ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συστήματα Παράλληλης Επεξεργασίας

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Θεοδωρόπουλος Δ. Παναγιώτης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός Ε.Μ.Π.

Επιβλέπων καθηγητής: Τσανάκας Παναγιώτης (Καθηγητής Ε.Μ.Π.)

ΑΘΗΝΑ 2008

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Διδακτορική Διατριβή

Επιβλέπων Καθηγητής:

Τσανάκας Παναγιώτης,	Καθηγητής Ε.Μ.Πολυτεχνείου

Συμβουλευτική Επιτροπή:

Τσανάκας Παναγιώτης,	Καθηγητής Ε.Μ.Πολυτεχνείου
Παπακωνσταντίνου Γεώργιος,	Καθηγητής Ε.Μ.Πολυτεχνείου
Σκορδαλάκης Εμμανουήλ,	Καθηγητής Ε.Μ.Πολυτεχνείου

Εξεταστική Επιτροπή:

Τσανάκας Παναγιώτης,	Καθηγητής Ε.Μ.Πολυτεχνείου
Παπακωνσταντίνου Γεώργιος,	Καθηγητής Ε.Μ.Πολυτεχνείου
Σταφυλοπάτης Ανδρέας-Γεώργιος,	Καθηγητής Ε.Μ.Πολυτεχνείου
Πεκμεστζή Κιαμάλ,	Καθηγητής Ε.Μ.Πολυτεχνείου
Κοζύρης Νεκτάριος,	Επ. Καθηγητής Ε.Μ.Πολυτεχνείου
Οικονομάκος Γεώργιος,	Λέκτορας Ε.Μ.Πολυτεχνείου
Νικολόπουλος Σταύρος,	Καθηγητής Πανεπιστημίου Ιωαννίνων

στους γονείς μου
Δημοσθένη και Ελένη

στα αδέλφια μου
Ασημίνα και Γιάννη

στα ανίψια μου
Μαρία, Ελένη, Στάθη

ΠΕΡΙΛΗΨΗ

Η εξέλιξη της τεχνολογίας των υπολογιστών ήταν αλματώδης τις τελευταίες δύο δεκαετίες. Εξίσου θεαματική είναι ωστόσο η αύξηση των απαιτήσεων σε υπολογιστική ισχύ. Για την αντιμετώπιση των απαιτητικών και χρονοβόρων αλγορίθμων εισάγουμε την παράλληλη επεξεργασία. Η παράλληλη επεξεργασία έχει χρησιμοποιηθεί για πολλά χρόνια, κυρίως στους υπολογισμούς υψηλών επιδόσεων (*high performance computing*), αλλά το ενδιαφέρον για αυτήν έχει γίνει ακόμα μεγαλύτερο τα τελευταία χρόνια, λόγω των φυσικών περιορισμών που εμποδίζουν την παραπέρα κλιμάκωση της συχνότητας των επεξεργαστών. Έτσι, έχει πρόσφατα καταστεί το κυρίαρχο πρότυπο στην αρχιτεκτονική υπολογιστών, κυρίως με τη μορφή πολυπύρηνων επεξεργαστών (*multicore processors*).

Η συγγραφή παράλληλων προγραμμάτων είναι όμως σημαντικά δυσκολότερη από εκείνη των αντίστοιχων ακολουθιακών, επειδή η παραλληλία στην εκτέλεση εισάγει αρκετές νέες κατηγορίες πιθανών σφαλμάτων του λογισμικού. Επιπλέον, ο συγχρονισμός και το κόστος επικοινωνίας μεταξύ των παραλλήλων διεργασιών είναι συνήθως τα μεγαλύτερα εμπόδια για να έχουμε καλές επιδόσεις και ικανοποιητικές επιταχύνσεις από τα παράλληλα προγράμματα. Τέλος, παρά την εργασία δεκαετιών από ερευνητές στον τομέα των μεταγλωττιστών, η αυτόματη παραλληλοποίηση ακολουθιακών προγραμμάτων είχε περιορισμένη μόνο επιτυχία. Οι βασικές γλώσσες παράλληλου προγραμματισμού παραμένουν είτε ρητά παράλληλες ή, στην καλύτερη περίπτωση, εν μέρει κατευθυνόμενες, με τον προγραμματιστή να δίνει οδηγίες στο μεταγλωττιστή για την παραλληλοποίηση του κώδικα.

Αντικείμενο της διατριβής είναι η εισαγωγή και παρουσίαση στον προγραμματιστή τρόπων διευκόλυνσης του προγραμματισμού σε συστήματα παράλληλης επεξεργασίας, καθώς και η αποδοτική υλοποίηση των αντίστοιχων μηχανισμών. Το πρόβλημα προσεγγίζεται από δύο διαφορετικές οπτικές γωνίες, δίνοντας στον προγραμματιστή τη μέγιστη ευελιξία για να επιτύχει τα βέλτιστα αποτελέσματα από το παράλληλο πρόγραμμα.

Η πρώτη προσέγγιση είναι από την πλευρά του συστήματος, ώστε να δοθούν στον προγραμματιστή παραλλήλων εφαρμογών επιλογές για ευκολότερη χειρόγραφη συγγραφή παράλληλου κώδικα και οδήγησε στην υλοποίηση του μηχανισμού των καθολικών σηματοφορέων. Ο μηχανισμός αυτός έχει υλοποιηθεί και μεταφερθεί σε διάφορα παράλληλα περιβάλλοντα, με κυριότερα τις πλατφόρμες παράλληλου

προγραμματισμού PVM και MPI και είναι διαθέσιμος σαν μια βιβλιοθήκη της γλώσσας C (*runtime library*) και μερικά επιπρόσθετα αρχεία ορισμών (*include files*). Με τη χρήση του ο προγραμματιστής των παραλλήλων εφαρμογών μπορεί να αντιμετωπίσει τα προβλήματα συγχρονισμού παραλλήλων διεργασιών, αμοιβαίου αποκλεισμού κρισίμων τμημάτων και διάθεσης πόρων του συστήματος.

Η δεύτερη προσέγγιση είναι από την πλευρά του μεταγλωττιστή, ώστε μια κατηγορία ακολουθιακών αλγορίθμων να μπορούν να μεταφράζονται αυτόματα σε ισοδύναμο παράλληλο κώδικα και οδήγησε στην υλοποίηση του εργαλείου CRONUS. Η πλατφόρμα CRONUS αποτελεί ένα αυτοματοποιημένο εργαλείο για την παραλληλοποίηση γενικευμένων φωλιασμένων βρόχων που βασίζεται σε μεθόδους υπολογιστικής γεωμετρίας. Οι φωλιασμένοι βρόχοι περιλαμβάνουν σύνθετα σώματα βρόχων (αναθέσεις, συνθήκες, επαναλήψεις) και επιδεικνύουν ομοιόμορφες εξαρτήσεις μέσω του σώματος του βρόχου. Ο νεωτερισμός του εργαλείου CRONUS είναι διπλός: πρώτον, προσδιορίζει το ιδανικό υπερ-επίπεδο δρομολόγησης χρησιμοποιώντας τον αλγόριθμο QuickHull, ο οποίος είναι πιο αποδοτικός από προηγούμενα χρησιμοποιηθείσες μεθόδους και δεύτερον, υλοποιεί έναν απλό και γρήγορο δυναμικό κανόνα, που ονομάζουμε Διαδοχική Δυναμική Δρομολόγηση (SDS), για τη δρομολόγηση κατά το χρόνο εκτέλεσης των επαναλήψεων του βρόχου κατά μήκος του ιδανικού υπερ-επιπέδου. Αυτή η πολιτική δρομολόγησης ενισχύει την τοπικότητα των δεδομένων και βελτιώνει το συνολικό χρόνο εκτέλεσης. Το CRONUS παρέχει μια αποτελεσματική προγραμματιστική βιβλιοθήκη για το χρόνο εκτέλεσης (*runtime library*), ειδικά σχεδιασμένη για να ελαχιστοποιεί το κόστος επικοινωνίας και λειτουργεί τόσο σε συστήματα μοιραζόμενης μνήμης όσο και σε συστήματα κατανεμημένης μνήμης. Χρησιμοποιεί για συγχρονισμό και ανταλλαγή δεδομένων δύο διαφορετικά μοντέλα, με το πρώτο να βασίζεται αποκλειστικά και μόνο σε κλήσεις ανταλλαγής μηνυμάτων, ενώ το δεύτερο συνδυάζει το μηχανισμό των καθολικών σηματοφορέων για συγχρονισμό και ασύγχρονων μηνυμάτων για ανταλλαγή δεδομένων. Τέλος, σε πραγματικά παραδείγματα επιτυγχάνει σημαντικές επιταχύνσεις και αποδίδει πολύ καλύτερα από άλλα ανάλογα εργαλεία της βιβλιογραφίας.

ABSTRACT

The evolution of computer technology has been rapid over the past two decades. Equally striking, however, is the increase of needs in computing power. To meet the demanding and lengthy algorithms we introduce parallel processing. Parallel processing has been used for many years, mainly in the field of high performance computing, but interest in it has become even greater in recent years due to physical constraints that prevent further escalation of processors' frequency. Thus, it has recently become the dominant standard in computer architecture, mainly in the form of multicore processors.

The coding of parallel programs is nonetheless significantly more difficult than that of the corresponding sequential, as the parallelism in the execution introduces several new categories of software errors. In addition, synchronization and communication costs between the parallel processes are usually the greatest obstacles in order to achieve good performance and decent acceleration from parallel programs. Finally, despite decades of work by researchers in the field of compilers, automatic parallelization of sequential programs had only limited success. The main parallel programming languages are explicitly parallel or, at best, partially driven, with the developer giving the compiler instructions for the parallelization of the code.

The purpose of this thesis is the introduction and presentation to the developer of ways to facilitate programming in parallel processing systems and the efficient implementation of the respective mechanisms. The problem is approached from two different angles, giving the developer maximum flexibility in order to achieve the best results from the parallel program.

The first approach is from the system level, in order to give the developer options for handwriting parallel code more easily and led to the implementation of the global semaphores mechanism. This mechanism has been implemented and ported to several different parallel programming environments, with the main being the parallel platforms PVM and MPI and is available as a C language library (*runtime library*) and some additional definitions files (*include files*). By using it, the parallel applications developer can address the problems of synchronizing parallel processes, mutual exclusion of critical sections and system resource allocation.

The second approach is from the perspective of the compiler, so that a category of sequential algorithms can be automatically translated into equivalent parallel code and led

to the implementation of the tool CRONUS. The CRONUS platform is an automated tool for the parallelization of general nested loops and is based on methods of computational geometry. Nested loops include complex loop bodies (assignments, conditionals, repetitions) and exhibit uniform dependencies through the body of the loop. The novelty of the CRONUS tool is twofold: firstly, it determines the optimal scheduling hyperplane using the QuickHull algorithm, which is more efficient than previously used methods and secondly, it implements a simple and efficient dynamic rule we call Successive Dynamic Scheduling (SDS), for the runtime scheduling of the loop iterations along the optimal hyperplane. This scheduling policy enhances data locality and improves the overall execution time. CRONUS provides an efficient runtime library, specifically designed to minimize the cost of communication and works both on shared memory and distributed memory systems. It uses two different models for data exchange and synchronization, with the first to rely solely on message passing calls, while the second combines the mechanism of global semaphores for synchronization and asynchronous message passing for data exchange. Finally, in real examples, it achieves significant speedups and performs much better than other similar tools of the literature.

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ

ΑΝΤΙΣΤΟΙΧΙΑ ΕΛΛΗΝΙΚΩΝ-ΑΓΓΛΙΚΩΝ ΟΡΩΝ

Για τη διευκόλυνση του αναγνώστη, παραθέτουμε ένα συγκεντρωτικό πίνακα αντιστοίχισης των ελληνικών όρων που εμφανίζονται στο κείμενο και των αντίστοιχων αγγλικών. Έγινε προσπάθεια να ακολουθηθεί η ορολογία που χρησιμοποιείται σε αντίστοιχα ελληνικά κείμενα. Οι όροι παρατίθενται αλφαβητικά.

Πίνακας Α.1: Αντιστοιχία Ελληνικών-Αγγλικών όρων στην παρούσα διατριβή

Ελληνικός όρος	**Αγγλικός όρος**	**Συμβολισμός**
Αδιέξοδο	Deadlock	
Απογραφή εφαρμογών	Application check-pointing	
Διάνυσμα εξάρτησης	Dependence vector	$\vec{d}$
Διεργασία/Νήμα	Process/Thread	
Δρομολόγηση/Χρονοδρομολόγηση	Scheduling	
Εξαρτήσεις	Dependencies	d_i, $1 \leq i \leq m$
Καθολικός σηματοφορέας	Global semaphore	
Κυρτό περίγραμμα	Convex Hull	
Μελέτη περίπτωσης	Case study	
Μετανάστευση διεργασιών	Process migration	
Μετροπρογράμματα	Benchmarks	
Ομοιόμορφες εξαρτήσεις	Uniform dependencies	
Περιγραφητής σηματοφορέα	Semaphore descriptor	
Πληθικότητα	Cardinality	
Ρυθμαπόδοση	Throughput	
Σηματοφορέας	Semaphore	
Σημείο χώρου δεικτών	Index point	i
Σύστημα κατανεμημένης μνήμης	Distributed memory system	
Σύστημα μοιραζόμενης μνήμης	Shared memory system	

Συστοιχία σταθμών εργασίας	Cluster of workstations	
Τερματικό σημείο	Terminal point	U
Υπερ-επίπεδο	Hyperplane	Π
Υποσύστημα	Module	
Φωλιασμένος βρόχος	Nested loop	
Χρόνος εκτέλεσης	Execution time	
Χρόνος μετάφρασης	Compilation time	
Χώρος δεικτών	Index space	J

ΑΝΤΙ ΠΡΟΛΟΓΟΥ

Η διδακτορική αυτή διατριβή εκπονήθηκε στον τομέα *Τεχνολογίας Πληροφορικής και Υπολογιστών* της σχολής *Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών* του Εθνικού Μετσοβίου Πολυτεχνείου. Περιλαμβάνει την έρευνα και τα αποτελέσματα που προέκυψαν κατά τη διάρκεια των μεταπτυχιακών μου σπουδών στο *Εργαστήριο Υπολογιστικών Συστημάτων* (cslab) του εν λόγω τμήματος και το οποίο κατά την εισαγωγή μου ως μεταπτυχιακού φοιτητή ήταν γνωστό ως *Εργαστήριο Ψηφιακών Συστημάτων και Υπολογιστών* (dsclab), με γνωστικό αντικείμενο τα Συστήματα Παράλληλης Επεξεργασίας.

Μέσα από τις γραμμές αυτές θα ήθελα να εκφράσω τις ευχαριστίες μου σε ένα πλήθος ανθρώπων που, όχι μόνο με βοήθησαν, αλλά συνέβαλαν αποφασιστικά στην πραγματοποίηση της παρούσας διατριβής. Πρωτίστως θα ήθελα να ευχαριστήσω τα μέλη της τριμελούς συμβουλευτικής μου επιτροπής. Τον επιβλέποντα καθηγητή μου κ. Παναγιώτη Τσανάκα για την εμπιστοσύνη που μου επέδειξε σε αναπτυξιακά έργα, τον κ. Γεώργιο Παπακωνσταντίνου για την επιστημονική επίβλεψη που μου παρείχε και την πολύτιμη βοήθειά του στον ερευνητικό τομέα, αλλά κυρίως για την ακεραιότητα και την ηθική του και τέλος, τον κ. Εμμανουήλ Σκορδαλάκη, που συνεχώς με παρότρυνε να ολοκληρώσω τη διατριβή μου, ιδιαίτερα τα τελευταία χρόνια και το ενδιαφέρον του οποίου για την πρόοδό μου είναι κάτι παραπάνω από αξιοσημείωτο.

Ιδιαιτέρως θα ήθελα να ευχαριστήσω τους συναδέρφους με τους οποίους είχα στενή και άψογη συνεργασία και με βοήθησαν τα μέγιστα στην ολοκλήρωση της διατριβής μου. Και πρώτα από όλους, τον καλό φίλο Θοδωρή Ανδρόνικο, ο οποίος μετά το τέλος της στρατιωτικής μου θητείας με μάζεψε και με έστρωσε ξανά στη δουλειά για τη συνέχιση της διατριβής μου. Είναι σχεδόν βέβαιο πως χωρίς τη συμβολή του θα είχα εγκαταλείψει κάθε απόπειρα για την περάτωσή της. Επίσης, τη Florina Monica Ciorba που ακούραστα και με θαυμαστό ζήλο συντόνιζε την ομάδα μας και οργάνωνε την έρευνα μας. Τέλος, τους Μάριο Καλαθά και Δημήτρη Καμμενόπουλο, με τους οποίους είχα αρμονική συνεργασία στην ανάπτυξη κώδικα για την επιβεβαίωση των θεωρητικών συμπερασμάτων και τη διεξαγωγή συναφών μετρήσεων.

Ξεχωριστή θέση στις ευχαριστίες μου έχει ο καλός φίλος και συνάδερφος Γιάννης Δροσίτης, με τον οποίο περάσαμε μαζί αμέτρητες ώρες στο εργαστήριο, εργαζόμενοι σε ερευνητικά και αναπτυξιακά θέματα. Η επιμονή μου να κάτσει στο διπλανό γραφείο έχει δικαιωθεί εδώ και καιρό, καθώς πρόκειται για έναν ξεχωριστό και σπάνιο άνθρωπο. Η φιλία μας αποτελεί για μένα το μεγαλύτερο κέρδος από την παρουσία μου εργαστήριο. Θα ήταν παράληψη να μην αναφερθώ και στους υπόλοιπους συναδέρφους του εργαστηρίου με

τους οποίους είχαμε κοινή πορεία, με πρώτο τον Ανδρέα Κουλούρη που πάντα έκανε πιο ευχάριστα τα διαλλείματα από τη δουλειά, καθώς και τους Νατάσσα Μανουσοπούλου, Νίκο Βλάσση, Φειδία Μπουρλά, Γιώργο Οικονομάκο, Γιώργο Μανή, Νάσο Παπακώστα, Φώτη Σταματελόπουλο, Γιάννη Παναγόπουλο και Χρήστο Παυλάτο.

Θέλω επίσης να εκφράσω τις ευχαριστίες μου στους φίλους και συμφοιτητές με τους οποίους συμπληρώσαμε πάνω από μια δεκαετία στους χώρους του Πολυτεχνείου, μα περισσότερο στο φίλο και κουμπάρο μου Κώστα Βερύκιο, που με τίμησε ιδιαίτερα με την επιλογή του αυτή. Στον καλό φίλο Κώστα Παγώνα, που γνωριστήκαμε κατά τη εκπόνηση της διπλωματικής του εργασίας στο εργαστήριο και που πάντα επιδεικνύει ξεχωριστό ενδιαφέρον για μένα, αλλά και στο φίλο Ανδρέα Παπαντωνίου που συνδεθήκαμε με στενή φιλία κατά τη διάρκεια της στρατιωτικής μας θητείας στο τριεθνές του Έβρου και που θα τον αισθάνομαι για πάντα σαν αδερφό μου. Στους παλαιότερους συναδέρφους Νόντα Μασαλή, Θέμη Παταβάλη, Κώστα Πραματάρη και Παντελή Ορφανίδη, όπως και στους νεότερους συναδέρφους Φώτη Μαρκόπουλο και Αλέξανδρο Λέγκα, καθώς η συνεργασία μαζί τους σε επαγγελματικό επίπεδο ήταν πάντα απροβλημάτιστη και βοήθησαν όχι μόνο στη δημιουργία ενός ευχάριστου εργασιακού περιβάλλοντος, αλλά και στο μετα-σχηματισμό του ψυχοφθόρου εργασιακού χρόνου σε χρόνο δημιουργίας και διασκέδασης, δίνοντάς μου δυνάμεις για να συνεχίσω την ερευνητική μου προσπάθεια μετά το πέρας της εργασίας μας, κατά τις πιο προχωρημένες ώρες της ημέρας.

Η εργασία αυτή αφιερώνεται σε όσους επιμένουν να μοχθούν παρά τη μη καταξίωση των επιστημονικών τους ερευνών και σε όσους αγαπώ και νιώθω κοντά μου. Μα πάνω από όλους στην οικογένεια μου, που στάθηκε ακούραστα δίπλα μου κατά τη διαδρομή μου αυτή. Στους γονείς μου Δημοσθένη και Ελένη, οι οποίοι με ανέχθηκαν με υπομονή και αγάπη όλα αυτά τα χρόνια, καθώς ο χρόνος που αφιέρωσα σε αυτούς ήταν τα σύντομα διαλλείματα ανάμεσα στα μεγάλα κενά απουσίας μου από το σπίτι, στα αδέρφια μου Ασημίνα και Γιάννη και στα ανίψια μου Μαρία, Ελένη και Στάθη.

Αθήνα, Μάιος 2008

ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

Η παράλληλη επεξεργασία είναι μια μορφή υπολογιστικής επεξεργασίας κατά την οποία πολλές εντολές εκτελούνται ταυτόχρονα. Η παράλληλη επεξεργασία λειτουργεί με την αρχή ότι μεγάλα υπολογιστικά προβλήματα μπορούν σχεδόν πάντα να διαιρεθούν σε μικρότερα, τα οποία μπορούν να επιλύονται ταυτόχρονα ("εν παραλλήλω"). Η παράλληλη επεξεργασία παρουσιάζεται με αρκετές διαφορετικές μορφές: παραλληλία σε επίπεδο bit (*bit-level parallelism*), παραλληλία σε επίπεδο εντολών (*instruction level parallelism*), παραλληλία δεδομένων (*data parallelism*) και παραλληλία εντολών (*task parallelism*). Έχει χρησιμοποιηθεί για πολλά χρόνια, κυρίως στους υπολογισμούς υψηλών επιδόσεων (*high performance computing*), αλλά το ενδιαφέρον για αυτήν έχει γίνει ακόμα μεγαλύτερο τα τελευταία χρόνια, λόγω των φυσικών περιορισμών που εμποδίζουν την παραπέρα κλιμάκωση της συχνότητας των επεξεργαστών. Η παράλληλη επεξεργασία έχει πρόσφατα καταστεί το κυρίαρχο πρότυπο στην αρχιτεκτονική υπολογιστών, κυρίως με τη μορφή πολυπύρηνων επεξεργαστών (*multicore processors*).

Η συγγραφή παράλληλων προγραμμάτων ηλεκτρονικών υπολογιστών είναι σημαντικά δυσκολότερη από εκείνη αντίστοιχων ακολουθιακών, επειδή η παραλληλία στην εκτέλεση εισάγει αρκετές νέες κατηγορίες πιθανών σφαλμάτων του λογισμικού, εκ των οποίων πιο συχνές είναι οι συνθήκες ανταγωνισμού (*race conditions*). Ο συγχρονισμός και η επικοινωνία μεταξύ των διαφόρων διαδικασιών είναι συνήθως τα μεγαλύτερα εμπόδια για να πάρουμε καλές επιδόσεις από το παράλληλο πρόγραμμα. Η επιτάχυνση ενός προγράμματος ως αποτέλεσμα της παραλληλοποίησης του δίνεται από το νόμο του Amdahl και αποτελεί μέτρο της αποτελεσματικότητάς του.

1.1 Παράλληλος προγραμματισμός

Ένας αριθμός μοντέλων παράλληλου προγραμματισμού, παράλληλων γλωσσών προγραμματισμού, βιβλιοθηκών και APIs έχουν κατά καιρούς προταθεί ή/και

δημιουργηθεί για τον προγραμματισμό παράλληλων υπολογιστικών συστημάτων. Όλα αυτά μπορούν γενικά να χωριστούν σε κατηγορίες με βάση τις παραδοχές που κάνουν για την υποκείμενη αρχιτεκτονική μνήμης, σε μοντέλα μοιραζόμενης μνήμης, κατανεμημένης μνήμης, ή κατανεμημένης μοιραζόμενης μνήμης. Στο μοντέλο προγραμματισμού μοιραζόμενης μνήμης η επικοινωνία γίνεται με τη χρήση μεταβλητών μοιραζόμενης μνήμης (*shared variables*) ενώ το μοντέλο προγραμματισμού κατανεμημένης μνήμης χρησιμοποιεί την ανταλλαγή μηνυμάτων (*message passing*). Τα POSIX threads και το OpenMP είναι δύο από τα πιο ευρέως χρησιμοποιούμενα APIs μοιραζόμενης μνήμης, ενώ το MPI (Message Passing Interface) είναι το πιο διαδεδομένο και ευρύτατα χρησιμοποιούμενο API με χρήση ανταλλαγής μηνυμάτων.

1.1.1 Αυτόματη παραλληλοποίηση

Η αυτόματη παραλληλοποίηση ενός ακολουθιακού προγράμματος με τη χρήση ενός μεταγλωττιστή αποτελεί την τελική επιδίωξη της παράλληλης επεξεργασίας. Παρά την εργασία δεκαετιών από ερευνητές στον τομέα των μεταγλωττιστών, η αυτόματη παραλληλοποίηση είχε περιορισμένη μόνο επιτυχία. Οι βασικές γλώσσες παράλληλου προγραμματισμού παραμένουν είτε ρητά παράλληλες ή, στην καλύτερη περίπτωση, εν μέρει κατευθυνόμενες, με τον προγραμματιστή να δίνει οδηγίες στο μεταγλωττιστή για την παραλληλοποίηση του κώδικα. Λίγες πλήρως κατευθυνόμενες γλώσσες παράλληλου προγραμματισμού υπάρχουν για λογισμικό αλλά και για FPGAs, αλλά γενικά πρόκειται για εξειδικευμένες γλώσσες που δεν χρησιμοποιούνται ευρέως.

1.1.2 Σημεία απογραφής εφαρμογών

Όσο μεγαλύτερο και πιο περίπλοκο γίνεται ένα υπολογιστικό σύστημα, τόσο περισσότερα μπορούν να πάνε στραβά και τόσο μικρότερος γίνεται ο μέσος χρόνος μεταξύ βλαβών. Το ίδιο συμβαίνει και με την αύξηση της πολυπλοκότητας του λογισμικού, τόσο του λειτουργικού συστήματος όσο και των εφαρμογών. Η απογραφή εφαρμογών (*application checkpointing*) είναι μια τεχνική σύμφωνα με την οποία το σύστημα λαμβάνει μια "φωτογραφία" της εφαρμογής, δηλαδή ένα στιγμιότυπο της κατάστασης των μεταβλητών και όλων των υφιστάμενων δεσμευμένων πόρων της

εφαρμογής, ανάλογο του αντίγραφου πυρήνα (*core dump*). Οι πληροφορίες αυτές μπορούν στη συνέχεια να χρησιμοποιηθούν για να επαναφέρουμε το πρόγραμμα της εφαρμογής στη συγκεκριμένη κατάσταση, σε περίπτωση που ο υπολογιστής αποτύχει. Η απογραφή εφαρμογών σημαίνει ότι το πρόγραμμα σε περίπτωση σφάλματος θα έχει μόνο να κάνει επανεκκίνηση από το τελευταίο σημείο απογραφής και όχι από την αρχή. Για μια εφαρμογή που μπορεί να πάρει αρκετές μέρες ή και μήνες για να ολοκληρωθεί, αυτό είναι εξαιρετικά σημαντικό. Επιπλέον, τα σημεία απογραφής εφαρμογών μπορούν να χρησιμοποιηθούν επίσης για να διευκολυνθεί η διαδικασία της μετανάστευσης διεργασιών σε διαφορετικούς επεξεργαστές ή και συστήματα.

1.2 Αντικείμενο της διατριβής

Η εξέλιξη της τεχνολογίας των υπολογιστών ήταν αλματώδης τις τελευταίες δύο δεκαετίες. Εξίσου θεαματική ωστόσο είναι η αύξηση των απαιτήσεων σε υπολογιστική ισχύ. Για την αντιμετώπιση των απαιτητικών και χρονοβόρων αλγορίθμων εισάγουμε την παράλληλη επεξεργασία. Νέα προβλήματα προκύπτουν, σε επίπεδο συστημάτων και σε επίπεδο εφαρμογών. Τέτοια προβλήματα είναι ο συγχρονισμός διεργασιών, ο αμοιβαίος αποκλεισμός κρισίμων τμημάτων, η διάθεση των πόρων του παράλληλου συστήματος, αλλά και άλλα πιο εξειδικευμένα όπως ο συγχρονισμός ρολογιών για τον ορισμό παγκόσμιου χρόνου σε κατανεμημένα συστήματα.

Αντικείμενο της διατριβής είναι η εισαγωγή και παρουσίαση στον προγραμματιστή παραλλήλων εφαρμογών τρόπων διευκόλυνσης του προγραμματισμού σε συστήματα παράλληλης επεξεργασίας, προσεγγίζοντας το πρόβλημα από δύο διαφορετικές οπτικές γωνίες. Η πρώτη είναι από την πλευρά του συστήματος, ώστε να δώσουμε στον προγραμματιστή παραλλήλων εφαρμογών επιλογές για ευκολότερη χειρόγραφη συγγραφή παράλληλου κώδικα και οδήγησε στην υλοποίηση του μηχανισμού των καθολικών σηματοφορέων. Η δεύτερη είναι από την πλευρά του μεταγλωττιστή, ώστε μια κατηγορία ακολουθιακών αλγορίθμων να μπορούν να μεταφράζονται αυτόματα σε ισοδύναμο παράλληλο κώδικα και οδήγησε στην υλοποίηση του εργαλείου CRONUS, το οποίο αποτελεί μια πλατφόρμα για τη δημιουργία παράλληλου κώδικα βασισμένη σε μεθόδους υπολογιστικής γεωμετρίας. Σημειώνουμε ότι στη δεύτερη περίπτωση γίνεται επιλεκτική χρήση σε κάποιες περιπτώσεις των διευκολύνσεων που

παρέχονται από την πρώτη προσέγγιση, δηλαδή του μηχανισμού των καθολικών σηματοφορέων, για θέματα συγχρονισμού.

Με τις δύο αυτές προσεγγίσεις και τα αντίστοιχα εργαλεία που παρέχονται, ο προγραμματιστής έχει τη μέγιστη ευελιξία ώστε, είτε να διευκολυνθεί στη χειρόγραφη συγγραφή του παράλληλου κώδικα, είτε να βασιστεί σε μια αυτοματοποιημένη διαδικασία παραλληλοποίησης υφιστάμενου ακολουθιακού κώδικα.

1.3 Οργάνωση της διατριβής

Η παρούσα διατριβή προσεγγίζει την παράλληλη επεξεργασία στο επίπεδο του λογισμικού και παρουσιάζει τρόπους διευκόλυνσης του προγραμματισμού σε συστήματα παράλληλης επεξεργασίας, ενώ ασχολείται επίσης και με την αποδοτική υλοποίηση αντίστοιχων μηχανισμών.

Στο κεφάλαιο 2 παρουσιάζεται ο μηχανισμός των καθολικών σηματοφορέων, ο οποίος έχει υλοποιηθεί και μεταφερθεί σε διάφορα παράλληλα προγραμματιστικά περιβάλλοντα. Ο συνολικός μηχανισμός είναι διαθέσιμος σαν μια βιβλιοθήκη της γλώσσας C και μερικά επιπρόσθετα αρχεία ορισμών (include files). Ο προγραμματιστής των παραλλήλων εφαρμογών χρησιμοποιεί τις συναρτήσεις της βιβλιοθήκης αυτής με το συνήθη τρόπο και μπορεί να αντιμετωπίσει τα προβλήματα συγχρονισμού διεργασιών, αμοιβαίου αποκλεισμού κρισίμων τμημάτων και διάθεσης πόρων του συστήματος.

Στο κεφάλαιο 3 δίνεται το θεωρητικό υπόβαθρο για την επεξήγηση της λειτουργίας του εργαλείου CRONUS, ενός αυτοματοποιημένου εργαλείου για την παραλληλοποίηση γενικευμένων φωλιασμένων βρόχων που βασίζεται σε μεθόδους υπολογιστικής γεωμετρίας. Οι γενικευμένοι φωλιασμένοι βρόχοι περιλαμβάνουν σύνθετα σώματα βρόχων (αναθέσεις, συνθήκες, επαναλήψεις) και επιδεικνύουν ομοιόμορφες εξαρτήσεις μέσω του σώματος του βρόχου. Ο νεωτερισμός του εργαλείου CRONUS είναι διπλός: πρώτον, προσδιορίζει το ιδανικό υπερ-επίπεδο δρομολόγησης χρησιμοποιώντας τον αλγόριθμο QuickHull, ο οποίος είναι πιο αποδοτικός από προηγούμενα χρησιμοποιηθείσες μεθόδους και δεύτερον, υλοποιεί έναν απλό και γρήγορο δυναμικό κανόνα, που ονομάζουμε Διαδοχική Δυναμική Δρομολόγηση (SDS), για τη δρομολόγηση κατά το χρόνο εκτέλεσης των επαναλήψεων του βρόχου κατά

μήκος του ιδανικού υπερ-επιπέδου. Αυτή η πολιτική δρομολόγησης ενισχύει την τοπικότητα των δεδομένων και βελτιώνει το συνολικό χρόνο εκτέλεσης.

Στο κεφάλαιο 4 παρουσιάζεται η εσωτερική οργάνωση και λειτουργία του εργαλείου CRONUS και ο σωστός τρόπος χρήσης του. Το CRONUS παρέχει μια αποτελεσματική προγραμματιστική βιβλιοθήκη για το χρόνο εκτέλεσης (runtime library), ειδικά σχεδιασμένη για να ελαχιστοποιεί το κόστος επικοινωνίας και η οποία αποδίδει καλύτερα από άλλα γενικότερα συστήματα, όπως είναι η πλατφόρμα UPC του πανεπιστημίου του Berkeley.

Στο κεφάλαιο 5 παρατίθενται οι δοκιμαστικές περιπτώσεις και ερμηνεύονται τα πειραματικά αποτελέσματα για το εργαλείο CRONUS, ενώ γίνεται σύγκριση της επίδοσής του με άλλα συστήματα. Η συνολική απόδοση του εργαλείου CRONUS και της συνοδευτικής βιβλιοθήκης του, αποτιμήθηκε με μια σειρά εξαντλητικών δοκιμών. Τρεις αντιπροσωπευτικές περιπτώσεις μελετήθηκαν: ο αλγόριθμος διάχυσης Floyd-Steinberg, ο αλγόριθμος μεταβατικού κλεισίματος Transitive Closure και τέλος, ο αλγόριθμος εκτίμησης κίνησης FSBM. Τα πειραματικά αποτελέσματα επιβεβαιώνουν την αποδοτικότητα του παράλληλου κώδικα, ιδιαίτερα σε συστήματα μοιραζόμενης μνήμης σε σχέση με συστήματα κατανεμημένης μνήμης. Εξάλλου, το εργαλείο CRONUS υπερτερεί και της πλατφόρμας UPC με ποσοστό που κυμαίνεται από 5% έως και 95%, ανάλογα με το υπό εξέταση παράδειγμα.

Τέλος, τα συμπεράσματα της παρούσας διατριβής καθώς και κάποιες προτάσεις για μελλοντική εργασία παρουσιάζονται στο κεφάλαιο 6. Ακολουθεί για λόγους πληρότητας το παράρτημα, με την ανάλυση των αλγορίθμων του κεφαλαίου 3, γενικευμένων στις n διαστάσεις.

1.4 Συμβολή της διατριβής

Η συμβολή της παρούσας διατριβής μπορεί να συνοψιστεί στα παρακάτω σημεία:

- Προτείνουμε ένα μοντέλο συγχρονισμού διεργασιών σε παράλληλα και κατανεμημένα περιβάλλοντα με την παρουσίαση του μηχανισμού των καθολικών σηματοφορέων και δίνουμε μια υλοποίηση για τις πιο διαδεδομένες πλατφόρμες παράλληλου προγραμματισμού PVM και MPI.

- Παρουσιάζουμε ένα εργαλείο αυτόματης παραλληλοποίησης φωλιασμένων βρόχων που κάνει χρήση μεθόδων υπολογιστικής γεωμετρίας. Λειτουργεί τόσο σε συστήματα μοιραζόμενης μνήμης όσο και σε συστήματα κατανεμημένης μνήμης. Χρησιμοποιεί για συγχρονισμό και ανταλλαγή δεδομένων δύο διαφορετικά μοντέλα, εκ των οποίων το πρώτο βασίζεται αποκλειστικά και μόνο σε κλήσεις ανταλλαγής μηνυμάτων ενώ το δεύτερο συνδυάζει το μηχανισμό των καθολικών σηματοφορέων για συγχρονισμό και ασύγχρονων μηνυμάτων για ανταλλαγή δεδομένων. Τέλος, σε πραγματικά παραδείγματα επιτυγχάνει σημαντικές επιταχύνσεις και αποδίδει πολύ καλύτερα από άλλα ανάλογα εργαλεία της βιβλιογραφίας.

ΚΕΦΑΛΑΙΟ 2. ΚΑΘΟΛΙΚΟΙ ΣΗΜΑΤΟΦΟΡΕΙΣ ΣΕ ΠΑΡΑΛΛΗΛΟ ΠΕΡΙΒΑΛΛΟΝ

Ένα από τα σημαντικότερα προβλήματα στον παράλληλο προγραμματισμό είναι ο συντονισμός των διεργασιών που εκτελούνται ταυτόχρονα ώστε να λύνουν ένα δεδομένο πρόβλημα, διατηρώντας πάντα την ορθότητα και την εγκυρότητα των αποτελεσμάτων του αντίστοιχου ακολουθιακού αλγορίθμου.

Στο κεφάλαιο αυτό παρουσιάζεται ένας μηχανισμός καθολικών σηματοφορέων, ο οποίος έχει υλοποιηθεί και μεταφερθεί σε διάφορα παράλληλα προγραμματιστικά περιβάλλοντα. Ο συνολικός μηχανισμός είναι διαθέσιμος μέσω μιας βιβλιοθήκης της γλώσσας C και μερικά επιπρόσθετα αρχεία ορισμών. Ο προγραμματιστής των παραλλήλων εφαρμογών χρησιμοποιεί τις συναρτήσεις της βιβλιοθήκης αυτής με το συνήθη τρόπο και δεν χρειάζεται να ασχολείται με τις λεπτομέρειες και τις ιδιαιτερότητες της υλοποίησης.

2.1 Συγχρονισμός Διεργασιών σε Παράλληλο Περιβάλλον

Ο Dijkstra [2.1] το 1965 για να λύσει το πρόβλημα του κρισίμου τμήματος στον ταυτόχρονο προγραμματισμό αλλά και του συγχρονισμού διεργασιών, εισήγαγε την έννοια των σηματοφορέων. Ένας σηματοφορέας (*semaphore*) ορίζεται ως μια ακέραια μεταβλητή. Ο Dijkstra όρισε δύο νέες βασικές λειτουργίες επί των σηματοφορέων, την P (ή *wait ή down*) και την V (ή *signal ή up*), που απλοποίησαν σημαντικά το πρόβλημα του συγχρονισμού διεργασιών και ορίζονται ως εξής:

P(s) :
```
while s ≤ 0 do skip;
s := s − 1;
```

V(s) :
```
s := s + 1;
```

Οι λειτουργίες P και V θεωρείται ότι εκτελούνται αδιαίρετα. Εάν δύο διεργασίες προσπαθήσουν να εκτελέσουν ταυτόχρονα λειτουργίες P ή V, τότε αυτές θα εκτελεσθούν ακολουθιακά (*σειριακά*) και με τυχαία σειρά.

