ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

Παραλληλοποίηση Αλγορίθμου Επίλυσης Αραιών Γραμμικών Συστημάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γερμανός Ι. Τιμολέων

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2012

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

Παραλληλοποίηση Αλγορίθμου Επίλυσης Αραιών Γραμμικών Συστημάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γερμανός Ι. Τιμολέων

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 15/11/2012

Ν. Κοζύρης	Π. Τσανάκας	Δ. Τσουμάκος
Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επ. Καθηγητής Ιονίου Πανεπιστημίου

Αθήνα, Νοέμβριος 2012

.....
Τιμολέων Ι. Γερμανός
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Τιμολέων Ι. Γερμανός, 2012
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις
του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Στόχος αυτής της διπλωματικής εργασίας είναι να ασχοληθεί και να εξερευνήσει την επίδοση στη λύση αραιών συστημάτων γραμμικών εξισώσεων. Με αυτά τα προβλήματα έρχονται αντιμέτωπες οι φυσικές,οικονομικές και γενικά κάθε είδους επιστήμες. Η απαίτηση για αποδοτική και γρήγορη λύση αυτών των προβλημάτων ωθεί την Επιστήμη των Υπολογιστών να αναζητήσει αποδοτικές λύσεις. Ιδιαίτερα ο κλάδος που ασχολείται με συστήματα Υψηλής Επίδοσης προσπαθεί να παράγει μοντέλα που καθιστούν την χρονοβόρα διαδικασία εκτέλεσης τέτοιων απαιτητικών προβλημάτων βιώσιμη.

Εξετάζεται η επαναληπτική μέθοδος σύγκλισης Conjugate Gradient,η οποία μπορεί να χειριστεί συμμετρικά συστήματα γραμμικών εξισώσεων. Αναλύεται ο αλγόριθμος που χρησιμοποιεί η μέθοδος,εντοπίζονται τα σημεία τα οποία μπορούν να παραλληλοποιηθούν με ορισμένα προγραμματιστικά μοντέλα και αναλύεται η επίδοση τους με στόχο να εξαχθεί συμπέρασμα όσο αφορά το κέρδος από την όποια παραλληλοποίησή τους.

Η παραλληλοποίηση γίνεται σε συστήματα

- κοινής μνήμης
- κατανεμημένης μνήμης

Και στις δύο αυτές πολύ διαφορετικές περιπτώσεις παραλληλισμού,προσπαθούμε να αναλύσουμε τις ιδιομορφίες που υπάρχουν,τις αδυναμίες και τις δυνατότητες με στόχο να εξάγουμε ένα συμπέρασμα για το βαθμό που κλιμακώνεται η επίδοση όσο αυξάνουμε την ισχύ του συστήματος.
Αναλύεται η έννοια της συμμετρικότητας ενός αραιού πίνακα και μελετάμε την επίδοση δύο συμμετρικών εκδόσεων του πυρήνα SpMV που αξιοποιούν την ιδιότητα αυτή ενός συμμετρικού αραιού πίνακα.

Ειδικότερα,στο κομμάτι του προγραμματισμού σε σύστημα κατανεμημένης μνήμης εξετάζεται ο βαθμός που θα διασπαστεί το αρχικό πρόβλημα σε υποπροβλήματα μικρότερου μεγέθους έτσι ώστε το κέρδος που θα έχουμε από τη διάσπαση του προβλήματος να μην εκτοπίζεται από το κόστος της επικοινωνίας που εισάγεται από τη φύση του κατανεμημένου συστήματος.

Από την άλλη, στο σύστημα κοινής μνήμης,αναλύεται η βελτίωση της επίδοσης σε συνάρτηση με τον αριθμό των χρησιμοποιούμενων πυρήνων που ενυπάρχουν στους επεξεργαστές κι αναλύεται το φράγμα που τοποθετεί η ταχύτητα της κύριας μνήμης όταν εντοπίζεται αστοχία στα δεδομένα. Τέλος υλοποιούμε μια ιδέα που έχει να κάνει με τη συμπίεση των δεδομένων έτσι ώστε αυτά να καταφθάνουν ταχύτερα από την κύρια μνήμη σε περίπτωση αστοχίας κι αναλύουμε την επίδοσή της.

Abstract

The aim of this thesis is to explore and deal with the performance of solving sparse linear systems. Physical, economic and generally any kind of scientists are dealing with these problems. the requirement for efficient and fast solution to these problems is driving science computer seek efficient solutions. Particularly high performance computing science is attempting to produce models that make the lengthy implementation process of challenging such problems viable.

We examine the iterative convergence method conjugate gradient, which can handle symmetrical linear systems. We analyze the algorithm that the method uses,choose the points which can be parallelised using certain programming models and analyze their performance in order to reach a conclusion regarding profit from any parallelization on them.

The parallelization is done in

- shared memory systems
- distributed memory systems

In these two very different cases of parallelism, we try to analyze the peculiarities that exist, weaknesses and opportunities in order to draw a conclusion about the extent to which the performance scales when increasing the efficiency of the system.

We analyze the concept of a sparse matrix symmetry and study the performance of two symmetric spmv kernel versions that utilize this characteristic of a symmetric sparse matrix.

Specifically, in distributed memory system programming we examine the extent to decompose the original problem into smaller subproblems so that the gain by splitting the problem can not be displaced by the cost of communication introduced by the distributed memory system nature.

On the other hand, the shared memory system, we analyze the performance improvement in terms of the number of used cores and analyze the barrier puts the speed of main memory when a failure in cache occurs.

Also, we implement an idea in compressing the data so that they arrive faster from the main memory in case of failure, and analyze its performance.

Περιεχόμενα

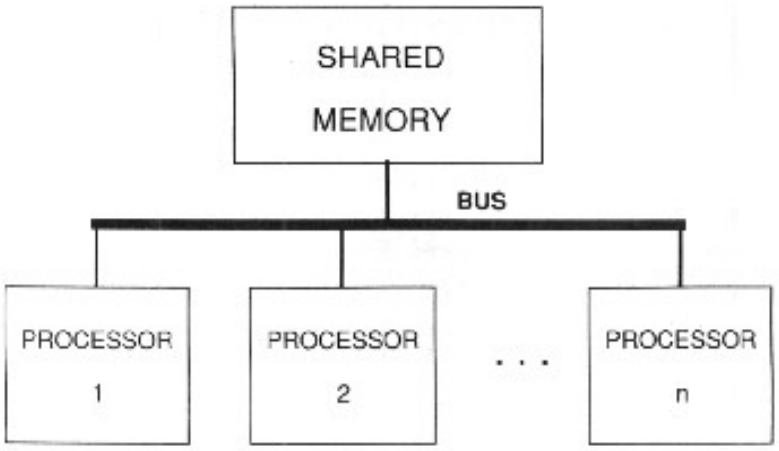
Σχήματα και Εικόνες

Κεφάλαιο 0 : Εισαγωγικές Έννοιες

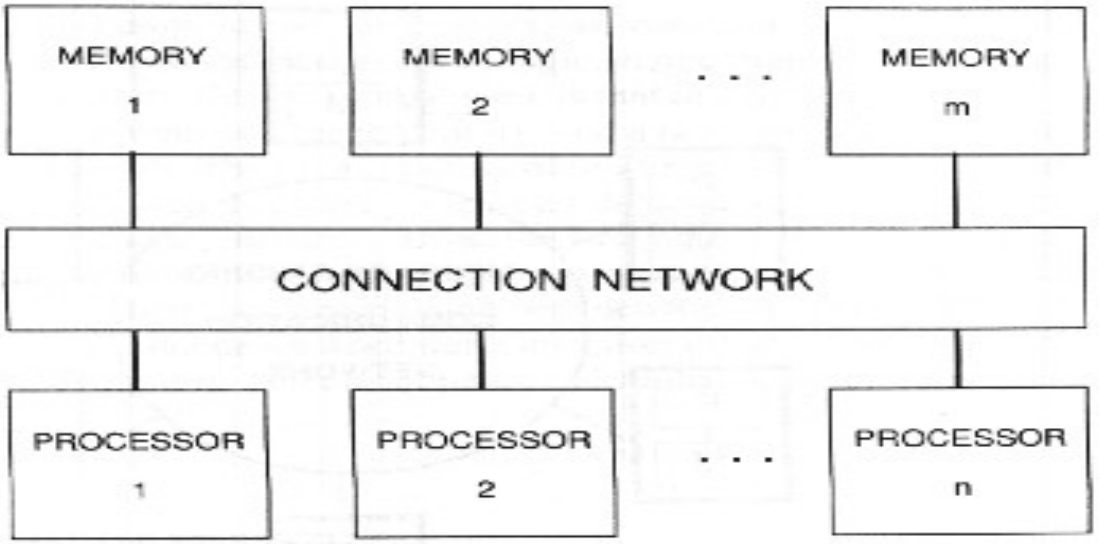
Η έννοια του παράλληλου συστήματος

Η βασική ιδέα πίσω από ένα παράλληλο σύστημα είναι απλά να υπάρχει πάνω από ένας επεξεργαστής στο σύστημα. Τα παράλληλα συστήματα μπορεί να έχουν πολύ λίγους επεξεργαστές για παράδειγμα 10 ή πάρα πολλούς π.χ. 50000. Το κλειδί για να χαρακτηριστεί ένα σύστημα σαν παράλληλο, είναι ότι όλοι οι επεξεργαστές του είναι ικανοί να λειτουργούν ταυτόχρονα. Αν ο κάθε επεξεργαστής μπορεί να εκτελέσει 1 εκατομμύριο λειτουργίες ανά δευτερόλεπτο, τότε 10 επεξεργαστές μπορούν να εκτελέσουν 10 εκατομμύρια λειτουργίες ανά δευτερόλεπτο, οι 100 επεξεργαστές μπορούν να εκτελέσουν 100 εκατομμύρια λειτουργίες ανά δευτερόλεπτο κ.λ.π.

Η χρήση των πολλαπλών παράλληλων επεξεργαστών στο ίδιο υπολογιστικό σύστημα εισάγει μερικές επιπρόσθετες απαιτήσεις στην αρχιτεκτονική του συστήματος. Για να μπορέσουν πολλοί επεξεργαστές να δουλέψουν ταυτόχρονα πάνω στο ίδιο υπολογιστικό πρόβλημα, πρέπει να είναι ικανοί να μοιράζονται δεδομένα και να επικοινωνούν μεταξύ τους. Υπάρχουν, προς το παρόν, δύο βασικές αρχιτεκτονικές προσεγγίσεις για την εκπλήρωση αυτής της απαίτησης: η διαμοιραζόμενη μνήμη και το πέρασμα μηνυμάτων. Στα συστήματα διαμοιραζόμενης μνήμης, που συχνά ονομάζονται και συστήματα πολυεπεξεργασίας όλοι οι μεμονωμένοι επεξεργαστές έχουν να προσπελάσουν μία κοινή μνήμη, και τους επιτρέπεται η κοινή χρήση των ποικίλων, τιμών δεδομένων και δομών δεδομένων που είναι αποθηκευμένες στη μνήμη. Στα συστήματα αρχιτεκτονικής περάσματος μηνυμάτων, που συχνά ονομάζονται και παράλληλα συστήματα κατανεμημένης μνήμης, ο κάθε επεξεργαστής έχει τη δική του τοπική μνήμη, και οι επεξεργαστές μοιράζονται δεδομένα μεταξύ τους, με το μηχανισμό περάσματος μηνυμάτων μέσω ενός είδους δικτύου επικοινωνίας επεξεργαστών. Και οι δύο αυτοί τύποι αρχιτεκτονικής παράλληλων συστημάτων έχουν σίγουρα πλεονεκτήματα και μειονεκτήματα. Θα μελετήσουμε τον προγραμματισμό τόσο των συστημάτων διαμοιραζόμενης μνήμης όσο και των παράλληλων συστημάτων κατανεμημένης μνήμης.



Εικόνα 0.1 - Οργάνωση Συστήματος Διαμοιραζόμενης Μνήμης με Κοινό Δίαυλο



Εικόνα 0.2 - Σύστημα Κατανεμημένης Μνήμης με Πολλαπλά Τμήματα

Συστήματα Κοινής μνήμης

Στα συστήματα αυτού τύπου, ο καθένας είναι δυνατόν να προσπελάσει την κεντρική διαμοιραζόμενη μνήμη μέσω μιας δομής κοινού διαύλου. Ο κοινός δίαυλος είναι υπεύθυνος για τη διαιτησία ανάμεσα στις ταυτόχρονες απαιτήσεις μνήμης των διαφόρων επεξεργαστών και για την εξασφάλιση της δίκαιας εξυπηρέτησης όλων των επεξεργαστών με την ελάχιστη καθυστέρηση προσπέλασης. Η λειτουργία του κάθε επεξεργαστή είναι ίδια με τη λειτουργία που εκτελείται από έναν επεξεργαστή σε ένα συνηθισμένο σειριακό σύστημα. Ο κάθε επεξεργαστής εξακολουθεί να διαβάζει τιμές δεδομένων από τη κοινή μνήμη, να υπολογίζει καινούργιες τιμές, και να τις γράφει πάλι πίσω στη κοινή μνήμη. Αυτή η υπολογιστική λειτουργία εκτελείται από όλους τους επεξεργαστές παράλληλα. Έτσι αν υπάρχουν n επεξεργαστές, τότε το υπολογιστικό σύστημα έχει μέγιστη υπολογιστική ικανότητα n φορές περισσότερη από ένα απλό σύστημα ενός επεξεργαστή.

Ο αριθμός των επεξεργαστών που είναι διαθέσιμος σε τέτοια συστήματα σήμερα ποικίλει με μέγιστο από 40 έως 100. Ένα από τα βασικά προβλήματα των συστημάτων διαμοιραζόμενης μνήμης είναι ο ανταγωνισμός πρόσβασης στη μνήμη ανάμεσα στους επεξεργαστές, ο οποίος αυξάνεται όσο περισσότεροι επεξεργαστές προστίθενται. Όταν πολλοί από τους επεξεργαστές προσπαθούν να προσπελάσουν την κοινή μνήμη μέσα σε σύντομο χρονικό διάστημα, η μνήμη δεν θα είναι ικανή να εξυπηρετήσει όλες τις απαιτήσεις ταυτόχρονα, και μερικοί από τους επεξεργαστές πρέπει να περιμένουν ενώ άλλοι θα εξυπηρετούνται. Στο σχεδιασμό των συστημάτων διαμοιραζόμενης μνήμης, μεγάλη προσοχή πρέπει να δοθεί στη μείωση της πιθανότητας ανταγωνισμού των τμημάτων μνήμης.

Μια μεγάλη ποικιλία τεχνικών χρησιμοποιείται για να βοηθήσει τη μείωση του ανταγωνισμού πρόσβασης στη μνήμη και να κάνει το σύστημα πιο αποδοτικό. Μια τέτοια τεχνική είναι να επιτρέπεται σε κάθε επεξεργαστή να έχει μία τοπική κρυφή μνήμη, η οποία χρησιμοποιείται για να κρατάει αντίγραφα από τα πιο πρόσφατα χρησιμοποιημένα τμήματα μνήμης. Από τη στιγμή που ο κάθε επεξεργαστής έχει τη δική του τοπική κρυφή μνήμη, τα δεδομένα της μπορούν να προσπελαστούν πολύ γρήγορα χωρίς πιθανότητα ανταγωνισμού. Όμως έτσι εισάγεται το πρόβλημα της συνοχής των τμημάτων κρυφής μνήμης: είναι δυνατό να υπάρχουν πολλαπλά αντίγραφα της ίδιας μεταβλητής σε διαφορετικές τοπικές κρυφές μνήμες, γεγονός που οδηγεί στην πιθανότητα ύπαρξης μη ενημερωμένων τιμών μετά από κάποια ενημέρωση. Μια ποικιλία εξεζητημένων τεχνικών για τη συνοχή της κρυφής μνήμης έχει αναπτυχθεί σε διάφορα συστήματα για να λύσει αυτό το πρόβλημα. Μια τέτοια τεχνική είναι ο κάθε επεξεργαστής να έχει μια κρυφή μνήμη με έλεγχο (snooping cache) η οποία να παρακολουθεί αδιάκοπα το κοινό δίαυλο και να καθιστά άκυρες τις τιμές μνήμης που έχουν ενημερωθεί (τροποποιηθεί) από άλλους επεξεργαστές.

Μια άλλη τεχνική για τη μείωση του ανταγωνισμού των τμημάτων μνήμης στα συστήματα διαμοιραζόμενης μνήμης είναι η διαίρεση της κοινής μνήμης σε χωριστά τμήματα, τα οποία μπορούν να προσπελαστούν παράλληλα από διαφορετικούς επεξεργαστές. Τα διαμοιραζόμενα δεδομένα κατανέμονται σε πολλά χωριστά τμήματα μνήμης, έτσι ώστε να μειώνεται η πιθανότητα της ταυτόχρονης ζήτησης του ίδιου τμήματος μνήμης από διαφορετικούς επεξεργαστές. Καθένας από τους n επεξεργαστές μπορεί να προσπελάσει οποιοδήποτε τα m τμήματα μνήμης, μέσω ενός δικτύου επικοινωνίας μνήμης. Αυτό το δίκτυο επικοινωνίας είναι ικανό ως ένα βαθμό για εσωτερικό παραλληλισμό, έτσι ώστε πολλοί επεξεργαστές να μπορούν να προσπελάσουν διαφορετικά τμήματα

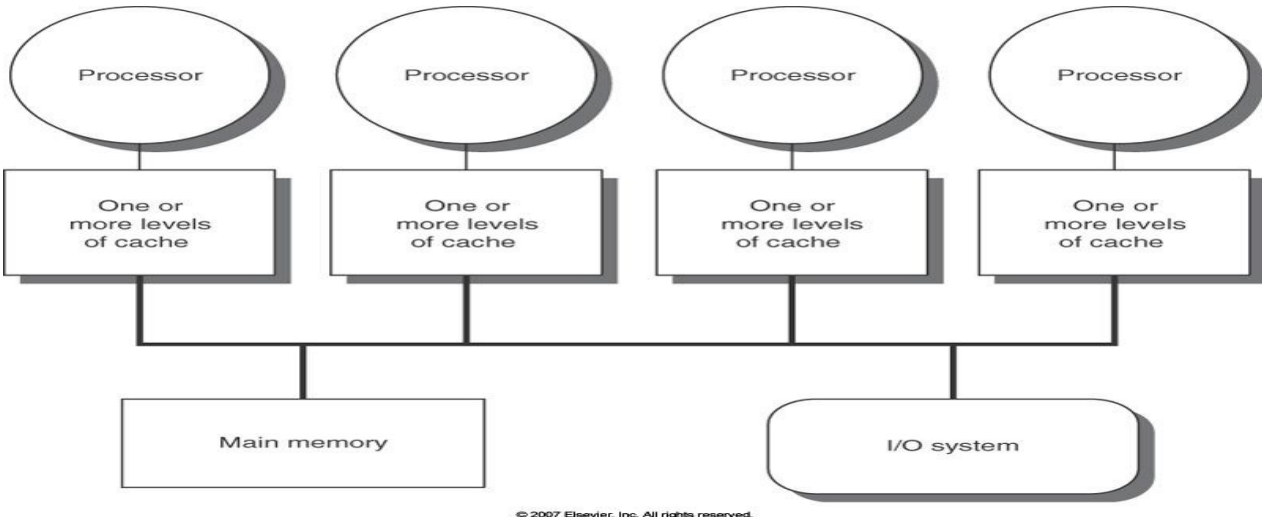
μνήμης ταυτόχρονα. Το κόστος και η εκτέλεση αυτού του μοντέλου εξαρτάται από τον εσωτερικό σχεδιασμό του δικτύου επικοινωνίας.

Η ύπαρξη πολλαπλών τμημάτων μνήμης μπορεί να βελτιώσει την απόδοση του συστήματος διαμοιραζόμενης μνήμης επειδή πολλά τμήματα μνήμης είναι δυνατόν να προσπελαστούν παράλληλα. Ο παραλληλισμός στην προσπέλαση μνήμης βοηθάει την αύξηση του παραλληλισμού στη λειτουργία των επεξεργαστών: οι πιθανότητες ανταγωνισμού πρόσβασης στη μνήμη μειώνονται σημαντικά. Είναι σαν να συγκρίνεις μία τράπεζα που έχει πολλά ταμεία με μία άλλη που έχει μόνο ένα. Ο αυξημένος αριθμός ταμείων ότι μειώνει την πιθανότητα αναμονής του πελάτη. Με τον ίδιο τρόπο ο αυξημένος αριθμός τμημάτων μνήμης μειώνει την πιθανότητα ο επεξεργαστής να πρέπει να περιμένει έως ότου να προσπελάσει την μνήμη.

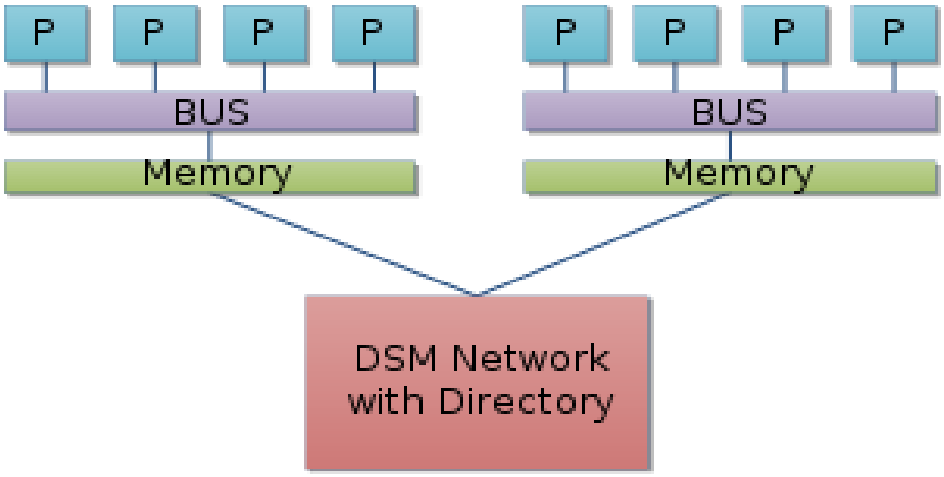
Είναι σημαντικό να θυμόμαστε γι΄ αυτό το μοντέλο διαμοιραζόμενης μνήμης, ότι τα πολλαπλά τμήματα μνήμης μαζί σχηματίζουν μια απλή διαμοιραζόμενη μνήμη, η οποία είναι προσπελάσιμη απ' όλους τους επεξεργαστές. Τη λειτουργία του δικτύου επικοινωνίας χειρίζεται εξ' ολοκλήρου το υλικό προσπέλασης μνήμης, και δεν είναι ορατό στον προγραμματιστή, ο οποίος απλά βλέπει μια ενιαία κεντρική διαμοιραζόμενη μνήμη. Υπάρχει ένας μοναδικός χώρος διεύθυνσης μνήμης που είναι ορατός πανομοιότυπα απ' όλους τους επεξεργαστές. Αυτές οι διευθύνσεις μνήμης στην πραγματικότητα κατανέμονται στα πολλαπλά τμήματα μνήμης, αλλά αυτό δεν είναι ορατό στους επεξεργαστές. Όταν ο επεξεργαστής διαβάζει ή γράφει μία συγκεκριμένη διεύθυνση μνήμης, η δραστηριότητα του δικτύου επικοινωνίας και η επιλογή του κατάλληλου τμήματος μνήμης ρυθμίζεται αυτόματα από το υλικό προσπέλασης μνήμης.

Συνοψίζοντας έχουμε για τα χαρακτηριστικά των συστημάτων κοινής μνήμης:

- Οι επεξεργαστές έχουν κοινή μνήμη
- Κάθε επεξεργαστής διαθέτει τοπική ιεραρχία κρυφών μνημών
- Συνήθως η διασύνδεση γίνεται μέσω διαδρόμου μνήμης (memory bus) αλλά και πιο εξελιγμένα δίκτυα διασύνδεσης
- Ομοιόμορφη ή μη-ομοιόμορφη προσπέλαση στη μνήμη (Uniform Memory Access – UMA, Non uniform Memory Access - NUMA)
- Η κοινή μνήμη διευκολύνει τον παράλληλο προγραμματισμό
- Δύσκολα κλιμακώσιμη αρχιτεκτονική – τυπικά μέχρι λίγες δεκάδες κόμβους (δεν κλιμακώνει το δίκτυο διασύνδεσης)



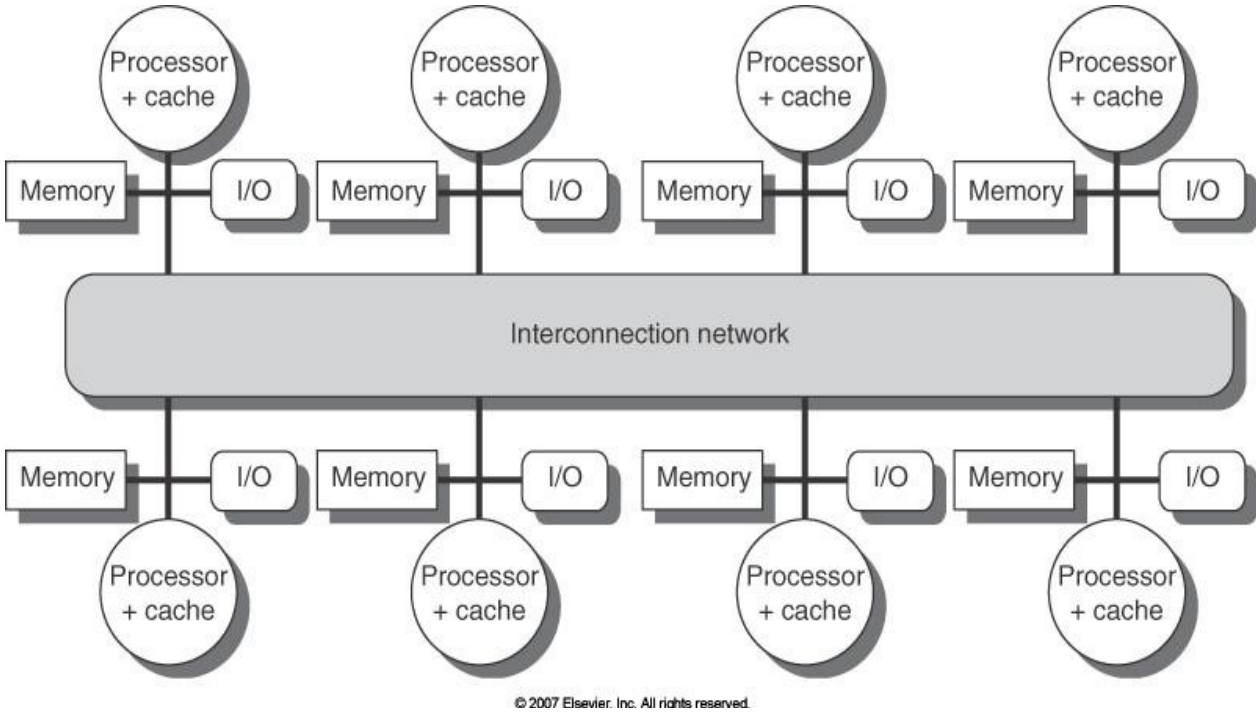
Εικόνα 0.3 - SMP



Εικόνα 0.4 - NUMA

Συστήματα κατανεμημένης μνήμης

Μία άλλη προσέγγιση για τη μείωση του ανταγωνισμού των τμημάτων μνήμης στα συστήματα παράλληλης επεξεργασίας είναι η ολοκληρωτική εξάλειψη της διαμοιραζόμενης μνήμης και η εξασφάλιση μιας αρκετά μεγάλης τοπικής μνήμης για κάθε επεξεργαστή καθώς και ενός δικτύου επικοινωνίας για την αλληλεπίδραση και των επεξεργαστών μέσω ενός μηχανισμού περάσματος μηνυμάτων. Ο κάθε επεξεργαστής λειτουργεί αυτόνομα, χρησιμοποιώντας τα δεδομένα που είναι αποθηκευμένα στο δικό του τμήμα τοπικής μνήμης. Ο κάθε επεξεργαστής μπορεί επίσης να στέλνει και να λαμβάνει δεδομένα προς και από οποιονδήποτε άλλο επεξεργαστή, χρησιμοποιώντας το δίκτυο επικοινωνίας περάσματος μηνυμάτων.



Εικόνα 0.5 - Οργάνωση Συστήματος Κατανεμημένης Μνήμης

Η βασική οργάνωση του παράλληλου συστήματος κατανεμημένης μνήμης είναι ριζικά διαφορετική από την οργάνωση του συστήματος διαμοιραζόμενης μνήμης που συζητήθηκε προηγούμενα. Ένα παράλληλο σύστημα κατανεμημένης μνήμης συμπεριφέρεται με εντελώς διαφορετικό τρόπο απ' ότι ένα σύστημα διαμοιραζόμενης μνήμης και απαιτεί ένα διαφορετικό εννοιολογικό μοντέλο προγραμματισμού για την ανάπτυξη του λογισμικού. Τώρα ο επεξεργαστής γνωρίζει τη διαφορά ανάμεσα στα τοπικά και απομακρυσμένα τμήματα μνήμης. Κάθε ζευγάρι επεξεργαστή-μνήμης συμπεριφέρεται σαν ένα μικρό, αυτόνομα ενιαίο σύστημα υπολογιστή. Ο επεξεργαστής μπορεί να διαβάζει και να γράφει δεδομένα ελεύθερα, χρησιμοποιώντας την δική του τοπική μνήμη. Όταν οι επεξεργαστές θέλουν να ανταλλάξουν δεδομένα, τότε αυτό πρέπει να γίνει μέσω μιας ρητής ενέργειας ανταλλαγής μηνυμάτων χρησιμοποιώντας το δίκτυο επικοινωνίας. Ο επεξεργαστής έχει άμεση πρόσβαση μόνο στα δικά του τοπικά τμήματα μνήμης και όχι στα τμήματα μνήμης που είναι συνδεδεμένα με άλλους επεξεργαστές. Όμως κάθε επεξεργαστής μπορεί να διαβάζει

τιμές δεδομένων από τη δική του τοπική μνήμη και να στέλνει αυτά τα δεδομένα σε οποιονδήποτε άλλον επεξεργαστή. Έτσι αυτά τα δεδομένα μπορούν να μοιράζονται και να ανταλλάσσονται ελεύθερα ανάμεσα στους επεξεργαστές.

Υπάρχει μια μεγάλη ποικιλία διαφορετικών τοπολογιών δικτύου επικοινωνίας, που έχουν αναπτυχθεί για παράλληλα συστήματα κατανεμημένης μνήμης. Σκοπός αυτών των τοπολογιών, είναι να προσπαθήσουν να μειώσουν το κόστος και την πολυπλοκότητα του δικτύου, όταν η γρήγορη επικοινωνία μεταξύ των επεξεργαστών επιτρέπεται. Γενικά δεν είναι δυνατό σε μεγάλα παράλληλα συστήματα κατανεμημένης μνήμης να εξασφαλιστεί επικοινωνία ανάμεσα σε κάθε ζευγάρι επεξεργαστών, μια και αυτό θα απαιτούσε n^2 κανάλια επικοινωνίας για n επεξεργαστές. Για να διατηρηθεί ένα λογικό κόστος και να επιτραπεί στο σύστημα να αυξήσει εύκολα τον αριθμό των επεξεργαστών του, ο αριθμός των μονοπατιών επικοινωνίας που φτάνουν σε κάθε επεξεργαστή πρέπει να είναι σταθερός(ανεξάρτητος από τον συνολικό αριθμό επεξεργαστών) ή να αυξάνει λογαριθμικά με το συνολικό αριθμό επεξεργαστών.

Σε αυτό το κείμενο, γίνεται μια σημαντική διάκριση ανάμεσα στα συστήματα διαμοιραζόμενης μνήμης και στα συστήματα κατανεμημένης μνήμης. Ο προγραμματισμός αυτών των δύο κύριων κατηγοριών παράλληλων συστημάτων είναι αρκετά διαφορετικός, και απαιτεί διαφορετικές αντιλήψεις του υλικού του συστήματος. Αρχίζοντας, ο προγραμματισμός των συστημάτων διαμοιραζόμενης μνήμης είναι μάλλον πιο εύκολος, γιατί μοιάζει περισσότερο με τον προγραμματισμό του απλού σειριακού συστήματος επεξεργασίας με το οποίο είμαστε ήδη εξοικειωμένοι. Ο προγραμματισμός παράλληλων συστημάτων κατανεμημένης μνήμης, είναι περισσότερο πολύπλοκος γιατί το πρόγραμμα πρέπει να κάνει σαφή μεταφορά δεδομένων μεταξύ των επεξεργαστών. Αυτό γίνεται ακόμα πιο περίπλοκο από το γεγονός ότι η τοπολογία διασύνδεσης των επεξεργαστών πρέπει επίσης να μελετηθεί για τον σχεδιασμό του προγράμματος.

