ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση του αλγορίθμου LMSOR με τεχνικές παράλληλου προγραμματισμού (MPI και OpenMP) και μετρήσεις στον υπερυπολογιστή Aris.

Αθανάσιος-Φ-Μελής
Παρασκευάς-Π-Μανωλάς

Επιβλέπων **Ιωάννης Κοτρώνης, Αναπληρωτής Καθηγητής**

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2017

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση του αλγορίθμου LMSOR με τεχνικές παράλληλου προγραμματισμού (MPI και OpenMP) και μετρήσεις στον υπερυπολογιστή Aris .

Αθανάσιος-Φ-Μελής

Α.Μ.: 1115201200108

Παρασκευάς-Π-Μανωλάς

Α.Μ.: 1115201200100

ΕΠΙΒΛΕΠΟΝΤΕΣ: Ιωάννης Κοτρώνης, Αναπληρωτής Καθηγητής

ΠΕΡΙΛΗΨΗ

Σκοπός αυτής της πτυχιακής εργασίας είναι η ανάπτυξη κώδικα για την υλοποίηση του αλγορίθμου LMSOR (Local Modified Successive Over-Relaxation), ο οποίος επιλύει διαφορικές εξισώσεις, με τεχνικές παραλληλοποίησης MPI και OpenMP. Το πρώτο μας βήμα ήταν να μελετήσουμε ένα δοσμένο πρόγραμμα που υλοποιούσε τον αλγόριθμο LMSOR με OpenMP παραλληλοποίηση για την επίλυση της Εξίσωσης Διάχυσης Θερμότητας 2ας τάξης. Αυτός ο κώδικας υλοποιούσε OpenMP παραλληλοποίηση σε blocks στους αρχικούς πίνακες που χρησιμοποιεί ο LMSOR. Έγιναν οι απαραίτητες τροποποιήσεις, έτσι ώστε η παραλληλοποίηση αυτή να γίνεται στους επαναληπτικούς βρόγχους, οι οποίοι αποτελούν τον “πυρήνα” του αλγορίθμου. Εν συνεχεία, πραγματοποιήθηκε η υλοποίηση της παραλληλοποίησης σε processes (MPI) και ένα υβριδικό πρόγραμμα MPI και OpenMP. Εκτενείς μετρήσεις, πάνω σε διαφορετικά μεγέθη πινάκων, αριθμούς διεργασιών και αριθμούς νημάτων, πάρθηκαν στον υπερυπολογιστή Aris και τα αποτελέσματα μελετήθηκαν με σκοπό να καταλήξουμε στον ιδανικό αριθμό διεργασιών και νημάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Παράλληλα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: MPI, OpenMP, Aris HPC, LMSOR, red black reordering

ABSTRACT

The purpose of this thesis is the code development for the implementation of the LMSOR (Local Modified Successive Over-Relaxation) algorithm, which solves differential equations, by using parallelization techniques such as MPI and OpenMP. Our first step, was to study a given program, that implemented the LMSOR algorithm using OpenMP parallelization in order to solve the second order heat dissipation equation. This given code implemented OpenMP parallelization on blocks of the initial arrays that are being used by LMSOR. The necessary changes were made, so that the OpenMP parallelization is done on the repetitive loops,which are the “core” of the algorithm. Thereinafter, the implementation of parallelization in processes(MPI) and a hybrid program with both MPI and OpenMP were carried out. Extended calculations, based on different array sizes, number of processes and number of threads, were taken using the Aris Hypercomputer and the results were studied in order to end up to the ideal number of processes and threads that can be used.

SUBJECT AREA: Parallel Systems

KEYWORDS: MPI, OpenMP, Aris HPC, LMSOR, red black reordering

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα θέλαμε να ευχαριστήσουμε θερμά τον επιβλέποντα καθηγητή Δρ. Ι. Κοτρώνη για τη συνεργασία και την πολύτιμη συμβολή του στην ολοκλήρωση της. Επίσης, θα θέλαμε να ευχαριστήσουμε τους κυρίους Η. Κωσταντινίδη και Π. Γιαννακόπουλο, καθώς δίχως τις δικές τους πτυχιακές εργασίες, αυτή η πτυχιακή εργασία δεν θα ήταν εφικτή.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

1. ΕΙΣΑΓΩΓΗ

Στη σύγχρονη επόχη, οι επεξεργαστές (CPU) επιτελούν έναν ολοένα και σημαντικότερο ρόλο στην καθημερινότητά μας. Η πλειοψηφία των συσκευών που χρησιμοποιούνται καθημερινά από τον μέσο πολίτη, διαθέτουν πολυπύρηνους επεξεργαστές. Για παράδειγμα, τα κινητά τηλέφωνα (smartphones) μπορεί να διαθέτουν μέχρι και επεξεργαστές δέκα πυρήνων. Αυτό τους καθιστά ένα αναπόσπαστο κομμάτι της τεχνολογικής ανάπτυξης των ημερών μας. Η εις βάθος γνώση και εξοικείωση με αυτούς μπορεί να οδηγήσει σε καινοτόμα επιστημονικά εγχειρήματα.

Η πραγματική δύναμη των επεξεργαστών κρύβεται στην έννοια της παραλληλίας. Όπως προαναφέρθηκε, συναντάει κανείς συσκευές με πολλαπλούς πυρήνες, ίσως και δεκάδες και είναι πλέον αναγκαία η ανάπτυξη εφαρμογών με τέτοιο τρόπο ώστε να εκμεταλλεύονται στο έπακρο τις δυνατότητες τους. Σε αυτό το σημείο, αντιλαμβανόμαστε την ανάγκη της παραλληλίας. Είναι κοινή πρακτική να μετατρέπονται τα σειριακά προγράμματα σε παράλληλα, ώστε να βελτιώνονται οι επιδόσεις τους. Οι προγραμματιστές έχουν στη διάθεση τους αρκετά εργαλεία, για να επιτελέσουν αυτές τις μετατροπές. Παραλληλία μπορεί να επιτευχθεί είτε σε επίπεδο διεργασιών (processes) είτε σε επίπεδο νημάτων (threads). Όσον αφορά στο επίπεδο διεργασιών, χρησιμοποιείται ευρέως το MPI και όσον αφορά στο επίπεδο νημάτων το OpenMP. Επίσης, μπορούν να συνδυαστούν τα δύο παραπάνω για την δημιουργία ενός υβριδικού (hybrid) προγράμματος που εκμεταλλεύεται την παραλληλία και στα δύο επίπεδα.

Η βελτίωση της απόδοσης ενός προγράμματος που έχει μετατραπεί από σειριακό σε παράλληλο είναι ακόμα πιο εμφανής όταν αυτό εκτελείται σε έναν υπερυπολογιστή. Στην Ελλάδα, έχουμε τον υπερυπολογιστή Aris, στον οποίο μας δίνεται η δυνατότητα να τρέξουμε παράλληλα προγράμματα σε πολύ μεγάλο αριθμό πυρήνων [1].

Στην παρούσα εργασία, έχει υλοποιηθεί ένα πρόγραμμα που αξιοποιεί την τεχνική παραλληλοποίησης με διεργασίες (MPI), έχουν γίνει κάποιες μελέτες πάνω στο OpenMP και την υβριδική του συνύπαρξη με το MPI και έχουν παρθεί εκτενείς μετρήσεις στον υπερυπολογιστή Aris. Με την χρήση του υπερυπολογιστή έγινε εμφανής η αλλαγή στην κλιμάκωση (scalability) του προγράμματος μας μέσα από τις μετρήσεις που πήραμε για διαφορετικούς αριθμούς πυρήνων και διαφορετικά μεγέθη προβλήματος. Αφετηρία αυτής της εργασίας αποτέλεσε μια προϋπάρχουσα υλοποίηση με OpenMP του αλγορίθμου LMSOR [2]. Στόχος της πτυχιακής αυτής εργασίας είναι η επίλυση της εξίσωσης διάχυσης θερμότητας 2ας τάξης με χρήση της τοπικής μεθόδου LMSOR (Local Modified Successive Over-Relaxation) [3], τροποποιημένης ώστε να εκμεταλλεύεται πλήρως τις δυνατότητες που προσφέρει ο υπερυπολογιστής Aris (Aris Hypercomputer).

Στη συνέχεια ακολουθούν κάποια εισαγωγικά στοιχεία για το MPI, το mpiP, το OpenMP, τον υπερυπολογιστή Aris, τον αλγόριθμο LMSOR και ορισμοί διάφορων εννοιών.

1.1 Εισαγωγή στο MPI

Το μοντέλο μεταβίβασης μηνυμάτων που υλοποιείται με τη βοήθεια των συναρτήσεων της βιβλιοθήκης του MPI (Message Passing Interface) είναι ένα από τα προγραμματιστικά μοντέλα που μπορούμε να χρησιμοποιήσουμε για την ανάπτυξη παράλληλων εφαρμογών σε παράλληλους υπολογιστές MIMD (Multiple Instructions

Multiple Data) κατανεμημένης μνήμης, αν και μιλώντας γενικά, η χρήση του προτύπου είναι δυνατή ακόμη και σε συστήματα κοινής μνήμης. Αυτό το μοντέλο, ορίζει τους θεμελιώδεις κανόνες επικοινωνίας διεργασιών δια της ανταλλαγής κατάλληλα διαμορφωμένων μηνυμάτων και περιλαμβάνει μια βιβλιοθήκη συναρτήσεων που μπορεί να χρησιμοποιηθεί μέσα από προγράμματα που είναι γραμμένα σε κάποια από τις γλώσσες προγραμματισμού C, C++ και Fortran για την ανάπτυξη παράλληλων εφαρμογών. Αυτή η βιβλιοθήκη συναρτήσεων μπορεί να χρησιμοποιηθεί από οποιαδήποτε γλώσσα προγραμματισμού που μπορεί να διασυνδέεται με τέτοιες βιβλιοθήκες και άρα υπάρχουν εφαρμογές MPI υλοποιημένες σε Java, C# και Python [4].

Το MPI εμφανίστηκε σχετικά πρόσφατα στο χώρο της παράλληλης επεξεργασίας αλλά δεν άργησε να καθιερωθεί ως το πρότυπο της ανάπτυξης εφαρμογών αυτού του τύπου, που χαρακτηρίζεται από φορητότητα και λειτουργία σε συνθήκες υψηλής απόδοσης. Μέχρι σήμερα παραμένει το κυρίαρχο μοντέλο που χρησιμοποιείται σε υπολογιστές υψηλής απόδοσης. Η ταχεία εξέλιξη της τεχνολογίας των παράλληλων αρχιτεκτονικών οδήγησε στη σχεδίαση του προτύπου με τέτοιο τρόπο, έτσι ώστε να μπορεί να καλύψει ανάγκες που θα προκύψουν στο μέλλον. Για αυτό το λόγο χαρακτηρίζεται από εύκολες διαδικασίες επέκτασης και παραμετροποίησης. Ταυτόχρονα, το MPI μας παρέχει τη δυνατότητα εκτέλεσης του σε μια μεγάλη ποικιλία παράλληλων συστημάτων αλλά ακόμη και σε ετερογενή συστήματα, δηλαδή σε συστήματα που αποτελούνται από υπολογιστικούς κόμβους διαφορετικού τύπου. Συνεπώς, το MPI διέπεται από φορητότητα, επεκτασιμότητα και υψηλή απόδοση και αυτοί είναι οι στόχοι πάνω στους οποίους βασίστηκε ο σχεδιασμός του.

Αν και το MPI δεν σχεδιάστηκε ως ένα κατανεμημένο λειτουργικό σύστημα αλλά ως μια βιβλιοθήκη συναρτήσεων για την ανάπτυξη παράλληλων εφαρμογών, εν τούτοις επιτρέπει την πραγματοποίηση κάποιων από τις διαδικασίες ενός λειτουργικού συστήματος όπως είναι για παράδειγμα η δημιουργία διεργασιών. Οι βασικές δομικές μονάδες αυτού του προτύπου είναι οι διεργασίες, οι οποίες επικοινωνούν μεταξύ τους μέσα από ειδικά περιβάλλοντα επικοινωνίας που είναι γνωστά ως communicators.

Σε αντίθεση με το σειριακό μοντέλο προγραμματισμού στο οποίο λαμβάνει χώρα η χρήση ενός και μοναδικού επεξεργαστή που επικοινωνεί συνεχώς με τη κεντρική μνήμη του συστήματος δια μέσου διαδικασιών ανάγνωσης και εγγραφής, στο αντίστοιχο παράλληλο μοντέλο, υπονοείται η χρήση πολλών διαφορετικών επεξεργαστών κάθε ένας εκ των οποίων χρησιμοποιεί τη δική του περιοχή μνήμης και τις δικές του μεταβλητές προγράμματος. Αυτό σημαίνει πως οι εν λόγω επεξεργαστές δεν μοιράζονται κάποιο κοινό φυσικό χώρο διευθύνσεων μνήμης και επομένως οι ενέργειες μιας διεργασίας που εκτελείται σε έναν υπολογιστή, δεν είναι ορατές από τις υπόλοιπες διεργασίες του συστήματος. Εάν όμως, κάποια διεργασία απαιτεί τη χρήση μιας μεταβλητής η οποία ανήκει στο χώρο κάποιας άλλης διεργασίας, η μεταβλητή αυτή δεν είναι κοινή στις δύο διεργασίες, αλλά θα πρέπει να αποσταλεί από τη μια διεργασία στην άλλη δια μέσου κάποιου μηνύματος (message).

