιο καθιστά πιο εύκολη την μελέτη και την κατανόηση των βασικών τμημάτων του προβλήματος όπως των υπολογισμό των δυνάμενων του συστήματος. Έπειτα θα γίνει αναγωγή του προβλήματος αυτού σε ένα σύστημα που αποτελείται από ένα μεγαλύτερο πλήθος σωμάτων. Μέσω αυτής της οπτικής θα μελετηθούν οι μετατροπές που υφίστανται οι αρχικές εξισώσεις για τον υπολογισμό των δυνάμεων και θέσεων των εν λόγο σωμάτων. Για την υλοποίηση του προβλήματος αυτού θα γίνει χρήση της άμεσης μεθόδου η οποία είναι η πιο απλή μέθοδος για τον υπολογισμό των νέων θέσεων των σωμάτων, ταυτόχρονα είναι και η πιο ζημιογόνα καθώς υλοποιεί πλήθος περίπλοκων υπολογισμός και δίνει αποτελέσματα μεγάλης ακρίβειας.

6.1 Η δύναμη F μεταξύ των σωμάτων

Βασικό μέλημα για τον υπολογισμό των νέων θέσεων των σωμάτων είναι η δύναμη η οποία ασκείται σε αυτά από τα υπόλοιπα σώματα. Για των υπολογισμό την δύναμης αυτής ακολουθείται η αρχή του Νεύτωνα μεταξύ των ουράνιων σωμάτων. Έστω πως υπάρχουν δύο

σώματα στο σύστημα το οποίο μελετάται. Η δύναμη F η οποία ασκείται μεταξύ των δύο αυτών σωμάτων αντιστοιχεί στον παρακάτω τύπο.

$$F = G\frac{m_i m_j}{|\overrightarrow{r_{ij}}|^2} \text{ [7,13]}$$

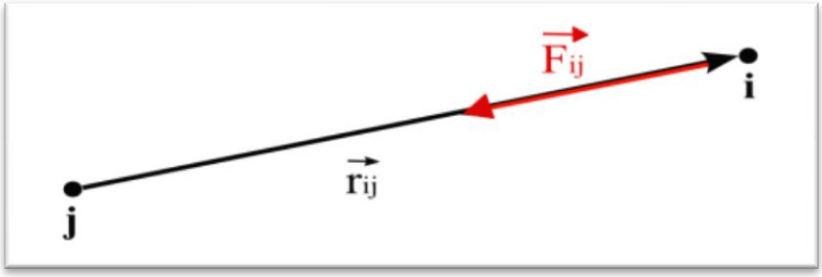
Η μεταβλητή m αντιστοιχεί στις μάζες των σημειακών σωμάτων και οι δείκτες i και j αντιπροσωπεύουν το σώμα στο οποίο αντιστοιχεί αυτή. Το διάνυσμα r αντιπροσωπεύει την απόσταση μεταξύ των σωμάτων i και j. Για τον υπολογισμό της τιμής του r απαραίτητες είναι η συντεταγμένες στους άξονες x, y και z για την ακριβή θέση δύο σωμάτων. Για τον υπολογισμό του διανύσματος r_{ij} χρησιμοποιείται ο ακόλουθος τύπος.

$$|r_{ij}| = |r_i - r_i| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} \text{ [6,7]}$$

Η μεταβλητή G αντιπροσωπεύει την βαρυτική σταθερά η οποία χρησιμοποιείται για τον υπολογισμό της δύναμης F. Η σταθερά G είναι ίση με $6{,}673 \cdot 10^{-11} m^3 kg^{-1} s^{-2}$. Η δύναμη F όμως πέραν από μέτρο έχει και διεύθυνση, αποτέλεσμα αυτού είναι ο πιο πάνω τύπος της δύναμης F να μετασχηματιστεί ως εξής ια τον σωστό υπολογισμό της.

$$\vec{F} = -G\frac{m_i m_j \overrightarrow{r_{ij}}}{|\overrightarrow{r_{ij}}|^3} \text{ [7,13]}$$

Οι τύποι οι οποίοι αναφέρθηκαν μέχρι τώρα έχουν εφαρμογή πάνω σε ένα σύστημα αποτελούμενα από δύο και μόνο σώματα όπως φαίνονται και στην εικόνα που ακολουθεί (Εικόνα 28).

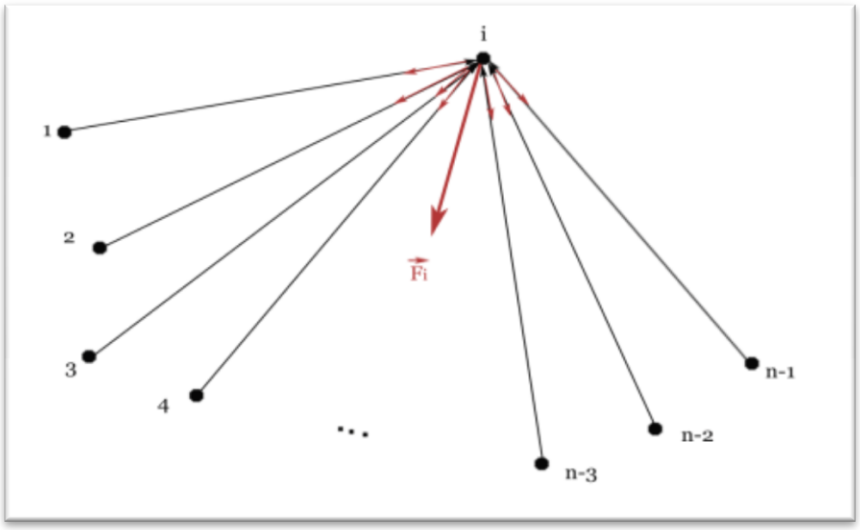


Εικόνα 28. Το πρόβλημα των δύο σωμάτων [7]

Για την επίλυση προβλημάτων που εμπλέκουν περισσότερα από δύο σώματα τότε απαραίτητες είναι κάποιες μετατροπές στον τύπου που προηγήθηκε. Η δύναμη F πλέων δεν υπολογίζεται μεταξύ δύο σωμάτων, αποτέλεσμα αυτού είναι πως η δύναμη που ασκείται σε κάθε σώμα αποτελεί το άθροισμα της δύναμης μεταξύ του σώματος για το οποίο γίνεται ο υπολογισμός και κάθε άλλου που συνυπάρχει εντός του συστήματος διαδοχικά. Ο τύπος που αντιστοιχεί στον υπολογισμό της δύναμης F για περισσότερα από δύο σώματα φαίνεται παρακάτω.

$$\vec{F_i} = -Gm_i \sum_{j=1}^{N} \frac{m_j(\vec{r_j} - \vec{r_i})}{|\vec{m_j} - \vec{m_i}|^3} \quad [7,13]$$

Για να μπορέσει να γίνει πιο κατανοητό το πρόβλημα στο οποίο αντιστοιχεί η πιο πάνω εξίσωση ένα πρόβλημα Ν-δωμάτων θα έχει την μορφή της εικόνας που ακολουθεί (Εικόνα 29).



Εικόνα 29. Το πρόβλημα των N-σωμάτων [7]

Στις πιο πάνω εξισώσεις εμφανίζεται το αρνητικό πρόσημο (-) πριν από τον υπολογισμό των δυνάμεων κάτι το οποίο συμβολίζει την φορά του διανύσματος της δύναμης F που είναι αντίθετη με αυτή του διανύσματος r. Κάτι τέτοιο ερμηνεύεται ως ότι η δύναμη αυτή αποτελεί μία ελκτική δύναμη μεταξύ των δύο σωμάτων i και j. Όπως φαίνεται και στις εικόνες που προηγήθηκαν (Εικόνα 28, Εικόνα 29).