Στα συμβατικά λειτουργικά συστήματα οι κύριες λειτουργίες που επιτελούν οι σηματοφορείς είναι ο συγχρονισμός διεργασιών και ο αμοιβαίος αποκλεισμός κρισίμων τμημάτων. Επιπλέον, οι σηματοφορείς συνεισφέρουν στη λύση άλλων προβλημάτων, όπως είναι η διάθεση στις διάφορες διεργασίες των πόρων (*resources*) του συστήματος έτσι ώστε αν διασφαλίζεται η ακεραιότητα της λειτουργίας του.

Η χρήση λειτουργιών σηματοφορέων είναι ιδανική λύση για υλοποίηση σε ένα τυπικό λειτουργικό σύστημα ενός επεξεργαστή ή ενός συστήματος στενά συνδεδεμένων επεξεργαστών με καταμερισμού χρόνου ανάμεσα στις διάφορες διεργασίες. Ωστόσο, η υλοποίηση σε ένα παράλληλο ή κατανεμημένο περιβάλλον επεξεργασίας δεν είναι τόσο προφανής.

Με τον όρο *καθολικοί σηματοφορείς* (*global semaphores*) σε ένα κατανεμημένο περιβάλλον παράλληλης επεξεργασίας εννοούμε την υλοποίηση των λειτουργιών των σηματοφορέων με τέτοιο τρόπο ώστε να είναι δυνατός ο συγχρονισμός διεργασιών που εκτελούνται σε διαφορετικούς επεξεργαστές, σε διαφορετικά ίσως υπολογιστικά συστήματα, τα οποία συνδέονται και επικοινωνούν μεταξύ τους μέσω δικτύου. Αναγκαία βέβαια είναι η ύπαρξη μιας πλατφόρμας παράλληλου προγραμματισμού για την ενοποίηση των διαφορετικών επεξεργαστών σε μια ενιαία "εικονική παράλληλη μηχανή". Τέτοιες πλατφόρμες είναι οι PVM[2.2], MPI[2.3], Orchid[2.4].

Η πλατφόρμα PVM (*Parallel Virtual Machine*) έχει αναπτυχθεί στα εργαστήρια Oak Ridge National Labs (ORNL) στις Ηνωμένες Πολιτείες, ενώ το MPI (*Message Passing Interface*) αποτελεί ένα διεθνές πρότυπο για την επικοινωνία με ανταλλαγή μηνυμάτων και υλοποιήσεις του έχουν αναπτυχθεί και είναι διαθέσιμες τόσο από εμπορικές εταιρείες παράλληλων υπολογιστικών συστημάτων όσο και από πανεπιστημιακά και ερευνητικά ιδρύματα. Οι πλατφόρμες PVM και MPI μπορούν να χρησιμοποιηθούν τόσο σε παράλληλα υπολογιστικά συστήματα στενά συνδεδεμένων επεξεργαστών, όπου η επικοινωνία μεταξύ των επεξεργαστών γίνεται είτε μέσω κοινής μνήμης είτε μέσω ειδικού δικτύου συνδέσμων επικοινωνίας, όσο και σε χαλαρά συνδεδεμένα υπολογιστικά συστήματα, όπως για παράδειγμα αριθμός από σταθμούς

εργασίας (ακόμα και διαφορετικής αρχιτεκτονικής) που μπορούν να επικοινωνούν μέσω τοπικού δικτύου. Η πλατφόρμα Orchid έχει αναπτυχθεί στο Εργαστήριο Υπολογιστικών Συστημάτων του Εθνικού Μετσοβίου Πολυτεχνείου, απευθύνεται σε περιβάλλον σταθμών εργασίας UNIX της ίδιας αρχιτεκτονικής, συνδεδεμένων μέσω τοπικού δικτύου Ethernet και αποτελεί μια από τις πρώτες προσπάθειες στο χώρο της παράλληλης επεξεργασίας με τη χρήση δικτύου σταθμών εργασίας.

Η σημασία της σωστής επιλογής ενός μηχανισμού συγχρονισμού διεργασιών ή προστασίας κρισίμων τμημάτων είναι φανερή από τη συνεχιζόμενη έρευνα που γίνεται στην περιοχή και από τις νέες αναφορές που κατά καιρούς παρουσιάζονται. Χαρακτηριστικά αναφέρεται το άρθρο [2.5], που συγκρίνει την αποτελεσματικότητα διαφόρων μηχανισμών κλειδώματος και αμοιβαίου αποκλεισμού. Επιπλέον, τα διάφορα μοντέλα Κατανεμημένης Μοιραζόμενης Μνήμης (*Distributed Shared Memory models*) απαιτούν κάποια γενικά μέσα συγχρονισμού ώστε να διασφαλίζεται η ακεραιότητα και η ορθότητα των περιεχομένων της μνήμης (*memory coherence and consistency*) [2.6]. Οι σηματοφορείς είναι μια δοκιμασμένη λύση που επιλύουν, τις περισσότερες φορές με το βέλτιστο τρόπο, τα προβλήματα αυτά.

Στη διεθνή βιβλιογραφία λίγα είναι τα συστήματα παράλληλου προγραμματισμού που παρουσιάζονται να παρέχουν μηχανισμούς συγχρονισμού με παραπλήσια λειτουργικότητα με τους καθολικούς σηματοφορείς. Για παράδειγμα, το σύστημα Clouds [2.7] που υλοποιεί και αυτό σηματοφορείς, υλοποιεί το μηχανισμό τους ως τμήμα του μοντέλου κατανεμημένης μοιραζόμενης μνήμης που διαθέτει. Στο σύστημα αυτό ο προγραμματιστής είναι υπεύθυνος να φέρει την εγγραφή της μνήμης που αντιστοιχεί σε κάποιο σηματοφορέα, να κάνει την όποια ενημέρωση χρειάζεται και στη συνέχεια να ειδοποιήσει όλες τις άλλες διεργασίες για την αλλαγή του περιεχομένου της μνήμης. Τέτοιες δυνατότητες συγχρονισμού, όπως στο προτεινόμενο σύστημα, απουσιάζουν επίσης από τα περισσότερα κατανεμημένα λειτουργικά συστήματα. Για παράδειγμα, το κατανεμημένο λειτουργικό σύστημα Amoeba [2.8][2.9] παρέχει στον προγραμματιστή του δυνατότητες για συγχρονισμό μόνο μεταξύ νημάτων (*threads*) και όχι μεταξύ διεργασιών (*processes*). Σημειώνεται ότι τα νήματα μιας διεργασίας στο λειτουργικό Amoeba μοιράζονται τις ίδιες εικονικές διευθύνσεις και χρονοδρομολογούνται όλα στον ίδιο επεξεργαστή, περιορίζοντας έτσι τις δυνατότητες για πλήρη εκμετάλλευση της παραλληλίας του συστήματος.

Στην παρούσα διατριβή περιγράφεται η υλοποίηση των καθολικών σηματοφορέων πάνω από πλατφόρμες παράλληλου προγραμματισμού. Ο μηχανισμός των καθολικών σηματοφορέων που υλοποιήθηκε είναι αρκετά γενικός και δίνει στις παράλληλες πλατφόρμες δυνατότητες συγχρονισμού που λείπουν από τα περισσότερα ήδη διαδεδομένα συστήματα. Αρχικά, στην επόμενη ενότητα, δίνεται μια περιγραφή των γενικών αρχών που διέπουν τη σχεδίαση του μηχανισμού αυτού. Στη συνέχεια περιγράφεται το σύνολο των κλήσεων που παρέχονται στον τελικό χρήστη, δηλαδή στον προγραμματιστή των παραλλήλων εφαρμογών. Τέλος, δίνονται οι λεπτομέρειες της υλοποίησης και γίνεται μια αξιολόγηση για τις διάφορες υλοποιήσεις που έχουν γίνει σε διαφορετικές πλατφόρμες παράλληλου προγραμματισμού.

2.2 Γενική Περιγραφή Καθολικών Σηματοφορέων

Η προτεινόμενη υλοποίηση των καθολικών σηματοφορέων απευθύνεται σε περιβάλλοντα παράλληλου προγραμματισμού κατανεμημένης μνήμης, καθώς στην περίπτωση συστημάτων μοιραζόμενης μνήμης η υλοποίηση είναι εύκολη και υπάρχουν ήδη διαθέσιμα εμπορικά συστήματα. Στα συστήματα κατανεμημένης μνήμης ο κύριος μηχανισμός επικοινωνίας και συγχρονισμού είναι το πέρασμα μηνυμάτων. Όπως είναι γνωστό, ο μηχανισμός περάσματος μηνυμάτων είναι ισοδύναμος με εκείνον των σηματοφορέων καθώς και με τη δομή Monitor, αφού και με τους τρεις αυτούς μηχανισμούς μπορούμε να επιτύχουμε συγχρονισμό διεργασιών και αμοιβαίο αποκλεισμό κρισίμων τμημάτων [2.18][2.19]. Συνεπώς ίσως φανεί πως η ύπαρξη μηχανισμού σηματοφορέων σε περιβάλλον περάσματος μηνυμάτων είναι πλεονασμός ή και εντελώς περιττός. Ωστόσο, στα συστήματα που χρησιμοποιούν το μηχανισμό περάσματος μηνυμάτων, για να επιτευχθεί συγχρονισμός μεταξύ δύο διεργασιών είναι απαραίτητο να είναι γνωστά τα ονόματα και των δύο διεργασιών. Το πρόβλημα γίνεται μεγαλύτερο όταν υπάρχει μεγάλος αριθμός διεργασιών που πρέπει να συγχρονισθούν ή όταν κάποιες διεργασίες τερματίζονται ενώ άλλες ξεκινούν την εκτέλεσή τους. Αντίθετα, ο μηχανισμός των σηματοφορέων είναι πιο γενικός από εκείνον των μηνυμάτων, καθώς με τη χρήση του μπορεί να συγχρονισθεί μεγάλος αριθμός διεργασιών χωρίς να είναι απαραίτητο κάθε διεργασία να γνωρίζει πόσες και ποιες άλλες διεργασίες πρέπει να πάρουν μέρος στο συγχρονισμό αυτό. Αρκεί να

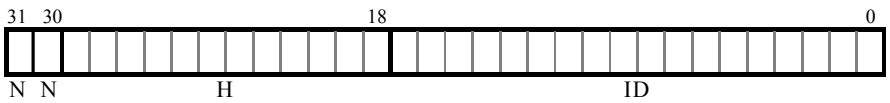
γνωρίζει το όνομα ενός καθολικού σηματοφορέα. Το ίδιο ισχύει και για τον αμοιβαίο αποκλεισμό κρισίμων τμημάτων καθώς και τη διάθεση των πόρων του παραλλήλου συστήματος. Όσον αφορά τη διάθεση των αγαθών του συστήματος σε ένα περιβάλλον κατανεμημένου παράλληλου προγραμματισμού, ένα τέτοιο αγαθό μπορεί να είναι η ανάγκη για τη διαφύλαξη της ακεραιότητας των περιεχομένων μιας Κατανεμημένης Μοιραζόμενης Μνήμης (Distributed Shared Memory), όπως για παράδειγμα στην πλατφόρμα Orchid [2.4].

Σημαντική διαφορά υπάρχει στον τρόπο διαχείρισης των πόρων του συστήματος μεταξύ παράλληλων και κατανεμημένων συστημάτων. Σε ένα περιβάλλον παράλληλου προγραμματισμού τόσο ο χρήστης όσο και ο προγραμματιστής έχουν γενικά γνώση των πόρων του συστήματος που χρησιμοποιούν και πού αυτοί βρίσκονται φυσικώς. Για παράδειγμα, ο προγραμματιστής είναι πάντα ενήμερος για την τοπολογία της παράλληλης μηχανής που χρησιμοποιεί και μπορεί εύκολα να προσδιορίσει τους επεξεργαστές στους οποίους εκτελούνται οι διεργασίες του. Επιπλέον, οι διεργασίες σπανίως μεταναστεύουν και έτσι μπορούν να αναγνωρίζονται από καθορισμένες παγκόσμιες σταθερές. Σε αντίθεση, σε ένα κατανεμημένο περιβάλλον τα αγαθά διαχειρίζονται με ένα πολύ πιο διαφανή τρόπο και έτσι τόσο ο τελικός χρήστης όσο και ο προγραμματιστής δεν έχουν σαφή εικόνα της τοπολογίας του δικτύου και δεν γνωρίζουν το πού εκτελούνται οι διάφορες διεργασίες που έχουν δημιουργήσει. Επιπλέον δυσκολίες εισάγει ο μηχανισμός μετανάστευσης διεργασιών, όπου αυτός είναι υλοποιημένος. Σε αυτά τα περιβάλλοντα κρίνεται σκόπιμη η απομόνωση των χρηστών από τη διαχείριση των πόρων του συστήματος. Ως συνέπεια των παραπάνω σημειώνεται πως η προτεινόμενη σχεδίαση και υλοποίηση απευθύνεται αποκλειστικά σε κατανεμημένα περιβάλλοντα παράλληλου προγραμματισμού, καθώς θεωρούμε ότι κάθε επεξεργαστής στην τοπολογία της παράλληλης μηχανής μπορεί να αναγνωρισθεί με ένα μοναδικό σταθερό αναγνωριστικό και ότι οι καθολικοί σηματοφορείς καθώς και οι εξυπηρετητές τους δεν μεταναστεύουν.

Οι γενικές αρχές που διέπουν τη σχεδίαση του μηχανισμού των καθολικών σηματοφορέων είναι η κατανεμημένη τους διαχείριση και η αποτελεσματική εξυπηρέτηση των αιτήσεων για ενέργειες σε αυτούς, με την ελάχιστη δυνατή επιβάρυνση του δικτύου επικοινωνίας της παράλληλης μηχανής. Η βασική ιδέα της

υλοποίησης είναι η αναγωγή μιας αίτησης για καθολικό σηματοφορέα σε μια αντίστοιχη για κάποιον απλό (συνήθη) σηματοφορέα. Για να επιτευχθεί αυτό, σε κάθε επεξεργαστή της παράλληλης μηχανής εκτελείται μια ειδική διεργασία, που ονομάζεται *εξυπηρετητής καθολικών σηματοφορέων* (*global semaphore server*). Κάθε εξυπηρετητής σηματοφορέων διατηρεί μια λίστα με όλους τους σηματοφορείς που έχουν δημιουργηθεί στον επεξεργαστή που ο ίδιος εκτελείται. Κάθε στοιχείο της λίστας περιλαμβάνει το όνομα του σηματοφορέα, την ταυτότητα/αναγνωριστικό του σηματοφορέα, το μετρητή του, την τρέχουσα τιμή του, ένα δείκτη στην ουρά αναμονής του και τέλος, ένα δείκτη στο επόμενο στοιχείο της λίστας.

Για να αναφερθούμε σε ένα σηματοφορέα μέσα στην παράλληλη μηχανή, η βιβλιοθήκη των καθολικών σηματοφορέων χρησιμοποιεί ένα *περιγραφητή σηματοφορέων* (*semaphore descriptor*), ο οποίος ταυτοποιεί με μοναδικό τρόπο το σηματοφορέα αυτό στο σύνολο της παράλληλης μηχανής. Για λόγους ταχύτητας και απόδοσης, έχει ληφθεί μέριμνα ώστε κάθε περιγραφητής να χωρά στο μεγαλύτερο τύπο δεδομένων που είναι διαθέσιμος στους περισσότερους υπολογιστές, ο οποίος είναι ο τύπος του μεγάλου ακεραίου (long integer) και έχει μέγεθος 32 bits (βλ. Σχήμα 2.1). Στον περιγραφητή κωδικοποιούνται δύο πληροφορίες. Η πρώτη πληροφορία είναι η ταυτότητα του επεξεργαστή στον οποίο εκτελείται ο εξυπηρετητής στον οποίο ανήκει ο σηματοφορέας. Η δεύτερη πληροφορία χρησιμοποιείται εσωτερικά από τον ιδιοκτήτη εξυπηρετητή ως ταυτότητα, για να διακρίνει το συγκεκριμένο καθολικό σηματοφορέα από τους άλλους σηματοφορείς, που ο εξυπηρετητής αυτός έχει ήδη δημιουργήσει. Με την κωδικοποίηση αυτή, γνωρίζοντας μόνο τον περιγραφητή του καθολικού σηματοφορέα είναι δυνατόν να καθοριστεί άμεσα τόσο ο επεξεργαστής στον οποίο έχει ανοίξει ο σηματοφορέας αυτός, συνεπώς και ο αντίστοιχος ιδιοκτήτης εξυπηρετητής, όσο και η σχετική δομή του ιδιοκτήτη εξυπηρετητή που αναφέρεται και περιγράφει το σηματοφορέα αυτό.



Σχήμα 2.1: Περιγραφητής καθολικού σηματοφορέα

Για να γίνει πιο σαφής η λειτουργία του περιγραφητή καθολικών σηματοφορέων θα αναλύσουμε στη συνέχεια την περίπτωση της υλοποίησης στο PVM. Στο PVM η βιβλιοθήκη καθολικών σηματοφορέων χρησιμοποιεί τη δομή που φαίνεται στο Σχήμα 2.1 για να απεικονίσει ένα σηματοφορέα εντός της παράλληλης εικονικής μηχανής. Ο περιγραφητής είναι παρόμοιος, ως προς τη δομή, με τα αναγνωριστικά λειτουργιών του PVM (Task Identifiers - TIDs, [2.2]) και έχει μέγεθος 32-bits. Το πεδίο H είναι το ίδιο με το αντίστοιχο πεδίο στο TID, περιέχει δηλαδή ένα αριθμό επεξεργαστή σχετικό με την εικονική μηχανή. Το πεδίο ID περιέχει έναν ακέραιο αριθμό 18-bit για τον προσδιορισμό των διαφόρων σηματοφορέων που μπορούν να ανοιχθούν σε ένα συγκεκριμένο εξυπηρετητή καθολικών σηματοφορέων. Τέλος, τα bits 30 και 31, που χαρακτηρίζονται με το γράμμα N, δεν χρησιμοποιούνται επί του παρόντος στην τρέχουσα υλοποίηση.

Όλοι οι εξυπηρετητές καθολικών σηματοφορέων στο PVM είναι μέλη μιας παγκόσμιας ομάδας εξυπηρετητών με το όνομα “GSSERVER_GROUP”. Επίσης, κάθε εξυπηρετητής καθολικών σηματοφορέων κατά την εκκίνηση του δημιουργεί μια νέα ομάδα με το όνομα “GSSERVER_XXX”, όπου το XXX αναπαριστά το πεδίο H του TID του. Στην τρέχουσα υλοποίηση στο PVM, δεν υπάρχει κανένας λόγος για να δημιουργήσουμε έναν εξυπηρετητή σηματοφορέων σε κάθε επεξεργαστή της εικονικής μηχανής. Όταν ένας συγκεκριμένος επεξεργαστής δεν είναι ιδιοκτήτης κανενός σηματοφορέα, δεν εκτελείτε στον επεξεργαστή αυτό κάποιος εξυπηρετητής σηματοφορέων.

Η δημιουργία των εξυπηρετητών είναι μια δυναμική διαδικασία. Αρχικά κανένας εξυπηρετητής δεν εκτελείται πουθενά. Με την πρώτη αίτηση για το άνοιγμα ενός καθολικού σηματοφορέα σε καθορισμένο επεξεργαστή, ένας εξυπηρετητής ξεκινά τοπικά στον επεξεργαστή για να ικανοποιήσει το αίτημα και εάν η αίτηση πρέπει να διαβιβαστεί σε κάποιο άλλο επεξεργαστή, ένας νέος εξυπηρετητής ξεκινά επίσης στον επεξεργαστή αυτό. Με αυτή τη ρύθμιση, αν έχουμε μια εικονική μηχανή με ένα μεγάλο αριθμό επεξεργαστών, αλλά θέλουμε να χρησιμοποιήσουμε τις λειτουργίες καθολικών σηματοφορέων σε διεργασίες που εκτελούνται σε λίγους μόνο από τους επεξεργαστές αυτούς, επιτυγχάνεται εξοικονόμηση πόρων και μια πιο αποτελεσματική υλοποίηση, καθώς σε ορισμένες περιπτώσεις γίνεται χρήση συλλογικών λειτουργιών μεταξύ όλων των εξυπηρετητών.

Τα επιμέρους στοιχεία που καθορίζουν τη σχεδίαση και στη συνέχεια τον τρόπο υλοποίησης του μηχανισμού των καθολικών σηματοφορέων είναι με σειρά σπουδαιότητας τα παρακάτω δύο:

i. Ορισμός των διαθέσιμων προς τον τελικό χρήστη "συναρτήσεων βιβλιοθήκης", ορισμός δηλαδή του προγραμματιστικού API (*Application Programming Interface*). Οι συναρτήσεις που παρέχονται, οι οποίες και θεωρήθηκαν απολύτως αναγκαίες, είναι οι: gs_init(), gs_open(), gs_close(), gs_reset(), gs_count(), gs_wait(), gs_signal(), gs_signaln(). Είναι επίσης απαραίτητος ο μονοσήμαντος ορισμός της λειτουργίας των παραπάνω συναρτήσεων σε κάθε περίπτωση και για όλες τις τιμές των παραμέτρων, σωστές και εσφαλμένες.

ii. Σχεδίαση του τρόπου/σχήματος λειτουργίας των παραπάνω συναρτήσεων με τρόπο ώστε να διασφαλίζονται οι αρχές και η σημασιολογία (*semantics*) των γενικών σηματοφορέων.

Το στοιχείο (i) είναι σημαντικότερο του στοιχείου (ii) γιατί η σχεδίαση και η υλοποίηση ενός μηχανισμού εξαρτώνται άμεσα από το πλήθος, το είδος και τον τρόπο λειτουργίας των συναρτήσεων που παρέχονται. Έτσι στη συνέχεια, αρχικά περιγράφονται οι συναρτήσεις που απαρτίζουν τη βιβλιοθήκη των καθολικών σηματοφορέων, καθώς αυτές μπορούν να επηρεάσουν άμεσα το γενικό σχήμα της υλοποίησης. Ακολουθεί μια πιο λεπτομερής περιγραφή του τρόπου λειτουργίας του μηχανισμού των καθολικών σηματοφορέων.

2.3 Περιγραφή Συναρτήσεων Βιβλιοθήκης (API)

Οι συναρτήσεις που απαρτίζουν τη βιβλιοθήκη των καθολικών σηματοφορέων είναι οκτώ και δίνονται στον τελικό χρήστη ως συναρτήσεις βιβλιοθήκης για τη γλώσσα C.

Η διαχείριση ενός σηματοφορέα γίνεται μέσω ενός περιγραφητή (*descriptor*), όπως ήδη αναφέρθηκε. Ο περιγραφητής αυτός είναι ένας ακέραιος με μέγεθος 32 bits και έχει θετική τιμή για όλους τους έγκυρους καθολικούς σηματοφορείς. Στη συνέχεια δίνεται μια σύντομη περιγραφή της λειτουργίας των συναρτήσεων και εξηγείται η σύνταξη και τα σημασιολογικά στοιχεία όλων των διαθέσιμων κλήσεων καθολικών σηματοφορέων.

Πίνακας 2.1: Συναρτήσεις βιβλιοθήκης Καθολικών Σηματοφορέων

α/α	Συνάρτηση Καθολικών Σηματοφορέων	Περιγραφή Λειτουργίας Καθολικών Σηματοφορέων
1	`int gs_init()`	Αρχικοποίηση βιβλιοθήκης καθολικών σηματοφορέων
2	`int gs_open()`	Άνοιγμα καθολικού σηματοφορέα
3	`int gs_close()`	Κλείσιμο καθολικού σηματοφορέα
4	`int gs_reset()`	Επαναφορά και αρχικοποίηση καθολικού σηματοφορέα
5	`int gs_count()`	Επιστροφή τρέχουσας τιμής καθολικού σηματοφορέα
6	`int gs_wait()`	Λειτουργία P (wait) καθολικού σηματοφορέα
7	`int gs_signal()`	Λειτουργία V (signal) καθολικού σηματοφορέα
8	`int gs_signaln()`	n x Λειτουργία V (n x signal) καθολικού σηματοφορέα

- `int gs_init( void );`

  Η συνάρτηση gs_init() δεν δέχεται παραμέτρους και χρησιμοποιείται για την αρχικοποίηση του περιβάλλοντος της βιβλιοθήκης των καθολικών σηματοφορέων στο πρόγραμμα που την καλεί, έτσι ώστε όλες οι μετέπειτα κλήσεις στις άλλες συναρτήσεις της βιβλιοθήκης να δουλέψουν σωστά. Σε ένα πρόγραμμα που θέλει να κάνει χρήση του μηχανισμού των καθολικών σηματοφορέων, η συνάρτηση αυτή πρέπει να είναι η πρώτη συνάρτηση της βιβλιοθήκης των καθολικών σηματοφορέων που καλείται, πριν από οποιαδήποτε άλλη. Αν η συνάρτηση αυτή κληθεί επαναληπτικά στον κώδικα ενός προγράμματος δεν θα προκληθεί ζημιά στην εκτέλεση του προγράμματος ή στη συμπεριφορά της βιβλιοθήκης γενικότερα.

  Πρέπει ωστόσο να σημειωθεί ότι η παρούσα υλοποίηση δεν είναι *πολυνηματικά ασφαλής* (*thread-safe*), δηλαδή σε μια διεργασία που έχει πολλά νήματα, αν περισσότερα του ενός νήματα επιχειρήσουν να κάνουν χρήση των συναρτήσεων της βιβλιοθήκης, τότε η συμπεριφορά θα είναι απρόβλεπτη. Μια πολυνηματικά ασφαλής έκδοση μπορεί να προκύψει με μικρές αλλαγές στον κώδικα των άλλων συναρτήσεων της βιβλιοθήκης και εξαλείφοντας τελείως τη συνάρτηση

αρχικοποίησης. Οι επιπτώσεις στην ταχύτητα εκτέλεσης των άλλων συναρτήσεων της βιβλιοθήκης θα είναι γενικά αμελητέες. Αποφασίστηκε ωστόσο να παραμείνει η συνάρτηση αρχικοποίησης gs_init(), ώστε να υπάρχει η δυνατότητα να αντιμετωπιστούν άμεσα οι όποιες δυσκολίες προκύψουν κατά τη μεταφορά του μηχανισμού των καθολικών σηματοφορέων σε κάποια άλλη πλατφόρμα παράλληλου προγραμματισμού, που ίσως απαιτεί μερικές πρόσθετες αρχικοποιήσεις, χωρίς να υπάρχει διαφοροποίηση του API που δίνεται στον προγραμματιστή από πλατφόρμα σε πλατφόρμα.

- `int gs_open( char *gsemname, char *hostname, int initvalue );`

  Η συνάρτηση gs_open() δημιουργεί ένα νέο καθολικό σηματοφορέα με όνομα 'gsemname', στον επεξεργαστή με όνομα 'hostname' και τον αρχικοποιεί στην τιμή 'initvalue', η οποία πρέπει να είναι μη αρνητική. Η συνάρτηση επιστρέφει έναν θετικό ακέραιο αριθμό, ο οποίος είναι ο περιγραφητής του νέου σηματοφορέα. Ο περιγραφητής αυτός θα πρέπει να χρησιμοποιείται στις μετέπειτα κλήσεις συναρτήσεων της βιβλιοθήκης για αναφορά στο σηματοφορέα αυτό.

  Αν στον επεξεργαστή 'hostname' υπάρχει ήδη σηματοφορέας με το όνομα 'gsemname' τότε δεν δημιουργείται νέος σηματοφορέας αλλά επιστρέφεται ο περιγραφητής του ήδη υπάρχοντος, ενώ η τιμή 'initvalue' αγνοείται. Αν το όνομα του σηματοφορέα είναι η κενή συμβολοσειρά ('gsemname' = "") τότε ανοίγεται οπωσδήποτε νέος σηματοφορέας, εάν βέβαια αυτό είναι δυνατό, αν δηλαδή το επιτρέπουν οι πόροι του συστήματος.

  Αν το όρισμα 'hostname' έχει την τιμή "localhost", τότε ο σηματοφορέας θα ανοίξει τοπικά στον επεξεργαστή που εκτελείται το πρόγραμμα, ενώ αν έχει την τιμή "anyhost", τότε γίνεται προσπάθεια να βρεθεί ένας σηματοφορέας με όνομα 'gsemname' σε οποιονδήποτε επεξεργαστή της παράλληλης μηχανής. Αν κάτι τέτοιο επιτύχει, επιστρέφεται στην καλούσα διεργασία ο περιγραφητής του ήδη υπάρχοντος σηματοφορέα. Αν βρεθούν περισσότεροι του ενός, τότε επιστρέφεται ο περιγραφητής του σηματοφορέα με το μικρότερο κωδικό. Εάν κανένας ανοικτός σηματοφορέας σε κανέναν επεξεργαστή της παράλληλης μηχανής δεν έχει αυτό το όνομα, τότε ένας νέος σηματοφορέας δημιουργείται τοπικά στον ίδιο

επεξεργαστή που τρέχει το καλούν πρόγραμμα και επιστρέφεται ο νέος αυτός περιγραφητής.

Η συνάρτηση gs_open() επιστρέφει κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν δεν μπορεί να δημιουργηθεί νέος σηματοφορέας, είτε επειδή έχουν ήδη δημιουργηθεί πάρα πολλοί σηματοφορείς στον εν λόγω επεξεργαστή, είτε επειδή κάποιο από τα ορίσματα έχει λάθος τιμή, εάν για παράδειγμα, έχει δοθεί ένα όνομα επεξεργαστή που δεν υπάρχει στην τρέχουσα ενεργή τοπολογία της παράλληλης μηχανής ή εάν η τιμή της παραμέτρου 'initvalue' είναι αρνητική.

Όταν ολοκληρωθεί η χρήση ενός σηματοφορέα που δημιουργήθηκε με την κλήση της συνάρτησης gs_open() και πλέον ο σηματοφορέας δεν χρειάζεται, ο σηματοφορέας αυτός πρέπει να "κλείνεται" με μια κατάλληλη κλήση στη συνάρτηση gs_close(), ώστε να αποδεσμεύεται και τελικά να απομακρύνεται και από τη λίστα του ιδιοκτήτη εξυπηρετητή του.

- `int  gs_close( gsem_t gsem );`

  Η συνάρτηση gs_close() κλείνει τον ενεργό σηματοφορέα με περιγραφητή 'gsem' που έχει ήδη δημιουργηθεί με μια προηγούμενη κλήση στη συνάρτηση gs_open(). Αν η gs_close() κληθεί από όλες τις διεργασίες που έχουν "ανοίξει" το σηματοφορέα αυτό, δηλαδή από όλες τις διεργασίες στις οποίες η κλήση της συνάρτησης gs_open() επέστρεψε τον περιγραφητή 'gsem', τότε ο σηματοφορέας αυτός απομακρύνεται από τη λίστα του ιδιοκτήτη εξυπηρετητή του. Σημειώνεται ότι κατά τον τερματισμό μιας διεργασίας δεν κλείνουν αυτόματα οι καθολικοί σηματοφορείς που δημιούργησε ή άνοιξε η εν λόγω διεργασία, καθώς ένας τέτοιος έλεγχος είναι δύσκολος για μια εφαρμογή έξω από το λειτουργικό σύστημα.

  Η συνάρτηση gs_close() επιστρέφει είτε OK (0) για επιτυχημένη ολοκλήρωση είτε κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν το όρισμα 'gsem' δεν είναι νόμιμος περιγραφητής ενός ανοικτού (ενεργού) σηματοφορέα.

- `int  gs_reset( gsem_t gsem, int value );`

  Η συνάρτηση gs_reset() αδειάζει την ουρά αναμονής του σηματοφορέα 'gsem', απελευθερώνοντας όλες τις διεργασίες που βρίσκονται μπλοκαρισμένες

(εμποδισμένες) σε αυτήν και καθιστώντας τις πάλι έτοιμες προς εκτέλεση. Στο σηματοφορέα δίνεται ως νέα του τιμή η τιμή της παραμέτρου 'value', η οποία πρέπει να είναι μη αρνητική.

Η συνάρτηση gs_reset() επιστρέφει είτε OK (0) για επιτυχημένη ολοκλήρωση είτε κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν το όρισμα 'gsem' δεν αντιστοιχεί σε ανοικτό (ενεργό) σηματοφορέα ή αν η παράμετρος 'value' έχει αρνητική τιμή.

- `int  gs_count( gsem_t gsem, int *value );`

Η συνάρτηση gs_count() επιστρέφει στην παράμετρο 'value' την τρέχουσα τιμή του σηματοφορέα 'gsem'. Αρνητική τιμή της παραμέτρου 'value' δηλώνει τον αριθμό των διεργασιών που βρίσκονται μπλοκαρισμένες (εμποδισμένες) και περιμένουν στην ουρά του σηματοφορέα 'gsem', ενώ μη αρνητική τιμή δηλώνει τον αριθμό των κλήσεων gs_wait() που μπορούν να εκτελεστούν πάνω στο σηματοφορέα 'gsem', προτού οι καλούσες διεργασίες αρχίσουν να μπλοκάρονται (εμποδίζονται) στην ουρά του.

Η συνάρτηση gs_count() επιστρέφει είτε OK (0) για επιτυχημένη ολοκλήρωση είτε κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν το όρισμα 'gsem' δεν αντιστοιχεί σε ανοικτό (ενεργό) σηματοφορέα.

- `int  gs_wait( gsem_t gsem );`

Η συνάρτηση gs_wait() ελαττώνει την τιμή του σηματοφορέα 'gsem' κατά ένα. Αν η νέα τιμή του σηματοφορέα είναι αρνητική, τότε η καλούσα διεργασία μπλοκάρεται (εμποδίζεται) και μπαίνει στο τέλος της ουράς αναμονής του σηματοφορέα. Η διεργασία απελευθερώνεται, απομακρύνεται δηλαδή από την ουρά αναμονής του σηματοφορέα και γίνεται έτοιμη να συνεχίσει την εκτέλεσή της, μόνον όταν κάποια άλλη διεργασία εκτελέσει τον κατάλληλο αριθμό κλήσεων στη συνάρτηση gs_signal() ή μια μοναδική κλήση στη συνάρτηση gs_reset() για το σηματοφορέα 'gsem'.

Η συνάρτηση gs_wait() επιστρέφει κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν το όρισμα 'gsem' δεν αντιστοιχεί σε ανοικτό (ενεργό) σηματοφορέα. Σε περίπτωση επιτυχίας, η συνάρτηση επιστρέφει OK (0) είτε άμεσα αν ο

σημαφορέας εξακολουθεί να έχει θετική τιμή είτε μόλις η καλούσα διεργασία ελευθερωθεί από την ουρά αναμονής του σηματοφορέα σε περίπτωση που ο σηματοφορέας απέκτησε μη θετική τιμή.

- `int gs_signal( gsem_t gsem );`
- `int gs_signaln( gsem_t gsem, int n );`

Η συνάρτηση gs_signal() αυξάνει την τιμή του σηματοφορέα 'gsem' κατά ένα. Αν υπάρχουν μπλοκαρισμένες (εμποδισμένες) διεργασίες οι οποίες περιμένουν στην ουρά αναμονής του σηματοφορέα, τότε απελευθερώνεται η πρώτη από αυτές, η οποία και γίνεται πάλι έτοιμη προς εκτέλεση. Η συνάρτηση gs_signaln() ισοδυναμεί με την επανειλημμένη κλήση της συνάρτησης gs_signal() n φορές.

Και οι δύο συναρτήσεις gs_signal() και gs_signaln() επιστρέφουν είτε OK (0) για επιτυχημένη ολοκλήρωση είτε κωδικό λάθους, έναν αρνητικό ακέραιο αριθμό, αν το όρισμα 'gsem' δεν αντιστοιχεί σε ανοικτό (ενεργό) σηματοφορέα.

2.4 Υλοποίηση Μηχανισμού Καθολικών Σηματοφορέων

Ο μηχανισμός των καθολικών σηματοφορέων που προτείνεται στηρίζεται, όπως ήδη αναφέρθηκε, στην ύπαρξη ενός εξυπηρετητή καθολικών σηματοφορέων (global semaphore server) σε κάθε επεξεργαστή της παράλληλης μηχανής. Κάθε διεργασία, που θέλει να πραγματοποιήσει κάποια λειτουργία σχετική με καθολικούς σηματοφορείς, πρέπει να επικοινωνήσει με τον κατάλληλο εξυπηρετητή, μέσω του μηχανισμού περάσματος μηνυμάτων που παρέχει η υφιστάμενη πλατφόρμα παράλληλου προγραμματισμού. Η πραγματική επικοινωνία ωστόσο που λαμβάνει χώρα είναι ενσωματωμένη στις συναρτήσεις βιβλιοθήκης των καθολικών σηματοφορέων και έτσι ο προγραμματιστής δεν έχει να ανησυχεί ή να κάνει κάτι περισσότερο σχετικά με αυτό.

Ο μηχανισμός των καθολικών σηματοφορέων θα μπορούσε να έχει υλοποιηθεί με τη χρήση ενός μοναδικού εξυπηρετητή για το σύνολο των επεξεργαστών που απαρτίζουν την παράλληλη μηχανή. Στην περίπτωση αυτή, οι αιτήσεις για τη διαχείριση και για ενέργειες πάνω σε καθολικούς σηματοφορείς από όλες τις διεργασίες που εκτελούνται

σε οποιοδήποτε επεξεργαστή της παράλληλης μηχανής, θα κατευθύνονταν σε αυτόν τον εξυπηρετητή, ο οποίος και "κεντρικά" θα τις διαχειριζόταν όλες. Το σχήμα αυτό έχει το πλεονέκτημα της εύκολης και γρήγορης υλοποίησης, παρουσιάζει όμως σημαντικά μειονεκτήματα, όπως εξάλλου και τα περισσότερα μοντέλα κεντρικής διαχείρισης σε κατανεμημένα περιβάλλοντα. Για παράδειγμα, η ύπαρξη ενός μόνο εξυπηρετητή σε έναν συγκεκριμένο επεξεργαστή επιφέρει μεγάλο φόρτο στο δίκτυο επικοινωνίας, αφού η πλειοψηφία των αιτήσεων πρέπει γενικά να εξυπηρετούνται σε "απομακρυσμένο" επεξεργαστή. Η μεγαλύτερη επιβάρυνση θα παρατηρείται όταν στον επεξεργαστή που εκτελείται ο εξυπηρετητής δεν τρέχει καμία διεργασία που να θέλει να κάνει χρήση καθολικών σηματοφορέων, αλλά σε κάποιους άλλους επεξεργαστές τρέχουν αρκετές διεργασίες που θέλουν να εκτελέσουν τέτοιες ενέργειες και τις οποίες θα πρέπει συνεπώς να προωθήσουν στον απομακρυσμένο εξυπηρετητή. Άλλο σημαντικό μειονέκτημα της παραπάνω τεχνικής είναι η μικρή ανθεκτικότητα σε σφάλματα. Τέτοια σφάλματα είναι, για παράδειγμα, η μείωση της ρυθμαπόδοσης (*throughput*) του δικτύου, η παροδική ή και ολοκληρωτική κατάρρευση του δικτύου επικοινωνίας ή η κατάρρευση του κόμβου (επεξεργαστή ή υπολογιστή) στον οποίο εκτελείται ο εξυπηρετητής.