Έχουν υπάρξει πολλές εκδόσεις και υλοποιήσεις του MPI. Μετά την κυκλοφορία της πρώτης έκδοσης του προτύπου MPI, άρχισαν να εμφανίζονται οι πρώτες υλοποιήσεις του προτύπου, τόσο υπό την μορφή ανοικτού λογισμικού όσο και υπό τη μορφή εμπορικών εφαρμογών που έχουν κατασκευαστεί από τις μεγάλες εταιρείες λογισμικού. Κάποιες από τις πιο σημαντικές υλοποιήσεις είναι οι MPICH, LAM (Local Area Multicomputer), CHIMP (Common High Level Interface to Message Passing), IBM MPI, HP MPI, Sun MPI, Digital MPI και SGI MPI. Όσον αφορά στις εκδόσεις, η πρώτη

έκδοση υποστηρίζει δύο τύπους επικοινωνίας, τις επικοινωνίες από σημείο σε σημείο και τις συλλογικές επικοινωνίες. Στην επικοινωνία από σημείο σε σημείο συμμετέχουν δύο διεργασίες, η μία ως αποστολέας και η άλλη ως παραλήπτης, ενώ στη συλλογική επικοινωνία συμμετέχουν όλες οι διεργασίες της τρέχουσας ομάδας διεργασιών. Η δεύτερη έκδοση εισήγαγε τις μονομερείς επικοινωνίες στις οποίες οι διαδικασίες διακίνησης δεδομένων πραγματοποιούνται μόνο από τη μία διεργασία, η οποία προσπελαύνει την περιοχή μνήμης της απομακρυσμένης διεργασίας με τη βοήθεια ειδικών συναρτήσεων που παρέχει το πρότυπο. Πέρα από αυτούς τους τρόπους επικοινωνίας το πρότυπο παρέχει και άλλα πολλά χαρακτηριστικά όπως η δημιουργία παραγόμενων τύπων δεδομένων που επιτρέπουν την οργάνωση των δεδομένων σε πολύπλοκες δομές, η υποστήριξη τοπολογιών διεργασιών, καθώς και η δυναμική δημιουργία και διαχείρηση διεργασιών.

1.2 Εισαγωγή στο mpiP

Το mpiP είναι ένα πρόγραμμα μέτρησης απόδοσης για εφαρμογές MPI. Χάρη στο mpiP, μπορεί να βρεθεί πλέον η πλειοψηφία των προβλημάτων ενός προγράμματος MPI, έως και 90%, καθώς μας παρέχει πληροφορίες που αφορούν τις καθυστερήσεις κατά την εκτέλεση του. Επειδή συλλέγει μόνο στατιστικά δεδομένα σχετικά με τις λειτουργίες MPI, το mpiP παράγει σημαντικά λιγότερα αποτελέσματα και πολύ λιγότερα δεδομένα από τα εργαλεία εντοπισμού. Χρησιμοποιεί μόνο επικοινωνία κατά τη δημιουργία αναφορών, συνήθως στο τέλος του πειράματος, για τη συγχώνευση αποτελεσμάτων από όλες τις εργασίες σε ένα αρχείο εξόδου. Το mpiP έχει δοκιμαστεί σε μια πληθώρα απο C, C++ και Fortran εφαρμογές με 2 έως και 262.144 διεργασίες.

Η χρήση του mpiP είναι πολύ απλή και εύκολη. Το mpiP συγκεντρώνει δεδομένα MPI και είναι μια βιβλιοθήκη που χρησιμοποιείται κατα το στάδιο που καλείται ο linker (link-time library). Συνεπώς, δεν είναι απαραίτητη η επαναμεταγλώττιση μιας εφαρμογής για να χρησιμοποιηθεί το mpiP.

Το mpiP δεν συγκεντρώνει πληροφορίες για όλες τις κλήσεις των MPI συναρτήσεων. Τοπικές κλήσεις, όπως για παράδειγμα η κλήση της συνάρτησης MPI_Comm_size , παραλείπονται από τις μετρήσεις της βιβλιοθήκης για να ελαχιστοποιηθεί η έξοδος του mpiP.

Μετά το πέρας του τρεξίματος μιας εφαρμογής, το mpiP δημιουργεί ένα αρχείο με τα αποτελέσματα του στον φάκελο του προγράμματος που εκτελέσαμε. Τα αποτελέσματα αυτού του αρχείου εξόδου χωρίζονται σε 5 τμήματα. Το πρώτο τμήμα αποτελείται από πληροφορίες κεφαλίδας, που μας παρέχουν βασικές πληροφορίες σχετικά με την απόδοση του πειράματος μας. Στη συνέχεια, παρέχεται μια σύνοψη του χρόνου της εφαρμογής στο MPI. Ως AppTime μας δίνεται το χρονικό διάστημα που ξεκινά από τη στιγμή που τελειώνει η συνάρτηση MPI_Init και λήγει μόλις καλεστεί η MPI_Finalize. Επίσης, ως MPI_Time μας δίνεται ο χρόνος που καταλαμβάνουν όλες οι κλήσεις MPI συναρτήσεων μέσα στο χρονικό διάστημα AppTime και αντίστοιχα μας δίνεται ο επί τοις εκατό λόγος της ποσότητας MPI_Time ως προς την ποσότητα AppTime. Στο τρίτο τμήμα των αποτελεσμάτων, υποδεικνύονται όλες οι MPI κλήσεις μέσω της εφαρμογής. Για κάθε κλήση MPI συνάρτησης δίνεται ένας ξεχωριστός αριθμός ταυτοποίησης (callsite ID) που αφορά στο συγκεκριμένο αρχείο, ο τύπος της εκάστοτε MPI κλήσης, το όνομα της συνάρτησης, το όνομα του αρχείου και η γραμμή κατά την οποία καλέστηκε η συνάρτηση. Το επόμενο τμήμα αποτελεσμάτων είναι παρόμοιο με το προηγούμενο,

αλλά ομαδοποιεί και εμφανίζει τις κορυφαίες 20 κλήσεις με βάση το συνολικό μέγεθος των μηνυμάτων που στάλθηκαν. Τέλος, εμφανίζονται κάποια στατιστικά στοιχεία για κάθε κλήση σε κάθε διεργασία με βάση το χρόνο διάρκειας και τα μεγέθη των μηνυμάτων.

Περισσότερες πληροφορίες σχετικά με τις εντολές τρεξίματος και άλλα πολλά μπορούν να βρεθούν στον επίσημο ιστότοπο [5] .

1.3 Εισαγωγή στο OpenMP

Το OpenMP (Open Multi-Processing) είναι μια διεπαφή προγραμματισμού εφαρμογών (API) που υποστηρίζει προγραμματισμό πολλαπλών επεξεργαστών κοινής μνήμης σε C, C++ και Fortran στις περισσότερες αρχιτεκτονικές συνόλων εντολών και λειτουργικά συστήματα HP-UX, Linux, macOS και Windows [6]. Αποτελείται από ένα σύνολο οδηγιών μεταγλωττιστή, ρουτίνες βιβλιοθηκών και μεταβλητές περιβάλλοντος, που επηρεάζουν τη συμπεριφορά κατά την εκτέλεση.

Το OpenMP το διαχειρίζεται η συμβουλευτική επιτροπή OpenMP Architecture Review (ή OpenMP ARB), η οποία ορίζεται από κοινού από μια ομάδα σημαντικών προμηθευτών υλικού και λογισμικού, συμπεριλαμβανομένων των AMD, IBM, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation και πολλών άλλων.

Χρησιμοποιεί ένα φορητό, κλιμακωτό μοντέλο που δίνει στους προγραμματιστές μια απλή και ευέλικτη διεπαφή για την ανάπτυξη παράλληλων εφαρμογών για πλατφόρμες που κυμαίνονται από τον τυπικό επιτραπέζιο υπολογιστή έως τον υπερυπολογιστή.

Μια εφαρμογή που έχει κατασκευαστεί με το υβριδικό μοντέλο παράλληλου προγραμματισμού μπορεί να εκτελεστεί σε ένα σύμπλεγμα υπολογιστών, που χρησιμοποιεί και το OpenMP και τη διεπαφή μηνυμάτων MPI, έτσι ώστε το OpenMP να χρησιμοποιείται για παραλληλισμό εντός ενός κόμβου (multi-core) ενώ το MPI χρησιμοποιείται για παραλληλισμό μεταξύ κόμβων.

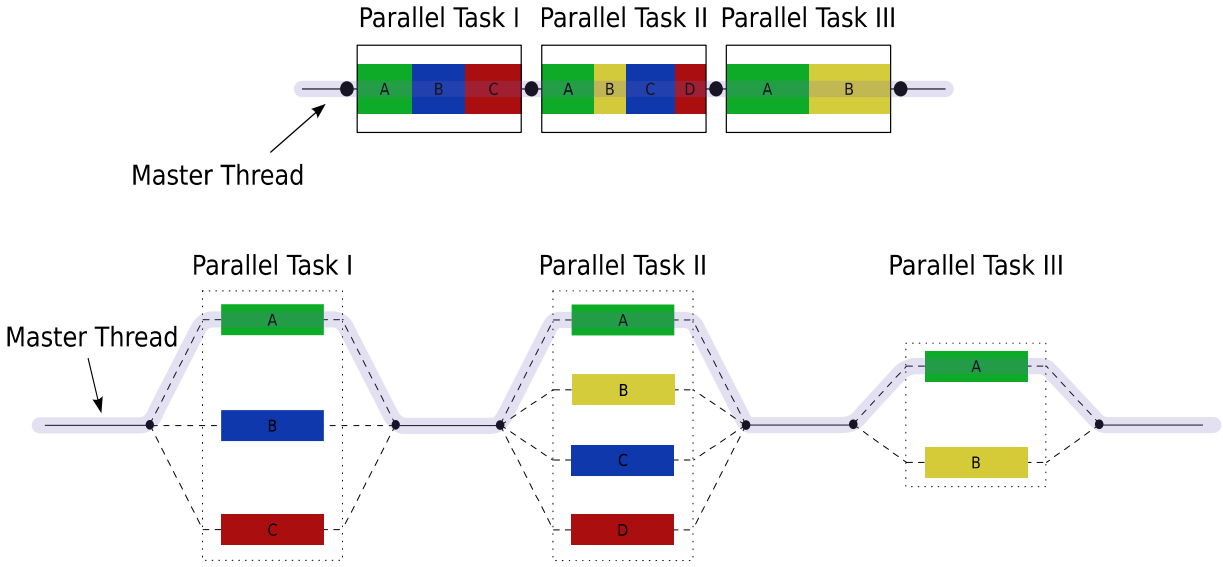
Το OpenMP είναι μια υλοποίηση πολλαπλών νημάτων, μια μέθοδος παραλληλισμού, όπου ένα κύριο νήμα (μια σειρά οδηγιών που εκτελούνται διαδοχικά), δημιουργεί (fork) ένα καθορισμένο αριθμό νημάτων σκλάβων και το σύστημα διαιρεί μια εργασία σε σκέλη τα οποία τα κατανέμει μεταξύ τους. Στη συνέχεια, τα νήματα τρέχουν ταυτόχρονα, με το περιβάλλον χρόνου εκτέλεσης που κατανέμει τα νήματα σε διαφορετικούς επεξεργαστές. Ο αριθμός των νημάτων μπορεί να καθοριστεί από το χρήστη χρησιμοποιώντας τις ανάλογες συναρτήσεις του OpenMP.

Το τμήμα του κώδικα που προορίζεται να τρέχει παράλληλα σημειώνεται αντίστοιχα, με μια οδηγία μεταγλωττιστή που θα προκαλέσει τη δημιουργία των νημάτων πριν την εκτέλεση του κώδικα αυτού. Κάθε νήμα έχει ένα αναγνωριστικό (id) προσαρτημένο σε αυτό, το οποίο μπορεί να ληφθεί χρησιμοποιώντας μια συνάρτηση (ονομάζεται omp_get_thread_num()). Αυτό το id του νήματος είναι ένας ακέραιος αριθμός και το κύριο νήμα έχει id ίσο με μηδέν. Μετά την εκτέλεση του παραλληλοποιημένου κώδικα, τα νήματα συνδέονται πίσω στο κύριο νήμα, το οποίο συνεχίζει μέχρι το τέλος του προγράμματος.