Καθώς το σύστημα που μελετάται δεν περιέχει συγκρούσεις θα πρέπει να ληφθεί υπόψη πως δεν επιτρέπετε τα σώματα να πλησιάζουν αρκετά το ένα με το άλλο. Το πρόβλημα που δημιουργείται σε τέτοιες καταστάσεις είναι πως δύο σώματα που η απόσταση σου τους είναι υπερβολικά μικρή δηλαδή τείνει προς το μηδέν έχει ως αποτέλεσμα οι δυνάμεις που ασκούνται μεταξύ αυτών να είναι τεράστιες. Αποτέλεσμα αυτού οι επιταχύνσεις και οι ταχύτητες που θα υπολογιστούν με αυτά τα δεδομένα να οδηγήσουν σε μεγάλα σφάλματα. Για την αποφυγή τέτοιων καταστάσεων εφαρμόζεται ξανά μια τροποποίηση στην εξίσωση της δύναμης F με σκοπό την απαλοιφή τέτοιων καταστάσεων. Πιο συγκεκριμένα προσθέτουμε μια μεταβλητή ε η οποία ονομάζεται μεταβλητή απόσβεσης και αντιστοιχεί σε ένα πραγματικό αριθμό. Η εξίσωση της δύναμης F σύμφωνα με την μεταβλητή απόσβεσης φαίνεται στον τύπο που ακολουθεί.

$$\vec{F} = -Gm_i \frac{\sum_{j=1}^{N} m_j(\vec{r_j} - \vec{r_i})}{\left(\left|\vec{r_j} - \vec{r_i}\right|^2 + \varepsilon^2\right)^{3/2}} \quad [7]$$

Έτσι με την χρήση της μεταβλητής ε στον τύπο της δύναμης που ασκείται μεταξύ των σωμάτων διασφαλίζεται η αποφυγή σφαλμάτων σε σώματα τα οποία έχουν μικρές αποστάσεις.

6.2 Υπολογισμός νέας θέσης σώματος

Σκοπός του συγκεκριμένου προβλήματος είναι μετά το πέρας ενός συγκεκριμένου χρονικού διαστήματος να μπορούν να υπολογιστούν οι νέες θέσεις των σωμάτων που απαρτίζουν το σύστημα που μελετάται. Απαραίτητο κομμάτι για τον υπολογισμό των θέσεων αυτών αποτελεί ο υπολογισμός την επιτάχυνσης και των ταχυτήτων που προκύπτουν από τον υπολογισμό της δύναμης F της προηγούμενης ενότητας. Για τον υπολογισμό των μεγεθών που αναφέρθηκαν χρησιμοποιούνται οι εξισώσεις που ακολουθούν.

$$\overrightarrow{F_n} = m_n\overrightarrow{a_n} \Leftrightarrow \overrightarrow{a_n} = \overrightarrow{F_n}m_n$$
$$\overrightarrow{v_{n+1}} = \overrightarrow{v_n} + \overrightarrow{a_n}\Delta t$$
$$\overrightarrow{r_{n+1}} = \overrightarrow{r_n} + \overrightarrow{v_n}\Delta t$$
[6,7]

Όπως φαίνεται και από τις εξισώσεις που προηγήθηκαν αφού πλέων υπολογιστούν οι δυνάμεις F, τότε μέσω αυτών γίνεται υπολογισμός των επιταχύνσεων των σωμάτων σύμφωνα με την αθροιστική δύναμη που ασκείται στο κάθε ένα. Έπειτα κάνοντας χρήση της επιτάχυνσης αυτής και της είδη υπάρχουσας ταχύτητας ενός σώματος υπολογίζεται η νέα ταχύτητα αυτού. Τέλος αφού έχουν οι υπολογιστεί όλα τα παραπάνω μεγέθη τότε είναι εφικτός ο υπολογισμός την νέας θέσεις ενός σώματος. Με Δt συμβολίζεται η διαφορά του χρόνου μεταξύ τον υπολογισμών των δυνάμεων F που ασκούνται σε κάθε σώμα και των νέων θέσεων που τους αντιστοιχούν. Ο χρόνος αυτός για κάθε κύκλο επαναλήψεων παραμένει ο ίδιος κάτι που σημαίνει πως $\Delta t = t_{τελικό} - t_{αρχικό} = t_1 - t_0 = t_2 - t_1 = t_{n+1} - t_n$.

6.3 Υπολογισμός αρχικών τιμών προβλήματος

Βασική προϋπόθεση για των σωστό υπολογισμό των θέσεων των σωμάτων είναι τα αρχικά δεδομένα του προβλήματος. Αρχικά για τον υπολογισμό των αρχικών θέσεων των σωμάτων παράγεται ένας αριθμός x στο διάστημα [0,1] που ακολουθεί την κανονική κατανομή και είναι μοναδική για το κάθε ένα από αυτά. Για την διασφάλιση πως η τιμή x ακολουθεί την κανονική κατανομή ακολουθούμε την μέθοδο της αναλογίας. Πιο συγκεκριμένα παράγονται δύο τυχαίοι αριθμοί u και v. Μέσω αυτών ορίζεται η τιμή της μεταβλητής x σύμφωνα με τον τύπο που ακολουθεί.

$$x = \sqrt{\frac{8}{e}}(v - 0.5)/u \quad [1,2]$$

Για να κριθεί κατάλληλη η μεταβλητή x θα πρέπει να ικανοποιεί κάποιες συγκεκριμένες συνθήκες ώστε να συνεχίζουμε. Αν ικανοποιείται η ακόλουθη συνθήκη τότε η μεταβλητή x γίνεται δεκτή και έπειτα η διαδικασία ξεκινά από την αρχή για του επόμενου x.

$$x^2 \leq 5 - 4e^{1/4}u \quad [1,2]$$

Αντίθετα αν ικανοποιείται η συνθήκη που ακολουθεί τότε η τιμή x απορρίπτεται και όλη η διαδικασία ξεκινά από την αρχή με τον ορισμό νέων u και v.

$$x^2 \geq \frac{4e^{-1.35}}{u} \quad [1,2]$$

Οι δύο συνθήκες αυτές είναι προαιρετικό να ελεγχθούν για να κριθεί η καταλληλότητα της μεταβλητής x. Τι θέση τους μπορεί να πάρει μία τρίτη συνθήκη που μπορεί να καθορίσει το αν η τιμή x θα γίνει δεκτή.

$$x^2 \leq -4\ln(u) \quad [1,2]$$

Σε περίπτωση που η συνθήκη αυτή και μόνο αυτή ικανοποιείται τότε το x αποτελεί τμήμα της κανονικής κατανομής. Μετά τον υπολογισμό των απαραίτητων στοιχείων που εξαρτώνται από το x η διαδικασία ξεκινά από την αρχή.

Στην συνέχεια με την χρήση της μεταβλητής x υπολογίζεται η τιμή r που είναι απαραίτητη για τις ακριβείς θέσεις των σωμάτων. Η τιμή r αντιπροσωπεύει την απόσταση του εκάστοτε σώματος από το κέντρο του συστήματος, για τον υπολογισμό της ακολουθείται ο τύπος που ακολουθεί.

$$r = \left(X_1^{2/3} - 1\right)^{-1/2} \quad [7]$$

Πέραν από την απόσταση του σώματος με το κέντρο του συστήματος υπάρχουν άλλες δύο μεταβλητές που θα βοηθήσουν στον καθορισμό της θέσεις ενός σώματος. Οι μεταβλητές αυτές είναι η θ και φ και ακολουθείται η ίδια διαδικασία παραγωγής όπως και με την x. Η μόνη διαφορά είναι πως η θ παίρνει τιμές μεταξύ [0,π] και η φ μεταξύ [0,2π]. Με την χρήση των παραπάνω μεταβλητών πλέων μπορεί να γίνει ο υπολογισμός των καρτεσιανών συντεταγμένων των σωμάτων. Για τον υπολογισμό των συντεταγμένων στους τρεις άξονες χρησιμοποιούνται οι εξής τρεις τύποι σε σχέση με την μεταβλητή r.

$$x = r \sin\theta \cos\varphi$$
$$y = r \sin\theta \sin\varphi$$
$$z = r \cos\theta$$
[6,7]

Πέραν από την αρχική θέση των σωμάτων πολύ σημαντική αποτελεί και η αρχική ταχύτητα που έχει κάθε ένα από αυτά. Κάθε σώμα το οποίο συμμετάσχει στο σύστημα μπορεί να έχει μέγιστη τιμή ταχύτητας την V_{esc} η οποία καλείται ταχύτητα αποφυγής. Σώματα τα οποία φτάνουν σε μεγαλύτερες ταχύτητες της ταχύτητας αποφυγής διαφεύγουν του συστήματος. Για τον υπολογισμό της ταχύτητας αποφυγής χρησιμοποιείται ο ακόλουθος τύπος.

$$V_{esc} = \sqrt{2}\left((1 + r^2)^{-1/4}\right) \quad [7]$$

Όπως και με τον προσδιορισμό της θέση των σωμάτων έτσι και εδώ γίνεται χρήση των θ και φ σε συνδυασμό με την ταχύτητα αποφυγής που υπολογίστηκε πιο πάνω για τον υπολογισμό της αρχικής ταχύτητας των σωμάτων. Οι τύποι που ακολουθούνται για τον υπολογισμό των ταχυτήτων φαίνονται στη συνέχεια.

$$V_x = V_{esc} \sin\theta \cos\varphi$$
$$V_y = V_{esc} \sin\theta \sin\varphi$$
$$V_z = V_{esc} \cos\theta$$
[6,7]

Η διαδικασία αυτή επαναλαμβάνεται τόσες φορές όσες και τα σώματα που αποτελούν το σύστημα το οποίο μελετάται έτσι ώστε να υπολογιστούν όλες οι απαραίτητες αρχικές θέσεις και ταχύτητες των αυτών.