Για να ξεπεραστούν τα παραπάνω προβλήματα υιοθετήθηκε η στρατηγική της χρήσης ενός εξυπηρετητή καθολικών σηματοφορέων ανά επεξεργαστή της παράλληλης μηχανής. Η τεχνική αυτή επιβαρύνει κατά το ελάχιστο το φόρτο του δικτύου επικοινωνίας, καθώς είναι δυνατό μια διεργασία να κάνει χρήση σηματοφορέων που έχουν ανοίξει τοπικά, στον ίδιο επεξεργαστή που και αυτή εκτελείται κι έτσι να μην είναι απαραίτητη η επικοινωνία με κανέναν άλλο "απομακρυσμένο" εξυπηρετητή για τη σωστή διεκπεραίωση της αίτησης. Στην περίπτωση αυτή δεν δημιουργείται καθόλου επιπρόσθετη κίνηση στο δίκτυο επικοινωνίας που συνδέει τους επεξεργαστές της παράλληλης μηχανής.

Πιο συγκεκριμένα, το σύνολο των συναρτήσεων της βιβλιοθήκης που παρέχονται προς τον τελικό χρήστη, δηλαδή προς τον προγραμματιστή παραλλήλων εφαρμογών, χωρίζεται σε δύο κατηγορίες:

- Στην πρώτη κατηγορία ανήκουν οι συναρτήσεις gs_init() και gs_open().
- Στην δεύτερη κατηγορία ανήκουν όλες οι άλλες συναρτήσεις της βιβλιοθήκης.

Όλες οι συναρτήσεις της δεύτερης κατηγορίας παίρνουν τουλάχιστον μια παράμετρο, τον περιγραφητή του σηματοφορέα για τον οποίο ζητείται να γίνει η αντίστοιχη ενέργεια. Ο περιγραφητής αυτός είναι ένας θετικός ακέραιος αριθμός στον οποίο κωδικοποιούνται δύο σημαντικές πληροφορίες, ο κωδικός του εξυπηρετητή καθολικών σηματοφορέων στον οποίο έχει ανοίξει και ανήκει ο εν λόγω σηματοφορέας και ο κωδικός (*αριθμός ταυτότητας*) του σηματοφορέα στον εξυπηρετητή αυτό. Όταν μια διεργασία καλεί μια συνάρτηση αυτής της κατηγορίας για κάποιον συγκεκριμένο σηματοφορέα, το σύστημα αυτομάτως στέλνει ένα κατάλληλο μήνυμα στον ιδιοκτήτη εξυπηρετητή του σηματοφορέα. Το μήνυμα περιέχει οπωσδήποτε τον αριθμό ταυτότητας της καλούσας διεργασίας και τον περιγραφητή του σηματοφορέα. Η καλούσα διεργασία μπλοκάρεται και περιμένει μήνυμα με το αποτέλεσμα της αίτησης. Μόλις ένας εξυπηρετητής λάβει μια τέτοια αίτηση, την ελέγχει για την εγκυρότητα του περιγραφητή, αν δηλαδή έχει κάποιον αντίστοιχο ενεργό σηματοφορέα με αυτόν τον περιγραφητή και αν πράγματι έχει τότε συνεχίζει με τη σωστή διεκπεραίωση της αίτησης. Σε αντίθετη περίπτωση επιστρέφεται στην καλούσα διεργασία ένας κωδικός λάθους.

Πιο πολύπλοκη από άποψη λειτουργιών είναι η συνάρτηση gs_open(), που ανήκει στην πρώτη κατηγορία. Η λειτουργία της συνάρτησης αυτής διαφοροποιείται σημαντικά ανάλογα με τις τιμές των ορισμάτων της. Η καλούσα διεργασία αρχικά στέλνει ένα κατάλληλο μήνυμα στον τοπικό της εξυπηρετητή, στον εξυπηρετητή δηλαδή που εκτελείται στον ίδιο επεξεργαστή με αυτήν, και στη συνέχεια ο εξυπηρετητής είναι αυτός που αναλαμβάνει τη σωστή διεκπεραίωση της κλήσης. Στην περίπτωση που το όρισμα 'hostname' έχει την τιμή "localhost" ή το πραγματικό όνομα του επεξεργαστή στον οποίο εκτελείται η καλούσα διεργασία, ο εξυπηρετητής καθολικών σηματοφορέων ανοίγει ένα νέο σηματοφορέα τοπικά και γίνεται ο ιδιοκτήτης του. Για την ειδική περίπτωση που το όρισμα 'hostname' έχει την τιμή "anyhost", τότε ο εξυπηρετητής που έχει δεχθεί την αίτηση αυτή ξεκινά μια διαδικασία ψηφοφορίας (*election*) ανάμεσα σε όλους τους εξυπηρετητές, ώστε να βρεθεί αν υπάρχει ήδη κάποιος κατάλληλος ενεργός σηματοφορέας. Αν βρεθεί ένας τέτοιος περιγραφητής σηματοφορέα, η αντίστοιχη αίτηση προωθείται στον αντίστοιχο εξυπηρετητή, διαφορετικά η αίτηση εξυπηρετείται τοπικά, με τη δημιουργία ενός νέου σηματοφορέα στον τοπικό εξυπηρετητή. Σε κάθε περίπτωση ο εξυπηρετητής ο οποίος είναι ο

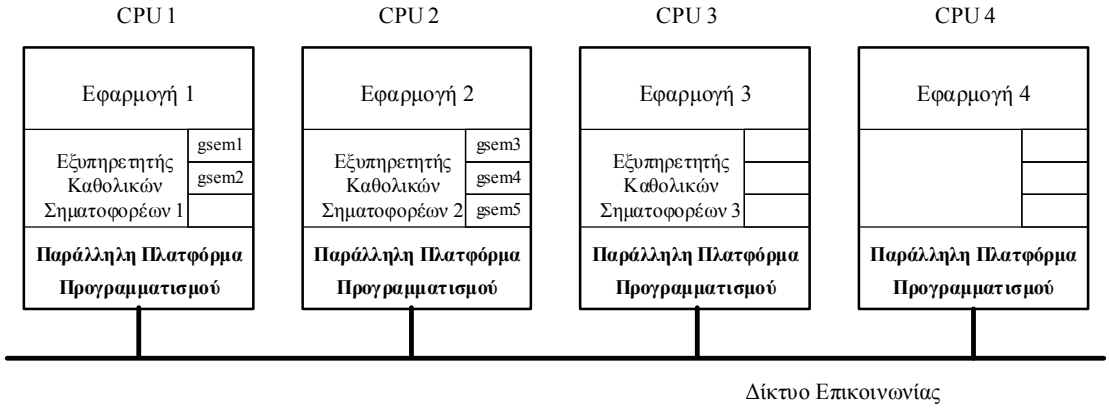
ιδιοκτήτης του νέου σηματοφορέα είναι αυτός που απαντά στην καλούσα διεργασία, στέλνοντάς της ένα μήνυμα με τον περιγραφητή του σηματοφορέα.

Τέλος, η συνάρτηση gs_init(), η οποία ανήκει και αυτή στην πρώτη κατηγορία, είναι η πιο απλή από όλες τις συναρτήσεις της βιβλιοθήκης και σκοπός της είναι η σωστή αρχικοποίηση του περιβάλλοντος της βιβλιοθήκης των καθολικών σηματοφορέων, έτσι ώστε όλες οι μετέπειτα κλήσεις να λειτουργήσουν σωστά. Με την κλήση της συνάρτησης αυτής δεν γίνεται καμία πραγματική επικοινωνία και δεν δημιουργείται κανένας εξυπηρετητής καθολικών σηματοφορέων.

Σημειώνεται ότι όλες οι παραπάνω λειτουργίες εκτελούνται από το μηχανισμό των καθολικών σηματοφορέων διαφανώς από το χρήστη. Οι απαραίτητες ανταλλαγές μηνυμάτων είναι ενσωματωμένες στις συναρτήσεις της βιβλιοθήκης και ο προγραμματιστής δεν χρειάζεται να κάνει κάποιες άλλες ενέργειες. Επιπλέον, η διαδικασία της δημιουργίας εξυπηρετητών καθολικών σηματοφορέων είναι δυναμική. Αρχικά και μετά από την αρχικοποίηση του περιβάλλοντος των καθολικών σηματοφορέων κανένας εξυπηρετητής δεν εκτελείται αυτόματα. Με την πρώτη αίτηση για να ανοιχθεί ένας καθολικός σηματοφορέας μέσω της συνάρτησης βιβλιοθήκης gs_open(), αυτόματα εκτελείται ένας εξυπηρετητής στον ίδιο επεξεργαστή. Εάν η αίτηση πρέπει να προωθηθεί σε κάποιον άλλο επεξεργαστή, τότε ο εξυπηρετητής αυτός ενεργοποιεί έναν καινούργιο εξυπηρετητή στον επεξεργαστή αυτό και του στέλνει την αίτηση. Η αίτηση στη συνέχεια θα διεκπεραιωθεί με το συνήθη τρόπο.

Με το δυναμικό αυτό μοντέλο διαχείρισης των εξυπηρετητών υπάρχει το όφελος ότι εκτελείται ο ελάχιστος αναγκαίος αριθμός εξυπηρετητών που πραγματικά χρειάζονται τα προγράμματά μας, χωρίς να εκτελούνται άεργοι εξυπηρετητές και χωρίς να δεσμεύονται άσκοπα πόροι (*resources*) του συστήματος. Αυτό είναι δυνατό καθώς δεν υπάρχει πραγματική ανάγκη να εκτελείται ένας εξυπηρετητής καθολικών σηματοφορέων σε έναν επεξεργαστή αν η παρουσία του δεν είναι αναγκαία, δηλαδή αν καμιά από τις διεργασίες που τρέχουν στον επεξεργαστή αυτόν δεν θέλει να εκτελέσει κάποια λειτουργία καθολικών σηματοφορέων. Τούτο μπορεί να φανεί ιδιαίτερα χρήσιμο σε περιπτώσεις όπου η τοπολογία του δικτύου περιλαμβάνει μεγάλο αριθμό επεξεργαστών, αλλά διεργασίες που τρέχουν σε ένα μικρό μόνο αριθμό από αυτούς θέλουν να χρησιμοποιήσουν τις διευκολύνσεις του μηχανισμού των καθολικών σηματοφορέων. Με άλλα λόγια, η προσέγγιση αυτή είναι πολύ προσεκτική με την

ανάλωση των πόρων του συστήματος. Επιπλέον ο μηχανισμός αυτός παρέχει μια βασική μέθοδο για ανάνηψη (*recovery*) μετά από σφάλματα όπως την αστοχία και την επανεκκίνηση ενός επεξεργαστή της παράλληλης μηχανής.



Σχήμα 2.2: Παράδειγμα χρήσης του μηχανισμού καθολικών σηματοφορέων

Για την καλύτερη κατανόηση του τρόπου λειτουργίας του μηχανισμού των καθολικών σηματοφορέων δίνεται το ακόλουθο παράδειγμα (Σχήμα 2.2). Έστω ότι η παράλληλη μηχανή αποτελείται από τέσσερεις επεξεργαστές, οι οποίοι είναι γνωστοί με τα αναγνωριστικά CPU1, CPU2, CPU3 και CPU4, διασυνδεδεμένων μεταξύ τους μέσω δικτύου, η τοπολογία και η αρχιτεκτονική του οποίου δεν έχει ιδιαίτερη σημασία, ώστε να μπορούν να επικοινωνούν μεταξύ τους. Σε κάθε επεξεργαστή εκτελείται κώδικας της παράλληλης πλατφόρμας προγραμματισμού, ο οποίος είναι υπεύθυνος για τη διαχείριση των διεργασιών και το πέρασμα μηνυμάτων μεταξύ των διεργασιών. Στους επεξεργαστές CPU 1, CPU 2 και CPU 3 τρέχουν τρεις εφαρμογές οι οποίες θέλουν να κάνουν χρήση των συναρτήσεων των καθολικών σηματοφορέων, ενώ στον επεξεργαστή CPU 4 η εφαρμογή 4 δεν θέλει να χρησιμοποιήσει καμιά τέτοια συνάρτηση. Αυτό έχει σαν συνέπεια να εκτελείται από ένας εξυπηρετητής καθολικών σηματοφορέων μόνο στους τρεις πρώτους επεξεργαστές, ενώ στον τέταρτο η ύπαρξη εξυπηρετητή δεν είναι απαραίτητη. Στον εξυπηρετητή 1 έχουν ανοίξει δύο καθολικοί σηματοφορείς, οι gsem1 και gsem2, στον εξυπηρετητή 2 έχουν ανοίξει τρεις καθολικοί σηματοφορείς, οι gsem3, gsem4 και gsem5, ενώ στον εξυπηρετητή 3 δεν υπάρχει κανένας ανοικτός σηματοφορέας.

Όταν η εφαρμογή 3 ζητήσει να ανοίξει το σηματοφορέα gsem4 που βρίσκεται στον εξυπηρετητή 2, τότε οι συναρτήσεις βιβλιοθήκης θα στείλουν το κατάλληλο μήνυμα

στον εξυπηρετητή 3. Ο εξυπηρετητής 3 θα στείλει ένα μήνυμα σε όλους τους άλλους εξυπηρετητές, ώστε να εντοπιστεί αν ο ζητούμενος σηματοφορέας έχει ήδη δημιουργηθεί. Στην περίπτωση που εξετάζεται, ο σηματοφορέας υπάρχει ήδη και ανήκει στον εξυπηρετητή 2, ο οποίος είναι τελικά αυτός που θα στείλει το κατάλληλο μήνυμα με τον περιγραφητή του σηματοφορέα πίσω στην εφαρμογή 3. Όλες οι επόμενες λειτουργίες της εφαρμογής 3 στο σηματοφορέα gsem4, όπως για παράδειγμα gs_wait() και gs_signal(), θα σταλούν από τις αντίστοιχες συναρτήσεις βιβλιοθήκης κατευθείαν στον εξυπηρετητή 2 μέσω του δικτύου επικοινωνίας. Αν αντίθετα η εφαρμογή 2 θελήσει να εκτελέσει κάποια λειτουργία στο σηματοφορέα gsem4 τότε το αντίστοιχο μήνυμα θα σταλεί στον εξυπηρετητή 2 χωρίς να περάσει μέσα από το δίκτυο επικοινωνίας που συνδέει τους επεξεργαστές. Είναι φανερό πως η διαφορά απόδοσης ανάμεσα στις δύο αυτές περιπτώσεις, δηλαδή μεταξύ της εξυπηρέτησης μιας αίτησης για σηματοφορέα ορισμένο τοπικά ή απομακρυσμένα, εξαρτάται από τον τρόπο που τις χειρίζεται η παράλληλη πλατφόρμα προγραμματισμού που χρησιμοποιείται, από το αν δηλαδή η πλατφόρμα διαφοροποιεί σημαντικά τις περιπτώσεις τοπικής και απομακρυσμένης επικοινωνίας.

2.5 Αξιολόγηση Υλοποιήσεων

Ο μηχανισμός των καθολικών σηματοφορέων που παρουσιάστηκε στις προηγούμενες ενότητες, έχει υλοποιηθεί σε τρεις διαφορετικές πλατφόρμες παράλληλου προγραμματισμού. Αυτές είναι οι PVM[2.11], MPI[2.12] και Orchid[2.4]. Ο κώδικας έχει μεταφερθεί στις πλατφόρμες αυτές χωρίς ιδιαίτερες δυσκολίες. Στον προγραμματιστή παραλλήλων εφαρμογών σε κάθε περίπτωση δίνεται το ίδιο σύνολο συναρτήσεων που έχουν ακριβώς την ίδια σύνταξη. Μόνο ένα μικρό μέρος της υλοποίησης αλλάζει κάθε φορά ανάλογα και με τις δυνατότητες που παρέχει η χρησιμοποιούμενη παράλληλη πλατφόρμα. Οι διαφορές αυτές δεν επηρεάζουν τη λειτουργικότητα των συναρτήσεων, ούτε γίνονται αντιληπτές από τον προγραμματιστή των παραλλήλων εφαρμογών.

Στον Πίνακας 2.2 συνοψίζονται οι διαφορές που εμφανίζονται στις υλοποιήσεις του μηχανισμού των καθολικών σηματοφορέων στις τρεις παράλληλες πλατφόρμες που έχει μεταφερθεί. Οι αστερίσκοι δείχνουν πόσο καλά λύνεται το αντίστοιχο πρόβλημα σε μια πλατφόρμα, με περισσότερους αστερίσκους να αντιπροσωπεύουν καλύτερη

λύση. Έτσι, όσον αφορά το πρόβλημα της επιβάρυνσης της επικοινωνίας (*communication overhead*) η υλοποίηση στην πλατφόρμα PVM είναι η πιο αναποτελεσματική καθώς γίνεται χρήση του μηχανισμού ομαδικής επικοινωνίας (*group communication*), που στο PVM υλοποιείται μέσω μιας μοναδικής διεργασίας για το σύνολο της εικονικής παράλληλης μηχανής. Τέτοια προβλήματα δεν παρουσιάζονται στις δύο άλλες πλατφόρμες καθώς δεν χρησιμοποιούνται σε αυτές τέτοιοι μηχανισμοί επικοινωνίας. Η αντιμετώπιση αυτού του προβλήματος έχει αντιστρόφως ανάλογη επίπτωση στην ευκολία υλοποίησης, με την πιο εύκολη υλοποίηση του μηχανισμού των καθολικών σηματοφορέων να έχει γίνει στην πλατφόρμα PVM.

Πίνακας 2.2: Αξιολόγηση υλοποιήσεων καθολικών σηματοφορέων

Παράλληλες Πλατφόρμες	**PVM**	**MPI**	**Orchid**
Επιβάρυνση Επικοινωνίας	**	*****	****
Ευκολία Υλοποίησης	*****	***	**
Αξιοπιστία	*****	*****	*

Τέλος, όσον αφορά την αξιοπιστία του μηχανισμού, οι πιο αξιόπιστες υλοποιήσεις είναι αυτές για τις πλατφόρμες PVM και MPI, ενώ η υλοποίηση στην πλατφόρμα Orchid υπολείπεται σημαντικά. Αυτό συμβαίνει γιατί η πλατφόρμα Orchid δεν διαθέτει συνάρτηση για τη λήψη μηνύματος από συγκεκριμένη διεργασία (συνάρτηση *receivefrom*) αλλά και επειδή σε αυτήν δεν είναι δυνατόν να διαφοροποιούμε τα μηνύματα που ανταλλάσουν μεταξύ τους οι διεργασίες, σύμφωνα με την τιμή κάποιων σταθερών αναγνωριστικών (*message tags*). Δηλαδή, στην πλατφόρμα Orchid, αντίθετα με τις πλατφόρμες PVM και MPI, ένα μήνυμα δεν μπορεί να διαφοροποιείται και να χαρακτηρίζεται με βάση ένα αναγνωριστικό και αντίστοιχα μια διεργασία δε μπορεί να ζητήσει τη λήψη κάποιου μηνύματος από συγκεκριμένη διεργασία που να προσδιορίζεται και από ένα συγκεκριμένο αναγνωριστικό. Αυτό έχει ως άμεση συνέπεια την ύπαρξη πιθανότητας το σύστημα να περιέλθει σε κατάσταση αδιεξόδου (*deadlock*), καθώς είναι δυνατόν μια διεργασία να περιμένει μήνυμα από κάποια

συγκεκριμένη διεργασία αλλά να λάβει κάποιο άλλο μήνυμα από διαφορετική διεργασία.

Για να δικαιολογηθεί η αξιολόγηση και η κατάταξη που πρόεκυψε για τις τρεις παραλλήλες πλατφόρμες σε σχέση με την επιβάρυνση επικοινωνίας, παρατίθεται ο Πίνακας 2.3. Στον πίνακα αυτό γίνεται αντιπαράθεση των χρόνων που απαιτούνται για την πλήρη ανταλλαγή ενός μηνύματος μεταξύ δύο διεργασιών, δηλαδή του χρόνου που μεσολαβεί από την αποστολή (*send*) μέχρι τη λήψη (*receive*) ενός μηνύματος από μια διεργασία μέσω μιας δεύτερης διεργασίας και των χρόνων που απαιτούνται για την ολοκλήρωση μιας λειτουργίας σε καθολικούς σηματοφορείς. Μια λειτουργία καθολικών σηματοφορέων γενικά απαιτεί την αποστολή ενός μηνύματος ώστε να προωθηθεί η ζητούμενη αίτηση στον κατάλληλο εξυπηρετητή καθολικών σηματοφορέων και στη συνέχεια τη λήψη από την καλούσα διεργασία ενός άλλου μηνύματος, το οποίο αποστέλλεται από τον εξυπηρετητή που τελικά διεκπεραίωσε την εν λόγω αίτηση και στο οποίο έχει κωδικοποιηθεί το αποτέλεσμα της αίτησης. Η συνάρτηση που χρησιμοποιήθηκε για να αξιολογηθεί η επίδοση του μηχανισμού των καθολικών σηματοφορέων είναι η συνάρτηση gs_signal(), καθώς είναι μια συνάρτηση η εκτέλεση της οποίας δεν εμποδίζεται ποτέ (*non-blocking call*).

Πίνακας 2.3: Μετρήσεις ελαχίστων χρόνων εκτέλεσης

Παράλληλη Πλατφόρμα	**Τοπικό σε Τοπικό**		**Τοπικό σε Απομακρυσμένο**	
	μsec/Send-Receive	**μsec/gs-signal()**	**μsec/Send-Receive**	**μsec/gs-signal()**
PVM	3961	7689	7126	10969
MPI	3030	3187	3364	3460
Orchid	1677	1798	2851	2994

Γενικά διακρίνουμε δύο περιπτώσεις, είτε ο αποστολέας και ο παραλήπτης του μηνύματος εκτελούνται στον ίδιο επεξεργαστή είτε εκτελούνται σε διαφορετικούς επεξεργαστές, οι οποίοι συνδέονται μέσω κάποιου συνδέσμου επικοινωνίας, συνήθως τοπικό δίκτυο. Οι μετρήσεις που αναφέρονται στις "Τοπικό σε Τοπικό" ανταλλαγές μηνυμάτων έγιναν από μετροπρογράμματα (*benchmarks*) που εκτελέστηκαν σε σταθμό εργασίας Sun SparcStation 5, συχνότητας 110 MHz. Για τις μετρήσεις που αναφέρονται στις "Τοπικό σε Απομακρυσμένο" ανταλλαγές μηνυμάτων χρησιμοποιήθηκε

επιπρόσθετα ένας σταθμός εργασίας Sun SparcStation 5, συχνότητας 70 MHz. Το δίκτυο επικοινωνίας ήταν ένα τυπικό δίκτυο Ethernet ταχύτητας 10Mbit (10Base-T). Πρέπει να επισημανθεί ότι οι απόλυτες τιμές που παρουσιάζονται στον παραπάνω πίνακα προέκυψαν ως οι ελάχιστες τιμές μιας σειράς μετρήσεων για κάθε μια ανεξάρτητη περίπτωση. Το εύρος στο οποίο οι τιμές αυτές κυμάνθηκαν ήταν αρκετά μεγάλο καθώς οι μετρήσεις επηρεάζονταν κάθε φορά αισθητά, τόσο από το συνολικό υπολογιστικό φορτίο του κάθε επεξεργαστή όσο και από τη χρησιμοποίηση και τη ρυθμαπόδοση του δικτύου τη χρονική στιγμή που εκτελούνταν τα μετροπρογράμματα.

Από τα αποτελέσματα που παρουσιάζονται στον πίνακα, είναι φανερό ότι η πλατφόρμα PVM έχει τη λιγότερο αποτελεσματική υλοποίηση καθώς επίσης και τη μεγαλύτερη αύξηση στην επιβάρυνση της επικοινωνίας, όταν αποστολέας και παραλήπτης εκτελούνται σε διαφορετικούς επεξεργαστές. Οι επιδόσεις για τις πλατφόρμες MPI και Orchid είναι σημαντικά καλύτερες. Πρέπει ωστόσο να επισημανθεί ότι η μικρή επιβάρυνση επικοινωνίας που παρουσιάζει η πλατφόρμα Orchid οφείλεται μερικώς στο ότι η πλατφόρμα Orchid δεν μετατρέπει τα δεδομένα που έχει να μεταφέρει μέσω του δικτύου στον τρόπο αναπαράστασης του δικτύου (*network byte order*), αλλά θεωρεί ότι η επικοινωνία γίνεται μεταξύ επεξεργαστών και υπολογιστικών συστημάτων γενικότερα που χρησιμοποιούν την ίδια εσωτερική αναπαράσταση δεδομένων.

2.6 Συμπεράσματα

Ο μηχανισμός των καθολικών σηματοφορέων που προτείνεται παραπάνω είναι απλός στη σχεδίαση, εύκολος και αποτελεσματικός στην υλοποίηση. Η διαχείριση των καθολικών σηματοφορέων και των αιτήσεων για ενέργειες σε αυτούς γίνεται με κατανεμημένο τρόπο, με τη χρήση ενός εξυπηρετητή καθολικών σηματοφορέων σε κάθε επεξεργαστή που μετέχει στην τοπολογία της παράλληλης μηχανής. Κάθε καθολικός σηματοφορέας ανήκει σε έναν μόνο εξυπηρετητή καθολικών σηματοφορέων και ο εξυπηρετητής αυτός είναι ο μοναδικός υπεύθυνος για τη διαχείριση του σηματοφορέα και την εξυπηρέτηση όλων των αιτήσεων για το σηματοφορέα αυτό. Ένας εξυπηρετητής γενικά δεν γνωρίζει τον αριθμό ή την κατάσταση των καθολικών σηματοφορέων που υπάρχουν στους άλλους

εξυπηρετητές. Το κύριο πλεονέκτημα της μεθόδου είναι η μικρή επιβάρυνση του δικτύου επικοινωνίας της παράλληλης μηχανής, η άμεση εξυπηρέτηση των πιο κοινών ενεργειών σε καθολικούς σηματοφορείς από τους κατάλληλους ιδιοκτήτες-εξυπηρετητές με την ανταλλαγή κατά μέσο όρο δύο μηνυμάτων και η δυνατότητα να υλοποιηθεί ο μηχανισμός αυτός ακόμα και σε συστήματα όπου το λειτουργικό τους σύστημα δεν παρέχει απλούς (συνήθεις) σηματοφορείς. Η χρήση των καθολικών σηματοφορέων, με το σύνολο των συναρτήσεων βιβλιοθήκης που δίνονται στον τελικό χρήστη και προγραμματιστή παραλλήλων εφαρμογών, είναι αρκετά απλή και εύκολη καθώς και σύμφωνη με τα όσα ήδη ισχύουν για τη χρήση των κοινών σηματοφορέων σε υφιστάμενα πολυδιεργαστικά λειτουργικά συστήματα ευρείας αποδοχής.

2.7 Αναφορές

[2.1] E. W. Dijkstra, "Co-operating Sequential Processes", in Programming Languages, *Genuys, F. (Ed.), London, Academic Press, 1965*.

[2.2] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, "PVM: Parallel Virtual Machine", *The MIT Press, 1994*.

[2.3] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The Complete Reference", *The MIT Press, 1996*.

[2.4] C. Voliotis, G. Manis, Ch. Lekatsas, P. Tsanakas and G. Papakonstantinou, "ORCHID: A Portable Platform for Parallel Programming", *Journal of Systems Architecture, April 1997, Volume 43, Issue 6-7, pp. 459-478*.

[2.5] P. E. McKenney, "Selecting Locking Primitives for Parallel Programming", *Communications of the ACM, October 1996, Vol.39, No.10*.

[2.6] Sarita V. Adve, Kourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial", *IEEE Computer, December 1996, Vol.29, No.12*.

[2.7] P. Dasgupta, R. J. LeBlanc, Jr. M. Ahamad and U. Ramachandran, "The Clouds Distributed Operating System", *IEEE Computer, November 1991, Vol.24, No.11, pp. 34-44*.

[2.8] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse and H. van. Staveren, "Amoeba: a Distributed Operating System for the 1990s", *IEEE Computer, Vol.23, pp.44-53, May 1990*.

[2.9] A. S. Tanenbaum, R. van Renesse, H. van. Staveren, G. J. Sharp, S. J. Mullender, A. J. Jansen and G. van Rossum, "Experiences with the Amoeba Distributed Operating System", *Communication ACM, Vol.33, pp.46-63, December 1990*.

[2.10] R. van Renesse, H. van Staveren and A. S. Tanenbaum, "Performance of the Amoeba Distributed Operating System", *Software-Practice and Experience, Vol.19, pp.223-234, March 1989*.

[2.11] M. F. Kaashoek, A. S. Tanenbaum and K. Verstoep, "Group Communication in Amoeba and its Applications", *Distributed Systems Engineering Journal, Vol.1, pp.48-58, July 1993*.

[2.12] F. Douglis, J. K. Ousterhout, M. F. Kaashoek and A. S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite", *Computing Systems, Vol.4, No.3, pp.353-384, December 1991*.

[2.13] A. Forin, J. Barrera, M. Young and R. Rashid, "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach", *Proceedings Winter USENIX Conference, Jan. 1989*.

[2.14] M. J. Bach, "The Design of the UNIX Operating System", *Englewood Cliffs, New Jersey, Prentice-Hall, 1987*.

[2.15] S. R. Bourne, "The UNIX System", *Reading, MA, Addison-Wesley, 1982*.

[2.16] B. W. Kernighan and R. Pike, "The UNIX Programming Environment", *Englewood Cliffs, New Jersey, Prentice-Hall, 1984*.

[2.17] B. W. Kernigham and D. M. Ritchie, "The C Programming Language", 2nd Edition, *Englewood Cliffs, New Jersey, Prentice-Hall, 1988*.

[2.18] Abraham Silberschatz, James L. Peterson, Peter B. Galvin, "Operating System Concepts", *Third Edition, Addison-Wesley, 1991*.

[2.19] Andrew S. Tanenbaum, "Operating Systems: Design and Implementation", *Englewood Cliffs, New Jersey, Prentice-Hall, 1987*.

[2.20] Andrew S. Tanenbaum, "Modern Operating Systems", *Englewood Cliffs, New Jersey, Prentice-Hall, 1992*.

[2.21] P. Theodoropoulos, G. Manis, P. Tsanakas and G. Papakonstantinou, "Extending Synchronization PVM Mechanisms", *Proceedings of the Third European PVM Conference, Springer, October 1996*.

[2.22] J.J. Dongarra, S.W.Otto, M.Snir and D.Walker, "A Message Passing Standard for MPP and Workstations", *Communications of the ACM, July 1996, Vol.39, No.7*.

ΚΕΦΑΛΑΙΟ 3. ΕΡΓΑΛΕΙΟ CRONUS

Στο κεφάλαιο αυτό περιγράφεται το CRONUS, ένα αυτοματοποιημένο εργαλείο για την παραλληλοποίηση γενικευμένων φωλιασμένων βρόχων που βασίζεται σε μεθόδους υπολογιστικής γεωμετρίας. Οι γενικευμένοι φωλιασμένοι βρόχοι περιλαμβάνουν σύνθετα σώματα βρόχων (αναθέσεις, συνθήκες, επαναλήψεις) και επιδεικνύουν ομοιόμορφες εξαρτήσεις δια του σώματος του βρόχου. Ο νεωτερισμός του εργαλείου CRONUS είναι διπλός: πρώτον, προσδιορίζει το ιδανικό υπερ-επίπεδο δρομολόγησης χρησιμοποιώντας τον αλγόριθμο QuickHull, ο οποίος είναι πιο αποδοτικός από προηγούμενα χρησιμοποιηθείσες μεθόδους και δεύτερον, υλοποιεί έναν απλό και γρήγορο δυναμικό κανόνα, που ονομάζουμε Διαδοχική Δυναμική Δρομολόγηση (Successive Dynamic Scheduling – SDS), για τη δρομολόγηση κατά το χρόνο εκτέλεσης των επαναλήψεων του βρόχου κατά μήκος του ιδανικού υπερ-επιπέδου. Αυτή η πολιτική δρομολόγησης ενισχύει την τοπικότητα των δεδομένων και βελτιώνει το συνολικό χρόνο εκτέλεσης. Για το συγχρονισμό των διεργασιών του παράλληλου προγράμματος, το CRONUS έχει τη δυνατότητα να επιλέξει μεταξύ του συνδυασμού της χρήσης του μηχανισμού των καθολικών σηματοφορέων που περιγράφηκε στο προηγούμενο κεφάλαιο και περάσματος μηνυμάτων για ανταλλαγή των πραγματικών δεδομένων και της αποκλειστικής χρήσης ανταλλαγής μηνυμάτων.

Το εργαλείο CRONUS παρέχει τελικά μια αποτελεσματική προγραμματιστική βιβλιοθήκη για το χρόνο εκτέλεσης (*runtime library*), ειδικά σχεδιασμένη για να ελαχιστοποιεί το κόστος επικοινωνίας, η οποία αποδίδει καλύτερα από άλλα γενικότερα συστήματα όπως είναι η πλατφόρμα UPC του πανεπιστημίου του Berkeley. Η συνολική απόδοση του εργαλείου CRONUS και της συνοδευτικής βιβλιοθήκης του, αποτιμήθηκε με σειρά εξαντλητικών δοκιμών. Τρεις αντιπροσωπευτικές περιπτώσεις μελετήθηκαν: ο αλγόριθμος διάχυσης Floyd-Steinberg, ο αλγόριθμος μεταβατικού κλεισίματος Transitive Closure και τέλος, ο αλγόριθμος εκτίμησης κίνησης FSBM. Τα πειραματικά αποτελέσματα επιβεβαιώνουν την αποδοτικότητα του παράλληλου κώδικα. Τα πειράματα παρουσιάζουν επιτάχυνση που κυμαίνεται από 1,18 (με ιδανική

τιμή το 4) μέχρι 12,19 (με ιδανική τιμή το 16) σε συστήματα κατανεμημένης μνήμης και από 3,60 (με ιδανική τιμή το 4) μέχρι 15,79 (με ιδανική τιμή το 16) σε συστήματα μοιραζόμενης μνήμης. Το εργαλείο CRONUS υπερσκελίζει την πλατφόρμα UPC σε ποσοστό από 5% έως και 95%, ανάλογα με το υπό εξέταση παράδειγμα.

3.1 Εισαγωγή

Η παραλληλοποίηση υπολογιστικά απαιτητικών προγραμμάτων οδηγεί σε δραματική βελτίωση της απόδοσής τους. Συνήθως αυτού του είδους τα προγράμματα περιέχουν συνεχόμενες επαναλήψεις, η πλειοψηφία των οποίων έχουν την μορφή φωλιασμένων βρόχων. Οι επαναλήψεις μέσα στο σώμα ενός βρόχου μπορεί να είναι είτε ανεξάρτητες επαναλήψεις είτε εξαρτώμενες από προγενέστερες επαναλήψεις. Οι τελευταίες μπορεί να είναι ομοιόμορφες (σταθερές) ή μη ομοιόμορφες κατά τη διάρκεια εκτέλεσης του προγράμματος. Στη διατριβή αυτή ασχολούμαστε με τις περιπτώσεις ομοιόμορφων εξαρτήσεων και παρουσιάζουμε μια πρωτότυπη πλατφόρμα δρομολόγησης και αυτόματης παραγωγής παράλληλου κώδικα με το όνομα CRONUS.

3.1.1 Σχετική εργασία

Η δρομολόγηση φωλιασμένων βρόχων με ομοιόμορφες εξαρτήσεις αρχικά μελετήθηκε από τον Lamport, ο οποίος τεμάχισε το χώρο δεικτών σε υπερ-επίπεδα [3.1], με βασική ιδέα ότι όλα τα σημεία που βρίσκονται στο ίδιο υπερ-επίπεδο μπορούν να εκτελεσθούν παράλληλα. Ο Darte απέδειξε ότι η μέθοδος αυτή είναι σχεδόν βέλτιστη [3.2]. Οι Moldovan, Shang και Darte έκαναν χρήση ιδιοφαντικών εξισώσεων [3.3], γραμμικού προγραμματισμού σε υποχώρους [3.4] και ακέραιου προγραμματισμού [3.2], αντίστοιχα, στη προσπάθεια να βρουν το ιδανικό υπερ-επίπεδο. Ένα κοινό χαρακτηριστικό όλων αυτών των μεθόδων είναι η χρήση αλγορίθμων εκθετικής πολυπλοκότητας. Στο [3.5] αποδείχθηκε ότι η χρονική πολυπλοκότητα μπορεί να βελτιωθεί κάνοντας χρήση αλγορίθμων από το χώρο της υπολογιστικής γεωμετρίας για τον προσδιορισμό του βέλτιστου υπερ-επιπέδου. Τέλος, μια διαφορετική και ευρέως χρησιμοποιούμενη τεχνική για την παραλληλοποίηση και δημιουργία κώδικα για φωλιασμένους βρόχους είναι το tiling [3.6], [3.7], [3.8], [3.9].

Οι επαναλήψεις του σώματος του βρόχου μπορούν να δρομολογηθούν στατικά, κατά τη διάρκεια του χρόνου μετάφρασης του προγράμματος ή δυναμικά, κατά τη διάρκεια του χρόνου εκτέλεσής του. Η *στατική δρομολόγηση* (*static scheduling*) περιλαμβάνει τη δημιουργία, κατά το χρόνο μετάφρασης, ενός χρονοδιαγράμματος για την εκτέλεση των επαναλήψεων, τέτοιου ώστε να μην παραβιάζονται οι περιορισμοί προτεραιότητας και συμπληρώνεται με τη χωρική διάταξη των υπολογισμών. Η *δυναμική δρομολόγηση* (*dynamic scheduling*) περιλαμβάνει την εξεύρεση ενός κατάλληλου κανόνα τέτοιου ώστε να προσδιορίζεται κατά τη διάρκεια του χρόνου εκτέλεσης ποια επανάληψη του βρόχου να υπολογίζει κάθε επεξεργαστής, αντί για τον επακριβή προσδιορισμό αυτής της ακολουθίας κατά τη διάρκεια του χρόνου μετάφρασης του προγράμματος.

Μέχρι τώρα, οι αλγόριθμοι δυναμικής δρομολόγησης χρησιμοποιούνταν κυρίως για βρόχους χωρίς εξαρτήσεις. Στο κεφάλαιο αυτό παρουσιάζεται πως είναι δυνατό να σχεδιασθούν αποδοτικοί αλγόριθμοι δυναμικής δρομολόγησης ακόμα και για βρόχους με εξαρτήσεις, υπό την προϋπόθεση ότι ο κανόνας με τον οποίο προσδιορίζεται ποια επανάληψη του βρόχου πρέπει να εκτελεστεί στη συνέχεια είναι υπολογιστικά ανέξοδη, όπως ισχύει και στις περιπτώσεις που εξετάζουμε. Πιο συγκεκριμένα, στη διατριβή αυτή περιλαμβάνουμε μια προσέγγιση λεπτοκομμένης παραλληλοποίησης για τη δυναμική δρομολόγηση βρόχων με εξαρτήσεις, η οποία ωστόσο μπορεί να αποτελέσει και τη βάση για άλλες πιο εξειδικευμένες μεθόδους χοντροκομμένου προγραμματισμού.