Συνοψίζοντας τα χαρακτηριστικά των συστημάτων κατανεμημένης μνήμης είναι:

- Κάθε επεξεργαστής έχει τη δική του τοπική μνήμη κι ιεραρχία τοπικών μνημών.
- Κάθε επεξεργαστής διασυνδέεται με τους υπόλοιπους επεξεργαστές με κάποιο δίκτυο διασύνδεσης.
- Το μοντέλο κατανεμημένης μνήμης δυσκολεύει τον προγραμματισμό.
- Η αρχιτεκτονική κλιμακώνει σε χιλιάδες επεξεργαστές.

Τέλος υπάρχει η υβριδική αρχιτεκτονική που όπως φαίνεται κι από το όνομά της δεν είναι τίποτα άλλο από συνδυασμός των παραπάνω 2 αρχιτεκτονικών.

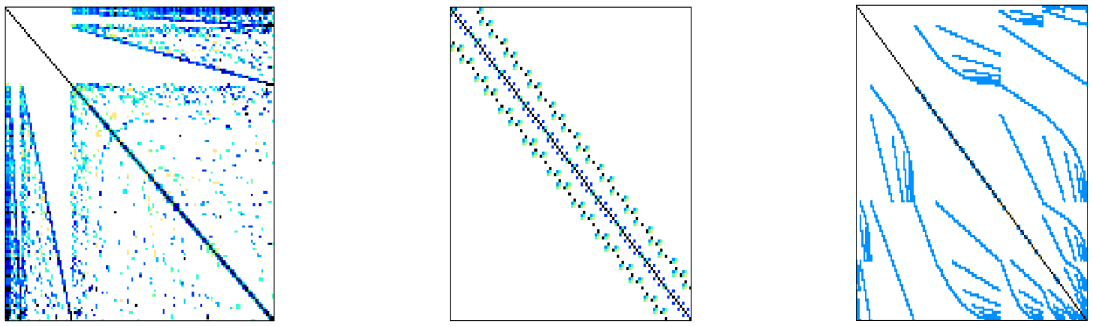
- Κόμβοι με αρχιτεκτονική κοινής μνήμης διασυνδέονται με κάποιο δίκτυο διασύνδεσης σε αρχιτεκτονική κατανεμημένης μνήμης.
- Είναι τυπική αρχιτεκτονική των σύγχρονων συστοιχιών-υπερυπολογιστών.

ΑΡΑΙΟΙ ΠΙΝΑΚΕΣ

Αραιός χαρακτηρίζεται ο πίνακας διαστάσεων N*M, στον οποίο κυριαρχούν τα μηδενικά στοιχεία. Ειδικές τεχνικές μπορούν να εφαρμοστούν πάνω σε αυτούς τους πίνακες λαμβάνοντας υπόψιν το μεγάλο αριθμό μηδενικών στοιχείων και επίσης τις σχετικές θέσεις τους,δηλαδή την τοπολογία των μη μηδενικών στοιχείων.

Μπορούμε να εξάγουμε συμπέρασμα ότι ένας πίνακας είναι αραιός όταν τα μη μηδενικά του στοιχεία είναι μερικές τάξεις μεγέθους σε σχέση με το πλήθος των συνολικών του στοιχείων το οποίο είναι ίσο με N*M.

Μερικά παραδείγματα-εικόνες αραιών πινάκων φαίνονται στις παρακάτω εικόνες.



Εικόνα 0.6 - Εικόνες Αραιών Πινάκων

Οι αραιοί πίνακες συναντώνται σε επιστημονικές εφαρμογές κι εφαρμογές μηχανικών και προκύπτουν κατά βάση όταν μελετάμε συστήματα τα οποία είναι χαλαρά συνδεδεμένα. Τα μεγάλα αραιά συστήματα εμφανίζονται κατά τη διαδικασία διακριτοποίησης όταν λύνουμε μερικές διαφορικές εξισώσεις(PDE).Ειδικότερα, ο συνήθης τρόπος λύσης μερικών διαφορικών εξισώσεων είναι η διακριτοποίηση όπως είναι το Finite Element Method (FEM),η οποία συνήθως οδηγεί σε προβλήματα με μεγάλους αραιούς πίνακες.

Μια άλλη χρησιμότητα των αραιών πινάκων είναι στην αναπαράσταση μεγάλων γραφημάτων χρησιμοποιούμενοι ως πίνακες γειτνίασης. Ένα τέτοιο παράδειγμα είναι το World-Wide Web,όπου κάθε κορυφή του γραφήματος είναι μια ιστοσελίδα και κάθε κατευθυνόμενη ακμή από τον A στον B αναπαριστά την ύπαρξη ενός URL συνδέσμου στην ιστοσελίδα A το οποίο οδηγεί στην ιστοσελίδα B.

Γενικά,η γραφοθεωρία και οι αραιοί πίνακες είναι δυο έννοιες πολύ στενά συνδεδεμένες:οι αλγόριθμοι γραφημάτων χρησιμοποιούνται πάνω στους αραιούς πίνακες π.χ. στο partitioning των αραιών πινάκων,κι αντίστροφα οι αλγόριθμοι γραφημάτων μπορούν να εκφραστούν και να μοντελοποιηθούν μέσω υπολογισμών πάνω σε αραιούς πίνακες.

SpMV

Μια σημαντική λειτουργία των αραιών πινάκων είναι ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα. Σε αυτήν τη διαδικασία ένας αραιός πίνακας διαστάσεων N*M πολλαπλασιάζεται με ένα πυκνό διάνυσμα διάστασης M και προκύπτει αποτέλεσμα ένα πυκνό διάνυσμα διάστασης N. Είναι ξεκάθαρο ότι τα μηδενικά στοιχεία του αραιού πίνακα δεν συνεισφέρουν στον πολλαπλασιασμό και μπορούν να αγνοηθούν.

Το SpMV χρησιμοποιείται κατά κόρον σε μια μεγάλη γκάμα εφαρμογών στους επιστημονικούς υπολογισμούς και στον τομέα των μηχανικών. Είναι η βασική λειτουργία επαναληπτικών μεθόδων όπως η Conjugate Gradient(CG) και Generalized Minimum Residual (GMRES)οποίες χρησιμοποιούνται για να λύσουν αραιά γραμμικά συστήματα τα οποία προκύπτουν από εξομοίωση φυσικών διαδικασιών που περιγράφονται από μερικές διαφορικές εξισώσεις.

Επιπλέον σημαντικός αριθμός αλγορίθμων γραφημάτων μπορεί να εκφραστεί μέσω του πολλαπλασιασμού του πίνακα γειτνίασης πολλοί από αυτούς τους αλγορίθμους είναι επαναληπτικοί στους οποίους ο χρόνος κάθε επανάληψης χαρακτηρίζεται και φράσσεται-κάτω από τη η λειτουργία SpMV.

Τέλος το SpmV έχει χαρακτηριστεί ως ένας από τους νάνους,μια κλάση προβλημάτων-εφαρμογών οι οποίες πιστεύεται ότι θα είναι μεγίστης σημασίας για την επόμενη δεκαετία τουλάχιστον.

Το SpMV μπορεί να αναπαρασταθεί από την παρακάτω σχέση:

$$y = Ax$$

CSR

Μέχρι στιγμής δεν έχουμε αναλύσει καθόλου τον τρόπο αποθήκευσης των αραιών πινάκων. Ας δούμε πως αποθηκεύεται ένας κοινός πυκνός πίνακας. Ένας πυκνός πίνακας αποθηκεύεται στη μνήμη συνεχόμενα και χρειαζόμαστε χώρο αποθήκευσης ανάλογο των διαστάσεων n και m του πίνακα. Μπορούμε να τον αποθηκεύσουμε κατά γραμμές(C style) το οποίο είναι και συνηθέστερο, η κατά στήλες(FORTRAN style).Για έναν πίνακα διαστάσεων N*M αποθηκευμένο κατά γραμμές,ξέρουμε ότι το στοιχείο aij βρίσκεται στη θέση j + (i * M),ενώ αν είναι αποθηκευμένος κατά στήλες βρίσκεται στη θέση i + (j * N).

Για έναν αραιό πίνακα όμως,επειδή ξέρουμε ότι το ποσοστό χρήσιμων στοιχείων του,δηλαδή των μη-μηδενικών,είναι πολύ μικρο θα ήταν άσκοπο να αποθηκεύουμε ολόκληρο δίνοντας στα μηδενικά στοιχεία του ίση σημαντικότητα με τα μη-μηδενικά. Άρα η φιλοσοφία είναι ότι αποθηκεύουμε μόνο αυτά που χρειάζονται και συνεισφέρουν στον πολλαπλασιασμό δηλαδή μόνο τα μη-,μηδενικά. Γι'αυτό το σκοπό, έχουν αναπτυχθεί διάφορες μέθοδοι αποθήκευσης αραιών πινάκων που αποσκοπούν στη μείωση του κόστους αποθήκευσης και χρόνου προσπέλασης των στοιχείων τέτοιων πινάκων.

Ωστόσο,χρειαζόμαστε κάποια πληροφορία για τη θέση κάθε στοιχείου που αποθηκεύεται. Γι'αυτό το λόγο χωρίζουμε τα δεδομένα αποθήκευσης του αραιού πίνακα σε 2 βασικές κατηγορίες. Στα δεδομένα δεικτοδότησης που μας δίνουν πληροφορία για τη σχετική θέση κάθε μη-μηδενικού στοιχείου στον πίνακα και στα δεδομένα που μας δίνουν την τιμή του μη-μηδενικού στοιχείου.

Η πιο διαδεδομένη μορφή αποθήκευσης αραιών πινάκων είναι η compressed storage row format (CSR). Η μορφή CSR αποθηκεύει τον πίνακα ως αραιά διανύσματα ένα για κάθε γραμμή του πίνακα, κι επιτρέπει την τυχαία προσπέλαση σε τυχαία γραμμή. Ειδικότερα,ο πίνακας αποθηκεύεται σε τρία διανύσματα: values, row_ptr and col_ind.

Το διάνυσμα values κρατάει τις τιμές των μη μηδενικών στοιχείων κατά γραμμές. Το διάνυσμα row_ptr περιέχει για κάθε γραμμή την θέση μέσα στο διάνυσμα values του πρώτου μη μηδενικού στοιχείου του πίνακα. Και τέλος,το διάνυσμα col_ind περιέχει τον αριθμό της στήλης στην οποία ανήκει το κάθε μη μηδενικό στοιχείο.

Ένα παράδειγμα αποθήκευσης αραιού πίνακα με τη μορφή CSR φαίνεται στην παρακάτω εικόνα.

A

1.3						
	2.98			15.83		
	-2.6					
		1.2			25.6	
						-98.4
				-0.25		
			12.6			8.9
						2.4

*

x

-

=

y

CSR Format of *A* matrix

row_ptr = 0 1 3 4 6 7 8 10 11

col_ind = 0 1 4 1 2 5 6 4 3 6 6

values = 1.3 2.98 15.83 -2.6 1.2 25.6 -98.4 -0.25 12.6 8.9 2.4

Το πλήθος των διανυσμάτων values και col_ind είναι ίσο με το πλήθος των μη-μηδενικών στοιχείων του πίνακα ενώ το πλήθος των στοιχείων του διανύσματος row_ptr είναι ίσο με τον αριθμό των γραμμών του πίνακα συν ένα. Ασυμπτωτικά μπορούμε να πούμε ότι το κόστος αποθήκευσης σε χώρο ενός πυκνού πίνακα είναι O(N*M) ενώ το κόστος για έναν αραιό πίνακα με τη μορφή CSR είναι O(nnz),όπου nnz είναι το πλήθος των μη-μηδενικών στοιχείων(number of non-zeros).

Καταλαβαίνουμε εύκολα ότι γλιτώνουμε πολύ χώρο για την αποθήκευση ενός αραιού πίνακα με αυτή την μορφή αποθήκευσης (CSR) αλλά και γενικότερα με όλες τις μορφές που αξιοποιούν την ιδιότητα ενός πίνακα να είναι αραιός.

Συγκεκριμένα για τον πίνακα Α που απεικονίζεται παρακάτω το row_ptr[2] δηλαδή αυτό που μας δίνει πληροφορία για τη γραμμή 3,μας λέει ότι υπάρχουν 3 μη μηδενικά στοιχεία πριν από την 3η γραμμή. Δηλαδή,το πρώτο στοιχείο της 3ης γραμμής είναι το 4ο κατά σειρά μη μηδενικό στοιχείο του πίνακα Α αφού υπάρχουν 3 στοιχεία πριν από αυτήν τη γραμμή. Με την ίδια λογική η 3η γραμμή έχοντας ένα μη – μηδενικό στοιχείο κάνει το row_ptr[3] της 4ης γραμμής να είναι ίσο με 3+1=4Τελος, το τελευταίο στοιχείο του διανύσματος row_ptr το οποίο αντιστοιχεί σε υποθετική γραμμή μετά το τέλος του πίνακα δείχνει τελικά πόσα στοιχεία έχει ο πίνακας μετά και τη συνεισφορά της τελευταίας γραμμής.

Ο αλγόριθμος του SpMV με χρήση του CSR,βελτιστοποιημένος με τη χρήση βοηθητικών μεταβλητών είναι ο ακόλουθος:

```
for (i=0; i<nrows; i++){

        row_i_plus_one=row_ptr[i+1];
        y_reg=0;

        for (j=row_ptr[i]; j<k; j++)
                y_reg += values[j]*x[col_ind[j]]

        y[i]=y_reg;
}
```

Ο πυρήνας του πολλαπλασιασμού αποτελείται κατ' ουσίαν από 2βρόχους:

- Έναν εξωτερικό που διατρέχει τον πίνακα κατά γραμμές χρησιμοποιώντας το διάνυσμα row_ptr και
- Έναν εσωτερικό ο οποίος διατρέχει τα στοιχεία κάθε γραμμής και δίνει ως αποτέλεσμα σε μια πλήρη εκτέλεση του ένα στοιχείο του διανύσματος-αποτελέσματος

Εκτός από τη μορφή αποθήκευσης CSR υπάρχουν κι άλλες μορφές οι οποίες συνοπτικά αναφέρονται παρακάτω:

- CSC:είναι παρεμφερής με τη μορφή CSR,με τη διαφοροποίηση ότι χρησιμοποιεί στήλες αντί για γραμμές,δηλαδή επιτρέπει την τυχαία προσπέλαση σε στήλες τις οποίες αποθηκεύει ως αραιά διανύσματα,ότι κάνει δηλαδή κι ο CSR με τις γραμμές.
- DIAG:Είναι της ίδιας λογικής με τους παραπάνω με τη διαφοροποίηση ότι ότι κάνουν οι προηγούμενοι με γραμμές και τις στήλες αντίστοιχα αυτός το κάνει με τις διαγώνιους.
- BCSR:είναι παραλλαγή του CSR ο οποίος είναι ίδιος με τον CSR μέσα σε ένα block του πίνακα ,καθολικά όμως αποθηκεύει με προτεραιότητα κατά block.

Πληροφορίες για τους πίνακες που θα χρησιμοποιήσουμε

Όνομα	N	nnz
helm2d03	392,257	2,741,935
pwtk	217,918	11,524,432
parabolic_fem	525,825	3,674,625
ship_001	34,920	3,896,496
hood	220,542	9,895,422
rajat31	4,690,002	20,316,253
hamrle3	1,447,360	5,514,242
nd24k	72,000	28,715,634
G3_Circuit	1,585,478	7,660,826
Si41Ge41H72	185,639	15,011,265
thermal2	1,228,045	8,580,313
af_5_k101	503,625	17,550,675
offshore	259,789	4,242,673
kkt_power	2,063,494	12,771,361
Freescale1	3,428,755	17,052,626
ASIC_680k	682,862	2,638,997
Si87H76	240,369	10,661,631
crankseg_2	63,838	14,148,858
Ga41As41H72	268,096	18,488,476
F1	343,791	26,837,113
nd6k	18,000	6,897,316

Κεφάλαιο 1ο: Η Μεθοδος Conjugate Gradient

Η μέθοδος που θα μας απασχολήσει και θα μελετήσουμε είναι η επαναληπτική μέθοδος σύγκλισης Conjugate Gradient.

Η μέθοδος αυτή έχει σκοπό την αριθμητική επίλυση συστημάτων γραμμικών εξισώσεων,συστημάτων των οποίων ο πίνακας είναι συμμετρικός και θετικά ορισμένος. Η μέθοδος αυτή μπορεί να εφαρμοστεί σε πολύ μεγάλα αραιά συστήματα εκεί όπου οι απευθείας μέθοδοι αποτυγχάνουν. Η γενίκευση της, η BiConjugate Gradient δεν έχει την παραπάνω απαίτηση της συμμετρικότητας του πίνακα του γραμμικού συστήματος.

Η ιδέα της μεθόδου είναι,δεδομένου του πολλαπλασιασμού A*x=b, έχοντας έναν πίνακα A κι ένα διάνυσμα b, πώς θα ανακτήσουμε το διάνυσμα x.

Αναλυτικά η μέθοδος είναι η εξής:

$r_0=b-A*x_0$
$p_0=r$
$k=0$
repeat
    $alpha=t^T_k*r_k/p^T_{k*}Ap_k$
    $x_{k+1}=x_k+alpha_{k*}p_k$
    $r_{k+1}=r_k-alpha_{k*}A*p_k$
    if r_{k+1} is sufficiently small then exit
    $beta_k=r^T_{k+1*}r_{k+1}/r^T_{k+1*}r_{k+1}$
    $p_{k+1}=r_{k+1} + beta_{k*}p_k$
    $k++$
end

Το αποτέλεσμα είναι στο X_{k+1}

Θεωρητικά,η μέθοδος συγκλίνει στη λύση σε n βήματα όση κι η διάσταση του συστήματος. Ωστόσο λόγω σφαλμάτων στρογγυλοποίησης τα οποία είναι αναπόφευκτα δουλεύοντας με αριθμούς κινητής υποδιαστολής,ίσως να χρειαστούν περισσότερα βήματα για τη σύγκλιση η ακόμα και να μην συγκλίνει.

Κώδικας σειριακού αλγορίθμου

Ο κώδικας σε C για το σειριακό αλγόριθμο για κάθε επανάληψη για τη μέθοδο CG είναι ο παρακάτω:

```
spm_csr_mulv(A, p, z); //z = Ap


rr = cblas_ddot(N, rv,  rv);
zp = cblas_ddot(N, zv,  pv);
alpha = rr / zp;
cblas_daxpy(N, alpha, pv,  xv);
cblas_daxpy(N, -alpha, zv,  rv); //r = r - alpha*A*p
rr_new = cblas_ddot(N, rv,  rv);
beta = rr_new / rr;
cblas_axpyz(N,beta, rv, pv);
```

Κεφάλαιο 2ο: Παράλληλη Υλοποίηση σε συστήματα κατανεμημένης μνήμης

Message Passing Interface(MPI)

Το MPI είναι ένα πρωτόκολλο επικοινωνίας ανεξάρτητο γλώσσας προγραμματισμού που χρησιμοποιείται στον προγραμματισμό σε παράλληλες εφαρμογές. Υποστηρίζονται τόσο καθολική επικοινωνία όσο και επικοινωνία σημείου προς σημείο. Οι στόχοι αυτής της προγραμματιστικής διεπαφής είναι η επίδοση η επεκτασιμότητα κι η φορητοτητα. Το MPI είναι το κατά κόρον χρησιμοποιούμενο μοντέλο προγραμματισμού στα συστήματα υψηλής επίδοσης σήμερα.

Χρησιμοποιείται σε συστήματα που εμπλέκονται πολλοί υπολογιστές δηλαδή συστήματα κατανεμημένης μνήμης. Το MPI-1 μοντέλο δεν έχει καθόλου υποστήριξη για in-node επικοινωνία ενώ το MPI-2 έχει μια υποτυπώδη τέτοια δυνατότητα. Χρησιμοποιείται κυρίως σε συστήματα μνήμης NUMA.

Παρόλο που το MPI ανήκει στο επίπεδο 5 του πρωτοκόλλου OSI μερικές υλοποιήσεις καλύπτουν κι άλλα επίπεδα αυτού του πρωτοκόλλου με sockets και το tcp στο επίπεδο μεταφοράς.

Οι περισσότερες υλοποιήσεις του MPI αποτελούνται από ρουτίνες κατευθείαν καλούμενες από γλώσσες όπως h C, η FORTRAN κι η C++ ,γλώσσες δηλαδή οι οποίες είναι σε θέση να επικοινωνούν με αυτή τη βιβλιοθήκη κι επίσης η C#,Java,Python. Τα πλεονεκτήματα του MPI απέναντι σε παλιότερα μοντέλα περάσματος μηνυμάτων είναι η φορητοτητα (επειδή το MPI έχει υλοποιηθεί για κάθε σχεδόν τύπο αρχιτεκτονικής κατανεμημένης μνήμης) καθώς κι η ταχύτητα(επειδή κάθε υλοποίηση είναι βελτιστοποιημένη για το υλικό πάνω στο οποίο τρέχει).

Το MPI Interface προσφέρει την απαραίτητη εικονική τοπολογία ,συγχρονισμό και λειτουργικότητα επικοινωνίας μεταξύ των διεργασιών(οι οποίες έχουν αντιστοιχηθεί σε κόμβους,εξυπηρετητές η στιγμιότυπα υπολογιστών) σε τρόπο ανεξάρτητο από γλώσσα προγραμματισμού. Το MPI πάντα δουλεύει με την έννοια της διεργασίας αλλά είναι σύνηθες για τους προγραμματιστές να εννοούν στη θέση της εννοια3ς διεργασία την έννοια επεξεργαστής. Τυπικά ένας επεξεργαστής ένας πυρήνας σε ένα πολυπύρηνο σύστημα θα έχει στην ευθύνη του μια ακριβώς διεργασία. Αυτή η ανάθεση διεργασιών σε επεξεργαστές η πυρήνες γίνεται κατά το χρόνο εκτέλεσης μέσω του Mpirun ή του mpiexec.

Με βάση τα παραπάνω έχουμε κατασκευάσει κώδικα ο οποίος διαχειρίζεται την επίλυση πολύ μεγάλων αραιών συστημάτων. Το σημείο που εστιάζουμε είναι ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα. Στοχεύοντας στην αποδοτική υλοποίηση του πολλαπλασιασμού θα σχεδιάσουμε εκτέλεση του σε συστοιχία επεξεργαστών-κόμβων διασυνδεδεμένων μεταξύ τους. Το προγραμματιστικό μοντέλο που θα χρησιμοποιηθεί είναι το Message Passing Interface(MPI).

Το MPI είναι ένα προγραμματιστικό μοντέλο, που χρησιμοποιείται σε συστήματα κατανεμημένης μνήμη,δηλαδή σε συστοιχίες υπολογιστών που συνδέονται μεταξύ τους με ένα δίκτυο διασύνδεσης. Το MPI περιέχει εντολές ανταλλαγής μηνυμάτων και συγχρονισμού μεταξύ των διασυνδεδεμένων υπολογιστών.

Θα σπάσουμε το πρόβλημα δηλαδή σε υποπροβληματα μικρότερου μεγέθους. Η φιλοσοφία είναι ίδια με αυτήν του “διαίρει και βασίλευε”.Χωρίζουμε τον πίνακα Α σε κομμάτια σύμφωνα με τον

αριθμό των γραμμών και όλα τα διανύσματα της μεθόδου σύγκλισης σε υποδιανύσματα. Έτσι κάθε διεργασία στην παράλληλη υλοποίηση έχει έναν τοπικό πίνακα A_local και διανύσματα τοπικά.

Σε κάθε επανάληψη ο πολλαπλασιασμός δίνει ως αποτέλεσμα ένα τοπικό αποτέλεσμα. Οι διεργασίες ανταλλάσσουν η μια με την άλλη τα τοπικά τους διανύσματα για να φτιάξουν τα καθολικά. Το βασικό διάνυσμα που μας ενδιαφέρει είναι το p το οποίο το χρειαζόμαστε ολόκληρο για να γίνει ο πολλαπλασιασμός. Έτσι στο τέλος κάθε επανάληψης η κάθε διεργασία μαζεύει από όλες τις άλλες τα τοπικά διανύσματα p και συνθέτει το καθολικό p.

Άρα από τη σκοπιά του χρόνου έχουμε:

i)το χρόνο του πολλαπλασιασμού SpMV
ii)τους χρόνους εσωτερικών γινομένων και προσθαφαιρέσεων πάνω στα διανύσματα
iii)το χρόνο της Reduce για τα alpha και beta
iv)το χρόνο της All_Gather για την ανταλλαγή των p_local

Αλγόριθμος παραλληλης υλοποίησης σε MPI

Ο κώδικας με τη χρήση του MPI είναι ο εξής:

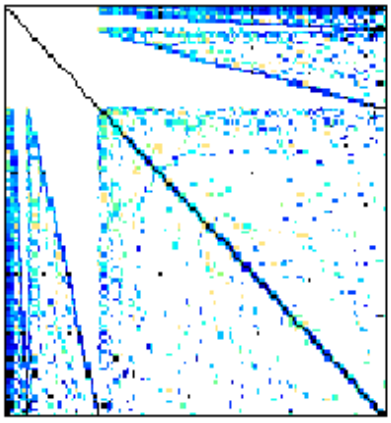
```
if(NO_SYMMETRY == 1){
        spm_csr_mulv(Al, p, zl,row_ptr_off_myrank);
}

else{
        spm_csr_mulv(Al, p, z,row_ptr_off_myrank);

        MPI_Allreduce(zv, zhv, N, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

        for(i=0; i<l_nr_rows; i++)
                zlv[i]=zhv[row_ptr_off[myrank]+i];
}


lrr = cblas_ddot(l_nr_rows, rlv,  rlv); //rl*rl

MPI_Allreduce(&lrr, &rr, 1, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);
//all     locals lrr to rr

lzp = cblas_ddot(l_nr_rows, zlv,  plv);
MPI_Allreduce(&lzp, &zp, 1, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

alpha = rr / zp;

cblas_daxpy(l_nr_rows, alpha, plv,  xlv); //x = x + alpha*p
cblas_daxpy(l_nr_rows, -alpha, zlv,  rlv); //r = r - alpha*A*p

lrr_new = cblas_ddot(l_nr_rows, rlv,  rlv);

MPI_Allreduce(&lrr_new, rr_new,1,MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

beta = rr_new / rr;


cblas_axpyz(l_nr_rows,beta, rlv, plv); // p = r + beta*p

MPI_Allgatherv(plv,  l_nr_rows,  MPI_SPM_VALUE,  pv,  row_sendcnts,     row_ptr_off,
MPI_SPM_VALUE, MPI_COMM_WORLD);
```

Κεφάλαιο 3ο : Συμμετρικότητα Αραιού Πίνακα

Στην εργασία αυτή πειραματιζόμαστε κυρίως με συμμετρικούς πίνακες. Μέχρι στιγμής όμως δεν έχουμε αξιοποιήσει αυτήν τους την ιδιότητα.

Συμμετρικός είναι ο πίνακας ο οποίος αν πάρουμε τον ανάστροφο του είναι ο ίδιος με τον αρχικό. Δηλαδή:A^T=A. Με απλά λόγια αν ο πίνακας έχει στοιχείο στις συντεταγμένες (I,j) θα έχει το ίδιο στοιχείο στις συντεταγμένες (j,i).Αξιοποιώντας αυτήν την ιδιότητα μπορούμε να γλιτώσουμε χώρο κατά την αποθήκευση του,αφού η πληροφορία που περιέχεται στο κάτω τριγωνικό μέρος του (η ισοδύναμα στο άνω τριγωνικό μέρος του) αρκεί για την αναπαράσταση του. Δηλαδή αν ξέρουμε το κάτω τριγωνικό μέρος του μπορούμε να ανακατασκευάσουμε το άνω κι αντίστροφα. Βεβαίως θα πρέπει να γνωρίζουμε τα στοιχεία της διαγωνίου του.

Παρακάτω φαίνεται μια εικόνα ενός συμμετρικού αραιού πίνακα:



Αλγόριθμος Συμμετρικού Πολλαπλασιασμού

Εκτός από τη μείωση του κόστους αποθήκευσης για έναν συμμετρικό πίνακα,θέτουμε το εξής ερώτημα:Μήπως μπορούμε να έχουμε και μείωση του χρόνου εκτέλεσης;Γι'αυτό το λόγο αλλάζουμε τον πυρήνα του πολλαπλασιασμού και γίνεται όπως φαίνεται παρακάτω.

```
for (i = 0; i < nr_rows; i++) {
        _y = 0.0;
        for (j = row_ptr[i]; j < row_ptr[i+1]; j++){
                _y += values[j]* x[col_ind[j]];
                if(i != col_ind[j])
                        y[col_ind[j]] += x[i] * values[j];
        }
        y[i]+=_y;
}
```

Βλέπουμε ότι η διαφορά σε σχέση με τον μη συμμετρικό πολλαπλασιασμό είναι ότι για κάθε στοιχείο του πίνακα που πολλαπλασιάζουμε,ελέγχουμε αν είναι πάνω στη διαγώνιο, κι αν δεν είναι τότε κάνουμε έναν ακόμη πολλαπλασιασμό για το συμμετρικό του στοιχείο.

Συνεχίζοντας την εξερεύνηση μας πάνω στον συμμετρικό πολλαπλασιασμό, υλοποιούμε και μια άλλη εκδοχή. Για να αποφύγουμε αυτόν το διαρκή έλεγχο που γίνεται για κάθε στοιχείο,σπάμε τον πίνακα στο κάτω τριγωνικό μέρος του και στη διαγώνιο του. Δηλαδή εκτελούμε τον πολλαπλασιασμό σαν να μην είχε ο πίνακας διαγώνιο,ύστερα εκτελούμε ένα εσωτερικό γινόμενο μεταξύ της διαγωνίου και του διανύσματος x και τέλος προσθέτουμε το αποτέλεσμα. Ο κώδικας γι αυτήν την υλοποίηση του πολλαπλασιασμού φαίνεται αμέσως παρακάτω.