Από προεπιλογή, κάθε νήμα εκτελεί ανεξάρτητα το παραλληλοποιημένο τμήμα του κώδικα. Μας δίνεται η δυνατότητα να διαιρέσουμε μια εργασία μεταξύ των νημάτων έτσι ώστε κάθε νήμα να εκτελέσει το μέρος του κώδικα που του έχει ανατεθεί. Τόσο η

παραλληλοποίηση των εργασιών όσο και των δεδομένων μπορούν να επιτευχθούν με τη χρήση του OpenMP με αυτό το τρόπο.

Το περιβάλλον εκτέλεσης κατανέμει τα νήματα σε επεξεργαστές ανάλογα με τη χρήση, το φορτίο του μηχανήματος και άλλους παράγοντες. Το περιβάλλον εκτέλεσης μπορεί να αντιστοιχίσει τον αριθμό των νημάτων με βάση μεταβλητές περιβάλλοντος ή μπορεί γίνει μέσα από τον κώδικα χρησιμοποιώντας ειδικές συναρτήσεις. Οι συναρτήσεις OpenMP περιλαμβάνονται σε ένα αρχείο κεφαλίδας με την ονομασία omp.h στη C/C++.



Σχήμα 1: Απεικόνιση πολυνηματισμού,όπου το master thread δημιουργεί άλλα νήματα που εκτελούν τμήματα κώδικα παράλληλα.

1.4 Εισαγωγή στον Aris Hypercomputer (HPC)

Οι υπερυπολογιστές είναι υπολογιστικά συστήματα που αξιοποιούνται σε επιστημονικές εφαρμογές οι οποίες απαιτούν την εκτέλεση πολλών εκατομμυρίων μαθηματικών πράξεων ή την επεξεργασία μεγάλου όγκου δεδομένων. Λόγω αυτών των απαιτήσεων τέτοιου είδους προβλήματα είτε θα χρειάζονταν απαγορευτικά μεγάλο χρόνο για να ολοκληρωθούν σε έναν απλό υπολογιστή γραφείου είτε λόγω περιορισμένων πόρων (π.χ. κεντρική μνήμη, αποθηκευτικός χώρος) δεν είναι εφικτό να πραγματοποιηθούν καθόλου. Οι υπερυπολογιστές ξεπερνούν τους περιορισμούς αυτούς χρησιμοποιώντας εξειδικευμένο υλικό τελευταίας τεχνολογίας κάθε χρονική στιγμή, εκμεταλλευόμενοι παράλληλα την υπολογιστική ισχύ από πολλαπλές υπολογιστικές μονάδες. Ένας υπερυπολογιστής σήμερα είναι στην πραγματικότητα ένα σύστημα από εκατοντάδες ή και χιλιάδες υπολογιστές (στους οποίους αναφερόμαστε συνήθως ως κόμβους) που επικοινωνούν μεταξύ τους χρησιμοποιώντας ένα πολύ γρήγορο δίκτυο και οι οποίοι συνεργατικά μπορούν να επιλύουν προβλήματα με μεγάλη ταχύτητα.

Ένας υπερυπολογιστής είναι ένα πανίσχυρο εργαλείο πραγματοποίησης έρευνας. Σήμερα οι υπερυπολογιστές χρησιμοποιούνται για να επιλύσουν μερικά από τα σημαντικότερα προβλήματα της ανθρωπότητας, όπως η προέλευση του σύμπαντος, η ανακάλυψη νέων φαρμάκων, οι έρευνες για την κλιματική αλλαγή και πολλά άλλα. Στην Ελλάδα έχουμε τον υπερυπολογιστή Aris [7]. Ο υπερυπολογιστής Aris έχει εγκατασταθεί και λειτουργεί από το ΕΔΕΤ (Ελληνικό Δίκτυο Έρευνας και Τεχνολογίας) στην Αθήνα. Το όνομα Aris σημαίνει Advanced Research Information System. Πρόκειται για ένα σύστημα με στόχο να στηρίξει προγράμματα ερευνητικού σκοπού γραμμένα για συστήματα παράλληλης αρχιτεκτονικής.

Το σύστημα έχει μια θεωρητική μέγιστη απόδοση (Rpeak) των 444 TFLOPS και προσφέρει πολλαπλές δυνατότητες επεξεργασίας δεδομένων. Κατετάγη #468 στη λίστα των TOP500 πιο ισχυρών συστημάτων στον κόσμο, όταν εγκαταστάθηκε (Ιούνιος 2015).

Στην παρακάτω εικόνα (βλέπε Εικόνα 1), μπορείτε να δείτε συνοπτικά τα τεχνικά χαρακτηριστικά του Aris:

Υλοποίηση του αλγορίθμου LMSOR με τεχνικές παράλληλου προγραμματισμού (MPI και OpenMP) και μετρήσεις στον υπερυπολογιστή Aris .

ARIS compute nodes	
Architecture	x86-64
System	IBM NeXtScale nx360 M4
Total number of nodes	426
Total number of cores	8520
Total amount of RAM [TByte]	27
Total Linpack Performance [TFlop/s]	180
Components	
Processor Type	Ivy Bridge - Intel Xeon E5-2680v2
Nominal Frequency [GHz]	2.8
Processors per Node	2
Cores per Processor	10
Cores per Node	20
Hyperthreading	OFF
Memory	
Memory per Node [GByte]	64 (usable 57)
Interconnect	
Technology	Infiniband FDR
Topology	Fat tree
Bandwidth [Gb/s]	56
Storage	
Type	IBM GPFS
Size [PByte]	1
Bandwidth [GB/s]	6
System Software	
Operating system	RedHat Linux 6.6
Batch system	SLURM
System Management	xCat IBM
Monitoring	Nagios, Ganglia

Login nodes	
Number of Nodes	2
Processor Type	Intel(R) Xeon(R) CPU E5-2640 v2
Nominal Frequency [GHz]	2.00GHz
Processors per Node	2
Cores per Processor	16
Threads per Processor	32
Hyperthreading	ON
Memory per Node [GByte]	128

Εικόνα 1: Τεχνικά χαρακτηριστικά του υπερυπολογιστή Aris.

Ο ARIS συνδυάζει 4 διαφορετικές αρχιτεκτονικές διαμοιρασμένες σε αντίστοιχες “νησίδες κόμβων”. Αναλυτικά, η υποδομή αποτελείται από:

- Μία νησίδα η οποία διαθέτει 426 υπολογιστικούς κόμβους (**thin nodes**). Κάθε κόμβος διαθέτει δύο επεξεργαστές και κάθε επεξεργαστής περιέχει 10 επεξεργαστικούς πυρήνες προσφέροντας έτσι συνολικά 8.520 πυρήνες (CPU cores). Οι κόμβοι αυτοί είναι κατάλληλοι για εφαρμογές υψηλής παραλληλίας που μπορούν να σπάσουν τα δεδομένα τους σε πολλά μικρά κομμάτια πριν τα επεξεργαστούν.
- Μια νησίδα κόμβων μεγάλης μνήμης (**fat nodes**) που αποτελείται από 44 κόμβους. Κάθε κόμβος προσφέρει 4 επεξεργαστές, 40 πυρήνες και 512 GB κεντρικής μνήμης ανά κόμβο. Οι κόμβοι αυτοί είναι κατάλληλοι για εφαρμογές που χρειάζονται πολύ μεγάλη κεντρική μνήμη και όχι τόσο για υψηλή κλιμάκωση.
- Μια νησίδα κόμβων επιταχυντών GPU (**gpu nodes**) που αποτελείται από 44 κόμβους. Κάθε κόμβος περιέχει 2 επεξεργαστές με 10 πυρήνες ανά επεξεργαστή, 64 GB μνήμης και 2 κάρτες γραφικών GPU NVidia K40. Οι κόμβοι αυτοί είναι κατάλληλοι για εφαρμογές που υλοποιούν υπολογιστικές πράξεις που μπορούν να αξιοποιήσουν τις κάρτες γραφικών ως συνεπεξεργαστές για επιτάχυνση των υπολογισμών.
- Μια νησίδα κόμβων επιταχυντών Xeon Phi (**phi nodes**) που αποτελείται από 18 κόμβους, καθένας εκ των οποίων περιέχει 2 επεξεργαστές με 10 πυρήνες, 64 GB μνήμης και 2 συνεπεξεργαστές Intel Xeon Phi 7120P. Είναι κατάλληλη για παράλληλες εφαρμογές που αξιοποιούν την τεχνολογία συνεπεξεργαστών της Intel Xeon Phi.

Μια νησίδα κόμβων είναι μια ομάδα υπολογιστικών μονάδων οι οποίες έχουν όμοια αρχιτεκτονική, μοιράζονται το ίδιο δίκτυο επικοινωνίας και έχουν πρόσβαση σε κοινό σύστημα αρχείων.

Ο ARIS χρησιμοποιεί το λειτουργικό σύστημα Linux στις διανομές (distributions) RedHat 6.9 και Centos 6.9. Η αλληλεπίδραση με το σύστημα γίνεται μέσω γραμμής εντολών και όχι από γραφικό περιβάλλον (όπως τυπικά γίνεται π.χ. σε ένα σύστημα MS Windows).

Η γραμμή εντολών Linux είναι ένα ισχυρό εργαλείο, με σημαντικά περισσότερες δυνατότητες έναντι του γραφικού περιβάλλοντος. Για τη χρήση του συστήματος είναι απαραίτητη η γνώση των βασικότερων εντολών που θα χρησιμοποιηθούν κατά τη σύνδεση, τη μεταφορά των αρχείων, τη μεταγλώττιση εφαρμογών και την εκτέλεση εργασιών στο σύστημα.
Ο μόνος τρόπος σύνδεσης στο σύστημα γίνεται με χρήση του πρωτοκόλλου ssh (secure shell). Το ssh επιτρέπει σε ένα χρήστη να συνδεθεί μέσω Internet από τον δικό του υπολογιστή και να χρησιμοποιήσει έναν απομακρυσμένο υπολογιστή από τη γραμμή εντολής, ακριβώς όπως θα δούλευε στο δικό του τοπικό σύστημα. Η σύνδεση στο σύστημα δε γίνεται με κωδικό πρόσβασης (password) αλλά χρησιμοποιείται τεχνολογία δημοσίου κλειδιού (public key cryptography). Κάθε χρήστης έχει ένα μοναδικό κλειδί το οποίο το ssh το χρησιμοποιεί μαζί με το όνομα χρήστη, αντί κωδικού για τη σύνδεση.

Η πρόσβαση γίνεται από τον υπολογιστή του χρήστη στους κόμβους διασύνδεσης (login nodes) του ARIS. Οι κόμβοι διασύνδεσης είναι οι μόνοι κόμβοι του συστήματος ARIS οι οποίοι έχουν σύνδεση με το Internet. Από εκεί αποστέλλονται οι εργασίες για εκτέλεση σε μία από τις νησίδες υπολογιστικών κόμβων.

Κάθε αρχείο που θα χρησιμοποιηθεί στο ARIS, είτε αυτό είναι η ίδια η εφαρμογή είτε τα αρχεία δεδομένων της εφαρμογής, πρέπει να μεταφερθεί από τον υπολογιστή του χρήστη στους κόμβους διασύνδεσης χρησιμοποιώντας τα αντίστοιχα πρωτόκολλα για τη μεταφορά αρχείων scp ή sftp.

Ο ARIS διαθέτει όλα τα απαραίτητα εργαλεία για την ανάπτυξη εφαρμογών όπως μεταγλωττιστές, λογισμικό αποσφαλμάτωσης (debugging) και λογισμικό ανάλυσης απόδοσης (profiling) που βοηθούν στη βελτιστοποίηση των εφαρμογών για το σύστημα.

Για τη μεταγλώττιση πηγαίου κώδικα σε γλώσσες C/C++ και Fortran παρέχονται τρεις διαφορετικές σουίτες μεταγλωττιστών:

- GNU (gcc, g++, gfortran)
- Intel® Parallel Studio XE Cluster Edition for Linux
- PGI Cluster Development Kit for Linux

Για τις παράλληλες βιβλιοθήκες MPI διατίθενται οι αντίστοιχες εκδόσεις λογισμικού OpenMPI και IntelMPI. Για τη μεταγλώττιση κώδικα που θα εκτελείται σε κάρτες γραφικών διατίθεται η έκδοση μεταγλωττιστή της NVIDIA, για το προγραμματιστικό μοντέλο CUDA (Compute Unified Device Architecture) nvcc.