Όπως αποδεικνύεται και στο [3.5], το πρόβλημα της ανεύρεσης του βέλτιστου υπερ-επιπέδου μπορεί να αναχθεί στο απλούστερο της ανεύρεσης του κυρτού περιγράμματος (*convex hull*) των διανυσμάτων εξάρτησης και του τερματικού σημείου του βρόχου. Η χρήση υπολογιστικής γεωμετρίας για τον προσδιορισμό του κυρτού περιγράμματος και συνεπώς του βέλτιστου υπερ-επιπέδου είναι αποδοτικότερη από όλες τις άλλες προσεγγίσεις καθώς προσδίδει μικρότερη χρονική πολυπλοκότητα [3.5]. Το CRONUS, είναι ένα εργαλείο παραγωγής παράλληλου κώδικα, που χρησιμοποιεί τον αλγόριθμο κυρτού περιγράμματος QuickHull [3.12] για να υπολογίσει το βέλτιστο υπερ-επίπεδο. Έτσι, σαρώνοντας το χώρο δεικτών ενός βρόχου κατά τη διεύθυνση αυτού του υπερ-επιπέδου, οδηγούμαστε στο βέλτιστο χρόνο εκτέλεσης. Σημειώνουμε ότι αρχικές εκδόσεις της πλατφόρμας CRONUS παρουσιάστηκαν στο [3.10] και [3.11],

αλλά δεν περιελάμβαναν τον αλγόριθμο QuickHull, υστερώντας έτσι σημαντικά σε απόδοση.

3.1.2 Λειτουργία CRONUS

Το CRONUS πραγματοποιεί τη δρομολόγηση υπολογισμών, τη χαρτογράφηση υπολογισμών και την παραγωγή παράλληλου κώδικα για γενικούς βρόχους που έχουν αυθαίρετους χρόνους εκτέλεσης και επικοινωνίας. Γενικοί βρόχοι είναι εκείνοι οι φωλιασμένοι βρόχοι για τους οποίους το σώμα του βρόχου αποτελείται από γενικές εντολές προγράμματος (όπως αναθέσεις, συνθήκες και επαναλήψεις) και οι οποίοι είτε εμφανίζουν ομοιόμορφες εξαρτήσεις που μεταφέρονται με το σώμα του βρόχου είτε αντίθετα δεν παρουσιάζουν καθόλου τέτοιες εξαρτήσεις μέσα στο βρόχο. Αυτό συνεπάγεται τον προσδιορισμό του πότε (*χρόνος*) και του πού (*τόπος*) οι διάφοροι υπολογισμοί εκτελούνται, καθώς και τη δημιουργία παράλληλου κώδικα έτσι ώστε κάθε επεξεργαστής να εκτελεί ρητά τους αναλογούντες υπολογισμούς (*computation*) και επικοινωνία (*communication*). Στη συνέχεια, ο παράλληλος κώδικας μεταγλωττίζεται και εκτελείται στο υπολογιστικό σύστημα προορισμό (*target system*). Όπως εξηγείται αναλυτικότερα και σε επόμενη ενότητα, είσοδος στο εργαλείο CRONUS είναι ένα αρχείο που περιέχει τμήματα κώδικα C και διάφορες οδηγίες για το εργαλείο. Το αρχείο εισόδου μετατρέπεται σε ένα πλήρες παράλληλο πρόγραμμα C που βασίζεται και εκτελείται σε MPI και το οποίο χρησιμοποιεί επιπροσθέτως ρουτίνες από τη βιβλιοθήκη χρόνου εκτέλεσης του CRONUS για να χειριστεί τέτοιου είδους καθήκοντα, όπως την ικανοποίηση των εξαρτήσεων μεταξύ των επαναλήψεων του βρόχου. Υπό την έννοια αυτή, το CRONUS χρησιμοποιεί την ίδια προσέγγιση με άλλα εργαλεία όπως το Berkeley UPC [3.13], τα οποία επίσης λειτουργούν με τη μετατροπή των δεδομένων εισόδου τους σε κώδικα C, βασιζόμενα σε μια βιβλιοθήκη χρόνου εκτέλεσης που χειρίζεται τα καθήκοντα που σχετίζονται με τη σωστή παράλληλη εκτέλεση του προγράμματος.

Οι πρώτες υλοποιήσεις του CRONUS δεν περιελάμβαναν τον αλγόριθμο QuickHull και βασίζονταν αποκλειστικά στο μηχανισμό των καθολικών σηματοφορέων για το συγχρονισμό των παραλλήλων διεργασιών. Αυτό έγινε για λόγους ευκολίας και ταχύτητας, ώστε να κωδικοποιηθούν γρήγορα οι πρώτες εκδόσεις του CRONUS και να μπορέσουμε να έχουμε μια πρώτη επιβεβαίωση ότι ο συνολικός τρόπος αντιμετώπισης

του προβλήματος της αυτόματης παραλληλοποίησης κώδικα που ακολουθούμε είναι ρεαλιστικός, δίνει σωστά αποτελέσματα και κάποια μετρήσιμη επιτάχυνση. Ύστερα από τα πρώτα θετικά αποτελέσματα, ακολούθησε η πρώτη βελτίωση που ήταν η υιοθέτηση του αλγορίθμου QuickHull και οδήγησε σε αξιοσημείωτη αύξηση της απόδοσης του παράλληλου προγράμματος. Ωστόσο, σε πολλές περιπτώσεις η επιτάχυνση δεν ήταν η αναμενόμενη και με πληρέστερη ανάλυση αποδείχθηκε πως η χρήση του μηχανισμού των καθολικών σηματοφορέων για συγχρονισμό των παραλλήλων διεργασιών χωρίς τροποποιήσεις και προσαρμογή στις ιδιαιτερότητες των αλγορίθμων με φωλιασμένους βρόχους εισάγει επιπλέον καθυστερήσεις. Αυτό βέβαια ήταν ίσως αναμενόμενο, καθώς η γενικότητα και η ευκολία στον προγραμματισμό έχει συνήθως αρνητικό αντίκτυπο στην απόδοση ενός προγράμματος. Για την αντιμετώπιση των καθυστερήσεων αυτών οδηγηθήκαμε τελικά στην αντικατάσταση της αποκλειστικής χρήσης του μηχανισμού των καθολικών σηματοφορέων για συγχρονισμό και της υιοθέτησης δύο άλλων πολυπλοκότερων προγραμματιστικά αλλά αποδοτικότερων μεθόδων, όπως αναλυτικά περιγράφεται και στην παράγραφο 3.1.3.

Η τελική προσέγγιση μας αποτελείται από τρία βήματα:

Βήμα 1 (*Preprocessing*): Προσδιορισμός της βέλτιστης ομάδας (οικογένειας) υπερ-επιπέδων χρησιμοποιώντας τον αλγόριθμο κυρτού περιγράμματος QuickHull. Καθορισμός της λεξικογραφικής διάταξης στα υπερ-επίπεδα αυτά, υπολογίζοντας το *ελάχιστο* και το *μέγιστο* σημείο για κάθε τέτοιο υπερ-επίπεδο.

Βήμα 2 (*Scheduling*): Δρομολόγηση των επαναλήψεων κατά μήκος του βέλτιστου υπερ-επιπέδου δυναμικά κατά το χρόνο εκτέλεσης (*on-the-fly*) χρησιμοποιώντας τον αλγόριθμο Διαδοχικής Δυναμικής Δρομολόγησης (*SDS*).

Βήμα 3 (*Code Generation*): Παραγωγή φορητού παράλληλου κώδικα με χρήση του εργαλείου CRONUS.

Το βήμα 1 πραγματοποιείται κατά τη μεταγλώττιση του προγράμματος, ενώ τα βήματα 2 και 3 πραγματοποιούνται συνδυασμένα κατά την εκτέλεση του

προγράμματος. Το βήμα 2 πραγματοποιείται κατά το χρόνο εκτέλεσης γιατί μπορούμε να εκμεταλλευτούμε την ομοιομορφία του χώρου δεικτών και να χρησιμοποιήσουμε κατά το βέλτιστο τον προσαρμοστικό κανόνα Διαδοχικής Δυναμικής Δρομολόγησης (*SDS*).

3.1.3 Μέθοδοι συγχρονισμού στο CRONUS

Ένας από τους πιο σημαντικούς παράγοντες που επηρεάζουν την αποτελεσματικότητα του παραγόμενου παράλληλου κώδικα είναι ο τρόπος συγχρονισμού των παράλληλων διεργασιών που δημιουργούνται για την επίλυση του προβλήματος, καθώς λόγω των εξαρτήσεων είναι αναγκαία η ανταλλαγή δεδομένων μεταξύ των διεργασιών, δεδομένων που έχουν ήδη υπολογιστεί σε προηγούμενες επαναλήψεις του σώματος του βρόχου.

Η πιο απλή μέθοδος συγχρονισμού θα ήταν η χρήση ομάδων (*groups*) και φραγμάτων συγχρονισμού (*barrier synchronization*). Η μέθοδος αυτή, ωστόσο, επιφέρει υψηλό διαχειριστικό κόστος για τη δημιουργία και την ενημέρωση των ομάδων των διεργασιών που πρέπει να ανταλλάξουν μηνύματα με τα ήδη υπολογισμένα δεδομένα, ενώ η υποχρέωση απόλυτου συγχρονισμού όλων των διεργασιών μιας ομάδας αυξάνει τον άεργο χρόνο των επιμέρους διεργασιών, οδηγώντας έτσι σε σημαντική αύξηση του συνολικού παράλληλου χρόνου εκτέλεσης. Για τους λόγους αυτούς στο εργαλείο CRONUS δε γίνεται καθόλου χρήση barriers.

Η μέθοδος συγχρονισμού που χρησιμοποιείται στο CRONUS είναι η ανταλλαγή μηνυμάτων σημείο-προς-σημείο (*point-to-point communication*) με χρήση των blocking κλήσεων του MPI MPI_Send() και MPI_Recv(). Ο συγχρονισμός επιτυγχάνεται πάντα μεταξύ δύο διεργασιών και κάθε διεργασία στέλνει από ένα μήνυμα σε κάθε διεργασία που περιμένει δεδομένα από αυτή και στη συνέχεια περιμένει ένα μήνυμα από κάθε διεργασία από την οποία αναμένει δεδομένα. Με αυτό τον τρόπο, όταν μια διεργασία λάβει το σύνολο των δεδομένων που περιμένει από άλλες διεργασίες μπορεί να συνεχίσει την εκτέλεσή της, άσχετα με την κατάσταση των υπολοίπων διεργασιών σε σχέση και με την τρέχουσα επανάληψη του σώματος του βρόχου που εκτελείται.

Εναλλακτικά, ως μέθοδος συγχρονισμού στο CRONUS δοκιμάστηκε ο συνδυασμός της χρήσης του μηχανισμού των καθολικών σηματοφορέων και των non-blocking κλήσεων

του MPI MPI_Isend() και MPI_Irecv(). Το θεωρητικό πλεονέκτημα της μεθόδου είναι ότι ουσιαστικά γίνεται ένας λεπτός διαχωρισμός του συγχρονισμού των παράλληλων διεργασιών και της ανταλλαγής των δεδομένων. Ο συγχρονισμός γίνεται με το μηχανισμό των καθολικών σηματοφορέων, όπως περιγράφεται και στο κεφάλαιο 2, και όταν φτάσουμε στο σημείο όπου δύο διεργασίες είναι πραγματικά έτοιμες για την ανταλλαγή δεδομένων, οι διεργασίες απελευθερώνονται από τις ουρές των καθολικών σηματοφορέων και προχωρούν άμεσα στην κλήση των non-blocking συναρτήσεων MPI_Isend() και MPI_Irecv(), ώστε να κερδίσουμε σε απόδοση με την απαλοιφή της αντιγραφής των δεδομένων σε ενδιάμεσους απομονωτές (*buffers*).

Έχοντας υλοποιήσει τις δύο παραπάνω μεθόδους συγχρονισμού, βασιστήκαμε σε αναλυτικά πειραματικά δεδομένα για την αξιολόγησή τους. Από τα αποτελέσματα των συγκριτικών πειραματικών μετρήσεων αποδείχθηκε ότι σε συστήματα κοινής μοιραζόμενης μνήμης η δεύτερη μέθοδος με τη χρήση του μηχανισμού των καθολικών σηματοφορέων υστερεί της πρώτης με την άμεση χρήση blocking κλήσεων του MPI. Αυτό είναι αναμενόμενο, καθώς ο μηχανισμός των καθολικών σηματοφορέων απευθύνεται κυρίως σε συστήματα κατανεμημένης μνήμης ενώ στα συστήματα μοιραζόμενης μνήμης επιβαρύνει την εκτέλεση του παράλληλου προγράμματος καθώς εισάγει ένα νέο σύνολο μηνυμάτων που ανταλλάσσονται μεταξύ των διεργασιών του παράλληλου προγράμματος.

Στην περίπτωση των συστημάτων κατανεμημένης μνήμης τα πράγματα είναι πιο συγκεχυμένα ως προς τη σχετική απόδοση των δύο μεθόδων συγχρονισμού, καθώς η δεύτερη μέθοδος με τη χρήση του μηχανισμού των καθολικών σηματοφορέων δίνει αποτελέσματα που κυμαίνονται από 5% χειρότερα έως 7% καλύτερα σε σχέση με τα αντίστοιχα παραδείγματα με χρήση της πρώτης μεθόδου συγχρονισμού, ανάλογα με το υπό εξέταση παράδειγμα και τα χαρακτηριστικά του μοτίβου επικοινωνίας που παρουσιάζει. Δυστυχώς στην παρούσα υλοποίηση του εργαλείου CRONUS δεν είναι δυνατός ο εκ των προτέρων, δηλαδή κατά τη φάση της μεταγλώττισης και παραλληλοποίησης ενός προγράμματος, προσδιορισμός της αποδοτικότερης μεθόδου συγχρονισμού για συστήματα κατανεμημένης μνήμης. Έτσι έχει υιοθετηθεί η αποκλειστική χρήση της πρώτης μεθόδου συγχρονισμού σε όλους τους τύπους συστημάτων, καθιστώντας δυνατή και την άμεση σύγκριση της απόδοσης του εργαλείου CRONUS σε συστήματα μοιραζόμενης και κατανεμημένης μνήμης για το

ίδιο παράδειγμα. Φυσικά η επιλογή αυτή, δηλαδή να μην χρησιμοποιηθούν καθολικοί σηματοφορείς, υποχρεώνει σε περισσότερη προγραμματιστική προσπάθεια, η οποία αντισταθμίζεται από την καλύτερη απόδοση σε κρίσιμες εφαρμογές, όπως αυτή του CRONUS. Σε περιπτώσεις όμως που ο χρόνος υλοποίησης είναι σημαντικότερος της όποιας πιθανής μικρής μείωσης της απόδοσης, τότε οι καθολικοί σηματοφορείς είναι η καλύτερη επιλογή.

3.1.4 Συνεισφορά και πλεονεκτήματα CRONUS

Η συνεισφορά της πλατφόρμας παραγωγής παράλληλου κώδικα CRONUS είναι:

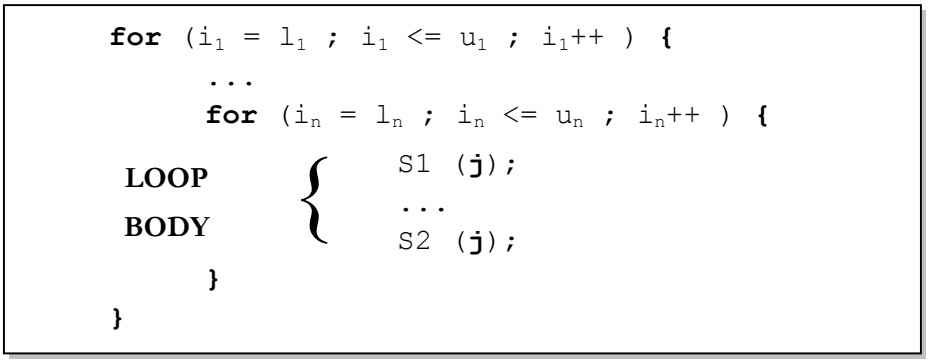
- Υποστηρίζει τη χρήση των γεωμετρικών μεθόδων και ιδιαίτερα του αλγορίθμου κυρτού περιγράμματος QuickHull, με τα πλεονεκτήματα που αυτό συνεπάγεται, δηλαδή απλότητα και μικρότερη πολυπλοκότητα έναντι άλλων μεθόδων.
- Υλοποιεί έναν αλγόριθμο αποκεντρωμένης δυναμικής εξισορρόπησης φορτίου για βρόχους με εξαρτήσεις, που ονομάζουμε Διαδοχική Δυναμική Δρομολόγηση (*Successive Dynamic Scheduling - SDS*).
- Λειτουργεί με ημιαυτόματο τρόπο, μπορεί να χειριστεί τόσο βρόχους με εξαρτήσεις όσο και γενικούς βρόχους χωρίς εξαρτήσεις και παράγει αποδοτικό παράλληλου κώδικα έτοιμο προς εκτέλεση.

Τα πλεονεκτήματα του εργαλείου CRONUS συνοψίζονται παρακάτω:

- Η αποτελεσματικότητα και η επεκτασιμότητα του αλγορίθμου Διαδοχικής Δυναμικής Δρομολόγησης (*SDS*) εξασφαλίζει ότι κατά τη διάρκεια της εκτέλεσης του παραλλήλου κώδικα δεν προκύπτουν τεχνητές καθυστερήσεις επικοινωνίας ή υπολογισμού, που δεν υπάρχουν στο αρχικό ακολουθιακό πρόγραμμα. Κάθε επεξεργαστής δαπανά το μεγαλύτερο μέρος του χρόνου εκτελώντας την κατάλληλη επανάληψη παρά για την επικοινωνία δεδομένων. Επιπρόσθετα, καθώς ακολουθεί ένα αποκεντρωμένο/κατανεμημένο σχήμα, μπορεί και κλιμακώνεται με το διαθέσιμο αριθμό επεξεργαστών.
- Η σχεδόν τέλεια εξισορρόπηση φορτίου που επιτυγχάνεται με την ισότιμη κατανομή των επαναλήψεων του βρόχου μεταξύ των επεξεργαστών σε ένα κυκλικό (*round-robin*) σχήμα, με την παραδοχή ότι το σύστημα είναι ομογενές.

- Η βελτιστοποιημένη βιβλιοθήκη χρόνου εκτέλεσης, η οποία χρησιμοποιείται από τα δημιουργούμενα παράλληλα προγράμματα ώστε να διαχειρίζονται αυτόματα και διαφανώς θέματα επικοινωνίας και τοπικότητας των δεδομένων. Τα πειράματα δείχνουν ότι ο κώδικας που υποστηρίζεται από τη βιβλιοθήκη χρόνου εκτέλεσης του CRONUS εκτελείται 5% – 95% ταχύτερα από τον ισοδύναμο παράλληλο κώδικα που εκτελείται κάτω από το Berkeley UPC.
- Η χρήση μιας ημιαυτόματης γεννήτριας κώδικα, η οποία διευκολύνει τη παραγωγή παράλληλου κώδικα χωρίς σημαντική συμβολή από την πλευρά του χρήστη, μειώνοντας σημαντικά τις καθυστερήσεις που συνδέονται με τη χειρόγραφη συγγραφή προγραμμάτων.
- Η φορητότητα του παράλληλου κώδικα που δημιουργείται. Καθώς το εργαλείο CRONUS χρησιμοποιεί μόνο κλήσεις της παράλληλης πλατφόρμας MPI, ο παράλληλος κώδικας μπορεί να εκτελεστεί άμεσα και χωρίς τροποποιήσεις σε συστήματα μοιραζόμενης μνήμης, συστήματα κατανεμημένης μνήμης, συστοιχίες σταθμών εργασίας, SMPs και MPPs.

Το υπόλοιπο του παρόντος κεφαλαίου είναι οργανωμένο ως εξής. Στην ενότητα 3.2 εισάγεται η ορολογία και παρέχονται οι βασικοί ορισμοί. Στην ενότητα 3.3 παρουσιάζεται η μέθοδος για την ανεύρεση του βέλτιστου υπερ-επιπέδου δρομολόγησης. Η λεξικογραφική διάταξη η οποία αποτελεί τη θεωρητική βάση της πλατφόρμας CRONUS παρουσιάζεται στην ενότητα 3.4 και στην ενότητα 3.5 εξηγείται πως ο λειτουργεί ο αλγόριθμος δρομολόγησης του CRONUS. Ο τρόπος χρήσης, πειραματικά αποτελέσματα αλλά και αξιολόγηση του CRONUS παραθέτονται σε επόμενα κεφάλαια.



Σχήμα 3.1: Υπολογιστικό μοντέλο

3.2 Ορολογία και ορισμοί

Ορισμός 3.1: Η κατηγορία των *γενικών βρόχων* περιέχει τους *φωλιασμένους βρόχους* που περιέχουν στο σώμα του βρόχου τους γενικές δηλώσεις προγράμματος όπως εντολές συνθήκης `if` και άλλους βρόχους `for/while`.

Ορισμός 3.2: Ο χώρος δεικτών ενός φωλιασμένου βρόχου ορίζεται σαν ένας n-διάστατος καρτεσιανός χώρος $\mathbf{J} = \{ j = (j_1, \ldots, j_n) \in \mathbb{N}^n \mid l_r \le j_r \le u_r, 1 \le r \le n \}$, όπου ο δείκτης j ορίζει μια επανάληψη του φωλιασμένου βρόχου. Τα κάτω και άνω όρια για τους δείκτες του βρόχου είναι $l_1, \ldots, l_n$ και $u_1, \ldots, u_n$, αντίστοιχα. Το βάθος της φωλιάς του βρόχου είναι n, το οποίο είναι ίσο με τη διάσταση του χώρου δεικτών J. Η πρώτη επανάληψη του φωλιασμένου βρόχου, που δηλώνεται ως $\mathbf{L} = (l_1, \ldots, l_n)$, είναι το *αρχικό σημείο* του χώρου δεικτών και η τελευταία επανάληψη, που δηλώνεται ως $\mathbf{U} = (u_1, \ldots, u_n)$, είναι το *τερματικό σημείο* του χώρου δεικτών. Τα στοιχεία του χώρου J, που ονομάζονται επίσης και σημεία του χώρου δεικτών, μπορούν να διαταχθούν λεξικογραφικά όπως εξηγείται στη συνέχεια στην ενότητα 3.4.

Ορισμός 3.3: Ένα σημείο **j** του χώρου δεικτών εξαρτάται, εν γένει, από άλλα σημεία του χώρου δεικτών $i_1, \ldots, i_m$. Η εξάρτηση αυτή εκφράζεται μέσω των *διανυσμάτων εξάρτησης*. Τα διανύσματα εξάρτησης είναι πάντα μεγαλύτερα από το διάνυσμα $\mathbf{0} = (0, \ldots, 0)$. Στο μοντέλο μας όλα τα διανύσματα εξάρτησης είναι ομοιόμορφα, δηλαδή τα στοιχεία τους είναι σταθερές και όχι συναρτήσεις ορισμένες επί των δεικτών του χώρου. Το σύνολο των ομοιόμορφων διανυσμάτων εξάρτησης ονομάζεται $\mathbf{DS} = \{\vec{d}_1, \ldots, \vec{d}_m\}$, $m \ge n$, όπου m είναι ο αριθμός των διανυσμάτων εξάρτησης.

```
for (i1 = 0 ; i1 <= N1 ; i1++ ) {
    for (i2 = 0 ; i2 <= N2 ; i2++ ) {

LOOP                S1: A[i1][i2] = 2*A[i1-1][i2-8]+A[i1-8][i2-1]+3*B[i1-2][i2-5];
BODY                S2: B[i1][i2] = 0;
                    S3: for (i3 = 0 ; i3 < i2 ; i3++ )
                            B[i1][i2] += C[i1][i3]*C[i1][i2];
                    S4: B[i1][i2] += B[i1-3][i2-3]+B[i1-6][i2-2];
    }
}
```

Σχήμα 3.2: Δισδιάστατος βρόχος με ένα γενικό σώμα βρόχου και διανύσματα εξαρτήσεων $\vec{d}_1$ =(1, 8), $\vec{d}_2$ =(2, 5), $\vec{d}_3$ =(3, 3), $\vec{d}_4$ =(6, 2) και $\vec{d}_5$ =(8, 1)

Παράδειγμα 3.1: Το Σχήμα 3.2 παρουσιάζει ένα παράδειγμα φωλιασμένου βρόχου δύο διαστάσεων (2D), με ένα τεχνητό γενικό σώμα βρόχου και μέγεθος χώρου δεικτών $| J | = N_1 \times N_2$. Σε αυτό το σχήμα, το σημείο $A[i_1][i_2]$ εξαρτάται από το σημείο $A[i_1\text{-}1][i_2\text{-}8]$ και αυτό αντικατοπτρίζεται στην αναπαράσταση του χώρου δεικτών από το διάνυσμα εξάρτησης που συνδέει το σημείο του χώρου δεικτών (i_1, i_2) με το σημείο $(i_1 - 1, i_2 - 8)$, όπως φαίνεται και στο Σχήμα 3.3. Η ύπαρξη ενός διανύσματος εξάρτησης μεταξύ των δυο σημείων του χώρου δεικτών σημαίνει ότι για να υπολογίσουμε το σημείο $A[i_1][i_2]$, το στοιχείο του πίνακα $A[i_1\text{-}1][i_2\text{-}8]$ πρέπει να έχει ήδη υπολογιστεί. Οι συντεταγμένες του παραπάνω διανύσματος εξάρτησης είναι $(i_1 - (i_1 - 1), i_2 - (i_2 - 8)) = (1, 8)$. Τα υπόλοιπα διανύσματα εξάρτησης, τα οποία καθορίζονται χρησιμοποιώντας την ίδια μέθοδο, είναι: $\vec{d}_1 = (1, 8)$, $\vec{d}_2 = (2, 5)$, $\vec{d}_3 = (3, 3)$, $\vec{d}_4 = (6, 2)$ και $\vec{d}_5 = (8, 1)$.

Αυτά τα διανύσματα εξάρτησης απεικονίζεται στο Σχήμα 3.3. Σημειώνουμε ότι τα διανύσματα αυτά είναι σταθερά, δηλαδή είναι το ίδια για κάθε σημείο του χώρου δεικτών. Αυτό συνεπάγεται ότι κάθε σημείο του χώρου δεικτών (i_1, i_2) εξαρτάται πάντα από τα ίδια ήδη υπολογισμένα σημεία του χώρου δεικτών $(i_1 - 1, i_2 - 8)$, $(i_1 - 2, i_2 - 5)$, $(i_1 - 3, i_2 - 3)$, $(i_1 - 6, i_2 - 2)$ και $(i1 - 8, i2 - 1)$. Ωστόσο, εάν μια δήλωση του φωλιασμένου βρόχου, για παράδειγμα η S_1, ήταν η $A[i_1][i_2] = A[i_1 - i_2][i_2 - i_1]$, τότε το αντίστοιχο διάνυσμα εξάρτησης θα ήταν $\vec{d}' = (i_2, i_1)$. Αυτό το διάνυσμα εξάρτησης δεν είναι ομοιόμορφο, καθώς ορίζεται ως μια συνάρτηση των δεικτών i_1 και i_2.

3.3 Αναζήτηση του βέλτιστου υπερ-επιπέδου δρομολόγησης χρησιμοποιώντας τον αλγόριθμο κυρτού περιγράμματος

Η μέθοδος του υπερ-επιπέδου [3.1] (hyperplane or wavefront method) ήταν μια από τις πρώτες μεθόδους που χρησιμοποιήθηκαν για την παραλληλοποίηση φωλιασμένων βρόχων με ομοιόμορφες εξαρτήσεις και ως τέτοια αποτέλεσε τη βάση των περισσοτέρων ευριστικών αλγορίθμων.

Ορισμός 3.4: Ένα υπερ-επίπεδο είναι ένα n-διάστατο επίπεδο που αποτελείται από ένα υποσύνολο ανεξαρτήτων σημείων επανάληψης του χώρου δεικτών ([3.1], [3.14, σ.75]). Τα σημεία αυτά εξαρτώνται μόνο από προγενέστερες χρονικά επαναλήψεις με ήδη υπολογισμένα σημεία του χώρου δεικτών και συνεπώς μπορούν να εκτελεστούν

παράλληλα, το οποίο έχει ως αποτέλεσμα τη σημαντική μείωση του συνολικού χρόνου εκτέλεσης. Τυπικά, ένα υπερεπίπεδο $\Pi_k(a_1, \ldots, a_n)$ όπου a_i, $k \in \mathbb{N}$, αποτελείται από τα σημεία του χώρου δεικτών $\mathbf{j} = (j_1, \ldots, j_n) \in J$ που ικανοποιούν την εξίσωση $a_1j_1 + \ldots + a_nj_n = k$. Ο αριθμός του συνόλου των σημείων του υπερ-επιπέδου Π_k ονομάζεται πληθικότητα του Π_k και σημειώνεται ως $|\Pi_k|$.

Για όλα τα σημεία του χώρου δεικτών $\mathbf{j} = (j_1, \ldots, j_n)$ που περιλαμβάνονται στο υπερ-επίπεδο $\Pi_k(a_1, \ldots, a_n)$, το άθροισμα $\sum_{r=1}^{n} a_r j_r$ έχει την ίδια τιμή k. Ειδικότερα, το τερματικό σημείο $\mathbf{U} = (u_1, \ldots, u_n)$ βρίσκεται στο υπερ-επίπεδο $k = a_1u_1 + \ldots + a_nu_n$.

Παράδειγμα 3.2: Θεωρούμε ένα δισδιάστατο χώρο δεικτών (2D) και μια οικογένεια υπερ-επιπέδων που ορίζονται από την εξίσωση $x_1 + x_2 = k$. Για $k = 1$, τα σημεία του χώρου δεικτών που βρίσκονται επί του υπερ-επιπέδου Π_1 είναι τα (0, 1) και (1, 0) και η πληθικότητα αυτού του υπερ-επιπέδου είναι $|\Pi_1| = 2$. Τα σημεία του χώρου δεικτών που βρίσκονται επί του υπερ-επιπέδου Π_2 είναι τα (0, 2), (1, 1) και (2, 0), και η πληθικότητα αυτού του υπερ-επιπέδου είναι $|\Pi_2| = 3$.

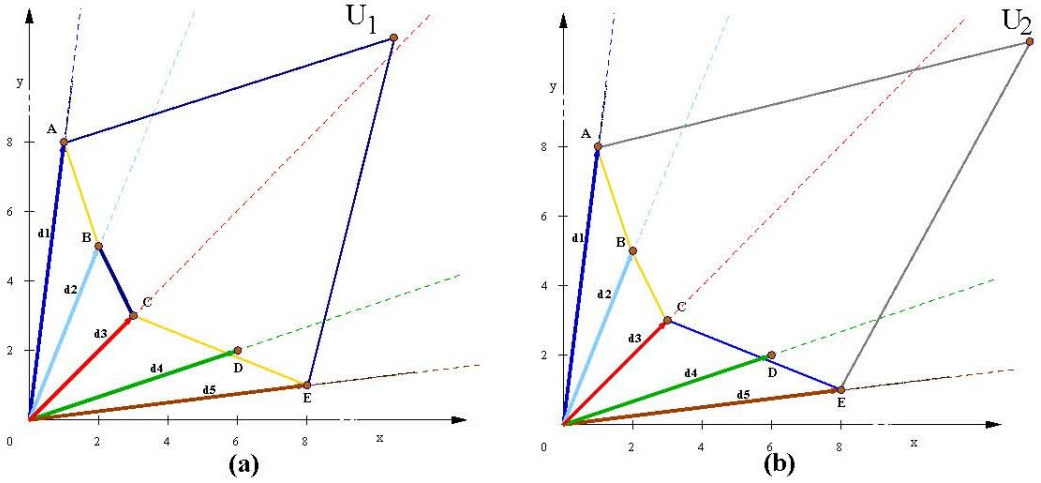
Οι ερευνητές χρησιμοποίησαν ιδιοφαντικές εξισώσεις, γραμμικό προγραμματισμό σε υποχώρους ή ακέραιο προγραμματισμό, προκειμένου να υπολογίσουν το βέλτιστο υπερ-επίπεδο για ένα δεδομένο χώρο δεικτών. Στα [3.5] και [3.15], το πρόβλημα της εύρεσης του βέλτιστου υπερ-επιπέδου για φωλιασμένους βρόχους ομοιόμορφων εξαρτήσεων μειώθηκε στο πρόβλημα υπολογισμού του κυρτού περιγράμματος των διανυσμάτων εξάρτησης και του τερματικού σημείου. Το κυρτό περίγραμμα που σχηματίζεται από τα σημεία $j_1, \ldots, j_m$ ορίζεται ως $CH = \{ j \in \mathbb{N}^n \mid j = \lambda_1 j_1 + \ldots + \lambda_m j_m$, όπου $\lambda_1, \ldots, \lambda_m \geq 0$ και $\lambda_1 + \ldots + \lambda_m = 1 \}$.

Παράδειγμα 3.3: Στο Σχήμα 3.3 τα διανύσματα εξάρτησης είναι $\vec{d_1} = (1, 8)$, $\vec{d_2} = (2, 5)$, $\vec{d_3} = (3, 3)$, $\vec{d_4} = (6, 2)$ και $\vec{d_5} = (8, 1)$. Εξετάζουμε τις ακόλουθες δύο περιπτώσεις:

- $N_1 = 75$, $N_2 = 90$, με αποτέλεσμα το τερματικό σημείο $U_1 = (75, 90)$ (βλ. Σχήμα 3.3(a)), και
- $N_1' = 105$, $N_2' = 90$, με αποτέλεσμα το τερματικό σημείο $U_2 = (105, 90)$ (βλ. Σχήμα 3.3(b)).

Όπως φαίνεται στο Σχήμα 3.3, στην πρώτη περίπτωση το κυρτό περίγραμμα είναι το πολύγωνο $ABCEU_1$ και στη δεύτερη περίπτωση το κυρτό περίγραμμα είναι το

πολύγωνο ABCEU$_2$. Το τερματικό σημείο U$_1$ ανήκει στον κώνο που ορίζεται από τα διανύσματα εξάρτησης $\vec{d}_2$ και $\vec{d}_3$ (βλ. Σχήμα 3.3(a)). Αυτό σημαίνει ότι το τερματικό σημείο U$_1$ μπορεί να γραφτεί ως $\lambda\vec{d}_2 + \lambda'\vec{d}_3$ με λ και $\lambda' \geq 0$. Ομοίως, το τερματικό σημείο U$_2$ ανήκει στον κώνο που καθορίζεται από τα $\vec{d}_3$ και $\vec{d}_5$ (βλ. Σχήμα 3.3(b)) και μπορεί επομένως να γραφτεί ως $\lambda\vec{d}_3 + \lambda'\vec{d}_5$ με λ και $\lambda' \geq 0$.



Σχήμα 3.3: Βέλτιστο υπερ-επίπεδο για δυο διαφορετικούς χώρους δεικτών

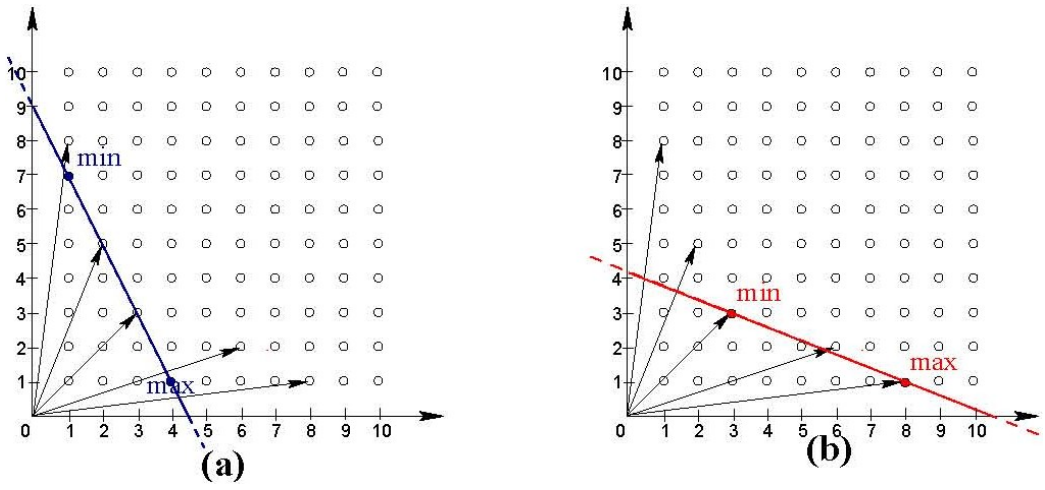
Στο [3.5] έχει αποδειχτεί ότι το βέλτιστο υπερ-επίπεδο ορίζεται από τις απολήξεις των διανυσμάτων εξάρτησης που διαμορφώνουν τον κώνο που περιέχει το τερματικό σημείο. Έτσι, το βέλτιστο υπερ-επίπεδο στην πρώτη περίπτωση είναι η γραμμή που ορίζεται από τις απολήξεις των διανυσμάτων ($\vec{d}_2$, $\vec{d}_3$), ενώ στη δεύτερη περίπτωση είναι η γραμμή που ορίζεται από τις απολήξεις των ($\vec{d}_3$, $\vec{d}_5$). Σημειώνουμε ότι το $\vec{d}_4$ είναι ένα εσωτερικό σημείο του κυρτού περιγράμματος και ως εκ τούτου δεν διαδραματίζει κανένα ρόλο στον καθορισμό του βέλτιστου υπερ-επιπέδου (για περισσότερες λεπτομέρειες ανατρέξτε στο [3.5]). Η ευθεία που διέρχεται από τα σημεία (2, 5) και (3, 3) βρίσκεται από την επίλυση της παρακάτω εξίσωσης:

$$\begin{vmatrix} x_1 - 2 & x_2 - 5 \\ 3 - 2 & 3 - 5 \end{vmatrix} = 0 \Leftrightarrow 2x_1 + x_2 = 9$$

Ομοίως, η εξίσωση της ευθείας που διέρχεται από τα σημεία (3, 3) και (8, 1) είναι η $2x_1 + 5x_2 = 21$. Ως εκ τούτου, για τους χώρους δεικτών των σχημάτων 3.3(a) και 3.3(b) η οικογένεια των βέλτιστων υπερ-επιπέδων περιγράφεται από τις $\Pi_k(2, 1) : 2x_1 + x_2 = k$ και $\Pi_k(2, 5) : 2x_1 + 5x_2 = k$, $k \in \mathbb{N}$, αντίστοιχα. Ο αλγόριθμος που χρησιμοποιείται για να υπολογιστεί το κυρτό περίγραμμα είναι ο αλγόριθμος QuickHull [3.12].

3.4 Λεξικογραφική διάταξη σε υπερ-επίπεδα

Ένα σημαντικό θέμα στη δυναμική δρομολόγηση είναι η εξεύρεση ενός προσαρμοστικού κανόνα ώστε να διατάζεται κάθε επεξεργαστής το τι πρέπει να κάνει κατά το χρόνο εκτέλεσης του προγράμματος αντί αυτό να προσδιορίζεται ρητά κατά το χρόνο μεταγλώττισης. Ο προσαρμοστικός κανόνας καθορίζει το επόμενο προς εκτέλεση σημείο του χώρου δεικτών καθώς και τα απαιτούμενα και ήδη εκτελεσμένα σημεία για κάθε επανάληψη του βρόχου. Για να καθοριστεί αποτελεσματικά ένας τέτοιος κανόνας, μια συνολική διάταξη (ή σε σειρά ταξινόμηση) όλων των σημείων του χώρου δεικτών είναι απαραίτητη. Στη συνέχεια καθορίζεται μια σειριακή ταξινόμηση που θα διευκολύνει τη στρατηγική δρομολόγησης που παρουσιάζεται.



Σχήμα 3.4: Ελάχιστα και μέγιστα σημεία σε υπερ-επίπεδα

Ορισμός 3.5: Ας υποθέσουμε ότι $\mathbf{i} = (i_1, \ldots, i_n)$ και $\mathbf{j} = (j_1, \ldots, j_n)$ είναι δύο σημεία του χώρου δεικτών. Λέμε ότι το σημείο i *είναι μικρότερο* από το σημείο j σύμφωνα με τη λεξικογραφική διάταξη και γράφουμε $\mathbf{i < j}$, εάν $i_1 = j_1 \wedge \ldots \wedge i_{r-1} = j_{r-1}$ και $i_r < j_r$ για κάποιο r, $1 \le r \le n$.