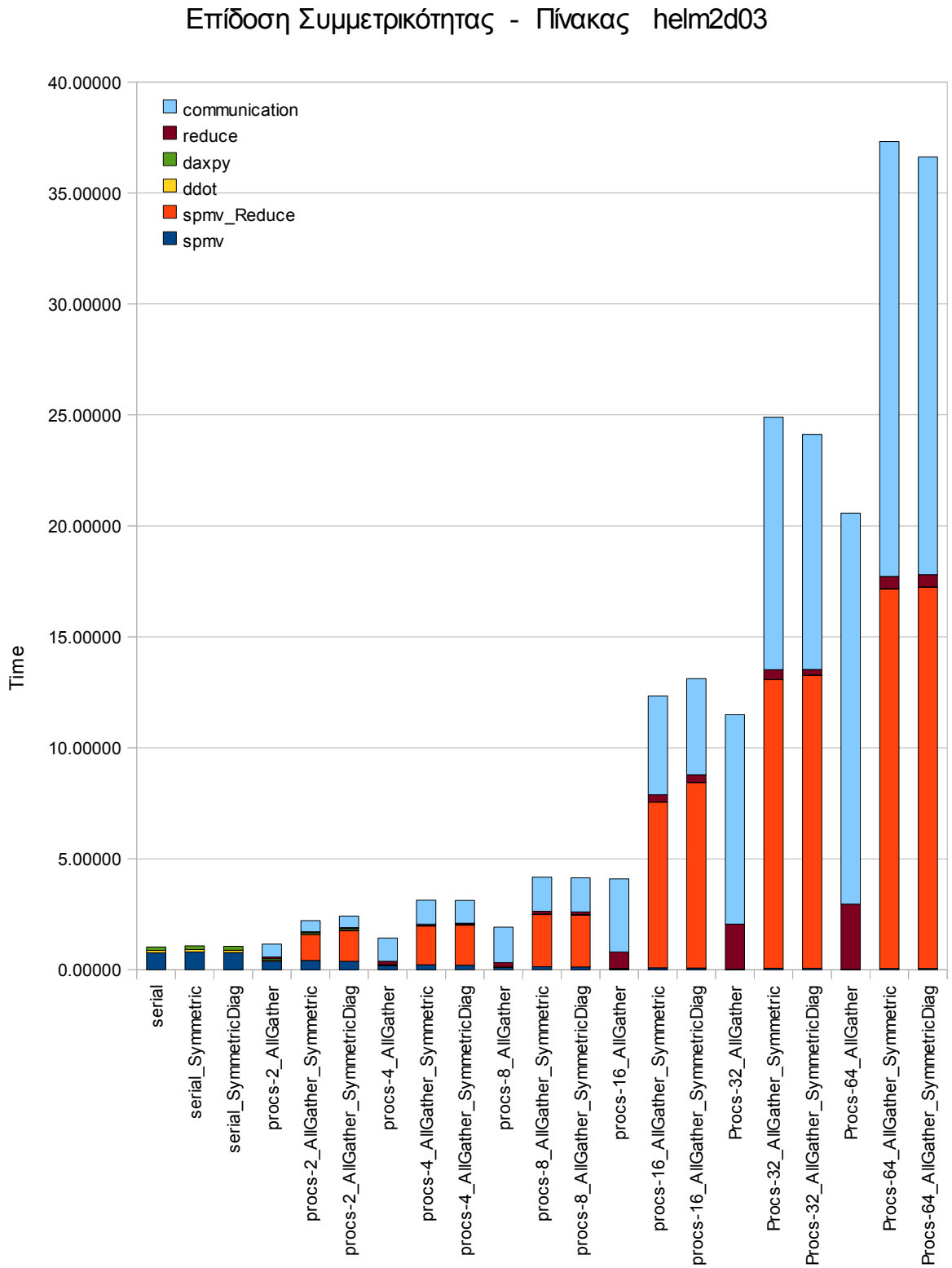
```
for (i = 0; i < nr_rows; i++) {
        _y = 0.0;
        for (j = row_ptr[i]; j < row_ptr[i+1]; j++){
                _y += values[j]* x[col_ind[j]];
                if(i != col_ind[j])
                        y[col_ind[j]] += x[i] * values[j];
        }
        y[i]+=_y;
}

for(j=0;j<nr_rows;j++){
        y[j]+=diag[j]*x[j];
}
```

Επίσης,για την υλοποίηση σε σύστημα κατανεμημένης μνήμης χρειαζόμαστε τον παρακάτω κώδικα,επειδή η κάθε διεργασία κάνει update ολόκληρο το διάνυσμα εξόδου, και χρειάζεται να γίνει πρόσθεση των επιμέρους καθολικών διανυσμάτων εξόδου:

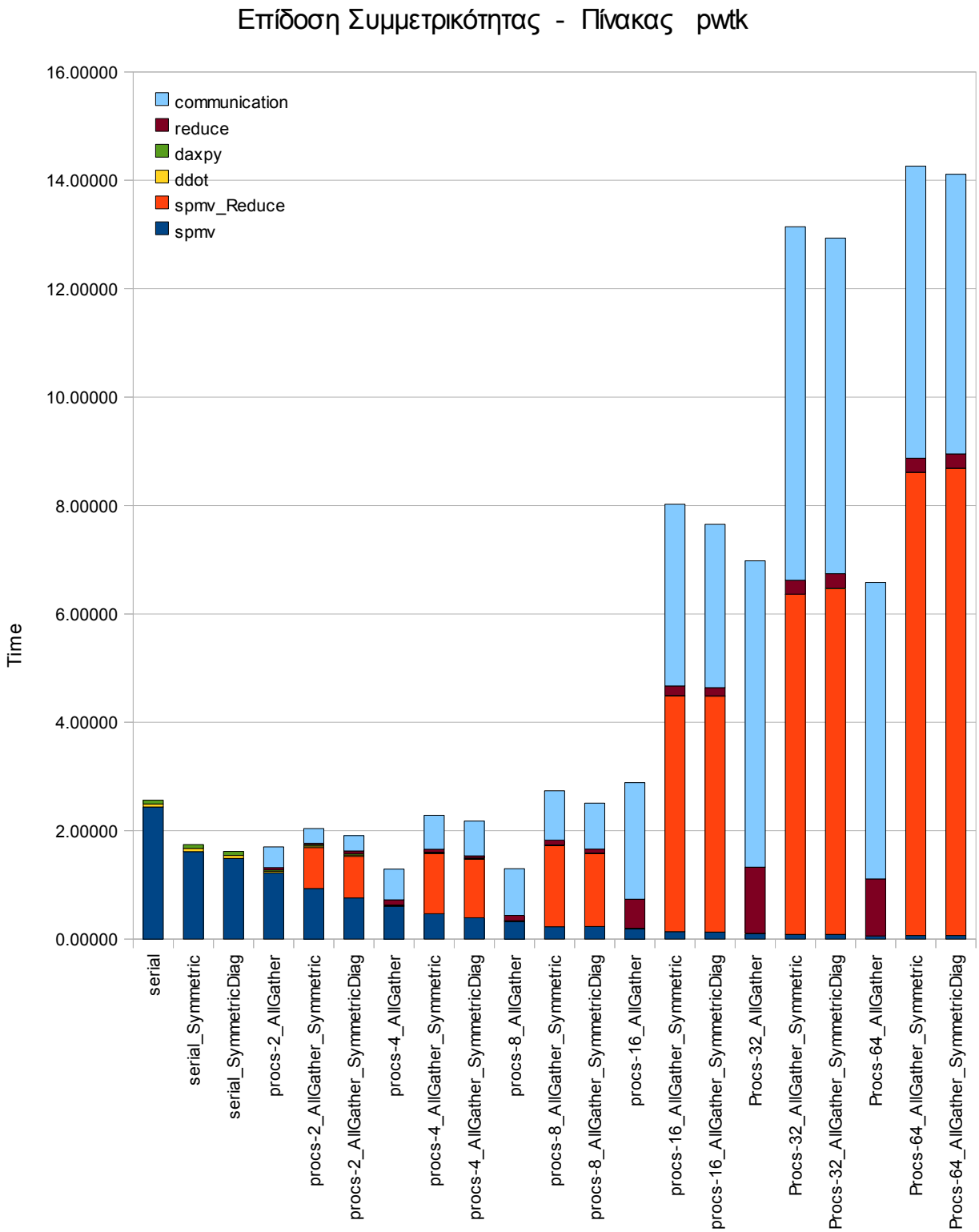
```
MPI_Allreduce(zv, zhv, N, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

for(i=0; i<l_nr_rows; i++)
        zlv[i]=zhv[row_ptr_off[myrank]+i];
```

Πειραματικό Μέρος Κεφαλαίου

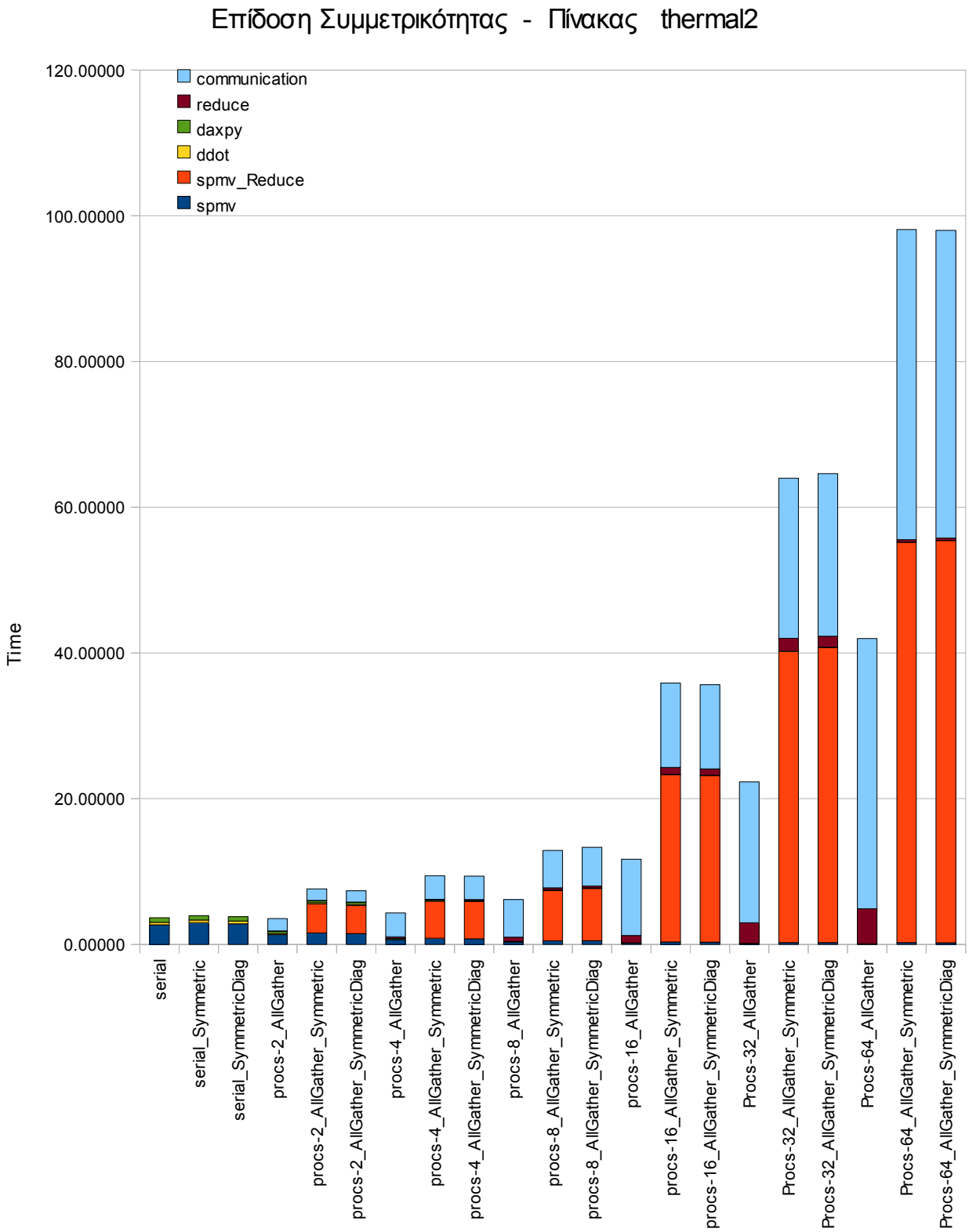
Παρακάτω φαίνονται τα διαγράμματα που πήραμε από την εκτέλεση των συμμετρικών εκδόσεων έναντι στη μη συμμετρική έκδοση του SpMV. Τρέξαμε την επαναληπτική μέθοδο σύγκλισης CG για 30 επαναλήψεις και για αριθμό διεργασιών 1,2,4,8,16,32,64 χρησιμοποιώντας 8 κόμβους του cluster. Όπου δεν δείχνεται στο διαγράμματα οι επιδόσεις για πολλές διεργασίες π.χ. 32 η 64 είναι γιατί η επίδοση ήταν πολύ κακή και πλέον δεν φαίνονταν καλά τα υπόλοιπα που μας ενδιαφέρουν.



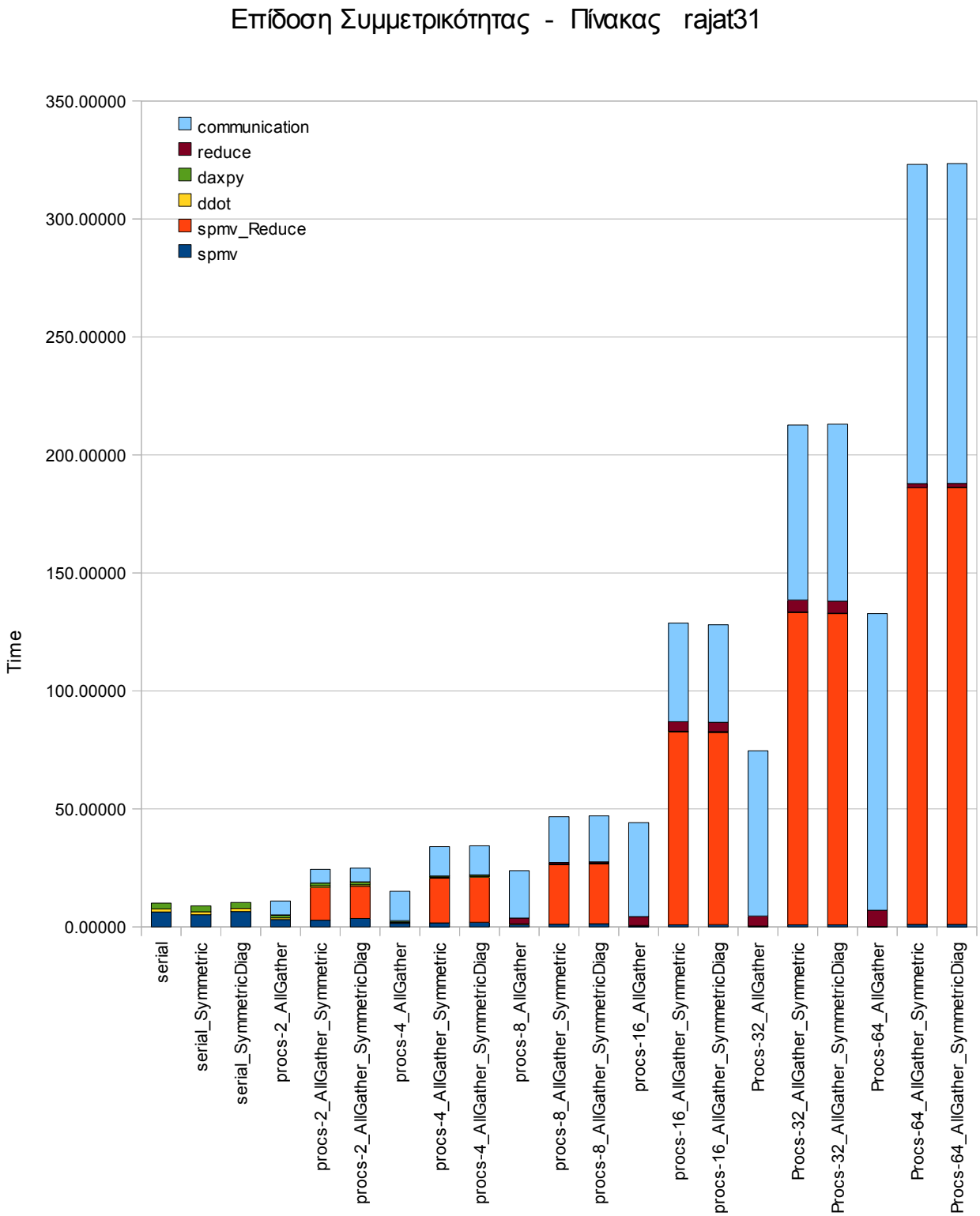
Σχήμα 3.1 - Επίδοση Συμμετρικότητας - Πίνακας helm2d03



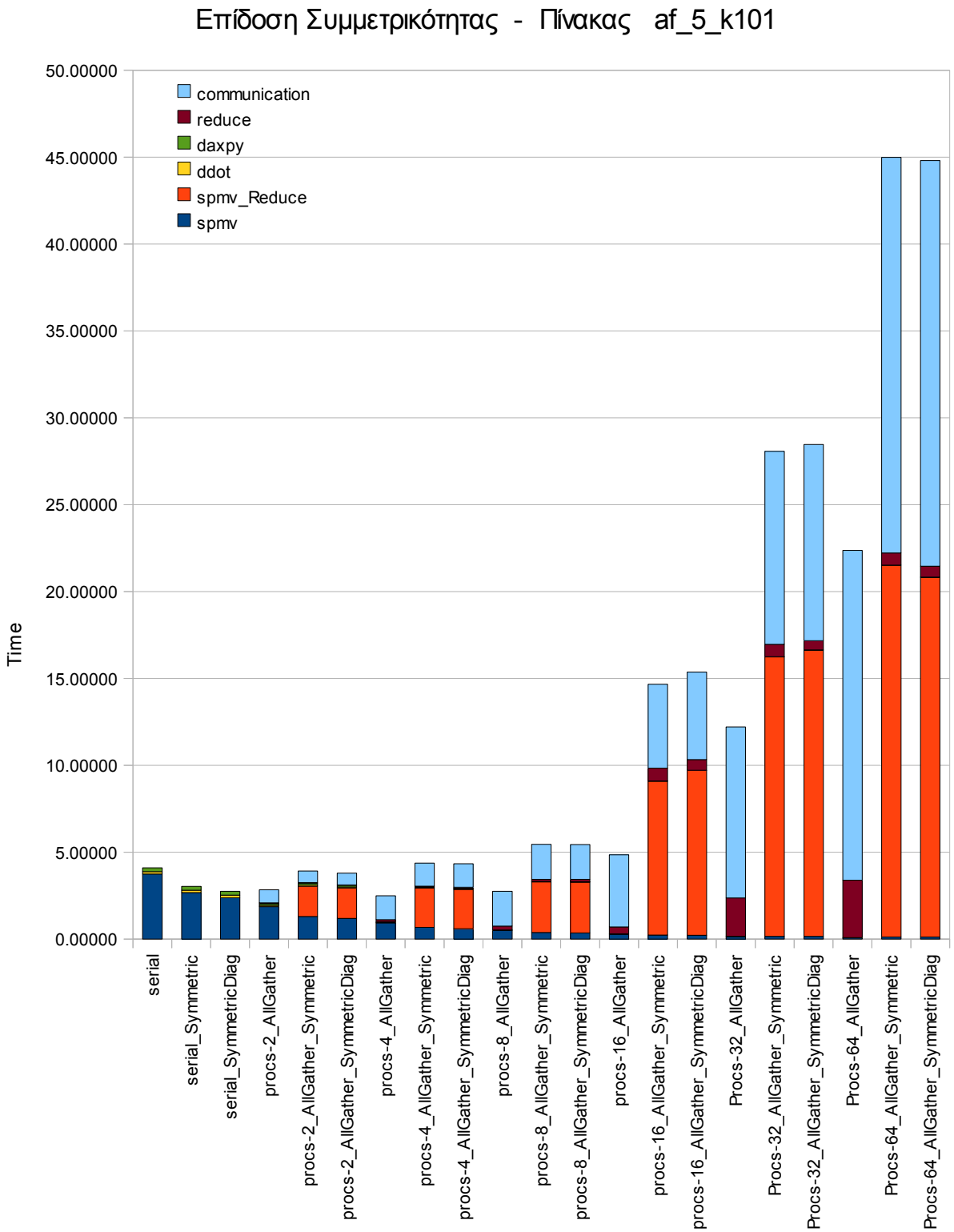
Σχήμα 3.2 - Επίδοση Συμμετρικότητας - Πίνακας pwtk



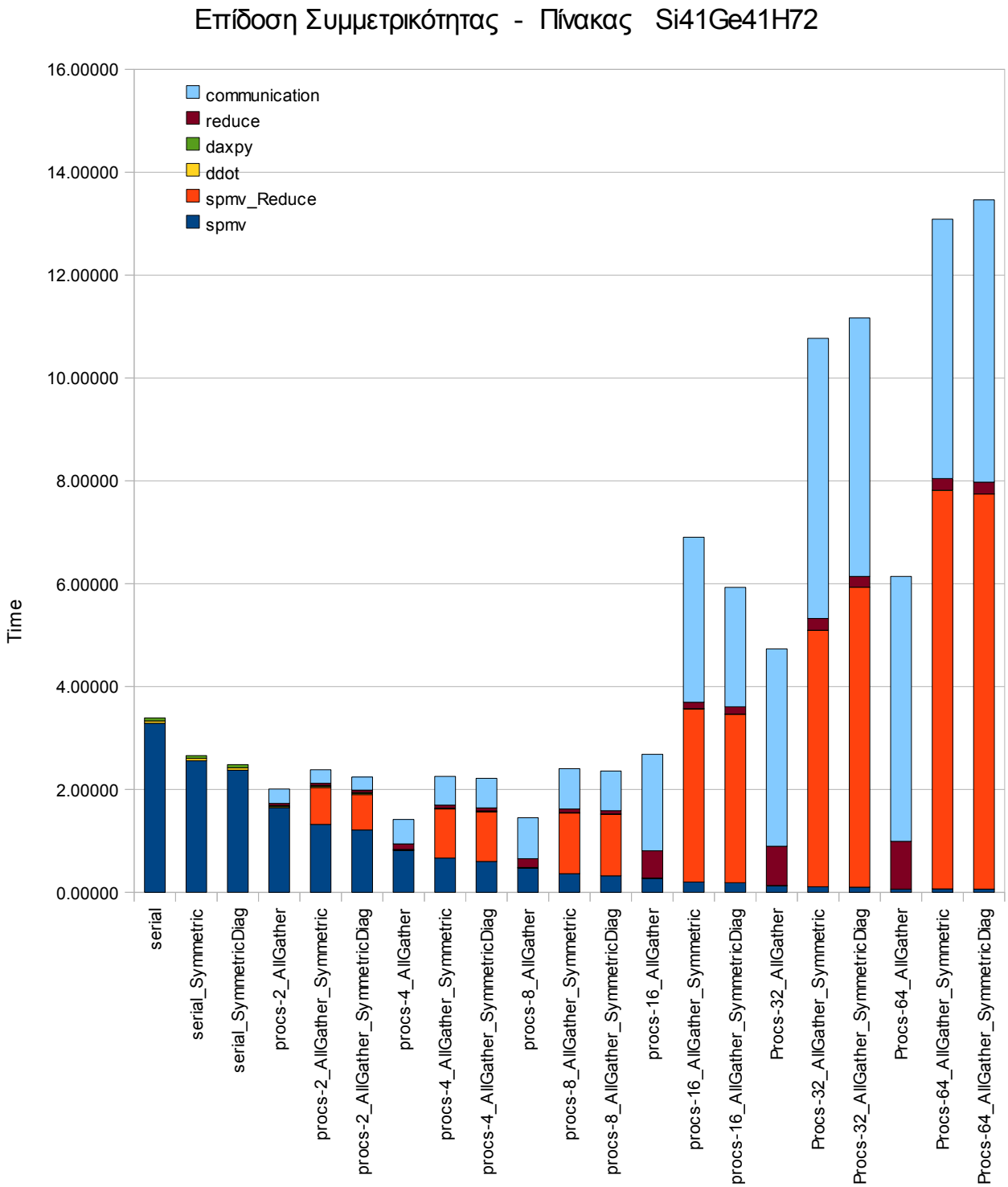
Σχήμα 3.3 - Επίδοση Συμμετρικότητας - Πίνακας thermal2



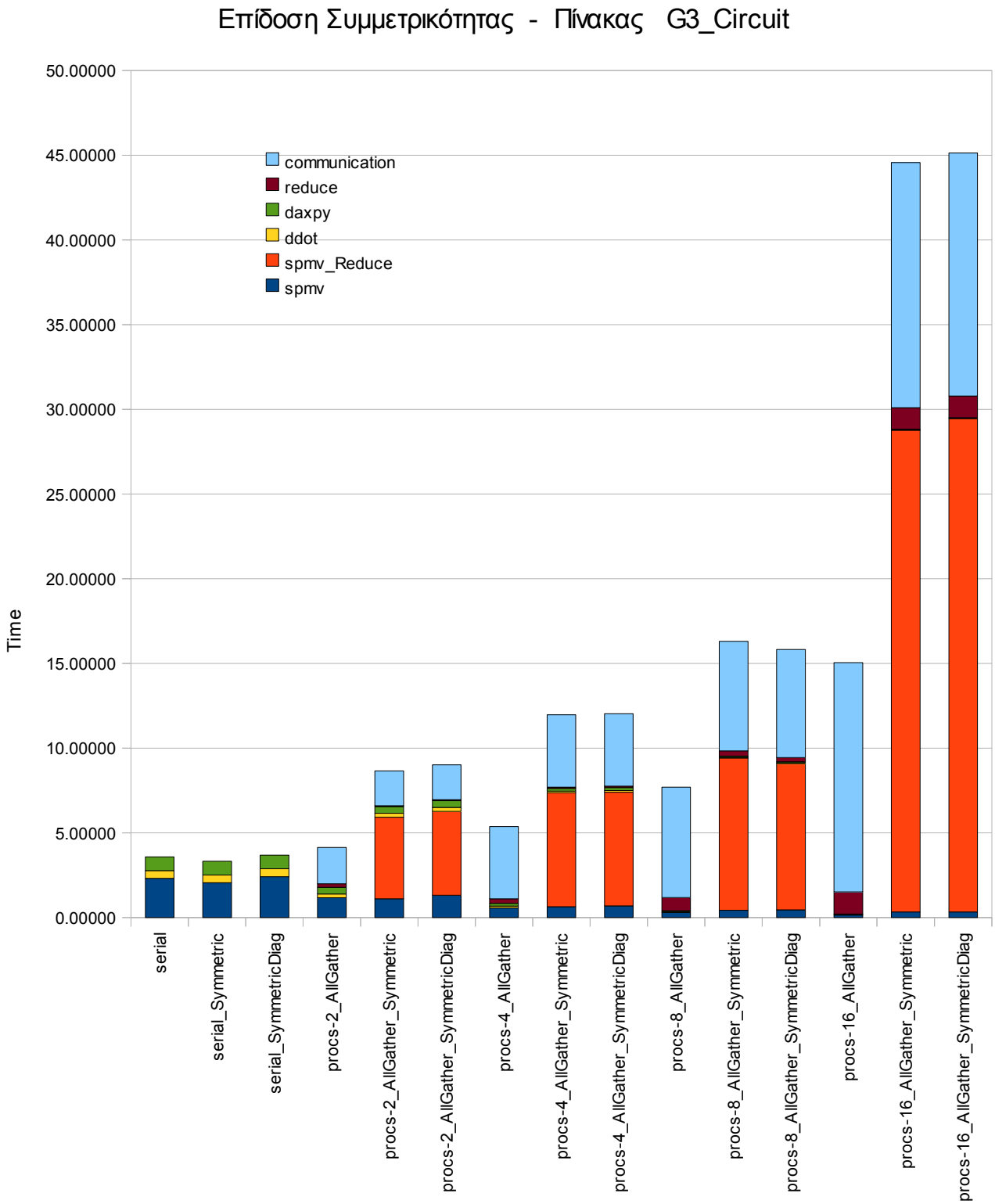
Σχήμα 3.4 - Επίδοση Συμμετρικότητας - Πίνακας rajat31



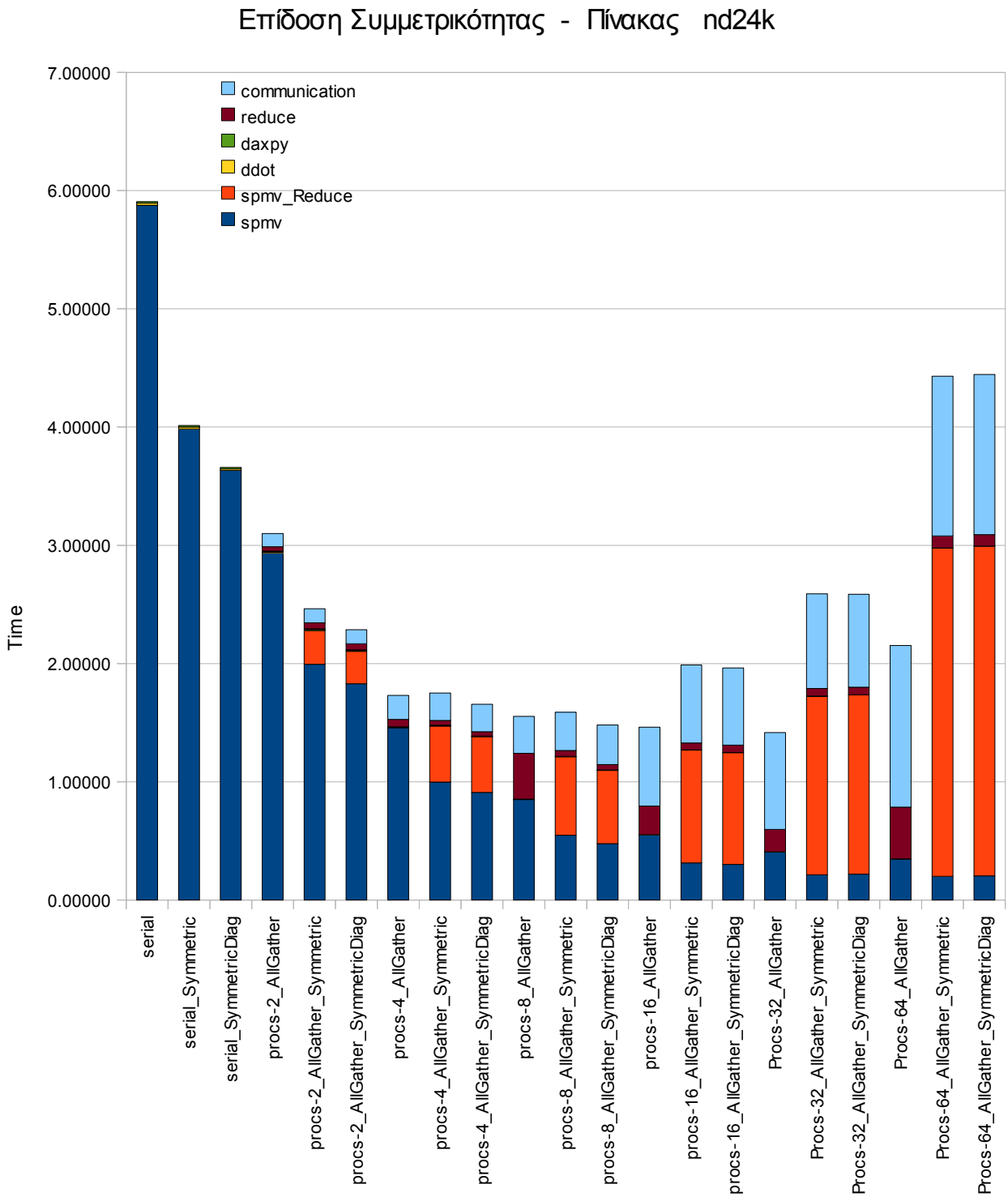
Σχήμα 3.5 - Επίδοση Συμμετρικότητας - Πίνακας af_5_k101



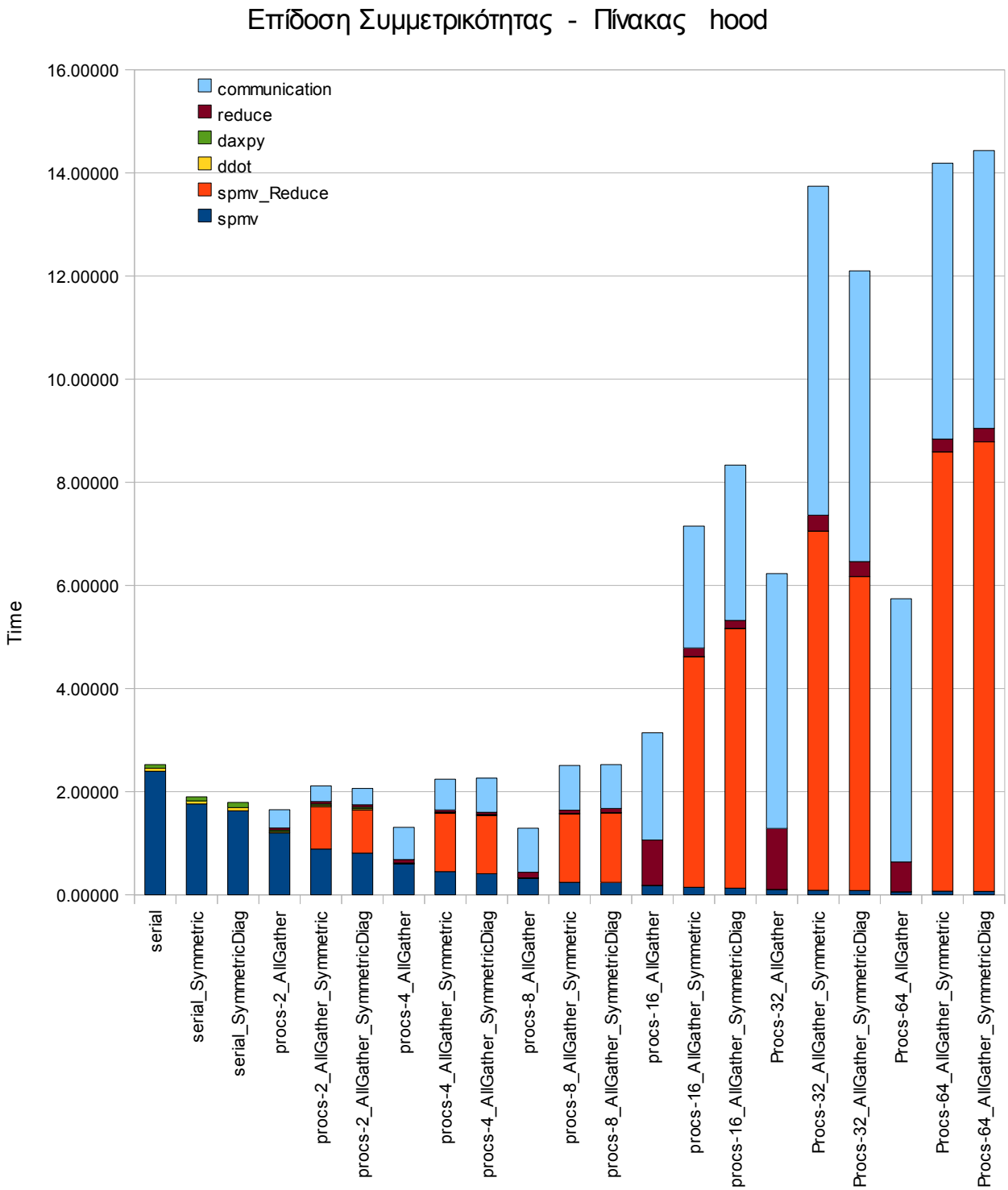
Σχήμα 3.6 - Επίδοση Συμμετρικότητας - Πίνακας Si41Ge41H72