7 Προγραμματιστική επίλυση του προβλήματος των Ν-σωμάτων

Στο προηγούμενο κεφάλαιο έγινε ανάλυση όλων τον απαραίτητων στοιχείων για την επίλυση του προβλήματος των Ν-σωμάτων. Σε συνέχεια αυτής θα γίνει εφαρμογή των εξισώσεων αυτών για την προγραμματιστική επίλυση του προβλήματος. Αρχικά θα γίνει η σειριακή υλοποίηση αυτού για διάφορα πλήθη σωμάτων για συνόλου χρονικών βημάτων. Αφού γίνει κατανοητό πως ακριβώς λειτουργεί ο κώδικας που θα υλοποιηθεί τότε με γνώμονα αυτών θα γίνει αναγωγή του προβλήματος στην κατανεμημένη επίλυση αυτού με την χρήση της MPI. Προχωρώντας και χρησιμοποιώντας τα ίδια δεδομένα θα παρουσιαστεί η παράλληλη επίλυση του προβλήματος με την χρήση της OpenMP και με τον συνδυασμό των MPI και OpenMP. Τέλος θα γίνει χρήση της κάρτας γραφικών και θα υλοποιηθεί το εν λόγο πρόβλημα με χρήση CUDA. Οι εκτέλεση των προσομοιώσεων θα γίνει σε ένα υπολογιστή που διαθέτει ένα τετραπύρηνο επεξεργαστή Ryzen 5 2400G στα 3.6GHz και 8GB ram και κάρτα γραφικών NVIDIA GTX 1660Ti. Καθώς δεν υπάρχει δυνατότητα εκτέλεσης του κώδικα σε μια συστοιχία μηχανημάτων για την συνδυαστική μέθοδο MPI-OpenMP θα παρουσιαστεί μόνο ο κώδικας για την επίλυση του προβλήματος.

Σκοπός του κεφαλαίου είναι να φανούν οι διαφορές που υπάρχουν ανάλογα με την επιλογή τρόπου υλοποίησης του προβλήματος και κατά πόσο και αν υπάρχει κάποιο κέρδος με την χρήση αυτών. Κάτι τέτοιο θα γίνει εμφανή καθώς κάθε τρόπος επίλυσης θα χρονομετρηθεί και θα αξιολογηθεί σύμφωνα με τον χρόνο που απαιτήθηκε για την διεξαγωγή των υπολογισμών και όχι μόνο. Έπειτα θα γίνει σύγκριση των χρόνων αυτών σε σχέση με τις υπόλοιπες μεθόδους επίλυσης με σκοπό την σύγκριση των αποτελεσμάτων.

7.1 Σειριακή επίλυση

Αρχικά με την εντολή #define προσδιορίζουμε τις μεταβλητές που καθ’ όλοι την διάρκεια του προβλήματος δεν θα χρειαστεί να υποστούν κάποια μεταβολή. Όπως αναφέρθηκαν και στις προηγούμενες ενότητες δηλώνεται η μεταβλητή G που αντιπροσωπεύει την βαρυτική σταθερά, η μεταβλητή M που αντιπροσωπεύει την μάζα κάθε σώματος, η μεταβλητή Dt που αναπαριστά το χρονικό βήμα μεταξύ των υπολογισμών των θέσεων των σωμάτων και την και τέλος η μεταβλητή Exp η οποία αντιστοιχεί στην μεταβλητή απόσβεσης για την εξάλειψη

σφαλμάτων. Οι μεταβλητές και τιμές αυτών φαίνονται στην εικόνα που ακολουθεί (Εικόνα 30).

```
5
6   #define G 6.673e-11         // Βαρυτική σταθερά
7   #define M 100               // Μάζα σωμάτων
8   #define Dt 0.03             // Χρονικό διάστημα μεταξύ επαναλλήψεων
9   #define Exp 0.01            // Μεταβλητή απόσβεσης
10
```

Εικόνα 30. Σταθερές κώδικα των Ν-σωμάτων

Έπειτα ακολουθούν οι δηλώσεις των μεταβλητών που θα χρειαστούν για την υλοποίηση των υπολογισμών. Γίνεται οι συλλογή των δεδομένων από τον χρήστη που αντιστοιχούν στο πλήθος επαναλήψεων που θα εκτελεστούν οι υπολογισμοί, το πλήθος των σωμάτων τα οποία υπάρχουν εντός του συστήματος που μελετάται. Αφού πλέων έχουν συλλεχθεί όλα τα απαιτούμενα δεδομένα μπορεί να γίνει ο υπολογισμός των αρχικών θέσεων και ταχυτήτων των σωμάτων του συστήματος.

Για την διεξαγωγή της προσομοίωσης χρειάζεται η όλη διαδικασία να συμπεριληφθεί μέσα σε τρεις εμφωλευμένους βρόχους for. Ο πρώτος βρόχος for με δείκτη i αντιστοιχεί στις επαναλήψεις που έχουν οριστεί από τον χρήστη, κάθε επανάληψη αντιπροσωπεύει ένα χρονικό βήμα μεγέθους Dt μεταξύ των υπολογισμών των δυνάμεων και θέσεων των σωμάτων. Ο δεύτερος βρόχος με δείκτη j αντιστοιχεί στο πλήθος των σωμάτων του συστήματος στα οποία αναφέρονται οι υπολογισμοί που θα υλοποιηθούν. Ο τρίτος σε σειρά εσωτερικός βρόχος με δείκτη k αντιστοιχεί στο σώμα στο οποίο ασκεί δύναμη στο σώμα του βρόχου j. Όλοι οι υπολογισμοί που θα διεξαχθούν ελέγχονται με μία συνθήκη if για την διασφάλιση πως δεν θα γίνουν υπολογισμοί μεταξύ των ίδιων σωμάτων καθώς μηδενικές αποστάσεις θα επιφέρουν σφάλματα στα τελικά αποτελέσματα.

Αρχικά για κάθε ένα από τα σώματα υπολογίζεται η απόσταση του με τα υπόλοιπα σώματα του συστήματος διαδοχικά όπως ορίζεται από τους βρόχους j και k στους τρεις άξονες, έπειτα υπολογίζεται οι συνολική τιμή της απόστασης των δύο σωμάτων. Αφού υπολογιστεί η εν λόγο απόσταση, σειρά έχει ο υπολογισμός της δύναμης που ασκεί το κάθε ένα σώμα στο

σώμα στο οποίο αναφέρονται οι υπολογισμοί που περιγράφονται. Οι δυνάμεις υπολογίζονται αθροιστικά σε σχέση με τα σώματα του βρόχου j ώστε να υπολογιστεί η συνολική δύναμη που θα ασκηθεί σε κάθε ένα από αυτά.