Στο υπόλοιπο της διατριβής χρησιμοποιείται πάντα η λεξικογραφική διάταξη μεταξύ των σημείων του χώρου δεικτών, δηλαδή, όταν εμφανίζεται μια σχέση **i < j** αυτό σημαίνει ότι το σημείο **i** είναι λεξικογραφικά μικρότερο από το σημείο **j**. Ο χώρος δεικτών μπορεί να διασχίζεται ακολουθώντας τα υπερ-επίπεδα, δημιουργώντας ένα μονοπάτι (γράφο) με μορφή ζιγκ-ζαγκ σε ένα δισδιάστατο χώρο δεικτών (2D), ή ένα σπειροειδές μονοπάτι (γράφο) σε ένα τρισδιάστατο (3D) ή υψηλότερης τάξης χώρο δεικτών. Με δεδομένο ένα συγκεκριμένο υπερ-επίπεδο, η λεξικογραφική διάταξη επάγει μια *καθολική διάταξη* των σημείων του.

Ορισμός 3.6: Το σημείο **j** του χώρου δεικτών που ανήκει στο υπερ-επίπεδο Π_k είναι το *ελάχιστο σημείο* του Π_k, αν το υπερ-επίπεδο Π_k δεν περιέχει κανένα άλλο σημείο **i** τέτοιο ώστε **i < j**. Ομοίως, το σημείο **j** του χώρου δεικτών που ανήκει στο υπερ-επίπεδο Π_k είναι το *μέγιστο σημείο* του Π_k, αν το υπερ-επίπεδο Π_k δεν περιέχει κανένα άλλο σημείο **i** τέτοιο ώστε **i > j**.

Είναι πιθανόν ένα συγκεκριμένο υπερ-επίπεδο να μην περιέχει κανένα σημείο του χώρου δεικτών. Κατά συνέπεια, αυτό το υπερ-επίπεδο δεν έχει ελάχιστο και μέγιστο σημείο. Για παράδειγμα, το υπερ-επίπεδο $\Pi_1(2, 5) : 2x_1 + 5x_2 = 1$ για το χώρο δεικτών του σχήματος 3.4, δεν περιέχει κανένα σημείο του χώρου δεικτών.

Η ανεύρεση του ελαχίστου σημείου ενός δεδομένου n-διάστατου υπερ-επίπεδου είναι ένα πρόβλημα βελτιστοποίησης. Επιπλέον, παρουσιάζει την ιδιότητα της βέλτιστης υποδομής, δηλαδή, η βέλτιστη λύση για την n-διαστάσεων περίπτωση εμπεριέχει τη βέλτιστη λύση για την (n–1)-διαστάσεων περίπτωση. Κατά συνέπεια, το ελάχιστο σημείο μπορεί να υπολογιστεί από έναν απλό αλγόριθμο δυναμικού προγραμματισμού με τη χρήση αναδρομής. Ο ψευδοκώδικας για τον αλγόριθμο αυτό παρατίθεται στο παράρτημα (Σχήμα Α.1). Με συμμετρικό τρόπο, ανάλογα υπολογίζονται και τα μέγιστα σημεία n-διάστατων υπερ-επίπεδων.

Ωστόσο, όταν ασχολούμαστε με φωλιασμένους βρόχους δύο διαστάσεων (2D) μπορούμε να επωφεληθούμε από τεχνικές δανεισμένες από το πεδίο των ιδιοφαντικών εξισώσεων, για την επιτάχυνση του υπολογισμού του ελαχίστου σημείου ενός υπερ-επίπεδου δύο διαστάσεων. Προτού όμως αναλύσουμε ποιες είναι οι τεχνικές αυτές, είναι ίσως αναγκαίο να συμφωνηθεί ότι παρόλο που είναι ειδική, η δισδιάστατη περίπτωση είναι πράγματι πολύ σημαντική για δυο βασικούς λόγους:

(1) Οι περισσότερες πρακτικές περιπτώσεις (τα περισσότερα από τα παραδείγματα στη βιβλιογραφία) εμπίπτουν σε αυτή την κατηγορία.

(2) Ακόμη και σε περιπτώσεις υψηλοτέρων διαστάσεων, θα μπορούσαμε να χρησιμοποιήσουμε, αν είναι απαραίτητο, τεχνικές μετασχηματισμού βρόχων (όπως την ανταλλαγή βρόχων, για περισσότερες λεπτομέρειες βλέπε [3.14]), προκειμένου να αντιμετωπιστεί ο υψηλότερης διάστασης βρόχος ως ένα απλός δισδιάστατος βρόχος. Αυτό δεν είναι μονό δυνατόν, αλλά είναι επίσης και συμφέρον, διότι οδηγεί σε υπολογιστικά απαιτητικότερα σώματα βρόχων, επιτρέποντας έτσι μια πιο τραχιά (*coarse grain*) προσέγγιση παραλληλισμού. Σημειώνεται ότι η πλατφόρμα CRONUS μπορεί να χειριστεί εύκολα πολύπλοκα σώματα βρόχων.

Θεωρούμε την εξίσωση:

$$a_1x_1 + a_2x_2 = k \qquad (3.1)$$

όπου $a_1, a_2, x_1, x_2, k \in \mathbb{Z}$. Μια λύση της εξίσωσης (3.1) είναι ένα διατεταγμένο ζευγάρι (j_1, j_2), όπου $j_1, j_2 \in \mathbb{Z}$, τέτοια ώστε $a_1j_1 + a_2j_2 = k$. Υποθέτοντας ότι $g = \gcd(a_1, a_2)$ και ότι δεν είναι ταυτόχρονα και το a_1 και το a_2 ίσα με μηδέν, τότε γνωρίζουμε ότι η εξίσωση (3.1) έχει απείρως πολλές λύσεις αν και μονό αν το g διαιρεί το k. Δεν έχει καμιά λύση αν το g δε διαιρεί το k (βλ. [3.16] και [3.17]). Εξάλλου, εάν (j_1, j_2) είναι μια λύση της εξίσωσης (3.1), τότε όλες οι υπόλοιπες είναι της μορφής:

$$\left(j_1 + \frac{la_2}{g},\ j_2 - \frac{la_1}{g} \right) \qquad (3.2)$$

όπου $l \in \mathbb{Z}$.

Απομένει να δείξουμε το πώς βρίσκουμε τις λύσεις της εξίσωσης (3.1), υποθέτοντας φυσικά ότι αυτές υπάρχουν. Αυτό μπορεί να γίνει εύκολα και αποτελεσματικά χρησιμοποιώντας τον βασιζόμενο σε πίνακες αλγόριθμο που παρουσιάζεται στο [3.16], και ο οποίος αποτελεί μια τροποποίηση της γκαουσιανής μεθόδου απαλοιφής. Η εξήγηση της τεχνικής αυτής γίνεται μέσω της παράθεσης του παρακάτω παραδείγματος.

Παράδειγμα 3.4: Ας υποθέσουμε ότι θέλουμε να βρούμε τις λύσεις της εξίσωσης $2x_1 + x_2 = 9$ (βλέπε και Σχήμα 3.4(a)). Για να επιτευχτεί αυτό γράφουμε διαδοχικά:

```
2 1 9          1 1 9          1 0 9
1 0      ⇒     1 0      ⇒     1 -1
0 1            -1 1           -1 2
```

Αν οι μεταβλητές που υπονοούνται στον πίνακα είναι οι u και v, όπου u, v ∈ ℤ, θα έχουμε τελικά ότι u = 9. Οι γενικές λύσεις δίνονται τότε από τις σχέσεις:

$$x_1 = u - v = 9 - v \quad \text{και} \quad x_2 = -u + 2v = 2v - 9$$

Ομοίως, μπορούμε να βρούμε τις λύσεις της εξίσωσης $2x_1 + 5x_2 = 21$ (βλέπε και Σχήμα 3.4(b)).

```
2 5 21         2 3 21         2 1 21         1 1 21         1 0 21
1 0      ⇒     1 -1     ⇒     1 -2     ⇒     3 -2     ⇒     3 -5
0 1            0 1            0 1            -1 1           -1 2
```

Υποθέτοντας ότι οι μεταβλητές που ενυπάρχουν στον τελευταίο πίνακα είναι οι u, v ∈ ℤ, βρίσκουμε τελικά ότι u = 21. Κατά συνέπεια, οι γενικές λύσεις δίνονται από τις σχέσεις:

$$x_1 = 3u - 5v = 3 \times 21 - 3v = 63 - 5v \quad \text{και} \quad x_2 = -u + 2v = 2v - 21$$

Στην περίπτωσή μας, μας ενδιαφέρουν μόνο τα σημεία του δισδιάστατου χώρου που βρίσκονται στο πρώτο τεταρτημόριο, δηλαδή τα σημεία εκείνα που έχουν και τις δυο συντεταγμένες τους μη αρνητικές. Λαμβάνοντας υπόψη τη συνθήκη αυτή, η εξίσωση (3.2) δίνει ότι $j_1 + \frac{la_2}{g} \geq 0$ και $j_2 - \frac{la_1}{g} \geq 0$, το οποίο με τη σειρά του συνεπάγεται ότι $la_2 + gj_1 \geq 0$ και $-la_1 + gj_2 \geq 0$ και τελικά, ότι:

$$-\frac{gj_1}{a_2} \leq l \leq \frac{gj_2}{a_1} \qquad (3.3)$$

Η παραπάνω εξίσωση είναι πολύ χρήσιμη, καθώς ο αριθμός των ακεραίων που κείτονται σε αυτό το διάστημα δίνει την πληθικότητα του υπερ-επίπεδου $\Pi_k(a_1, a_2)$.

Επιπλέον, το ελάχιστο σημείο του χώρου δεικτών του $\Pi_k(a_1, a_2)$ υπολογίζεται από την εξίσωση (3.2) αντικαθιστώντας σε αυτή την ελάχιστη τιμή του ακεραίου l που ικανοποιεί την ανισότητα (3.3). Αντίστοιχα και συμμετρικά, το μέγιστο σημείο του χώρου δεικτών του $\Pi_k(a_1, a_2)$ υπολογίζεται από την εξίσωση (3.2) με αντικατάσταση της μέγιστης τιμής του ακεραίου l που ικανοποιεί την ανισότητα (3.3).

Παράδειγμα 3.5: Συνεχίζοντας το προηγούμενο παράδειγμα, βρίσκουμε τον αριθμό των μη αρνητικών λύσεων της εξίσωσης $2x_1 + x_2 = 9$, το οποίο ισοδυναμεί με την ανεύρεση του συνόλου των σημείων του χώρου δεικτών που ανήκουν στο υπερ-επίπεδο $\Pi_9(2, 1)$ (Σχήμα 3.4(a)). Από την ανισότητα (3.3) παίρνουμε $-\frac{j_1}{1} \le l \le \frac{j_2}{2}$. Υπενθυμίζουμε ότι (j_1, j_2) είναι μια λύση της εξίσωσης $2x_1 + x_2 = 9$. Από το προηγούμενο παράδειγμα γνωρίζουμε ότι $x_1 = 9 - v$ και $x_2 = 2v - 9$, όπου $v \in \mathbb{Z}$. Μπορούμε να χρησιμοποιήσουμε την τιμή $v = 1$ και επομένως να πάρουμε τη λύση $j_1 = 8$ και $j_2 = -7$. Αντικαθιστώντας τις συγκεκριμένες αυτές τιμές για τα j_1 και j_2 καταλήγουμε στο $-8 \le l \le \frac{-7}{2} \Rightarrow -8 \le l \le -3.5 \Rightarrow l = -8$ ή $l = -7$ ή $l = -6$ ή $l = -5$ ή $l = -4$. Αυτό σημαίνει ότι υπάρχουν 5 σημεία του χώρου δεικτών που βρίσκονται επί του υπερ-επιπέδου $2x_1 + 5x_2 = 21$. Χρησιμοποιώντας την εξίσωση (3.2) βρίσκουμε ότι το ελάχιστο σημείο, που αντιστοιχεί στην ελάχιστη τιμή του $l = -8$, είναι το (0, 9), και το μέγιστο σημείο, που αντιστοιχεί στη μέγιστη τιμή του $l = -4$, είναι το (4, 1).

Τέλος, βρίσκουμε τον αριθμό των μη αρνητικών σημείων του χώρου δεικτών της ευθείας $2x_1 + 5x_2 = 21$ (Σχήμα 3.4(b)). Από την ανισότητα (3.3) έχουμε $-\frac{j_1}{5} \le l \le \frac{j_2}{2}$. Υπενθυμίζουμε ότι το σημείο (j_1, j_2) είναι μια λύση της εξίσωσης $2x_1 + 5x_2 = 21$. Από το προηγούμενο παράδειγμα γνωρίζουμε ότι $x_1 = 63 - 5v$ και $x_2 = 2v - 21$, όπου $v \in \mathbb{Z}$. Μπορούμε να χρησιμοποιήσουμε την τιμή $v = 1$ και κατ' αυτόν τον τρόπο να λάβουμε $j_1 = 58$ και $j_2 = -19$. Αντικαθιστώντας τις συγκεκριμένες αυτές τιμές για τα j_1 και j_2 καταλήγουμε στο ότι $-\frac{58}{5} \le l \le \frac{-19}{2} \Rightarrow -11.6 \le l \le -9.5 \Rightarrow l = -11$ ή $l = -10$. Αυτό σημαίνει ότι υπάρχουν δυο μη αρνητικά σημεία του χώρου δεικτών που κείτονται επί της ευθείας $2x_1 + 5x_2 = 21$. Χρησιμοποιώντας την εξίσωση (3.2) βρίσκουμε ότι το ελάχιστο σημείο, που αντιστοιχεί στην ελάχιστη τιμή του $l = -11$, είναι το (3, 3) και αντίστοιχα το μέγιστο σημείο, που αντιστοιχεί στη μέγιστη τιμή του $l = -10$, είναι το (8, 1).

Εισάγουμε τώρα δυο νέες έννοιες, την έννοια του *διαδόχου* (*successor*) και την έννοια του *προκατόχου* (*predecessor*) ενός σημείου του χώρου δεικτών.

Ορισμός 3.7: Ας υποθέσουμε ότι τα **i** και **j** είναι δυο σημεία του χώρου δεικτών που ανήκουν στο ίδιο υπερ-επίπεδο Π_k. Λέμε ότι το σημείο **j** είναι ο διάδοχος του σημείου **i** (ισοδύναμα, ότι το σημείο **i** είναι ο προκάτοχος του σημείου **j**) σύμφωνα με τη λεξικογραφική διάταξη, αν ισχύει **i < j** και δεν υπάρχει κανένα άλλο σημείο **j'** του ίδιου υπερ-επιπέδου τέτοιο ώστε **i < j' < j**, δηλαδή δεν υπάρχει κανένα άλλο σημείου του ίδιου υπερ-επιπέδου που να παρεμβάλλεται μεταξύ των σημείων **i** και **j**.

<u>Σημείωση</u>: Το μέγιστο σημείο ενός υπερ-επιπέδου εξ ορισμού δεν έχει διάδοχο (διαφορετικά δεν θα ήταν το μέγιστο σημείο).

Ο υπολογισμός όλων των ελαχίστων και των μεγίστων σημείων στο εργαλείο CRONUS εκτελείται κατά τη διάρκεια του βήματος 1 (προ-επεξεργασία), δηλαδή, όταν θα υπολογιστεί το κυρτό περίγραμμα των διανυσμάτων εξάρτησης. Είναι λογικό να εκτελούνται οι υπολογισμοί αυτοί πριν την εκτέλεση του προγράμματος (*offline*) κατά το χρόνο μεταγλώττισης (*compilation time*), καθώς αφού υπολογισθούν όλα τα ελάχιστα και μέγιστα σημεία, αποθηκεύονται σε πίνακες οι οποίοι αντιγράφονται σε όλες τις παράλληλες διεργασίες, ώστε να χρησιμοποιούνται και να γίνεται γρήγορη ανάκτηση από αυτούς κατά τη διαδικασία προσδιορισμού του διαδόχου και του προκατόχου κάθε σημείου στο χρόνο εκτέλεσης του προγράμματος (*runtime*).

Η ανεύρεση του διαδόχου ενός σημείου του χώρου δεικτών είναι μια απλή διαδικασία αν τα ελάχιστα σημεία των υπερ-επιπέδων είναι ήδη γνωστά. Ο ψευδοκώδικας του αλγορίθμου για τον υπολογισμό του διαδόχου ενός σημείου στη γενική περίπτωση των n-διαστάσεων παρατίθεται στο παράρτημα (Σχήμα Β.2). Ειδικά στην περίπτωση δισδιάστατων βρόχων (2D), η διαδικασία υπολογισμού του διαδόχου ενός σημείου είναι τετριμμένη. Ας υποθέσουμε ότι τα σημεία i και j ανήκουν στο ίδιο υπερ-επίπεδο $\Pi_k(a_1, a_2)$. Στην περίπτωση αυτή τα i και j μπορούν να προκύψουν από την εξίσωση (3.2) για διαφορετικές τιμές του l. Υποθέτοντας ότι l_i και l_j είναι οι τιμές που αντιστοιχούν στα i και j, συμπεραίνουμε αμέσως από την εξίσωση (3.2) ότι:

$$l_i < l_j \Leftrightarrow \mathbf{i < j} \qquad (3.4)$$

Ως εκ τούτου, το σημείο j είναι ο διάδοχος του σημείου i, αν και μόνο αν $l_j = l_i + 1$.

Παράδειγμα 3.6: Στο προηγούμενο παράδειγμα, έχουμε διαπιστώσει ότι το υπερ-επίπεδο $\Pi_9(2, 1)$ περιέχει 5 σημεία του χώρου δεικτών που βρίσκονται επί του υπερ-επίπεδου $2x_1 + 5x_2 = 21$. Τα σημεία αυτά μπορούν να υπολογιστούν από την εξίσωση (3.2) αντικαθιστώντας το $j_1 = 8$ και $j_2 = -7$ και υποχρεώνοντας τη σταθερά l να λάβει τις τιμές –8, –7, –6, –5, –4. Το σημείο του χώρου δεικτών (2, 5) αντιστοιχεί στην τιμή l = –6. Ως εκ τούτου, ο διάδοχος του σημείου (2, 5) είναι το σημείο του χώρου δεικτών που αντιστοιχεί στην τιμή l = –5. Η εξίσωση (3.2) με l = –5 δίνει το σημείο του χώρου δεικτών (3, 3), το οποίο είναι πράγματι ο διάδοχος του σημείου (2, 5).

Ορισμός 3.8: Δεδομένου ενός σημείου **i** του χώρου δεικτών που ανήκει στο υπερ-επίπεδο Π_k, ορίζουμε ως **successor(i)** το διάδοχό του, εάν αυτός υπάρχει, ή το ελάχιστο σημείο του υπερ-επιπέδου Π_{k+1}, εάν ο διάδοχός του δεν υπάρχει.

3.5 Διαδοχική Δυναμική Δρομολόγηση (SDS)

Το σχήμα δρομολόγησης που ακολουθούμε βασίζεται στην έννοια του διαδόχου. Χρησιμοποιώντας το διάδοχο ο χώρος δεικτών μπορεί να διασχίζεται υπερ-επίπεδο προς υπερ-επίπεδο, διασχίζοντας κάθε υπερ-επίπεδο σύμφωνα με τη λεξικογραφική του διάταξη, αποδίδοντας τελικά ένα μονοπάτι σε σχήμα ζιγκ-ζαγκ ή σπειροειδές γενικότερα. Αυτό διασφαλίζει την *εγκυρότητα* και το *βέλτιστο* της απόδοσής του. Η πρώτη συνθήκη ικανοποιείται επειδή τα σημεία του τρέχοντος υπερ-επίπεδου εξαρτώνται μόνον από σημεία προηγουμένων υπερ-επιπέδων και η δεύτερη γιατί ακολουθώντας το βέλτιστο υπερ-επίπεδο αξιοποιείται στο μέγιστο όλος ο εγγενής παραλληλισμός.

Προτείνουμε τη χρήση της συνάρτησης του διαδόχου ώστε να επαναλαμβάνεται κατά μήκος του βέλτιστου υπερ-επιπέδου ως ένας προσαρμοστικός δυναμικός κανόνας. Για το σκοπό αυτό ορίζεται η ακόλουθη συνάρτηση:

$$\text{Succ}(\mathbf{j}, r) = \begin{cases} \mathbf{j}, & \text{if } r = 0; \ /* \text{ the same point j } */ \\ \mathbf{successor}(\mathbf{j}), & \text{if } r = 1; \ /* \text{ the immediate successor of point j } */ \\ \mathbf{successor}(\text{Succ}(\mathbf{j}, r-1)), & \text{if } r > 1; \ /* \text{ the r-th successor of point j } */ \end{cases}$$

Η παραπάνω συνάρτηση παρέχει τα εξής:

- τον κανόνα με το όποιο κάθε επεξεργαστής αποφασίζει ποια επανάληψη να εκτελέσει στο επόμενο βήμα, και
- έναν αποτελεσματικό αλγόριθμο που επιτρέπει σε κάθε επεξεργαστή να προσδιορίζει από ποιον ή ποιους επεξεργαστές να λάβει τα ήδη υπολογισμένα δεδομένα τους και σε ποιον ή ποιους επεξεργαστές να στείλει δεδομένα που έχουν ήδη υπολογιστεί τοπικά σε αυτόν.

Η χρήση της έννοιας του διαδόχου είναι αποτελεσματική διότι επάγει μια αμελητέα επιβάρυνση στην εκτέλεση του παράλληλου προγράμματος και δεν επιφέρει κανένα επιπλέον κόστος επικοινωνίας στις πλατφόρμες κατανεμημένης μνήμης.

Η αποκεντρωμένη πολιτική δρομολόγησης που ακολουθείται λέει πως εάν υποτεθεί ότι υπάρχουν NP διαθέσιμοι επεξεργαστές (P_1, …, P_{NP}), ο P_1 εκτελεί το αρχικό σημείο του χώρου δεικτών L, ο P_2 εκτελεί το σημείο που αντιστοιχεί στο Succ(L, 1), ο P_3 εκτελεί το σημείο που αντιστοιχεί στο Succ(L, 2), και ούτω καθεξής, έως ότου όλοι οι επεξεργαστές εμπλακούν στη συνολική εκτέλεση του προγράμματος για πρώτη φορά. Με την ολοκλήρωση του αρχικού σημείου L, ο επεξεργαστής P_1 υπολογίζει το επόμενο εκτελέσιμο σημείο, το οποίο προσδιορίζει υπερπηδώντας NP σημεία του υπερ-επιπέδου. Οι συντεταγμένες του σημείου αυτού προκύπτουν από την εφαρμογή της λειτουργίας Succ NP φόρες επί του σημείου που εκτελείται επί του παρόντος από τον επεξεργαστή P_1, δηλαδή με την κλήση *Succ(L, NP)*. Ομοίως, μετά την ολοκλήρωση υπολογισμού του τρέχοντος σημείου j, ο επεξεργαστής P_2 εκτελεί το σημείο *Succ(j, NP)* και ούτω καθεξής, έως ότου εξαντληθούν όλα τα σημεία του χώρου δεικτών. Ο αλγόριθμος δυναμικής δρομολόγησης SDS τελειώνει όταν εκτελεστεί (από κάποιους επεξεργαστές προσπεραστεί) και το τερματικό σημείο U.

Το σχήμα επικοινωνίας που χρησιμοποιείται από την πλατφόρμα CRONUS είναι επίσης ικανό για το δυναμικό προσδιορισμό κατά το χρόνο εκτέλεσης του επεξεργαστή ή των επεξεργαστών από τους οποίους πρέπει να λάβει δεδομένα αλλά και αυτόν ή αυτούς στους οποίους πρέπει να στείλει δεδομένα που ήδη υπολόγισε τοπικά. Κάθε επεξεργαστής λαμβάνει τα απαραίτητα δεδομένα από τους άλλους επεξεργαστές πριν από την έναρξη υπολογισμού της τρέχουσας επανάληψης του βρόχου και στείλει στη συνεχεία τα απαραίτητα δεδομένα στους άλλους επεξεργαστές πριν προχωρήσει στο επόμενο βήμα. Με άλλα λόγια, κάθε επεξεργαστής εκτελεί τα ακόλουθα βήματα:

(1) Λήψη δεδομένων

(2) Υπολογισμός τρέχουσας επανάληψης

(3) Αποστολή δεδομένων

(4) Προσδιορισμός επομένης επανάληψης

Για να εξηγήσουμε πως αυτό λειτουργεί στην πλατφόρμα CRONUS, ας υποθέσουμε ότι στον επεξεργαστή P_i έχει ανατεθεί η επανάληψη j. Ο επεξεργαστής P_i πρέπει να ακολουθήσει τα παρακάτω βήματα:

(1) Προσδιορισμός των επαναλήψεων από τις οποίες εξαρτάται το σημείο j σύμφωνα με τα διανύσματα εξάρτησης.

(2) Ανεύρεση των επεξεργαστών οι οποίοι εκτέλεσαν αυτές τις επαναλήψεις· στην πλατφόρμα παράλληλου προγραμματισμού MPI αυτό σημαίνει να βρεθεί ο κατατακτήριος αριθμός τους (rank).

(3) Εκτέλεση των απαραίτητων κλήσεων λήψης δεδομένων του MPI, χρησιμοποιώντας αυτούς τους κατατακτήριους αριθμούς.

(4) Μετά την παραλαβή των δεδομένων, υπολογισμός της επανάληψης j.

(5) Με την ολοκλήρωση του υπολογισμού, καθορισμός των επαναλήψεων που εξαρτώνται από το j.

(6) Προσδιορισμός των κατατακτήριων αριθμών των επεξεργαστών στους οποίους έχουν εκχωρηθεί οι επαναλήψεις αυτές.

(7) Εκτέλεση των κατάλληλων κλήσεων αποστολής δεδομένων του MPI στους αντίστοιχους με τους κατατακτήριους αυτούς αριθμούς επεξεργαστές.

(8) Προσδιορισμός της επόμενης προς εκτέλεση επανάληψης, καλώντας τη συνάρτηση Succ με παραμέτρους τα j και NP.

Προκειμένου να βρεθεί ο κατατακτήριος αριθμός του επεξεργαστή από τον οποίο κάποιος πρέπει να λάβει δεδομένα, ή ο κατατακτήριος αριθμός του επεξεργαστή στον οποίο κάποιος πρέπει να στείλει δεδομένα, χρησιμοποιούμε την απλή τεχνική υπολογισμού του διάδοχου σημείου της προηγούμενης ενότητας όταν ασχολούμαστε

με περιπτώσεις στις δύο διαστάσεις (2D), ή το γενικό αλγόριθμο εύρεσης του διάδοχου σημείου ο οποίος παρουσιάζεται στο παράρτημα, για περιπτώσεις μεγαλυτέρων διαστάσεων.

Όπως αναφέρθηκε και προηγουμένως, η χρήση ενός μοντέλου παραλληλοποίησης λεπτού κόκκου (*fine grain parallelization*) προϋποθέτει ότι το σώμα του βρόχου είναι απαιτητικό υπολογιστικά, διαφορετικά το κέρδος απόδοσης ακυρώνεται. Ειδικότερα, όταν τέτοια μοντέλα εφαρμόζονται σε αρχιτεκτονικές ανταλλαγής μηνυμάτων, το κόστος που υπεισέρχεται από τις ρουτίνες επικοινωνίας θα πρέπει να είναι μικρότερο από το κόστος υπολογισμού των αντιστοιχών επαναλήψεων. Ως εκ τούτου, η εισαγωγή κλήσεων της πλατφόρμας MPI σε κάθε επανάληψη του βρόχου δεν είναι απαγορευτική για όσο διάστημα το κόστος υπολογισμού του σώματος του βρόχου ξεπερνά το κόστος αυτό. Στο κεφάλαιο 5 που υπάρχει η αξιολόγηση των επιδόσεων του εργαλείου CRONUS, παρουσιάζεται το ποσοστό του χρόνου επικοινωνίας έναντι του χρόνου υπολογισμού για τις περιπτώσεις των δοκιμών, καθώς και το ποσοστό της επιβάρυνσης του αλγορίθμου δρομολόγησης επί του συνολικού χρόνου επικοινωνίας (Σχήμα 5.7, Σχήμα 5.10, Σχήμα 5.13). Όπως καταδεικνύεται και από τα αντίστοιχα σχήματα, η επιβάρυνση του αλγορίθμου δρομολόγησης είναι αποδεκτή, καθώς κυμαίνεται από 7% έως 30% του συνολικά μετρούμενου χρόνου επικοινωνίας. Μια άλλη πτυχή της προσέγγισής μας είναι η αποκέντρωση και ο καταμερισμός του κόστους δρομολόγησης. Αυτό σημαίνει ότι κάθε επεξεργαστής είναι υπεύθυνος μόνο για τη λήψη των απαραίτητων αποφάσεων δρομολόγησης που τον αφορούν. Η αποκεντρωμένη προσέγγιση δρομολόγησης είναι καλύτερα προσαρμοσμένη σε σχήματα παραλληλισμού λεπτού κόκκου σε σχέση με οποιαδήποτε άλλη συγκεντρωτική/κεντρική. Αυτό ισχύει καθώς αν το έργο της δρομολόγησης είχε παραμείνει σε έναν και μόνο κύριο επεξεργαστή-συντονιστή, ο οποίος θα έπρεπε να επικοινωνεί με πολλαπλούς περιφερειακούς επεξεργαστές-δούλους μετά από κάθε επανάληψη, η επιβάρυνση επικοινωνίας που θα προέκυπτε θα ήταν υπερβολική.

Η προηγούμενη συζήτηση σκιαγραφεί τα βασικά χαρακτηριστικά του αλγορίθμου Διαδοχικής Δυναμικής Δρομολόγησης (SDS), που είναι:

- *Ένας βαθμός δυναμικότητας*, υπό την έννοια ότι χρονική δρομολόγηση και υπολογισμός επαναλήψεων επιτελούνται μόνο κατά το χρόνο εκτέλεσης.

- *Κατανεμημένη συμπεριφορά*, καθώς η δρομολόγηση εργασιών και η συνολική πληροφορία δρομολόγησης είναι κατανεμημένη μεταξύ των επεξεργαστών και των επιμέρους τοπικών τους μνημών.
- *Αυτόνομη καθοδήγηση*, επειδή κάθε επεξεργαστής καθορίζει την επόμενη επανάληψη του μόνος του, με κατάλληλη αύξηση των δεικτών του βρόχου.
- *Ιδανική εξισορρόπηση φορτίου*, γιατί αναθέτει επαναλήψεις στους επεξεργαστές με βάση ένα επαναληπτικό κυκλικό μοντέλο ανάθεσης (round-robin).

3.6 Αρχιτεκτονικές Προορισμού

Το εργαλείο CRONUS δημιουργεί παράλληλο κώδικα που περιέχει κλήσεις του MPI. Ο κώδικας αυτός μπορεί να χρησιμοποιηθεί σε συστήματα μοιραζόμενης μνήμης, σε συστήματα κατανεμημένης μνήμης, συστοιχίες σταθμών εργασίας, SMPs και MPPs. Αυτό επιφέρει πολύ υψηλή φορητότητα για το CRONUS. Το κόστος επικοινωνίας αποτελεί μείζον ζήτημα που επηρεάζει τη συνολική απόδοση των παράλληλων προγραμμάτων, ενώ το μέγεθος της επικοινωνίας αυξάνει γενικά με τον αριθμό των επεξεργαστών. Με αυτά τα δεδομένα η πλατφόρμα CRONUS αναμένεται να αποδίδει καλύτερα σε συστήματα μοιραζόμενης μνήμης, λόγω του μικρότερου ή και μηδενικού σε πολλές περιπτώσεις κόστους επικοινωνίας.

3.7 Αναφορές

[3.1] L. Lamport, "The Parallel Execution of DO Loops", *Communication of the ACM, 37(2):83–93, Feb. 1974*.

[3.2] A. Darte, L. Khachiyan, and Y. Robert, "Linear scheduling is nearly optimal", *Par. Proc. Letters, 1.2:73–81, 1991*.

[3.3] D. I. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays", *IEEE Transactions on Computers, C-35(1):1–11, 1986*.

[3.4] W. Shang and J.A.B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies", *IEEE Transactions on Computers, 40(6):723–742, 1991*.

[3.5] G. Papakonstantinou, T. Andronikos, and I. Drositis, "On the parallelization of UET/UET-UCT loops", *NPSC J. on Computing, 2001*.

[3.6] F. Irigoin and R. Triolet, "Supernode partitioning", *In Proceedings of the 15th Annual ACM SIG ACT-SIGPLAN Symposium on Principles of Programming Languages, pages 319–329, January 1988*.

[3.7] Jingling Xue, "On tiling as a loop transformation", *Parallel Processing Letters, 7(4):409–424, 1997*.

[3.8] Jingling Xue, "Loop Tiling for Parallelism", *Kluwer Academic Publishers, 2000*.

[3.9] N. Goumas, G. Drosinos, M. Athanasaki, and N. Koziris, "Compiling Tiled Iteration Spaces for Clusters", *In Proceedings of the 2002 IEEE International Conference on Cluster Computing, pages 360–369, Chicago, Illinois, Sep 2002*.

[3.10] F.-M. Ciorba, T. Andronikos, D. Kamenopoulos, P. Theodoropoulos, and G. Papakonstantinou, "Simple code generation for special UDLs", *In 1st Balkan Conference in Informatics (BCI'03), November 2003*.

[3.11] T. Andronikos, F.-M. Ciorba, P. Theodoropoulos, Kamenopoulos D., and G. Papakonstantinou, "Code Generation For General Loops Using Methods From", *In IASTED Parallel and Distributed Computing and Systems Conference (PDCS'04), 2004*.

[3.12] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa, "The Quickhull Algorithm for Convex Hulls", *ACM Transactions on Mathematical Software, 22(4):469–483, 1996*.

[3.13] LBNL and UC Berkeley, Berkeley Unified Parallel C.

[3.14] D. I. Moldovan, "Parallel Processing: From Applications to Systems", *M. Kaufmann, California, 1993*.

[3.15] I. Drositis, T. Andronikos, M. Kalathas, G. Papakonstantinou, and N. Koziris, "Optimal loop parallelization in n-dimensional index spaces", *In Proc. of the 2002 Int'l Conf. on Par. and Dist. Proc. Techn. and Appl. (PDPTA'02), 2002*.

[3.16] I. Niven, H. Zuckerman, and H. Montgomery, "An Introduction to the Theory of Numbers", *John Wiley & Sons, 5th edition, 1991*.

[3.17] J. Silverman, "A Friendly Introduction to Number Theory", *Prentice Hall, 2nd edition, 2001*.

[3.18] MPI Forum, "Message Passing Interface Standard", *http://www-unix.mcs.anl.gov/mpi/indexold.html, 2002*.

[3.19] R.W. Floyd and L. Steinberg, "An adaptive algorithm for spatial grey scale", *In Proc. Soc. Inf. Display, volume 17, pages 75–77, 1976*.

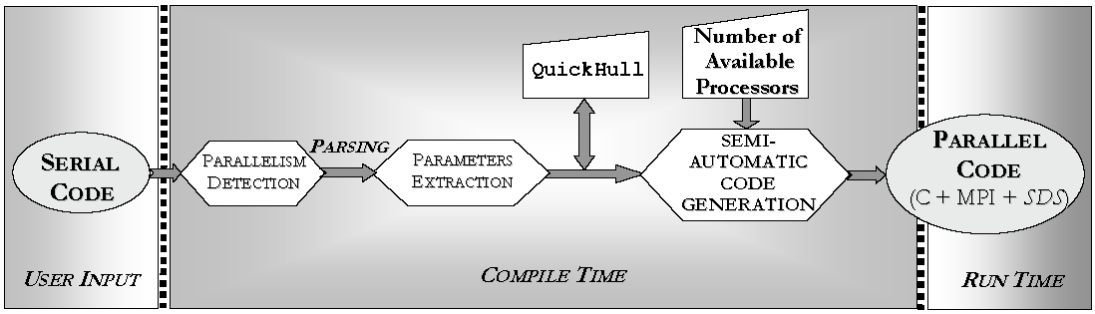
[3.20] H. Yee and Yu Hen Hu, "A novel modular systolic array architecture for full-search block matching motion estimation", *IEEE Transactions on Circuits and Systems for Video Technology, 5(5):407–416, October 1995*.

ΚΕΦΑΛΑΙΟ 4. ΛΕΙΤΟΥΡΓΙΑ ΚΑΙ ΧΡΗΣΗ CRONUS

Το CRONUS είναι μια υλοποιημένη λειτουργική ημιαυτόματη πλατφόρμα παραλληλοποίησης κώδικα. Μπορούμε να πούμε ότι ανήκει στην κατηγορία των παράλληλων μετα-μεταγλωττιστών (*metacompiler*) και χρησιμοποιεί γεωμετρικές τεχνικές για τη δρομολόγηση βρόχων με ομοιόμορφες εξαρτήσεις (UDLs). Είναι δομημένο σε υποσυστήματα (*modules*) που χειρίζονται αντίστοιχα στάδια λειτουργίας του (βλέπε Σχήμα 4.1). Έτσι, στο πρώτο στάδιο (Είσοδος Χρήστη – *USER INPUT*) ο χρήστης καταχωρεί ένα ακολουθιακό πρόγραμμα. Το επόμενο στάδιο (Χρόνος Μεταγλώττισης - *COMPILE TIME*) αποτελείται από τρεις φάσεις, με πρώτη τη φάση ανίχνευσης της φωλιάς των βρόχων (Ανίχνευση Παραλληλισμού – *PARALLELISM DETECTION*). Εάν δεν μπορεί να βρεθεί καμιά φωλιά βρόχου στο ακολουθιακό πρόγραμμα, το εργαλείο CRONUS σταματά. Καθόλη τη διάρκεια της δεύτερης φάσης (Εξαγωγή Παραμέτρων – *PARAMETERS EXTRACTION*), το πρόγραμμα αναλύεται και εξάγονται οι ακόλουθες βασικές παράμετροι: το βάθος της φωλιάς του βρόχου (n), το μέγεθος του χώρου δεικτών (|J|) και το σύνολο των διανυσμάτων εξάρτησης (DS). Μόλις οι απαραίτητες παράμετροι είναι διαθέσιμες, το πρόγραμμα καλεί τον αλγόριθμο κυρτού περιγράμματος QuickHull, ο οποίος επιστρέφει το βέλτιστο υπερ-επίπεδο. Σε αυτό το σημείο, ο διαθέσιμος αριθμός επεξεργαστών (NP) είναι απαραίτητος ως είσοδος. Μια ημιαυτόματη γεννήτρια κώδικα παράγει τον κατάλληλο παράλληλο κώδικα για το δεδομένο αριθμό επεξεργαστών NP στην τελευταία φάση (Ημιαυτόματη Παραγωγή Κώδικα – *SEMI-AUTOMATIC CODE GENERATION*). Αυτό επιτυγχάνεται με τη βοήθεια ενός προγράμματος σε γλώσσα Perl που λειτουργεί με βάση ένα αρχείο ρυθμίσεων, το οποίο και περιέχει όλες τις απαιτούμενες πληροφορίες.