Ο ARIS όπως όλα τα μεγάλα συστήματα του είδους, ακολουθεί τη λογική της εκτέλεσης των εφαρμογών σε ομάδες εργασιών (batch job execution). Για να εκτελεστεί μία εφαρμογή πρέπει πρώτα να περιγραφεί ως εργασία και να αποσταλεί σε ένα σύστημα χρονοπρογραμματισμού (scheduler) που αναλαμβάνει να τη δρομολογήσει για εκτέλεση στους υπολογιστικούς κόμβους. Ο ARIS χρησιμοποιεί το πρόγραμμα SLURM (Simple Linux Utility for Resource Management) για τη διανομή εργασιών στον υπερυπολογιστή. Το SLURM έχει τρεις βασικές λειτουργίες. Πρώτον, διαθέτει αποκλειστική ή μη αποκλειστική πρόσβαση στους πόρους (υπολογιστικούς κόμβους) στους χρήστες για ορισμένο χρονικό διάστημα, ώστε να μπορούν να εκτελούν εργασία. Δεύτερον, παρέχει ένα πλαίσιο για την εκκίνηση, εκτέλεση και παρακολούθηση εργασιών. Τέλος, εξασφαλίζει τον διαμοιρασμό των υπολογιστικών κόμβων σε πολλούς χρήστες ταυτόχρονα αποφεύγοντας προβλήματα ανταγωνισμού μεταξύ τους. Αυτό επιτυγχάνεται με μια ουρά εργασιών για κάθε νησίδα κόμβων, στην οποία εφαρμόζεται

η δίκαιη και ισότιμη προτεραιότητα για κάθε χρήστη ανάλογα με τους πόρους που απαιτεί από την εφαρμογή.

Ο χρήστης μπορεί να δρομολογήσει εργασίες από τον κόμβο διασύνδεσης στο σύστημα μέσω κατάλληλης εντολής του SLURM (sbatch). Κάθε εργασία για να γίνει αποδεκτή και να μπορέσει να δρομολογηθεί πρέπει να καθορίζει κάποιες ελάχιστες προδιαγραφές: την νησίδα που προτιμάται για την εκτέλεση, το πλήθος των κόμβων που απαιτούνται, το πλήθος πυρήνων ανά κόμβο, το μέγιστο χρόνο εκτέλεσης και την εντολή προγράμματος που θα εκτελεστεί. Οι προδιαγραφές αυτές πρέπει να οριστούν σε ένα αρχείο κειμένου που ονομάζεται σενάριο εργασίας (batch script).

Μία εργασία όταν δρομολογηθεί μπορεί να μην εκτελεστεί αμέσως αν δεν είναι διαθέσιμοι οι απαραίτητοι πόροι. Σε αυτή την περίπτωση μπαίνει στην κατάλληλη ουρά προτεραιότητας και περιμένει έως ότου ελευθερωθούν οι πόροι που χρειάζεται (π.χ. ο αριθμός κόμβων που απαιτείται για την εκτέλεσή της) οπότε το SLURM θα αναλάβει αυτόματα να τις ξεκινήσει. Ο χρόνος αναμονής εξαρτάται από τις απαιτήσεις που έχει προδιαγράψει η εργασία και κυμαίνεται από μερικά λεπτά έως αρκετές ώρες.

Όπως φαίνεται από τα παραπάνω ο ARIS είναι κατάλληλος για εφαρμογές οι οποίες είναι παράλληλες, εκτελούνται αυτόνομα στο πλαίσιο μιας εργασίας, με συγκεκριμένο χρόνο εκτέλεσης και δεν επικοινωνούν με τον χρήστη ή με άλλες εφαρμογές. Είναι σημαντικό να τονιστεί ότι δεν υπάρχει δυνατότητα επικοινωνίας της εφαρμογής μέσω Internet με εξωτερικούς στο σύστημα χρήστες ή η παροχή της σε μορφή υπηρεσίας ιστού (web service) η οποία εκτελείται στο διηνεκές (persistant services).

Τα αποτελέσματα των εργασιών αποθηκεύονται στο κοινό σύστημα αρχείων GPFS το οποίο είναι προσβάσιμο από όλους τους κόμβους των νησίδων του ARIS και τον κόμβο διασύνδεσης. Μια απλοϊκή προσέγγιση είναι να θεωρήσουμε το GPFS ως ένα πολύ μεγάλο σκληρό δίσκο στον οποίο αποθηκεύονται όλα τα δεδομένα και οι εφαρμογές που βρίσκονται στο ARIS.

Ειδικότερα το σύστημα αρχείων του ARIS προσφέρει δύο αποθηκευτικούς χώρους ανάλογα με τα δεδομένα που αποθηκεύονται και την χρήση τους:

- Το $HOME όπου δίνεται έμφαση στην αξιοπιστία έναντι της απόδοσης. Κατάλληλο για αποθήκευση αρχείων πηγαίου κώδικα, τη μεταγλώττιση εφαρμογών και προσωρινή αποθήκευση αρχείων δεδομένων.
- Το $WORKDIR με έμφαση στην απόδοση (ταχύτητα). Κατάλληλο για αποθήκευση μεγάλων αρχείων, για την εκτέλεση εργασιών και εξαγωγή αποτελεσμάτων.

Οι χρήστες περιορίζονται από τα όρια χρήσης του συστήματος αρχείων. Τα όρια ορίζονται ανάλογα με τις απαιτήσεις του έργου για πόρους αποθήκευσης.

1.5 Εισαγωγή στην τοπική μέθοδο SOR

Η επαναληπτική τοπική μέθοδος της Διαδοχικής Υπερομαλοποίησης (Successive Over-Relaxation-SOR) έχει μορφή (1)

$$u_{ij}^{(n+1)} = (1-\omega)\,u_{ij}^{(n)} + \omega(l_{ij}u_{i-1j}^{(n+1)} + r_{ij}u_{ij+1}^{(n)} + t_{ij}\,u_{ij+1}^{(n)} + b_{ij}u_{ij-1}^{(n+1)})$$

και είναι πολύ σημαντική και χρήσιμη για την επίλυση μεγάλων γραμμικών συστημάτων. Η πρωταρχική μορφή όμως της SOR είναι σειριακή και για αυτό το λόγο έχουν γίνει ευρείες μελέτες παράλληλων εκδόσεων της μεθόδου SOR με χρήση διάφορων τεχνικών. Χαρακτηριστικό παράδειγμα αποτελεί η τοπική μέθοδος SOR (Local SOR) που προτάθηκε από του Ehrlich, Botta και Veldman σε μία προσπάθεια να επιταχυνθεί ο ρυθμός σύγκλισης της SOR. Αυτό που κάνει την μέθοδο αυτή να ξεχωρίζει είναι ο διαφορετικός παράγοντας ομαλοποίησης ωij. Η τοπική μέθοδος SOR σε συνδυασμό με την red black αναδιάταξη, είναι κατάλληλη για παράλληλη υλοποίηση πάνω σε διάταξη πλεγματικά συνδεόμενων επεξεργαστών.

Η πτυχιακή αυτή εργασία, καθώς και η προϋπάρχουσα υλοποίηση που αποτέλεσε την αφετηρία της μελέτης μας, έχει ως στόχο την επίλυση της Εξίσωσης Διάχυσης Θερμότητας 2ας τάξης (2)

$$\Delta u - f(x,y)\frac{dy}{dx} - g(x,y)\frac{dy}{dx} = 0$$

σε ένα χώρο Ω={(x,y) | 0<=x<=1 , 0<=y<=1 }, όπου η u = u(x,y) ορίζεται στο όριο dΩ. Η διακριτή μορφή της εξίσωσης σε ένα ορθογώνιο πλέγμα M1 x M2 = N αγνώστων μέσα στο χώρο Ω είναι (3)

$$u_{ij} = l_y u_{i-1j} + r_{ij}u_{i+1j} + t_{ij}u_{ij+1} + b_{ij}u_{ij-1}$$

με i = 1, 2, ..., M1 και j = 1, 2, ..., M2 όπου

$$l_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1+\frac{1}{2}hf_{ij}\right)$$

$$r_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1-\frac{1}{2}hf_{ij}\right)$$

$$t_{ij} = \frac{k^2}{2(\mathrm{k}^2+\mathrm{h}^2)}\left(1-\frac{1}{2}kg_{ij}\right)$$

$$b_{ij} = \frac{k^2}{2(\mathrm{k}^2+\mathrm{h}^2)}\left(1+\frac{1}{2}kg_{ij}\right)$$

με h = 1/(M1+1), k = (M2+1) και

$$f_{ij} = f(\mathrm{ih}, \mathrm{jk}) \qquad\qquad g_{ij} = g(\mathrm{ih}, \mathrm{jk})$$

Παρατηρούμε ότι για μια συγκεκριμένη διάταξη των σημείων του πλέγματος, η διακριτή εξίσωση δίνει μεγάλο, αραιό, γραμμικό σύστημα εξισώσεων τάξης N της μορφής Au = b.

Η προαναφερθείσα τοπική μέθοδος SOR για τη διακριτοποίηση 5 σημείων της (3) δίνεται από την (1). Έχουμε τη δυνατότητα να επιλέξουμε να καλούμε ένα σημείο του πλέγματος (i, j), που να είναι κόκκινο (red) όταν το άθροισμα i+j είναι άρτιο και μαύρο (black) όταν το άθροισμα i+j είναι περιττό. Έστω τα δύο διαφορετικά σύνολα παραμέτρων ω1ij και ω2ij για τα κόκκινα και τα μαύρα σημεία του πλέγματος αντίστοιχα. Συνεπώς αν Ω = diag(ω1, ω2,....., ωN), όπου ωi = 1, 2, ….N πραγματικοί αριθμοί η (1) ισοδυναμεί για i+j άρτιο με (4)

$$u_{ij}^{(n+1)} = (1-\omega_{1ij})\,\mathrm{u}_{ij}^{(n)} + \omega_{1ij} J_{ij} u_{ij}^{(n)}$$

και για i+j περιττό με

$$u_{ij}^{(n+1)} = (1-\omega_{2ij})\,\mathrm{u}_{ij}^{(n)} + \omega_{2ij} J_{ij} u_{ij}^{(n)}$$

όπου

$$J_{ij}u_{ij}^{(n)} = l_{ij}u_{i-1\,j}^{(\mathrm{n})} + r_{ij}u_{i+1\,j}^{(\mathrm{n})} + t_{ij}u_{ij+1}^{(\mathrm{n})} + b_{ij}u_{ij-1}^{(\mathrm{n})}$$

και Jij είναι ο τοπικός συντελεστής Jacobi. Οι παράμετροι ω1ij, ω2ij καλούνται τοπικοί παράμετροι ομαλοποίησης και η (4) καλείται τοπική τροποποιημένη μέθοδος SOR (Local Modified SOR – LMSOR).

1.6 Ορισμοί Speed Up και Efficiency

Ως λόγος επιτάχυνσης (speed up) S ορίζεται η αναλογία του χρόνου εκτέλεσης ενός προγράμματος σε έναν επεξεργαστή με το χρόνο εκτέλεσης σε N επεξεργαστές. Η βέλτιστη τιμή Speed Up σε N επεξεργαστές είναι S(N) = N, όπου N ο αριθμός των επεξεργαστών.

Ως αποδοτικότητα (efficiency) E ορίζουμε το μέτρο της αποτελεσματικότητας της χρήσης του επεξεργαστή. Η μεγαλύτερη τιμή της αποδοτικότητας E(N) μπορεί να είναι 1 όταν έχουμε γραμμική επιτάχυνση. Αν έχουμε απόδοση 1, τότε λέμε πως το πρόγραμμα μας είναι αποδοτικό.

Δηλαδή,

$$S(\mathrm{n}) = \frac{t_s}{t_n} \qquad\qquad E = \frac{S(\mathrm{n})}{n}$$

Στα πειράματα μας, χρησιμοποιήσαμε τις παραπάνω έννοιες του λόγου επιτάχυνσης (speed up) και της αποδοτικότητας (efficiency) ως δείκτες μέτρησης της απόδοσης των αποτελεσμάτων του υπερυπολογιστή Aris.

Με τον όρο κλιμάκωση αναφερόμαστε στη δυνατότητα ενός παράλληλου αλγορίθμου με αποδοτικότητα E όταν χρησιμοποιεί x επεξεργαστές, να λύνει ένα μεγαλύτερο πρόβλημα με την ίδια αποδοτικότητα σε x+1 επεξεργαστές.

2. ΥΛΟΠΟΙΗΣΗ

2.1 Υλοποίηση MPI

2.1.1 Υλοποίηση MPI με επικοινωνία των μεγάλων πινάκων & reordering

Το πρώτο μας βήμα, ήταν η υλοποίηση παραλληλοποίησης MPI με την εξής λογική: οι μεγάλοι πίνακες του αλγορίθμου της LMSOR, χωρίζονται σε τόσα μπλοκ, όσες είναι και οι διεργασίες (processes) που έχουν δωθεί ως παράμετρος κατά το τρέξιμο του προγράμματος (βλ. Εικόνα 2). Αναθέτουμε σε κάθε διεργασία την δημιουργία των δικών της μπλοκ (ένα για κάθε αρχικό πίνακα) (βλ.Εικόνα 3 και 4), τα οποία αρχικοποιούνται με τυχαίους αριθμούς που παράγονται από μια συνάρτηση γεννήτρια (βλ.Εικόνα 5). Σε αυτή την υλοποίηση οι γειτονικές διεργασίες επικοινωνούν μεταξύ τους, αποστέλωντας τα σύνορα των μπλοκ τους στα μπλοκ των γειτόνων τους, που αντιστοιχούν στον ίδιο πίνακα. Για να επιτευχθεί αυτή η επικοινωνία σε κάθε μπλοκ της εκάστοτε διεργασίας προστίθενται δύο επιπλέον γραμμές και στήλες οι οποίες επιτελούν τον ρόλο του αποθηκευτικού χώρου για τα στοιχεία που λαμβάνονται από τις γειτονικές διεργασίες.