Μετά τον υπολογισμό των δυνάμεων να έχει λάβει τέλος πλέων μπορεί να υπολογιστεί η επιτάχυνση, η ταχύτητα και η νέα θέση των σωμάτων που προκύπτει από αυτές. Για τους εν λόγο υπολογισμούς θα χρειαστεί να δημιουργηθούν δύο εμφωλευμένοι βρόχοι for οι οποίοι περικλείονται από τον αρχικό βρόχο που αναφέρεται στις επαναλήψεις. Ο δεύτερος βρόχος όπως και πριν χρησιμεύει στην εναλλαγή των σωμάτων που συμμετάσχουν στο σύστημα, ο τρίτος εξ αυτών με δείκτη k αφορά την εναλλαγή των τριών συντεταγμένων στις οποίες αναλύονται τα μεγέθη τα οποία πρέπει να υπολογιστούν. Στην εικόνα που ακολουθεί (Εικόνα 31) φαίνονται ακριβώς πως υλοποιούνται οι υπολογισμοί που περιγράφηκαν στην συγκεκριμένη ενότητα.

```
114   // Υπολογισμοί
115   for(i=0;i<loop;i++)       // i: επαναλήψης πειράματος
116   {
117     for(j=0;j<N;j++)        // j: σώμα στο οποίο ασκουντε δυνάμεις
118     {
119       for(k=0;k<N;k++)      // k: σώμα το οποίο ασκεί δύναμη στο σώμα j
120       {
121         if(j!=k)
122         {
123           dx=C[k*3+0]-C[j*3+0];
124           dy=C[k*3+1]-C[j*3+1];
125           dz=C[k*3+2]-C[j*3+2];
126
127           d=sqrt(pow(dx,2)+pow(dy,2)+pow(dz,2));
128           dSquare=pow(d,2);
129
130           F[j*3+0]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dx);    // Αθροιστική δύναμη που ασκείται στο σώμα j
131           F[j*3+1]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dy);    // απο όλα τα υπόλοιπα σώματα k του συστήματος
132           F[j*3+2]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dz);    // στους τρίς άξονες (x,y,z)
133         }
134       }
135     }
136
137     for(j=0;j<N;j++)                        // j: σώμα για το οποίο υπολογίζονται οι νέες συντεταγμένες
138     {
139       for(k=0;k<3;k++)                      // k: συντεταγμένη που υπολογίζεται σε καθε κύκλο (x,y,z)
140       {
141         a=F[j*3+k]/M;                       // Επιτάχυνση a=F/M
142
143         F[j*3+k]=0.0;                       // Επαναφορά της δύναμης F σε 0
144
145         V[j*3+k]=V[j*3+k]+a*Dt;             // Ταχύτητα V(n+1)=Vn+a*Δt
146         C[j*3+k]=C[j*3+k]+V[j*3+k]*Dt;      // θέση R(n+1)=Rn+Vn*Δt
147       }
148     }
149   }
```

Εικόνα 31. Υπολογισμοί δυνάμεων και νέων θέσεων

Οι πίνακες C, V και F αφορούν τις θέσεις, ταχύτητες και δυνάμεις αντίστοιχα στις τρεις καρτεσιανές συντεταγμένες x, y και z. Σε ιδανικές συνθήκες οι τρεις αυτοί πίνακες θα αποτελούνταν από δύο διαστάσεις και θα ήταν μεγέθους Nx3. Στην συγκεκριμένη περίπτωση επειδή θα χρειαστεί να γίνουν συγκρίσεις μεταξύ των διαφόρων υλοποιήσεων θα πρέπει να ακολουθείται μια κοινή γραμμή στην υλοποίηση του κώδικα. Αποτέλεσμα αυτού του περιορισμού είναι πως οι πίνακες θα πρέπει να υιοθετήσουν μία άλλη μορφή. Πιο συγκεκριμένα οι πίνακες θα είναι μονοδιάστατοι μεγέθους N*3. Αποτέλεσμα αυτού είναι και ο τρόπος που υπολογίζεται το κελί στο οποίο αντιστοιχεί η κάθε τιμή που υπολογίζεται.

Η σειριακή εκτέλεση όπως αναμένεται θα είναι και η πιο αργή μεταξύ των υπολοίπων. Αυτό συμβαίνει καθώς εκτελεί διαδοχικά έναν ένα τους υπολογισμούς τόσο για τις δυνάμεις εντός του συστήματος όσο και για τις θέσεις των σωμάτων. Στον πίνακα που ακολουθεί (Πίνακας 10) φαίνεται οι χρόνοι εκτέλεσης σε συνδυασμό με το πλήθος σωμάτων και επαναλήψεων της προσομοίωσης.

Πλήθος επαναλήψεων (loop) *10^5 **Πλήθος Σωμάτων (N)**	**1**	**100**	**500**
2	0,044	3,706	18,519
4	0,214	20,563	102,714
6	0,512	50,315	251,193
8	0,938	92,763	463,472
10	1,490	147,967	740,676

Πίνακας 10. Χρόνοι εκτέλεσης σειριακού κώδικα σε δευτερόλεπτα (επαναλήψεις * 105)

7.2 Κατανεμημένη υλοποίηση με την MPI

Για την υλοποίηση του ίδιου προβλήματος μέσω της MPI θα χρειαστούν κάποιες αλλαγές σε σχέση με την σειριακή μέθοδο που παρουσιάστηκε. Κάθε διεργασία της MPI αναλαμβάνει ένα συγκεκριμένα πλήθος μαζών προς υπολογισμό. Όπως φάνηκε και από τις εξισώσεις που αφορούν πλήθος σωμάτων μεγαλύτερο των δύο ο υπολογισμός των δυνάμεων απαιτεί την να είναι γνωστή η θέση όλων των σωμάτων σε κάθε επανάληψη. Αυτό έχει ως αποτέλεσμα σε κάθε νέο κύκλο υπολογισμών κάθε διεργασία πρέπει να ενημερωθεί για τις θέσεις των σωμάτων που υπολογίστηκαν από τις υπόλοιπες στον προηγούμενο κύκλο. Για τν σωστή διαχείριση των δεδομένων το πλήθος των σωμάτων διαιρείται με πλήθος των διεργασιών. Στην περίπτωση που η διαίρεση αυτή είναι ατελείς το δεδομένα που περισσεύουν δεν μπορούν να ανατεθούν σε μία τυχαία διεργασία καθώς αυτό θα δημιουργήσει πρόβλημα στην ενημέρωση των πινάκων στο τέλος κάθε επανάληψης. Για την σωστή λειτουργία κάθε διεργασία πρέπει να αναλάβει δεδομένα που βρίσκονται στην σειρά. Καθώς η MPI δεν παρέχει κάποια έτοιμη συνάρτηση που να καλύπτει αυτή την ανάγκη είναι απαραίτητο να δηλωθούν κάποιες επιπλέων μεταβλητές για αυτό τον σκοπό. Στην εικόνα που ακολουθεί (Εικόνα 32) φαίνονται οι μεταβλητές που θα χρειαστούν για την υλοποίηση του κώδικα με επιτυχία.

```
45      // part: άρτυα διαίρεση δεδομένων
46      // remain: υπόλοιπο διαίρεσης part
47      int part,remain;
48
49      // start: άρχη δεδομένων διεργασίας
50      // end: τέλος δεδομένων διεργασίας
51      // maltitude: πλήθος μαζών ανα διεργασία
52      // length: πλήθος δεδομένων ανα διεργασία
53      int start,end,maltitude,length;
54
55      // Counts: πίνακας πλήθους δεδομένων όλων των διεργασιών
56      // StartPoint: σημείο που ξεκινούν τα δεδομένα στους βασικούς πίνακες της διεργασίας 0
57      // EndPoint: σημείο που τελειώνουν τα δεομένα στους βασικούς πίνακες της διεργασίας 0
58      int *Counts,*StartPoint,*EndPoint;
```

Εικόνα 32. Μεταβλητές MPI

Η μεταβλητή part αντιστοιχεί στην άρτυα διαίρεση μεταξύ του πλήθους των σωμάτων και του πλήθους των διεργασιών που συμμετέχουν. Αντίστοιχα η μεταβλητή remain αντιπροσωπεύει το υπόλοιπο της διαίρεσης της μεταβλητής part. Με την χρήση αυτών των

δύο θα οριστεί το σύνολο των σωμάτων με το οποίο θα εργαστεί η κάθε διεργασία. Στην σειριακή εκτέλεση για την αποφυγή υπολογισμών μεταξύ των ίδιων σωμάτων υλοποιούταν μια συνθήκη ελέγχου. Δουλειά της ήταν να αποτρέπει αυτούς τους υπολογισμούς συγκρίνοντας του δείκτες i και j των δύο εσωτερικών επαναλήψεων. Με την MPI αυτό θα είναι δυνατό καθώς κάθε διεργασία αποθηκεύει τα δεδομένα σε πίνακες σχεδιασμένους ξεχωριστά για την κάθε μία σύμφωνα με τον αριθμό των σωμάτων που της αντιστοιχούν. Για την υλοποίηση αυτής της συνθήκης και εδώ έχουν δημιουργηθεί οι μεταβλητές start και end που συμβολίζουν το σώμα με το οποίο ξεκινά το εύρος μίας διεργασίας και το σώμα με το οποίο σταματά αντίστοιχα. Για τον ορισμό των μεγεθών των πινάκων υποδοχής για κάθε διεργασία έχει δημιουργηθεί η μεταβλητή length. Η μεταβλητή αυτή είναι ίση με το πλήθος των σωμάτων κάθε διεργασίας πολλαπλασιασμένο επί τρία λόγο των συντεταγμένων στους τρις άξονες (x,y,z). Οι μεταβλητές αυτές είναι μοναδικές για κάθε διεργασία καθώς έχουν διαφορετικές τιμές ανάλογα σε ποιά από αυτές ανήκουν.