Στο αρχείο ρυθμίσεων, ο χρήστης πρέπει επίσης να ορίσει μια συνάρτηση αρχικοποίησης γραμμένη σε γλώσσα C, η οποία καλείται αυτόματα από τον δημιουργούμενο παράλληλο κώδικα, για να επιτελέσει την αρχικοποίηση των δεδομένων σε κάθε επεξεργαστή. Ο παράλληλος κώδικας είναι γραμμένος σε γλώσσα C και περιέχει ρουτίνες για την εκτέλεση του αλγορίθμου Διαδοχικής Δυναμικής

Δρομολόγησης SDS και ρουτίνες MPI για την επικοινωνία δεδομένων μέσω ανταλλαγής μηνυμάτων. Ο τελικός κώδικας είναι κατάλληλος για την άμεση μεταγλώττιση και παράλληλη εκτέλεση στο διαθέσιμο πολύ-επεξεργαστικό υπολογιστικό σύστημα (στάδιο Χρόνου Εκτέλεσης – *RUN TIME*).



Σχήμα 4.1: Εσωτερική οργάνωση του CRONUS

4.1 Οργάνωση του CRONUS

Το εργαλείο CRONUS αποτελείται από:

- Μια δυναμική βιβλιοθήκη χρόνου εκτέλεσης για τη γλώσσα προγραμματισμού C, η οποία περιέχει τις ρουτίνες για το συγχρονισμό και τη δυναμική δρομολόγηση των παραλλήλων διεργασιών.
- Ένα σύνολο από Perl scripts που παράγουν, μεταγλωττίζουν και εκτελούν την ακολουθιακή και την παράλληλη έκδοση του αρχικού ακολουθιακού αλγορίθμου, διασυνδέοντας μια κατάλληλα τροποποιημένη (και δυναμικά δημιουργούμενη) έκδοση της βιβλιοθήκης με τις συναρτήσεις σε C.

Για τη σωστή λειτουργία του CRONUS οι ελάχιστες απαιτήσεις σε άλλα εγκατεστημένα προγράμματα και γενικό λογισμικό είναι:

- Μια υλοποίηση του MPI συμβατή με το πρότυπο MPI 1.1, όπως η mpich
- Μεταφραστής της γλώσσας Perl, έκδοσης 5.x ή μεταγενέστερος
- Ένας μεταγλωττιστής της γλώσσας C, συμβατός με το πρότυπο ANSI-C (C89), όπως ο gcc 2.95.x ή μεταγενέστερος
- Μερικά διαδεδομένα UNIX utilities (tar, gzip, κλπ.), τα οποία είναι συνήθως διαθέσιμα σε κάθε κλώνο του λειτουργικού συστήματος UNIX.

4.2 Λειτουργία του CRONUS

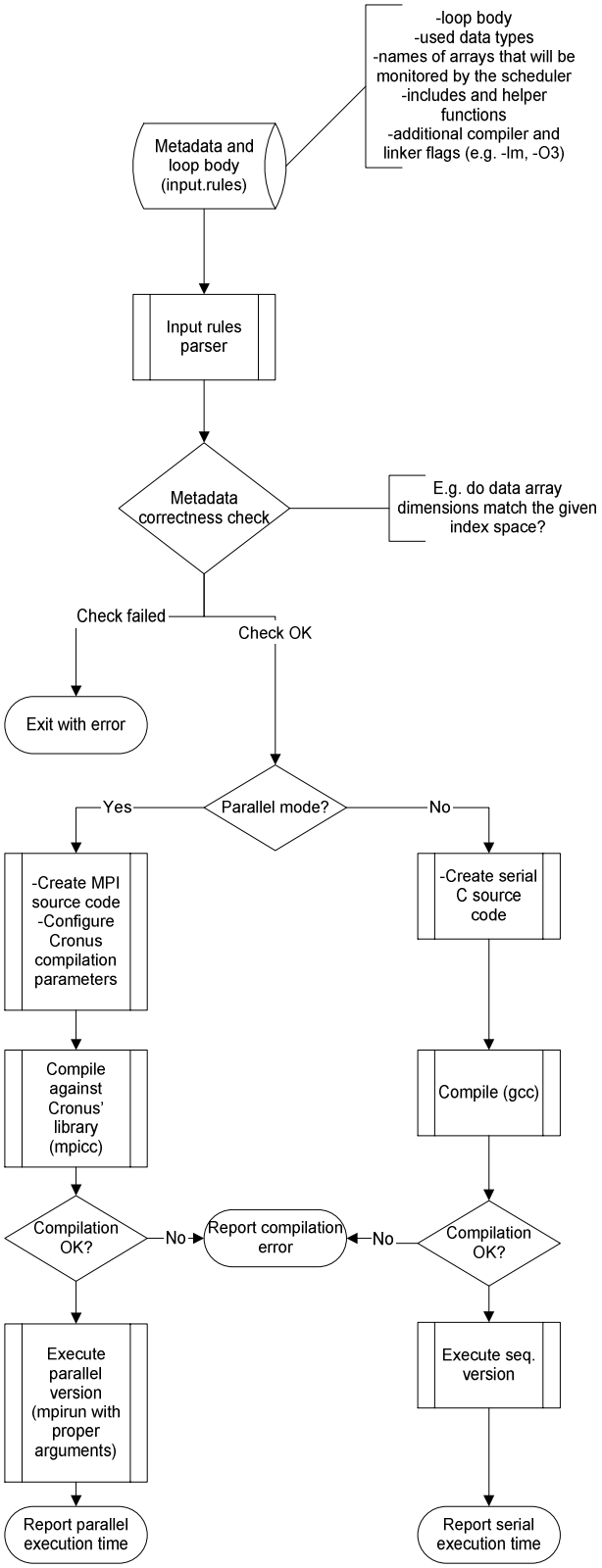
Το CRONUS διαβάζει την είσοδο του από ένα αρχείο κειμένου που φέρει το όνομα "input.rules". Το αρχείο αυτό είναι δομημένο σε ενότητες που περιλαμβάνουν τα παρακάτω δεδομένα:

- Το σώμα του βρόχου που θέλουμε να παραλληλοποιήσουμε (γραμμένο σε γλώσσα προγραμματισμού C).
- Τα ονόματα και τους τύπους δεδομένων των πινάκων εντός του βρόχου τα περιεχόμενα των οποίων πρέπει να συγχρονίζονται μεταξύ των επεξεργαστών.
- Τα ονόματα των μεταβλητών του βρόχου (συνήθως i, j, k, …).
- Ο επιθυμητός αριθμός και διαστάσεις των μπλοκ.
- Βοηθητικές λειτουργίες, όπως π.χ. συναρτήσεις αρχικοποίησης δεδομένων.
- Επικουρικά αρχεία ορισμών (*include files*) και ρυθμίσεις επιλογών για το μεταγλωττιστή/συνδέτη.

Ένα υποσύστημα ανάλυσης (*parsing module*) μετατρέπει όλα αυτά τα δεδομένα σε αναλυμένες πληροφορίες και στη συνέχεια τα περνά στο υποσύστημα εξαγωγής παραμέτρων (*parameter extraction module*), το οποίο αυτόματα προσδιορίζει και κάποια ακόμα απαραίτητα μεταδεδομένα, όπως π.χ. τα διανύσματα εξάρτησης δοθέντος του σώματος του βρόχου και τα ονόματα των μεταβλητών του βρόχου. Το υποσύστημα εξαγωγής παραμέτρων διενεργεί επίσης και κάποιους βασικούς ελέγχους λαθών των μεταδεδομένων. Για παράδειγμα, αν ο χρήστης προσδιορίζει στο αρχείο "input.rules" ότι επιθυμεί ένα μπλοκ μεγέθους 2x2x2, ωστόσο ο χώρος δεικτών φαίνεται να είναι δυο διαστάσεων, θα αναφερθεί ένα μήνυμα λάθους στο χρήστη και το εργαλείο θα τερματίσει την εκτέλεσή του.

Στη συνέχεια, οι αναλυμένες πληροφορίες και τα μεταδεδομένα διαβιβάζονται στο υποσύστημα παραγωγής του τελικού κώδικα. Δυο ανεξάρτητα υποσυστήματα έχουν υλοποιηθεί. Το ένα από αυτά δημιουργεί μια σειριακή έκδοση του προγράμματος, ενώ το άλλο μια παράλληλη έκδοση που βασίζεται στην πλατφόρμα παράλληλου προγραμματισμού MPI. Ανεξάρτητα από το ποιο είναι το υποσύστημα που τελικά χρησιμοποιείται, ο κώδικας καταρτίζεται, μεταγλωττίζεται και εκτελείται αυτόματα σε κάθε περίπτωση. Τέλος, το εργαλείο CRONUS παραθέτει το χρόνο εκτέλεσης του τελικού εκτελέσιμου προγράμματος, σειριακού ή παράλληλου.

Το μοντέλο λειτουργίας του CRONUS περιγράφεται σχηματικά στο διάγραμμα ροής του σχήματος 4.2.



Σχήμα 4.2: Διάγραμμα ροής λειτουργίας CRONUS

4.3 Χρήση του CRONUS

Γενικά το εργαλείο CRONUS είναι απλό στη χρήση. Η χρήση του γίνεται μέσω δύο βασικών προγραμμάτων, των `make_serial.pl` και `make_parallel.pl`. Αυτά τα προγράμματα καλούν τον ακολουθιακό και τον παράλληλο μετα-μεταγλωττιστή του CRONUS αντίστοιχα.

Καθώς το εργαλείο CRONUS δημιουργεί πολλά αρχεία σε κάθε εκτέλεσή του, κάθε παράδειγμα κώδικα που θέλουμε να παραλληλοποιήσουμε θα πρέπει να τοποθετείται σε ένα νέο κατάλογο. Το αρχείο "input.rules" που αναφέρεται στην προηγούμενη ενότητα, πρέπει να τοποθετηθεί σε αυτόν τον κατάλογο. Το εργαλείο CRONUS θα εκτελεστεί και θα τοποθετήσει τα αρχεία που δημιουργεί αποκλειστικά στον κατάλογο αυτό. Αν και δεν είναι απολύτως απαραίτητο, η παράθεση κάθε άλλου αρχείου σχετικού με το υπό εξέταση παράδειγμα (π.χ., ένα αρχείο που περιέχει δεδομένα εισόδου) στο συγκεκριμένο κατάλογο, συνιστάται ιδιαίτερα.

Στη συνέχεια δίνεται ένας συνοπτικός οδηγός χρήσης του εργαλείου CRONUS. Τα βήματα που ακολουθούμε για τη διεξαγωγή ενός πειράματος μπορούν να συνοψισθούν ως εξής:

(1) Δημιουργούμε έναν νέο υποκατάλογο για το υπό μελέτη παράδειγμα
(2) Γράφουμε στο νέο υποκατάλογο το βασικό αρχείο οδηγιών "input.rules"
(3) Εκτελούμε το πρόγραμμα `make_serial.pl` με παράμετρο το όνομα του υποκαταλόγου για να δημιουργηθεί και να εκτελεστεί η σειριακή έκδοση του παραδείγματος ή το πρόγραμμα `make_parallel.pl` με παράμετρο το όνομα του υποκαταλόγου για να δημιουργηθεί και να εκτελεστεί η παράλληλη έκδοση του παραδείγματος.

Σαφώς το πιο δύσκολο κομμάτι της διαδικασίας είναι η ορθή συγγραφή του αρχείου οδηγιών "input.rules" (βήμα 2 ανωτέρω). Η επόμενη ενότητα περιγράφει τη σύνταξη του αρχείου "input.rules".

4.4 Σύνταξη αρχείου "input.rules"

Η σύνταξη του αρχείου "input.rules" μοιάζει κάπως με τη σύνταξη των αρχείων αρχικοποίησης εφαρμογών. Ειδικότερα, το αρχείο είναι οργανωμένο σε ενότητες, με

κάθε ενότητα να ξεκινά με μια γραμμή "`.section section_name`". Αυτές οι γραμμές λειτουργούν επίσης ως διαχωριστικά ενοτήτων. Με άλλα λόγια, κάθε ενότητα τελειώνει εκεί όπου αρχίζει η επόμενη ενότητα ή, για την τελευταία, εκεί που τελειώνει το αρχείο "input.rules".

Οι ενότητες που πρέπει να περιλαμβάνονται στο αρχείο για κάθε πείραμα περιγράφονται στη συνέχεια.

4.4.1 .section metadata

Το τμήμα αυτό πρέπει να περιλαμβάνει μια βασική περιγραφή των δεδομένων που το υπό εξέταση πρόγραμμα πρόκειται να χειριστεί. Ένας περιορισμός που επιβάλλεται από το CRONUS είναι ότι τα δεδομένα θα πρέπει να οργανώνονται σε πίνακες του ιδίου τύπου δεδομένων (π.χ. ακεραίων ή αριθμών κινητής υποδιαστολής). Τα μεταδεδομένα που καθορίζονται σε αυτό το τμήμα είναι:

Κλειδί	**Περιγραφή**	**Παράδειγμα**
ElementType	Ο τύπος δεδομένων του πίνακα, π.χ. int εάν τα δεδομένα είναι οργανωμένα σε πίνακες ακεραίων	ElementType int
input_data_array	Το CRONUS χρειάζεται να ξέρει ποιοι πίνακες δεδομένων μοιράζονται μεταξύ των επεξεργαστών. Για καθέναν από αυτούς τους πίνακες, πρέπει να υπάρχει μια γραμμή input_data_array array_name	input_data_array A input_data_array B
output_data_array	Απαρχαιωμένο, λειτουργεί ακριβώς όπως και το input_data_array αλλά για πίνακες δεδομένων αποτελεσμάτων	output_data_array Z

4.4.2 .section block.inf

Στο τμήμα αυτό καθορίζεται ο αριθμός των μπλοκ που θα εκτελούνται ταυτόχρονα από το παράλληλο πρόγραμμα καθώς επίσης και οι διαστάσεις του κάθε μπλοκ. Τα μεταδεδομένα για το τμήμα αυτό είναι τα εξής:

Κλειδί	**Περιγραφή**	**Παράδειγμα**
number of blocks	Αριθμός των μπλοκ που θα εκτελούνται ταυτόχρονα	number of blocks 2
block dimensions	Διαστάσεις μπλοκ (σε σημεία)	block dimensions {4,4} (για ένα μπλοκ διαστάσεων 4x4)

4.4.3 .section loop.vars

Στο τμήμα αυτό καθορίζονται οι μεταβλητές του βρόχου και τα αντίστοιχα όρια του χώρου δεικτών. Η σύνταξη του είναι:

Loop_variable_name:lower_limit..upper_limit

Π.χ. αν ο βρόχος που θέλουμε να παραλληλοποιήσουμε είναι:

```
for (i=0; i<500; i++)
      for (j=0; j<400; j++)
            for (k=0; k<1000; k++) {
                  A[i][j][k] = B[i][j][k-1] + sin(A[i][j][k-2]);
            }
```

τότε η ενότητα .section loop.vars θα πρέπει να γράφεται ως εξής:

```
.section loop.vars
i:0..500
j:0..400
k:0..1000
```

Σημειώνουμε ότι στα πλαίσια της τρέχουσας έκδοσης του CRONUS το κατώτατο όριο θα πρέπει να είναι πάντα το μηδέν (0).

4.4.4 .section loop.body

Στο τμήμα αυτό παραθέτουμε σε γλώσσα C το σώμα του βρόχου που θέλουμε να παραλληλοποιήσουμε. Για παράδειγμα, για την περίπτωση βρόχου με δύο πίνακες τριών διαστάσεων A και B, θα πρέπει να γράψουμε:

```
.section loop.body
A[i][j][k] = B[i][j][k-1] + sin(A[i][j][k-2]);
```

4.4.5 .section data.init

Συνήθως οι πίνακες δεδομένων χρειάζεται να αρχικοποιηθούν πριν αρχίσει να εκτελείται το σώμα του βρόχου (είτε στον παράλληλο κώδικα είτε στον ακολουθιακό). Στην ενότητα αυτή γράφουμε μια συνάρτηση σε γλώσσα C που θα φροντίσει για την κατάλληλη αρχικοποίηση των δεδομένων. Η συνάρτηση αυτή πρέπει να διαθέτει την ακόλουθη μορφή:

```
void data_init(DataType *L, int max_i, int max_j)
```

Όλος ο κώδικας αρχικοποίησης πρέπει να τεθεί στη συνάρτηση data_init(). Το CRONUS θα προσθέσει τον κατάλληλο κώδικα και θα καλέσει αυτόματα τη συνάρτηση data_init() πριν αρχίσει η εκτέλεση του βρόχου. Σημειώνουμε ότι η data_init() θα κληθεί ξεχωριστά για κάθε πίνακα δεδομένων που καθορίζεται στην ενότητα .section metadata, γι' αυτό και χρειάζεται ιδιαίτερη προσοχή στη σύνταξη της συνάρτησης αυτής ώστε να μην έχουμε παρενέργειες και μη επιθυμητά αποτελέσματα.

4.4.6 .section data.exit

Στην ενότητα αυτή παραθέτουμε μια συνάρτηση σε γλώσσα C που θα κληθεί αμέσως μετά την περάτωση και της τελευταίας επανάληψης του βρόχου. Η συνάρτηση πρέπει να έχει την ακόλουθη μορφή:

```
void data_exit(DataType *L, int max_i, int max_j, int i, int j)
```

Ενέργειες που μπορούμε να τοποθετήσουμε στη συνάρτηση data_exit() περιλαμβάνουν κώδικα για ξεκαθάρισμα δεδομένων, εκτύπωση αποτελεσμάτων, κλπ.

4.4.7 .section includes

Σε αυτή την ενότητα γράφουμε (σε ANSI-C) όλες τις δηλώσεις και τους ορισμούς που απαιτούνται ώστε το σώμα του βρόχου να μεταγλωττιστεί και να εκτελεστεί σωστά.

Για παράδειγμα, το ακόλουθο σώμα βρόχου:

```
A[i][j] = B[j-1][j-1] + sin(C[i][j-2]) + myfunction(A[i][j]);
printf("A[%d][%d]=%f", i, j, A[i][j]);
```

απαιτεί στην ενότητα `.section includes` τουλάχιστον τα ακόλουθα:

```
.section includes
#include <stdio.h> /* for printf */
#include <math.h>  /* for sin */

double myfunction(double x) {
      return …;
}
```

Σημείωση: Κάθε βοηθητική μεταβλητή που δεν περιγράφεται στην ενότητα .section metadata ή στην ενότητα .section loop.vars πρέπει να δηλώνεται στην ενότητα .section includes ως παγκόσμια μεταβλητή. Στην ενότητα αυτή πρέπει επίσης να δηλώνονται όλες οι συναρτήσεις που δεν είναι διαθέσιμες σε πρότυπα αρχεία ορισμών (*standard includes*). Το CRONUS προς το παρόν δεν υποστηρίζει διασύνδεση με μη τυποποιημένες βιβλιοθήκες.

4.5 Αντιμετώπιση προβλημάτων

Εάν κάποιος από τους μετα-μεταγλωττιστές ή τα δημιουργηθέντα εκτελέσιμα τερματιστεί με κωδικό σφάλματος, θα πρέπει να ελέγξουμε τα ακόλουθα σημεία:

- Να βεβαιωθούμε ότι έχουμε εισαγάγει σωστά μεταδεδομένα. Για παράδειγμα, όταν έχουμε ένα βρόχο τριών διαστάσεων (δηλαδή τρεις μεταβλητές στην ενότητα .section loop.vars), αλλά δηλώνουμε μπλοκ δύο διαστάσεων στην ενότητα .section block.info, αυτό αποτελεί ένα μοιραίο σφάλμα για το μετα-μεταγλωττιστή.
- Να βεβαιωθούμε ότι η ενότητα .section includes περιλαμβάνει όλους τους απαιτούμενους ορισμούς και δηλώσεις.
- Να βεβαιωθούμε ότι όλες οι σχετικές με τη γλώσσα C ενότητες (π.χ. loop.body, includes, data.init) δεν περιέχουν συντακτικά ή σημασιολογικά λάθη. Σε αυτό βοηθά και ο έλεγχος των αρχείων serial.c και parallel.c που δημιουργούνται στον υποκατάλογο του πειράματος.
- Να αποφεύγουμε την άμεση διόρθωση και αλλαγή των αρχείων serial.c και parallel.c στον κατάλογο του πειράματος, καθώς τα αρχεία αυτά παράγονται αυτόματα εκ νέου κάθε φορά που καλείται κάποιος από τους μετα-μεταγλωττιστές του CRONUS.

- Μπορούμε να αποφύγουμε την εκ νέου κλήση των μετα-μεταγλωττιστών, θεωρώντας ότι δεν έχει αλλάξει το αρχείο οδηγιών "input.rules" και να εκτελέσουμε άμεσα τα αυτόματα δημιουργηθέντα προγράμματα serial και parallel που υπάρχουν στον κατάλογο του πειράματος για το σειριακό και τον παράλληλο κώδικα αντίστοιχα. Σημειώνουμε ότι το parallel είναι ένα παράλληλο πρόγραμμα MPI που πρέπει να εκτελείται μέσω της εντολής mpirun του MPI.

ΚΕΦΑΛΑΙΟ 5. ΕΦΑΡΜΟΓΕΣ ΚΑΙ ΑΠΟΤΕΛΕΣΜΑΤΑ ΔΟΚΙΜΩΝ CRONUS

Το εργαλείο CRONUS απαρτίζεται από τη βιβλιοθήκη χρόνου εκτέλεσης που είναι κωδικοποιημένη στη γλώσσα προγραμματισμού C και συμπληρώνεται από την ημιαυτόματη γεννήτρια κώδικα που είναι γραμμένη σε γλώσσα Perl. Ο παράλληλος κώδικας που παράγεται από το εργαλείο CRONUS χρησιμοποιεί όπου απαιτείται επικοινωνία κλήσεις σύγχρονης αποστολής και λήψης του MPI από σημείο σε σημείο. Όπως περιγράφηκε και στο κεφάλαιο 3, υλοποιεί δύο μηχανισμούς συγχρονισμού διεργασιών, με την πρώτη μέθοδο να κάνει αποκλειστική χρήση σύγχρονων blocking κλήσεων του MPI για συγχρονισμό διεργασιών και ανταλλαγή δεδομένων και με τη δεύτερη να κάνει χρήση του μηχανισμού των καθολικών σηματοφορέων για συγχρονισμό διεργασιών και ασύγχρονων non-blocking κλήσεων του MPI για ανταλλαγή δεδομένων. Καθώς στην παρούσα υλοποίηση του εργαλείου CRONUS δεν είναι δυνατός ο εκ των προτέρων, δηλαδή κατά τη φάση μεταγλώττισης και παραλληλοποίησης ενός προγράμματος, προσδιορισμός της αποδοτικότερης μεθόδου συγχρονισμού, ιδιαίτερα για συστήματα κατανεμημένης μνήμης, για όλα τα πειράματα και τις μετρήσεις που ακολουθούν στο παρόν κεφάλαιο χρησιμοποιήθηκε αποκλειστικά η πρώτη μέθοδος συγχρονισμού. Κατά αυτόν τον τρόπο καθίσταται και άμεσα συγκρίσιμη η απόδοση των ιδίων παραδειγμάτων στις δύο διαφορετικές αρχιτεκτονικές υπολογιστικών συστημάτων που χρησιμοποιήθηκαν για την εκτέλεσή τους.

Τα πειράματα διεξήχθησαν σε δύο διαφορετικές αρχιτεκτονικές:

(α) σε υπολογιστικό σύστημα SGI ORIGIN 2000, το οποίο είναι ένα πολυεπεξεργαστικό σύστημα μοιραζόμενης μνήμης αποτελούμενο από 16 RISC 64-bit επεξεργαστές R10000 ταχύτητας 300MHz, με συνολική κεντρική μνήμη RAM 4GB και με σκληρούς δίσκους συνολικής χωρητικότητας 300 GB οργανωμένους σε ένα Raid-5 CXFS σύστημα, που εκτελεί το λειτουργικό σύστημα IRIX 6.5, και

(β) σε μια συστοιχία (*cluster*) 16 ταυτόσημων προσωπικών υπολογιστών PC, με κάθε κόμβο να διαθέτει έναν επεξεργαστή Intel Pentium-III στα 500MHz, με 256MB κεντρική μνήμη RAM και σκληρό δίσκο 10 GB και εκτελεί το λειτουργικό σύστημα Linux με πυρήνα έκδοσης 2.4.24, διασυνδεδεμένων με τοπικό δίκτυο Fast Ethernet (100BASE-TX).

Και στις δύο αρχιτεκτονικές για την εκτέλεση των πειραμάτων βασιστήκαμε στην έκδοση του MPI γνωστή ως MPICH. Οι χρησιμοποιηθείσες υλοποιήσεις του MPICH ήταν προσαρμοσμένες στις ιδιαιτερότητες της αρχιτεκτονικής του κάθε συστήματος, τόσο στην περίπτωση (α) όπου η επικοινωνία γίνεται μέσω της κοινής μοιραζόμενης μνήμης του συστήματος όσο και στην περίπτωση (β) όπου η επικοινωνία γίνεται πάνω από τη διασύνδεση Fast Ethernet του τοπικού του δικτύου.

Η επιλογή του προτύπου MPI [5.1] ως της πλατφόρμας για την επικοινωνία μεταξύ επεξεργαστών και διεργασιών οφείλεται στο ότι είναι ευρέως διαδεδομένο και πολύ φορητό (*portable*). Επιπλέον, η υλοποίηση του MPI που χρησιμοποιείται στο σύστημα μοιραζόμενης μνήμης είναι μια υλοποίηση βελτιστοποιημένη για συστήματα κοινής μοιραζόμενης μνήμης, που επιπρόσθετα υποστηρίζει το πρότυπο MPI 1.2, όπως αυτό τεκμηριώνεται από το MPI Forum [5.2], καθώς επίσης και ορισμένα χαρακτηριστικά του νεότερου προτύπου MPI-2. Στο σύστημα αυτό, το MPI χρησιμοποιεί τον πρώτο μηχανισμό διασύνδεσης που μπορεί να ανιχνεύσει και να ρυθμίσει σωστά. Από προεπιλογή, εάν το MPI εκτελείται σε πολλούς υπολογιστές ή εάν δοθούν πολλαπλά εκτελέσιμα αρχεία ως ορίσματα στην εντολή mpirun, το λογισμικό αναζητά για διασυνδέσεις με την ακόλουθη σειρά (σε συστήματα IRIX): XPMEM, GSN, MYRINET, TCP/IP. Μόνο μια διασύνδεση ρυθμίζεται και αρχικοποιείται για ολόκληρη τη δουλειά του MPI, με την εξαίρεση του XPMEM. Αν σε ορισμένους κόμβους μπορεί να γίνει χρήση XPMEM, αλλά όχι σε μερικούς άλλους, ένα επιπρόσθετο δίκτυο διασύνδεσης επιλέγεται. Ωστόσο, αυτό δεν συμβαίνει στην περίπτωση των παραδειγμάτων μας καθώς το MPI εκτελείτο σε ένα ενιαίο πολυεπεξεργαστικό υπολογιστικό σύστημα 16 επεξεργαστών και όλες οι επικοινωνίες πραγματοποιούνταν χρησιμοποιώντας απομονωτές μοιραζόμενης μνήμης (*shared memory buffers*).

5.1 Μελέτη Περίπτωσης I: Αλγόριθμος Διάχυσης Λάθους Floyd-Steinberg (FS)

Ο υπολογιστικός αλγόριθμος διάχυσης λάθους Floyd-Steinberg [5.7] είναι ένας αλγόριθμος επεξεργασίας εικόνας που χρησιμοποιείται για τη διάδοση σφάλματος στην εξομάλυνση μιας *πλάτους επί ύψος* διαστάσεων ασπρόμαυρης (ακριβέστερα σε τόνους του γκρι) εικόνας. Οι οριακές συνθήκες αγνοούνται. Ο ψευδοκώδικας του αλγορίθμου δίνεται στο Σχήμα 5.1.

```
/* Floyd-Steinberg */
for (i=1; i<width; i++) {
    for (j=1; j<height; j++) {

  LOOP
  BODY   {  I[i][j] = trunk(J[i][j]) + 0.5;
            err = J[i][j] - I[i][j]*255;
            J[i+1][j]   += err*(7/16);
            J[i-1][j+1] += err*(3/16);
            J[i][j-1]   += err*(5/16);
            J[i+1][j+1] += err*(1/16);

    } // end j
} // end i
```

Σχήμα 5.1: Ο αλγόριθμος διάχυσης λάθους Floyd-Steinberg

5.2 Μελέτη Περίπτωσης II: Αλγόριθμος Μεταβατικού Κλεισίματος Transitive Closure (TC)

Ο αλγόριθμος μεταβατικού κλεισίματος χρησιμοποιείται για να βρούμε αν δύο οποιεσδήποτε κορυφές σε ένα γράφο είναι συνδεδεμένες. Επισήμως, αν G = (V, E) είναι ένας γράφος, τότε το μεταβατικό κλείσιμο του G ορίζεται ως ο γράφος $G^* = (V, E^*)$, όπου $E^* = \{ (i, j) \mid$ υπάρχει ένα μονοπάτι από το i στο j που περιέχεται στο G $\}$. Το μεταβατικό κλείσιμο ενός γράφου καθορίζεται υπολογίζοντας τον πίνακα συνδεσιμότητας (*connectivity matrix*) P, για τα στοιχεία του οποίου ισχύει ότι P[i][j] = 1 αν υπάρχει ένα μονοπάτι από το i στο j και 0 διαφορετικά. Ο πίνακας P μπορεί να ληφθεί χρησιμοποιώντας τον αλγόριθμο του Warshall, όπως φαίνεται στο Σχήμα 5.2. Σε αυτόν τον αλγόριθμο, ο πίνακας A είναι ο πίνακας γειτνίασης του G, που σημαίνει πως ισχύει ότι A[i][j] = 1 αν και μόνο αν $(i, j) \in E$.

```
/* Transitive Closure */
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        /* There's a path if there's an edge */
        P[i][j] = A[i][j];
for (k=0; k<N; k++) {
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if ( !P[i][j] )
                P[i][j] = P[i][k] && P[k][j];
        } // end j
    } // end i
} // end k
```

Σχήμα 5.2: Ο αλγόριθμος μεταβατικού κλεισίματος του Warshall

5.3 Μελέτη Περίπτωσης ΙΙΙ: Τεχνητό Απόσπασμα Κώδικα

Η επίδοση της πλατφόρμας CRONUS έχει επίσης αξιολογηθεί με εκτεταμένο πειραματισμό σε τεχνητά δημιουργηθέντα παραδείγματα. Για να δούμε πως ακριβώς η πλατφόρμα CRONUS χειρίζεται τέτοιες περιπτώσεις, έχουμε επιλέξει ως τρίτη περίπτωση μελέτης ένα τεχνητό απόσπασμα κώδικα. Ο ψευδοκώδικας για το παράδειγμα αυτό δίνεται στο Σχήμα 5.3, που είναι ο ίδιος με αυτόν του σχήματος 3.2 και επαναλαμβάνεται εδώ για ευκολία.

```
for (i1 = 0 ; i1 <= N1 ; i1++ ) {
    for (i2 = 0 ; i2 <= N2 ; i2++ ) {

LOOP BODY {
        S1: A[i1][i2] = 2*A[i1-1][i2-8]+A[i1-8][i2-1]+3*B[i1-2][i2-5];
        S2: B[i1][i2] = 0;
        S3: for (i3 = 0 ; i3 < i2 ; i3++ )
                B[i1][i2] += C[i1][i3]*C[i1][i2];
        S4: B[i1][i2] += B[i1-3][i2-3]+B[i1-6][i2-2];

    }
}
```

Σχήμα 5.3: Δισδιάστατος βρόχος με ένα γενικό σώμα βρόχου και διανύσματα εξαρτήσεων $\vec{d}_1$ =(1, 8), $\vec{d}_2$ =(2, 5), $\vec{d}_3$ =(3, 3), $\vec{d}_4$ =(6, 2) και $\vec{d}_5$ =(8, 1)

Παρουσιάζουμε το συγκεκριμένο παράδειγμα για να δείξουμε ότι το CRONUS μπορεί να χειριστεί και εφαρμογές που περιλαμβάνουν βρόχους που διέπονται από γενικά ομοιόμορφα διανύσματα εξάρτησης και όχι μόνο από μοναδιαία διανύσματα εξάρτησης, όπως συμβαίνει με τις εφαρμογές των δυο πρώτων περιπτώσεων μελέτης. Η δοκιμαστική περίπτωση είναι ένας φωλιασμένος βρόχος δύο διαστάσεων με ένα γενικό σώμα βρόχου, το οποίο περιλαμβάνει έναν επιπλέον τρίτο βρόχο. Έχει πέντε δισδιάστατα διανύσματα εξάρτησης, που προκύπτουν από τους δύο εξωτερικούς βρόχους.

5.4 Μελέτη Περίπτωσης IV: Αλγόριθμος Εκτίμησης Κίνησης FSBM

```
/* FSBM ME */
for (h=0; h<Nh; h++) {
    for (v=0; v<Nv; v++) {
        MV[h][v] = {0,0};
        Dmin[h][v] = maxint;
        for (m=0; m<=2*p; m++) {
            for (n=0; n<=2*p; n++) {
                MAD[m][n]=0;
                for (i=h*N; i<h*N+N; i++) {
                    for (j=v*N; j<v*N+N; j++) {
                        MAD[m][n] += fabsf(x[i][j]-y[i+m][j+n]);
                    } // end j
                } // end i
                if (Dmin[m][n]>MAD[m][n]) {
                    Dmin[m][n]=MAD[m][n];
                    MV[h][v] = {m,n};
                } // end if
            } // end n
        } // end m
    } // end v
} // end h
```

LOOP BODY

Σχήμα 5.4: Φωλιασμένος βρόχος έξι επιπέδων του αλγορίθμου εκτίμησης κίνησης πλήρους αναζήτησης με αντιστοίχιση τμημάτων

Ο αλγόριθμος Εκτίμησης Κίνησης Πλήρους Αναζήτησης με Ταύτιση Τμήματος (Full-Search Block-Matching Motion Estimation Algorithm - FSBM ME) [5.8] είναι μια μέθοδος αντιστοίχισης τμημάτων, για την οποία κάθε ψηφίδα (*pixel*) που ανήκει στην περιοχή

αναζήτησης δοκιμάζεται προκειμένου να βρεθεί το καλύτερα αντιστοιχιζόμενο τμήμα. Η εκτίμηση κίνησης τμημάτων σε πρότυπα κωδικοποίησης βίντεο όπως τα MPEG-1, 2, 4 και H.261 είναι μια από τις πιο υπολογιστικά απαιτητικές λειτουργίες πολυμέσων. Ο αλγόριθμος αντιστοίχισης τμημάτων είναι ένα θεμελιώδες στοιχείο στη συμπίεση βίντεο για την απαλοιφή της χρονικής επανάληψης μεταξύ γειτονικών πλαισίων (*frames*). Κάθε ψηφίδα σε κάθε τμήμα υποτίθεται ότι μετακινείται με την ίδια δισδιάστατη μετατόπιση που ονομάζεται διάνυσμα κίνησης (*motion vector*) και που προσδιορίζεται με βάση τον αλγόριθμο εκτίμησης κίνησης τμημάτων. Ο ψευδοκώδικας για τον αλγόριθμο εκτίμησης κίνησης FSBM δίδεται στο Σχήμα 5.4.

Σημειώνουμε ότι το ίδιο το σώμα του βρόχου περιέχει ένθετους βρόχους επαναλήψεων for. Ωστόσο, δεν έχει εξαρτήσεις όσον αφορά τους δυο εξωτερικούς βρόχους. Επιπλέον, ολόκληρο το σώμα του βρόχου εκτελείται ακολουθιακά. Αυτό σημαίνει ότι οι εξαρτήσεις στο εσωτερικό του σώματος του βρόχου ικανοποιούνται. Αυτό είναι αποδεκτό, καθώς το εργαλείο CRONUS είναι σχεδιασμένο να χειρίζεται τέτοιου είδους πολύπλοκα σώματα βρόχων. Επιπλέον, είναι επιθυμητό να έχουμε τέτοια υπολογιστικά απαιτητικά σώματα βρόχων, ώστε να μεγιστοποιείτε το κέρδος στην απόδοση του παράλληλου προγράμματος.

5.5 Αποτελέσματα Πειραματικών Δοκιμών

Στην ενότητα αυτή παρουσιάζονται τα αποτελέσματα των δοκιμών για τις περιπτώσεις μελέτης που περιγράφονται παραπάνω. Ο παράλληλος χρόνος εκτέλεσης (*parallel execution time*) είναι το άθροισμα του χρόνου επικοινωνίας (*communication time*), του χρόνου υπολογισμού (*computation time*) και του χρόνου αδράνειας (*idle time*), δηλαδή του χρόνου που σπαταλά ένας επεξεργαστής σε αναμονή δεδομένων έως ότου αυτά γίνουν διαθέσιμα ή σημείων μέχρι ότου αυτά να μπορούν να καταστούν επιλέξιμα για εκτέλεση. Αναλυτικότερα, ο χρόνος υπολογισμού είναι ο χρόνος εκτέλεσης του ίδιου του αλγορίθμου, ενώ ο χρόνος επικοινωνίας περιέχει τόσο το χρόνο της πραγματικής επικοινωνίας μεταξύ επεξεργαστών, όσο και τον πρόσθετο χρόνο που χρειάζεται ο αλγόριθμος δυναμικής δρομολόγησης. Αυτός ο πρόσθετος χρόνος είναι μια γνωστή επιβάρυνση όλων των τεχνικών δυναμικής δρομολόγησης κι έτσι πρέπει να φροντίζουμε πάντα ώστε να διατηρείτε στο ελάχιστο δυνατό. Στις περιπτώσεις

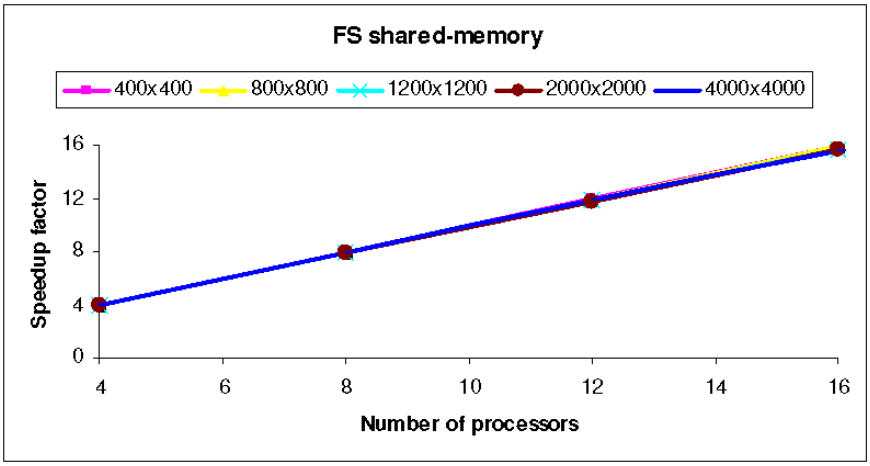
μελέτης των δοκιμών μας έχουμε απομονώσει και μετρήσει το χρόνο αυτό, ώστε να έχουμε το ποσοστό του χρόνου αυτού σε σχέση με το συνολικό χρόνο επικοινωνίας. Το ποσοστό αυτό βρέθηκε να διαφέρει σημαντικά ανάλογα με το υπό εξέταση παράδειγμα, αλλά σε όλες τις περιπτώσεις κυμάνθηκε μεταξύ 7% και 30% του συνολικά μετρημένου χρόνου επικοινωνίας. Ειδικότερα, για την περίπτωση μελέτης I (Floyd-Steinberg) αυτό κυμάνθηκε μεταξύ 15% και 25%, για την περίπτωση μελέτης II (Transitive Closure) περίπου 7% με 9% και για την περίπτωση μελέτης III (τεχνητό απόσπασμα κώδικα) περίπου 20% με 30%. Πιστεύουμε ότι οι τιμές αυτές είναι αποδεκτές καθόσον χρησιμοποιούμε έναν κατανεμημένο αλγόριθμο δυναμικής δρομολόγησης λεπτού κόκκου. Δεν έχουμε μετρήσει την επιβάρυνση αυτή στην περίπτωση μελέτης IV (FSBM), διότι στο παράδειγμα αυτό η απουσία εξαρτήσεων έχει ως αποτέλεσμα ένα αμελητέο συνολικά ποσοστό επικοινωνίας, όπως προκύπτει και από τους χρόνους του σχήματος 5.15.