Στη συνέχεια, πραγματοποιείται red black αναδιάταξη (red black reordering), η οποία χωρίζει τα μπλοκ των αρχικών πινάκων σε δύο red & black υποπίνακες, πάνω στους οποίους εκτελείται ο αλγόριθμος LMSOR. Αυτό που κάνει η red black αναδιάταξη είναι να τοποθετεί τα στοιχεία με περιττό άθροισμα συντεταγμένων στον έναν πίνακα και αυτά με άρτιο στον άλλο. Η αναδιάταξη αυτή βοηθάει στην παραλληλοποίηση του προγράμματος, καθώς χωρίζει τα στοιχεία των αρχικών πινάκων με τέτοιο τρόπο έτσι ώστε ο υπολογισμός της νέας τιμής των στοιχείων αυτών, με τον αλγόριθμο LMSOR, να μην εξαρτάται από στοιχεία του ίδιου πίνακα. Αυτό συμβαίνει γιατί ο αλγόριθμος χρησιμοποιεί για τον υπολογισμό του εκάστοτε στοιχείου τα 4 γειτονικά στοιχεία του στον αρχικό πίνακα τα οποία είναι το βόρειο, το νότιο, το ανατολικό και το δυτικό του. Αυτά τα στοιχεία μετά την red black αναδιάταξη, όπως περιγράφηκε παραπάνω, καταλήγουν πάντα στον άλλον πίνακα (αν πρόκειται για στοιχείο του red πίνακα τα 4 γειτονικά του θα βρίσκονται στον black πίνακα και αντίστοιχα τα 4 γειτονικά ενός στοιχείου του black πίνακα θα βρίσκονται στον red πίνακα). Με άλλα λόγια, τα στοιχεία ενός red πίνακα χρησιμοποιούν τα στοιχεία του αντίστοιχου black πίνακα για τον υπολογισμό της νέας τιμής του σε κάθε επανάληψη του αλγορίθμου. Αντίστοιχα, υπολογίζονται και οι νέες τιμές των στοιχείων των black πινάκων.

Αφού ολοκληρωθεί ο υπολογισμός των νέων τιμών των red black πινάκων, γίνεται μια αντίστροφη αναδιάταξη (backward reordering) ώστε να ξαναφτιαχτούν οι μεγάλοι πίνακες από τους αντίστοιχους red και black. Έπειτα, γίνεται η επικοινωνία μεταξύ των συνόρων των μεγάλων πινάκων ώστε να ενημερωθούν οι γείτονες του κάθε πίνακα για τις νέες τιμές των συνοριακών στοιχείων, τα οποία θα τους φανούν χρήσιμα στον επόμενο υπολογισμό τους. Σε κάθε επανάληψη, υπολογίζεται μια μεταβλητή sqrerror, η οποία είναι ενδεικτική του αν ο αλγόριθμος συγκλίνει ή αποκλίνει.

Όλη η παραπάνω διαδικασία είτε επαναλαμβάνεται για τον αριθμό των επαναλήψεων που έχουμε δώσει ως όρισμα είτε τερματίζει νωρίτερα αναλόγως αν το sqrerror έχει πάρει κάποια μη επιτρεπόμενη τιμή. Αυτές οι μη επιτρεπόμενες τιμές μας υποδεικνύουν ότι ο αλγόριθμος δεν έχει νόημα να συνεχίσει να εκτελείται είτε διότι αποκλίνει είτε διότι παρατηρείται ελάχιστη βελτίωση ανά επανάληψη.

Με βάση τον αρχικό δοσμένο κώδικα, μετά το πέρας των επαναλήψεων του αλγορίθμου

τα αποτελέσματα συλλέγονται και γράφονται σε ένα αρχείο. Για την εξοικονόμηση χρόνου, αφαιρέσαμε την λειτουργία αυτή και όλες οι μετρήσεις μας δεν την περιλαμβάνουν. Επίσης, με το τέλος του προγράμματος εμφανίζεται στον χρήστη ο συνολικός χρόνος τρεξίματος του προγράμματος και ο χρόνος εκτέλεσης των αναδιατάξεων (reordering).

2.1.2 Υλοποίηση MPI με επικοινωνία μεταξύ red black πινάκων χωρίς reordering

Αυτή η υλοποίηση ξεκινάει όπως η προηγούμενη, δηλαδή έχοντας αρχικά τους μεγάλους πίνακες που ανατίθενται σε διεργασίες, με κάθε διεργασία να χωρίζει το δικό της μπλοκ από κάθε αρχικό πίνακα σε δύο υποπίνακες red και black με τη μέθοδο red black αναδιάταξης. Η διαφορά αυτής της υλοποίησης με την προηγούμενη είναι ότι η επικοινωνία των διεργασιών γίνεται μεταξύ των red πινάκων και των black.

Αναλυτικότερα, η πρώτη red black αναδιάταξη γίνεται πριν ξεκινήσουν οι επαναλήψεις του αλγορίθμου LMSOR και άρα δεν ξαναγίνεται κάποια αναδιάταξη αφότου ξεκινήσουν αυτές οι επαναλήψεις. Στη συνέχεια, κατά τη διάρκεια των επαναλήψεων, επικοινωνούν οι red πίνακες των γειτονικών διεργασιών μεταξύ τους και αντίστοιχα γίνεται και με τους black πίνακες. Συγκεκριμένα, στην αρχή ξεκινάει η επικοινωνία των black πινάκων (MPI_Startall για τα Send & Receive) (βλ. Εικόνα 6). Στη συνέχεια, γίνεται ο υπολογισμός των εσωτερικών στοιχείων των red πινάκων τα οποία δεν χρειάζεται να περιμένουν να ολοκληρωθεί η αποστολή και λήψη των συνοριακών black στοιχείων. Με αυτό τον τρόπο, εκμεταλλευόμαστε αποδοτικά αυτό το ενδιάμεσο χρονικό διάστημα. Έπειτα, προχωρήσαμε σε 2 διαφορετικές προσεγγίσεις.

Στη πρώτη προσέγγιση, αφότου ολοκληρωθεί ο υπολογισμός των εσωτερικών red στοιχείων, καλέσαμε MPI_Waitall για τα Receive των συνοριακών black στοιχείων και αφότου ολοκληρώθηκε η λήψη όλων των απαραίτητων στοιχείων προχωρήσαμε στον υπολογισμό των συνοριακών red στοιχείων τα οποία θα χρειαστούν τα αντίστοιχα συνοριακά black στοιχεία (βλ. Εικόνα 7). Αυτή η προσέγγιση όμως, ειδικά σε μεγάλους πίνακες, δεν ήταν η πιο αποδοτική καθώς περιμένουμε για την λήψη των συνοριακών black στοιχείων και από τις 4 κατευθύνσεις πριν αρχίσουμε να υπολογίζουμε τα συνοριακά red στοιχεία, με αποτέλεσμα αν μία ή περισσότερες από τις λήψεις των συνοριακών black στοιχείων αργήσει περισσότερο από τις υπόλοιπες να την/τις περιμένουμε ενώ θα μπορούσαμε να εκμεταλλεύτουμε το γεγόνος ότι κάποια ή κάποιες από τις υπόλοιπες έχουν ήδη ολοκληρωθεί. Αυτό είναι ξεκάθαρο και στο αρχείο mpiP (profiler πρόγραμμα για MPI) της εκτέλεσης με 4 διεργασίες για πίνακα με μέγεθος προβλήματος NMAX = 16800 (βλ. Εικόνα 11, 12).

Το ζήτημα αυτό λύθηκε με την δεύτερη προσέγγιση μας στην οποία χρησιμοποιήσαμε την MPI_Test. Πιο συγκεκριμένα, ελέγχουμε συνεχώς για τυχόν ολόκληρωση της λήψης κάποιας από τις 4 κατευθύνσεις των black συνόρων και μόλις μία από αυτές ολοκληρωθεί προχωράμε στον υπολογισμό των αντίστοιχων συνοριακών red στοιχείων. Συνεχίζουμε με τον ίδιο τρόπο έως ότου ολοκληρωθεί η λήψη και ο υπολογισμός όλων των συνοριακών black και red στοιχείων αντίστοιχα. Με αυτόν τον τρόπο εκμεταλλευόμαστε τυχόν καθυστερήσεις στις λήψεις κάποιων από τις συνοριακές γραμμές ή στήλες των black πινάκων χρησιμοποιώντας αυτές οι οποίες έχουν ληφθεί πρώτες (βλ. Εικόνα 8). Αυτή η βελτίωση είναι φανερή ειδικά για πίνακα με NMAX = 16800 και εκτέλεση με 4 διεργασίες όπως φαίνεται και από τον Πίνακα 2 παρακάτω καθώς και από τις Εικόνες 13 και 14.

2.1.3 Επεξήγηση κώδικα και εικόνες

Στις δύο παραπάνω υλοποιήσεις χρησιμοποιήσαμε τις MPI_Send_init και MPI_Recv_init σε συνδιασμό με τις MPI_Start και MPI_Wait, έτσι ώστε να αρχικοποιούμε μία φορά μόνο τις επικοινωνίες, οι οποίες πρόκειται να επαναληφθούν πολλές φορές μέσα στο κώδικα και να τις ξεκινάμε απλά καλώντας τις MPI_Start και MPI_Wait. Με αυτό τον τρόπο, γλιτώνουμε αρκετό overhead (επιπλέον χρόνο) των απλών MPI συναρτήσεων MPI_Send, MPI_Recv, οι οποίες αν καλούνταν κάθε φορά, θα αρχικοποιούσαν ξανά τις παραμέτρους επικοινωνίας χωρίς να υπάρχει καμία αλλαγή σε αυτές.

Αξιοσημείωτο είναι επίσης ότι όλες οι συναρτήσεις μας έχουν οριστεί ως inline για να αποφευχθεί το overhead (επιπλέον χρόνος) των κλήσεων των συναρτήσεων. Αυτό που συμβαίνει πρακτικά είναι ότι ο μεταγλωττιστής αντικαθιστά την κλήση μιας συνάρτησης με τον κώδικα που εκτελείται όταν καλείται, αντιγράφωντας τον κώδικα αυτό στη διεύθυνση κάθε κλήσης συνάρτησης.

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

int block_size;
int root = sqrt(numtasks);
block_size = NMAX/root;
```

Εικόνα 2: Υπολογισμός του μεγέθους ενός μπλοκ,για το διαχωρισμό των αρχικών μεγάλων πινάκων σε μπλοκ.

```
class regular_array{
    double *av; //MAX MAX
    int size;
public:

    inline void set(int row, int col, double v){
        *(av+(row*(this->size))+col) = v;
    }
    inline double get(int row, int col){
        return *(av+(row*(this->size))+col);
    }
    inline double* _get_data(void){
        return &av[0];
    }
    inline double** _get_av(void){
        return &av;
    }
    inline void set_size(int size){
        this->size = size;
    }
    inline int get_size(){
        return this->size;
    }

};
```

Εικόνα 3: Ο ορισμός της κλάσης του κάθε πίνακα που αποτελεί ένα μπλοκ.

```
  arr_u.set_size(block_size);
  arr_w.set_size(block_size);

  double** arr_pointer = arr_u._get_av();
  *arr_pointer = new double[block_size*block_size];

  arr_pointer = arr_w._get_av();
  *arr_pointer = new double[block_size*block_size];


#ifdef _PRECALC_
    arr_ffh.set_size(block_size);
    arr_ggh.set_size(block_size);

    arr_pointer = arr_ffh._get_av();
    *arr_pointer = new double[block_size*block_size];

    arr_pointer = arr_ggh._get_av();
    *arr_pointer = new double[block_size*block_size];

#else

    arr_l.set_size(block_size);
    arr_r.set_size(block_size);
    arr_t.set_size(block_size);
    arr_b.set_size(block_size);

    arr_pointer = arr_l._get_av();
    *arr_pointer = new double[block_size*block_size];

    arr_pointer = arr_r._get_av();
    *arr_pointer = new double[block_size*block_size];

    arr_pointer = arr_t._get_av();
    *arr_pointer = new double[block_size*block_size];

    arr_pointer = arr_b._get_av();
    *arr_pointer = new double[block_size*block_size];



#endif
```

Εικόνα 4: Η δημιουργία των πινάκων του κάθε μπλοκ.

Υλοποίηση του αλγορίθμου LMSOR με τεχνικές παράλληλου προγραμματισμού (MPI και OpenMP) και μετρήσεις στον υπερυπολογιστή Aris .