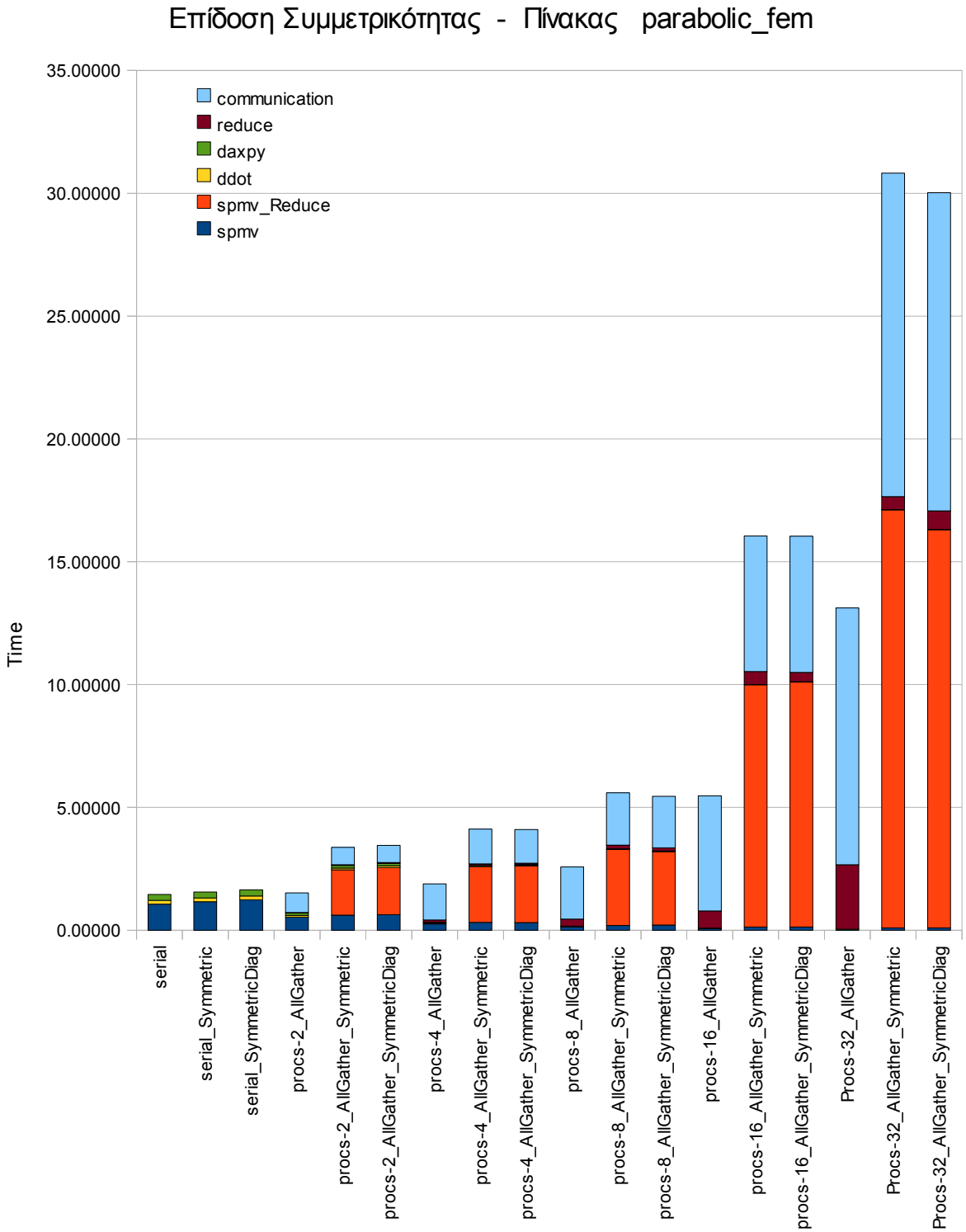
Σχήμα 3.7 - Επίδοση Συμμετρικότητας - Πίνακας G3_Circuit



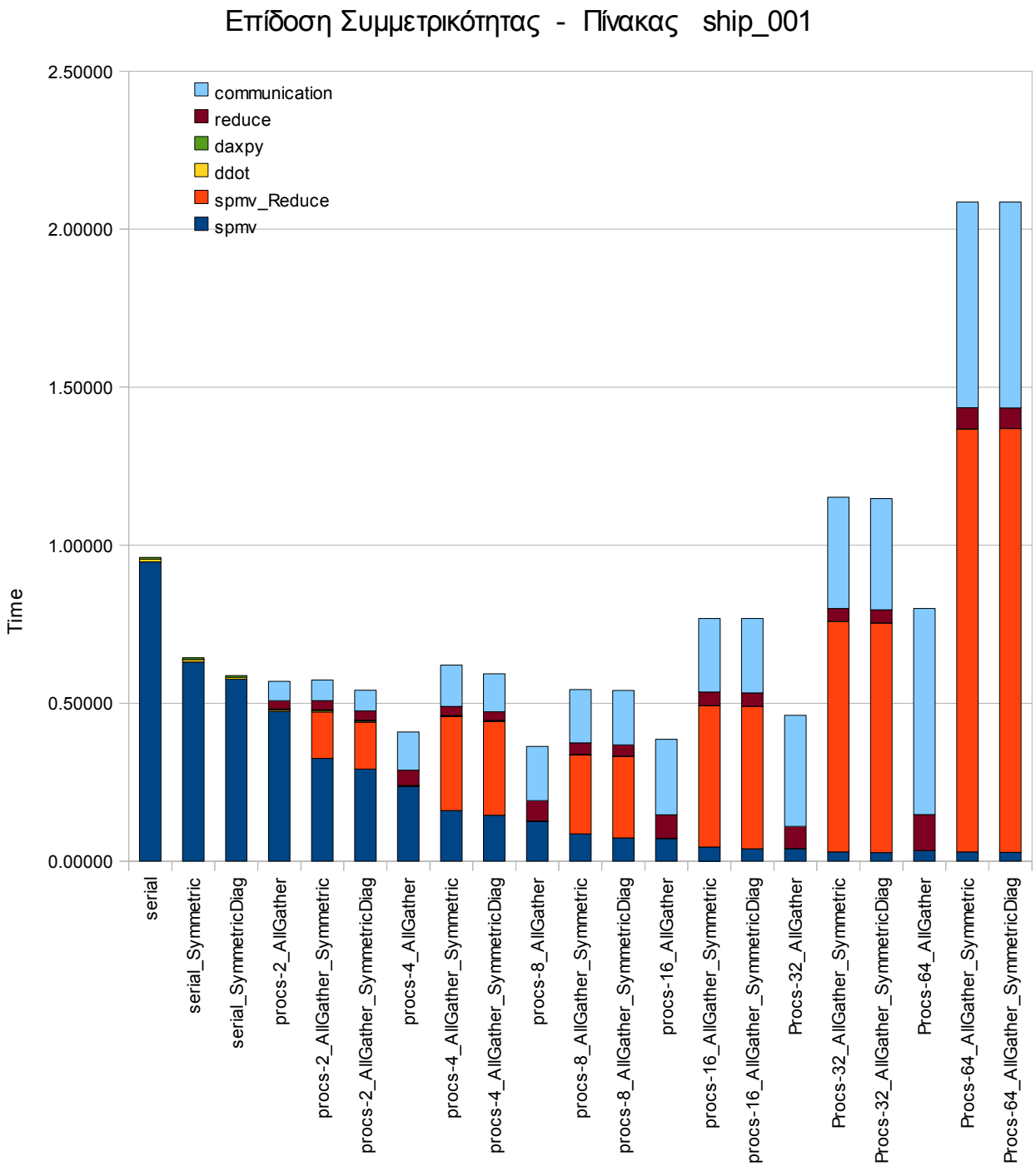
Σχήμα 3.8 - Επίδοση Συμμετρικότητας - Πίνακας nd24k



Σχήμα 3.9 - Επίδοση Συμμετρικότητας - Πίνακας hood

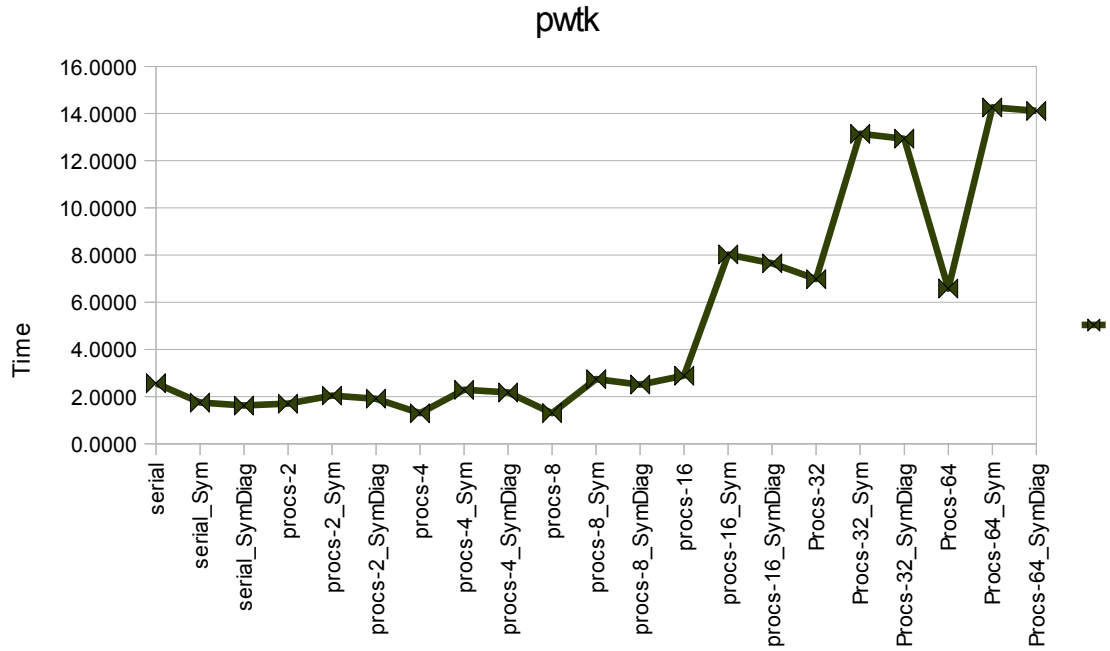


Σχήμα 3.10 - Επίδοση Συμμετρικότητας - Πίνακας parabolic

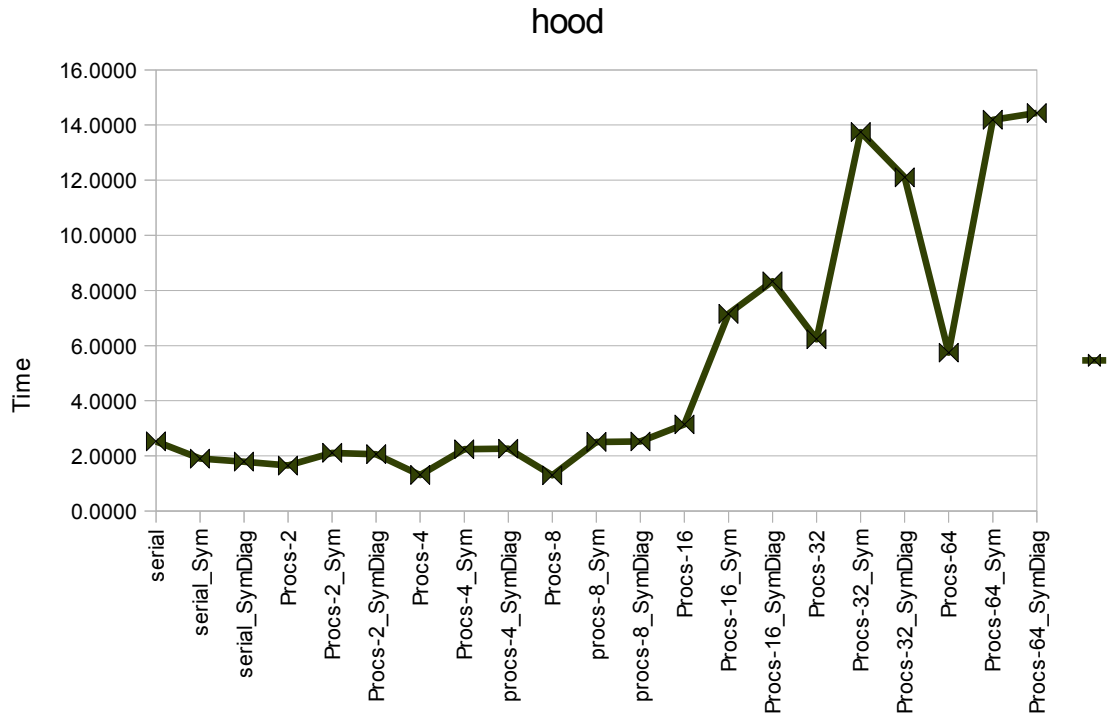


Σχήμα 3.11 - Επίδοση Συμμετρικότητας - Πίνακας ship

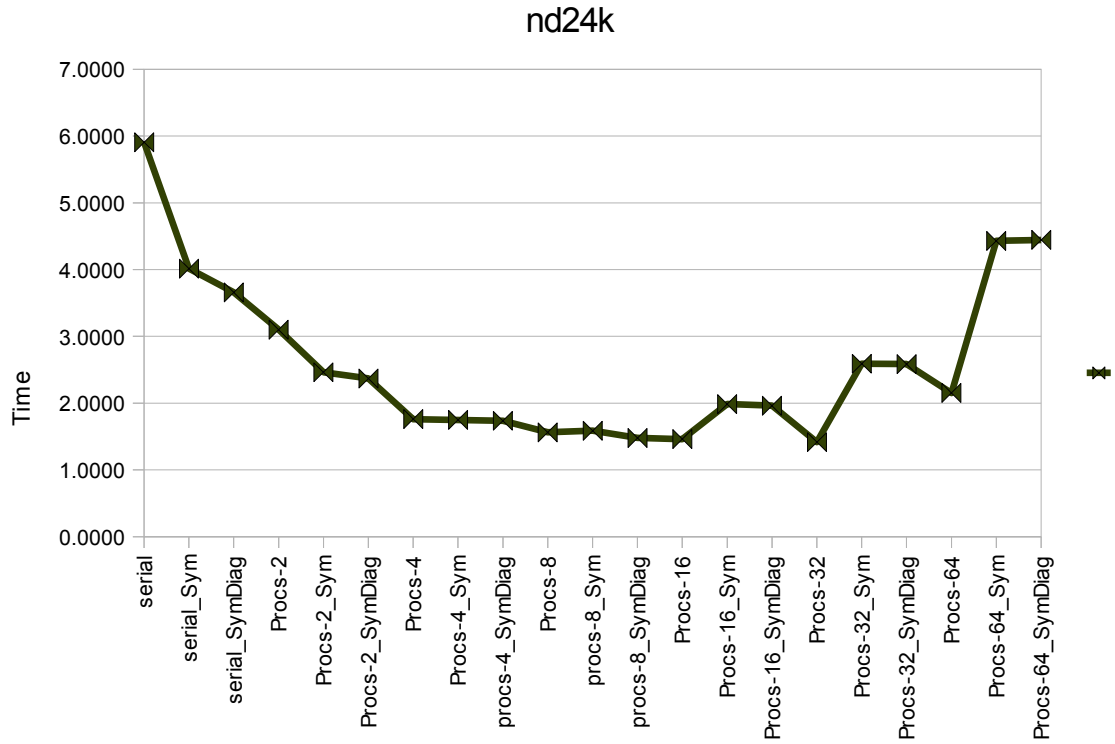
Παρακάτω φαίνεται ο συνολικός χρόνος της CG για 30 επαναλήψεις για όλους τους προηγούμενους πίνακες:



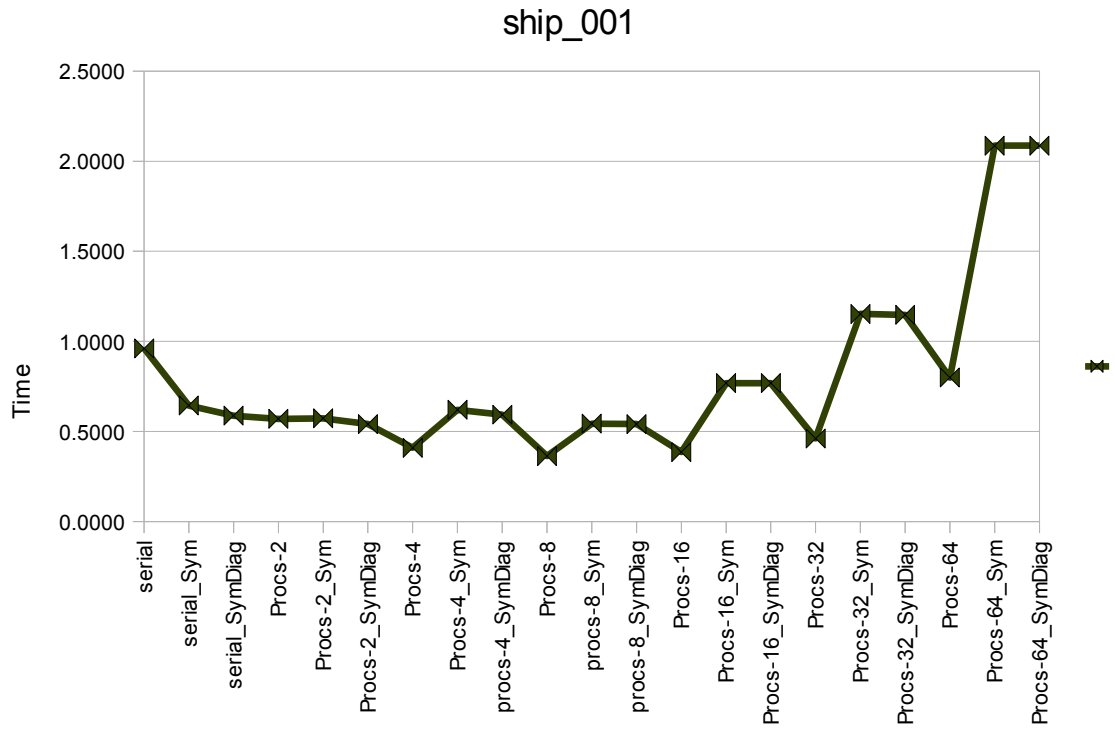
Σχήμα 3.12 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα pwtk



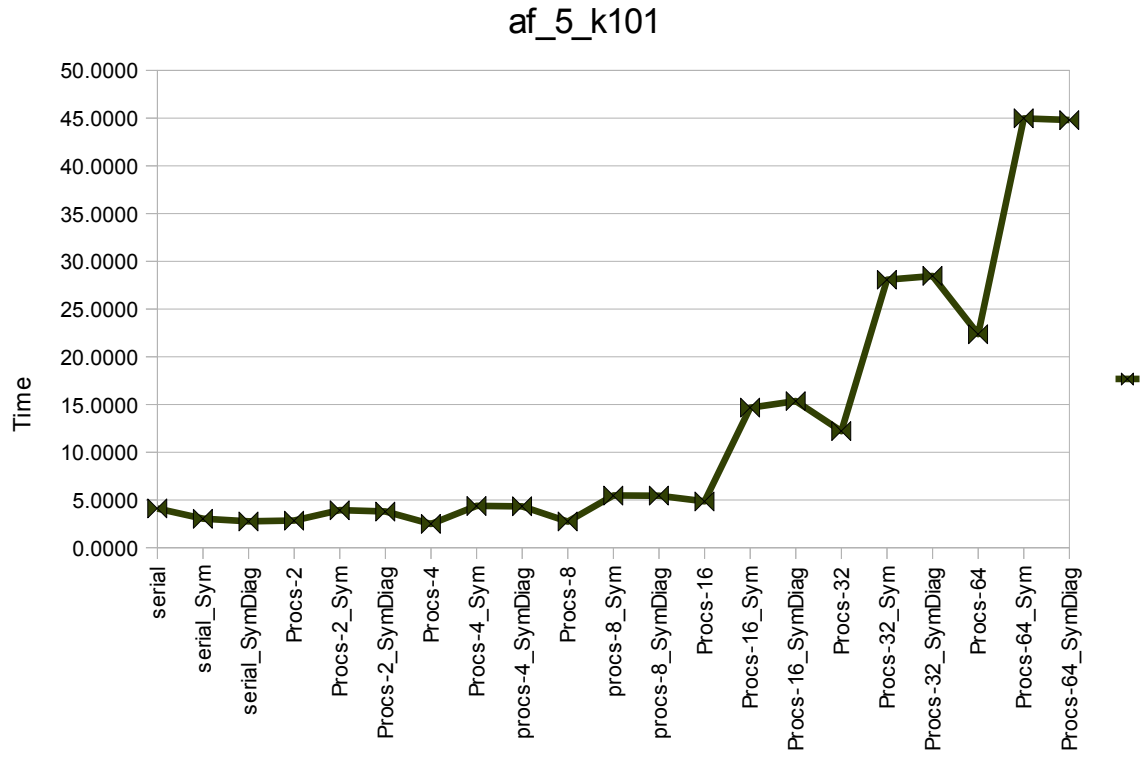
Σχήμα 3.13 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα hood



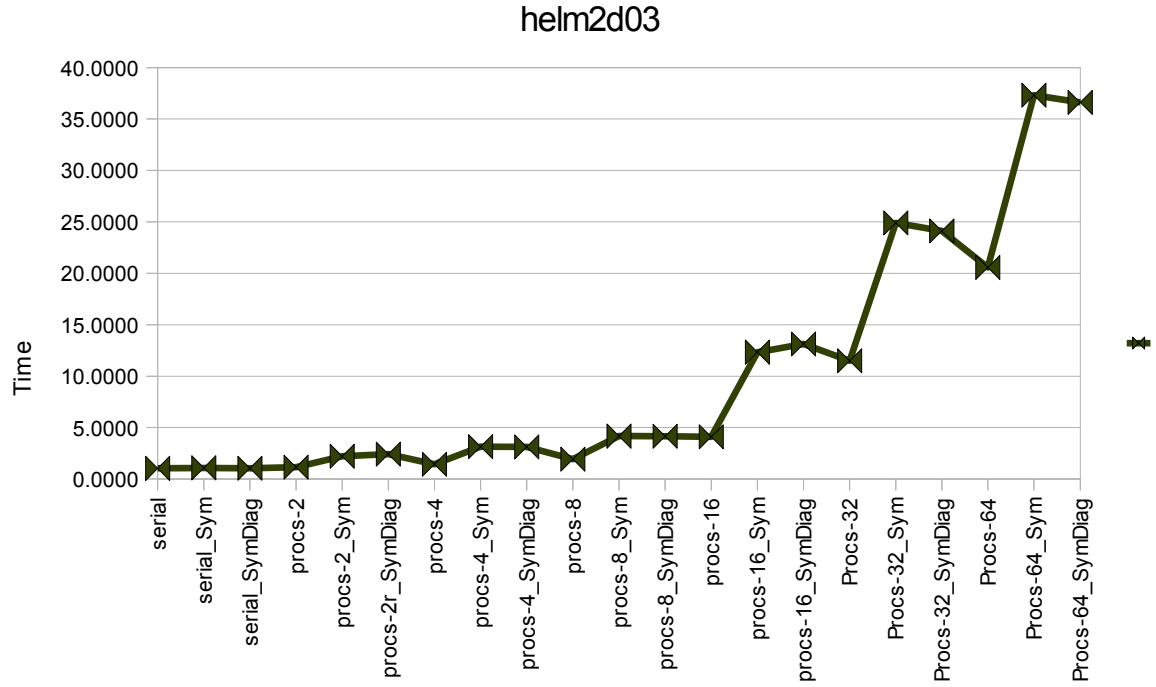
Σχήμα 3.14 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα nd24k



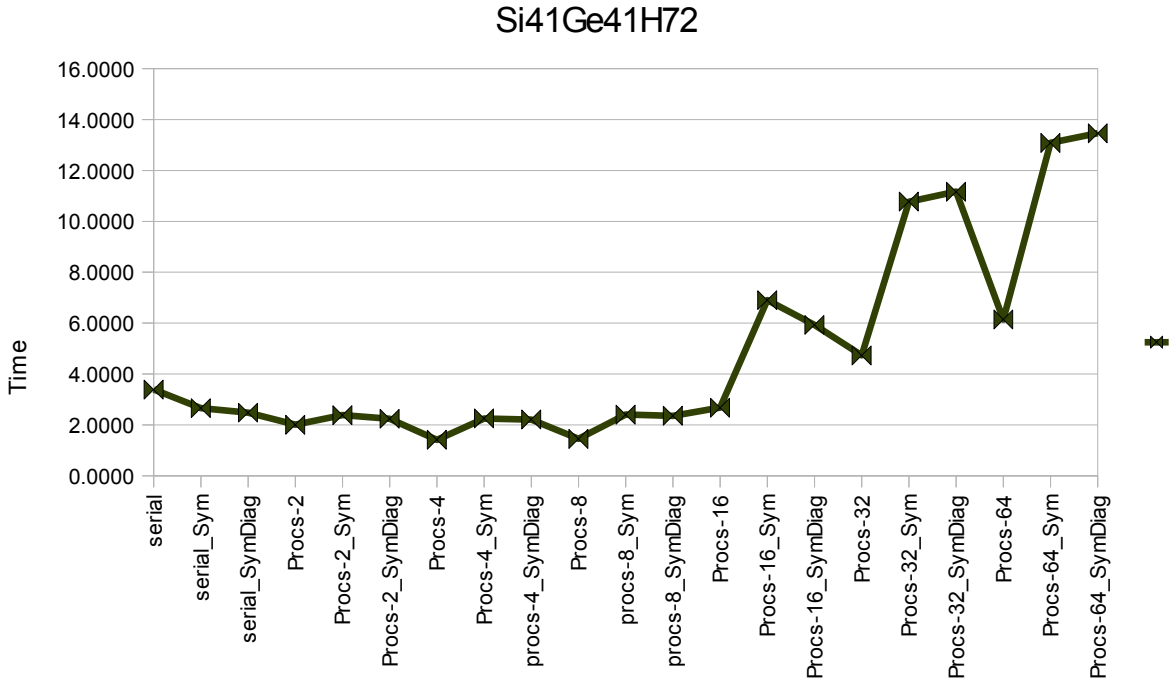
Σχήμα 3.15 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα ship_001



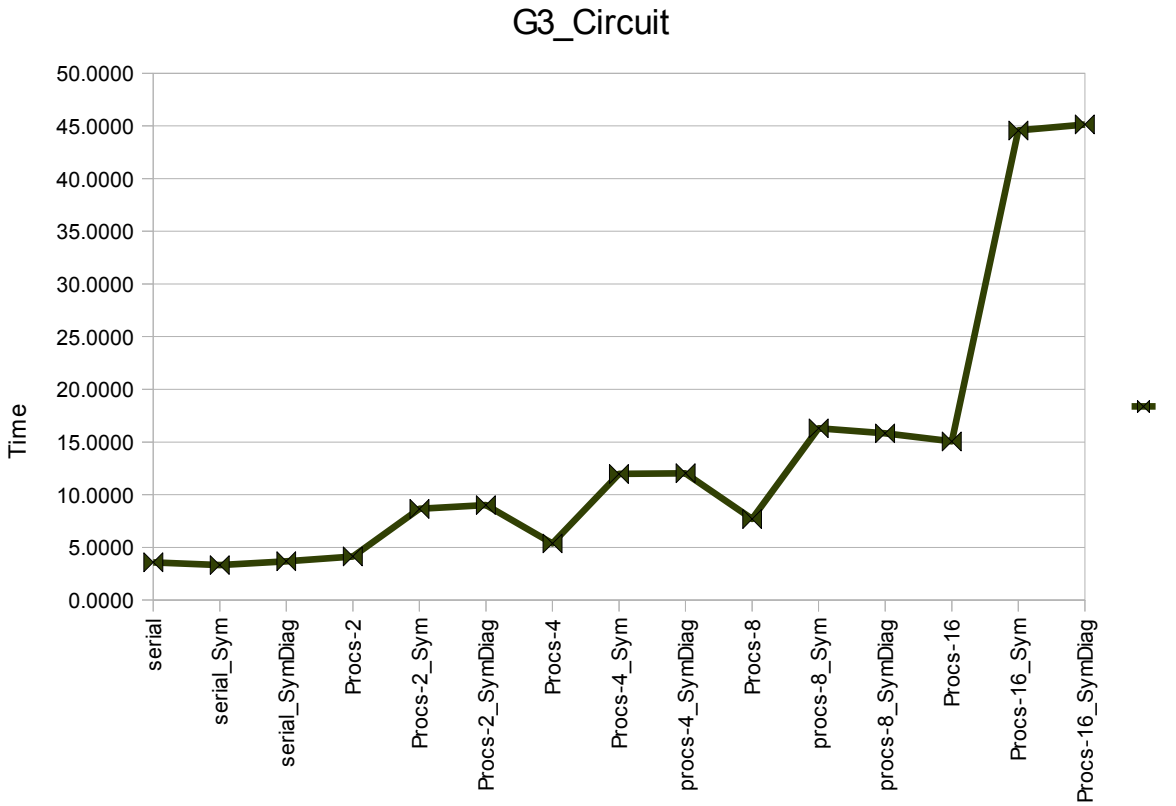
Σχήμα 3.16 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα af_5_k101



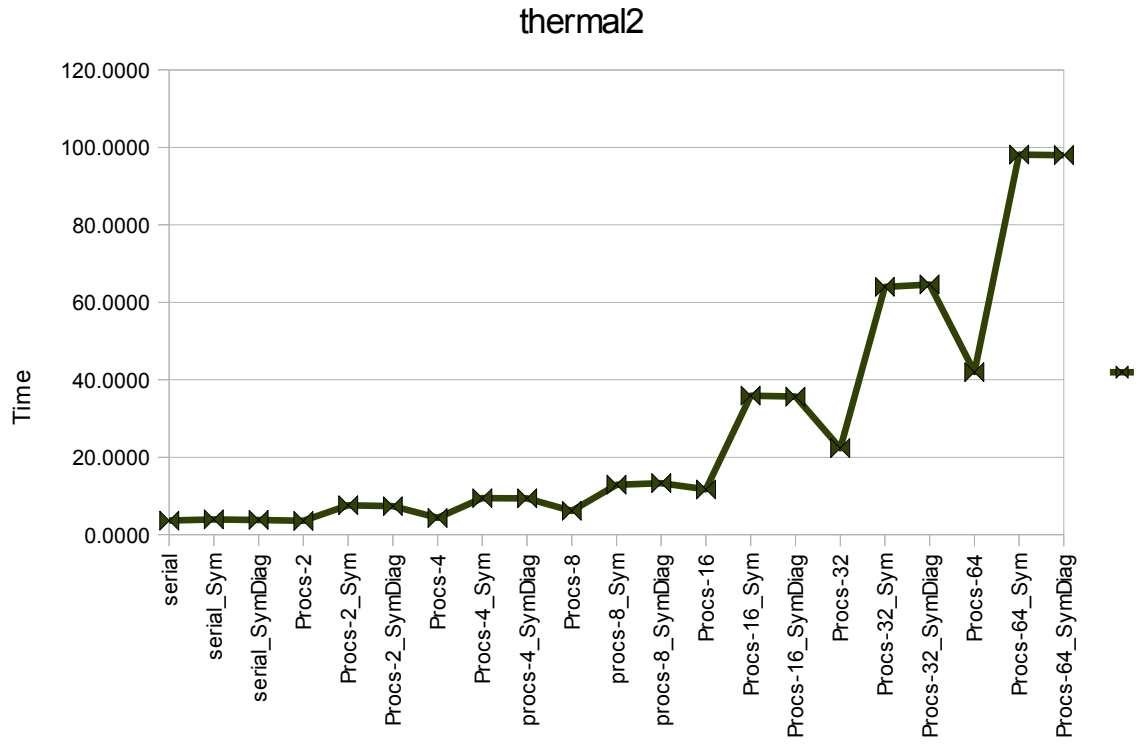
Σχήμα 3.17 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα helm2d03