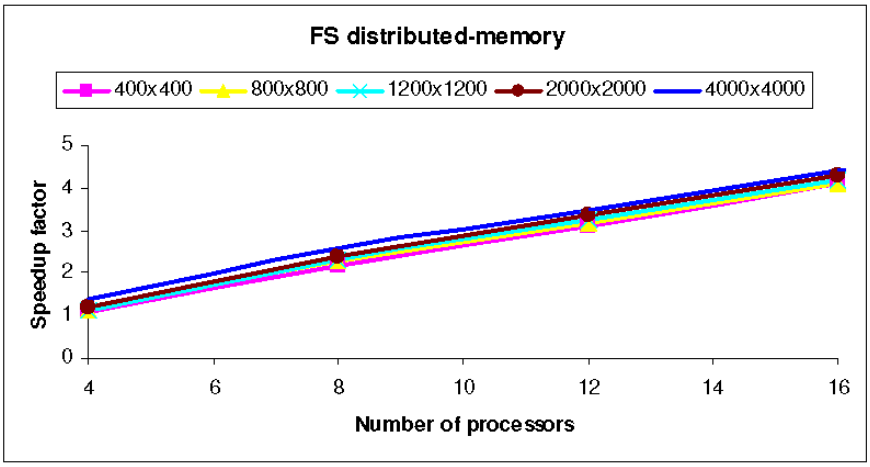
Η επιτάχυνση ενός προγράμματος, που συμβολίζεται με S_n, ορίζεται ως το πηλίκο του χρόνου εκτέλεσης του βέλτιστου ακολουθιακού αλγορίθμου (T_1) δια του χρόνου εκτέλεσης του παράλληλου προγράμματος (T_n) και δίνεται από τη σχέση: $S_n = \frac{T_1}{T_n}$.

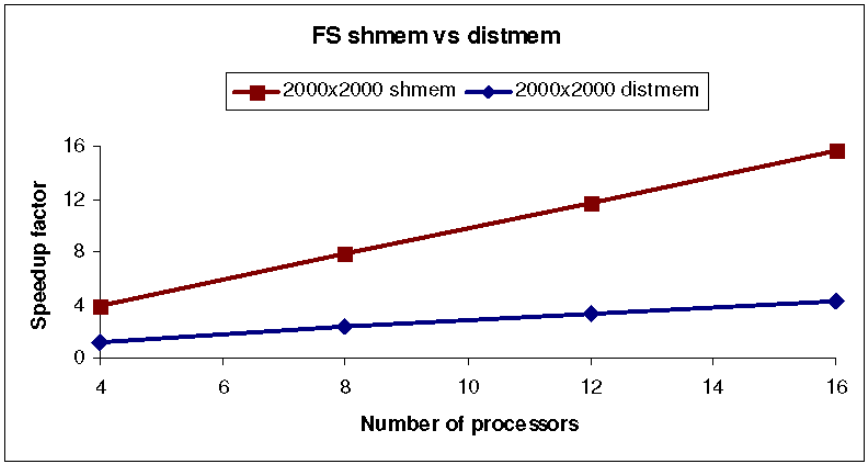
Τα σχήματα 5.5, 5.8, 5.11 και 5.14 στις επόμενες σελίδες απεικονίζουν την επιτάχυνση που επιτυγχάνουμε σε σχέση με τον αριθμό των χρησιμοποιούμενων επεξεργαστών, τόσο για το σύστημα κοινής μοιραζόμενης μνήμης όσο και για το σύστημα κατανεμημένης μνήμης, που έχουν ήδη περιγραφεί παραπάνω. Σημειώνουμε ότι οι παράλληλοι χρόνοι προέκυψαν ως ο μέσος όρος δεκάδων πειραματικών εκτελέσεων για κάθε περίπτωση μελέτης.

Έγινε προσπάθεια ώστε σε κάθε πείραμα τόσο το σύστημα κοινής μοιραζόμενης μνήμης όσο και το σύστημα κατανεμημένης μνήμης να είναι στη διάθεσή μας για σχεδόν αποκλειστική χρήση. Αυτό σημαίνει ότι δεν υπήρχε άλλο σημαντικό φορτίο στα συστήματα πέρα από τα δικά μας πειράματα, συνθήκη που μας επέτρεψε να επιτύχουμε την καλύτερη δυνατή απόδοση στα συστήματα αυτά, χρησιμοποιώντας την πλατφόρμα μας. Είναι προφανές ότι εάν υπήρχαν κι άλλα φορτία στα συστήματα, τότε τα αποτελέσματα, σε συνδυασμό και με το συνολικό αριθμό επεξεργαστών που χρησιμοποιείται σε κάθε πείραμα, θα επηρεάζονταν αρνητικά, δεν θα ήταν βέλτιστα και οπωσδήποτε όχι τόσο καλά όσο αυτά που παρουσιάζονται εδώ.

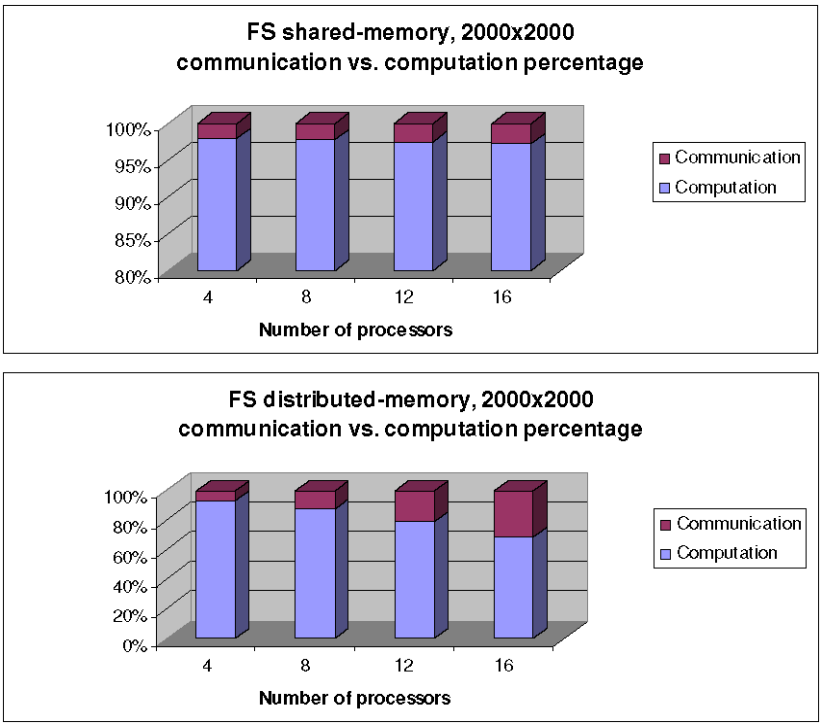
Στο Σχήμα 5.5 παρουσιάζεται η επιτάχυνση σε σχέση με τον αριθμό των επεξεργαστών για την περίπτωση του αλγόριθμου Floyd-Steinberg. Μπορεί κάνεις να δει ότι η επιτάχυνση του παράλληλου κώδικα κυμαίνεται από 3,95 (με ιδανική τιμή το 4) έως 15,70 (με ιδανική τιμή το 16) στην περίπτωση του συστήματος μοιραζόμενης μνήμης και από 1,18 (με ιδανική τιμή το 4) έως 4,21 (με ιδανική τιμή το 16) στην περίπτωση του συστήματος κατανεμημένης μνήμης. Στο τελευταίο γράφημα του σχήματος 5.5 συγκρίνουμε επίσης την επιτάχυνση που λάβαμε στο σύστημα μοιραζόμενης μνήμης με την επιτάχυνση που ελήφθη στο σύστημα κατανεμημένης μνήμης, για το παράδειγμα αυτό σε ένα δισδιάστατο χώρο ευρετηρίου διαστάσεων 2000 × 2000. Η σύγκριση επιβεβαιώνει την πρόβλεψη ότι το CRONUS αποδίδει πολύ καλύτερα σε συστήματα μοιραζόμενης μνήμης, στις περιπτώσεις εφαρμογών που είναι επικοινωνιακά έντονες. Αυτό απεικονίζεται και στο Σχήμα 5.6 όπου μπορεί κάνεις να δει ότι το ποσοστό του παράλληλου χρόνου εκτέλεσης που χρησιμοποιείται για επικοινωνία είναι πολύ υψηλότερο στο σύστημα κατανεμημένης μνήμης. Στο Σχήμα 5.7 αναλύεται παραπέρα πόσος από το χρόνο επικοινωνίας που έχει μετρηθεί στο Σχήμα 5.6 δαπανάται για πραγματική επικοινωνία, δηλαδή σε κλήσεις αποστολής και λήψης μηνυμάτων του MPI, και πόσος οφείλεται στην επιβάρυνση που επιφέρει ο αλγόριθμος δυναμικής δρομολόγησης.



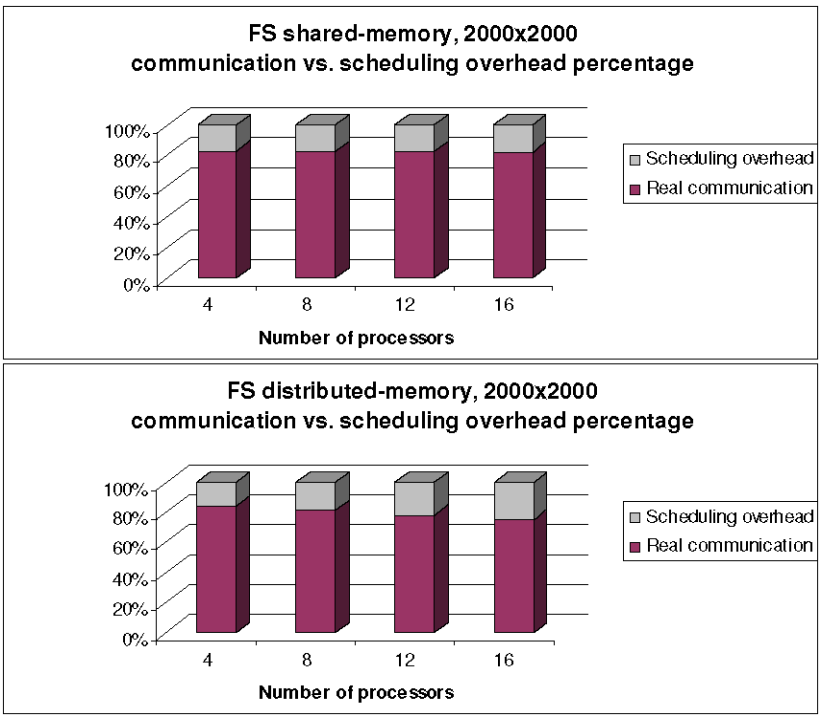




Σχήμα 5.5: Σύγκριση επιτάχυνσης για τη μελέτη της περίπτωσης Floyd-Steinberg

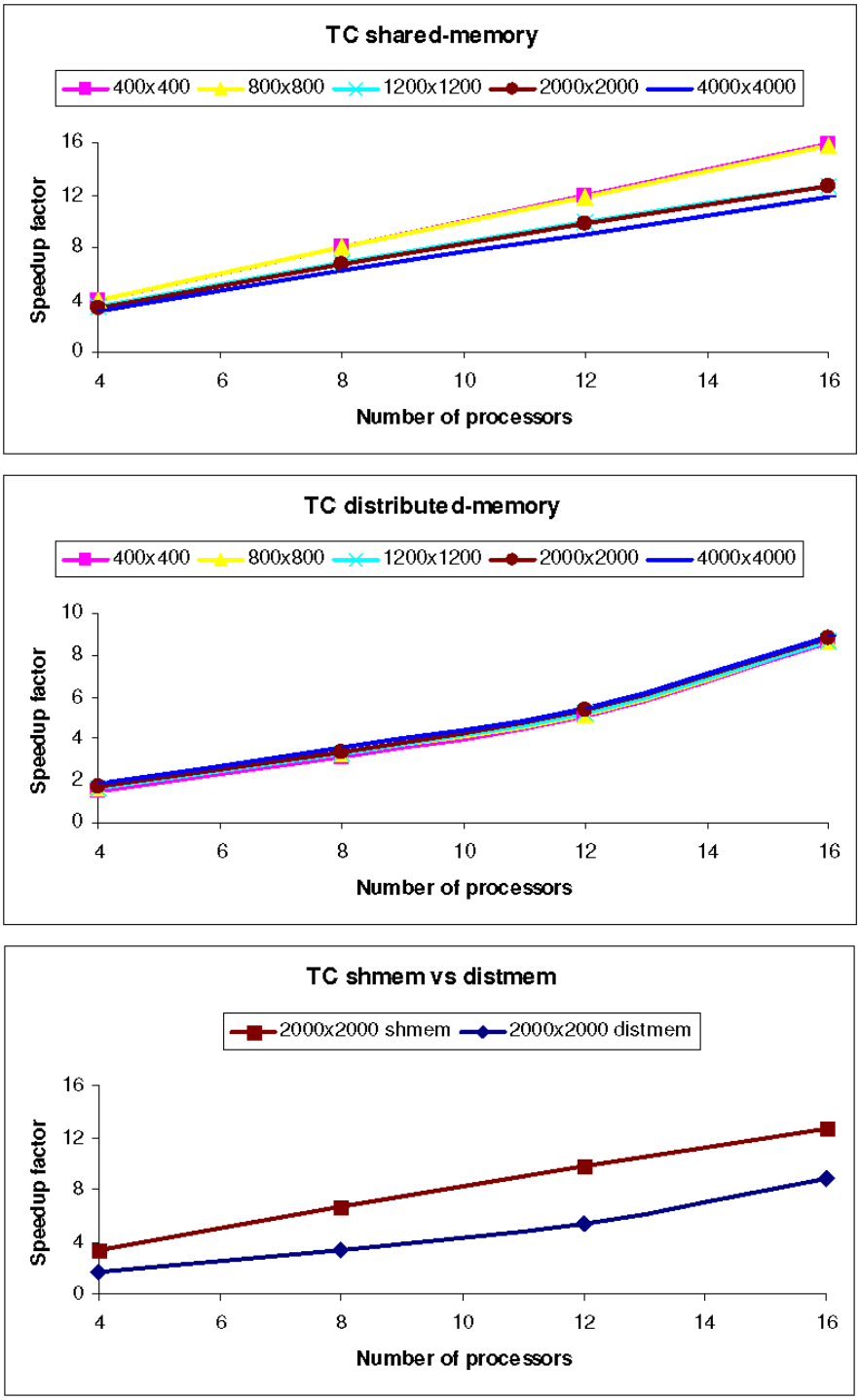


Σχήμα 5.6: Αντιπαράθεση ποσοστιαίων χρόνων επικοινωνίας και υπολογισμών για την περίπτωση μελέτης Floyd-Steinberg

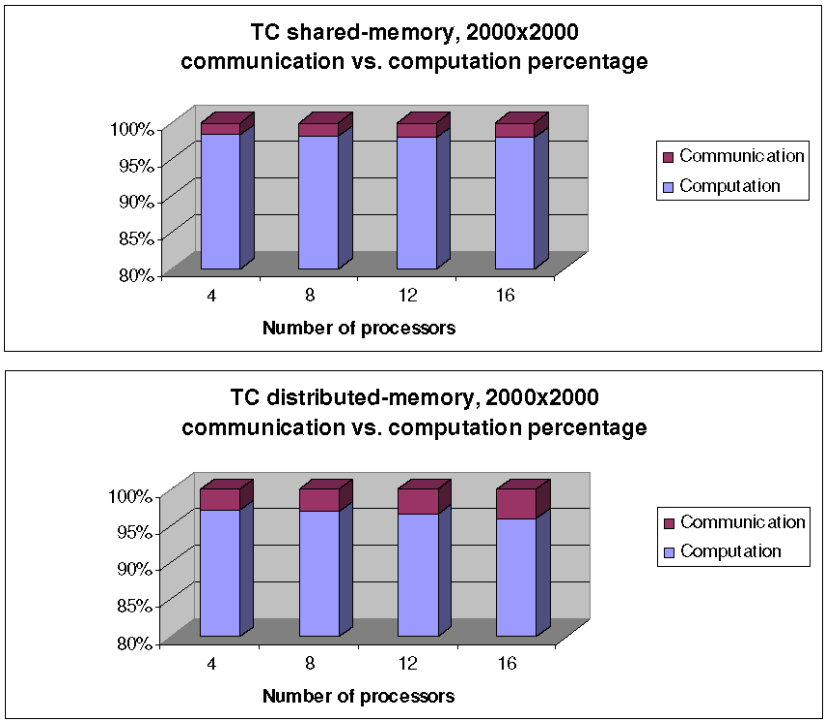


Σχήμα 5.7: Αντιπαράθεση ποσοστιαίων χρόνων πραγματικής επικοινωνίας και επιβάρυνσης δρομολόγησης για την περίπτωση μελέτης Floyd-Steinberg

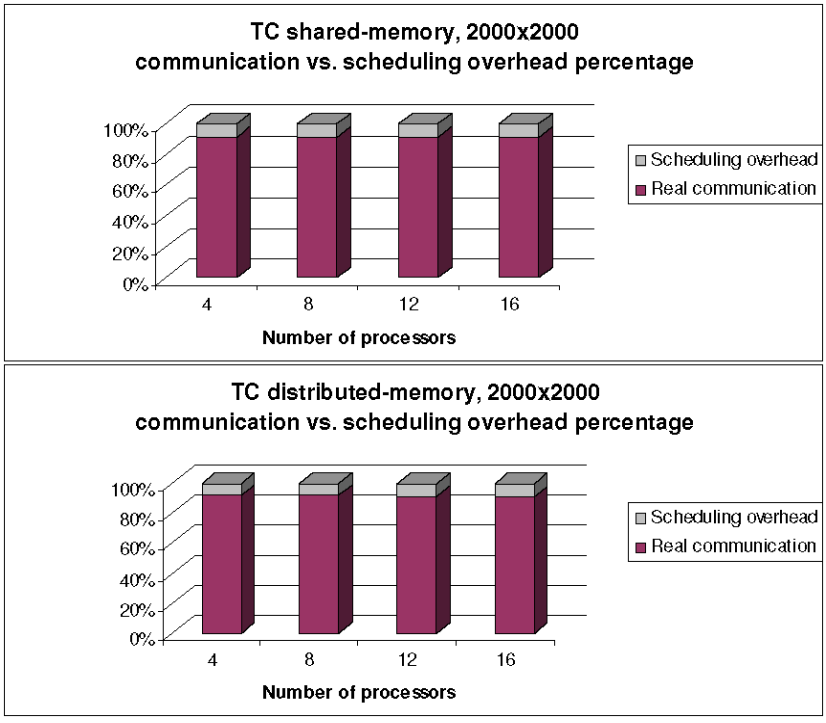
Στο Σχήμα 5.8 παρουσιάζονται τα αποτελέσματα για την περίπτωση του αλγορίθμου Transitive Closure. Η επιτάχυνση του παραλλήλου προγράμματος κυμαίνεται από 3,40 (με ιδανική τιμή το 4) έως 13,74 (με ιδανική τιμή το 16) για το σύστημα μοιραζόμενης μνήμης και από 1,64 (με ιδανική τιμή το 4) έως 8,70 (με ιδανική τιμή το 16) για το σύστημα κατανεμημένης μνήμης. Όπως και πριν, η επιτάχυνση που λαμβάνεται στο σύστημα μοιραζόμενης μνήμης αντιπαραβάλλεται με την επιτάχυνση που λαμβάνεται στο σύστημα κατανεμημένης μνήμης, πάλι για ένα δισδιάστατο χώρο ευρετηρίου διαστάσεων 2000 × 2000 και παρουσιάζεται στο τελευταίο γράφημα του σχήματος 5.8. Το Σχήμα 5.9 δείχνει τα ποσοστά επικοινωνίας και υπολογισμού για το παράδειγμα αυτό στα δυο συστήματα. Μπορεί κανείς να παρατηρήσει ότι η βελτίωση του ποσοστού επικοινωνίας του συστήματος μοιραζόμενης μνήμης επί του συστήματος κατανεμημένης μνήμης δεν είναι τόσο προφανής όπως στην περίπτωση Floyd-Steinberg. Αυτό οφείλεται στη φύση του προβλήματος, το οποίο είναι πιο απαιτητικό υπολογιστικά. Στο Σχήμα 5.10 αναλύεται παραπέρα πόσος από το χρόνο επικοινωνίας που έχει μετρηθεί στο Σχήμα 5.9 δαπανάται για πραγματική επικοινωνία, δηλαδή σε κλήσεις αποστολής και λήψης μηνυμάτων του MPI, και πόσος οφείλεται στην επιβάρυνση που επιφέρει ο αλγόριθμος δυναμικής δρομολόγησης.



Σχήμα 5.8: Σύγκριση επιτάχυνσης για τη μελέτη της περίπτωσης Transitive Closure

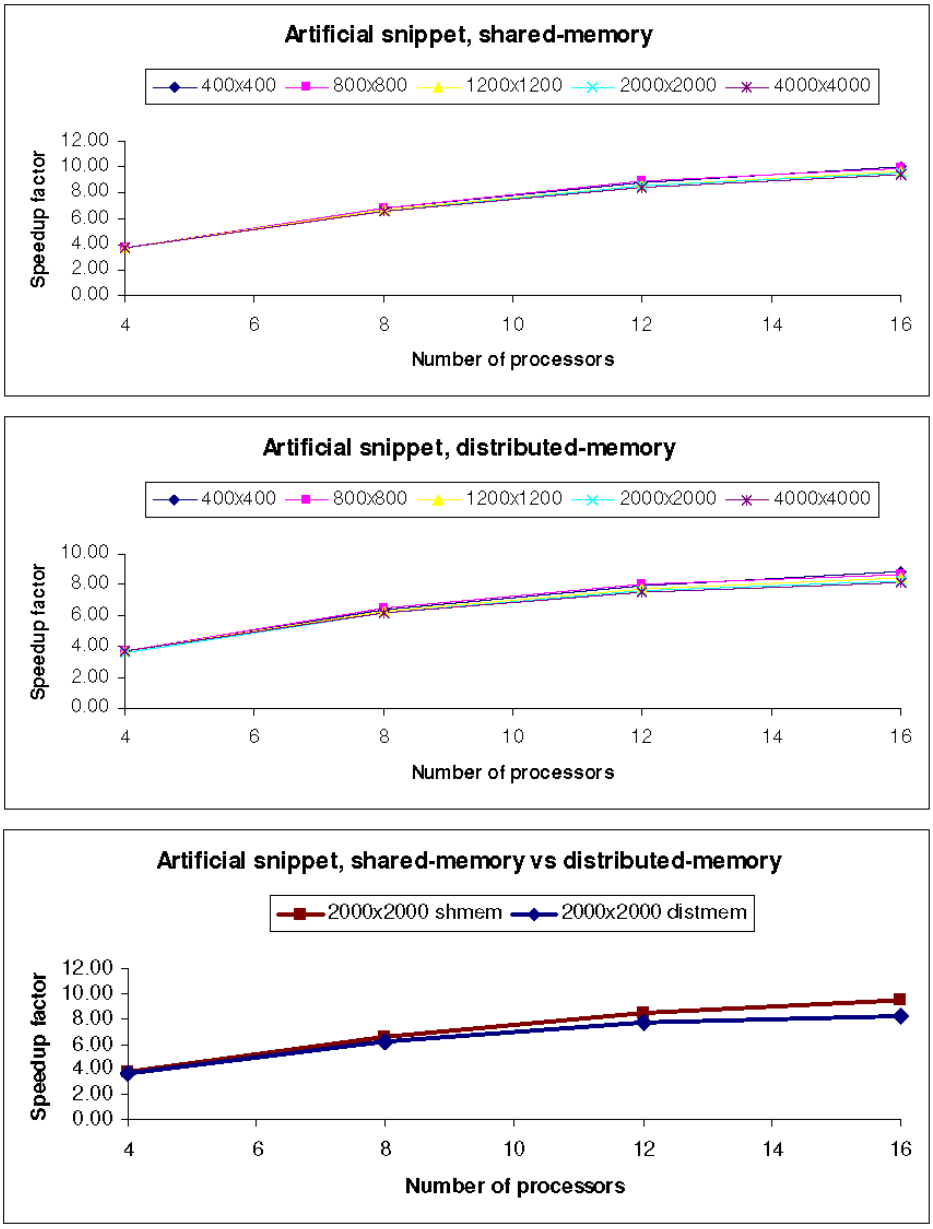


Σχήμα 5.9: Αντιπαράθεση ποσοστιαίων χρόνων επικοινωνίας και υπολογισμών για την περίπτωση μελέτης Transitive Closure

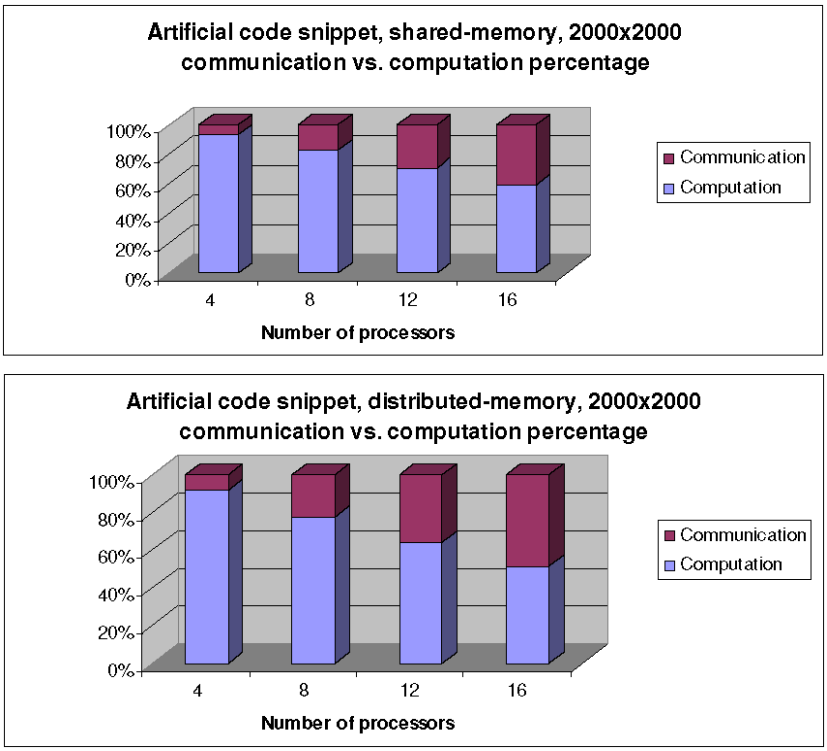


Σχήμα 5.10: Αντιπαράθεση ποσοστιαίων χρόνων πραγματικής επικοινωνίας και επιβάρυνσης δρομολόγησης για την περίπτωση μελέτης Transitive Closure

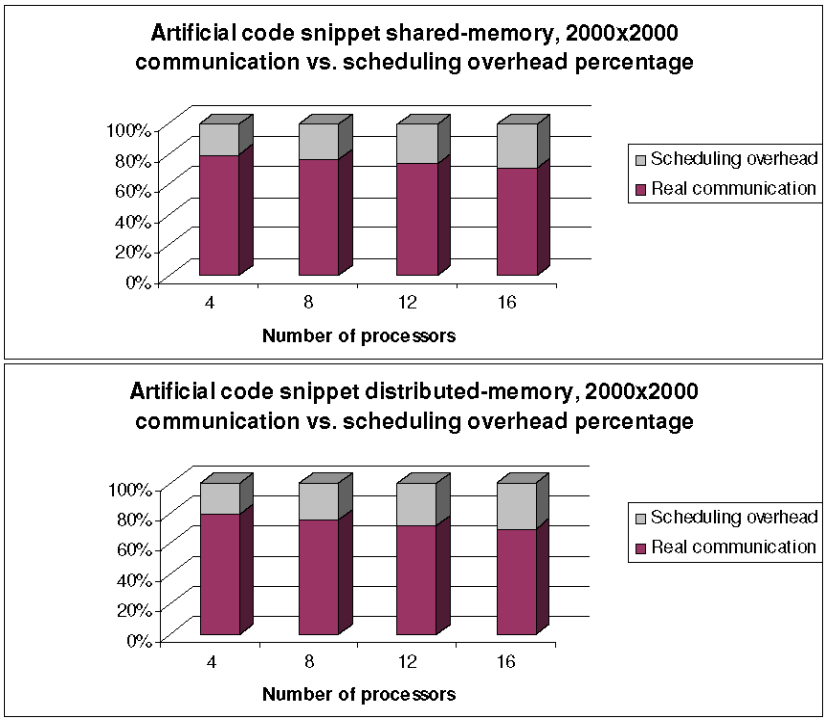
Στο Σχήμα 5.11 παρουσιάζονται τα αποτελέσματα για την περίπτωση του αποσπάσματος τεχνητού κώδικα. Η επιτάχυνση του παραλλήλου προγράμματος κυμαίνεται από 3,74 (με ιδανική τιμή το 4) έως 9,54 (με ιδανική τιμή το 16) για το σύστημα μοιραζόμενης μνήμης και από 3,66 (με ιδανική τιμή το 4) έως 8,27 (με ιδανική τιμή το 16) για το σύστημα κατανεμημένης μνήμης. Όπως και στα προηγούμενα παραδείγματα, η επιτάχυνση που λαμβάνεται στο σύστημα μοιραζόμενης μνήμης αντιπαραβάλλεται με την επιτάχυνση που λαμβάνεται στο σύστημα κατανεμημένης μνήμης, πάλι για ένα δισδιάστατο χώρο ευρετηρίου διαστάσεων 2000 × 2000 και παρουσιάζεται στο τελευταίο γράφημα του σχήματος 5.11. Το Σχήμα 5.12 παρουσιάζει τα ποσοστά επικοινωνίας και υπολογισμού για το παράδειγμα αυτό στα δυο συστήματα. Μπορεί κανείς να δει ότι η βελτίωση του ποσοστού της επικοινωνίας του συστήματος μοιραζόμενης μνήμης επί του συστήματος κατανεμημένης μνήμη είναι αρκετά προφανής. Αυτό οφείλεται στο γεγονός ότι το τεχνητό απόσπασμα κώδικα έχει περισσότερα διανύσματα εξάρτησης από τα υπόλοιπα παραδείγματα των δοκιμών μας, και γι' αυτό επωφελείται περισσότερο όταν εκτελείται σε ένα σύστημα μοιραζόμενης μνήμης. Στο Σχήμα 5.13 αναλύεται παραπέρα πόσος από το χρόνο επικοινωνίας που έχει μετρηθεί στο Σχήμα 5.12 δαπανάται για πραγματική επικοινωνία, δηλαδή σε κλήσεις αποστολής και λήψης μηνυμάτων του MPI, και πόσος οφείλεται στην επιβάρυνση του αλγορίθμου δυναμικής δρομολόγησης.



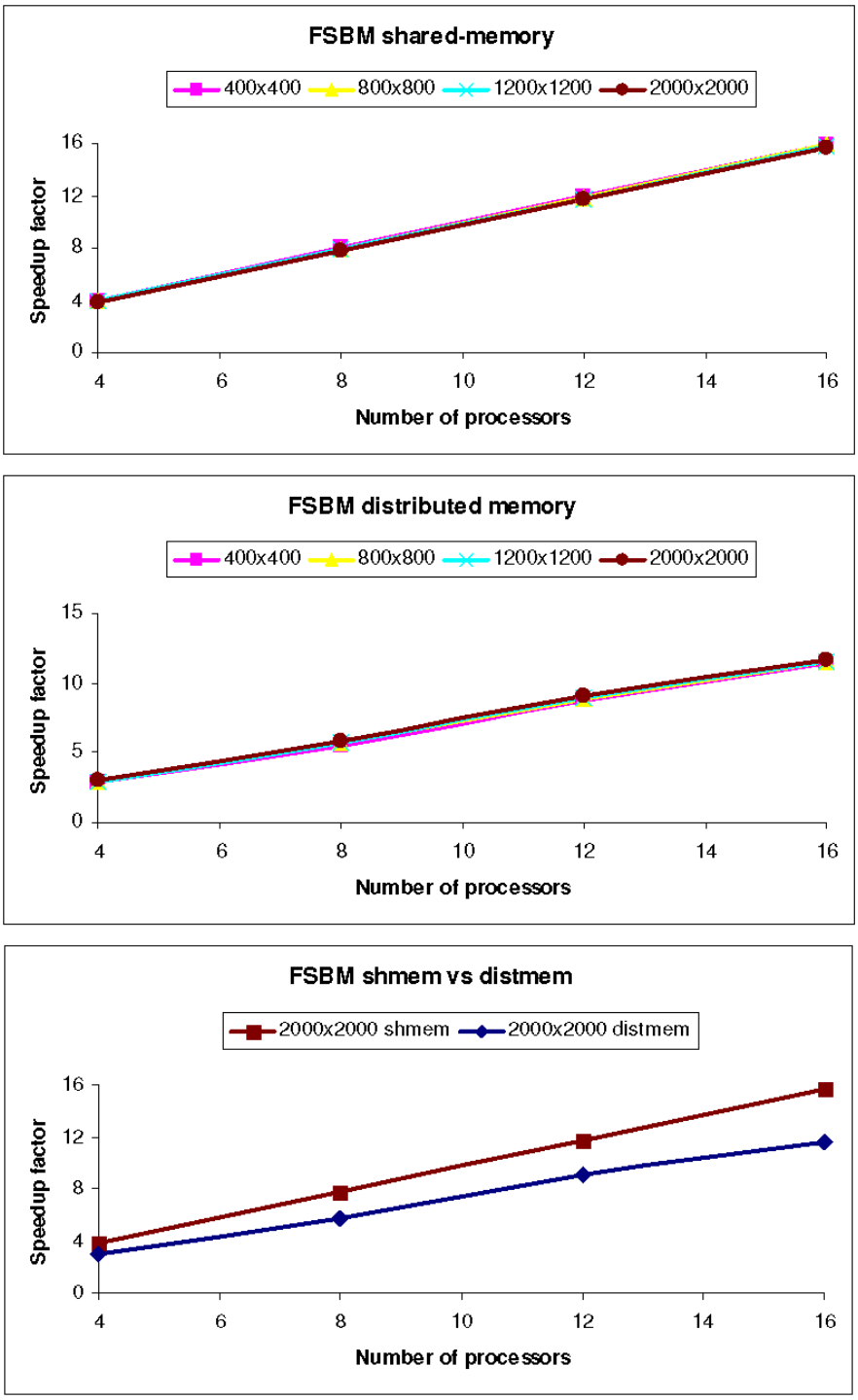
Σχήμα 5.11: Σύγκριση επιτάχυνσης για τη μελέτη της περίπτωσης Τεχνητού Κώδικα



Σχήμα 5.12: Αντιπαράθεση ποσοστιαίων χρόνων επικοινωνίας και υπολογισμών για την περίπτωση μελέτης Τεχνητού Κώδικα



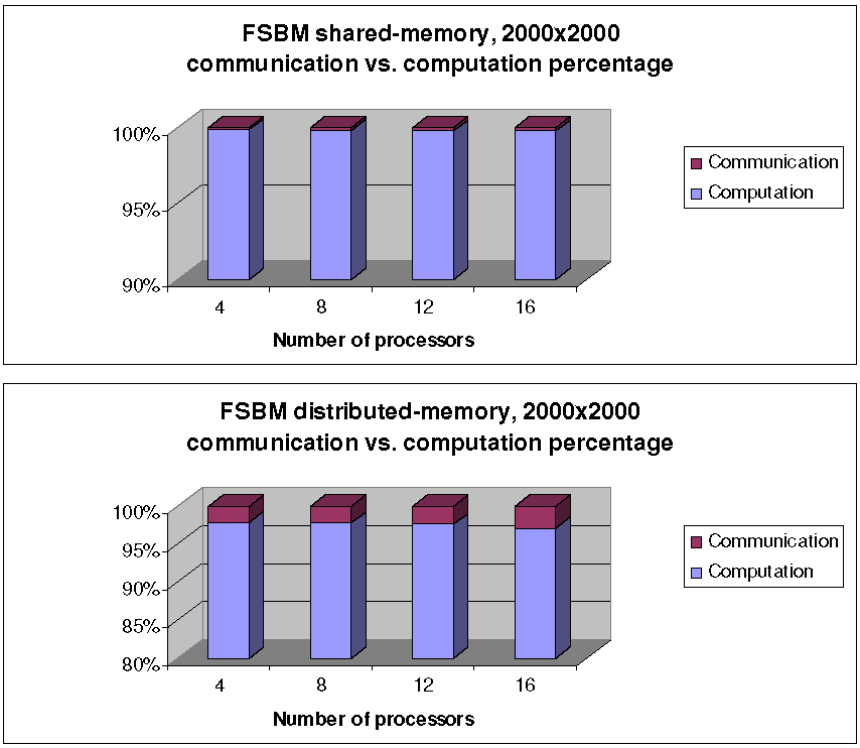
Σχήμα 5.13: Αντιπαράθεση ποσοστιαίων χρόνων πραγματικής επικοινωνίας και επιβάρυνσης δρομολόγησης για την περίπτωση μελέτης Τεχνητού Κώδικα



Σχήμα 5.14: Σύγκριση επιτάχυνσης για τη μελέτη της περίπτωσης FSBM

Στο Σχήμα 5.14 παρουσιάζονται τα αποτελέσματα για την επιτάχυνση του κώδικα του παραλλήλου προγράμματος για την περίπτωση του αλγορίθμου FSBM, η οποία κυμαίνεται από 3,92 (με ιδανική τιμή το 4) έως 15,79 (με ιδανική τιμή το 16) για το σύστημα μοιραζόμενης μνήμης και από 2,93 (με ιδανική τιμή το 4) έως 11,59 (με ιδανική τιμή το 16) για το σύστημα κατανεμημένης μνήμης. Όπως και πριν, η επιτάχυνση που

λαμβάνεται στο σύστημα μοιραζόμενης μνήμης αντιπαραβάλλεται με την επιτάχυνση που λαμβάνεται στο σύστημα κατανεμημένης μνήμης, πάλι για ένα δισδιάστατο χώρο ευρετηρίου διαστάσεων 2000 × 2000 και παρουσιάζεται στο τελευταίο γράφημα του σχήματος 5.14. Μπορεί κανείς να δει ότι υπάρχει μια μέτρια βελτίωση της απόδοσης και συνεπώς της επιτάχυνσης μεταξύ του συστήματος μοιραζόμενης μνήμης και του συστήματος κατανεμημένης μνήμης για το συγκεκριμένο παράδειγμα. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος FSBM δεν φέρει εξαρτήσεις στο βρόχο του. Η διαφορά στην απόδοση προκαλείται κυρίως από τα χαρακτηριστικά του συστήματος και δεν οφείλεται μόνο στο δίκτυο διασύνδεσής του. Αυτό μπορεί να επιβεβαιωθεί και από την εξέταση του σχήματος 5.15.