```
    // Support omega values input from the environment
    long cImags=0, cReals=0;
    char *envOmega1 = getenv("LMSOR_OMEGA1"), *envOmega2 = getenv("LMSOR_OMEGA2");
    int gperiptosi;
    gperiptosi = build_omega(arxeio, re, cImags, cReals, min_w1, max_w1, min_w2, max_w2, epilogi, min_m_low, max_m_low, min_m_up, max_m_up,block_size-2,taskid,atoi(argv[7]));

    if( envOmega1 && envOmega2 ){
        cImags = cReals = 0;
        min_w1 = max_w1 = min_w2 = max_w2 = min_m_low = max_m_low = min_m_up = max_m_up = 0.0;
        double omega1 = atof(envOmega1), omega2 = atof(envOmega2);
        printf("\nNote: Read omega values from the environment (%.3f, %.3f)\n", omega1, omega2);
        min_w1 = max_w1 = omega1;
        min_w2 = max_w2 = omega2;
        for(i=0; i<=n+1; i++) {
          for(j=0; j<=n+1; j++) {

            arr_w.set(i, j, ( (i+j)%2 == 0 ) ? omega1 : omega2);

          }
        }
    }

#ifdef MODIFIED_SOR
    printf("R/B LMSOR method (5 point)\n");
#else
    printf("R/B LOCAL SOR method (5 point)\n");
#endif

//printf("!!!! n=%d\n", n);

#pragma omp parallel for private(j,x,y) num_threads(atoi(argv[7]))
    for(i=0; i<=n+1; i++) {
      x=i*h;
      for(j=0; j<=n+1; j++) {
        y=j*h;
        arr_u.set(i, j, initial_guess(x,y));

      }
    }
```

Εικόνα 5: Η συνάρτηση γεννήτρια τυχαίων αριθμών που αρχικοποιεί τους αρχικούς πίνακες.

```
MPI_Startall(4, &black_reqs[0][0]); //start the sendings of black borders
MPI_Startall(4, &black_reqs[1][0]); //start the receptions of black borders
```

Εικόνα 6: Η επικοινωνία των μαύρων πινάκων με MPI_Startall.

```
    MPI_Waitall(4, &black_reqs[1][0], MPI_STATUSES_IGNORE); //wait for the receptions of black borders to complete

    if(no_error == 1){  //calculate borders of red


        //calculate new values of first row of the red array
        calcSegment2<0>(1, columns+1, 1, 2, tsqrerror, block_size, taskid, metr);

        //calculate new values of last row of the red array
        calcSegment2<0>(1, columns+1, block_size-2, block_size-1, tsqrerror, block_size, taskid, metr);

        //calculate new values of first column of the red array
        calcSegment2<0>(1, 2, 1, block_size-1, tsqrerror, block_size, taskid, metr);

        //calculate new values of last column of the red array
        calcSegment2<0>(columns, columns+1, 1, block_size-1, tsqrerror, block_size, taskid, metr);

    }
```

Εικόνα 7: Λήψη των συνοριακών μαύρων στοιχείων και υπολογισμός των συνοριακών κόκκινων στοιχείων.

```
mpi_black_test_flags[0] = 0;
mpi_black_test_flags[1] = 0;
mpi_black_test_flags[2] = 0;
mpi_black_test_flags[3] = 0;