Σε κάθε νέα επανάληψη των υπολογισμών κάθε διεργασία χρειάζεται να γνωρίζει τις νέα θέσεις των σωμάτων που προέκυψαν από την προηγούμενη επανάληψη. Οι θέσεις αυτές όμως είναι γνωστές μόνο στην διεργασία που τις υπολόγισε. Για την εκκίνηση της νέας επανάληψης θα πρέπει όλες οι διεργασίες να ενημερωθούν σε σχέση με τις θέσεις των σωμάτων. Αυτό γίνεται μέσω μίας εντολής συλλογικής επικοινωνίας που χρειάζεται κάποιες πληροφορίες για κάθε διεργασία. Η εντολή αυτή είναι η MPI_Allgatherv() οποία είναι υπεύθυνη για την ανταλλαγή δεδομένων μεταξύ όλων των διεργασιών και την τοποθέτηση τους με μία συγκεκριμένη σειρά. Πιο συγκεκριμένα αυτό που κάνει η εντολή αυτή είναι να δημιουργεί ένα πίνακα λαμβάνοντας δεδομένα από όλες τις διεργασίες και στην συνέχει να αποστέλλει τον συνδυασμό αυτών των δεομένων σε όλες τους. Για την σωστή υλοποίηση της εντολής αυτής θα χρειαστεί η μεταβλητή multitude που αντιπροσωπεύει το πλήθος των σωμάτων κάθε διεργασίας. Η τιμή της μεταβλητής αυτής θα συγκεντρωθεί στον πίνακα Counts και θα αποσταλεί σε κάθε διεργασία ώστε αυτή η πληροφορία κάθε διεργασίας να είναι γνωστή μεταξύ όλων. Οι πίνακες StartPoint και EndPoint αντιστοιχούν στο σημείο που κάθε διεργασία ξεκινά από τον πίνακα των βασικών δεδομένων και που τελειώνει αντίστοιχα.

Αφού πλέων έχουν οριστεί οι κατάλληλες μεταβλητές για την κατανομή των σωμάτων μένει να γίνει ο διαχωρισμός αυτών. Για την αντιστοιχία μεταξύ των σωμάτων και των διεργασιών που αυτά αντιστοιχούν αναλαμβάνει ο κώδικας που ακολουθεί (Εικόνα 33).

```
99      // Ορισμός εμβέλειας σωμάτων ανα διεργασία
100     if(rank<remain)
101     {
102       start=rank*(part+1);
103       end=start+part;
104     }
105     else
106     {
107       start=rank*part+remain;
108       end=start+(part-1);
109     }
110
111     maltitude=end-start+1;
112     length=maltitude*3;
```

Εικόνα 33. Εμβέλεια διεργασιών

Για να γίνει πιο κατανοητή η λειτουργία του πιο πάνω κώδικα θα μελετηθεί ένα σενάριο ως παράδειγμα. Έστω πως υπάρχουν τέσσερεις κόμβοι (size=4) και πως το σύστημα που μελετάτε αποτελείται από δέκα σώματα (N=10). Σύμφωνα με όσα ισχύουν ως τώρα η μεταβλητή part αντιστοιχεί στην ακέραια διαίρεση μεταξύ του πλήθος σωμάτων και του πλήθους διεργασιών (part=N/size) όπου στην συγκεκριμένη περίπτωση είναι ίση με δύο (part=2). Η μεταβλητή remain αντιστοιχεί στο υπόλοιπο της προηγούμενης διαίρεσης που στην περίπτωση αυτή είναι επίσης ίσο με δύο (remain=2). Με αυτές τις δύο τιμές και των κώδικα της πιο πάνω εικόνας (Εικόνα 33) μπορούν να υπολογιστούν οι τιμές του πίνακα που ακολουθεί (Πίνακας 11).

Διεργασία	**start**	**end**	**multitude**	**length**
0	0	2	3	9
1	3	5	3	9
2	6	7	2	6
3	8	9	2	6

Πίνακας 11. Μεταβλητές διαμοιρασμού σωμάτων στις διεργασίες

Χρησιμοποιώντας της μεταβλητή length δεσμεύουμε την κατάλληλη μνήμη για τους πίνακες υποδοχής των δεδομένων. Η διεργασία με τιμή rank ίση με μηδέν (rank=0) υπολογίζει τα αρχικά δεδομένα όπως περιγράφθηκε και στο Κεφάλαιο 6. Στην συνέχεια πρέπει να αποστείλει σε κάθε διεργασία τα δεδομένα που της αναλογούν. Καθώς το μήκος των τμημάτων που πρέπει να αποσταλούν σε κάθε διεργασία δεν είναι σταθερό αυτά θα πρέπει να αποσταλούν μεμονωμένα σε κάθε διεργασία. Αυτό θα γίνει με την εντολή MPI_Send() και κάθε διεργασία θα παραλάβει τα δεδομένα αυτά με την ανάλογη MPI_Recv(). Η διαδικασία αυτή φαίνεται στον κώδικα της εικόνας που ακολουθεί (Εικόνα 34).

```
193     if(rank==0)
194     {
195       for(i=1;i<size;i++)
196       {
197         for(j=StartPoint[i];j<=EndPoint[i];j++)
198         {
199           MPI_Send(&C[j],1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
200           MPI_Send(&V[j],1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
201         }
202       }
203
204       for(i=StartPoint[rank];i<=EndPoint[rank];i++)
205       {
206         Cw[i]=C[i];
207         Vw[i]=V[i];
208       }
209     }
210
211     // Παραλαβή δεδομένων απο τις διεργασίες (C,V)
212     if(rank!=0)
213     {
214       for(j=0;j<length;j++)
215       {
216         MPI_Recv(&Cw[j],1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
217         MPI_Recv(&Vw[j],1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
218       }
219     }
```

Εικόνα 34. Διαμοιρασμός αρχικών δεδομένων στις διεργασίες

Όπως και πριν κάθε διεργασία υπολογίζει την δύναμη που ασκείται στο σώμα j από όλα τα υπόλοιπα σώματα k εντός του συστήματος. Με την χρήση των μεταβλητών start και end είναι δυνατός ο έλεγχος για την αποφυγή υπολογισμός ενός σώματος με τον εαυτό του. Αυτό δημιουργεί ένα πρόβλημα όταν οι τιμές αυτές περνάνε στου πίνακες των υπολογισμών καθώς οι πίνακες κάθε διεργασίας είναι μικρότερο μεγέθους. Για να μην βγούμε εκτός ορίων των πινάκων οι δείκτες αυτών θα πρέπει να μετασχηματιστούν ώστε να αντιστοιχούν στα επιθυμητά κελιά. Στην εικόνα που ακολουθεί (Εικόνα 35) φαίνεται πως θα υλοποιηθούν οι αναγκαίοι υπολογισμοί [12].

```
229     // Υπολογισμοί
230     for(i=0;i<loop;i++)               // i: επαναλήψης πειράματος
231     {
232       for(j=start;j<=end;j++)         // j: σώμα στο οποίο ασκούνται δυνάμεις
233       {
234         for(k=0;k<N;k++)              // k: σώμα το οποίο ασκεί δύναμη στο σώμα j
235         {
236           if(j!=k)
237           {
238             dx=C[k*3+0]-C[j*3+0];
239             dy=C[k*3+1]-C[j*3+1];
240             dz=C[k*3+2]-C[j*3+2];
241
242             d=sqrt(pow(dx,2)+pow(dy,2)+pow(dz,2));
243             dSquare=pow(d,2);
244
245             Fw[(j-start)*3+0]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dx);     // Αθροιστική δύναμη που ασκείται στο σώμα j
246             Fw[(j-start)*3+1]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dy);     // απο όλα τα υπόλοιπα σώματα k του συστήματος
247             Fw[(j-start)*3+2]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dz);     // στους τρίς άξονες (x,y,z)
248           }
249         }
250       }
251
252       for(j=0;j<maltitude;j++)                      // j: σώμα για το οποίο υπολογίζονται οι νέες συντεταγμένες
253       {
254         for(k=0;k<3;k++)                            // k: συντεταγμένη που υπολογίζεται σε καθε κύκλο (x,y,z)
255         {
256           a=Fw[j*3+k]/M;                            // Επιτάχυνση a=F/M
257
258           Fw[j*3+k]=0.0;                            // Επαναφορά της δύναμης F σε 0
259
260           Vw[j*3+k]=Vw[j*3+k]+a*Dt;                 // Ταχύτητα V(n+1)=Vn+a*Δt
261           Cw[j*3+k]=C[(j+start)*3+k]+Vw[j*3+k]*Dt;  // θέση R(n+1)=Rn+Vn*Δt
262         }
263       }
264
265       // Ενημέρωση πίνακα C με τις νέες θέσεις των σωμάτων για τον επόμενο κύκλο loop
266       MPI_Allgatherv(&Cw[0],length,MPI_DOUBLE,&C[0],Counts,StartPoint,MPI_DOUBLE,MPI_COMM_WORLD);
267     }
```

Εικόνα 35. Υπολογισμοί MPI

Αφού πλέων είναι ξεκάθαρο το πως λειτουργεί ο κώδικας που παρουσιάστηκε πιο πάνω στον πίνακα που ακολουθεί (Πίνακας 12) φαίνονται οι χρόνοι της παράλληλης εκτέλεσης της MPI.