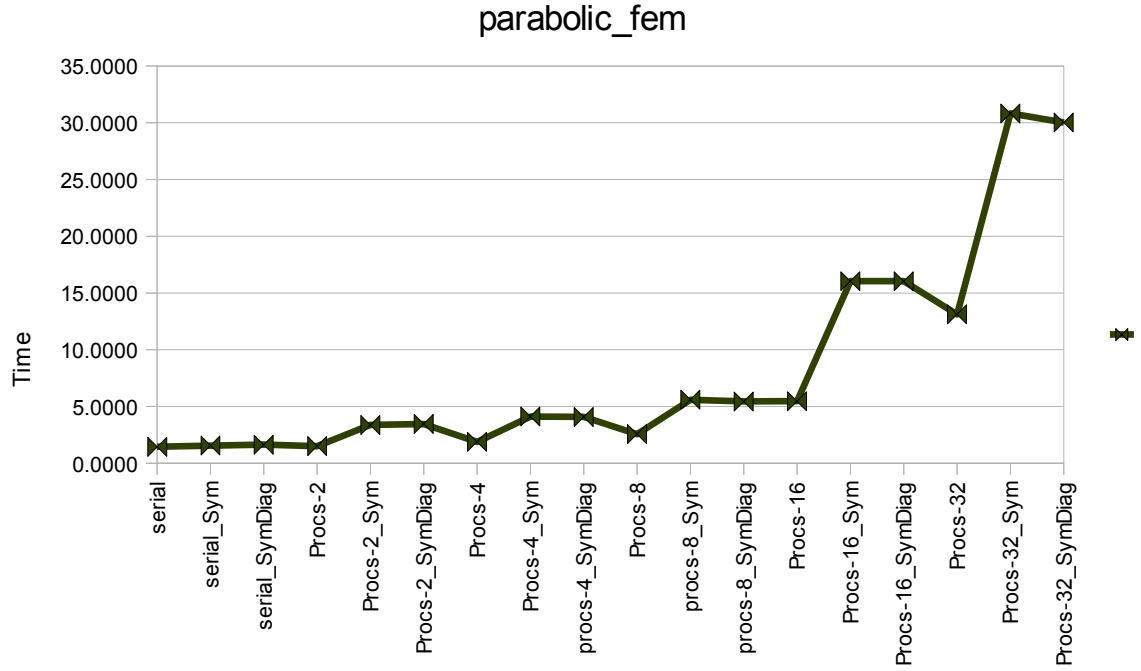
Σχήμα 3.18 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα Si41Ge41H72



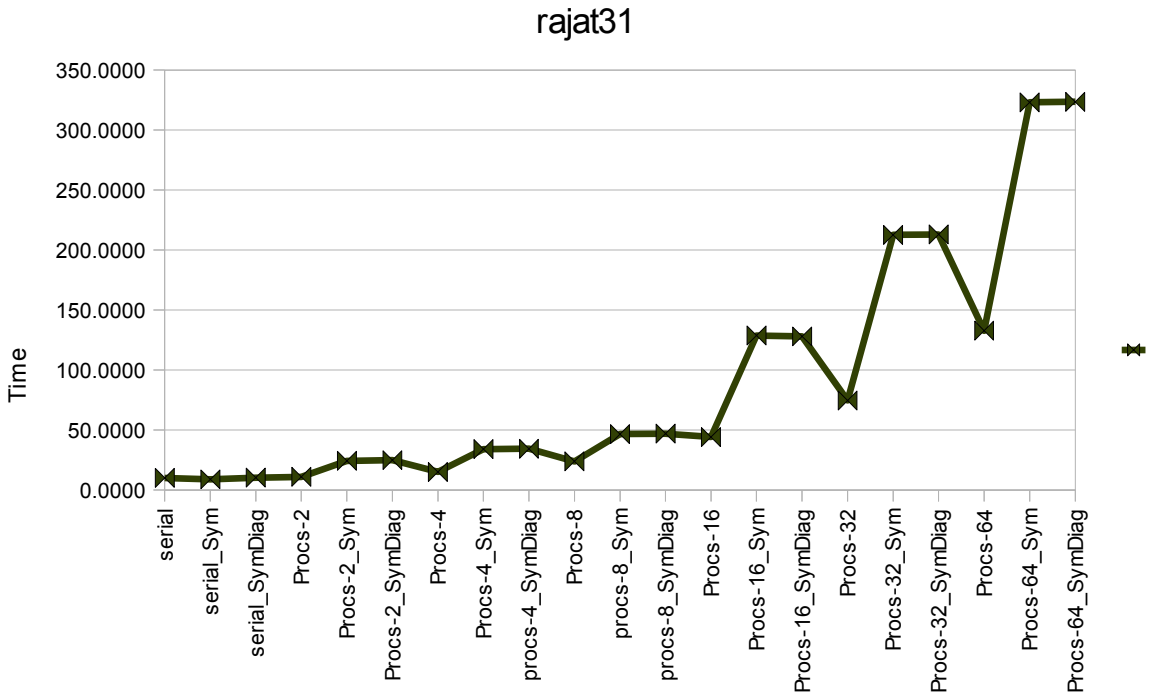
Σχήμα 3.19 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα G3_Circuit



Σχήμα 3.20 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα thermal2



Σχήμα 3.21 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα parabolic_fem



Σχήμα 3.22 - Συνολικός Χρόνος CG 30 επαναλήψεων για τον πίνακα rajat31

Συμπεράσματα για τη Συμμετρικότητα

Στα προηγούμενα διαγράμματα είδαμε την επίδοση των 2 συμμετρικών εκδόσεων που κατασκευάσαμε σε σύγκριση με τη συμμετρική υλοποίηση. Αυτό που παρατηρούμε ότι για όλους τους πίνακες εκτός από έναν ο οποίος είναι πολύ πυκνός οι συμμετρικές εκδόσεις έχουν σαφώς χειρότερη επίδοση.

Αυτό συμβαίνει γιατί για να γίνει συμμετρικά ο πολλαπλασιασμός χρειάζεται ακόμα μια MPI_AllReduce η οποία καταναλώνει απ' ότι βλέπουμε αρκετό χρόνο. Άνω στο σειριακό πείραμα αρκετές φόρες οι συμμετρικές εκδόσεις δουλεύουν καλύτερα από τη μη συμμετρική ,η MPI_AllReduce παίζει καθοριστικό ρόλο στο να αποφασίσουμε να εγκαταλείψουμε τις συμμετρικές εκδόσεις.

Στη συνέχεια, δεν ξαναεκτελούμε πείραμα με συμμετρική έκδοση του SpMV.

Κεφάλαιο 4ο : Μοντέλα Επικοινωνίας

Καθολική επικοινωνία

Με τον όρο καθολική επικοινωνία εννοούμε την επικοινωνία κι ανταλλαγή μηνυμάτων στο MPI,όπου κάθε διεργασία ανταλλάσσει δεδομένα με κάθε άλλη.
Το κομμάτι που γίνεται η ανταλλαγή ανάμεσα στις διεργασίες είναι το παρακάτω:

```
MPI_Allgatherv(plv, l_nr_rows, MPI_SPM_VALUE, pv, row_sendcnts, row_ptr_off, MPI_SPM_VALUE, MPI_COMM_WORLD);
```

Με αυτήν την εντολή της βιβλιοθήκης MPI συνενώνονται όλα τ κομμάτια p_local στις σωστές θέσεις για να φτιάξουν το καθολικό διάνυσμα p. Επίσης το καθολικό διάνυσμα p πηγαίνει σε όλες προφανώς τις διεργασίες. Καταλαβαίνουμε ότι αυτό είναι μια χρονοβόρα διαδικασία η οποία γίνεται πάνω σε ένα δίκτυο διασύνδεσης.

Ο κώδικας για κάθε επανάληψη της CG για αυτήν τη υλοποίηση είναι ο εξής:

```
spm_csr_mulv(Al, p, zl,row_ptr_off_myrank);

lrr = cblas_ddot(l_nr_rows, rlv,  rlv); //rl*rl

MPI_Allreduce(&lrr, &rr, 1, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);
//all local lrr to rr

lzp = cblas_ddot(l_nr_rows, zlv,  plv);

MPI_Allreduce(&lzp, &zp, 1, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

alpha = rr / zp;

cblas_daxpy(l_nr_rows, alpha, plv,  xlv); //x = x + alpha*p
cblas_daxpy(l_nr_rows, -alpha, zlv,  rlv); //r = r - alpha*A*p
lrr_new = cblas_ddot(l_nr_rows, rlv,  rlv);

MPI_Allreduce(&lrr_new, &rr_new, 1, MPI_SPM_VALUE, MPI_SUM, MPI_COMM_WORLD);

beta = rr_new / rr;

cblas_axpyz(l_nr_rows,beta, rlv, plv); // p = r + beta*p

MPI_Allgatherv(plv, l_nr_rows, MPI_SPM_VALUE, pv, row_sendcnts, row_ptr_off, MPI_SPM_VALUE, MPI_COMM_WORLD);
```

Επικοινωνία point to point

Η πρώτη βελτιστοποίηση που θα κάνουμε στον αρχικό κώδικα είναι να υλοποιήσουμε μια μέθοδο επικοινωνίας η οποία δεν θα κάνει περιττά πράγματα με σκοπό να γλιτώσουμε χρόνο επικοινωνίας.

Η ιδέα είναι ότι η κάθε διεργασία, ανάλογα με τον τοπικό πίνακα που έχει ίσως να μη χρειάζεται όλο το διάνυσμα p για να εκτελέσει τον πολλαπλασιασμό. Στην ουσία χρειάζεται μόνο εκείνα τα στοιχεία του p για τα οποία υπάρχει στοιχείο του πίνακα του A που θα πολλαπλασιαστεί μαζί του.

Έστω ότι στο παρακάτω παράδειγμα δημιουργούμε 4 διεργασίες,η κάθε μια από τις οποίες έχει 2 γραμμές του πίνακα. Αυτό σημαίνει ότι έχει και 2 στήλες του p και του z.

p

1η	1η	2η	2η	3η	3η	4η	4η

* A

1.3							
	2.98			15.83			
	-2.6					10.3	
		1.2			25.6		
							-98.4
				-0.25			
			12.6				8.9
							2.4

Για παράδειγμα η 1η διεργασία που έχει την 1η και την 2η γραμμή έχει 3 στοιχεία:ένα στην 1η στήλη ,ένα στην 2 η στήλη κι ένα στην 5η στήλη. Άρα για να κάνει τον πολλαπλασιασμό του τοπικού πίνακα A_local ο οποίος είναι οι 2 πρώτες γραμμές του πίνακα A,δε χρειάζεται ολόκληρο το διάνυσμα p αλλά μόνο το 1ο,το 2ο και το 5ο στοιχείο του p. Τώρα μένει να δούμε ποιος τα έχει αυτά τα στοιχεία. Το 1ο και το 2ο στοιχείο του p ανήκουν στην ίδια την 1η διεργασία άρα δε χρειάζεται να τα ζητήσει από καπου.Το5ο στοιχείο ανήκει στην 3η διεργασία άρα από αυτήν τη διεργασία θα ζητήσει αυτό το στοιχείο.

Βλέπουμε δηλαδή πόσα λιγότερα στοιχεία χρειάζεται στην ουσία και πόσο όφελος επικοινωνίας αναμένεται να έχουμε καθώς γλιτώνουμε όλες τις περιττές ανταλλαγές που είχαμε την ALL_Gatherv.

Άρα επιχειρούμε να υλοποιήσουμε μια point to point μέθοδο επικοινωνίας η οποία θα εκμεταλλεύεται αυτήν την αδυναμία. Δηλαδή κάθε διεργασία θα λαμβάνει ακριβώς όσα στοιχεία

χρειάζεται.

Δηλαδή με αυτόν τρόπο μπορούμε να μειώσουμε κατά πολύ τον όγκο δεδομένων που ανταλλάσσονται. Αυτό που θα χρειαστεί να πληρώσουμε είναι επεξεργαστική δουλειά για την ανακατασκευή του καθολικού διανύσματος p από τα διάφορα μεμονωμένα στοιχεία. Άρα στοχεύουμε να ανταλλάξουμε επεξεργαστική ισχύ με μείωση φόρτου στο δίκτυο διασύνδεσης των επεξεργαστικών κόμβων.

Στην επικοινωνία αυτή η φιλοσοφία είναι η ακόλουθη: Κάθε διεργασία ξέρει εξαρχής πριν την επαναληπτική μέθοδο CG τι έχει να στείλει στον καθέναν άλλον και τι ακριβώς έχει να λάβει από τους άλλους. Άρα στο τέλος κάθε επανάληψης όταν χρειαστεί να ανταλλάξει δεδομένα με τις υπόλοιπες διεργασίες:

1. Πακετάρει για κάθε άλλη διεργασία ένα πακέτο που πηγάζει από διάνυσμα p.
2. Το στέλνει ένα για κάθε διεργασία διαφορετικό.
3. Λαμβάνει από κάθε άλλη διεργασία ένα αντίστοιχο πακέτο.
4. Αποκωδικοποιεί αυτό που έλαβε και το ενσωματώνει στο διάνυσμα p που πρέπει να έχει.

Το κομμάτι που γίνεται η ανταλλαγή ανάμεσα στις διεργασίες και στην ουσία αντικαθιστά την MPI_AllGatherv της προηγούμενης υλοποίησης είναι το παρακάτω:

```
for (i=0; i < l_nr_rows; i++)
        pv[row_ptr_off[myrank]+i]=plv[i];

//pack for sending
for(i = 0; i < size; i++)
        if(i!=myrank)
                for(j = 0; j < send_vecs->count[i]; j++)
                        send_vecs->values[send_displacements[i] + j] =
                        plv[send_vecs->offsets[send_displacements[i] + j]];

//send-receive
t = 0;
for( i = 0; i < size; i++)
        if(i!=myrank){
                MPI_Isend(&(send_vecs->values[send_displacements[i]]), send_vecs->count[i],
MPI_SPM_VALUE, i, ty+myrank*100+i*99, MPI_COMM_WORLD, &(reqs[t++]));
                MPI_Irecv(&(recv_vecs->values[recv_displacements[i]]), recv_vecs->count[i],
MPI_SPM_VALUE, i, ty+i*100+myrank*99, MPI_COMM_WORLD, &(reqs[t++]));
        }

for(t = 0; t < 2*(size - 1); t++)
                MPI_Wait(&(reqs[t]),&status);

//unpack
for(i = 0; i < size; i++)
        if(i!=myrank)
                for(j = 0; j < recv_vecs->count[i]; j++)
                        pv[row_ptr_off[i] + recv_vecs->offsets[recv_displacements[i]+j]] =
                        recv_vecs->values[recv_displacements[i]+j];
```

Υλοποίηση χωρίς MPI_AllReduce

Το επόμενο βήμα είναι να αποφύγουμε το σχετικά μεγάλο κόστος της Reduce που γίνεται για να ανταλλαχθούν οι τιμές alpha και beta.

Έτσι,υλοποιούμε μια παραλλαγή της αρχικής υλοποίησης η οποία δεν θα περιέχει την MPI_AllReduce. Για να μην χρειάζεται η Reduce,θα πρέπει η κάθε διεργασία να βλέπει ολόκληρα τα διανύσματα κι όχι πια τα τοπικά και να εκτελεί εσωτερικά γινόμενα και πράξεις πάνω στα καθολικά διανύσματα. Αυτό εξυπακούεται ότι θα γίνεται περιττή δουλειά πάνω στα διανύσματα,δεχόμαστε όμως όπως και με την υλοποίηση της p2p μεθόδου επικοινωνίας,ότι ίσως η ανταλλαγή επεξεργαστικής ισχύος με μια μείωση στην επικοινωνία πάνω στο δίκτυο ίσως να μας συμφέρει.

Αντί να ανταλλάσσουμε τώρα τα p_local,πλέον δεν χρειάζεται να το κάνουμε αυτό γιατί έχουμε επεξεργασθεί το p καθολικά αφού όπως είπαμε ο καθένας τώρα επιτελεί εργασία πάνω στα καθολικά διανύσματα. Αντί αυτού χρειάζεται να ανταλλάξουμε το z_local που προκύπτει από τον πολλαπλασιασμό,ώστε να έχουμε ολόκληρο το z. Στην ουσία αυτό που γίνεται παράλληλα τώρα είναι μόνο ο πολλαπλασιασμός. Για να αποφεύγαμε την ανταλλαγή των z_local θα έπρεπε ο καθένας να εκτελούσε τον πολλαπλασιασμό καθολικά,κάτι που δεν θα είχε νόημα αφού τότε το πρόβλημα θα εκφυλιζόταν σε σειριακό.

Ο κώδικας σε κάθε επανάληψη της CG για αυτήν την υλοποίηση είναι ο εξής:

```
spm_csr_mulv(Al, p, zl,row_ptr_off_myrank);

MPI_Allgatherv(zlv, l_nr_rows, MPI_SPM_VALUE, zv, row_sendcnts, row_ptr_off,
MPI_SPM_VALUE, MPI_COMM_WORLD);

rr = cblas_ddot(N, rv,  rv); //rl*rl
zp = cblas_ddot(N, zv,  pv);
alpha = rr / zp;

cblas_daxpy(N, alpha, pv,  xv); //x = x + alpha*p
cblas_daxpy(N, -alpha, zv,  rv); //r = r - alpha*A*p

rr_new = cblas_ddot(N, rv,  rv);

beta = rr_new / rr;

cblas_axpyz(N,beta, rv, pv);
```

Πειραματικό Μέρος Κεφαλαίου

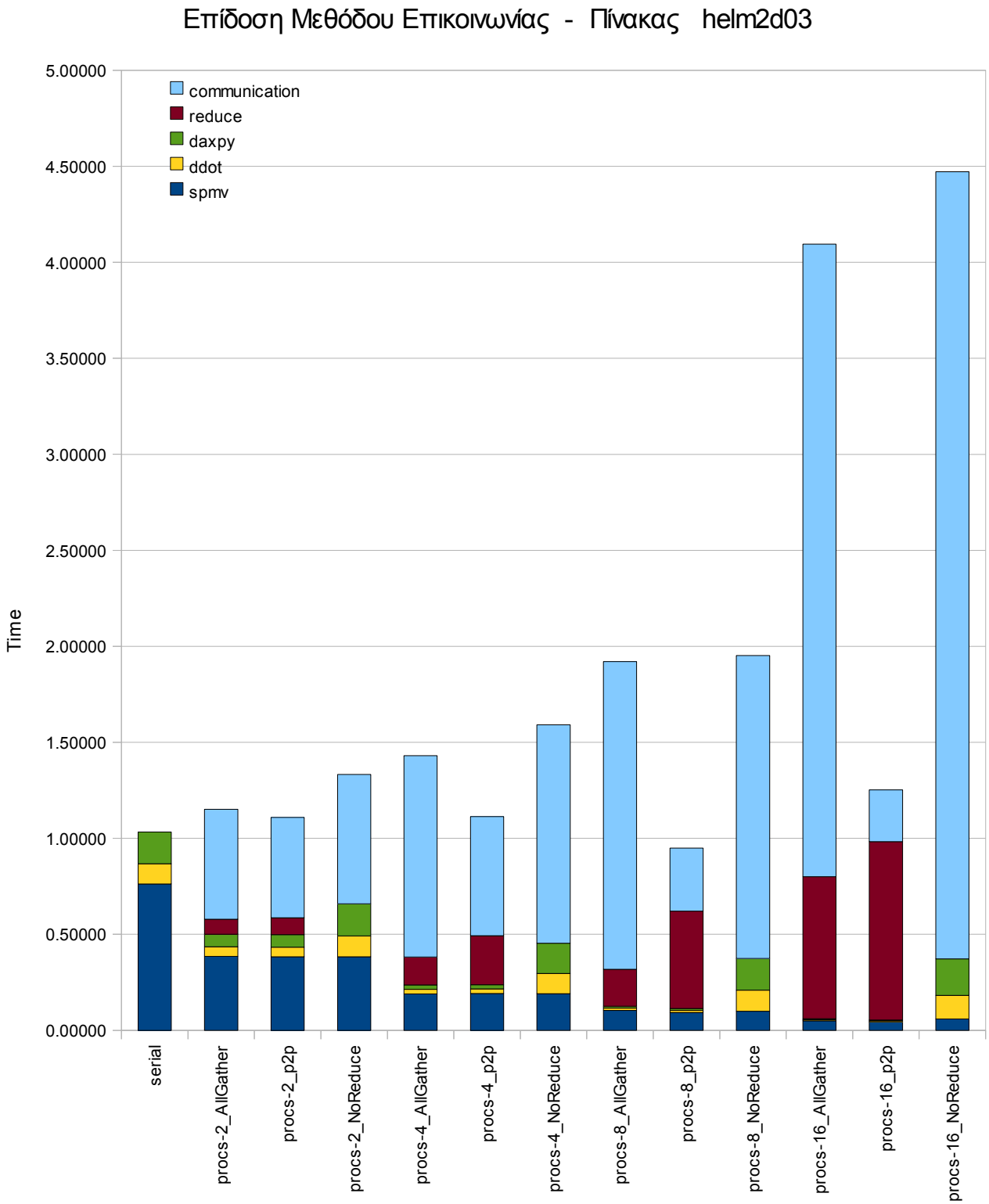
Παρακάτω δείχνεται σε διαγράμματα οι επιμέρους χρόνοι για τα

- SpMV
- ddot
- daxpy
- MPI_AllReduce
- Communication

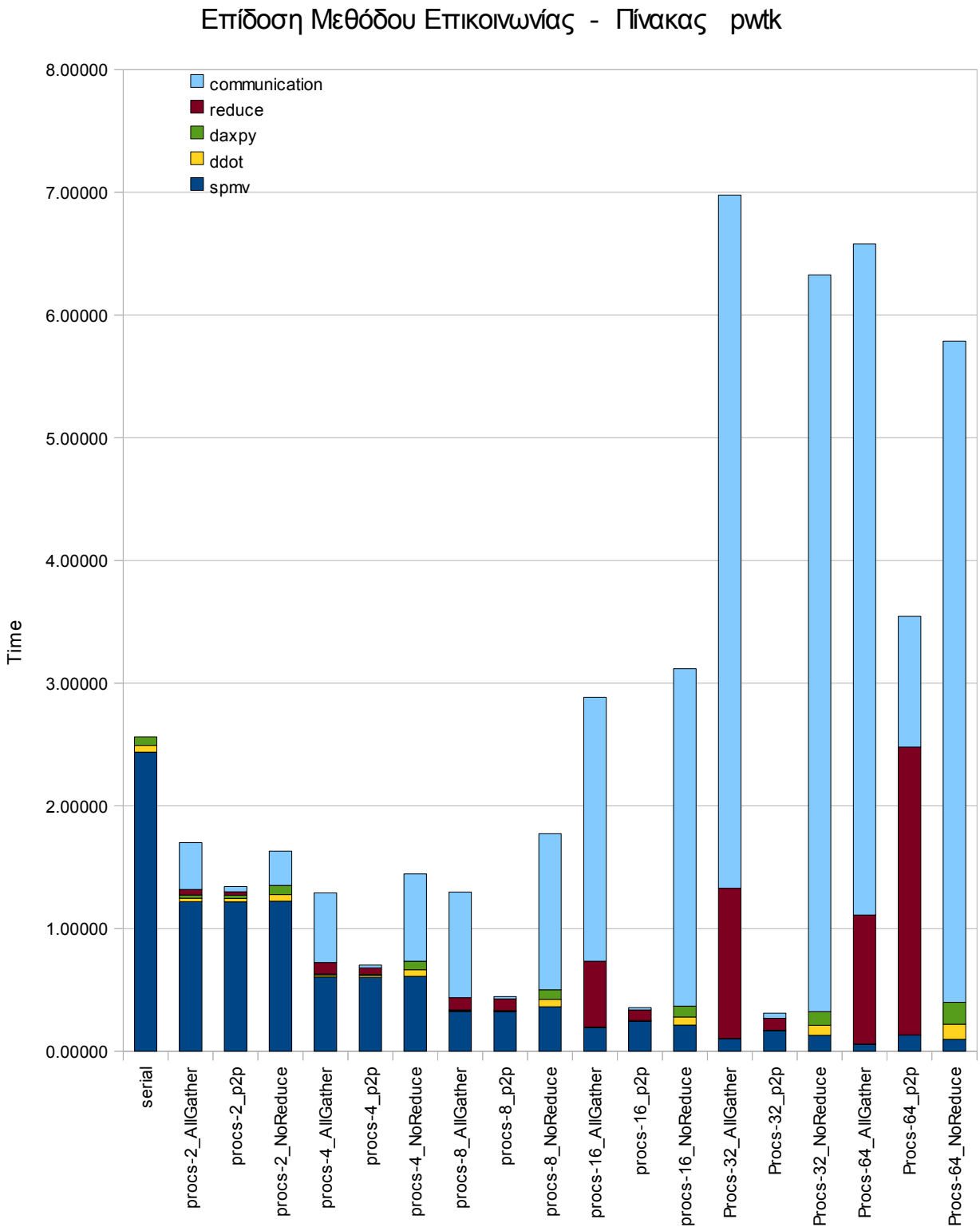
και για τις 3 μεθόδους που αναφέραμε παραπάνω δηλαδή για:

- την MPI_AllGatherv
- την point-to-point επικοινωνία
- και για την υλοποίηση χωρίς MPI_AllReduce.

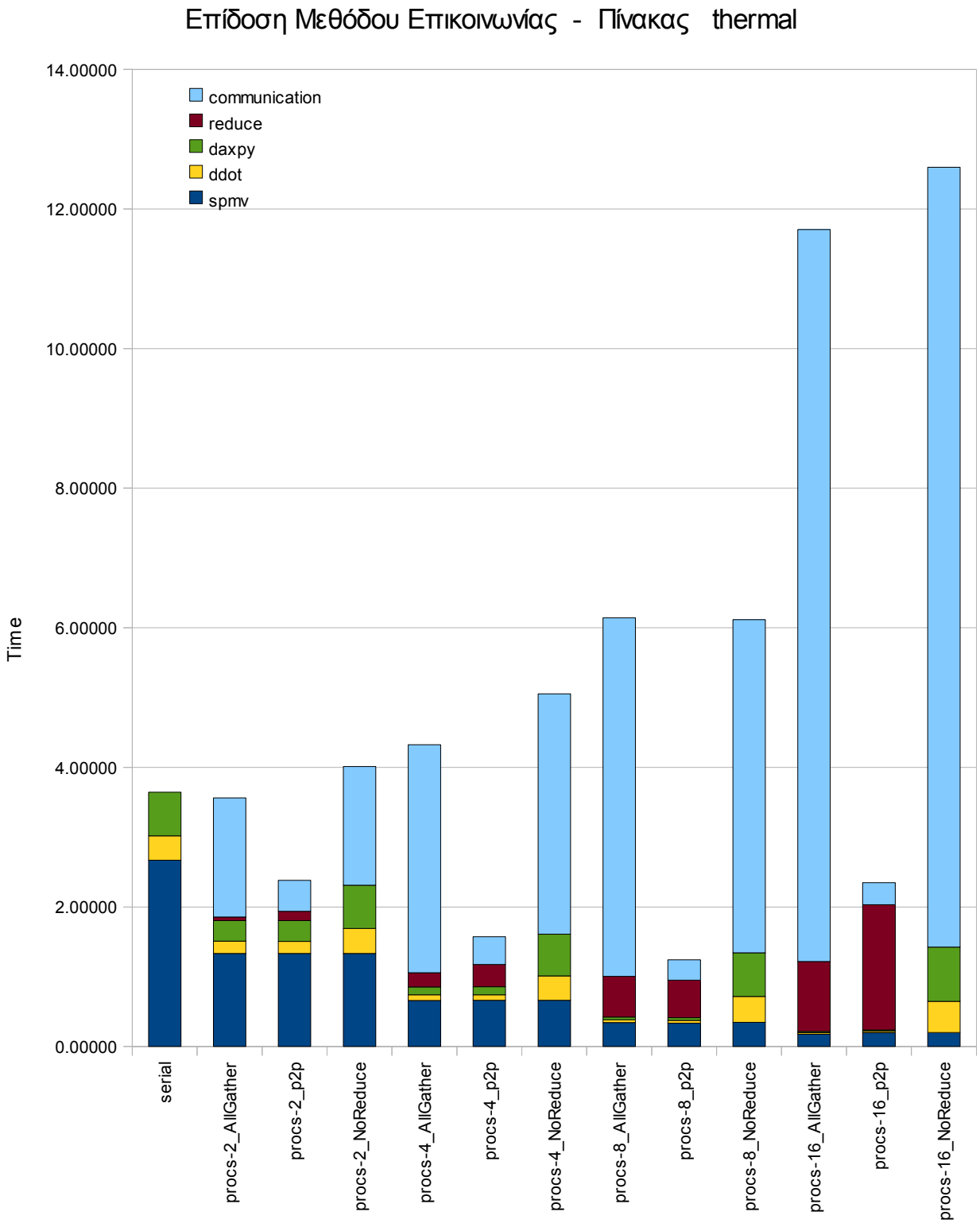
Τρέξαμε την επαναληπτική μέθοδο σύγκλισης CG για 30 επαναλήψεις και για αριθμό διεργασιών 1,2,4,8,16,32,64 χρησιμοποιώντας 8 κόμβους του cluster. Όπου δεν δείχνεται στο διαγράμματα οι επιδόσεις για πολλές διεργασίες π.χ. 32 η 64 είναι γιατί η επίδοση ήταν πολύ κακή και πλέον δεν φαίνονταν καλά τα υπόλοιπα που μας ενδιαφέρουν.