    while(!mpi_black_test_flags[0] || !mpi_black_test_flags[1] || !mpi_black_test_flags[2] || !mpi_black_test_flags[3]){
        for(int g = 0; g < 4; g++){
            if(!mpi_black_test_flags[g]){
                MPI_Test(&black_reqs[1][g], &mpi_black_test_flags[g], MPI_STATUSES_IGNORE);
                if(mpi_black_test_flags[g]==1 && no_error == 1){
                    if(g == 0){
                        //calculate new values of first row of the red array
                        calcSegment2<0>(1, columns+1, 1, 2, tsqrerror, block_size, taskid, metr);
                    }
                    else if(g == 1){
                        //calculate new values of last row of the red array
                        calcSegment2<0>(1, columns+1, block_size-2, block_size-1, tsqrerror, block_size, taskid, metr);
                    }
                    else if(g == 2){
                        //calculate new values of first column of the red array
                        calcSegment2<0>(1, 2, 1, block_size-1, tsqrerror, block_size, taskid, metr);
                    }
                    else{
                        //calculate new values of last column of the red array
                        calcSegment2<0>(columns, columns+1, 1, block_size-1, tsqrerror, block_size, taskid, metr);
                    }
                }
            }
        }
    }
```

Εικόνα 8: Χρήση της MPI_Test για τον υπολογισμό των κόκκινων συνοριακών στοιχείων δίχως την λήψη όλων μαύρων συνοριακών στοιχείων.

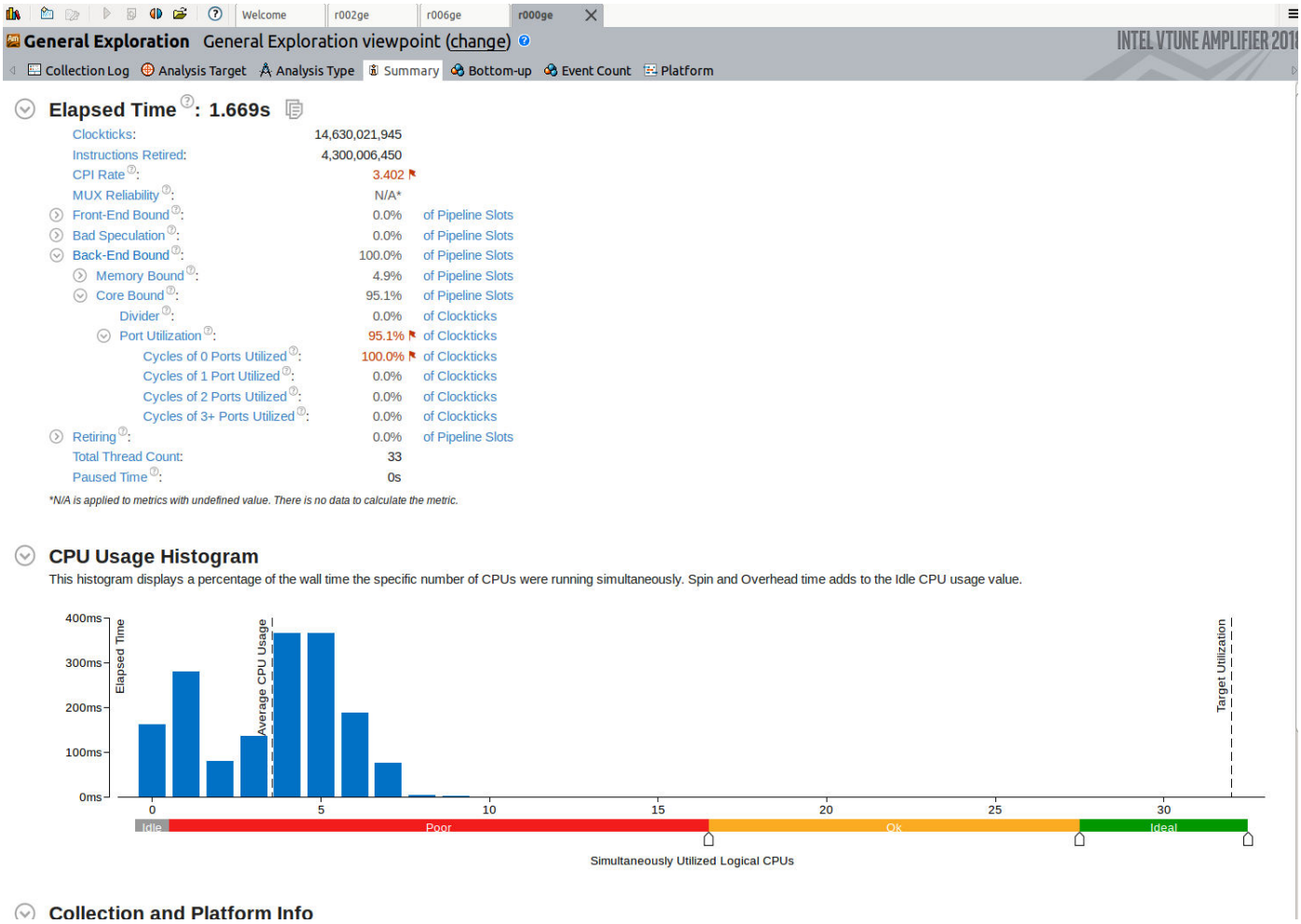
2.2 Υλοποίηση OpenMP

Αρχικά, χρησιμοποιήσαμε την ήδη υπάρχουσα υλοποίηση του OpenMP με την block partioning (διαχωρισμός σε μπλοκ) λογική, δηλαδή το κάθε thread να υπολογίζει το δικό του κομμάτι απο τον πίνακα (block) του process στο οποίο ανήκει. Το δοκιμάσαμε σε συνδιασμό και με την MPI υλοποίηση μας και παρατηρήσαμε ότι σε αντίθεση με τα processes, τα threads δεν κλιμακώνουν τόσο καλά, δηλαδή όσο τα αυξάνουμε δεν μας προσέφεραν ιδιαίτερη βελτίωση.

Έπειτα, αλλάξαμε την λογική της δοσμένης OpenMP υλοποίησης, αφαιρώντας την block partioning παραλληλοποίηση και κάνοντας παράλληλη την κεντρική for της βασικής συνάρτησης του αλγορίθμου ονόματι calcSegment. Αυτή άλλωστε είναι και η συνάρτηση που παίρνει το μεγαλύτερο κομμάτι του χρόνου της εκτέλεσης του προγράμματος. Δοκιμάσαμε τόσο απλή parallel for, όσο και βελτιώσεις της με dynamic scheduling, το οποίο βοηθάει ώστε να μην παραμένουν ανενεργά κάποια threads. Επίσης, έγινε χρήση του collapse clause του OpenMP, το οποίο προσφέρει βελτιστοποίηση όσον αφορά στην παραλληλοποίηση της διπλής for στην συνάρτηση calcSegment. Παρ' όλα αυτά, δεν είδαμε κάποια ιδιαίτερη βελτίωση ούτε στην απόδοση σε σύγκριση με την block partioning παραλληλοποίηση που μας είχε δωθεί, αλλά ούτε και στην κλιμάκωση σε σύγκριση με το MPI το οποίο παρουσιάζει πολύ καλύτερα αποτελέσματα.

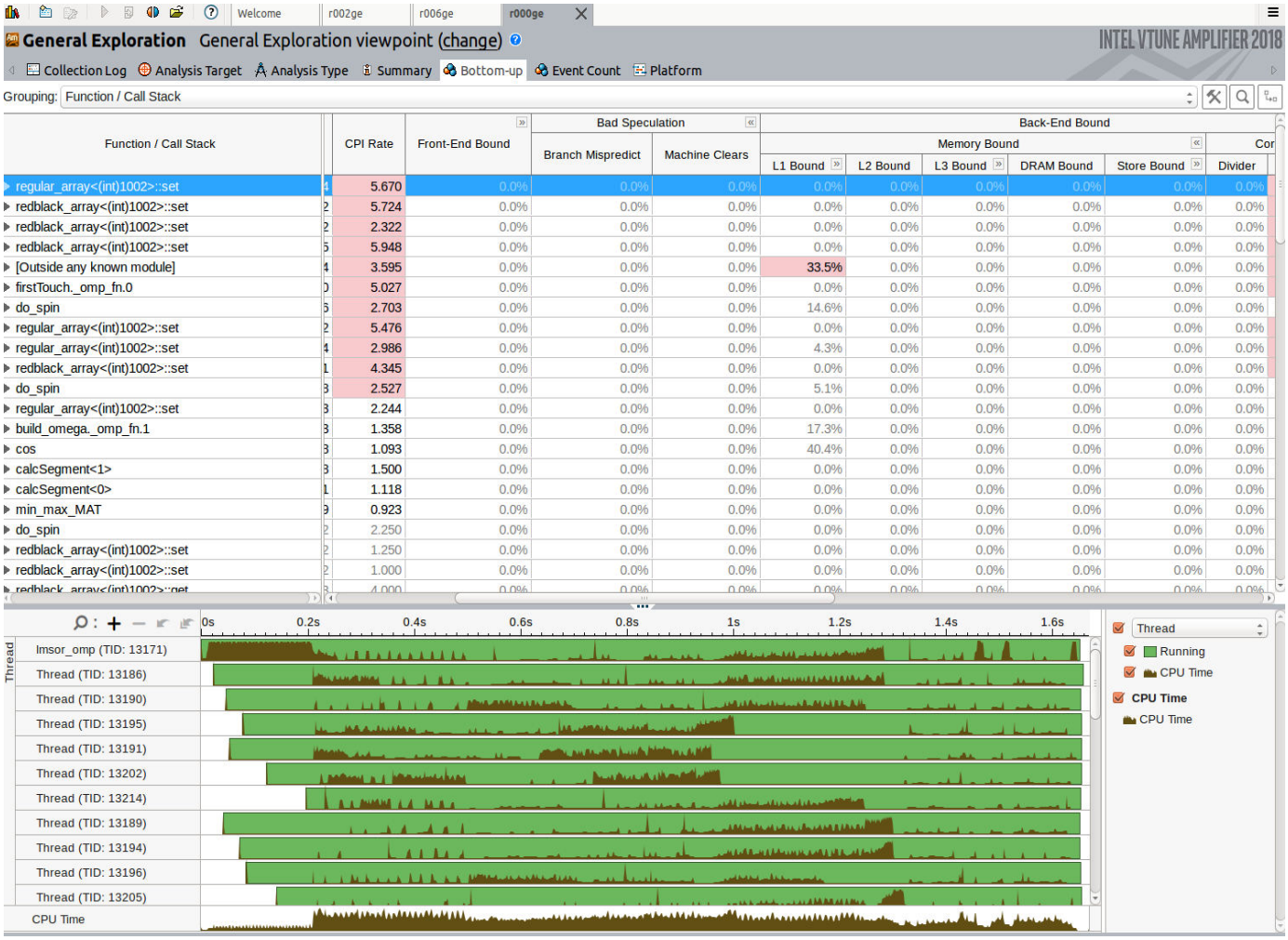
Αυτό που υποθέτουμε και αφήνουμε ως αντικείμενο μελλοντικής μελέτης, είναι ότι ευθύνεται το cache accessing για τα παραπάνω αποτελέσματα, καθώς είναι πολύ πιθανόν να παρατηρούνται ακόμα και φαινόμενα ψευδούς κοινής χρήσης (false sharing)

στην cache. Αυτό μπορούμε να το υποθέσουμε βασιζόμενοι και σε κάποια αποτελέσματα που πήραμε αναλύοντας το OpenMP πρόγραμμα με την βοήθεια του εργαλείου Intel Vtune, το οποίο προσφέρει αναλύση αποτελεσμάτων του OpenMP και διάφορους δείκτες οι οποίοι βοηθούν στον εντοπισμό των προβλημάτων του προγράμματος (βλ. Εικόνα 9 & Εικόνα 10).



Εικόνα 9: Το εργαλείο Vtune της Intel, μας υποδεικνύει τον φόρτο που επωμίστηκε η κάθε διεργασία, καθώς και το αν παρέμεινε ανενεργή (idle).

Υλοποίηση του αλγορίθμου LMSOR με τεχνικές παράλληλου προγραμματισμού (MPI και OpenMP) και μετρήσεις στον υπερυπολογιστή Aris .



Εικόνα 10: Το Vtune μας δείχνει αναλυτικά τους χρόνους εκτέλεσης όλων των συναρτήσεων.

3. ΜΕΤΡΗΣΕΙΣ ΣΤΟΝ ARIS

3.1 Μετρήσεις μόνο MPI

Αρχικά, έγιναν μετρήσεις στο πρόγραμμα μας κάνοντας χρήση μόνο MPI παραλληλοποίησης. Όλες οι μετρήσεις, έγιναν για τριακόσιες επαναλήψεις του επαναληπτικού αλγορίθμου LMSOR. Ως ορίσματα, κατά το τρέξιμο έχουν δωθεί σε όλες τις μετρήσεις μας η τιμή Re = 10.0 για τον συντελεστή Reynold καθώς και η τιμή 6 που χρησιμοποιείται ως συντελεστής για την δημιουργία των αρχικών πινάκων του πριγράμματος μας. Τα άλλα τρία ορίσματα έχουν τιμή 1 και αφορούν διαδοχικά την εκτύπωση του ω του αλγορίθμου LMSOR, την εκτύπωση λύσης και τον αριθμό των νημάτων που δημιουργούνται.

Οι συναρτήσεις MPI_Allreduce, που χρησιμοποιούνται για να ελεγχθεί εάν όλες οι διεργασίες έχουν φτάσει σε σύγκλιση οπότε και πρέπει να σταματήσει ο αλγόριθμος (έστω μία να μην έχει ο αλγόριθμος συνεχίζει κανονικά), έχουμε επιλέξει να γίνονται κάθε 100 επαναλήψεις του επαναληπτικού αλγορίθμου της LMSOR. Εάν τρέξουμε το πρόγραμμα μας, δίχως να εκτελούνται καθόλου οι MPI_Allreduce, τότε διαπιστώνουμε ότι υπάρχει μια μικρή βελτίωση στους τελικούς χρόνους.

Στα πειράματά μας μελετάμε την κλιμάκωση που επιτυγχάνεται χρησιμοποιώντας thin nodes στον Aris. Πήραμε μετρήσεις για διαφορετικά μεγέθη προβλήματος, δηλαδή για διαφορετικά μεγέθη πινάκων. Συγκεκριμένα για NMAX = 8400 και για NMAX = 16800, όπου ως NMAX ορίζεται το μέγεθος του συνολικού αρχικού πίνακα. Οι αριθμοί των διεργασιών που χρησιμοποιήσαμε δεν είναι τυχαίοι, καθώς το πρόγραμμα μας λειτουργεί παίρνοντας ως αριθμό διεργασιών έναν αριθμό που πρέπει να έχει ακέραιες ρίζες. Υπενθυμίζουμε ότι, όπως είχε αναφερθεί στην εισαγωγή που κάναμε για τον υπερυπολογιστή Aris, κάθε thin node στον Aris “χωράει” μέχρι και 20 διεργασίες. Οπότε, αναλόγως με τον αριθμό των διεργασιών που επιλέξαμε διαμορφώσαμε και τον αριθμό των nodes που χρησιμοποιήσαμε αντίστοιχα. Μέλημά μας ήταν να “γεμίζει” όσο το δυνατόν περισσότερο ο κάθε κόμβος που χρησημοποιούμε χωρίς να είναι πάντα αυτό εφικτό λόγω του ότι ο αριθμός των διεργασιών πρέπει να μοιραστεί ισόποσα σε κάθε κόμβο.

Παρακάτω παρατίθενται τα αποτελέσματα των μετρήσεων μας στον υπερυπολογιστή Aris.

Πίνακας 1: Μετρήσεις MPI για NMAX = 8400

Procceses	Node Type	Number of Nodes	Time(seconds)	Speedup	Efficiency
1	thin	1	206	-	-
4	thin	1	52,6	3,9	0,97
9	thin	1	26,2	7,8	0,87
16	thin	1	15	13,7	0,85
36	thin	2	7,1	29	0,8
64	thin	4	4,5	45,7	0,71

100	thin	5	3,4	60,6	0,6
144	thin	8	1,9	108	0,75
196	thin	14	1,15	179	0,91
256	thin	16	0,9	228	0,89

Πίνακας 2: Μετρήσεις MPI για NMAX = 16800

Procceses	Node Type	Number of Nodes	Time(seconds)	Speedup	Efficiency
1	thin	1	830	-	-
4 (χωρίς MPI_Test())	thin	1	335	2,4	0,61
4 (με MPI_Test())	thin	1	284	2,8	0,72
9	thin	1	96,5	8,5	0,94
16	thin	1	56	14,6	0,91
36	thin	2	27,4	30	0,83
64	thin	4	18,6	44	0,68
100	thin	5	12,2	67	0,67
144	thin	8	7,8	105,1	0,73
196	thin	14	6,6	124,2	0,63
256	thin	16	6,14	133,5	0,52
400	thin	20	4	205	0,51
576	thin	32	2,3	356	0,61
784	thin	49	1,5	546	0,69
900	thin	45	1,41	581	0,64
1600	thin	80	0,7	1171	0,73

3.2 Μετρήσεις υβριδικού MPI & OpenMP

Σε αυτή την ενότητα παραθέτουμε σε πίνακες τις μετρήσεις που έγιναν στο πρόγραμμα μας χρησιμοποιώντας συνδυασμό διεργασιών (MPI) και νημάτων (OpenMP). Όπως αναφέραμε στην ενότητα 2.2, η μελέτη των παρακάτω μετρήσεων μας οδήγησε στο συμπέρασμα ότι η χρήση νημάτων δεν παρουσιάζει καλή κλιμάκωση και επιδέχεται βελτίωση. Όλες οι παρακάτω μετρήσεις έγιναν για 100 επαναλήψεις του αλγορίθμου LMSOR.

Πίνακας 3: Μετρήσεις υβριδικού για NMAX = 2100

Processes	Threads	Node type	Time (seconds)
16	1	thin	3,2
4	4	thin	3,6
1	16	thin	8,6
4	36	thin	3,8
1	36	thin	8,6
4	9	fat	7,3
4	36	fat	9,3
9	4	fat	6
36	1	fat	1,75
1	36	fat	14,8

Πίνακας 4: Μετρήσεις υβριδικού για NMAX = 4200

Processes	Threads	Node type	Time (seconds)
4	4	thin	66
1	16	thin	162
16	1	thin	30
1	36	thin	169
4	36	thin	74
4	9	fat	7,4
4	36	fat	9,4
9	4	fat	5,3
36	1	fat	1,4
1	36	fat	14

Πίνακας 5: Μετρήσεις υβριδικού για NMAX = 8400

Processes	Threads	Node type	Time (seconds)
4	4	thin	250
4	36	thin	265
16	1	thin	182
1	16	thin	640
4	9	fat	24
4	36	fat	25
9	4	fat	24
1	36	fat	56

3.3 Μετρήσεις με mpiP

Σε όλες τις μετρήσεις που πραγματοποιήθηκαν παραπάνω και περιλαμβάνουν διεργασίες MPI χρησιμοποιήθηκε το εργαλείο mpiP, που παρουσιάσαμε και στην ενότητα 1.2. Χάρη στο mpiP βλέπουμε αναλυτικά την κατανομή του συνολικού χρόνου ανα διεργασία, ανα συνάρτηση και πολλά άλλα χρήσιμα στατιστικά (βλ. Εικόνες 11, 12, 13 & 14).

Αναλυτικότερα, οι εικόνες 11 και 12 αφορούν την μέτρηση του προγράμματος MPI για NMAX = 16800 και 4 διεργασίες χωρίς τη χρήση της MPI_Test συνάρτησης. Αντίστοιχα, οι εικόνες 13 και 14 αφορούν το πρόγραμμα με τις ίδιες παραμέτρους, αλλά περιλαμβανομένης της MPI_Test. Στις εικόνες 11 και 13 βλέπουμε το mpiP που μας δείχνει το χρονικό διάστημα που απαιτείται για να τρέξει ο κώδικας μεταξύ των εντολών MPI_Init και MPI_Finalize, καθώς και ο χρόνος και το ποσοστό που καταλαμβάνουν οι MPI συναρτήσεις. Τέλος, στις εικόνες 12 και 14 βλέπουμε το mpiP που κατατάσσει κατά φθίνουσα σειρά τις κλήσεις των συναρτήσεων με βάση τον χρόνο εκτέλεσης τους.

```
@--- MPI Time (seconds) --------------------------------------------------
--------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0        346        115    33.17
   1        346        119    34.33
   2        346        113    32.77
   3        346        119    34.50
   *   1.38e+03        466    33.69
--------------------------------------------------------------------------
```

Εικόνα 11: Το mpiP μας δείχνει το χρόνο ζωής των διεργασιών και τι ποσοστό αυτού καταλαβάνουν οι MPI συναρτήσεις στην πρώτη προσέγγιση με χρήση της MPI_Waitall.

```
--------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
--------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Waitall                89   2.32e+05   16.76   49.73    1.15
Waitall                50   2.31e+05   16.73   49.67    1.10
Allreduce              28   2.31e+03    0.17    0.50    1.13
Wait                   78        109    0.01    0.02    1.34
Barrier                98        108    0.01    0.02    1.13
Cart_create            62       67.5    0.00    0.01    1.52
Wait                   86       64.6    0.00    0.01    1.33
```

Εικόνα 12: Το mpiP κατατάσσει κατα φθίνουσα σειρά τις κλήσεις των συναρτήσεων με βάση τον χρόνο εκτέλεσης τους κατα την προσέγγιση με χρήση της MPI_Waitall.