Πλήθος επαναλήψεων (loop) *10^5 / **Πλήθος Σωμάτων (N)**	**1**	**100**	**500**
2	0,097	9,585	48,329
4	0,145	14,459	75,136
6	0,327	28,939	144,611
8	0,390	39,437	201,612
10	0,629	65,813	325,423

Πίνακας 12. Χρόνοι εκτέλεσης MPI με χρήση τεσσάρων διεργασιών

7.3 Παράλληλη υλοποίηση με την OpenMP

Σε αντίθεση με την MPI εδώ ο κώδικας έχει πιο απλή μορφή, είναι πιο κοντά στην σειριακή υλοποίηση της προηγούμενης ενότητας. Αφού οι κόμβοι της OpenMP χρησιμοποιούν κοινή μνήμη δεν υπάρχει η ανάγκη μεταφοράς δεδομένων και αλλαγών στην μέθοδο των υπολογισμών. Αρχικά υπολογίζονται η αρχική θέση και ταχύτητα κάθε σώματος και στην συνέχει γίνονται οι υπολογισμοί των νέων συντεταγμένων. Στην εικόνα που ακολουθεί (Εικόνα 36) φαίνονται οι υπολογισμοί που αναφέρθηκαν [12].

```
107   // Υπολογισμοί
108   #pragma omp parallel private(i,k,dx,dy,dz,d,dSquare,a)
109   {
110     for(i=0;i<loop;i++)                          // i: επαναλήψης πειράματως
111     {
112       #pragma omp for schedule(dynamic,1)
113         for(j=0;j<N;j++)                         // j: σώμα στο οποίο ασκουντε δυνάμεις
114         {
115           for(k=0;k<N;k++)                       // k: σώμα το οποίο ασκεί δύναμη στο σώμα j
116           {
117             if(j!=k)
118             {
119               dx=C[k*3+0]-C[j*3+0];
120               dy=C[k*3+1]-C[j*3+1];
121               dz=C[k*3+2]-C[j*3+2];
122
123               d=sqrt(pow(dx,2)+pow(dy,2)+pow(dz,2));
124               dSquare=pow(d,2);
125
126               F[j*3+0]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dx);          // Αθροιστική δύναμη που ασκείται στο σώμα j
127               F[j*3+1]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dy);          // απο όλα τα υπόλοιπα σώματα k του συστήματος
128               F[j*3+2]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dz);          // στους τρίς άξονες (x,y,z)
129             }
130           }
131         }
132       #pragma omp for schedule(dynamic,1)
133         for(j=0;j<N;j++)                         // j: σώμα για το οποίο υπολογίζω τις νέες συντεταγμένες
134         {
135           for(k=0;k<3;k++)                       // k: συντεταγμένη που υπολογίζεται σε καθε κύκλο x,y,z
136           {
137             a=F[j*3+k]/M;                        // Επιτάχυνση α=F/M
138
139             F[j*3+k]=0.0;                        // Επαναφορά της δύναμης F σε 0
140
141             V[j*3+k]=V[j*3+k]+a*Dt;              // Ταχύτητα V(n+1)=Vn+α*Δt
142             C[j*3+k]=C[j*3+k]+V[j*3+k]*Dt;       // θέση R(n+1)=Rn+Vn*Δt
143           }
144         }
145     }
146   }
```

Εικόνα 36. Υπολογισμοί OpenMP

Στην συγκεκριμένη περίπτωση ο εξωτερικός βρόχος δεν μπορεί να παραλληλιστεί καθώς σε κάθε επανάληψη χρειάζονται δεδομένα που υπολογίζονται από την προηγούμενη. Ο βρόχος που πρέπει να παραλληλιστεί είναι ο δεύτερος σε σειρά. Λόγο αυτής της δομής του προβλήματος δεν μπορεί να γίνει χρήση της εντολής collapse για να φτάσουμε στον εσωτερικό βρόχο. Μια λύση του προβλήματος θα ήταν να μεταφέρουμε την παραλληλία εντός του πρώτου βρόχου αλλά δεν θα ήταν αποδοτικό. Ο λόγος που αποφεύγεται αυτή η τεχνική είναι πως η κλήση της παραλληλίας είναι μία αρκετά δύσκολη διαδικασία και μέσα σε ένα βρόχο που θα εκτελεστεί χιλιάδες φορές θα επιβαρύνει κατά πολύ τον χρόνο διεξαγωγής των υπολογισμών. Με την τεχνική που χρησιμοποιείται στον πιο πάνω κώδικα γίνεται κλήση της παραλληλίας μία φορά πριν την διεξαγωγή των υπολογισμών. Με την δήλωση της μεταβλητής i στο σύνολο των private μεταβλητών κάθε κόμβος υλοποιεί εξ' ολοκλήρου τον πρώτο βρόχο και διασπά τον δεύτερο μεταξύ των αυτών όπως ορίζεται από την εντολή #pragma omp for. Οι μεταβλητές του συνόλου private είναι αυτές με τις οποίες κάθε κόμβος τελεί υπολογισμούς και μεταβάλει. Για τον λόγο αυτό έχουν οριστεί με αυτό

τον τρόπο για αποφυγή σφαλμάτων κατά την διάρκεια των υπολογισμών. Στον πίνακα που ακλουθεί (Πίνακας 13) φαίνονται οι χρόνοι εκτέλεσης με την χρήση της OpenMP.

Πλήθος επαναλήψεων (loop) *10^5 / **Πλήθος Σωμάτων (N)**	**1**	**100**	**500**
2	0,147	16,631	82,889
4	0,193	18,938	93,020
6	0,258	25,164	125,066
8	0,348	33,465	166,857
10	0,621	60,625	303,508

Πίνακας 13. Χρόνοι εκτέλεσης με την OpenMP

7.4 Υλοποίηση με συνδυασμό των MPI και OpenMP

Υλοποιώντας το πρόβλημα των N-σωμάτων με συνδυασμό των MPI και OpenMP δεν έχει κάποιο σημείο που θέλει προσοχή. Όπως και με την MPI χρειάζεται να δηλωθούν οι επιπλέων μεταβλητές όπως και πριν για την σωστή κατανομή των σωμάτων μεταξύ τον διεργασιών. Στην συνέχεια τα απαραίτητα δεδομένα πρέπει να αποσταλούν στις συμμετέχουσες διεργασίες. Κατά τον υπολογισμό ακολουθούνται και πάλι οι μετατροπές τον δεικτών των πινάκων για την αντιστοιχία των δεδομένων. Η διαδικασία που απολυθείτε είναι η διάσπαση του προβλήματος σε υποπροβλήματα και διανομή των αντίστοιχων δεδομένων στις διεργασίες. Έπειτα κάθε διεργασία χρησιμοποιεί τους κόμβους που έχει στην διάθεση της για τον παράλληλο υπολογισμό των δυνάμεων και θέσεων των σωμάτων.