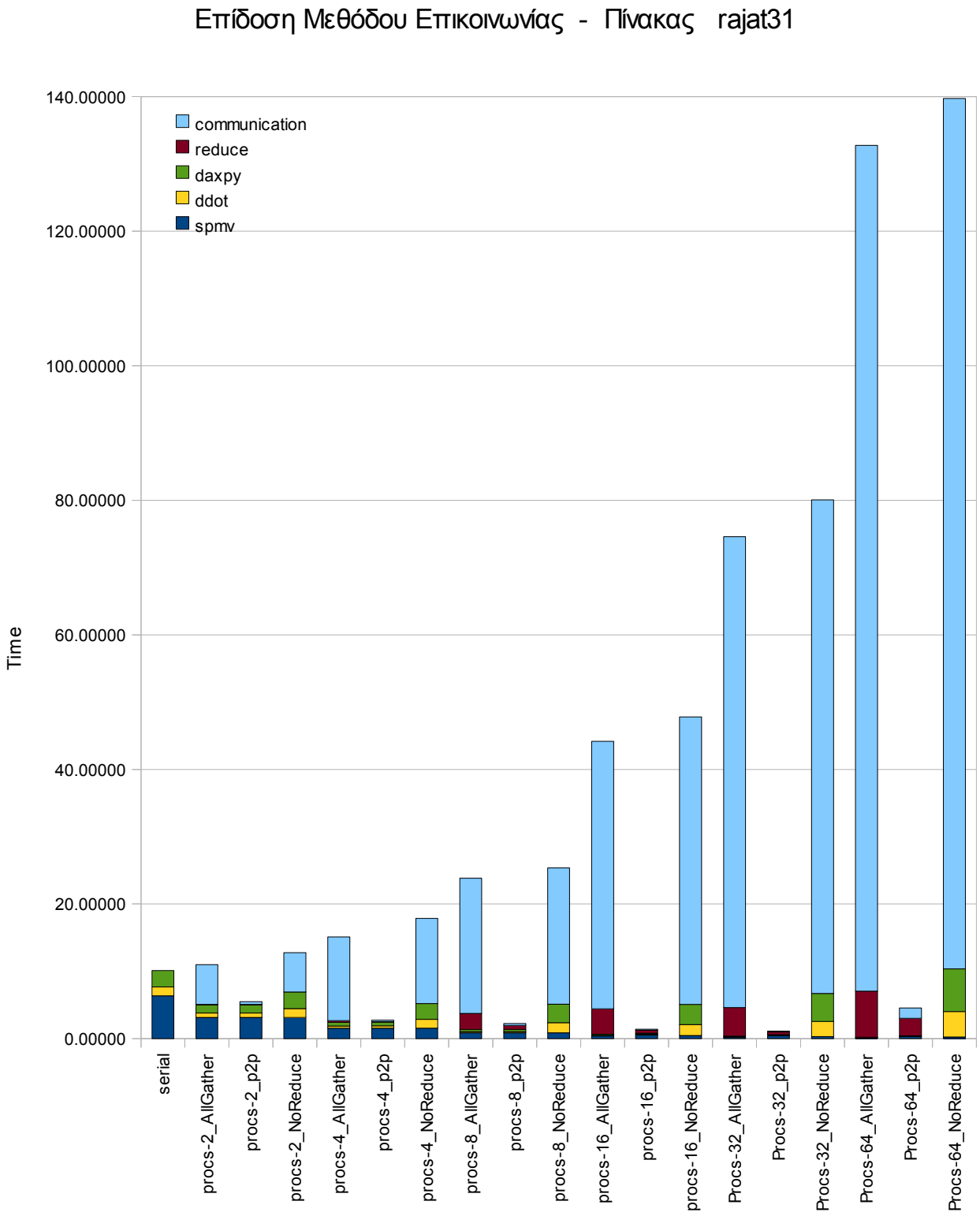
Σχήμα 4.1 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας helm2d03



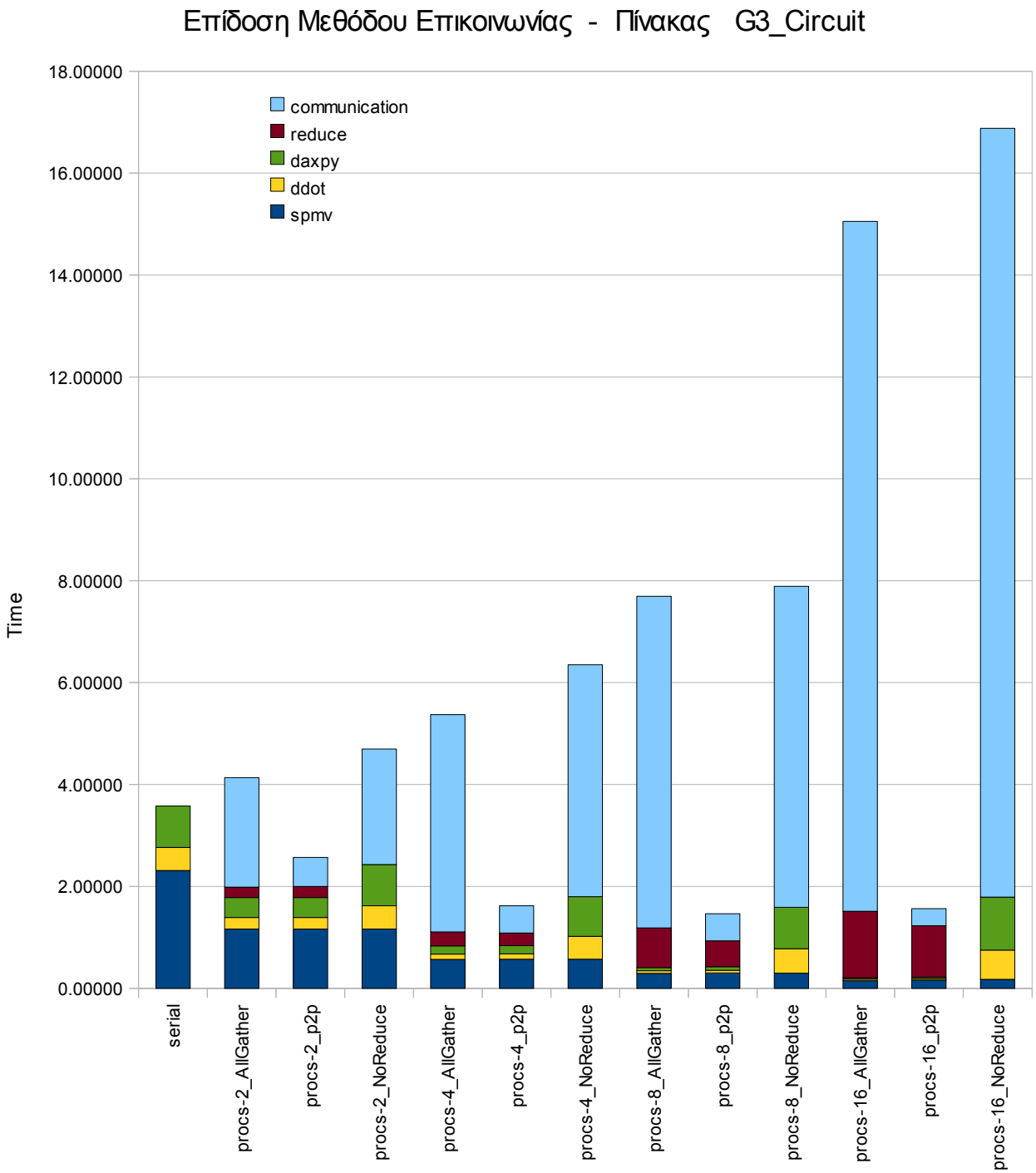
Σχήμα 4.2 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας pwtk



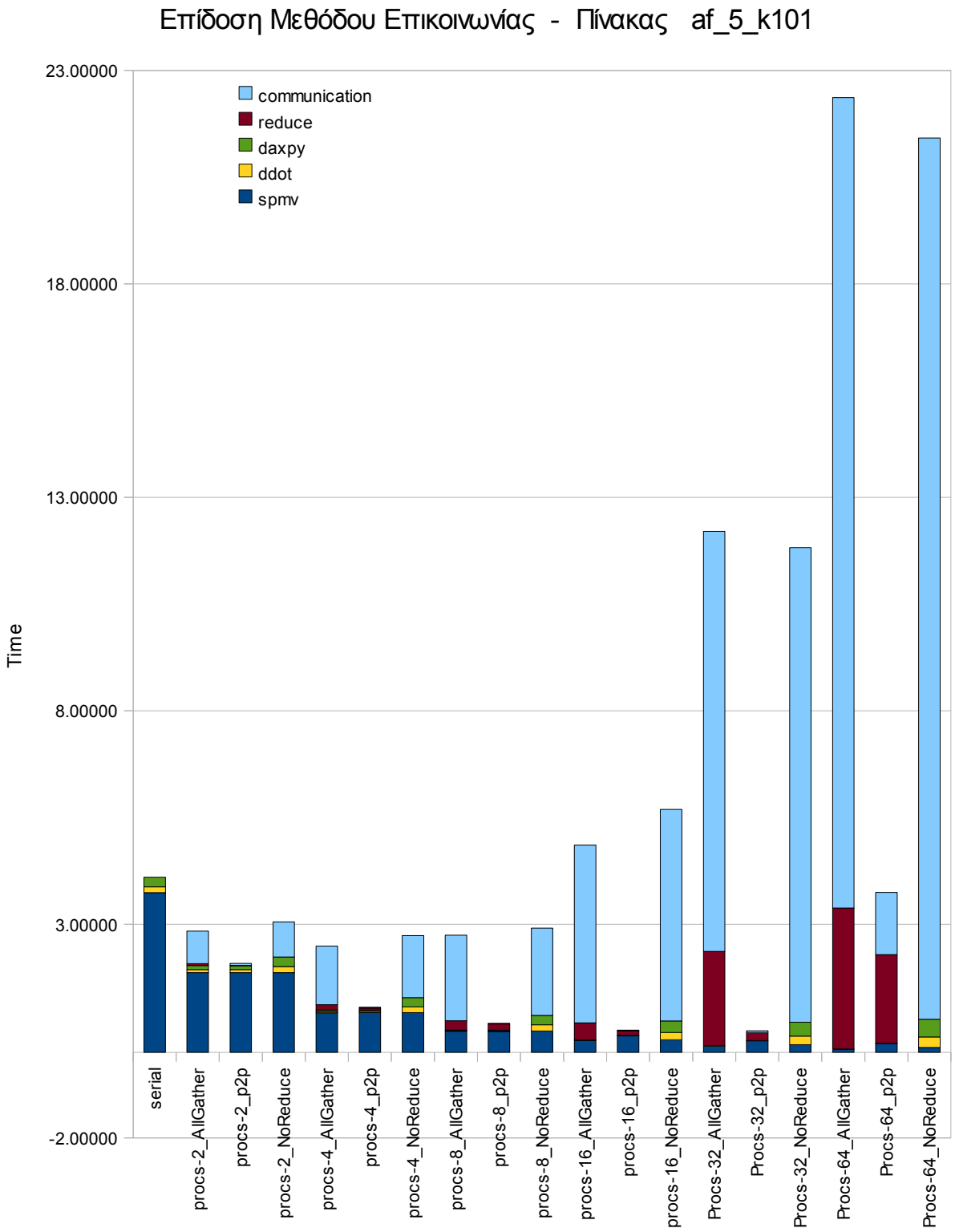
Σχήμα 4.3 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας thermal2



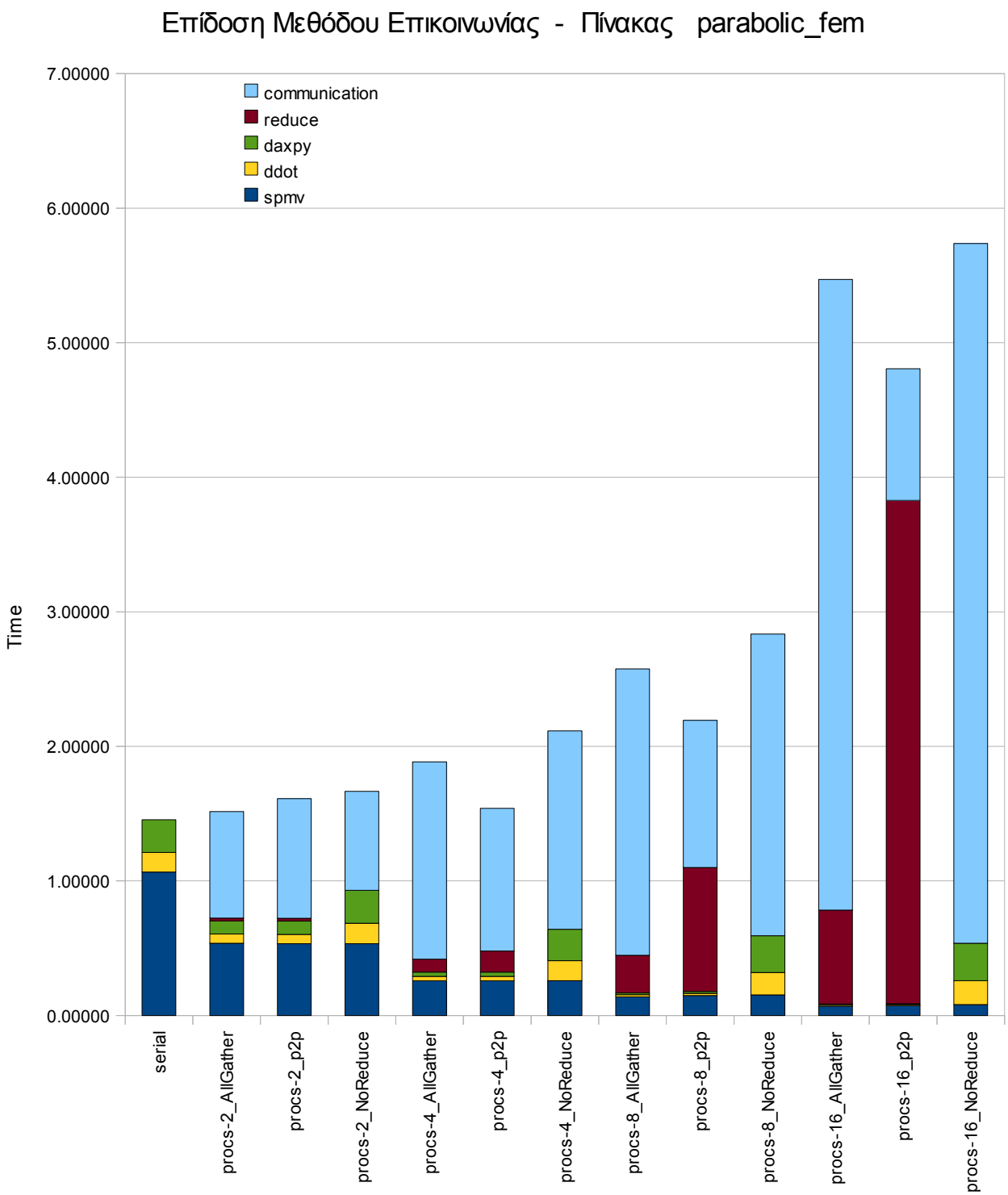
Σχήμα 4.4 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας rajat31



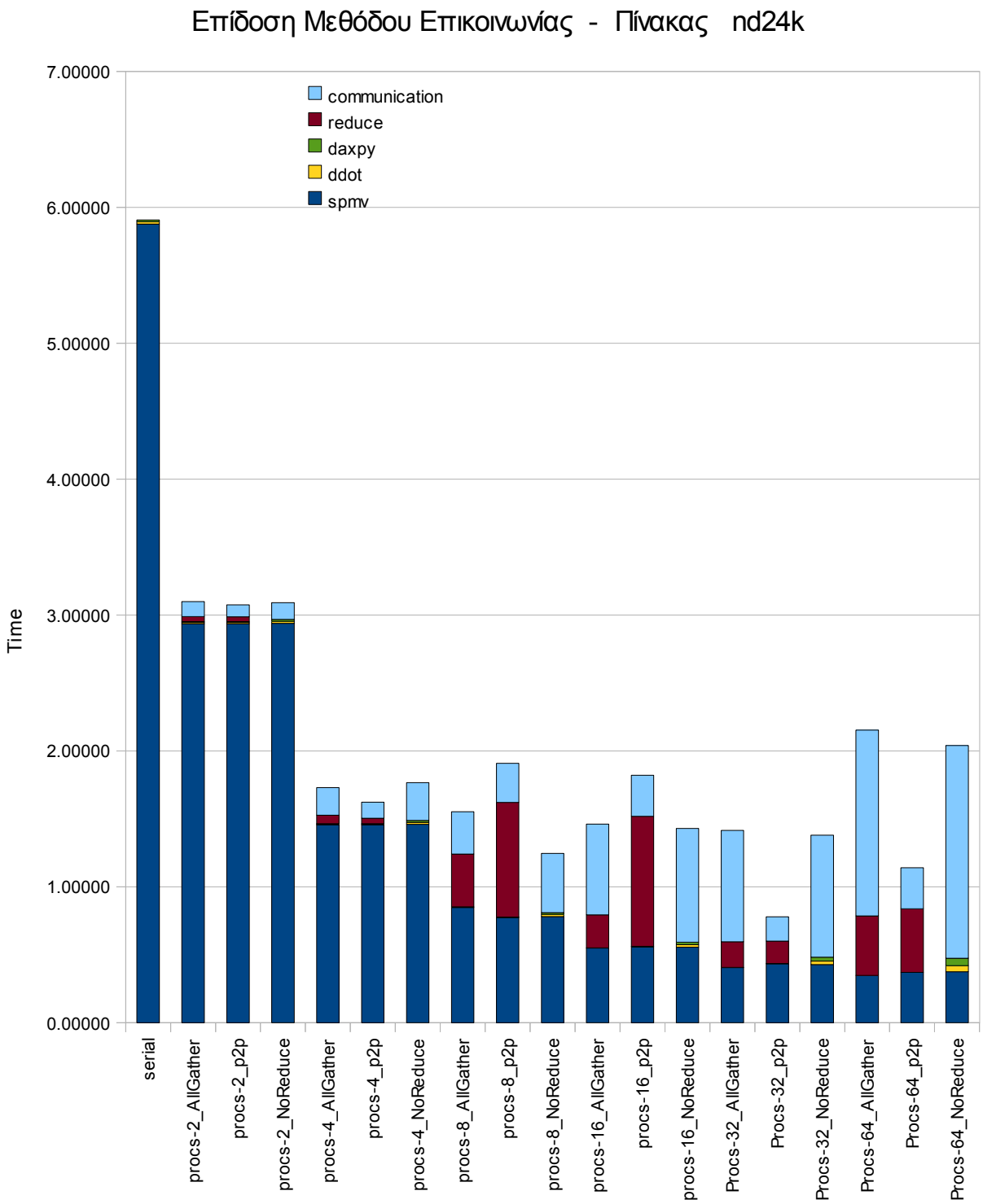
Σχήμα 4.5 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας g3_Circuit



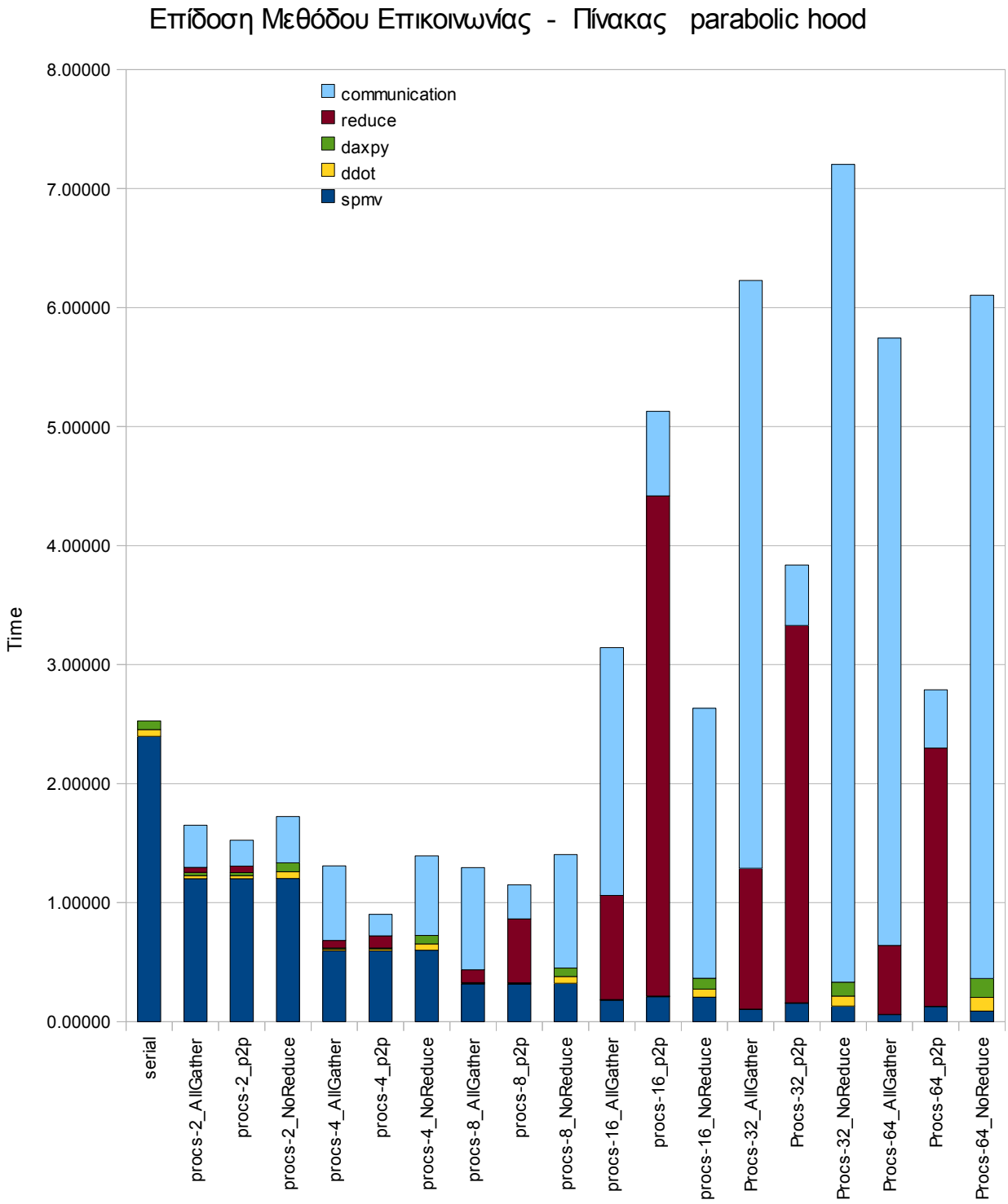
Σχήμα 4.6 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας af_5_k101



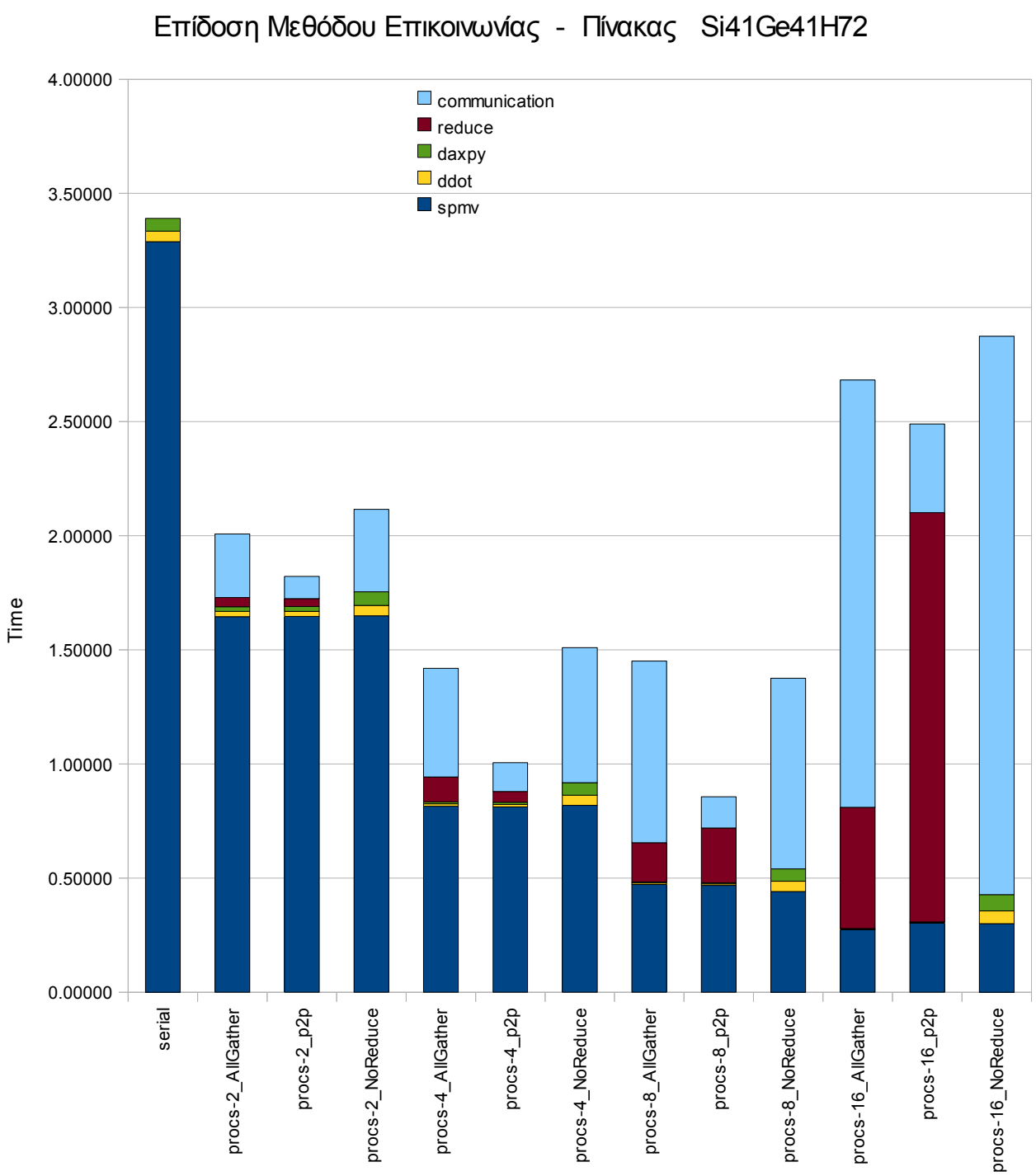
Σχήμα 4.7 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας parabolic



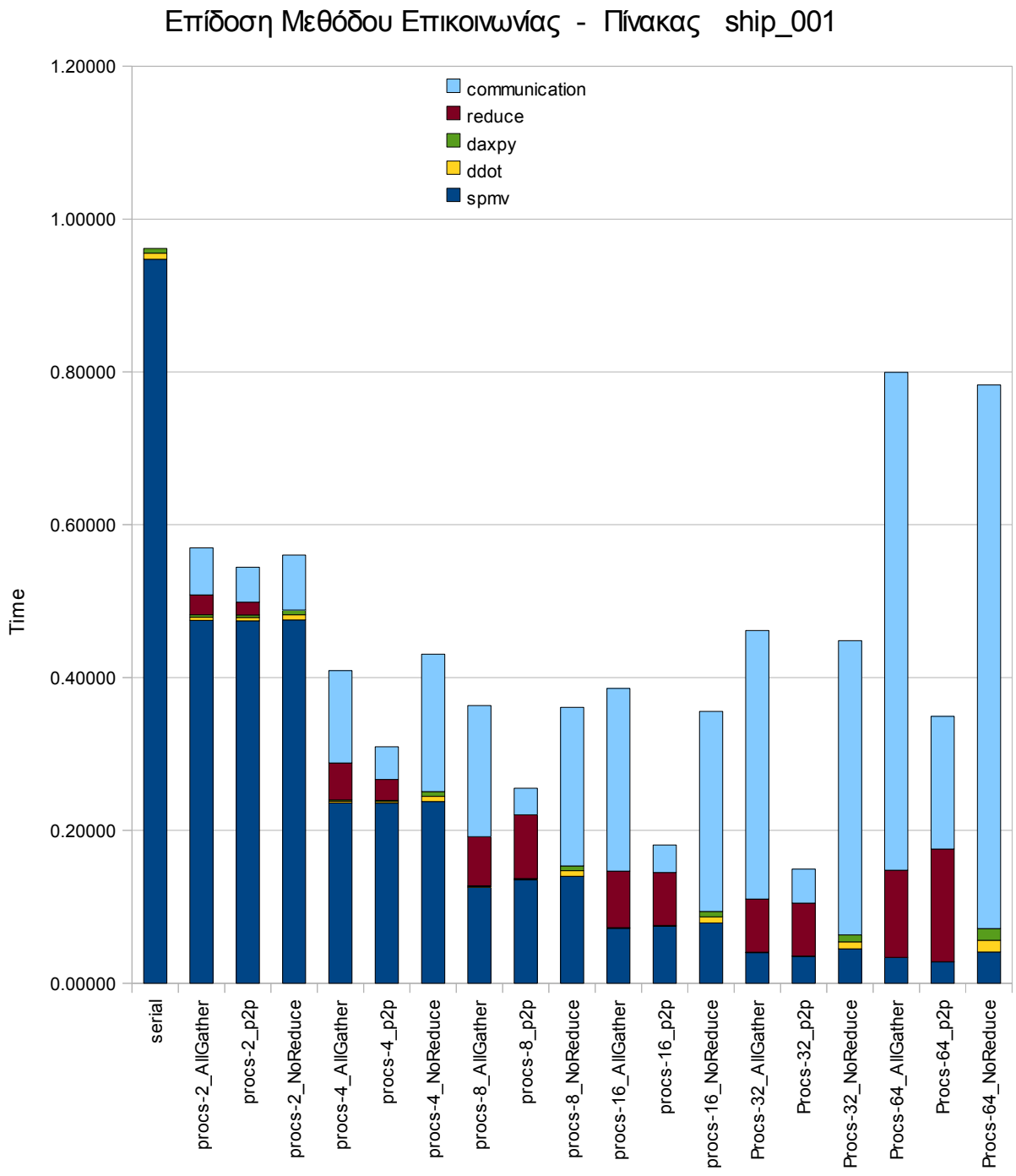
Σχήμα 4.8 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας nd24k



Σχήμα 4.9 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας hood

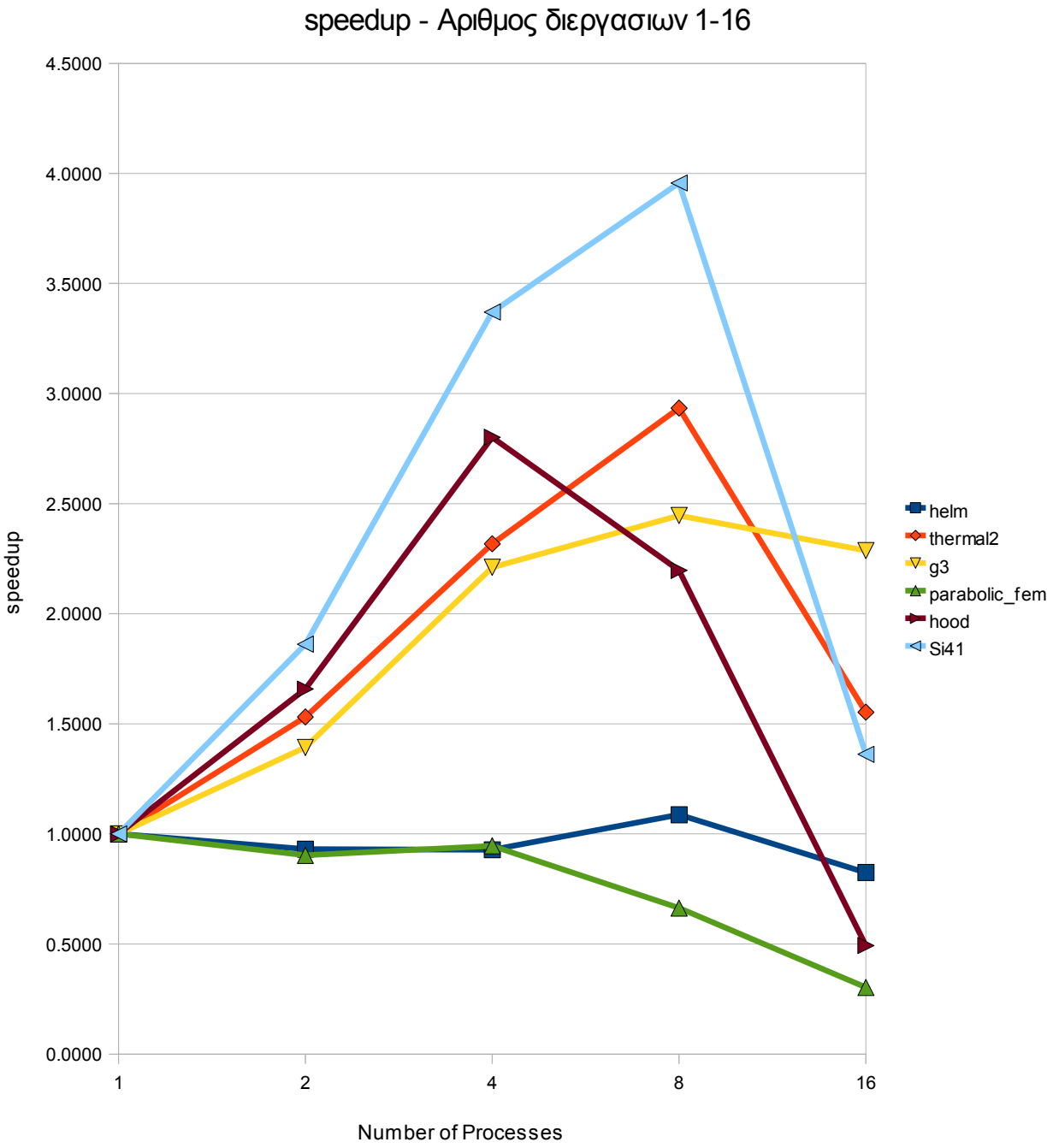


Σχήμα 4.10 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας Si41

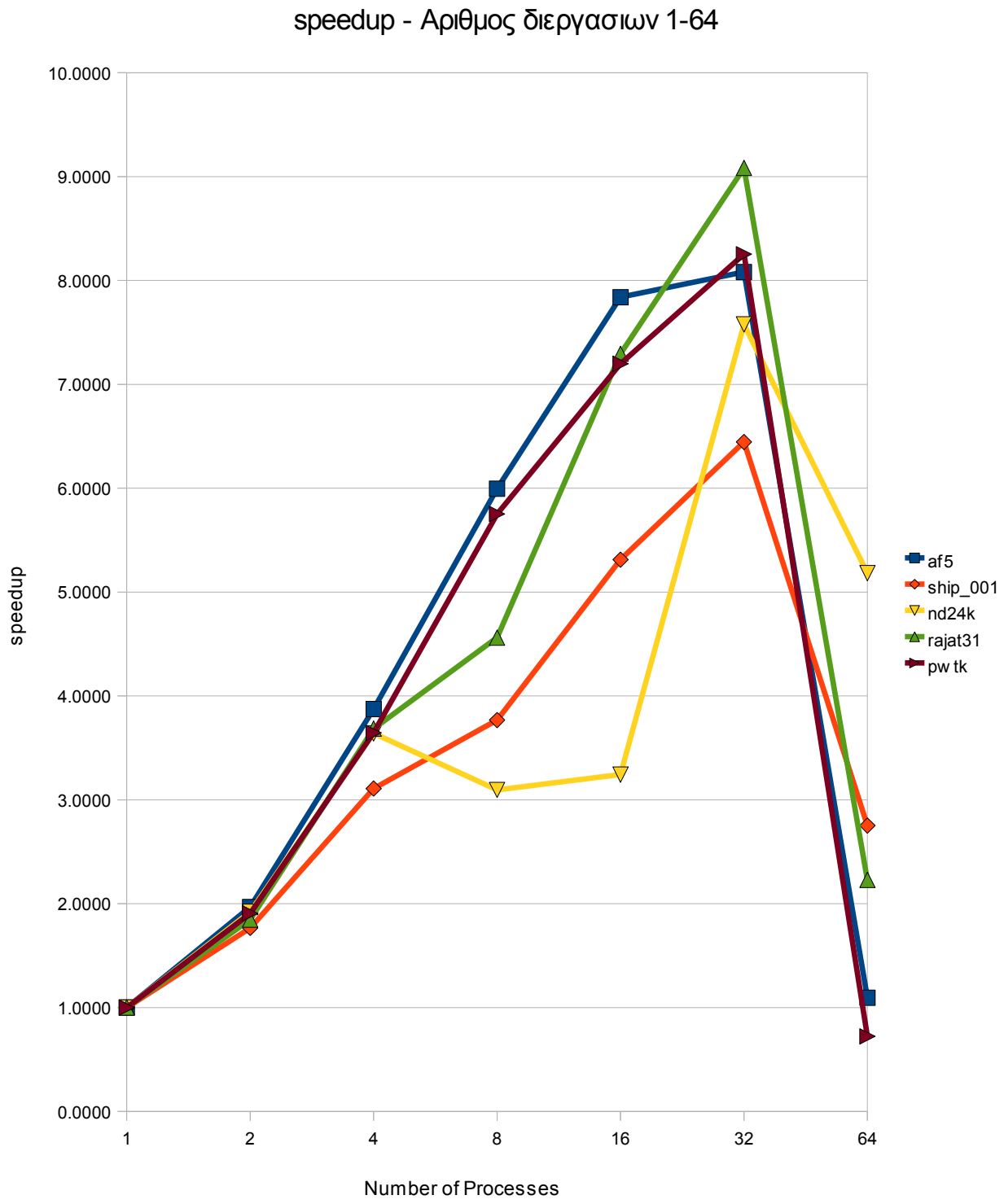


Σχήμα 4.11 - Επίδοση Μεθόδου Επικοινωνίας - Πίνακας ship_001

Στα δυο επόμενα διαγράμματα φαίνεται το Speed-up για τους 11 προηγούμενους πίνακες. Στο πρώτο διάγραμμα εμφανίζουμε τα αποτελέσματα για 16 το πολύ διεργασίες, καθώς στις 32 και 64 διεργασίες τα αποτελέσματα δεν είναι καθόλου καλά και χαλάει η εικόνα του διαγράμματος. Στο δεύτερο διάγραμμα φαίνεται το speedup μέχρι και τις 64 διεργασίες.