```
--------------------------------------------------------------------------
@--- MPI Time (seconds) --------------------------------------------------
--------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0        294       72.1    24.50
   1        294       71.2    24.20
   2        294       70.1    23.82
   3        294       70.2    23.85
   *   1.18e+03        284    24.09
--------------------------------------------------------------------------
```

Εικόνα 13: Ο χρόνος ζωής των διεργασιών και το ποσοστό που καταλαβάνουν οι MPI συναρτήσεις στην δεύτερη προσέγγιση με χρήση της MPI_Test σύμφωνα με το mpiP.

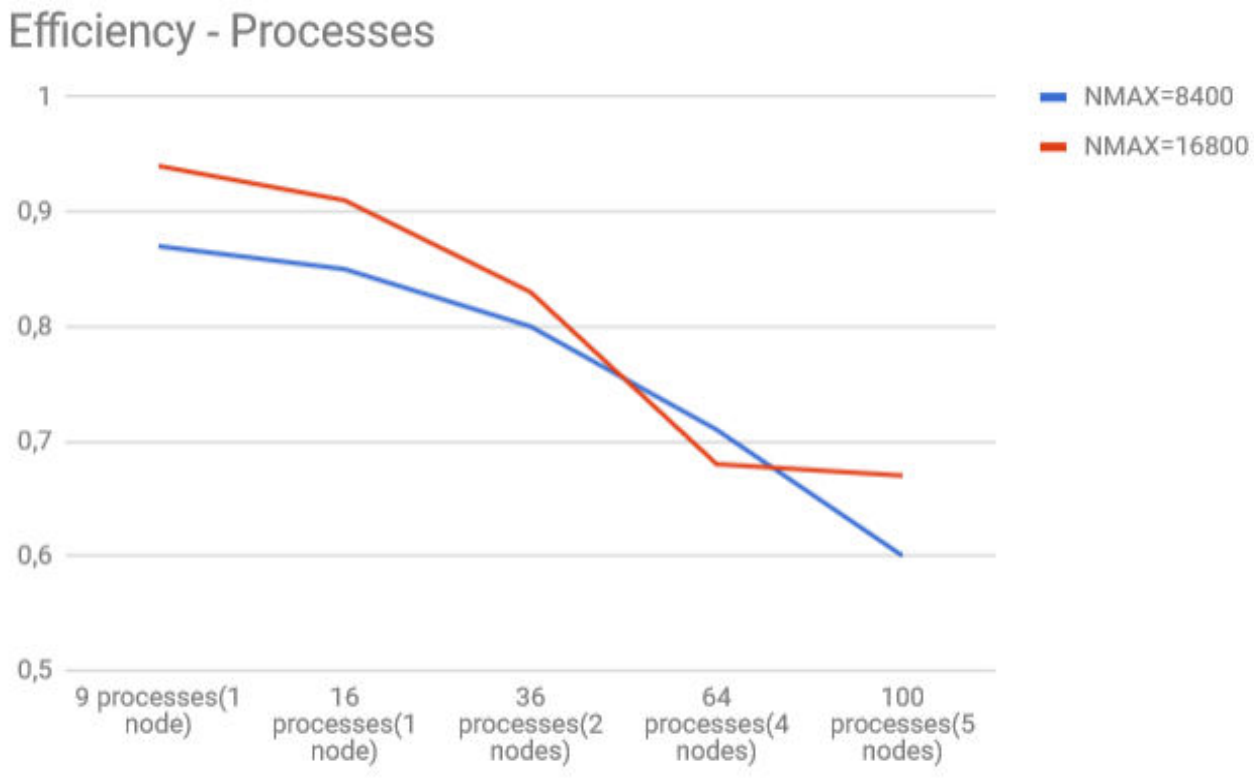
```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                    Site       Time    App%    MPI%      COV
Test                      35   1.42e+05   12.06   50.07     0.64
Test                      84   1.33e+05   11.33   47.03     0.69
Allreduce                 58   8.07e+03    0.69    2.84     0.31
Wait                      45       24.1    0.00    0.01     1.77
Wait                       8       21.5    0.00    0.01     2.00
Startall                  83       18.8    0.00    0.01     0.05
Startall                  39       18.6    0.00    0.01     0.04
```

Εικόνα 14: Αποτελέσματα του mpiP για τις κλήσεις των συναρτήσεων στο πρόγραμμα με χρήση της MPI_Test.

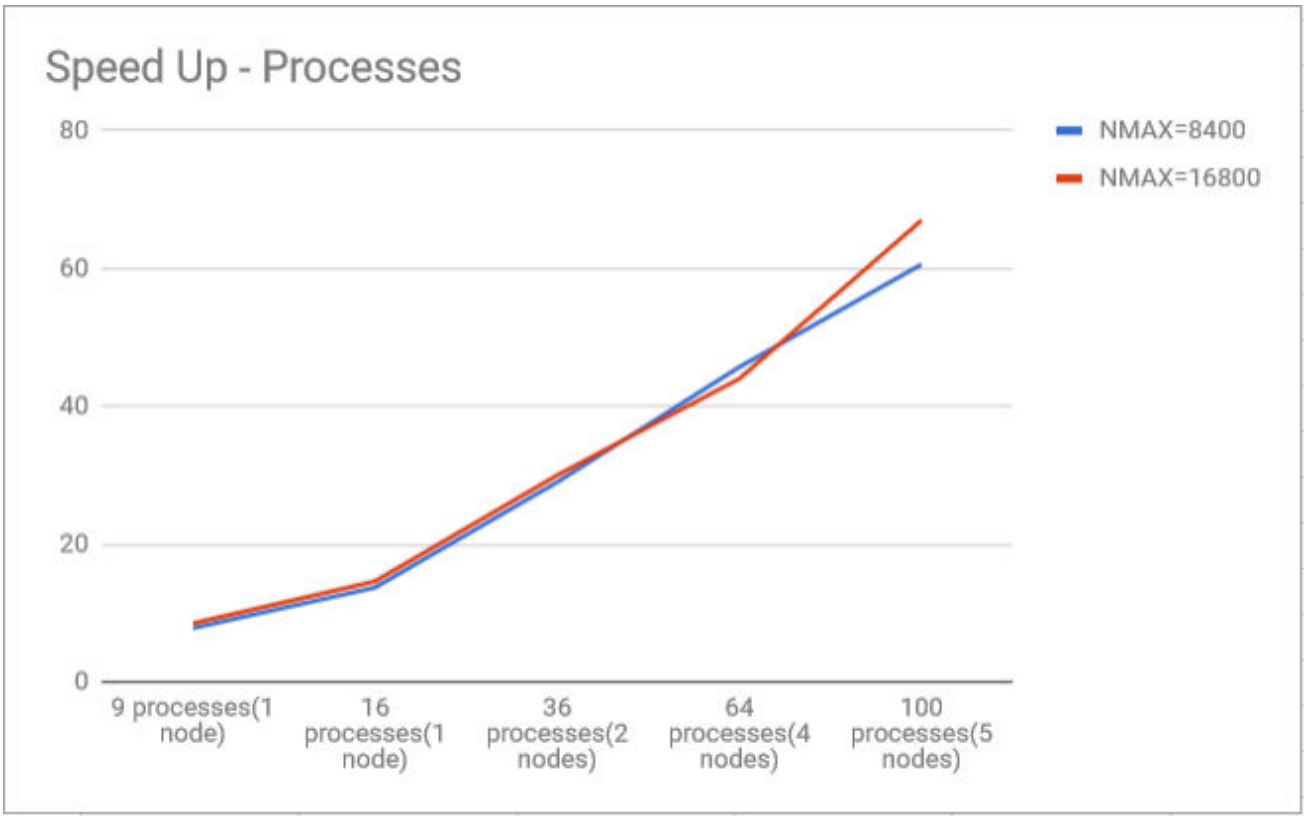
3.4 Διαγράμματα Speed Up & Efficiency

Με βάση τις μετρήσεις μας καταλήξαμε ότι το MPI πρόγραμμα μας παρουσιάζει κλιμάκωση σε αντίθεση με την υβριδική υλοποίηση MPI και OpenMP.

Στα παρακάτω διαγράμματα Speed Up – Processes και Efficiency – Processes παρουσιάζονται οπτικά κάποια αποτελέσματα των μετρήσεων μας που μας βοήθησαν να εξάγουμε τα τελικά συμπεράσματα μας όσον αφορά στη κλιμάκωση του MPI προγράμματος μας.



Εικόνα 15: Διάγραμμα Efficiency - Processes για NMAX=8400 & 16800.



Εικόνα 16: Διάγραμμα Speed Up - Processes για NMAX=8400 & 16800.

4. ΣΥΜΠΕΡΑΣΜΑΤΑ

Μετά από τις μετρήσεις που κάναμε καθώς και τις διάφορες αλλαγές που δοκιμάσαμε βλέπουμε ότι το MPI, όπως αναμενόταν, παρουσιάζει πολύ καλή κλιμάκωση, κάτι το οποίο είναι ακόμα πιο φανερό όταν εφαρμόζεται πάνω σε έναν υπερυπολογιστή, όπως ο ARIS HPC. Παρατηρήσαμε ότι όσο αυξάνουμε τον αριθμό των διεργασιών παρατηρείται μεγάλη βελτίωση στο χρόνο εκτέλεσης του αλγορίθμου, κάτι το οποίο φαίνεται και από τα σχήματα παραπάνω. Είναι δεδομένο ότι το MPI λόγω πολύ καλής τοπικότητας που έχει από τον σχεδιασμό του, παρουσιάζει πολύ καλά αποτελέσματα κάτι το οποίο επιβεβαιώνεται και από την δική μας μελέτη.

Όσον αφορά στο OpenMP, δοκιμάσαμε 2 διαφορετικούς τρόπους, τόσο το block partioning όσο και την παραλληλοποίηση της βασικής λούπας του αλγορίθμου χωρίς να δούμε ιδιαίτερες διαφορές μεταξύ των 2 αυτών μεθόδων. Τόσο στη μία, όσο και στην άλλη περίπτωση, η κλίμάκωση δεν ήταν η θεωρητικά αναμενόμενη, καθώς δεν μας προσέφερε η αύξηση του αριθμού των νημάτων αντίστοιχη επιτάχυνση με αυτή που μας προσέφεραν οι διεργασίες. Υποθέτουμε ότι για αυτό ευθύνεται το cache accesing (πρόσβαση κρυφής μνήμης), δηλαδή ο τρόπος με τον οποίο διαβάζονται τα δεδομένα απο τους πίνακες δεν είναι ο ιδανικός όσο αφορά στην καλύτερη δυνατή αξιοποίηση της cache και πολύ πιθανόν να παρατηρούνται φαινόμενα false sharing. Αφήνουμε την περαιτέρω μελέτη και βελτιστοποίηση των τεχνικών του OpenMP ως αντικείμενο μελλοντικών ερευνών.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Message	Μήνυμα
Process	Διεργασία
Hypercomputer	Υπερυπολογιστής
Hybrid	Υβριδικό
Red black reordering	Αναδιάταξη κόκκινου μαύρου
Thread	Νήμα
Thin node	Υπολογιστικός κόμβος
Fat node	Κόμβος μεγάλης μνήμης
Debugging	Αποσφαλμάτωση
Profiling	Ανάλυση απόδοσης
Distribution	Διανομή
Password	Κωδικός Πρόσβασης
Public key cryptography	Τεχνολογία Δημοσίου Κλειδιού
Login node	Κόμβος Διασύνδεσης
Batch job execution	Εκτέλεση σε ομάδα διεργασιών
Scheduler	Σύστημα χρονοπρογραμματισμού
Batch script	Σενάριο εργασίας
Web service	Υπηρεσία ιστού
Persistant service	Διηνεκής υπηρεσία
Successive Over-Relaxation	Διαδοχική Υπερμαλοποίηση
Local Successive Over-Relaxation	Τοπική μέθοδος διαδοχικής Υπερμαλοποίησης
Backward reordering	Αντίστροφη αναδιάταξη
Reordering	Αναδιάταξη
Overhead	Επιπλέον χρόνος
Block partioning	Διαχωρισμός σε μπλοκ
Block	Κομμάτι πίνακα
False sharing	Ψευδής κοινή χρήση
Cache accesing	Πρόσβαση κρυφής μνήμης
Speed Up	Λόγος επιτάχυνσης
Efficiency	Αποδοτικότητα
Scalability	Κλιμάκωση

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

LMSOR	Local Modified Successive Over-Relaxation
HPC	Hypercomputer
MPI	Message Passing Interface
MIMD	Multiple Instructions Multiple Data
LAM	Local Area Multicomputer
CHIMP	Common High Level Interface to Message Passing
API	Application Programming Interface
OpenMP	Open Multi-Processing
CPU	Central Processing Unit
SOR	Successive Over-Relaxation
PVM	Parallel Virtual Machine
ΕΔΕΤ	Ελληνικό Δίκτυο Έρευνας και Τεχνολογίας
Aris	Advanced Research Information System
SLURM	Simple Linux Utility for Resource Management
CUDA	Compute Unified Device Architecture
nvcc	NVidia cuda compiler

ΑΝΑΦΟΡΕΣ

[1] **http://doc.aris.grnet.gr/**

[2] **https://pergamos.lib.uoa.gr/uoa/dl/frontend/browse/1320502**

[3] Y. Cotronis et al., A comparison of CPU and GPU implementations for solving the Convection Diffusion equation using the local Modified SOR method, Parallel Comput. (2014), **http://dx.doi.org/10.1016/j.parco.2014.02.002** .

[4] Α. Ι. Μάργαρης, *MPI ΘΕΩΡΙΑ & ΕΦΑΡΜΟΓΕΣ,* ΕΚΔΟΣΕΙΣ ΤΖΙΟΛΑ, 2008.

[5] **http://mpip.sourceforge.net/**

[6] **http://www.openmp.org/**

[7] **https://pergamos.lib.uoa.gr/uoa/dl/frontend/file/lib/default/data/1324478/theFile**