Σχήμα 5.15: Αντιπαράθεση ποσοστιαίων χρόνων επικοινωνίας και υπολογισμών για την περίπτωση μελέτης του αλγορίθμου FSBM

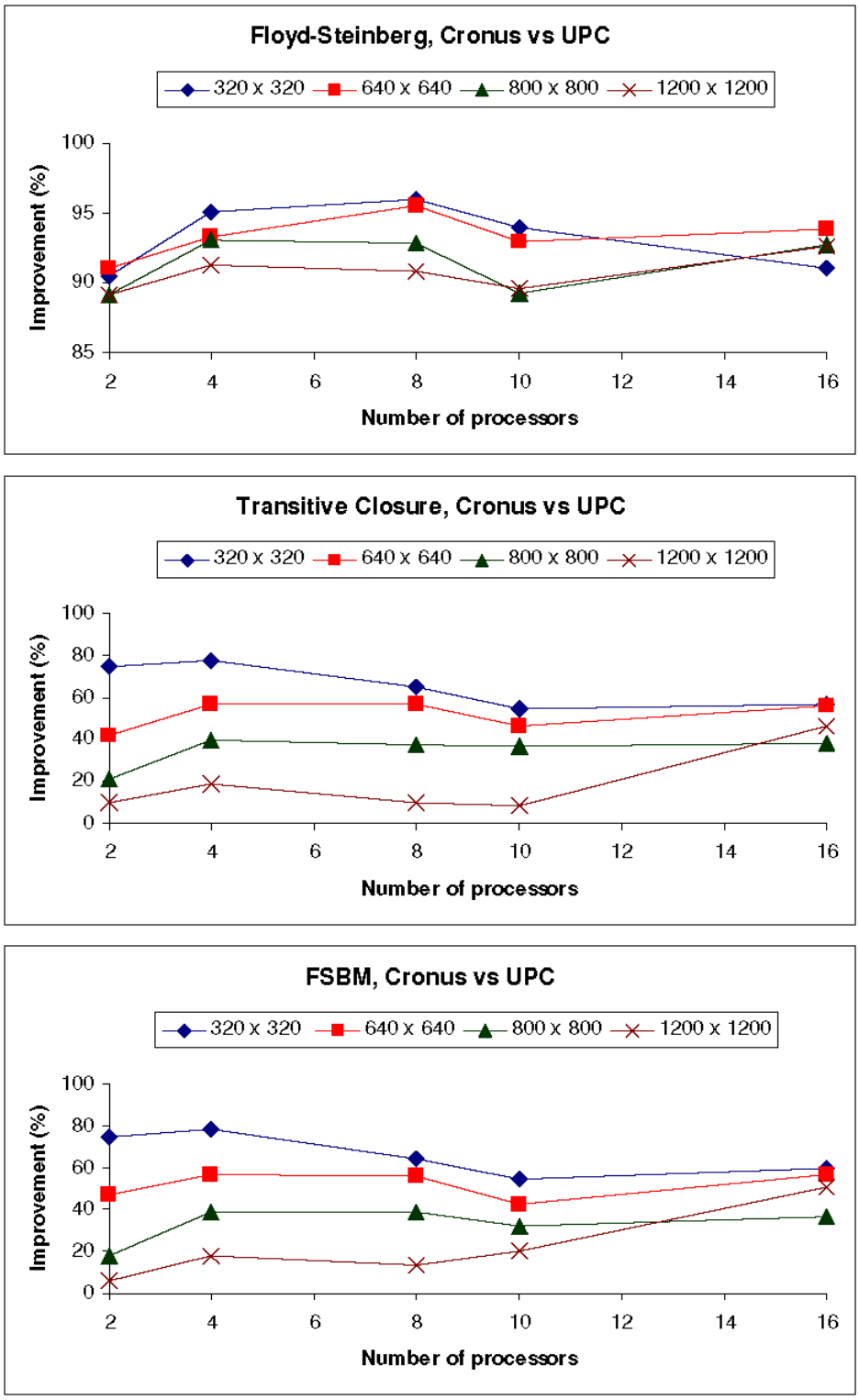
5.6 Σύγκριση CRONUS και Berkeley UPC

Το Berkeley UPC είναι μια υλοποίηση ανοιχτού κώδικα της γλώσσας προγραμματισμού Ενοποιημένη Παράλληλη C [5.4]. Η Ενοποιημένη Παράλληλη C (*Unified Parallel C - UPC*) είναι μια επέκταση της γλώσσας προγραμματισμού C, που σχεδιάστηκε για υπολογισμούς υψηλών επιδόσεων σε μεγάλης κλίμακας

πολυεπεξεργαστικά συστήματα, συστοιχίες προσωπικών υπολογιστών ή σταθμών εργασίας και ομάδες συστημάτων μοιραζόμενης μνήμης. Ο προγραμματιστής παραλλήλων εφαρμογών παρουσιάζεται με ένα ενιαίο, μοιραζόμενο, τεμαχισμένο χώρο διευθύνσεων, όπου οι μεταβλητές μπορούν να αναγνωσθούν και εγγραφούν άμεσα από κάθε επεξεργαστή, αλλά κάθε μεταβλητή είναι φυσικώς συνδεδεμένη με έναν μοναδικό επεξεργαστή. Το μοντέλο εκτέλεσης του UPC είναι παρόμοιο με εκείνο που χρησιμοποιείται από το μοντέλο προγραμματισμού ανταλλαγής μηνυμάτων, όπως αυτό χρησιμοποιείται κι από τις παράλληλες πλατφόρμες MPI και PVM. Αυτό το μοντέλο ονομάζεται μοντέλο Ενός Προγράμματος Πολλαπλών Δεδομένων (*Single Program Multiple Data - SPMD*) και είναι ένα δηλωτικό μοντέλο παραλληλισμού στο οποίο το ποσό της παραλληλίας καθορίζεται μοναδικά κατά το χρόνο εκκίνησης του προγράμματος (*startup time*). Δουλεύει μεταφράζοντας τον UPC κώδικα σε ισοδύναμο αυστηρό κώδικα C, που με τρόπο παρόμοιο με το CRONUS βασίζεται σε μια βιβλιοθήκη χρόνου εκτέλεσης για την επικοινωνία και την κατανομή των δεδομένων. Η ενυπάρχουσα βιβλιοθήκη επικοινωνίας είναι η GASNet [5.5], η οποία υποστηρίζει ένα ευρύ φάσμα αρχιτεκτονικών στόχων και μεθόδων επικοινωνίας (MPI, αυτόχθονα SMP, UPD πάνω από Ethernet κλπ.). Εσωτερικά, η επικοινωνία του GASNet βασίζεται σε μεγάλο βαθμό στο Μοντέλο Ενεργών Μηνυμάτων (*Active Messages Paradigm*) [5.6].

Σαφώς, το Berkeley UPC είναι ένα γενικότερο σύστημα από την πλατφόρμα CRONUS και η βιβλιοθήκη χρόνου εκτέλεσης του είναι σχεδιασμένη να εξυπηρετεί ένα πολύ ευρύτερο φάσμα προβλημάτων. Από την άλλη πλευρά η βιβλιοθήκη χρόνου εκτέλεσης του CRONUS έχει βελτιστοποιηθεί αποκλειστικά για γενικούς βρόχους, το οποίο σημαίνει ότι θα πρέπει να αποδίδει πολύ καλύτερα από μια γενική λύση. Για τη δοκιμή αυτής της υπόθεσης, συγκρίναμε την επίδοση του προγράμματος που δημιούργησε η πλατφόρμα CRONUS με την επίδοση του ισοδύναμου προγράμματος που δημιούργησε το UPC για τα ίδια παραδείγματα δοκιμών, με τη διενέργεια τους σε συστοιχία σταθμών εργασίας κατανεμημένης μνήμης. Χρησιμοποιήθηκε ο αλγόριθμος Διαδοχικής Δυναμικής Δρομολόγησης (SDS) και για τις δυο πλατφόρμες, όποτε στην πραγματικότητα τα μέρη του CRONUS που ασχολούνται με τον αλγόριθμο SDS χρειάστηκε να μεταφερθούν στο περιβάλλον του UPC. Αυτή ήταν μια σκόπιμη απόφαση, δεδομένου ότι αυτό που θέλαμε να μετρήσουμε ήταν πως οι δυο πλατφόρμες αποδίδουν όσον αφόρα την επικοινωνία (*communication*) και το χειρισμό

της τοπικότητας των δεδομένων (*data locality*). Συγκριτικά αποτελέσματα της πλατφόρμας CRONUS έναντι της πλατφόρμας UPC για τα παραδείγματα FS, TC και FSBM αντίστοιχα, συνοψίζονται στο Σχήμα 5.16.



Σχήμα 5.16: Σύγκριση επιδόσεων για τους παράλληλους κώδικες που παράγονται από τα εργαλεία CRONUS και UPC

Όλες οι περιπτώσεις δοκιμών εκτελέστηκαν στην ίδια συστοιχία σταθμών εργασίας κατανεμημένης μνήμης, όπως και στη μελέτη των άλλων περιπτώσεων. Η βελτίωση του CRONUS επί του UPC μετρήθηκε σύμφωνα με τον τύπο:

$$\frac{T_{UPC} - T_{CRONUS}}{T_{UPC}} \qquad (5.1)$$

όπου T_{CRONUS} και T_{UPC} είναι οι παράλληλοι χρόνοι εκτέλεσης που μετρήθηκαν για το CRONUS και το UPC, αντίστοιχα.

Το Σχήμα 5.16 παραθέτει το ποσοστό βελτίωσης έναντι του αριθμού των επεξεργαστών που χρησιμοποιούνται. Είναι σαφές ότι το CRONUS υπερτερεί του UPC σε όλες τις περιπτώσεις δοκιμών. Από το σχήμα αυτό φαίνεται ότι προκύπτει ένα κοινό μοτίβο σε κάθε περίπτωση δοκιμής: καθώς ο αριθμός των επεξεργαστών αυξάνει, το ποσοστό βελτίωσης συγκλίνει σε έναν σταθερό αριθμό. Πιο συγκεκριμένα, συγκλίνει στο 90% για τον αλγόριθμο Floyd-Steinberg, στο 55% για τον αλγόριθμο Transitive Closure και στο 50% για τον αλγόριθμο FSBM.

Για κάθε περίπτωση μελέτης πειραματιστήκαμε με διάφορα μεγέθη χώρου δεικτών και διαφορετικό αριθμό επεξεργαστών. Τα αποτελέσματα επιβεβαιώνουν την πρόβλεψη ότι το σύστημα μοιραζόμενης μνήμη υπερτερεί του συστήματος κατανεμημένης μνήμης σε όλες τις περιπτώσεις και μάλιστα κατά ένα συντελεστή που κυμαίνεται από 6% έως 60%. Αυτό δικαιολογείται από το γεγονός ότι η επιβάρυνση της επικοινωνίας ελαχιστοποιείται στα συστήματα μοιραζόμενης μνήμης. Ένα ακόμα συμπέρασμα που μπορεί να εξαχθεί από τα αποτελέσματα των δοκιμών είναι ότι το βάρος του σώματος του βρόχου του προβλήματος παίζει σημαντικό ρόλο στην απόδοση του παράλληλου κώδικα, ανεξάρτητα μάλιστα από την αρχιτεκτονική του υπολογιστικού συστήματος. Η υψηλότερη επίδοση του παράλληλου κώδικα για τον αλγόριθμο FSBM οφείλεται στο γεγονός ότι διαθέτει το βαρύτερο σώμα βρόχου από τις τρεις περιπτώσεις μελέτης, ενώ στον αντίποδα η επίδοση του παράλληλου κώδικα για τον αλγόριθμο Floyd-Steinberg είναι η φτωχότερη όλων εξαιτίας του πολύ ελαφρού σώματος του βρόχου του. Επιπλέον, ο παράλληλος κώδικας για τον αλγόριθμο Floyd-Steinberg που δημιουργήθηκε με την υποστήριξη του UPC απέδωσε πολύ χειρότερα από τις άλλες δυο περιπτώσεις δοκιμών, το όποιο και πάλι οφείλεται στο ελαφρύ σώμα του βρόχου του σε συνδυασμό με την ύπαρξη έντονης επικοινωνίας.

5.7 Αναφορές

[5.1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The Complete Reference", *The MIT Press, 1996*.

[5.2] MPI Forum, "Message Passing Interface Standard", *http://www-unix.mcs.anl.gov/mpi/indexold.html, 2002*.

[5.3] B. W. Kernigham and D. M. Ritchie, "The C Programming Language", 2nd Edition, *Englewood Cliffs, New Jersey, Prentice-Hall, 1988*.

[5.4] LBNL and UC Berkeley, Berkeley Unified Parallel C.

[5.5] Dan Bonachea, "Gasnet communication system", http://gasnet.cs.berkeley.edu/.

[5.6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation", *In 19th International Symposium on Computer Architecture, pages 256–266, Gold Coast, Australia, 1992*.

[5.7] R.W. Floyd and L. Steinberg, "An adaptive algorithm for spatial grey scale", *In Proc. Soc. Inf. Display, volume 17, pages 75–77, 1976*.

[5.8] H. Yee and Yu Hen Hu, "A novel modular systolic array architecture for full-search block matching motion estimation", *IEEE Transactions on Circuits and Systems for Video Technology, 5(5):407–416, October 1995*.

ΚΕΦΑΛΑΙΟ 6. ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΠΡΟΤΑΣΕΙΣ

Στη παρούσα διδακτορική διατριβή ασχοληθήκαμε με τρόπους διευκόλυνσης του προγραμματισμού σε συστήματα παράλληλης επεξεργασίας, προσεγγίζοντας το πρόβλημα από δύο διαφορετικές οπτικές γωνίες. Η πρώτη είναι από την πλευρά του συστήματος, ώστε να δώσουμε στον προγραμματιστή παραλλήλων εφαρμογών επιλογές για ευκολότερο παράλληλο προγραμματισμό και οδήγησε στην υλοποίηση του μηχανισμού των καθολικών σηματοφορέων. Η δεύτερη είναι από την πλευρά του μεταγλωττιστή, ώστε μια κατηγορία ακολουθιακών αλγορίθμων να μπορούν να μεταφράζονται αυτόματα σε ισοδύναμο παράλληλο κώδικα και οδήγησε στην υλοποίηση του εργαλείου CRONUS.

Αφορμή για την ανάπτυξη του μηχανισμού των καθολικών σηματοφορέων υπήρξε η προσπάθεια να δημιουργηθεί ένας μηχανισμός Κατανεμημένης Μοιραζόμενης Μνήμης (*Distributed Shared Memory - DSM*). Για να διασφαλίζεται η ακεραιότητα των περιεχομένων της κατανεμημένης μοιραζόμενης μνήμης, όταν γίνονται εγγραφές σε αυτήν από διαφορετικές διεργασίες που τρέχουν σε διαφορετικούς επεξεργαστές, είναι αναγκαία η ύπαρξη ενός μηχανισμού κλειδώματος και προστασίας των περιεχομένων της, ο οποίος να διασφαλίζει ότι όλες οι διεργασίες θα βλέπουν την κατανεμημένη μοιραζόμενη μνήμη να έχει τα ίδια περιεχόμενα. Ο μηχανισμός των καθολικών σηματοφορέων, που παρουσιάστηκε εδώ, είναι ιδανικός για τη λύση αυτού του προβλήματος, καθώς αφενός προσφέρει πλήρη λύση και αφετέρου απλοποιεί σημαντικά τον προγραμματισμό των μοντέλων Κατανεμημένης Μοιραζόμενης Μνήμης, ενώ επιπλέον προσφέρει λύση στο πρόβλημα του συγχρονισμού διεργασιών.

Από την άλλη, έχουμε αναπτύξει μια πλατφόρμα που ονομάζεται CRONUS, χρησιμοποιεί μια νέα πολιτική δυναμικής δρομολόγησης και εξισορρόπησης φορτίου και ενσωματώνει έναν αλγόριθμο από την υπολογιστική γεωμετρία, ώστε να παραλληλοποιεί αυτόματα γενικούς βρόχους. Η φιλοσοφία μας είναι ότι η απλότητα και η αποτελεσματικότητα αποτελούν τους βασικούς παράγοντες για την ελαχιστοποίηση του χρόνου εκτέλεσης του παράλληλου προγράμματος. Με αυτήν την

προσέγγιση, ο χρόνος μεταγλώττισης διατηρείται στο ελάχιστο, καθώς σε αυτή τη φάση δε διασχίζεται ο χώρος δεικτών και το πρόβλημα του ποια επανάληψη να εκτελεστεί και υπολογιστεί επόμενη από κάθε επεξεργαστή επιλύεται κατά το χρόνο εκτέλεσης. Αποφασίστηκε να γίνει αυτός ο συμβιβασμός διότι η έννοια του διαδόχου είναι πολύ απλή και αποδοτική και δεν επιφέρει μεγάλη χρονική επιβάρυνση. Η επιβάρυνση αυτή είναι πολύ μικρή σε σύγκριση με τα υπολογιστικά εντατικά σώματα βρόχων που στοχεύουμε. Το CRONUS δοκιμάστηκε τόσο σε ένα σύστημα κοινής μοιραζόμενης μνήμης όσο και σε συστοιχία σταθμών εργασίας κατανεμημένης μνήμης. Τα αποτελέσματα δείχνουν ότι το CRONUS αποδίδει αρκετά καλά σε συστήματα κατανεμημένης μνήμης, γεγονός που επιβεβαιώνεται και από τη σύγκριση με ένα πολύ γνωστό σύστημα όπως το Berkeley UPC. Επιπλέον, το CRONUS παρουσιάζει ακόμα καλύτερες επιδόσεις σε συστήματα μοιραζόμενης μνήμης.

Περαιτέρω εργασία, ως μελλοντική επέκταση της παρούσας διατριβής, θα μπορούσε να επικεντρωθεί στις ακόλουθες κατευθύνσεις:

(1) Την υλοποίηση λειτουργικών και αποδοτικών μοντέλων κατανεμημένης μοιραζόμενης μνήμης με τη χρήση του μηχανισμού των καθολικών σηματοφορέων, που θα διευκολύνουν περαιτέρω τον προγραμματιστή στην κωδικοποίηση των παράλληλων αλγορίθμων του·

(2) Τον προσδιορισμό των περιπτώσεων εκείνων, μέσω της απομόνωσης κάποιων χαρακτηριστικών όπως αριθμός εξαρτήσεων ή μοτίβα επικοινωνίας, που στα συστήματα κατανεμημένης μνήμης οδηγούν σε βελτίωση του χρόνου εκτέλεσης του παράλληλου προγράμματος με την υιοθέτηση του εναλλακτικού μοντέλου συγχρονισμού διεργασιών που κάνει χρήση του μηχανισμού των καθολικών σηματοφορέων, ώστε να μπορεί αυτό να επιλέγεται κατά το χρόνο μετάφρασης και αυτόματης παραλληλοποίησης ενός προγράμματος·

(3) Την τροποποίηση του CRONUS ώστε να λαμβάνει υπόψη την πιθανή ανομοιογένεια των κόμβων και των συνδέσμων επικοινωνίας σε συστήματα κατανεμημένης μνήμης, αλλά και την υιοθέτηση μιας περισσότερο χοντροκομμένης προσέγγισης παραλληλισμού για τα συστήματα αυτά με σκοπό τη βελτίωση της απόδοσης του παράλληλου κώδικα· και

(4) Την ολοκλήρωση του CRONUS σε ένα πλήρως αυτοματοποιημένο εργαλείο-μεταγλωττιστή για τη δημιουργία παράλληλου κώδικα που θα ενσωματώνει και τις ευκολίες του μηχανισμού των καθολικών σηματοφορέων. Αν και με το CRONUS η προσπάθεια εκ μέρους του χρήστη/προγραμματιστή είναι ήδη ελάχιστη σε σύγκριση με μια εξολοκλήρου χειρόγραφη προσέγγιση, το ιδανικό θα ήταν η άμεση λειτουργία του CRONUS επί του ακολουθιακού πηγαίου κώδικα χωρίς την ανάγκη βοηθητικών αρχείων οδηγιών ή οποιασδήποτε άλλης επέμβασης από τον προγραμματιστή.

ΔΗΜΟΣΙΕΥΣΕΙΣ

[1] "Cronus: A platform for parallel code generation based on computational geometry methods", T. Andronikos, F. M. Ciorba, P. Theodoropoulos, D. Kamenopoulos, G. Papakonstantinou, Journal of Systems and Software, 2007, Published by Elsevier Inc., doi:10.1016/j.jss.2007.11.715, http://dx.doi.org/10.1016/j.jss.2007.11.715.

[2] "Code generation for general loops using methods from computational geometry", T. Andronikos, F. M. Ciorba, D. Kamenopoulos, P. Theodoropoulos, G. Papakonstantinou, In Proceedings of the 16th IASTED Parallel and Distributed Computing and Systems Conference (PDCS 2004), pp.348-353, Cambrige, MA, USA, November 9-11, 2004.

[3] "Simple code generation for special UDLs", F. M. Ciorba, T. Andronikos, D. Kamenopoulos, P. Theodoropoulos, G. Papakonstantinou, In Proceedings of the 1st Balkan Conference in Informatics (BCI 2003), pp.466-475, Thessaloniki, Greece, November 21-23, 2003.

[4] "An efficient scheduling of uniform dependence loops", T. Andronikos, M. Kalathas, F. M. Ciorba, P. Theodoropoulos, G. Papakonstantinou, In Proceedings of the 6th Hellenic European Conference on Computer Mathematics and its Applications (HERCMA '03), Athens, Greece, September 25-27, 2003.

[5] "Scheduling nested loops with the least number of processors", T. Andronikos, M. Kalathas, F. M. Ciorba, P. Theodoropoulos, G. Papakonstantinou, P. Tsanakas, In Proceedings of the 21st IASTED International Conference on Applied Informatics (AI 2003), pp.713-718, Innsbruck, Austria, February 10-13, 2003.

[6] "TOPPER: A Software Environment for the Enhancement of Parallel Programs", D. Konstantinou, G. Goumas, P. Theodoropoulos, N. Koziris, G. Papakonstantinou, In Proceedings of the 8th Panhellenic Conference on Informatics (PCI 2001), Vol. 2, pp.48-57, Nicosia, Cyprus, November 8-10, 2001.

[7] "Artemis: A Digital Library environment for the Modern Greek Grey Literature", P. Theodoropoulos, J. Drossitis, P. Tsanakas, N. Koziris, V. Chrissikopoulos, D. Georgiou, K. Toraki, In Proceedings of the 3rd Greek Conference on Libraries, Athens, Greece, March 2001.

[8] "Καθολικοί σηματοφορείς σε παράλληλο περιβάλλον", Π. Θεοδωρόπουλος, Γ. Παπακωνσταντίνου, Π. Τσανάκας, 6ο Πανελλήνιο Συνέδριο Πληροφορικής, Αθήνα, 1997.

[9] "Global Semaphores in a Parallel Programming Environment", P. Theodoropoulos, P. Tsanakas, G. Papakonstantinou, In Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing interface, Krakow, Poland, November 03-05, 1997, M. Bubak, J. Dongarra, and J. Wasniewski, Eds. Lecture Notes In Computer Science, vol. 1332. Springer-Verlag, London, pp.151-158.

[10] "Extending Synchronization PVM Mechanisms", P. Theodoropoulos, G. Manis, P. Tsanakas, G. Papakonstantinou, In Proceedings of the 3rd European PVM Conference on Parallel Virtual Machine, Germany, October 07-09, 1996, A. Bode, J. Dongarra, T. Ludwig, and V. S. Sunderam, Eds. Lecture Notes In Computer Science, vol. 1156. Springer-Verlag, London, pp.315-318.

ΠΑΡΑΡΤΗΜΑ

Α. Υπολογισμός ελαχίστων σημείων σε υπερ-επίπεδα n-διαστάσεων

Η ανεύρεση του ελαχίστου σημείου ενός δεδομένου n-διάστατου υπερ-επιπέδου είναι ένα πρόβλημα βελτιστοποίησης. Επιπλέον, παρουσιάζει την ιδιότητα της βέλτιστης υποδομής, καθώς η βέλτιστη λύση για την περίπτωση των n-διαστάσεων εμπεριέχει τη βέλτιστη λύση για την περίπτωση των (n – 1)-διαστάσεων. Κατά συνέπεια, το ελάχιστο σημείο μπορεί να υπολογιστεί από έναν απλό αλγόριθμο δυναμικού προγραμματισμού με τη χρήση αναδρομής. Ο ψευδοκώδικας για τον αλγόριθμο αυτό δίνεται στο Σχήμα Α.1. Με συμμετρικό τρόπο, μπορεί κανείς να υπολογίσει το μέγιστο σημείο ενός n-διάστατου υπερ-επιπέδου.

Ακολουθώντας ένα τυπικό μοντέλο δυναμικού προγραμματισμού, ο αλγόριθμος του σχήματος Α.1 εργάζεται από κάτω προς τα πάνω (*bottom-up*). Αυτό σημαίνει ότι δοθέντος ενός χώρου δεικτών n-διαστάσεων, ο αλγόριθμος ξεκινά υπολογίζοντας πρώτα τα ελάχιστα σημεία για τα 1D υπερ-επίπεδα και μετακινείται μια διάσταση κάθε φορά μέχρι να υπολογίσει τα ελάχιστα σημεία και για τα υπερ-επίπεδα n-διαστάσεων του δεδομένου χώρου δεικτών. Οι λεπτομέρειες εξηγούνται με το ακόλουθο παράδειγμα.

Παράδειγμα Α.1: Ανατρέχοντας στο παράδειγμα 3.5, παρουσιάζουμε στο Σχήμα 3.4, το ελάχιστο και το μέγιστο σημείο των υπερ-επιπέδων $\Pi_9(2, 1)$: $2x_1 + x_2 = 9$ και $\Pi_{21}(2, 5)$: $2x_1 + 5x_2 = 21$. Το πρώτο υπερ-επίπεδο (Σχήμα 3.4(a)) περιλαμβάνει τα ακόλουθα πέντε σημεία του χώρου δεικτών (διατεταγμένα λεξικογραφικά): (0, 9) < (1, 7) < (2, 5) < (3, 3) < (4, 1), με ελάχιστο το σημείο (0, 9) και μέγιστο το σημείο (4, 1). Το τελευταίο υπερ-επίπεδο (Σχήμα 3.4(b)) περιλαμβάνει δυο σημεία του χώρου δεικτών: το σημείο (3, 3), το οποίο είναι το ελάχιστο, και το σημείο (8, 1), το οποίο είναι το μέγιστο.

Περιγράφουμε τώρα πως ο αλγόριθμος του σχήματος Α.1 υπολογίζει τα ελάχιστα σημεία. Προκειμένου να βρούμε τα ελάχιστα σημεία για το υπερ-επίπεδα $2x_1 + 5x_2 = k$,

$0 \le k \le 21$, αρχίζουμε με την εξέταση της οικογένειας των 1D υπερ-επιπέδων Π_k^1 (5), τα οποία ορίζονται από την εξίσωση $5x_1 = k$, $0 \le k \le 21$. Ένα 1D υπερ-επίπεδο έχει ελάχιστο σημείο αν και μόνο αν περιέχει ένα σημείο του χώρου δεικτών. Συνεπώς, στην προκειμένη περίπτωση η εξεύρεση του ελαχίστου σημείου ενός υπερ-επιπέδου (ή η διαπίστωση ότι το συγκεκριμένο υπερ-επίπεδο δεν περιέχει σημεία του χώρου δεικτών) είναι τετριμμένη. Για παράδειγμα, το ελάχιστο σημείο του Π_0^1 (5) είναι το (0), αλλά το Π_1^1 (5) δεν έχει ελάχιστο σημείο, γιατί δεν υπάρχει ακέραιος l που να ικανοποιεί την εξίσωση $5 \cdot l = 1$. Με αυτόν τον τρόπο, μπορούμε εύκολα να βρούμε τα ελάχιστα σημεία (εάν υπάρχουν) για τα υπερ-επίπεδα Π_k^1 (5), $0 \le k \le 21$.

```
for (hPlane = 0; hPlane ≤ maxHPlane; hPlane++) {
    posValue = min (hPlane / A[n], U[n]);
    isMin[hPlane][1] = (A[n] * posValue == hPlane);
    if (isMin[hPlane][1])
    minPoint[hPlane][1] = (posValue);
}

for (dim = 2; dim ≤ n; dim++)
    for (hPlane = 0; hPlane ≤ maxHPlane; hPlane++)
        for (value = hPlane; value ≥ 0; value--)
            if (isMin[value][dim-1]) {
                remPoints = hPlane – value;
                posValue = min (remPoints/A[n-dim], U[n-dim]);
                if (A[n-dim]*posValue == remPoints) {
                    isMin[hPlane][dim] = true;
                    minPoint[hPlane][dim]=(posValue) o minPoint[value][dim-1];
                    break;
                }
            }
```

*Το διάνυσμα **A** περιέχει τους συντελεστές της οικογένειας των υπερ-επιπέδων και το διάνυσμα **U** περιέχει τις συντεταγμένες του τερματικού σημείου. Για παράδειγμα, με αναφορά στο Σχήμα 3.4(a) του παραδείγματος 3.7, είναι **A** = { 2, 5 } και **U** = { 105, 90 }.*

*Ο πίνακας **isMin** είναι ένας δισδιάστατος πίνακας λογικών μεταβλητών, ορισμένων έτσι ώστε το **isMin**[k][n] είναι αληθές εάν το n-διαστάσεων υπερ-επίπεδο έχει ελάχιστο σημείο. Ο πίνακας **minPoint** είναι ένας δισδιάστατος πίνακας που αποθηκεύει τα ελάχιστα σημεία, δηλαδή το **minPoint**[k][n] περιέχει το ελάχιστο σημείο του n-διαστάσεων υπερ-επιπέδου k. Η συνένωση των διανυσμάτων δηλώνεται με το σύμβολο ◦, π.χ. το (3) ◦ (3) δίνει το διάνυσμα (3, 3).*

Σχήμα Α.1: Ψευδοκώδικας αλγορίθμου υπολογισμού ελαχίστου σημείου

Ας υποθέσουμε ότι θέλουμε να υπολογίσουμε το ελάχιστο σημείο $j_m = (j_1, j_2)$ του υπερ-επιπέδου $\Pi_{21}(2, 5)$. Το γεγονός ότι το j_m ανήκει στο $\Pi_{21}(2, 5)$ σημαίνει ότι $2j_1 + 5j_2 = 21$. Η λεξικογραφική διάταξη συνεπάγεται ότι το (j_2) είναι το ελάχιστο σημείο του Π^1_r (5), για κάποιο r, $0 \leq r \leq 21$. Ο αλγόριθμος αξιοποιεί τη γνώση των προηγουμένως υπολογισμένων ελαχίστων σημείων για τα 1D υπερ-επίπεδα Π^1_k (5), προκειμένου να καθορίσει την τιμή του r. Αυτό γίνεται ξεκινώντας από το 21 και κατεβαίνοντας κατά ένα μέχρι να βρούμε μια υποψήφια τιμή για το r. Η πρώτη τέτοια πιθανή τιμή είναι το 20, που αντιστοιχεί στο 1D υπερ-επίπεδο Π^1_{20} (5), με ελάχιστο σημείο το (4). Η τιμή αυτή είναι απορριπτέα, διότι δεν υπάρχει ακέραιος l, που να ικανοποιεί την εξίσωση $2 \cdot l = 21 - 20 = 1$. Έτσι, προχωρούμε για να βρούμε την επόμενη υποψήφια τιμή, η οποία είναι το 15, που αντιστοιχεί στο 1D υπερ-επίπεδο Π^1_{15} (5), με ελάχιστο σημείο το (3). Η τιμή αυτή είναι αποδεκτή, γιατί υπάρχει ένας ακέραιος l, που να ικανοποιεί την εξίσωση $2 \cdot l = 21 - 15 = 6$. Ως εκ τούτου, η τιμή της πρώτης συντεταγμένης είναι 3 και το ελάχιστο σημείο προκύπτει ως η συνένωση του (3) με το ελάχιστο σημείο του Π^1_{15} (5), είναι δηλαδή το σημείο (3, 3).

Β. Υπολογισμός διάδοχων σημείων σε υπερ-επίπεδα n-διαστάσεων

Η ανεύρεση του διάδοχου ενός σημείου του χώρου δεικτών είναι απλή και σαφής όταν τα ελάχιστα σημεία είναι ήδη γνωστά. Ο ψευδοκώδικας του αλγορίθμου για τον υπολογισμό του διάδοχου σημείου στη γενική περίπτωση n-διαστάσεων δίνεται στο Σχήμα Β.2.

Έστω $\mathbf{i} = (i_1, \ldots, i_n)$ ένα σημείου του χώρου δεικτών του υπερ-επιπέδου $\Pi_k(a_1, \ldots, a_n)$, διαφορετικό του μέγιστου, και έστω $\mathbf{j} = (j_1, \ldots, j_n)$ το διάδοχο σημείο του. Το γεγονός ότι τα **i** και **j** ανήκουν στο $\Pi_k(a_1, \ldots, a_n)$ σημαίνει ότι ισχύει $a_1i_1 + \ldots + a_ni_n = a_1j_1 + \ldots + a_nj_n = k$. Επιπλέον, ισχύει $\mathbf{i} < \mathbf{j}$, που σημαίνει ότι υπάρχει ένας ακέραιος r, $1 \leq r \leq n$ τέτοιος ώστε $i_1 = j_1, \ldots, i_{r-1} = j_{n-1}$ και $i_r < j_r$. Έτσι, προκειμένου να προσδιοριστεί το σημείο **j** ξεκινάμε με την ανεύρεση εκείνων των συντεταγμένων του **i** που μπορούν να αυξηθούν, οδηγώντας σε σημεία μεγαλύτερα του **i**, αλλά που εξακολουθούν να ανήκουν στο υπερ-επίπεδο k. Αυτό είναι τετριμμένο, καθώς η συντεταγμένη r του **i** μπορεί να αυξηθεί αν $i_r < u_r$, όπου u_r είναι το r-οστή συντεταγμένη του τερματικού σημείου **U** =

(u_1, …, u_n). Σημειώνουμε ότι είναι άσκοπο να αυξήσουμε τη n-οστή συντεταγμένη (δηλαδή την τελευταία) και χρησιμοποιούμε την παρατήρηση αυτή στον κώδικα ως βελτιστοποίηση. Από όλες τις πιθανές συντεταγμένες που μπορούμε να αυξήσουμε, χρησιμοποιούμε τη μέγιστη, διότι οποιαδήποτε άλλη υποψήφια συντεταγμένη μικρότερη της μέγιστης θα οδηγούσε σε ένα σημείο του χώρου δεικτών μεγαλύτερο από το διάδοχο του **i**. Αυτός είναι ο λόγος που αρχίζουμε την αναζήτηση μας από τα δεξιά προς τα αριστερά, δηλαδή από n – 1 προς 1. Αφού επιλέξουμε την υποψήφια συντεταγμένη, έστω r, προσπαθούμε να την αυξήσουμε κατά ένα, καθώς μια αύξηση μεγαλύτερη από ένα μπορεί να οδηγήσει σε σημείο μεγαλύτερο του διάδοχου, μέχρι να βρούμε το διάδοχο σημείο. Τα υπόλοιπα $k - (a_1j_1 + \ldots + a_rj_r)$ σημεία πρέπει να κατανεμηθούν στις τελευταίες (n – r) διαστάσεις με έναν έγκυρο τρόπο. Η κρίσιμη παρατήρηση εδώ είναι ότι για τον υπολογισμό του διάδοχου σημείου πρέπει να χρησιμοποιήσουμε το ελάχιστο σημείο του (n – r)-διαστάσεων υπερ-επιπέδου $k - (a_1j_1 + \ldots + a_rj_r)$. Η χρήση οποιοδήποτε άλλου σημείου θα έχει ως αποτέλεσμα ένα σημείο **j** μεγαλύτερο από το ζητούμενο διάδοχο σημείο.

Παράδειγμα Β.1: Συνεχίζοντας το προηγούμενο παράδειγμα, θα εξηγήσουμε πως βρίσκουμε το διάδοχο του σημείου (3, 3) του υπερ-επιπέδου $\Pi_{21}(2, 5)$. Χρησιμοποιώντας τον αλγόριθμο που δίνεται στο Σχήμα Β.2 προσπαθούμε να αυξήσουμε την πρώτη συντεταγμένη του σημείου (3, 3). Ξεκινάμε την αναζήτηση για τη σωστή τιμή από το $4 = 3 + 1$. Αυτό αφήνει $21 - 2 \cdot 4 = 13$ σημεία για τη δεύτερη συντεταγμένη. Δυστυχώς, το υπερ-επίπεδο Π_{13}^1 (5) δεν έχει ελάχιστο σημείο, όποτε θα πρέπει να απορρίψουμε την υποψήφια τιμή 4. Ομοίως, οι υποψήφιες τιμές 5, 6 και 7 απορρίπτονται. Ας εξετάσουμε τώρα πως λειτουργεί ο αλγόριθμος όταν προσπαθήσουμε με την υποψήφια τιμή 8. Στην περίπτωση αυτή, ελέγχουμε κατά πόσον το Π_5^1 (5) ($5 = 21 - 2 \cdot 8$) έχει ελάχιστο σημείο. Η απάντηση είναι θετική και το ελάχιστο σημείο είναι το (1). Ως εκ τούτου, ο διάδοχος του (3, 3) είναι το (8, 1).

```
for (dim = (n-1); dim ≥ 1; dim--)
    if (pointJ[dim] < U[dim]) {
        for (i = 1; i < dim; i++)
            successorJ[i] = pointJ[i];
        for (value = pointJ[dim] + 1; value ≤ U[dim]; value++) {
            successorJ[dim] = value;
            usedPoints = 0;
            for (i = 1; i ≤ dim; i++)
                usedPoints += A[i] * successorJ[i];
            remPoints = hPlane - usedPoints;
            if ( remPoints ≥ 0 && isMin[remPoints][n-dim] ) {
                for (i = dim + 1; i ≤ n; i++)
                    successorJ[i] = minPoint[remPoints][n-dim];
                break;
            }
        }
    }
```

Ο παραπάνω ψευδοκώδικας βρίσκει το διάδοχο του σημείου `pointJ` *που ανήκει στο υπερ-επίπεδο* `hPlane` *και το αποθηκεύει στο στοιχείο* `successorJ`*. Τα διανύσματα* ***A*** *και* ***U*** *περιέχουν τους συντελεστές της οικογένειας των υπερ-επιπέδων και τις συντεταγμένες του τερματικού σημείου, αντίστοιχα. Ο πίνακας* ***isMin*** *είναι ένας δισδιάστατος πίνακας λογικών μεταβλητών, για τον οποίο ισχύει ότι το* ***isMin****[k][n] είναι αληθές εάν το n-διάστατο υπερ-επίπεδο k έχει ελάχιστο σημείο. Ο πίνακας* ***minPoint*** *είναι ένας δισδιάστατος πίνακας που αποθηκεύει τα ελάχιστα σημεία, δηλαδή, το* ***minPoint****[k][n] περιέχει το ελάχιστο σημείο του n-διαστάσεων υπερ-επιπέδου k.*

Σχήμα Β.2: Ψευδοκώδικας αλγορίθμου ανεύρεσης διάδοχου σημείου