Σχήμα 4.12 - Απεικόνιση speedup με χρήση MPI - Αριθμός διεργασιών 1-16



Σχήμα 4.13 - Απεικόνιση speedup με χρήση MPI - Αριθμός διεργασιών 1-64

Συμπεράσματα για τις μεθόδους Επικοινωνίας

Βλέπουμε ότι οι επεξεργαστικές εργασίες έχουν κλιμάκωση:Δηλαδή,το SpMV,το εσωτερικό γινόμενο διανυσμάτων και οι προσθαφαιρέσεις διανυσμάτων,όταν διαιρούνται ανάμεσα σε πολλούς επεξεργαστικούς κόμβους,απαιτούν σαφώς λιγότερο χρόνο.

Από την άλλη,όσο αυξάνουμε τον αριθμό των κόμβων στο MPI,αυξάνονται οι χρόνοι επικοινωνίας δραματικά. Ερχόμαστε λοιπόν στη θέση να αποφασίσουμε σε ποιο βαθμό θα διαιρέσουμε το πρόβλημα.

Παρατηρούμε ότι όσο πιο πυκνός είναι ένας πίνακας,τόσο πιο πολλή επεξεργαστική δουλειά απαιτείται σε σχέση με το κόστος επικοινωνίας. Γιαυτό το λόγο,είδαμε πίνακες να έχουν την καλύτερη επίδοση στους 4,8,16 η 32 κόμβους-διεργασίες.

Γενικότερα όσο τα μη μηδενικά στοιχεία είναι πολλά και η διάσταση του πίνακα μικρή μας συμφέρει να σπάσουμε το πρόβλημα σε όσο περισσοτέρους κόμβους έχουμε. Αντίθετα αν μιλάμε για πολύ αραιούς πίνακες καλύτερα να μην επεκταθούμε σε πάνω από 8 κόμβους επειδή το επεξεργαστικό φορτίο δεν είναι τόσο πολύ ώστε να αντισταθμίζει το κόστος επικοινωνίας.

Σε κάθε περίπτωση βλέπουμε ότι η point to point επικοινωνία νικά κατά κράτος. Το πόσο καλύτερη είναι εξαρτάται από τον εκάστοτε πίνακα, δηλαδή από το μέγεθός του, τον αριθμό μη-μηδενικών στοιχείων του και την τοπολογία των μη-μηδενικών στοιχείων.

Συνοψίζοντας λοιπόν, έχουμε για την παραλληλοποίηση σε συστήματα κατανεμημένης μνήμης:

- Απορρίπτουμε τις συμμετρικές εκδόσεις του πυρήνα SpMV και κρατάμε μόνο τη μη-συμμετρική.
- Απορρίπτουμε την MPI_AllGatherv καθολική μέθοδο επικοινωνίας και την υλοποίηση χωρίς MPI_AllReduce και κρατάμε την επικοινωνία σημείου προς σημείο.

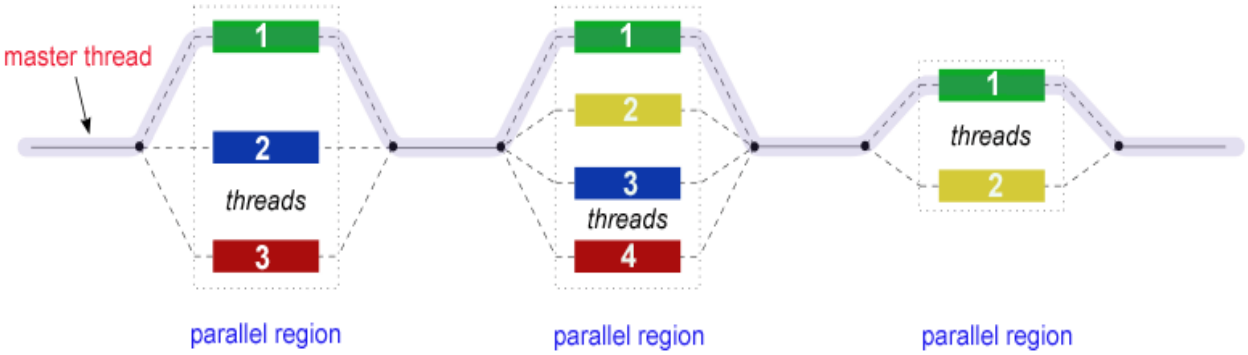
Κεφάλαιο 5ο : Παράλληλη υλοποίηση στο μοντέλο κοινής μνήμης

Παραλληλοποίηση με OpenMP

Το OpenMP (Open Multiprocessing) είναι μια διεπαφή που υποστηρίζει με μεγάλη φορητότητα παράλληλο προγραμματισμό κοινής μνήμης σε γλώσσες όπως η C,C++ και η Fortran, στις περισσότερες αρχιτεκτονικές υπολογιστών και στα περισσότερα λειτουργικά συστήματα όπως τα Solaris,AIX,HP-UX,GNU/Linux,MAC OS X και Windows.

Αποτελείται από ένα σύνολο από compiler directives,συναρτήσεις βιβλιοθήκης και μεταβλητές περιβάλλοντός που επηρεάζουν τη συμπεριφορά του προγράμματος στο χρόνο εκτέλεσης.

Το OpenMP είναι μια υλοποίηση πολυνηματισμού, μια μέθοδο παραλληλισμού στην οποία ένα master thread σπάει σε περισσότερα τα οποία εκτελούν είτε ίδιο κώδικα με διαφορετικά δεδομένα είτε τελείως διαφορετικό κώδικα και στο τέλος τη παράλληλης περιοχής παραμένει κι υφίσταται μόνο το Master thread.



Εικόνα 5.1 - Αναπαράσταση λειτουργίας OpenMP

Παρακάτω θα δείξουμε που έχουμε εισάγει στον κωδικά μας OpenMP directives ώστε να είναι παραλληλοποιήσιμος.

1) Στις συναρτήσεις της βιβλιοθήκης CBLAS

- Στην cblas_axpyz όπου γίνεται το Y[i] = X[i] + beta * Y[i];

```
void cblas_axpyz( const int N, const double beta, const double *X,  double *Y ){
        int i;
        double beta2=beta;

        if(openmp_on){
        omp_set_num_threads(nthreads);

        #pragma omp parallel for firstprivate(beta2) schedule (dynamic,chunk)
        for (i = 0; i < N; i++)
                Y[i] = X[i] + beta2 * Y[i];
        }

        else{
                for (i = 0; i < N; i++)
                        Y[i] = X[i] + beta2 * Y[i];

        }

}
```

- Στην cblas_daxpy όπου γίνεται το Y[i] =Y[i] + alpha * X[i];

```
void cblas_daxpy( const int N, const double alpha, const double *X,  double *Y ){
        int i;
        double alpha2=alpha;

        if(openmp_on){
                omp_set_num_threads(nthreads);

                #pragma omp parallel for firstprivate(alpha2) schedule (dynamic,chunk)
                for (i = 0; i < N; i++)
                        Y[i] += alpha2 * X[i];

        }

        else{
                for (i = 0; i < N; i++)
                        Y[i] += alpha2 * X[i];
        }
}
```

- Στην cblas_daxpy όπου γίνεται το εσωτερικό γινόμενο των X και Y:

```
double cblas_ddot( const int N, const double *X, const double *Y){

        double r  = 0.0;
        int    i;


        if(openmp_on){
                omp_set_num_threads(nthreads);

                #pragma omp parallel
                {
                        double ri  = 0.0;
                        #pragma omp for  schedule (dynamic,chunk)
                        for (i = 0; i < N; i++) {
                                ri  += X[i] * Y[i];
                        }


                        #pragma omp critical
                        {
                                r+=ri;
                        }
                }
        }
        else{
                        for (i = 0; i < N; i++) {
                                r  += X[i] * Y[i];
                        }
        }

        return r;
}
```

2) Στη ρουτίνα του SpmV

```
#pragma omp parallel for schedule(dynamic,chunk) private(j,_y,kk) num_threads(nthreads)
for (i = 0; i < nr_rows; i++) {

        _y = 0.0;
        kk=row_ptr[i+1];

        for (j = row_ptr[i]; j < kk; j++){
                _y += values[j] * x[col_ind[j]];
        }

        y[i]+=_y;

}
```

Στην ουσία έχουμε παραλληλοποιησει βρόχους for με τις απαραίτητες μεταβλητές που πρέπει να είναι ιδιωτικές ανάμεσα στα threads ορισμένες ως private.

Δρομολόγηση βρόχου

Στατική Δρομολόγηση

Την static δρομολόγηση,κατά την οποία χωρίζεται το μέγεθος του προβλήματος σε κομμάτια-υποπροβληματα με βάση των αριθμό των γραμμών. Αυτός ο διαμοιρασμός γίνεται στην αρχή της εκτέλεσης και δεν αλλάζει ανεξάρτητα με το αν κάποια threads τα πάνε καλύτερα από κάποια άλλα και θα μπορούσαν να δεχθούν κάποιο επιπλέον φορτίο. Αυτό δε σημαίνει απαραίτητα ότι έγινε άνιση κατανομή του φορτίου ανάμεσα στα threads αλλά για διάφορους λόγους κατά το χρόνο εκτέλεσης κάποια καθυστέρησαν.

Ωστόσο η διαφορά στο χρόνο ίσως -ανάλογα και με τον εκάστοτε πίνακα-να οφείλεται σε άνιση κατανομή του αριθμού των πολλαπλασιασμών ανάμεσα στα threads. Αυτό επειδή χωρίζοντας το ανά πλήθος γραμμών δεν γίνεται κάπου μέριμνα για την κατανομή των μη μηδενικών στοιχείων του πίνακα ανάμεσα στα τεμάχια γραμμών,γι'αυτο το λόγο κάποια κομμάτια είναι ίσως υπερφορτωμένα,όχι σε γραμμές αλλά σε πραγματικό φορτίο δηλαδή σε μη μηδενικά στοιχεία στα οποία αντιστοιχεί κι από ένας πολλαπλασιασμός.

Άλλος λόγος θα μπορούσε να είναι ο χρόνος που δαπανάται περιμένοντας ένα κομμάτι να έρθει από κύρια μνήμη. Κι όπως προείπαμε διάφοροι λόγοι καθυστέρησης κατά τη διάρκεια του χρόνου εκτέλεσης που ενυπάρχουν σε κάθε υπολογιστικό σύστημα που εκτελεί μια εφαρμογή.

Δυναμική Δρομολόγηση

Η dynamic δρομολόγηση η οποία όπως φαίνεται κι από το όνομα της είναι δυναμική με την έννοια ότι κάνει μια πιο έξυπνη δρομολόγηση σε σχέση με τη στατική,δηλαδή αναθέτει το επόμενο κομμάτι δουλειάς σε κάποιο από τα νήματα που δεν απασχολούνται εκείνη τη στιγμή,σε αντίθεση με τη στατική που φτιάχνει από την αρχή ένα πλάνο δρομολόγησης για το κάθε thread κι αυτό δεν αλλάζει.

Στο επόμενο από τα διαγράμματα, η διαφορά μεταξύ των δυο είναι σχεδόν αμελητέα,ωστόσο φαίνεται το όφελος της λογικής της δυναμικής δρομολόγησης. Αν συνυπολογίσουμε στο χρόνο της dynamic το αυξημένο χρόνο που χρειάζεται το OPENMP για να την εκτελέσει,εννοώντας ότι υπάρχει κάποιο αυξημένο overhead για να γίνει δυναμική δρομολόγηση καταλαβαίνουμε ότι αυτή η διαφορά θα ήταν ακόμα μεγαλύτερη αν δεν συνυπολογίζαμε το χρόνο του overhead. Αυτό δείχνει ακόμα καλύτερα τη διαφορά στο χρόνο που δαπανά κάθε thread.

OpenMP Tasks

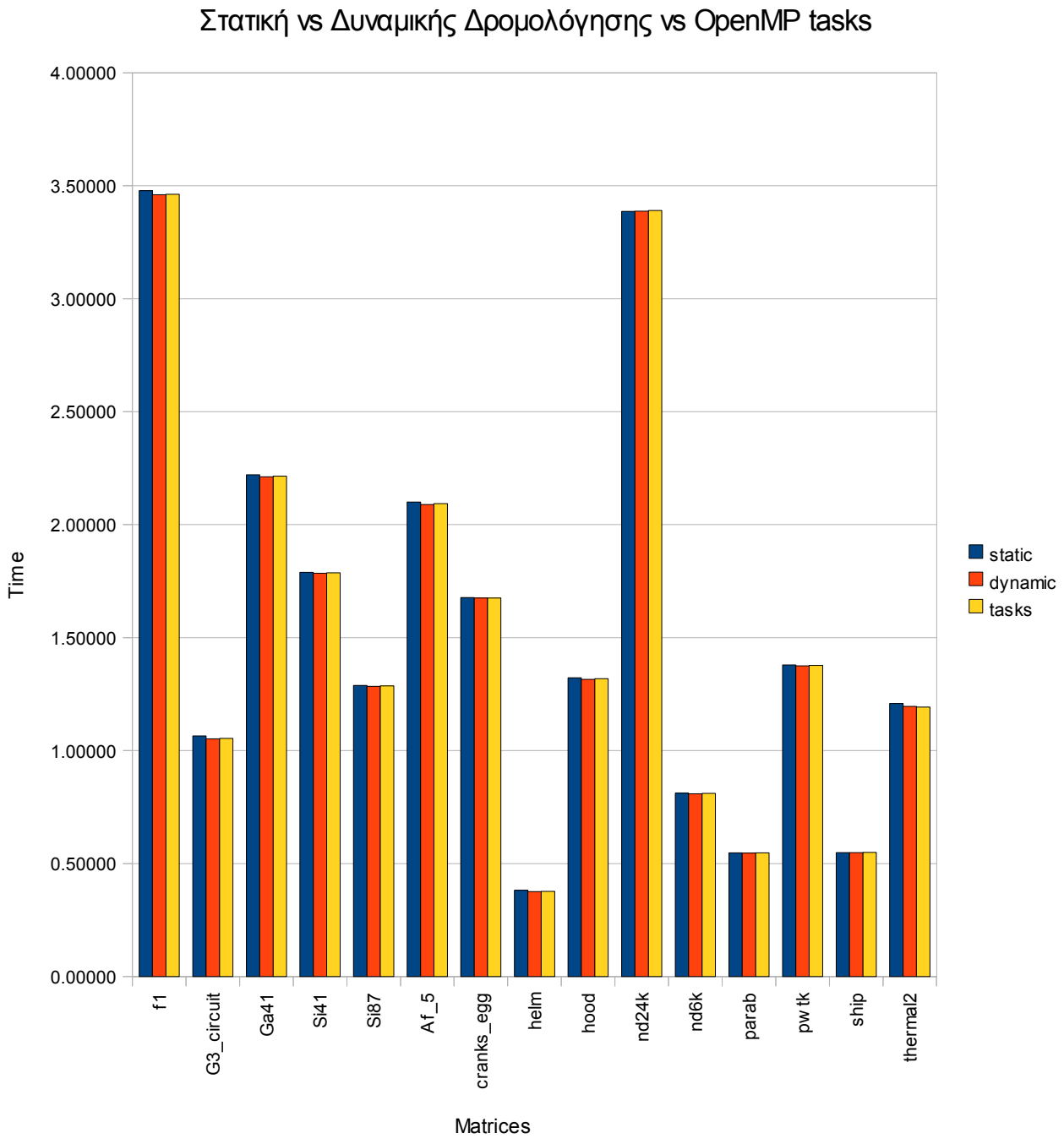
Η συγκεκριμένη δυνατότητα του OPENMP υπάρχει από την έκδοση 3.0 και μετά. Η λογική του είναι ότι σπάει το πρόβλημα σε κομμάτια δουλειάς. Το κάθε κομμάτι μπορούμε να το παρομοιάσουμε με ένα πακέτο το οποίο είναι μια αυτοτελής δουλειά κι ανατίθεται σε thread το οποίο δεν κάνει κάποια δουλειά εκείνη τη στιγμή.

Η λογική των tasks είναι παρεμφερής με αυτήν της dynamic δρομολόγησης στο συγκεκριμένο πρόβλημα που παραλληλοποιούμε την προσπέλαση ενός πίνακα με βάση τις γραμμές.

Σε άλλου τύπου προβλήματα θα μπορούσαμε να καταλάβουμε καλύτερα τη χρησιμότητα αυτής της τεχνικής. Στο παρόν πρόβλημα πάντως δεν έχει ουσιαστική διαφορά στην επίδοση σε σχέση με την dynamic.

Πειραματικό μέρος Κεφαλαίου

Παρακάτω φαίνεται η επίδοση σε χρόνο της static έναντι της dynamic δρομολόγησης κι έναντι στην επίδοση με OpenMP tasks για διάφορους πίνακες.



Σχήμα 5.2 - Στατική vs Δυναμικής Δρομολόγησης vs OpenMP tasks

Κεφάλαιο 6ο : Συμπίεση Πίνακα για SpMV

LZO Compression για SpMV

Η βιβλιοθήκη LZO είναι μια βιβλιοθήκη γραμμένη σε ANSI-C η οποία επιτελεί συμπίεση κι αποσυμπίεση δεδομένων. Το σημαντικό πλεονέκτημα της,που μας οδήγησε στο να την επιλέξουμε σε σχέση με άλλα προγράμματα συμπίεσης-αποσυμπίεσης είναι η πολύ υψηλή ταχύτητα αποσυμπίεσης. Στην εργασία μας αυτί,στην ουσία δεν μας ενδιαφέρει το pre-processing-cost παρά μόνο η επίδοση στο χρόνο μέσα στο κομμάτι της επαναληπτικής μεθόδου. Με άλλα λόγια μπορούμε να έχουμε μια μεγάλη συλλογή από συμπιεσμένους πίνακες που μπορεί να έχει προέλθει από αργή συμπίεση. Αυτό όμως θα γίνει μια φορά για έναν πίνακα. Αντιθέτως το εκάστοτε διάνυσμα με το οποίο θα γίνει ο πολλαπλασιασμός είναι αυτό που αλλάζει.

Όπως όλα τα εργαλεία συμπίεσης,έτσι κι η βιβλιοθήκη LZO μπορεί να επιτύχει υψηλότερα επίπεδα συμπίεσης πληρώνοντας βέβαια με χρόνο συμπίεσης. Με άλλα λόγια όσο μικρότερο λόγο συμπίεσης θέλουμε να πετύχουμε τόσο περισσότερο χρόνο θα καταναλώσουμε. Αυτός όμως ο pre-processing χρόνος δεν μας ενδιαφέρει.

Από τις μετρικές του εκδότη του LZO βλέπουμε ότι όσο πιο πολύ συμπιέσουμε τον πίνακα,τόσο μεγαλύτερη ταχύτητα αποσυμπίεσης θα επιτύχουμε.

Για νακαταλάβουμε καλύτερα την ταχύτητα αποσυμπίεσης αξίζει να αναφέρουμε ότι ο καλύτερος αλγόριθμος της βιβλιοθήκης LZO βελτιστοποιημένος σε assembly-i386 πετυχαίνει αποσυμπίεση 3 φορές πιο γρήγορη από το χρόνο που δαπανά μια απλή λειτουργία Memcpy του λειτουργικού.

Το μόνο αρνητικό στην περίπτωσή αυτή είναι ότι δεν διατίθεται έκδοση assembly για αρχτιτεκτονική σαν τη δική μας, δηλαδή, x64.

Σε περίπτωση που κάποιος θέλει να χρησιμοποιήσει τον LZO,αρκεί να κάνει τα παρακάτω:
Έστω ότι θέλει να συμπιέσει δεδομένα με τον αλγόριθμο LZO1X-1:

i)κάνει include την <lzo/lzo1x.h>
ii)καλεί την lzo_init()
iii)συμπιέζει η αντίστοιχα αποσυμπιέζει τα δεδομένα με την lzo1x_1_compress()
η lzo1x_1_compress()
iv) κάνει link την εφαρμογή με τη βιβλιοθήκη LZO

Λίγα λόγια για την ονοματολογία των διάφορων αλγορίθμων (απο)-συμπίεσης που υπάρχουν στη βιβλιοθήκη LZO και την επίδοσή τους:

Το όνομα κάθε αλγορίθμου είναι της μορφής LZOxx-N,όπου xx είναι το κυρίως όνομα του αλγορίθμου και N είναι το επίπεδο συμπίεσης. Για τιμές του N 1-9,έχουμε υψηλή ταχύτητα συμπίεσής χρησιμοποιώντας 64KB extra χώρο για τη συμπίεση. Όταν το N είναι 99 έχουμε καλύτερα επίπεδα συμπίεσης χρησιμοποιώντας 256KB επιπλέον μνήμη. Τέλος όταν το N είναι 999 η συμπίεση χαρακτηρίζεται σχεδόν βέλτιστη-κοστίζει βέβαια αρκετά σε χρόνο συμπίεσης και είναι κατάλληλη για παραγωγή pre-compressed δεδομένων.

Εμείς,ενδιαφερόμαστε για την τελευταία περίπτωση όπου μια φορά θα συμπιέσουμε τον πίνακα σε δεδομένη χρονική στιγμή και θα τον έχουμε έτσι αποθηκευμένο. Από εκεί κι έπειτα θα τον αποσυμπιέζουμε κατά βούληση όταν θέλουμε να τρέξουμε την εφαρμογή μας.

Παρακάτω φαίνεται ο πίνακας των διάφορων αλγορίθμων συμπίεσης-αποσυμπίεσης κι η επίδοση τους.

Σημείωση:

- CxB είναι ο αριθμός των blocks και
- K/s είναι η ταχύτητα σε KB/s αποσυμπιεσμένων δεδομένων

Algorithm	Length	CxB	ComLen	%Remn	Bits	Com K/s	Dec K/s
memcpy()	224401	1	224401	100	8	60956.83	59124.58
LZO1-1	224401	1	117362	53.1	4.25	4665.24	13341.98
LZO1-99	224401	1	101560	46.7	3.73	1373.29	13823.4
LZO1A-1	224401	1	115174	51.7	4.14	4937.83	14410.35
LZO1A-99	224401	1	99958	45.5	3.64	1362.72	14734.17
LZO1B-1	224401	1	109590	49.6	3.97	4565.53	15438.34
LZO1B-2	224401	1	106235	48.4	3.88	4297.33	15492.79
LZO1B-3	224401	1	104395	47.8	3.83	4018.21	15373.52
LZO1B-4	224401	1	104828	47.4	3.79	3024.48	15100.11
LZO1B-5	224401	1	102724	46.7	3.73	2827.82	15427.62
LZO1B-6	224401	1	101210	46	3.68	2615.96	15325.68
LZO1B-7	224401	1	101388	46	3.68	2430.89	15361.47
LZO1B-8	224401	1	99453	45.2	3.62	2183.87	15402.77
LZO1B-9	224401	1	99118	45	3.6	1677.06	15069.6
LZO1B-99	224401	1	95399	43.6	3.48	1286.87	15656.11
LZO1B-999	224401	1	83934	39.1	3.13	232.4	16445.05
LZO1C-1	224401	1	111735	50.4	4.03	4883.08	15570.91
LZO1C-2	224401	1	108652	49.3	3.94	4424.24	15733.14
LZO1C-3	224401	1	106810	48.7	3.89	4127.65	15645.69
LZO1C-4	224401	1	105717	47.7	3.82	3007.92	15346.44
LZO1C-5	224401	1	103605	47	3.76	2829.15	15153.88
LZO1C-6	224401	1	102585	46.5	3.72	2631.37	15257.58
LZO1C-7	224401	1	101937	46.2	3.7	2378.57	15492.49

LZO1C-8	224401	1	100779	45.6	3.65	2171.93	15386.07
LZO1C-9	224401	1	100255	45.4	3.63	1691.44	15194.68
LZO1C-99	224401	1	97252	44.1	3.53	1462.88	15341.37
LZO1C-999	224401	1	87740	40.2	3.21	306.44	16411.94
LZO1F-1	224401	1	113412	50.8	4.07	4755.97	16074.12
LZO1F-999	224401	1	89599	40.3	3.23	280.68	16553.9
LZO1X-1(11)	224401	1	118810	52.6	4.21	4544.42	15879.04
LZO1X-1(12)	224401	1	113675	50.6	4.05	4411.15	15721.59
LZO1X-1	224401	1	109323	49.4	3.95	4991.76	15584.89
LZO1X-1(15)	224401	1	108500	49.1	3.93	5077.5	15744.56
LZO1X-999	224401	1	82854	38	3.04	135.77	16548.48
LZO1Y-1	224401	1	110820	49.8	3.98	4952.52	15638.82
LZO1Y-999	224401	1	83614	38.2	3.05	135.07	16385.4
LZO1Z-999	224401	1	83034	38	3.04	133.31	10553.74
LZO2A-999	224401	1	87880	40	3.2	301.21	8115.75

Φαίνεται ξεκάθαρα ότι ο καλύτερος αλγόριθμος,ο οποίος θα χρησιμοποιηθεί από εμάς,είναι ο LZO-1X.Ειναι ο αλγόριθμος που πετυχαίνει τη μεγαλύτερη συμπίεση και παράλληλα την ταχύτερη αποσυμπίεση.

Παραθέτω τον κώδικα που εκτελεί το SpMV για την περίπτωση που χρησιμοποιούμε τον LZO:

```
d=(double *)malloc(max2); //buffer για την αποσυμπίεση, ιδιωτικό για κάθε tread
for (i = 0; i < nr_rows; i++) {
        _y = 0.0;
        for (j = row_ptr[i]; j < row_ptr[i+1]; j++){
                if(j==next){
                        int tt=0;
                        part_index++;
                        inlen=parts_lzo->sizes_comp[part_index];
                        outlen=parts_lzo->sizes_uncomp[part_index];
                        next=j + outlen/8;
                        if (lzo1x_decompress(parts_lzo->parts_p[part_index],inlen,
                        d,&outlen,NULL) != LZO_E_OK )
                                jj=0;
                        }
                _y += d[jj] * x[col_ind[j]];
                        jj++;
        }
        y[i]+=_y;

}
```

Το SpMV χαρακτηρίζεται γενικά από την χαμηλή χρησιμοποίηση των επεξεργαστικών πόρων επειδή οι αστοχίες στην cache αναγκάζουν τους επεξεργαστές να περιμένουν μέχρι να έρθουν τα δεδομένα από τη μνήμη. Ξέρουμε όμως ότι αυτή η μεταφορά χαρακτηρίζεται από χαμηλή ταχύτητα σε σχέση πάντα με τη μεταφορά από την cache που βρίσκεται κοντά στις cpu's.

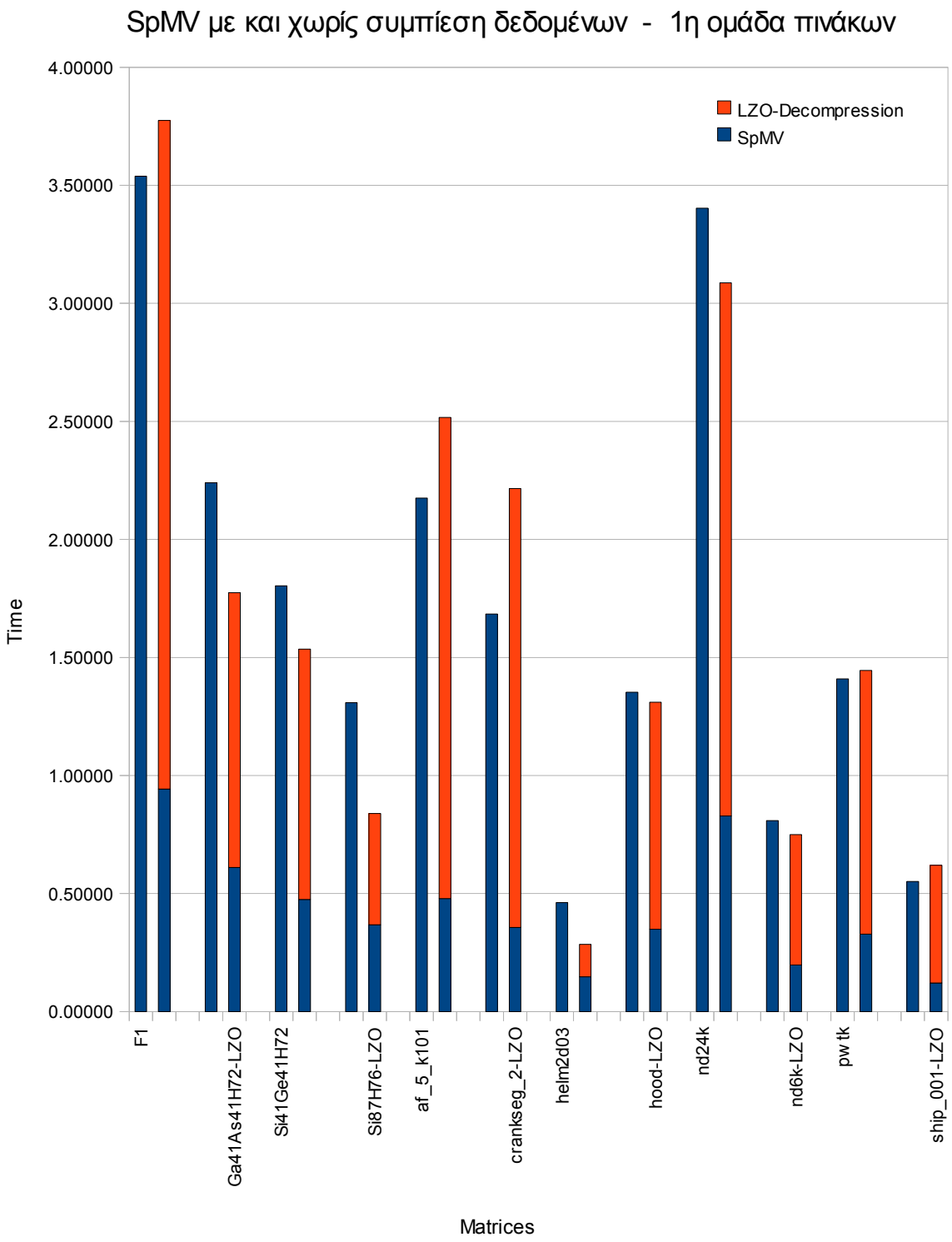
Αυτός είναι κι κύριος λόγος που το SpMV δεν έχει καλή κλιμάκωση όσο αυξάνουμε τους πυρήνες που χρησιμοποιούμε αφού όση επεξεργαστική ισχύ και να έχουμε στη διάθεσή μας, οι πυρήνες κάθε τόσο θα γίνονται idle περιμένοντας τα δεδομένα να έρθουν από τη ram.