Για την υλοποίηση τον υπολογισμών ακολουθείται η ίδια μέθοδος όπως και στην προηγούμενη ενότητα. Ο εξωτερικός βρόχος εκτελείται από κάθε κόμβο και οι εσωτερικοί βρόχοι εκτελούνται παράλληλα από κάθε έναν από αυτούς. Η μόνη διαφορά είναι η χρήση των μεταβλητών των διεργασιών της MPI. Στην εικόνα που ακολουθεί (Εικόνα 37) φαίνεται το τμήμα κώδικα που είναι υπεύθυνο για τους υπολογισμούς.

```
222   // Υπολογισμοί
223   #pragma omp parallel private(i,k,dx,dy,dz,d,dSquare,a)
224   {
225     for(i=0;i<loop;i++)                          // i: επαναλήψης πειράματως
226     {
227       #pragma omp for schedule(dynamic,1)
228         for(j=start;j<=end;j++)                  // j: σώμα στο οποίο ασκουντε δυνάμεις
229         {
230           for(k=0;k<N;k++)                       // k: σώμα το οποίο ασκεί δύναμη στο σώμα j
231           {
232             if(j!=k)
233             {
234               dx=C[k*3+0]-C[j*3+0];
235               dy=C[k*3+1]-C[j*3+1];
236               dz=C[k*3+2]-C[j*3+2];
237
238               d=sqrt(pow(dx,2)+pow(dy,2)+pow(dz,2));
239               dSquare=pow(d,2);
240
241               Fw[(j-start)*3+0]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dx);     // Αθροιστική δύναμη που ασκείται στο σώμα j
242               Fw[(j-start)*3+1]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dy);     // απο όλα τα υπόλοιπα σώματα k του συστήματος
243               Fw[(j-start)*3+2]-=G*mSquare*d/(pow(dSquare+expSquare,1.5)*dz);     // στους τρίς άξονες (x,y,z)
244             }
245           }
246         }
247
248       #pragma omp for schedule(dynamic,1)
249         for(j=0;j<maltitude;j++)                       // j: σώμα για το οποίο υπολογίζω τις νέες συντεταγμένες
250         {
251           for(k=0;k<3;k++)                             // k: συντεταγμένη που υπολογίζεται σε καθε κύκλο x,y,z
252           {
253             a=Fw[j*3+k]/M;                             // Επιτάχυνση a=F/M
254
255             Fw[j*3+k]=0.0;                             // Επαναφορά της δύναμης F σε 0
256
257             Vw[j*3+k]=Vw[j*3+k]+a*Dt;                  // Ταχύτητα V(n+1)=Vn+a*Δt
258             Cw[j*3+k]=C[(j+start)*3+k]+Vw[j*3+k]*Dt;   // θέση R(n+1)=Rn+Vn*Δt
259           }
260         }
261
262       // Ενημέρωση πίνακα C με τις νέες θέσεις των σωμάτων για τον επόμενο κύκλο loop
263       #pragma omp master
264       MPI_Allgatherv(&Cw[0],length,MPI_DOUBLE,&C[0],Counts,StartPoint,MPI_DOUBLE,MPI_COMM_WORLD);
265       #pragma omp barrier
266     }
267   }
```

Εικόνα 37. Υπολογισμοί MPI-OpenMP

7.5 Παράλληλη υλοποίηση με CUDA

Σε σχέση με ότι έχει παρουσιαστεί μέχρι στιγμής δεν υπάρχουν μεγάλες διαφορές στον τρόπο υλοποίησης του προβλήματος. Οι σταθερές μεταβλητές παραμένουν οι ίδιες όπως και σε κάθε μέθοδο. Οι αλλαγές ξεκινούν στην δήλωση των μεταβλητών του προβλήματος. Αφού πλέων η επίλυση του προβλήματος γίνεται μέσω της κάρτας γραφικών θα πρέπει πέραν από τις βασικές μεταβλητές στον host να δημιουργηθούν και οι κατάλληλες μεταβλητές στο device για την μεταφορά αυτών των δεδομένων. Οι μεταβλητές που είναι απαραίτητες για τον υπολογισμό των αρχικών δεδομένων (u,v,x,r,vesc,theta,phi) δεν θα χρειαστούν περαιτέρω δηλώσεις καθώς δεν είναι απαραίτητες στο device. Οι μόνες μεταβλητές που αιτούν δέσμευση μνήμης και στα δύο μέσα είναι οι βασικοί πίνακες (C,V,F) και τα δεδομένα

που παρέχονται από τν χρήστη για την διεξαγωγή του προβλήματος (loop.,N) [13]. Οι υπόλοιπες μεταβλητές του προβλήματος καθώς δεν έχουν λόγο ύπαρξης πέρα από του υπολογισμούς των που θα γίνουν για τις θέσεις των σωμάτων θα δηλωθούν εντός της συνάρτησης της κάρτας γραφικών. Η διαδικασία δεν αλλάζει καθόλου σε σχέση με την σειριακή υλοποίηση πέραν από την δέσμευση μνήμης μέσω της CUDA, την αντιγραφή των αρχικών πληροφοριών από τον host στο device και τέλος την αντιγραφή των αποτελεσμάτων πίσω στον host.

Το μόνο πρόβλημα που αντιμετωπίζει ο συγκεκριμένος κώδικας είμαι πως δεσμεύεται από τον ολοκλήρωση της εξωτερικής επανάληψης για να μπορέσει να προχωρήσει στον επόμενο υπολογισμό θέσεων των σωμάτων. Οι κάρτες γραφικών είναι κατάλληλες για προβλήματα όπου χιλιάδες δεδομένα μπορούν και πρέπει να υπολογιστούν ταυτόχρονα. Στο συγκεκριμένο πρόβλημα λόγο της επιλογής του τόπου αντιμετώπισης όπου το τμήμα που μπορεί να παραλληλιστεί (πλήθος σωμάτων N) μπορεί να φτάσει έως και δέκα (N=10) η χρήση της κάρτας γραφικών θα φέρει τα αντίθετα αποτελέσματα. Στον πίνακα που ακολουθεί (Πίνακας 14) φαίνεται φαίνονται οι χρόνοι υλοποίησης του προβλήματος με την χρήση CUDA για μικρό πλήθος επαναλήψεων σε σύγκριση με την σειριακή επίλυση για να γίνει πιο κατανοητή η διαφορά.

N	**Σειριακό**		**CUDA**	
	100	**1000**	**100**	**1000**
2	0,000	0,001	0,003	0,026
4	0,001	0,006	0,005	0,051
6	0,001	0,011	0,008	0,076
8	0,002	0,015	0,010	0,100
10	0,004	0,026	0,013	0,122

Πίνακας 14. Χρόνοι υλοποίησης CUDA σε σύγκριση με την σειριακή εκτέλεση

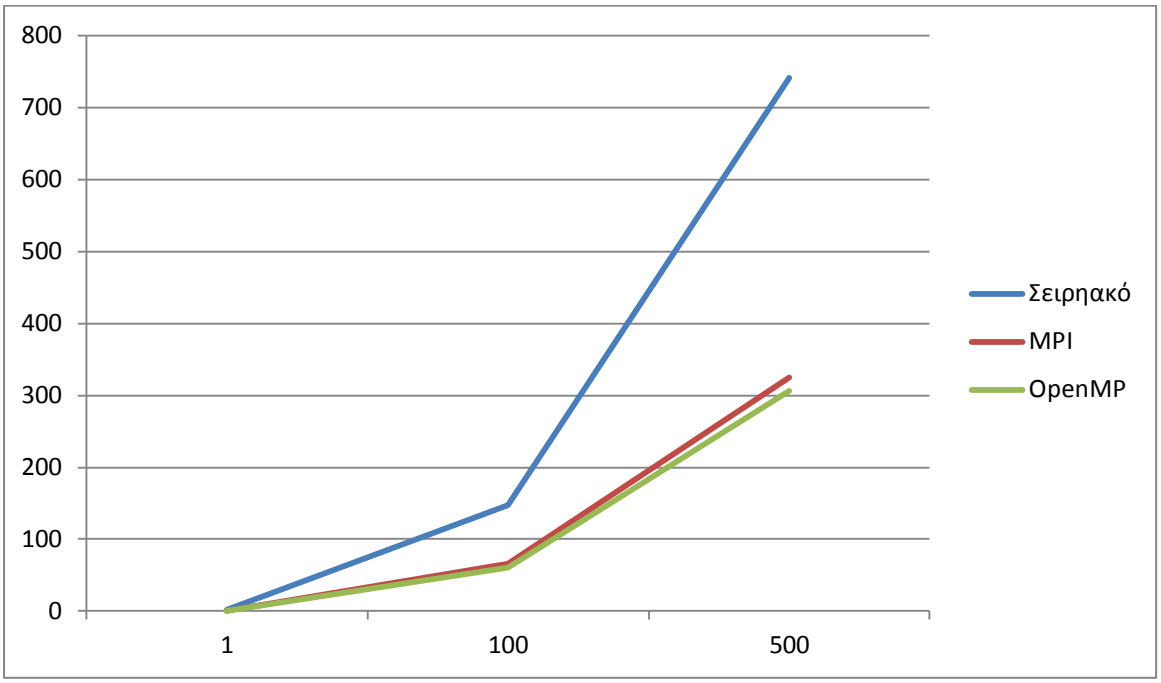
7.6 Σύγκριση μεθόδων

Σκοπός της παρούσας εργασίας είναι η σύγκρισή μεταξύ των μεθόδων που παρουσιάστηκαν ώστε να φανεί η αποδοτικότητα κάθε μίας από αυτές σε σχέση με τις υπόλοιπες. Αφού