Δηλαδή αντί να μεταφέρεται κάθε φορά που έχουμε miss στην cache ένα μέρος του διανύσματος values του πίνακα A,μεταφέρεται αντί γιαυτό το συμπιεσμένο κομμάτι το οποίο είναι πολύ μικρότερο σε μέγεθος.

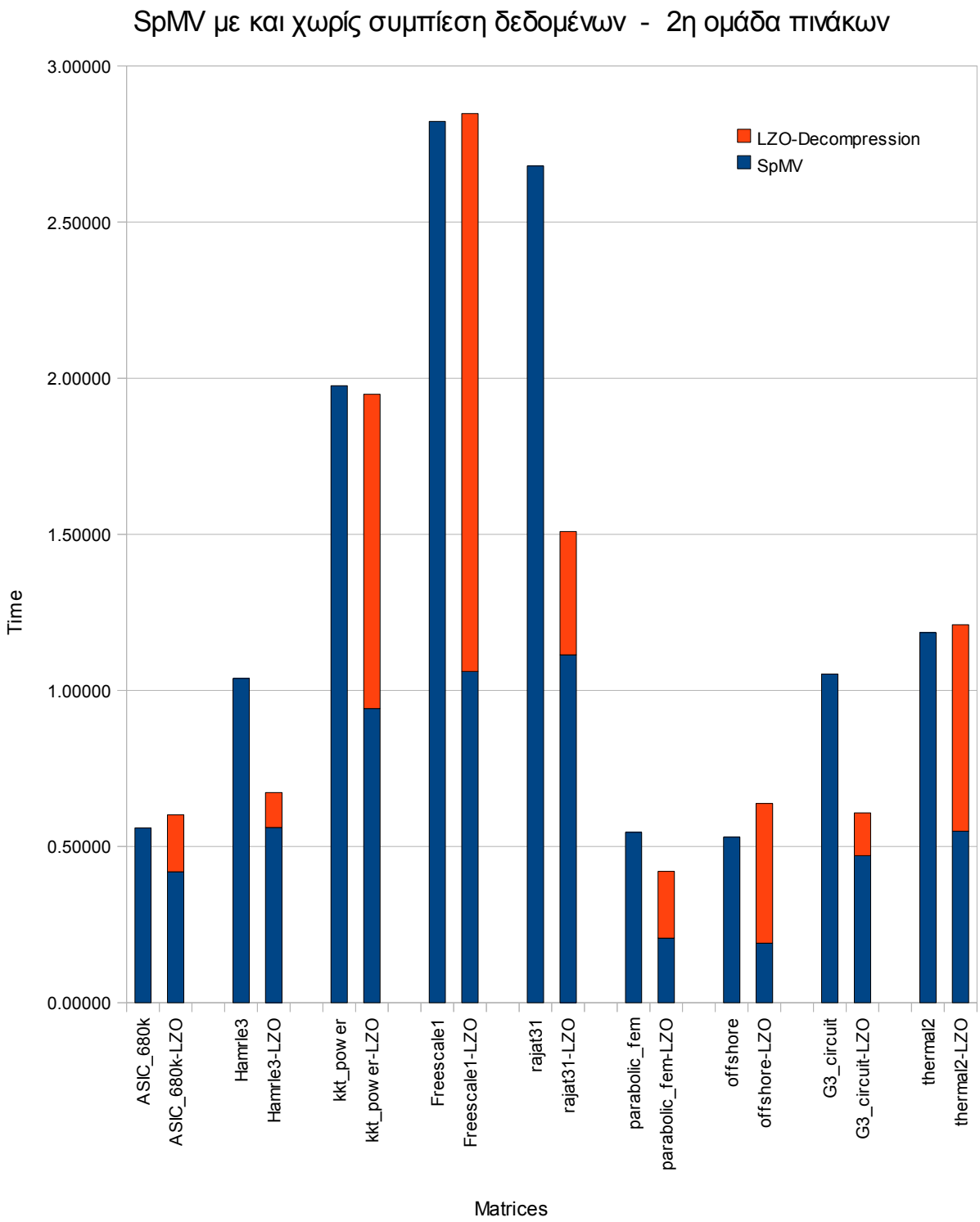
Πειραματικό Μέρος Κεφαλαίου

Στα δυο επόμενα διαγράμματα βλέπουμε το χρόνο του SpMV για υλοποίηση με και χωρίς συμπίεση LZO.

Σε μερικούς πίνακες το SpMV έχει συνολικά καλύτερη επίδοση χρησιμοποιώντας τον LZO. Σε κάθε περίπτωση το SpMV δείχνει το χρόνο του SpMV αν ο χρόνος που παίρνουν οι αποσυμπιέσεις των κομματιών δεν προσμετρηθούν. Βλέπουμε ότι ο χρόνος αυτός μειώνεται δραματικά καθώς μεταφέρεται από τη μνήμη πολύ μικρότερο κομμάτι σε σχέση με το ασυμπίεστο. Δεδομένου ότι η συμπίεση που πετυχαίνει ο συγκεκριμένος αλγόριθμος φτάνει το ratio=comp/uncomp να γίνεται περίπου 10% στους περισσότερους πίνακες,εύκολα καταλαβαίνουμε γιατί μειώνεται ο χρόνος SpMV τόσο πολύ.

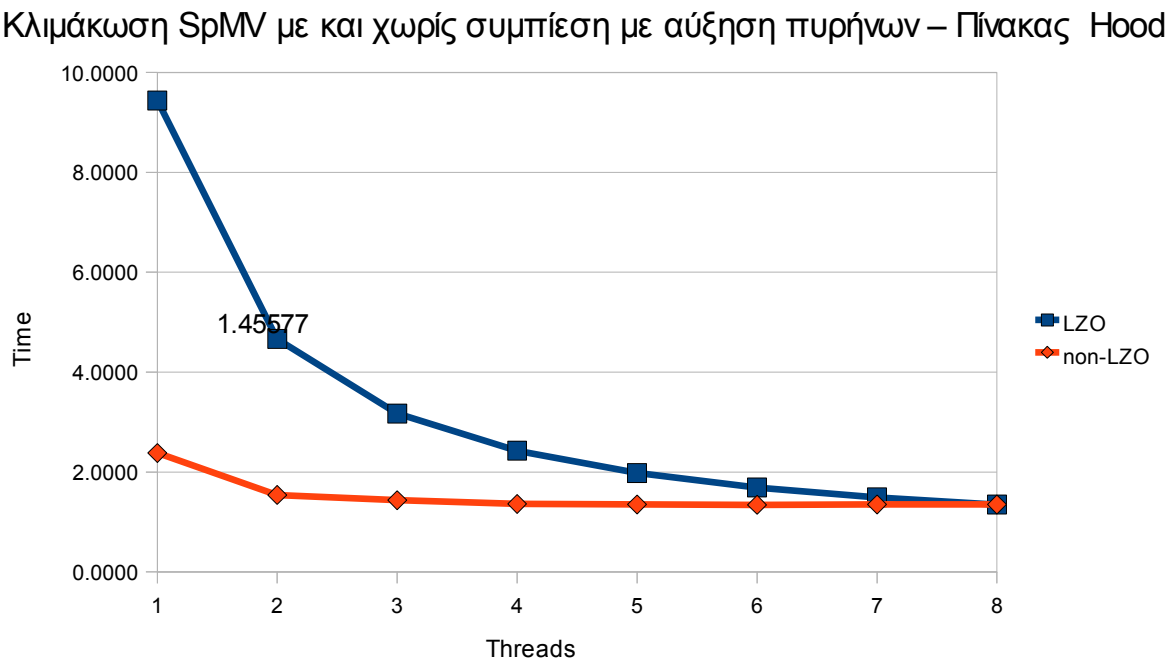


Σχήμα 6.1 - SpMV με και χωρίς συμπίεση δεδομένων - 1η ομάδα πινάκων

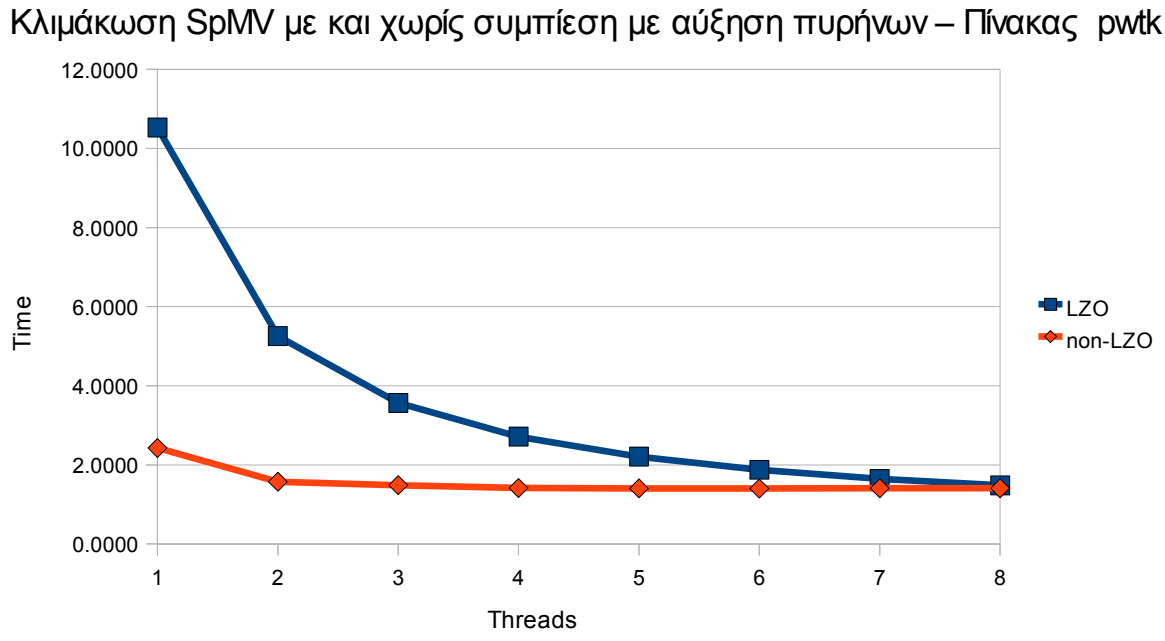


Σχήμα 6.2 - SpMV με και χωρίς συμπίεση δεδομένων - 2η ομάδα πινάκων

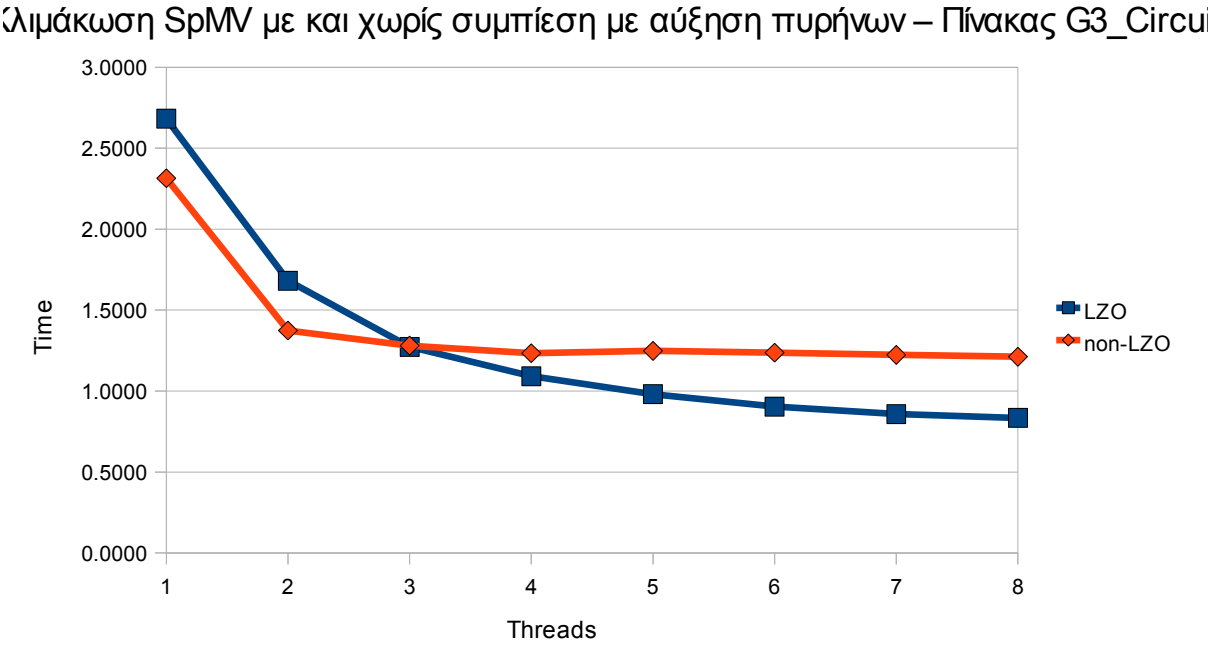
Στα παρακάτω 9 μικρά διαγράμματα βλέπουμε για κάθε έναν από 9 τυχαία επιλεγμένους πινάκες την κλιμάκωση που πετυχαίνεται σε χρόνο αυξάνοντας τον αριθμό των πυρήνων που χρησιμοποιούνται, από τη μια χωρίς συμπίεση κι από την άλλη με συμπίεση χρησιμοποιώντας τη βιβλιοθήκη LZO.



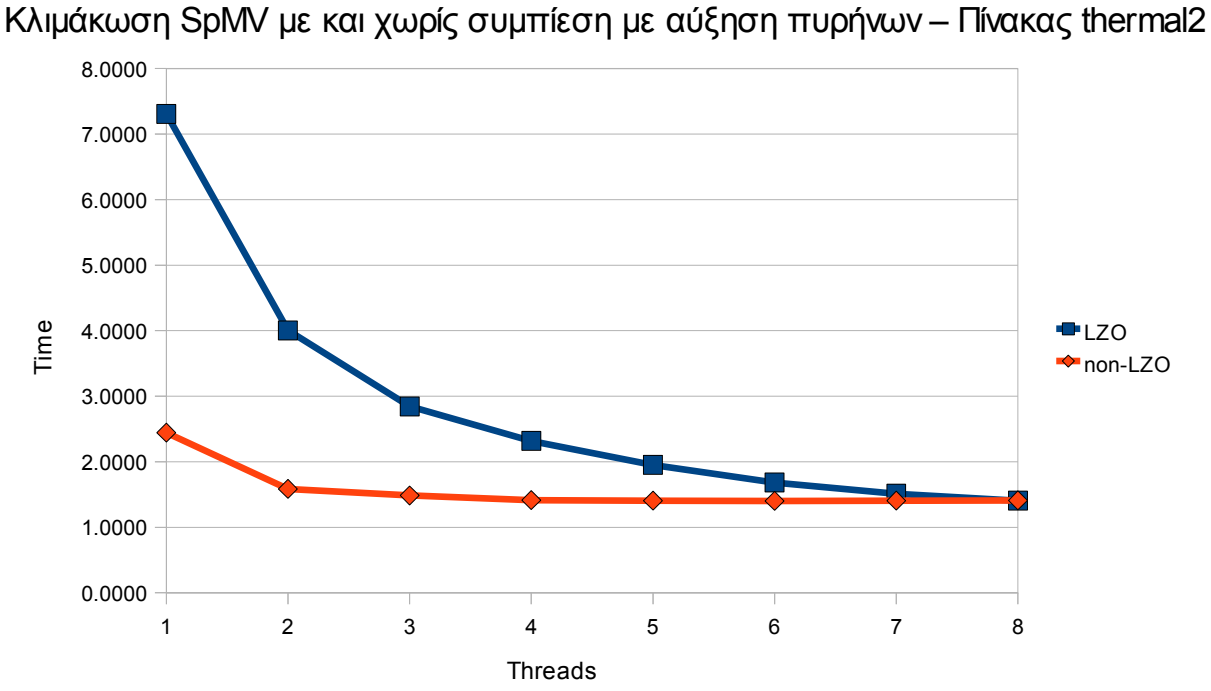
Σχήμα 6.3 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας hood



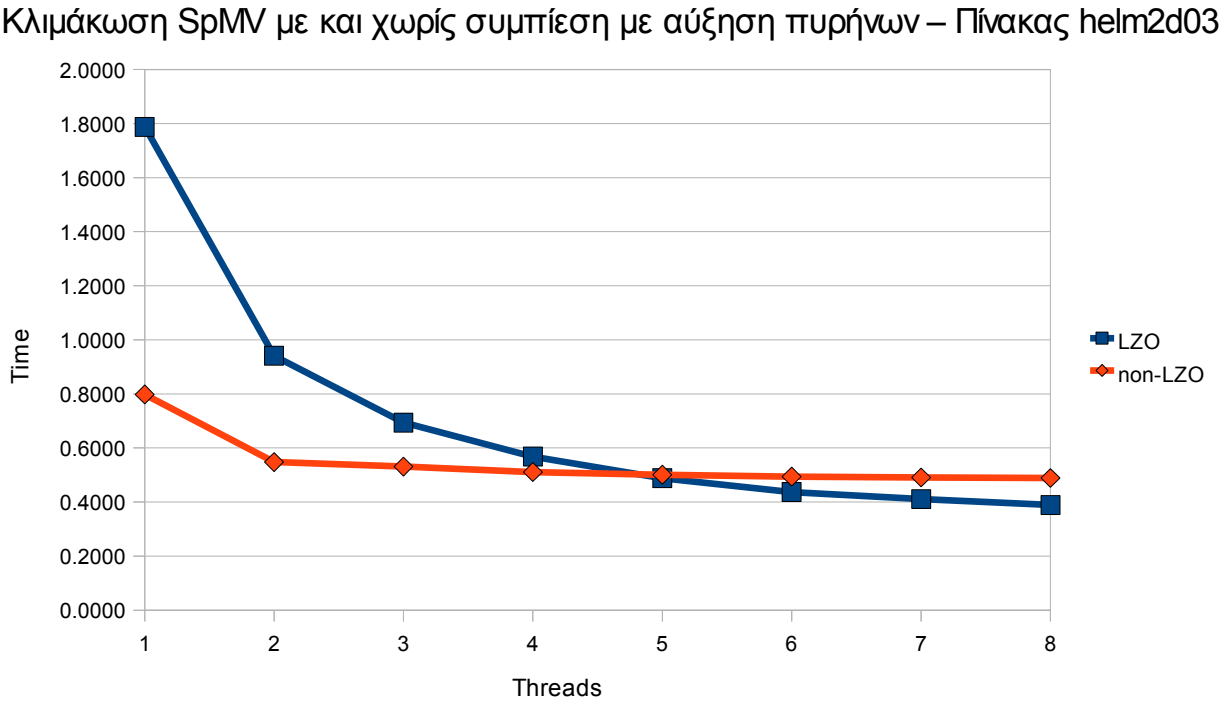
Σχήμα 6.4 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας pwtk



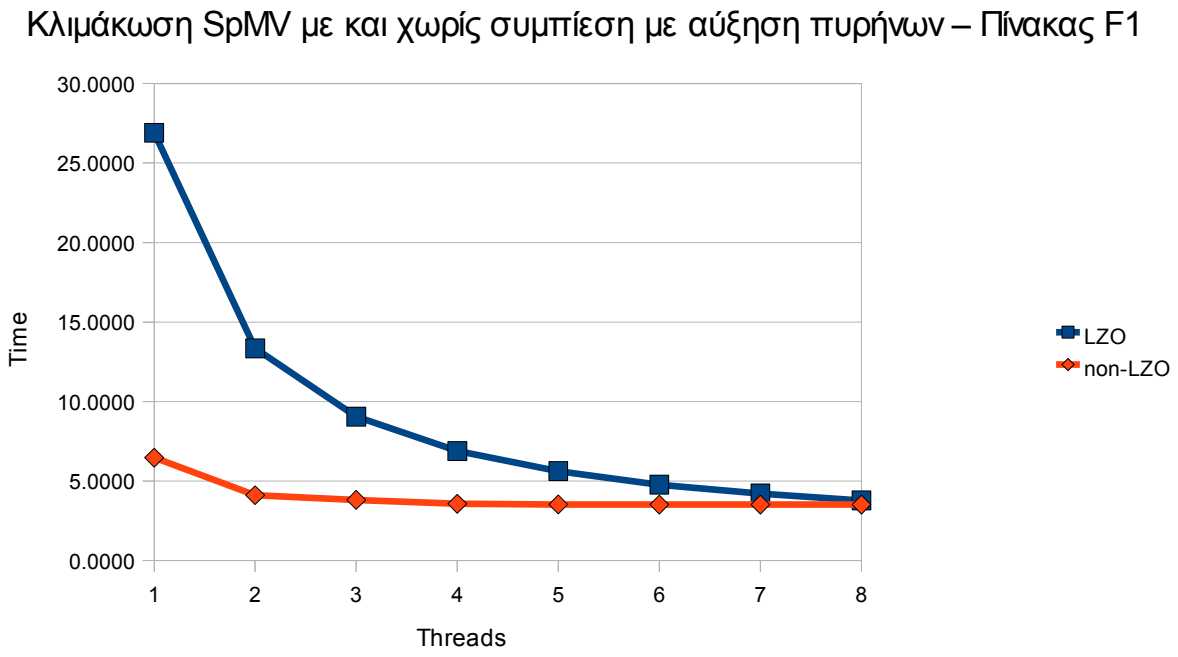
Σχήμα 6.5 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας G3_Circuit



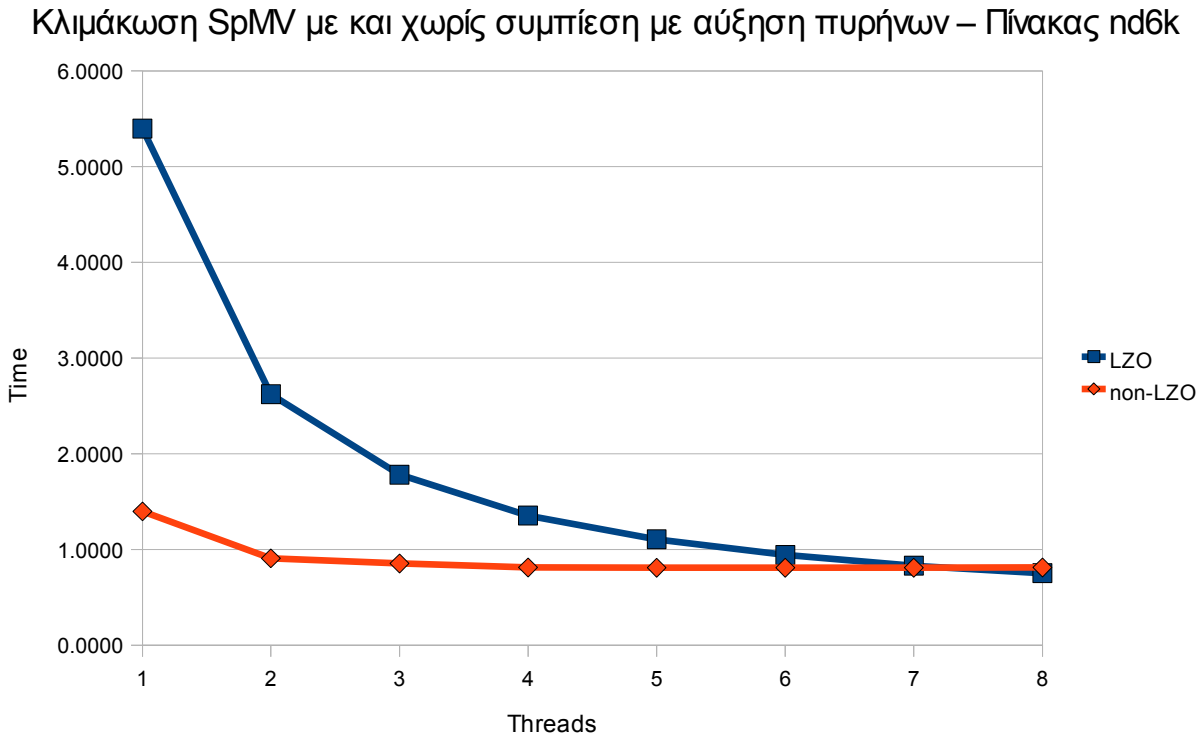
Σχήμα 6.6 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας thermal2



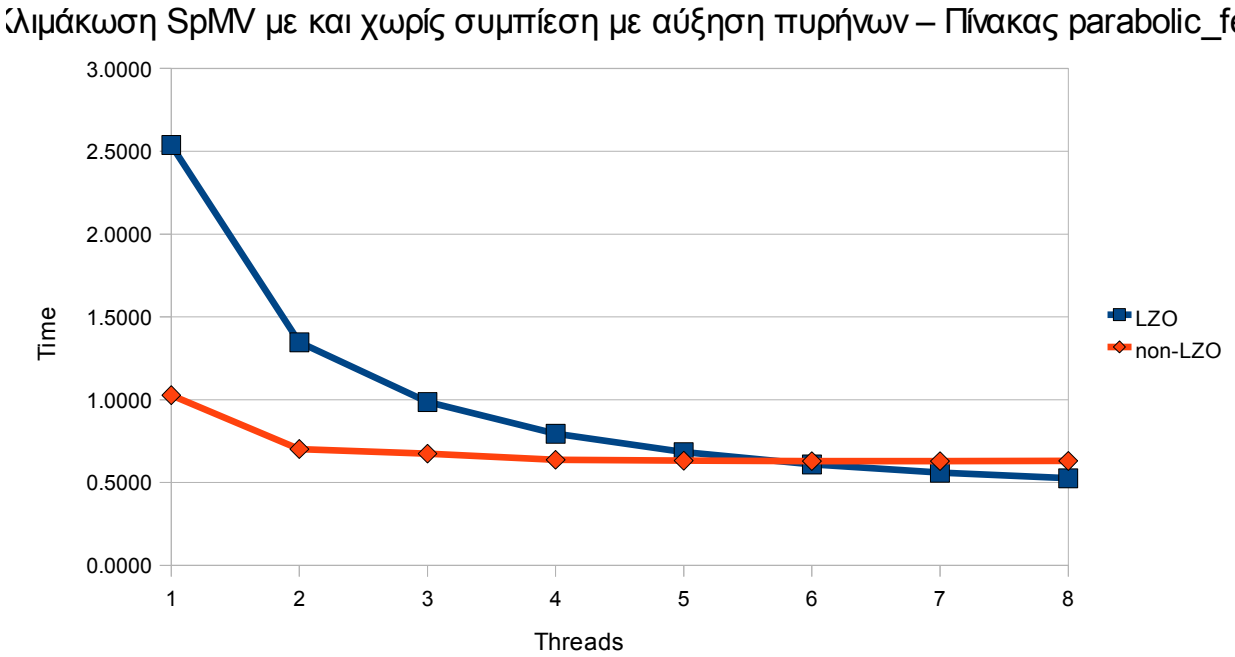
Σχήμα 6.7 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας helm2d03



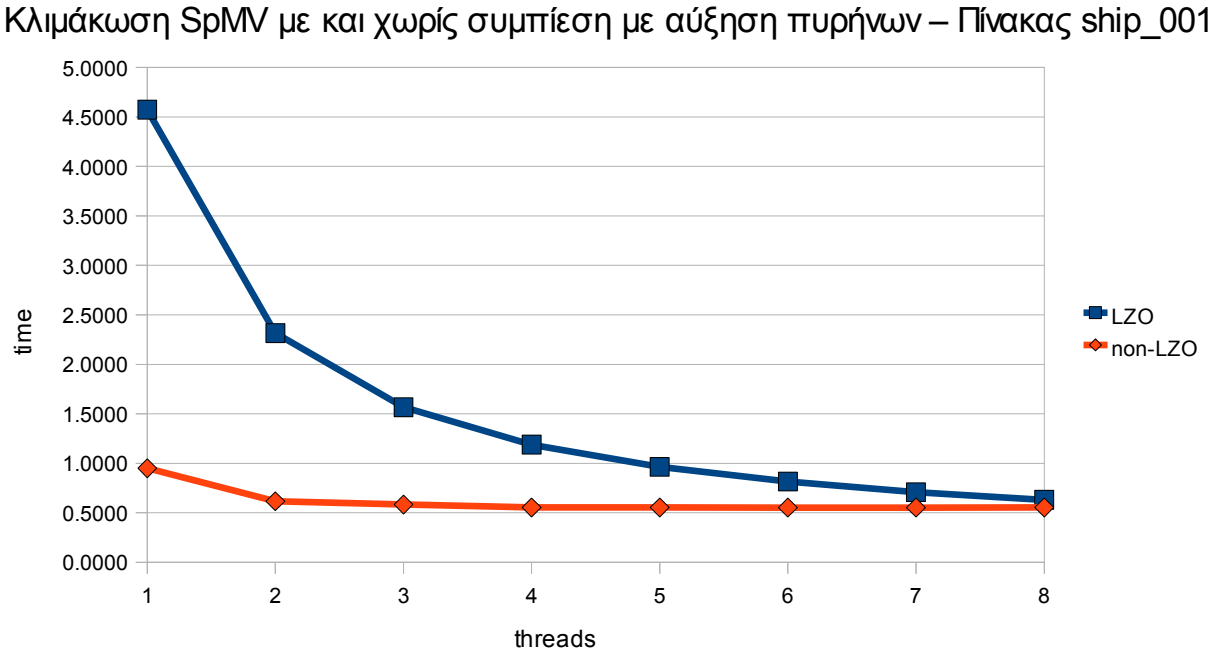
Σχήμα 6.8 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας F1



Σχήμα 6.9 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας helm2d03



Σχήμα 6.10 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας parabolic_fem



Σχήμα 6.11 – Κλιμάκωση SpMV με και χωρίς συμπίεση με αύξηση πυρήνων – Πίνακας ship_001

Βλέπουμε ότι σε αρκετές περιπτώσεις η υλοποίηση με συμπίεση είναι καλύτερη σε επίδοση από την ανταγωνιστική της. Σε κάθε περίπτωση όμως,ακόμα και στους πινάκες που δεν καταφέρνει να φέρει καλύτερη επίδοση,βλέπουμε ότι ισοφαρίζει την ανταγωνιστική χωρίς συμπίεση υλοποίηση.

Αυτό συμβαίνει όπως έχουμε πει και προηγουμένως,επειδή το SpMV έχει φραγμό που προκύπτει από τις αστοχίες στη μνήμη. Έτσι όσα νήματα και να έχουμε στη διάθεση μας να εκτελούν τον πολλαπλασιασμό αυτά στην ουσία περιμένουν τα δεδομένα να έρθουν από τη μνήμη. Με τη χρήση συμπίεσης μειώνουμε αυτήν την αναμονή αφοί έρχονται πλέον μικρότερα κομμάτια δεδομένων και βάζουμε τους πυρήνες του επεξεργαστή να αποσυμπιέζουν δηλαδή να κάνουν δουλειά. Αυτό το trade-off φαίνεται ότι αποφέρει κέρδος.

Κεφάλαιο 7ο : Επίλογος

Μελετήσαμε χρησιμοποιώντας μια επαναληπτική μέθοδο σύγκλισης τη συμπεριφορά ενός από τα σημαντικότερα υπολογιστικά προβλήματα τη εποχής – το SpMV.

Είδαμε πως κλιμακώνει σε συστοιχία υπολογιστών (cluster) και επίσης πως κλιμακώνει μέσα σε έναν υπολογιστή όσο αυξάνουμε τους πυρήνες που έχει ο υπολογιστής αυτός.

Είδαμε ότι με την παραλληλοποίηση στα ενδότερα ενός υπολογιστή ο φραγμός που σχεδόν απαγορεύει την κλιμάκωση στην επίδοση είναι οι αστοχίες της cache κι η χαμηλή σχετικά με την ταχύτητα του επεξεργαστή ταχύτητα μεταφοράς δεδομένων από την κυρία μνήμη.

Γι' αυτό το λόγο υλοποιήσαμε μέσω της βιβλιοθήκης LZO συμπίεση των δεδομένων ώστε αυτά να έρχονται πιο γρήγορα από την κύρια μνήμη. Είδαμε ότι παρά την καθυστέρηση λόγω της αποσυμπίεσης το πρόβλημα τώρα έχει κλιμάκωση και μπορούμε να πούμε ότι με τη συμπίεση σε αρκετές περιπτώσεις έχουμε καλύτερη επίδοση.

Επίσης εξετάσαμε 2 συμμετρικές εκδόσεις του πυρήνα SpMV για συμμετρικούς πίνακες κι είδαμε ότι η παραλληλοποίηση σε σύστημα κατανεμημένης μνήμης δεν δουλεύει γρήγορα λόγω μεγάλου κόστους επικοινωνίας. Γιαυτό το λόγο απορρίψαμε τη συμμετρική υλοποίηση του SpMV.