εξηγήθηκαν οι σχετικοί κώδικες επίλυσης κάθε μεθόδου και παρουσιάστηκαν οι χρόνοι εκτέλεσης τώρα θα γίνει σύγκριση μεταξύ αυτών. Σε κάθε περίπτωση έχει γίνει χρονομέτρηση της προσομοίωσης για διάφορο αριθμό επαναλήψεων καθώς και πλήθος σωμάτων. Σε κάθε εκτέλεση έχουν χρησιμοποιηθεί τα ίδια αρχικά δεδομένα και κάθε κώδικας επιστρέφει τα ίδια ακριβώς αποτελέσματα έτσι ώστε οι συγκρίσεις μεταξύ των μεθόδων να είναι αντιπροσωπευτικές. Για κάθε μέθοδο οι υπολογισμοί είναι ίδιοι αλλά απαιτούνται αλλαγές για την σωστή λειτουργία του κώδικα. Αυτό έχει ως αποτέλεσμα για να μπορέσει να αποτυπωθεί το κατά πόσο συμφέρουσα μια μέθοδος είναι σε σχέση με κάποια άλλη πρέπει να μέσα στις μετρήσεις να συμπεριληφθούν οι διαφορές μεταξύ αυτών. Παράδειγμα με την χρήση της MPI χρειάζονται κάποιες επιπλέων δεσμεύσεις μνήμης σε σχέση με την σειριακή μέθοδο καθώς και κάποια μεταφορά δεδομένων που θα πρέπει να συμπεριληφθούν μέσα στον χρόνο εκτέλεσης.

Όπως αναφέρθηκε και πιο πάνω λόγο της αδυναμίας εκτέλεσης του συνδυαστικού κώδικα MPI-OpenMP σε ένα σύνολο συνδεδεμένων μεταξύ τους υπολογιστών δεν θα συμπεριληφθεί στη σύγκριση. Ακόμη λόγο της φύσης του προβλήματος στην προηγούμενη ενότητα παρουσιάστηκε η μη καταλληλότητα της CUDA προσέγγισης αυτού, για τον λόγο αυτό δεν θα συμπεριληφθεί στις παρακάτω συγκρίσεις.

Στο διάγραμμα που ακλουθεί (Εικόνα 38) φαίνονται οι χρόνοι σειριακής, MPI και OpenMP εκτέλεσης σύμφωνα με τα δεδομένα από τους ανάλογους πιο πάνω πίνακες (Πίνακας 10,12,13)



Εικόνα 38. Σύγκριση χρόνων εκτέλεσης

8 Συμπέρασμα

Βλέποντας την πολυπλοκότητα κάθε κώδικα και τους χρόνους εκτέλεσης για διάφορο αριθμό επαναλήψεων και πλήθος σωμάτων το αποτέλεσμα είναι ξεκάθαρο. Έναντι της σειριακής εκτέλεσης και η MPI όπως και η OpenMP προσφέρουν σημαντική επιτάχυνση. Στο συγκεκριμένο πρόβλημα το οποίο αντιμετωπίστηκε με την πιο επιβαρυμένη μέθοδο και λόγο πράξεων καθώς και ακρίβειας αποτελεσμάτων οι χρόνοι είχαν αισθητή διαφορά. Αντίθετα με την χρήση της CUDA υπήρχε επιβράδυνση της επίλυσης λόγο των περιορισμών της μεθόδου.

Σημαντικό κομμάτι που πρέπει να ληφθεί υπόψη όσων αφορά την MPI είναι ο διαμοιρασμός και η μεταφορά των δεδομένων που θα πρέπει να γίνει. Κάθε διεργασία θα πρέπει να αναλάβει όσο το δυνατόν ίσου μεγέθους τμήματα για να έχει λόγο ύπαρξης η παραλληλία. Σε προβλήματα που είναι αναγκαία η συχνή μεταφορά δεδομένων μεταξύ των διεργασιών υπάρχει το ενδεχόμενο η καθυστέρηση που θα προκύψει από τιε επικοινωνίες να επιβαρύνει τον χρόνο επίλυσης αισθητά. Έτσι σημαντικό κομμάτι δεν είναι μόνο ο παραλληλισμός των πράξεων. Αντίθετα με την χρήση της OpenMP προβλήματα μεταφοράς δεδομένων δεν υφίστανται. Σημαντικό τμήμα είναι η διαχείριση των κοινών και ιδιωτικών μεταβλητών για την σωστή υλοποίηση του προβλήματος και η αποφυγή σφαλμάτων. Όσων αφορά την συνδυαστική μέθοδο επίλυσης μπορεί να μην υπάρχει η δυνατότητα χρονομέτρησης αλλά υλοποιώντας το κομμάτι κώδικα που αντιστοιχεί βγαίνουν τα ανάλογα συμπεράσματα. Με τον συνδυασμό των δύο τεχνικών είναι αρκετά σημαντική η διαχείριση και διαχωρισμός των δεδομένων μεταξύ διεργασιών και κόμβων καθώς τα προβλήματα που αντιμετωπίζει κάθε μια μέθοδος ξεχωριστά σε τέτοια μορφής κώδικα συνυπάρχουν και αυξάνουν την πιθανότητα να προκύψει κάποιο σφάλμα.

Τέλος η υλοποίηση ενός προβλήματος μέσω της κάρτας γραφικών με χρήση CUDA είναι ένα αρκετά σημαντικό εργαλείο καθώς μπορεί να επίσπευση σε πολύ μεγάλο βαθμό τους υπολογισμούς. Παρ’ όλα αυτά στην συγκεκριμένη περίπτωση αυτό δεν ήταν δυνατό λόγο των περιορισμών του προβλήματος βάση της εξάρτησης των δεδομένων. Αποτέλεσμα αυτού πως ανάλογα την φύση του προβλήματος που αντιμετωπίζεται ενδείκνυται και η ανάλογη μέθοδος ώστε η υλοποίηση να είναι όσο το δυνατόν πιο αποδοτική.

Βιβλιογραφία

Ακολουθούν οι βιβλιογραφικές αναφορές (πηγές) της Εργασίας.

[1] Kinderman Albert J., Monahan John F., (1997). Computer Generation of Random Variables using the Ratio of Uniform Deviates

[2] Leva Joseph F., (1992). A Fast Normal Random Number Generator

[3] Peter S. Pacheco, (2015). Εισαγωγή στον Παράλληλο Προγραμματισμό

[4] (2015). OpenMP Application Program Interface Ανακτήθηκε από openmp-4.5.pdf

[5] Βασίλειος Β. Δημακόπουλος, (2017). Παράλληλα Συστήματα και Προγραμματισμός

[6] Ζαχαρούλης Α. (2014). Φυσική Θεωρεία και Πρακτική

[7] Καβακιώτη Μαριόρα, (2010). Κατανεμημένη επίλυση προβλήματος Ν-σωμάτων

[8] Τσελίκης Γ. Σ., Τσελίκης Ν. Δ. (2012). C Από την θεωρεία στην Εφαρμογή Β΄Έκδοση

[9] Open MPI Documentation (open-mpi.org)

[10] CUDA Toolkit Documentation (nvidia.com)

[11] CUDA C/C++ Basics (nvidia.com)

[12] nBody.pdf (usask.ca) **sxolio mpi open solution**

[13] PowerPoint Presentation (rcsste.edu.jo)

[14] Chapter 31. Fast N-Body Simulation with CUDA | NVIDIA Developer

Υπεύθυνη Δήλωση Συγγραφέα:

Δηλώνω ρητά ότι, σύμφωνα με το άρθρο 8 του Ν.1599/1986, η παρούσα εργασία αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης.