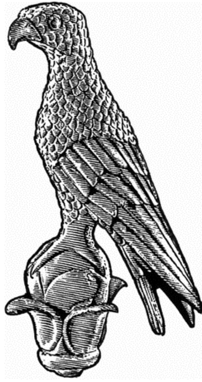


ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΣΧΟΛΗ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΜΣ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΒΕΛΤΙΣΤΗ ΕΚΤΕΛΕΣΗ ΕΡΓΑΣΙΩΝ ΣΕ ΠΑΡΑΛΛΗΛΑ ΕΤΕΡΟΓΕΝΗ ΥΠΟΛΟΓΙΣΤΙΚΑ ΠΕΡΙΒΑΛΛΟΝΤΑ

Αγγελική Σαλωνίτη

Επιβλέπων: Χρήστος Γκόγκος,

Αναπληρωτής Καθηγητής

Άρτα, Φεβρουάριος, 2019

ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΣΧΟΛΗ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΜΣ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΒΕΛΤΙΣΤΗ ΕΚΤΕΛΕΣΗ ΕΡΓΑΣΙΩΝ ΣΕ ΠΑΡΑΛΛΗΛΑ ΕΤΕΡΟΓΕΝΗ ΥΠΟΛΟΓΙΣΤΙΚΑ ΠΕΡΙΒΑΛΛΟΝΤΑ

Αγγελική Σαλωνίτη

Επιβλέπων: Χρήστος Γκόγκος,

Αναπληρωτής Καθηγητής ΤΕΙ Ηπείρου

Άρτα, Φεβρουάριος, 2019

OPTIMAL TASK SCHEDULING IN HETEROGENEOUS PARALLEL EXECUTION ENVIRONMENTS

Εγκρίθηκε από τριμελή εξεταστική επιτροπή

Άρτα, Φεβρουάριος, 2019

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

Επιβλέπων καθηγητής
Χρήστος Γκόγκος,
Αναπληρωτής Καθηγητής

Μέλος επιτροπής
Χρυσόστομος Στύλιος,
Καθηγητής

Μέλος επιτροπής
Δημήτριος Λιαροκάπης,
Λέκτορας

Ο Διευθυντής του ΠΜΣ

Χρυσόστομος Στύλιος,
Καθηγητής

Υπογραφή

© Σαλωνίτη, Αγγελική, 2019.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Δήλωση μη λογοκλοπής

Δηλώνω υπεύθυνα και γνωρίζοντας τις κυρώσεις του Ν. 2121/1993 περί Πνευματικής Ιδιοκτησίας, ότι η παρούσα μεταπτυχιακή εργασία είναι εξ ολοκλήρου αποτέλεσμα δικής μου ερευνητικής εργασίας, δεν αποτελεί προϊόν αντιγραφής ούτε προέρχεται από ανάθεση σε τρίτους. Όλες οι πηγές που χρησιμοποιήθηκαν (κάθε είδους, μορφής και προέλευσης) για τη συγγραφή της περιλαμβάνονται στη βιβλιογραφία.

Σαλωνίτη, Αγγελική

Υπογραφή

ΕΥΧΑΡΙΣΤΙΕΣ

Στην οικογένειά μου, στους συμφοιτητές μου, στους καθηγητές μου και ιδιαίτερα στον επιβλέποντα της διπλωματικής κ. Χρήστο Γκόγκο.

ΠΕΡΙΛΗΨΗ

Στα πλαίσια αυτής της εργασίας μελετήθηκαν διάφορες ακριβείς και ευρετικές τεχνικές που αποσκοπούν στη βέλτιστη εκτέλεση εργασιών σε παράλληλα ετερογενή υπολογιστικά περιβάλλοντα. Αναπτύχθηκε ένας ευρετικός αλγόριθμος, ο CPHEFT, που βασίζεται σε μια τεχνική η οποία ονομάζεται list scheduling. Αποτελεί μια τροποποίηση του αλγορίθμου HEFT, η οποία έγκειται στην ταξινόμηση των εργασιών με βάση την κρίσιμη διαδρομή στο γράφημα των εργασιών. Με την ταξινόμηση αυτή, ορίζεται η προτεραιότητα με την οποία πραγματοποιείται η ανάθεση των εργασιών σε καθέναν από τους διαθέσιμους επεξεργαστές ενός ετερογενούς συστήματος. Ο CPHEFT ανήκει στους αλγορίθμους οι οποίοι προγραμματίζουν την ανάθεση ενός κόμβου στον επεξεργαστή, που υποστηρίζει τον σχετικά προγενέστερο χρόνο έναρξης της εκτέλεσης αυτού του κόμβου (earliest start-time). Ο κύριος στόχος του αλγορίθμου είναι να ελαχιστοποιήσει τον συνολικό χρόνο εκτέλεσης των συσχετιζόμενων εργασιών (makespan). Οι εισροές, που χρησιμοποιήθηκαν για σύγκριση, ήταν DAGs τυχαία παραγόμενα από γεννήτρια, η οποία αναπτύχθηκε στα πλαίσια της εργασίας γι αυτό το σκοπό. Παρατηρήθηκε ότι ο αλγόριθμος CPHEFT επιτυγχάνει συγκρίσιμη απόδοση με τον αλγόριθμο HEFT και μπορεί να επιτύχει καλύτερη απόδοση για γράφους, οι οποίοι περιγράφουν προβλήματα με αυξημένο κόστος επικοινωνίας μεταξύ των εργασιών, καθώς οι μεταφορές δεδομένων διαδραματίζουν ουσιαστικό ρόλο στην εκτέλεση ροών εργασιών.

Λέξεις-κλειδιά: ετερογενή συστήματα, κατευθυνόμενοι μη κυκλικοί γράφοι, ευρετικοί αλγόριθμοι, ακέραιος προγραμματισμός

ABSTRACT

In this work, various exact methods and heuristic techniques that aim at optimal task scheduling in parallel heterogeneous computing environments have been studied. A new heuristic algorithm called CPHEFT, which is based on the *list scheduling* technique, was developed. The main difference between CPHEFT and HEFT algorithm is that CPHEFT orders the tasks based on the critical path of the associated DAG. This ordering specifies the priority assignment of tasks to each of the available processors of a heterogeneous system. CPHEFT belongs to a category of algorithms that program the assignment of a node to a processor that supports the earlier start-time of this node. The main target of the algorithm is to minimize the overall execution time of the associated tasks known as schedule length or makespan. The inputs, used for comparison, were DAGs randomly generated by a generator, developed especially for this purpose. It was observed that the CPHEFT algorithm achieves comparable performance to the HEFT algorithm and can achieve better performance for graphs that represent problems with increased cost of communication between tasks. This feature of CPHEFT is significant, since data transfers play an essential role in executing workflows.

Keywords: heterogeneous environments, directed acyclic graphs, heuristics, integer programming

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΕΥΧΑΡΙΣΤΙΕΣ iv

ΠΕΡΙΛΗΨΗ v

ABSTRACT vii

Πίνακας περιεχομένων....... ix

1. Εισαγωγή 1

2. Παράλληλα συστήματα και προγραμματισμός 3

2.1 Ομογενή και ετερογενή συστήματα 4

2.2 Μέθοδοι παραλληλοποίησης προγραμμάτων 5

2.3 JIT αλγόριθμοι και full-ahead αλγόριθμοι 6

3. Απεικόνιση παράλληλών εφαρμογών με κατευθυνόμενους μη κυκλικούς γράφους ... 11

3.1 Βασικές έννοιες γράφων 11

3.2 Αναπαράσταση γράφων στον Η/Υ 13

3.2.1 Λίστες ακμών....... 13

3.2.2 Πίνακες γειτνίασης....... 14

3.2.3 Λίστες γειτνίασης 15

3.3 Γράφοι εργασιών (task graphs)....... 16

3.4 Τοπολογική ταξινόμηση γράφων εργασιών 17

3.5 Ιδιότητες γράφων εργασιών 18

3.5.1 Κρίσιμο μονοπάτι 18

3.5.2 Επίπεδα κόμβων....... 19

4. Προγραμματισμός εργασιών 21

4.1 Προγραμματισμός εργασιών με και χωρίς κόστη επικοινωνίας 23

4.2 Προγραμματισμός εργασιών και NP πληρότητα 24

5. Βασικές ευρετικές τεχνικές 27

5.1 Προγραμματισμός λίστας (list scheduling) 27

5.1.1 Start Time Minimization 29

5.1.2 HLFET (highest levels first with estimated times) 29

5.1.3 MCP (Modified Critical path) 30

5.1.4 HEFT (Heterogeneous earliest time first) 31

5.1.5 Lookahead HEFT 35

5.1.6 Ο προτεινόμενος αλγόριθμος CPHEFT 36

5.2 Συσταδοποίηση (clustering) 40

5.2.1 Γραμμική συσταδοποίηση....... 41

5.3 Duplication 43
6. Προγραμματισμός εργασιών με ακριβείς τεχνικές.. 45
6.1 Ακέραιος προγραμματισμός 45
6.1.1 MATH... 46
6.1.2 MATHL 49
6.2 Προγραμματισμός με περιορισμούς.. 50
7. Πειράματα 53
7.1 Θέματα υλοποίησης... 53
7.1.1 Scheduling Length Ratio (SLR) 53
7.1.2 Speedup.. 53
7.1.3 Efficiency... 54
7.1.4 Processor Utilization.. 54
7.2 Γραφήματα επιστημονικών ροών εργασίας (scientific workflows)........................ 54
7.3 Αυτόματη δημιουργία συνθετικών γράφων 57
7.4 Αποτελέσματα πειραμάτων 60
7.5 Ερμηνεία αποτελεσμάτων.. 67
8. Συμπεράσματα 69
Αναφορές 71
ΠΑΡΑΡΤΗΜΑ... 74

1. ΕΙΣΑΓΩΓΗ

Η συνεχώς αυξανόμενη ζήτηση μεγαλύτερης υπολογιστικής ισχύος για την επίλυση προβλημάτων αιχμής της έρευνας και της τεχνολογίας έχει οδηγήσει στη χρήση υπολογιστών με πολλούς επεξεργαστές σε συνδυασμό με μεθόδους παράλληλης επεξεργασίας.

Η παράλληλη επεξεργασία αποτελεί μια εξέλιξη της σειριακής επεξεργασίας, η οποία προσπαθεί να μιμηθεί αυτό που σχεδόν πάντα συμβαίνει στο φυσικό κόσμο, δηλαδή πολλά σύνθετα, αλληλοσχετιζόμενα γεγονότα να συμβαίνουν ταυτόχρονα, το καθένα με τη δική του εσωτερική ακολουθία συμβάντων και με κάποια μορφή επικοινωνίας ή συγχρονισμού μεταξύ των γεγονότων.

Η παράλληλη επεξεργασία προσπαθεί να εκμεταλλευτεί όσο γίνεται πιο αποδοτικά τους υπολογιστικούς πόρους, έτσι ώστε με τη βοήθεια ειδικού λογισμικού να επιτύχει μέγιστη επεξεργασία δεδομένων στον ελάχιστο χρόνο.

Οι υπολογιστικοί πόροι μπορεί να είναι ένας υπολογιστής με πολλούς επεξεργαστές ή ένας επεξεργαστής με πολλούς πυρήνες. Η σύνδεση μεταξύ τους καθώς και με τη κεντρική μνήμη επιτυγχάνεται είτε μέσω εξειδικευμένου δικτύου διασύνδεσης, είτε μέσω τυπικού διαύλου. Επίσης ένα παράλληλο υπολογιστικό περιβάλλον μπορεί να αποτελείται από πολλούς υπολογιστές διασυνδεδεμένους είτε σε συστοιχίες μέσω εξειδικευμένης διασύνδεσης υψηλής απόδοσης και κοινά προσπελάσιμους δίσκους, είτε σε τυπικό τοπικό δίκτυο ή ακόμη και μέσω διαδικτύου.

Στο πιο σημαντικό υπολογιστικό μοντέλο παράλληλης επεξεργασίας διακρίνονται δύο αρχιτεκτονικές με βάση την οργάνωση της μνήμης· οι υπολογιστές με κοινή μνήμη (shared memory), οι οποίοι καλούνται και πολυεπεξεργαστές (multiprocessors) και υπολογιστές με κατανεμημένη μνήμη (distributed memory), οι οποίοι καλούνται και πολυυπολογιστές (multicomputers).

Ωστόσο, ο παραλληλισμός επηρεάζει το σύνολο ενός υπολογιστικού συστήματος, το οποίο περιλαμβάνει παράλληλες αρχιτεκτονικές, λειτουργικά συστήματα, συστήματα εκτέλεσης που υποστηρίζουν τον παραλληλισμό, παράλληλες γλώσσες, βιβλιοθήκες

προγραμματισμού, παράλληλους αλγορίθμους και μεθοδολογίες ανάπτυξης λογισμικού και εφαρμογών.

2. ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ ΚΑΙ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Ο αποδοτικός προγραμματισμός αποτελεί κρίσιμο παράγοντα για την επίτευξη βελτιωμένης απόδοσης των κατανεμημένων εφαρμογών και την εκμετάλλευση των υπολογιστικών υποδομών σε περιβάλλοντα παράλληλου προγραμματισμού. Μια εφαρμογή θεωρείται ως μια ομάδα αλληλένδετων εργασιών, που πρέπει να κατανεμηθούν κατάλληλα στους επεξεργαστές, λαμβάνοντας υπόψη τον χρόνο που απαιτείται, τόσο για την εκτέλεσή τους όσο και για την επικοινωνία μεταξύ των επεξεργαστών. Ο προγραμματισμός πρέπει να εκτελείται με τέτοιο τρόπο, ώστε να ελαχιστοποιεί το χρόνο ολοκλήρωσης του παράλληλου κώδικα, γεγονός που εγγυάται οικονομικά οφέλη καθώς και οφέλη για το περιβάλλον.

Υπάρχει μια ποικιλία μεθόδων που υποστηρίζουν τον αποτελεσματικό χρονοπρογραμματισμό μιας εφαρμογής σε ένα σύστημα πολυεπεξεργαστών. Όλες αυτές οι μέθοδοι, παρά τις διαφορές τους, εξυπηρετούν ένα κοινό σκοπό, την διάταξη των διαδοχικών τμημάτων μιας εφαρμογής σε μια σειρά, λαμβάνοντας υπόψη τις αμοιβαίες τους εξαρτήσεις.

Οι εξαρτήσεις μεταξύ διαδοχικών τμημάτων αντιπροσωπεύουν τη μεταφορά δεδομένων. Τα δεδομένα αυτά υπολογίζονται σε προηγούμενα τμήματα και χρησιμοποιούνται ως εισροές από τα επόμενα.

Οι σχέσεις μεταξύ των διαδοχικών τμημάτων μπορούν να μοντελοποιηθούν ως μη κυκλικός γράφος, στον οποίο όλες οι εξαρτήσεις πηγαίνουν από το παρελθόν στο μέλλον. Ένα τέτοιο μοντέλο ονομάζεται κατευθυνόμενος μη κυκλικός γράφος (DAG) και περιγράφει την εκτέλεση παράλληλων εφαρμογών. Ένα DAG μπορεί να περιέχει έναν μεγάλο αριθμό εργασιών που αντιπροσωπεύουν σύνθετα πραγματικά σενάρια.

Είναι γνωστό ότι, όταν η δομή μιας εφαρμογής περιγράφεται από ένα DAG, ο προγραμματισμός της σε ένα σύστημα πολυεπεξεργασίας είναι ένα πρόβλημα NP-πλήρες στη γενική του μορφή (Kwok και Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999).

Επιπλέον, για τον προσδιορισμό του βέλτιστου χρονοδιαγράμματος σε πολυωνυμικό χρόνο υπάρχουν περιορισμένα μοντέλα και δεν ενδείκνυται η χρήση τους στον προγραμματισμό πραγματικών παράλληλων εφαρμογών.

Οι δημοφιλείς τεχνικές χρονοπρογραμματισμού δεν επιδιώκουν την βέλτιστη λύση (Kwok and Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999). Αυτό που επιχειρούν είναι το να διατάξουν τους κόμβους ενός DAG σε μια σειρά τέτοια, ώστε αφενός μεν να λαμβάνονται υπόψη οι διαδοχικές τους εξαρτήσεις, αφετέρου δε να δίνεται προτεραιότητα σε μια εργασία που κρίνεται πιο σημαντική για τη συνολική πρόοδο της εκτέλεσης. Στην προσπάθεια αυτή, απαιτούνται εκτεταμένες γνώσεις σχετικά με τη δομή του προγράμματος. Επιπλέον, η αποτελεσματικότητα των διάφορων αλγορίθμων προγραμματισμού χαρακτηρίζεται από μεγάλη διακύμανση. Προκειμένου ο χρόνος εκτέλεσης του προγράμματος να είναι μικρότερος, κάποιες τεχνικές αυξάνουν, ενδεχομένως, την πολυπλοκότητα του αλγορίθμου, ενώ άλλες τεχνικές προγραμματισμού μπορεί να βελτιώνουν άλλες παραμέτρους, όπως την υψηλή βιωσιμότητα ως προς τις διακυμάνσεις στο περιβάλλον εκτέλεσης, την ομοιόμορφη διαχείριση κλπ (Kwok και Ahmad, Benchmarking and comparison of the task graph scheduling algorithms 1999). Μια άλλη κατηγορία αλγορίθμων επιλέγει πρότυπα, τα οποία δεν επενδύουν στην αναζήτηση πληροφοριών σε σχέση με τη δομή του προβλήματος, αλλά βελτιώνουν την απόδοση σε ένα δεδομένο περιβάλλον εκτέλεσης (Planeta 2015).

2.1 Ομογενή και ετερογενή συστήματα

Οι περισσότερες προσεγγίσεις προγραμματισμού DAG υποθέτουν ότι το σύστημα είναι ομογενές. Ωστόσο, μια ενιαία αρχιτεκτονική μηχανών με το σχετικό μεταγλωττιστή, το λειτουργικό σύστημα και τα εργαλεία προγραμματισμού, δηλαδή ένα ομογενές σύστημα, είναι αδύνατο να ικανοποιήσει εξίσου καλά όλες τις υπολογιστικές απαιτήσεις μιας εφαρμογής.

Κατά την παράλληλη εκτέλεση ενός προγράμματος, σύμφωνα με το νόμο του Amhdal, το τμήμα το οποίο εκτελείται σειριακά είναι αυτό που προκαλεί την καθυστέρηση (Planeta 2015). Αν ένα σύνολο επεξεργαστών ενός ομογενούς συστήματος αντικατασταθούν με ταχύτερους επεξεργαστές και ανατεθούν σ’ αυτούς οι σειριακοί υπολογισμοί ή άλλοι

κρίσιμοι υπολογισμοί, η ταχύτητα εκτέλεσης μπορεί να αυξηθεί. Το μοντέλο που προκύπτει στην περίπτωση αυτή, ως επέκταση του ομογενούς, αποτελεί ένα ετερογενές σύστημα επεξεργαστών.

Γενικά, ένα ετερογενές υπολογιστικό περιβάλλον που αποτελείται από μια ετερογενή σειρά μηχανών, διασυνδέσεις υψηλής ταχύτητας, λειτουργικά συστήματα, πρωτόκολλα επικοινωνίας και περιβάλλοντα προγραμματισμού παρέχει μια ποικιλία αρχιτεκτονικών δυνατοτήτων, οι οποίες μπορούν να ενορχηστρωθούν για την εκτέλεση μιας εφαρμογής που έχει διαφορετικές απαιτήσεις εκτέλεσης (Kwok και Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999).

Ωστόσο, η ετερογένεια σε ένα σύστημα επεξεργαστών, αναπόφευκτα, καθιστά το πρόβλημα πιο περίπλοκο. Επιπλέον, ένα πρόγραμμα που προκύπτει για ένα δεδομένο ετερογενές σύστημα, καθίσταται αμέσως άκυρο στην περίπτωση που κάποιοι από τους επεξεργαστές αντικατασταθούν από άλλους διαφορετικών δυνατοτήτων.

2.2 Μέθοδοι παραλληλοποίησης προγραμμάτων

Η παραλληλοποίηση προγραμμάτων είναι ένας φυσικός τρόπος, τόσο για την επιτάχυνση της διαδικασίας εκτέλεσης, όσο και για την μείωση της πολυπλοκότητας και την βελτίωση της ποιότητας της παραγόμενης λύσης. Επί πλέον, ο προγραμματισμός DAG σε ετερογενείς υπολογιστικές πλατφόρμες παρέχει ένα ακόμη υψηλότερο επίπεδο παραλληλισμού.

Στις περισσότερες περιπτώσεις, η παραλληλοποίηση προγραμμάτων πραγματοποιείται χειροκίνητα από τον σχεδιαστή τους. Μια δημοφιλής προσέγγιση για τον σκοπό αυτό είναι η ανταλλαγή μηνυμάτων (message-passing systems) μεταξύ των επεξεργαστών του παράλληλου συστήματος. Ο σχεδιαστής αποφασίζει ποια μέρη του προγράμματος πρέπει να εκτελεστούν ταυτόχρονα, ορίζοντας έτσι τα υποσύνολα εργασιών. Στη συνέχεια εισαγάγει τις οδηγίες μετάβασης μηνυμάτων για το συγχρονισμό και την επικοινωνία μεταξύ αυτών των εργασιών (Sinnen 2007).

Εκτός της μεθόδου message-passing systems, άλλα παραδείγματα μεθόδων παραλληλοποίησης αποτελούν τα fork-join pattern, map-reduce frameworks, future-based parallelism και asynchronous lambda approaches (Planeta 2015).

Ωστόσο, η εύρεση ενός βέλτιστου χρονοδιαγράμματος για παράλληλα προγράμματα είναι μια μη τετριμμένη εργασία και θεωρείται NP-πλήρης. Οι περισσότεροι από τους αλγορίθμους προγραμματισμού, για την ανάθεση εργασιών σε επεξεργαστές ετερογενών συστημάτων, χρησιμοποιούν άπληστη προσέγγιση (Khan 2012), κάνοντας σε κάθε βήμα επιλογή τοπικά βέλτιστη.

2.3 JIT αλγόριθμοι και full-ahead αλγόριθμοι

Στο μοντέλο που χαρακτηρίζεται Just-in-Time (JIT) δεν είναι γνωστή η δομή του προγράμματος εκ των προτέρων. Θεωρείται ότι, σε κάθε χρονική στιγμή, υπάρχουν πολύ περιορισμένες αλλά έγκυρες πληροφορίες και οι αποφάσεις λαμβάνονται και εκτελούνται άμεσα. Έχει το πλεονέκτημα να είναι απλούστερο και επομένως ευρέως διαδεδομένο σε χρήση στα πραγματικά συστήματα.

Στο μοντέλο Just-in-Time, όπως και στο μοντέλο full-ahead, το πρόγραμμα θεωρείται ότι αποτελείται από διαδοχικά τμήματα που ονομάζονται εργασίες και έχουν διαδοχικές εξαρτήσεις μεταξύ τους. Εργασίες με σύνολα εξαρτήσεων ξένα μεταξύ τους μπορούν να εκτελούνται παράλληλα σε διαφορετικούς επεξεργαστές. Το παράλληλο πρόγραμμα μπορεί να μοντελοποιηθεί ως ένα DAG με τους κόμβους και τις ακμές να απεικονίζουν τις εργασίες και τις εξαρτήσεις αντίστοιχα. Κάθε εργασία η οποία εκτελείται σε έναν επεξεργαστή χαρακτηρίζεται ενεργή. Αν η εκτέλεση μιας εργασίας Β εξαρτάται από την εκτέλεση μιας εργασίας Α, τότε οι Α και Β έχουν σχέση γονέα - παιδιού. Κάθε εργασία ανατίθεται για εκτέλεση όταν είναι έτοιμη, δηλαδή όταν ικανοποιούνται οι εξαρτήσεις της. Ο άπληστος αλγόριθμος είναι μια πολύ απλή Just-in-Time εφαρμογή.

Το μοντέλο Just-in-Time θεωρεί ότι υπάρχει μοναδική καθολικά προσπελάσιμη λίστα εργασιών, η οποία βρίσκεται σε κατάσταση ετοιμότητας και ονομάζεται έτοιμη ουρά. Ένας επεξεργαστής στον οποίο δεν εκτελείται κάποια εργασία (ενεργή εργασία) «τραβάει» την επόμενη από την έτοιμη ουρά. Δεδομένου ότι οι απαιτήσεις της καθολικά προσπελάσιμης ουράς προκαλούν καθυστέρηση στο δίκτυο, το μοντέλο - αν και απλό - καθίσταται μη ρεαλιστικό.

Το καθολικό μοντέλο Just-in-Time συχνά αγωνίζεται να επιτύχει υψηλή συνολική απόδοση σε μεγάλης κλίμακας συστήματα. Οι Anderson, Lazowska και Levy έχουν δείξει

ότι η διαμάχη για τον δίαυλο συστήματος μπορεί να μειώσει δραστικά, τόσο την καθυστέρηση όσο και την απόδοση του συστήματος. Επίσης, διάφορες έρευνες εξετάζουν την επίδραση της θέσης των δεδομένων στον αριθμό των αποτυχιών της cache μνήμης του επεξεργαστή καθώς και στα σφάλματα σελίδας. Λόγω της θέσης των δεδομένων, μπορεί να αυξηθεί η συμφόρηση στον δίαυλο διαμοιραζόμενης μνήμης και να μειωθεί σημαντικά η απόδοση. Αντίθετα, αν οι εργασίες τείνουν να παραμείνουν στον ίδιο επεξεργαστή με τις γονικές τους, αυξανομένου του αριθμού των επεξεργαστών η απόδοση μειώνεται βραδύτερα.

Στο μοντέλο Just-in-Time βασίζονται αρκετοί αλγόριθμοι, μεταξύ των οποίων οι HEFT, Min-min, Max-min, Sufferage. Οι αλγόριθμοι αυτοί λειτουργούν κυρίως σε ετερογενή περιβάλλοντα.

Το μοντέλο παράλληλου προγραμματισμού που βασίζεται σε νήματα και mutexes μπορεί να θεωρηθεί Just-in-Time μοντέλο. Τα κλειδώματα mutex μπορούν να θεωρηθούν ως σημεία συγχρονισμού σε αντιστοιχία με την δημιουργία εργασιών στο full-ahead μοντέλο. Ο προγραμματιστής γνωρίζει μόνο τα νήματα και τα mutexes που μπλοκάρουν ορισμένα νήματα, αλλά δεν γνωρίζει ποιο mutex ή νήμα έπεται. Ο προγραμματισμός γίνεται κατά την εκτέλεση, δηλαδή σε πραγματικό χρόνο, με την επιλογή ενός νήματος το οποίο δεν μπλοκάρεται από κάποιο mutex.

Εάν η δομή του προγράμματος είναι γνωστή εκ των προτέρων (full-ahead μοντέλο), μπορούν να αναπτυχθούν πιο περίπλοκοι αλγόριθμοι οι οποίοι διαφέρουν όχι μόνο στην προτεινόμενη προσέγγιση αλλά και στις λεπτομέρειες των μοντέλων στα οποία εφαρμόζονται.

Εκτός από τη δομή του DAG, σημαντικό χαρακτηριστικό για το μοντέλο του αλγορίθμου είναι οι πληροφορίες που είναι δεδομένες και αφορούν τις εργασίες και την μεταξύ τους επικοινωνία. Στην απλούστερη περίπτωση, μπορούμε να υποθέσουμε ότι υπάρχουν μόνο περιορισμοί προτεραιότητας. Σ’ αυτή την περίπτωση το υπολογιστικό κόστος των εργασιών θεωρείται μοναδιαίο. Αν το υπολογιστικό κόστος διαφοροποιείται, το βάρος του κόμβου μπορεί να έχει αυθαίρετη τιμή, η οποία περιγράφει τον χρόνο που απαιτείται για την ολοκλήρωση της εργασίας σε έναν επεξεργαστή. Αν το κόστος επικοινωνίας είναι αδιάφορο, το βάρος ακμής είναι είτε μηδενικό, είτε κοινό για κάθε

ακμή. Στον χώρο της έρευνας, υπάρχει ένα τυπικό μοντέλο που προτάθηκε από τους Rayward-Smith (1987), το UET-UCT (unit estimated time-unit communication time), το οποίο ορίζει τα κόστη υπολογισμού και επικοινωνίας να είναι μοναδιαίου βάρους (Planeta 2015).

Το μοντέλο macro dataflow λειτουργεί με αυθαίρετες τιμές για τα κόστη υπολογισμού και επικοινωνίας των εργασιών. Κάθε εργασία εκτελείται αποκλειστικά σε έναν συγκεκριμένο επεξεργαστή για χρονικό διάστημα, το οποίο εξαρτάται από το κόστος υπολογισμού της εργασίας και έχει αυθαίρετη πεπερασμένη τιμή. Επίσης, κάθε εργασία, πριν την έναρξή της, λαμβάνει δεδομένα από την γονική της. Ο απαιτούμενος χρόνος για την επίτευξη αυτής της λειτουργίας εξαρτάται από το κόστος επικοινωνίας των εργασιών και έχει επίσης αυθαίρετη πεπερασμένη τιμή που ισούται με το μηδέν στην περίπτωση που γονική και θυγατρική εκτελούνται στον ίδιο επεξεργαστή.

Η υπόθεση μηδενικού κόστους επικοινωνίας, λόγω εκτέλεσης στον ίδιο επεξεργαστή γονικής και θυγατρικής εργασίας, δικαιολογείται από το γεγονός ότι ο ρυθμός μετάδοσης στο δίκτυο είναι πολύ χαμηλότερος από τον ρυθμό μετάδοσης στην τοπική μνήμη. Επιπλέον, υπάρχει και η περίπτωση κατά την οποία δεδομένα που αποτελούν εκροές της μιας εργασίας να αποτελούν εισροές της άλλης, οπότε δεν χρειάζεται να μεταφερθούν. Ως εκ τούτου, το μοντέλο macro dataflow, αν και απλό, προσεγγίζει ικανοποιητικά την εκτέλεση του παράλληλου προγράμματος σε ένα σύστημα πολλαπλών επεξεργαστών.

Τα σύγχρονα συστήματα που είναι συχνά ετερογενή, η ετερογένεια μπορεί να αφορά τους επεξεργαστές, την μεταξύ τους επικοινωνία ή να συνδυάζει και τους δύο τομείς. Στην πρώτη περίπτωση, αντί για κόμβους κοινού βάρους, για κάθε εργασία ορίζεται πίνακας με τους χρόνους εκτέλεσής της σε κάθε επεξεργαστή του συστήματος. Στη δεύτερη περίπτωση, διαφορετικά κανάλια επικοινωνίας μεταξύ των επεξεργαστών, τα οποία πληρούν τις απαιτήσεις εξάρτησης μιας εργασίας, αντιστοιχούν σε διαφορετικούς χρόνους μεταφοράς των δεδομένων.

Το απρόβλεπτο του συστήματος μπορεί να προσεγγισθεί με διάφορους τρόπους, όπως με την υπερεκτίμηση των χρόνων υπολογισμού και επικοινωνίας. Οι αλγόριθμοι Just-in-Time, οι οποίοι λαμβάνουν αποφάσεις βασιζόμενοι στις πληροφορίες που είναι

διαθέσιμες στην τρέχουσα χρονική στιγμή, διαθέτουν μια φυσική ανοχή σε απροσδόκητα δεδομένα που αφορούν τους χρόνους εκτέλεσης εργασιών. Ωστόσο, σε περιπτώσεις κατά τις οποίες η εκτέλεση δεν αντιμετωπίζει αβεβαιότητες, οι δυναμικοί αλγόριθμοι έχουν χειρότερη απόδοση σε σύγκριση με τους βασιζόμενους σε DAG.

Το dynamic task rescheduling είναι μια υβριδική προσέγγιση, η οποία συνδυάζει χρονοπρογραμματισμό βασιζόμενο σε μια εκ των προτέρων γνώση του DAG αλλά και στην τρέχουσα πληροφορία. Σε περιπτώσεις αβεβαιότητας, τέτοιοι αλγόριθμοι τείνουν να είναι αποδοτικότεροι αν βασίζονται στην καλύτερη γνώση του DAG.

Το stochastic scheduling είναι μια προσέγγιση για περιπτώσεις που η φύση του απρόβλεπτου είναι γνωστή, για παράδειγμα ορίζονται οι αναμενόμενοι χρόνοι εκτέλεσης, καθώς και οι αποκλίσεις των αναμενόμενων χρόνων εκτέλεσης των εργασιών. Αυτοί οι επιλυτές αναθέτουν προτεραιότητες με βάση το επίπεδο αβεβαιότητας του χρόνου εκτέλεσης της κάθε εργασίας. Στο τέλος προετοιμάζουν μια λίστα προτεραιότητας των εργασιών, η οποία αναμένεται να διασφαλίζει μεγαλύτερη αποδοτικότητα από έναν τυπικό αλγόριθμο χρονοπρογραμματισμού (Planeta 2015).

3. ΑΠΕΙΚΟΝΙΣΗ ΠΑΡΑΛΛΗΛΩΝ ΕΦΑΡΜΟΓΩΝ ΜΕ ΚΑΤΕΥΘΥΝΟΜΕΝΟΥΣ ΜΗ ΚΥΚΛΙΚΟΥΣ ΓΡΑΦΟΥΣ

3.1 Βασικές έννοιες γράφων

Ορίζουμε ένα κατευθυνόμενο μη κυκλικό γράφο (DAG) ως ένα διατεταγμένο ζεύγος $G = (V, E)$, όπου V είναι ένα σύνολο του οποίου τα στοιχεία λέγονται κόμβοι και E είναι σύνολο του οποίου τα στοιχεία λέγονται ακμές. Οι κόμβοι $v_i \in V$ παριστάνουν τα διαδοχικά τμήματα του προγράμματος, τα οποία ονομάζονται εργασίες. Οι όροι κόμβος και εργασία χρησιμοποιούνται ισοδύναμα. Κάθε κόμβος εκτός από τον κόμβο εισόδου έχει τουλάχιστον έναν γονικό. Οι ακμές $e_{i,j} \in E$ για τις διάφορες τιμές των i και j παριστάνουν τις εξαρτήσεις δεδομένων, οι οποίες επηρεάζουν τις επικοινωνίες μεταξύ των εργασιών v_i και v_j, έτσι ώστε η μία εργασία να μην μπορεί να ξεκινήσει την εκτέλεσή της μέχρις ότου ολοκληρώσει την εκτέλεσή της η άλλη. Τα σύμβολα v και e εκφράζουν το πλήθος στοιχείων για καθένα από τα σύνολα V και E αντίστοιχα. Το περιβάλλον εκτέλεσης αποτελείται από το σύνολο P των διασυνδεδεμένων επεξεργαστών, όπου $p_i \in P$ είναι στοιχεία του συνόλου P και p το πλήθος των επεξεργαστών στο σύνολο P.

Οι εξαρτήσεις των εργασιών καθορίζονται από τους τελεστές $pred(v_i)$ και $succ(v_i)$. Ο τελεστής $pred(v_i)$ δίνει τη λίστα των εργασιών οι οποίες θέτουν εξαρτήσεις δεδομένων στην εργασία v_i, ενώ ο τελεστής $succ(v_i)$ δίνει τη λίστα των εργασιών των οποίων η εκτέλεση εξαρτάται από την εργασία v_i. Όταν μια εργασία v_i έχει προγραμματιστεί, ο τελεστής $p(v_i)$ παριστάνει τον επεξεργαστή στον οποίο έχει ανατεθεί η εκτέλεση της εργασίας v_i.

Το υπολογιστικό κόστος για την εργασία v_i παρίσταται από τον τελεστή $w(v_i)$. Το κόστος επικοινωνίας για τη μεταφορά δεδομένων από τον κόμβο v_i στον κόμβο v_j παρίσταται από τον τελεστή $c_{i,j}$. Στην περίπτωση που οι εργασίες v_i και v_j έχουν προγραμματιστεί στον ίδιο επεξεργαστή, ισχύει $c_{i,j} = 0$. Τα κόστη υπολογισμού και επικοινωνίας χαρακτηρίζονται ως βάρη.

Χωρίς περιορισμό της γενικότητας, υποθέτουμε ότι ένα DAG έχει έναν μόνο κόμβο εισόδου και έναν μόνο κόμβο εξόδου. Όλοι οι κόμβοι, με εξαίρεση τους κόμβους

εισόδου και εξόδου και όλες οι ακμές έχουν θετικά βάρη. Οι κόμβοι εισόδου και εξόδου μπορούν να έχουν μηδενικά βάρη.

Το μοντέλο ετερογενούς συστήματος είναι μια επέκταση του ομογενούς. Υπάρχουν πολλές παραλλαγές αυτού του μοντέλου, αλλά η βασική διαφορά είναι ότι το κόστος υπολογισμού ενός κόμβου v_i δεν είναι καθαρός αριθμός $w(v_i)$, αλλά ένα διάνυσμα $w(v_i, p_j)$. Το διάνυσμα $w(v_i, p_j)$ αντιπροσωπεύει το κόστος υπολογισμού του κόμβου v_i σε έναν επεξεργαστή p_j.

Πριν τον προγραμματισμό και την ανάθεση των εργασιών σε συγκεκριμένους επεξεργαστές, οι εργασίες χαρακτηρίζονται από τα μέσα κόστη υπολογισμού, τα οποία για κάθε εργασία v_i δίνονται από τον τύπο:

$$\overline{w(v_i)} = \left(\sum_{j=1}^{p} w(v_i, p_j)\right)/p$$

όπου p το πλήθος των επεξεργαστών.

Αν θεωρήσουμε ένα μοντέλο επικοινωνίας που αποτελείται από ένα δίκτυο πλήρως διασυνδεδεμένων επεξεργαστών, τα κόστη επικοινωνίας δημιουργούνται από δύο παραμέτρους:

1. Έναν *pxp* πίνακα B, καθένα από τα στοιχεία *($b_{i,j}$)* του οποίου εκφράζει τον ρυθμό μεταφοράς δεδομένων από τον επεξεργαστή p_i στον επεξεργαστή p_j .
2. Ένα p-διάστατο διάνυσμα L, καθένα από τα στοιχεία *(L_i)* του οποίου εκφράζει την καθυστέρηση του επεξεργαστή p_i .

Το βάρος των ακμών στο DAG, αντί του χρόνου που απαιτείται για τη μετάδοση δεδομένων, μπορεί να δείχνει το μέγεθος των δεδομένων που πρέπει να μεταδοθούν. Συνδυάζοντας τις παραμέτρους του δικτύου και το βάρος της ακμής, το κόστος επικοινωνίας για τη μετάδοση δεδομένων από τον κόμβο v_i (που εκτελείται στον επεξεργαστή $p(v_i)$) στον κόμβο v_j (που εκτελείται στον επεξεργαστή $p(v_j)$) δίνεται από τον τύπο:

$$c_{i,j} = L_{p(v_i)} + \frac{data_{i,j}}{b_{p(v_i),p(v_j)}}$$

Στην περίπτωση που οι κόμβοι v_i και v_j εκτελούνται στον ίδιο επεξεργαστή, ισχύει:

$$c_{i,j} = 0.$$

Πριν τον προγραμματισμό και την ανάθεση των εργασιών σε συγκεκριμένους επεξεργαστές, οι ακμές χαρακτηρίζονται από τα μέσα κόστη επικοινωνίας, τα οποία για κάθε ακμή $c_{i,j}$ δίνονται από τον τύπο:

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{b}}$$

όπου $\overline{b}$ ο μέσος ρυθμός μετάδοσης των επεξεργαστών και $\overline{L}$ η μέση χρονική υστέρηση επικοινωνίας (Topcuoglu, Hariri και Wu 2002).

Ορίζουμε, επίσης, τον λόγο CCR (Communication to Computation Ratio) ενός παράλληλου προγράμματος ως το μέσο κόστος επικοινωνίας προς το μέσο υπολογιστικό κόστος σε δεδομένο σύστημα. Αυτή η παράμετρος είναι ένα σημαντικό χαρακτηριστικό για ένα παράλληλο πρόγραμμα και ποικίλλει για διαφορετικές εφαρμογές. Εάν η τιμή CCR ενός DAG είναι πολύ χαμηλή, μπορεί να θεωρηθεί ως μια εφαρμογή υπολογιστικής έντασης (Ilavarasan και Thambidurai 2007).

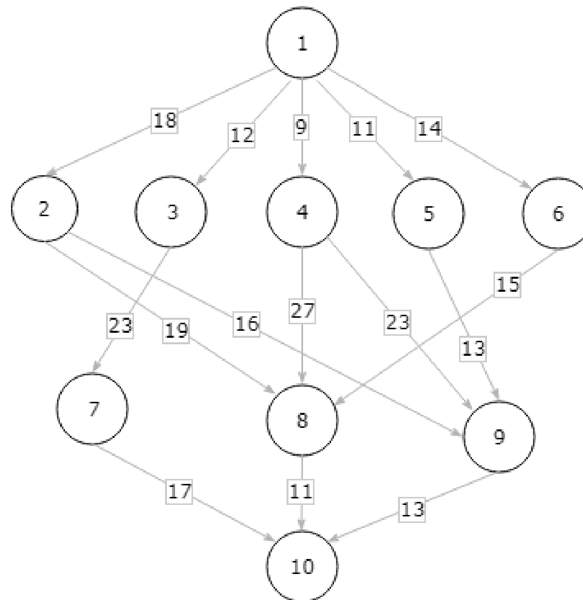
3.2 Αναπαράσταση γράφων στον Η/Υ

Ένας γράφος μπορεί να παρασταθεί με διάφορους τρόπους, καθένας με τα πλεονεκτήματα και τα μειονεκτήματά του. Ορισμένοι αλγόριθμοι που δέχονται γράφους ως είσοδο, απαιτούν μία αναπαράσταση και άλλοι απαιτούν μία διαφορετική. Τα κριτήρια, με τα οποία καθορίζονται τα πλεονεκτήματα και τα μειονεκτήματα ενός τρόπου αναπαράστασης, αναφέρονται τόσο στην μνήμη που απαιτείται για την αναπαράσταση, όσο και στον απαιτούμενο χρόνο εκτέλεσης για να καθοριστεί αν μια συγκεκριμένη ακμή υπάρχει στον γράφο ή για να βρεθούν οι γειτνιάζοντες ενός κόμβου.

3.2.1 Λίστες ακμών

Μια λίστα ακμών είναι ένας απλός τρόπος για την αναπαράσταση ενός γράφου. Πρόκειται για μια λίστα ή έναν πίνακα, κάθε στοιχείο του οποίου είναι ένα ζεύγος κόμβων που ορίζουν μία ακμή. Αν οι ακμές έχουν βάρη, για κάθε ακμή ορίζεται ένα τρίτο

στοιχείο, το βάρος. Προκειμένου να καθοριστεί αν μια συγκεκριμένη ακμή υπάρχει στον γράφο, απαιτείται σειριακή αναζήτηση της λίστας ακμών.



Σχήμα 3.1. Γράφος 10 εργασιών με κόστη ακμών (Valouxis, et al. 2013)

[[1, 2, 18], [1, 3, 12], [1, 4, 9], [1, 5, 11], [1, 6, 14], [2, 8, 19], [2, 9, 16], [3, 7, 23], [4, 8, 27], [4, 9, 23], [5, 9, 13], [6, 8, 15], [7, 10, 17], [8, 11, 10], [9, 10, 13]]

Σχήμα 3.2. Λίστα ακμών με κόστη – αναπαράσταση του γράφου του σχήματος 3.1.

3.2.2 Πίνακες γειτνίασης

Για έναν γράφο $G = (V, E)$, όπου V είναι το σύνολο των κόμβων του, με πλήθος στοιχείων v, ορίζουμε ως πίνακα γειτνίασης (adjacency matrix) έναν πίνακα $v \times v$ με στοιχεία μηδενικά και μονάδες. Το στοιχείο του πίνακα με δείκτες (i, j) έχει την τιμή 1 στην περίπτωση που στον γράφο υπάρχει ακμή από τον κόμβο v_i στον κόμβο v_j. Στην αντίθετη περίπτωση, το στοιχείο με δείκτες (i, j) έχει την τιμή 0. Αν οι ακμές έχουν βάρη, στη θέση των μονάδων τίθενται τα αντίστοιχα βάρη, ενώ η απουσία ακμών δηλώνεται στον πίνακα με μια τιμή null ή το άπειρο. Προκειμένου να καθοριστεί αν μια συγκεκριμένη ακμή υπάρχει στον γράφο, αρκεί να ελέγξουμε την αντίστοιχη καταχώρηση στον πίνακα γειτνίασης. Τα μειονεκτήματα του πίνακα γειτνίασης, ως τρόπου αναπαράστασης γράφων, είναι ότι αφενός μεν καταλαμβάνει μεγάλο χώρο

ακόμη και για αραιούς πίνακες (όπου πλειοψηφούν τα μηδενικά), αφετέρου δε, προκειμένου να βρεθούν οι γειτνιάζοντες ενός κόμβου, πρέπει να ελεγχθούν όλα τα στοιχεία της αντίστοιχης γραμμής του πίνακα. Ένας πίνακας γειτνίασης είναι συμμετρικός αν και μόνο αν ο γράφος είναι μη κατευθυνόμενος.

	1	2	3	4	5	6	7	8	9	10
1	∞	18	12	9	11	14	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞	19	16	∞
3	∞	∞	∞	∞	∞	∞	23	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	27	23	∞
5	∞	∞	∞	∞	∞	∞	∞	∞	13	∞
6	∞	∞	∞	∞	∞	∞	∞	15	∞	∞
7	∞	∞	∞	∞	∞	∞	∞	∞	∞	17
8	∞	∞	∞	∞	∞	∞	∞	∞	∞	11
9	∞	∞	∞	∞	∞	∞	∞	∞	∞	13
10	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Σχήμα 3.3. Πίνακας γειτνίασης - αναπαράσταση του γράφου του σχήματος 3.1.

3.2.1 Λίστες γειτνίασης

Οι λίστες γειτνίασης (adjacency list), ως τρόπος για την αναπαράσταση ενός γράφου, αποτελούν ένα συνδυασμό των πινάκων γειτνίασης και των λιστών ακμών. Για κάθε κόμβο v_i του γράφου αποθηκεύεται ένας πίνακας που περιέχει όλους τους γειτονικούς κόμβους του v_i. Τυπικά, για v πλήθους κόμβους ορίζονται v πλήθους λίστες γειτνίασης, μία για κάθε κόμβο. Θεωρούμε τους κόμβους αριθμημένους. Οι αριθμοί κόμβων σε μια λίστα γειτνίασης δεν απαιτείται να διατάσσονται με κάποια συγκεκριμένη σειρά, αν και είναι συχνά βολικό να διατάσσονται σε αύξουσα σειρά. Στην περίπτωση που οι ακμές έχουν βάρη, κάθε στοιχείο μιας λίστας γειτνίασης μπορεί να είναι ένας πίνακας δύο στοιχείων, του αριθμού του θυγατρικού κόμβου και του βάρους της αντίστοιχης ακμής. Το πλεονέκτημα των λιστών γειτνίασης είναι ότι μας επιτρέπουν να παραστήσουμε συμπαγώς έναν αραιό γράφο. Η λίστα γειτνίασης μας επιτρέπει επίσης να βρούμε εύκολα όλους τους θυγατρικούς ενός συγκεκριμένου κόμβου. Προκειμένου να καθοριστεί αν μια συγκεκριμένη ακμή υπάρχει στον γράφο, αρκεί να ελέγξουμε την αντίστοιχη λίστα γειτνίασης(Cormen και Balkcom n.d.).

1	→	2, 18	3, 12	4, 9	5, 11	6, 14
2	→	8, 19	9, 16			
3	→	7, 23				
4	→	8, 27	9, 23			
5	→	9, 13				
6	→	8, 15				
7	→	10, 17				
8	→	10, 11				
9	→	10, 13				

Σχήμα 3.4. Λίστα γειτνίασης με κόστη – αναπαράσταση του γράφου του σχήματος 3.1.

3.3 Γράφοι εργασιών (task graphs)

Ο προγραμματισμός ενός γράφου εργασιών είναι η δραστηριότητα που συνίσταται στη χαρτογράφηση του γράφου εργασιών σε μια πλατφόρμα προορισμού. Ο γράφος εργασιών είναι ένα DAG, δίνεται ως είσοδος στον επιλυτή και αντιπροσωπεύει την εφαρμογή. Οι κόμβοι υποδηλώνουν τις υπολογιστικές εργασίες, ενώ οι ακμές τους περιορισμούς προτεραιότητας μεταξύ των εργασιών. Για κάθε εργασία, προσδιορίζεται μια ανάθεση η οποία καθορίζει τον επεξεργαστή (όπου θα εκτελεστεί η εργασία) και ένα χρονοδιάγραμμα (που αφορά την χρονική στιγμή έναρξης της εκτέλεσης). Αυτές οι εργασίες συνδέονται με περιορισμούς προτεραιότητας. Για παράδειγμα, αν μια εργασία v_i παράγει κάποια αποτελέσματα που χρησιμοποιούνται (ως εισροές) από κάποιες άλλες εργασίες, η εκτέλεση αυτών δεν μπορεί να ξεκινήσει πριν την ολοκλήρωση της v_i (Robert 2011).

Το γεγονός ότι ο γράφος είναι μη κυκλικός σημαίνει ότι οι εργασίες δεν μπορούν να επιβάλλουν περαιτέρω εξαρτήσεις στις γονικές τους. Η μη κυκλική φύση του γράφου είναι σημαντική καθώς καταργεί την πιθανότητα αδιεξόδων μεταξύ των εργασιών, υπό την προϋπόθεση ότι οι εργασίες είναι πραγματικά ανεξάρτητες. Επομένως, οι εργασίες μπορούν να εκτελούνται παράλληλα ή σειριακά, ανάλογα με το πότε θα είναι διαθέσιμες οι εισροές (Miller, και συν. n.d.).

Οι ανεξάρτητες εργασίες μπορούν να εκτελούνται παράλληλα. Μια απαραίτητη συνθήκη ώστε οι εργασίες να είναι ανεξάρτητες, είναι το να μην ενημερώνουν την ίδια μεταβλητή

– μπορούν να διαβάζουν την ίδια τιμή, αλλά δεν μπορούν να γράφουν στην ίδια θέση μνήμης. Στην αντίθετη περίπτωση θα προκληθεί διένεξη και ακαθόριστο αποτέλεσμα (Robert 2011).

Ο στόχος είναι να επιτευχθεί μια αποτελεσματική εκτέλεση της εφαρμογής, η οποία μεταφράζεται στη βελτιστοποίηση κάποιας αντικειμενικής συνάρτησης — συνήθως του συνολικού χρόνου εκτέλεσης (makespan). Οι αλγόριθμοι προγραμματισμού είναι εντελώς ανεξάρτητοι από τα μοντέλα και τις μεθόδους που χρησιμοποιούνται για την εξαγωγή γράφων εργασιών (Miller, και συν. n.d.).

3.4 Τοπολογική ταξινόμηση γράφων εργασιών

Στην τοπολογική ταξινόμηση (topological sorting) ενός κατευθυνόμενου μη κυκλικού γράφου (DAG) επιδιώκεται η γραμμική διάταξη των κόμβων, έτσι ώστε κάθε γονικός να προηγείται των θυγατρικών του. Σε ένα DAG πολλές φορές μπορούμε να έχουμε κόμβους που δεν συνδέονται μεταξύ τους και αυτός είναι ο λόγος για τον οποίο η τοπολογική ταξινόμηση δεν έχει απαραιτήτως μοναδική λύση. Οι αλγόριθμοι τοπολογικής ταξινόμησης εκτελούνται σε γραμμικό χρόνο.

Ένας από αυτούς τους αλγορίθμους, περιγράφηκε αρχικά από τον Kahn (1962). Για την τοπολογική ταξινόμηση ενός DAG, αρχικά, οι κόμβοι έναρξης -που δεν έχουν εισερχόμενες ακμές- εισάγονται σε ένα σύνολο S. Αν υποθέσουμε ότι υπάρχει μοναδικός κόμβος εισόδου, v_{entry}, ο αλγόριθμος του Kahn διαμορφώνεται ως εξής:

```
L ← κενή λίστα που θα συμπεριλάβει τους ταξινομημένους κόμβους
S ← σύνολο με μοναδικό στοιχείο τον κόμβο εισόδου: v_entry
while S is non-empty do
        remove a node v from S
        add v to tail of L
        for each node u with an edge e from v to u do
                remove edge e from the graph
                if u has no other incoming edges then
                        insert u into S
return L   (L η λίστα των ταξινομημένων κόμβων)
```

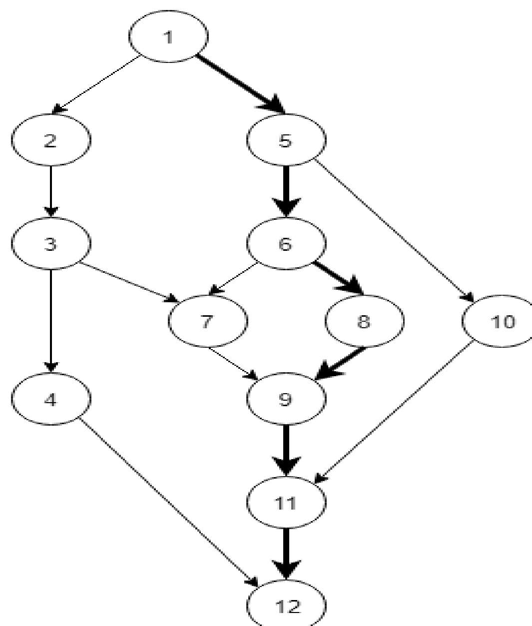
Με βάση τον αλγόριθμο του Kahn, η τοπολογική ταξινόμηση δεν έχει μοναδική λύση, καθώς η λύση που προκύπτει εξαρτάται από τη σειρά με την οποία αφαιρούνται οι κόμβοι από το σύνολο S (Wikipedia 2018).

Αλγόριθμοι τοπολογικής ταξινόμησης έχουν προταθεί επίσης από τους Wells, Knuth-Szwarcfiter και Varol-Rotem (Kalvin and Varol 1983). Για την τοπολογική ταξινόμηση μεγάλων γράφων έχουν προταθεί οι αλγόριθμοι ITERTS, PEELTS, SETS και REACHTS (Ajwani, Cosgaya-Lozano και Zeh 2012).

3.5 Ιδιότητες γράφων εργασιών

3.5.1 Κρίσιμο μονοπάτι

Ένα άθροισμα με όρους τα κόστη υπολογισμού των κόμβων μιας διαδρομής ενός DAG αποτελεί το μήκος αυτής της διαδρομής. Το μήκος της μεγαλύτερης διαδρομής από τον κόμβο εισόδου στον κόμβο εξόδου χαρακτηρίζεται ως κρίσιμη διαδρομή ή κρίσιμο μονοπάτι (critical path). Ένα DAG μπορεί να έχει αρκετές κρίσιμες διαδρομές ίσου μήκους. Η κρίσιμη διαδρομή είναι μια σημαντική παράμετρος ενός DAG, επειδή δείχνει το κάτω όριο για το μήκος κάθε πιθανού χρονοδιαγράμματος (makespan). Το όριο αυτό δεν μεταβάλλεται ακόμη και με την χρήση απεριόριστου πλήθους επεξεργαστών.



Σχήμα 3.5. Παράδειγμα γράφου εργασιών – Κρίσιμο μονοπάτι

Οι κόμβοι της κρίσιμης διαδρομής χαρακτηρίζονται σημαντικοί, καθώς η καθυστερημένη εκκίνηση εκτέλεσης οποιουδήποτε από αυτούς τους κόμβους οδηγεί άμεσα σε αύξηση του μήκους του χρονοδιαγράμματος. Ένας κόμβος που δεν ανήκει στην κρίσιμη διαδρομή, δεν είναι τόσο σημαντικός, με την προϋπόθεση ότι δεν καθυστερεί την εκτέλεση οποιουδήποτε κόμβου ανήκει στην κρίσιμη διαδρομή.

Στην περίπτωση κατά την οποία η κρίσιμη διαδρομή συνδέεται με συγκεκριμένο πρόγραμμα, που αναθέτει εργασίες σε επεξεργαστές, περιλαμβάνει επιπλέον και τα κόστη επικοινωνίας μεταξύ των κόμβων, αλλά στην περίπτωση αυτή, το μήκος κρίσιμης διαδρομής δεν είναι το ελάχιστο δυνατό μήκος του χρονοδιαγράμματος.

3.5.2 Επίπεδα κόμβων

Σε ένα DAG ορίζουμε σημαντικές παραμέτρους. Αυτές οι παράμετροι διακρίνονται σε στατικές ή δυναμικές. Οι στατικές παράμετροι περιγράφουν ένα DAG χωρίς να συνδέονται με συγκεκριμένο πρόγραμμα, που αναθέτει τις εργασίες του DAG σε επεξεργαστές.

Το **s-level** (static level) ενός κόμβου v_i είναι το μήκος της μεγαλύτερης διαδρομής από τον κόμβο v_i στον κόμβο εξόδου, συμπεριλαμβανομένου του βάρους του v_i. Δεν εξετάζει τα κόστη επικοινωνίας, γεγονός που το καθιστά ανεξάρτητο του χρονοδιαγράμματος. Το s-level του κόμβου εισόδου ισούται με το μήκος της κρίσιμης διαδρομής. Για τον κόμβο v_i το s-level δίνεται από την ακόλουθη αναδρομική σχέση:

$$sl(v_i) = \max_{v_j \in succ(v_i)} (sl(v_j)) + w(v_i)$$

ενώ για τον κόμβο εξόδου ισχύει:

$$sl(v_{exit}) = w(v_{exit})$$

Το **static t-level** (static top level) ενός κόμβου v_i είναι το μήκος της μεγαλύτερης διαδρομής από τον κόμβο εισόδου στον κόμβο v_i, εξαιρουμένου του βάρους του v_i. Το static t-level περιλαμβάνει όλα τα κόστη επικοινωνίας και υπολογίζεται ανεξάρτητα από το χρονοδιάγραμμα. Έτσι ο υπολογισμός του static t-level δεν εξετάζει τη δυνατότητα

μηδενισμού του κόστους επικοινωνίας, στην περίπτωση που εργασίες με σχέση εξάρτησης εκτελούνται στον ίδιο επεξεργαστή. Το static t-level ενός κόμβου v_i δίνεται από την ακόλουθη αναδρομική σχέση:

$$stl(v_i) = \max_{v_j \in pred(v_i)} (stl(v_j) + w(v_j) + c_{j,i})$$

ενώ για τον κόμβο εισόδου ισχύει:

$$stl(v_{entry}) = 0$$

Το **static b-level** (static bottom level) ενός κόμβου v_i είναι το μήκος της μεγαλύτερης διαδρομής από τον κόμβο v_i στον κόμβο εξόδου, συμπεριλαμβανομένου του βάρους του v_i. Ο υπολογισμός του static b-level περιλαμβάνει τα κόστη επικοινωνίας όπως και το static t-level. Το static b-level ενός κόμβου v_i δίνεται από την ακόλουθη αναδρομική σχέση:

$$sbl(v_i) = w(v_i) + \max_{v_j \in succ(v_i)} (c_{i,j} + sbl(v_j))$$

ενώ για τον κόμβο εξόδου ισχύει:

$$sbl(v_{exit}) = w(v_{exit})$$

Ως **total work** χαρακτηρίζεται η συνολική πολυπλοκότητα όλων των εργασιών ενός DAG. Ισούται με το μήκος του χρονοδιαγράμματος στην ιδανική περίπτωση κατά την οποία όλες οι εργασίες έχουν προγραμματιστεί στον ίδιο επεξεργαστή. Δίνεται από τον τύπο:

$$W = \sum_{v_i \in V} (w(v_i))$$

4. ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΕΡΓΑΣΙΩΝ

Ένας αλγόριθμος χρονοπρογραμματισμού, ο οποίος αναθέτει εργασίες σε ένα σύστημα πολλαπλών επεξεργαστών, περιγράφεται από κάποιες παραμέτρους οι οποίες χαρακτηρίζονται ως δυναμικές παράμετροι:

Start time ($ST(v_i)$) ενός κόμβου για ένα συγκεκριμένο χρονοδιάγραμμα**.**

Finish time ($FT(v_i)$) ενός κόμβου για ένα συγκεκριμένο χρονοδιάγραμμα.

Οι παράμετροι Start time και Finish time συνδέονται με τη σχέση:

$$FT(v_i) = ST(v_i) + w(v_i)$$

Το **dynamic t-level** (dynamic top level) ενός κόμβου v_i σε έναν επεξεργαστή p_i είναι το μήκος της μεγαλύτερης διαδρομής από τον κόμβο εισόδου στον κόμβο v_i, εξαιρουμένου του βάρους του v_i. Το dynamic t-level δηλώνει τον χρόνο κατά τον οποίο η εκτέλεση των γονικών κόμβων του v_i έχει ολοκληρωθεί και εκτελείται η εκκίνηση της εργασίας v_i. Το dynamic t-level ενός κόμβου v_i δίνεται από την ακόλουθη αναδρομική σχέση:

$$tl(v_i) = \max_{v_j \in pred(v_i)} (FT(v_j) + c_{j,i})$$

ενώ για τον κόμβο εισόδου ισχύει:

$$tl(v_{entry}) = 0$$

Το dynamic t-level υπολογίζεται σε ένα πλαίσιο συγκεκριμένου χρονοδιαγράμματος, που σημαίνει ότι, εάν η εκτέλεση των εργασιών v_i και v_j έχουν προγραμματιστεί στον ίδιο επεξεργαστή, τότε ισχύει $c_{j,i} = 0$. Το dynamic t-level δεν ελέγχει τη διαθεσιμότητα ενός επεξεργαστή για την εκτέλεση της εργασίας v_i, επομένως ο πραγματικός ενωρίτερος χρόνος εκκίνησης της εργασίας v_i μπορεί να είναι μεταγενέστερος.

Το **dynamic b-level** (dynamic bottom level) ενός κόμβου v_i σε έναν επεξεργαστή p_i είναι το μήκος της μεγαλύτερης διαδρομής από τον κόμβο v_i στον κόμβο εξόδου, συμπεριλαμβανομένου του βάρους του v_i. Το dynamic b-level ενός κόμβου v_i δίνεται από την ακόλουθη αναδρομική σχέση:

$$bl(v_i) = w(v_i) + \max_{v_j \in succ(v_i)} (c_{i,j} + bl(v_j))$$

ενώ για τον κόμβο εξόδου ισχύει:

$$bl(v_{exit}) = w(v_{exit})$$

Επίσης και η παράμετρος **critical path** (κρίσιμη διαδρομή) μπορεί να θεωρηθεί ως δυναμική ιδιότητα και ισούται με το dynamic b-level του κόμβου εισόδου. Τόσο το dynamic b-level όσο και το μήκος της κρίσιμης διαδρομής μπορούν να αλλάξουν ανάλογα με την διάταξη των κόμβων που περιλαμβάνουν.

Η παράμετρος **activity** ενός κόμβου v_i είναι λογικού τύπου και δηλώνει αν η εργασία v_i εκτελείται την χρονική στιγμή t.

Η παράμετρος **ready time** ($R_i(v_j)$) ενός επεξεργαστή p_i εκφράζει την ενωρίτερη δυνατή χρονική στιγμή κατά την οποία ο επεξεργαστής p_i μπορεί να εκκινήσει την εκτέλεση μιας εργασίας βάρους $w(v_j)$. Η παράμετρος αυτή λαμβάνει υπόψη τον χρόνο που απαιτείται για να ικανοποιηθούν οι εξαρτήσεις της εργασίας v_j, δηλαδή να έχει ολοκληρωθεί η εκτέλεση των γονικών της εργασιών.

Η παράμετρος **Earliest Start Time** αναφέρεται στην ενωρίτερη δυνατή χρονική στιγμή κατά την οποία ο επεξεργαστής p_j μπορεί να εκκινήσει την εκτέλεση μιας εργασίας v_i. Η παράμετρος αυτή λαμβάνει υπόψη τόσο το dynamic t-level του κόμβου v_i , όσο και το ready time του επεξεργαστή p_j.

$$EST(v_i, p_j) = max\{tl(v_i),\ R_j(v_i)\}$$

Η παράμετρος **Earliest Finish Time** αναφέρεται στην ενωρίτερη δυνατή χρονική στιγμή κατά την οποία η εκτέλεση της εργασίας v_i μπορεί να ολοκληρωθεί από τον επεξεργαστή p_j. Οι παράμετροι Earliest Start Time και Earliest Finish Time συνδέονται με τη σχέση:

$$EFT(v_i, p_j) = EST(v_i, p_j) + w(v_i)$$

Η παράμετρος **Actual Finish Time** αναφέρεται στην χρονική στιγμή κατά την οποία η εκτέλεση της εργασίας v_i ολοκληρώνεται από τον επεξεργαστή p_j, στον οποίο έχει

ανατεθεί βάσει συγκεκριμένου προγράμματος. Στην περίπτωση ομογενούς συστήματος, ισχύει $AFT(v_i) = EFT(v_i)$. Η παράμετρος Earliest Finish Time ενός κόμβου v_i και η παράμετρος Actual Finish Time των γονικών του κόμβων συνδέονται με τη σχέση:

$$EFT(v_i, p_j) = max\left\{R_j(v_i), \max_{v_k \in pred(v_i)}(AFT(v_k) + c_{k,i})\right\} + w(v_i)$$

Το **makespan** υποδηλώνει τον χρόνο τερματισμού της εργασίας εξόδου ή ισοδύναμα τον συνολικό χρόνο εκτέλεσης ή το μήκος του προγράμματος και ορίζεται ως:

$$makespan = AFT(v_{exit})$$

Το **As-late-as-possible** είναι ένα μέγεθος που δείχνει πόσο μπορεί να καθυστερήσει η χρονική στιγμή έναρξης για την εκτέλεση μιας εργασίας χωρίς να αυξηθεί το συνολικό μήκος του προγράμματος. Ισούται με το t-level όταν πρόκειται για εργασία που ανήκει στο κρίσιμο μονοπάτι. Γενικά δίνεται από τον τύπο:

$$ALAP(\ v_i) = \min_{v_j \in succ(v_i)}(ALAP(v_j) - c_{i,j}) - w(v_i)$$

ενώ για τον κόμβο εισόδου ισχύει:

$$ALAP(\ v_{entry}) = CP - w(v_{entry})$$

Ένα πρόγραμμα χαρακτηρίζεται βέλτιστο εάν έχει τις καλύτερες δυνατές παραμέτρους αξιολόγησης μεταξύ άλλων προγραμμάτων - για δεδομένο αριθμό επεξεργαστών.

Αν και υπάρχουν διάφορες αντικειμενικές παράμετροι, συνήθως για την αξιολόγηση ενός προγράμματος χρησιμοποιείται το makespan. Αυτό σημαίνει ότι ο αλγόριθμος με την καλύτερη απόδοση είναι αυτός που παράγει πρόγραμμα με το μικρότερο συνολικό χρόνο εκτέλεσης (Planeta 2015).

4.1 Προγραμματισμός εργασιών με και χωρίς κόστη επικοινωνίας

Ιστορικά, το θέμα των προβλημάτων προγραμματισμού εργασιών ξεκίνησε χωρίς να λαμβάνονται υπόψη τα κόστη επικοινωνίας. Σε ένα γράφο εργασιών χωρίς κόστη

επικοινωνίας οι ακμές αντιπροσωπεύουν αποκλειστικά τις σχέσεις εξάρτησης των κόμβων.

Από τα τέλη της δεκαετίας του 1980 οι καθυστερήσεις επικοινωνίας ενσωματώθηκαν στο πρόβλημα προγραμματισμού εργασιών. Ένα παράδειγμα κόστους επικοινωνίας αποτελεί το γεγονός ότι δύο εργασίες που εκτελούνται στον ίδιο επεξεργαστή μπορούν να επικοινωνούν μέσω της cache μνήμης του επεξεργαστή, ενώ όταν εκτελούνται σε διαφορετικούς επεξεργαστές πρέπει να επικοινωνούν μέσω της κύριας μνήμης. Επίσης, μια άλλη αιτία για το κόστος απομακρυσμένης επικοινωνίας μπορεί να είναι η ανταλλαγή μηνυμάτων μεταξύ των επεξεργαστών για την υλοποίηση της παράλληλης εκτέλεσης (Sinnen 2007).

Το πρόβλημα του προγραμματισμού των καθυστερήσεων επικοινωνίας αποδεικνύεται εκπληκτικά δύσκολο ακόμη και με απεριόριστα διαθέσιμους πόρους. Ωστόσο, έχουν αναπτυχθεί ευρετικοί αλγόριθμοι, τόσο για απεριόριστους πόρους όσο και για περιορισμένους πόρους, αποδεδειγμένα οικονομικά αποδοτικοί για πρακτικά προβλήματα (Darte, Robert και Vivien 2000).

4.2 Προγραμματισμός εργασιών και NP πληρότητα

Όταν δεν υπάρχει περιορισμός στον αριθμό των διαθέσιμων πόρων, οι βέλτιστοι αλγόριθμοι προγραμματισμού εργασιών ενός DAG έχουν πολυωνυμικό χρόνο εκτέλεσης. Με περιορισμένους πόρους το πρόβλημα καθίσταται NP-πλήρες, οπότε οι ευρετικοί αλγόριθμοι είναι η συνήθης προσέγγιση. Αποδεικνύεται ότι η εισαγωγή του κόστους επικοινωνίας ακόμα και με απεριόριστους πόρους καθιστά το πρόβλημα NP-πλήρες (Kwok and Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999).

Γενικά, το πρόβλημα προγραμματισμού του κατευθυνόμενου μη κυκλικού γράφου (DAG) είναι ένα NP-πλήρες πρόβλημα και οι γνωστοί αλγόριθμοι για τον βέλτιστο προγραμματισμό ενός DAG σε πολυωνυμικό χρόνο αφορούν μόνο τρεις απλές περιπτώσεις, για τις οποίες ισχύει η υπόθεση ότι το κόστος επικοινωνίας μεταξύ των εργασιών είναι μηδενικό.

Η πρώτη περίπτωση αναφέρεται στον προγραμματισμό ενός ελεύθερου δέντρου (free-tree) με ομοιόμορφα βάρη κόμβων σε έναν αυθαίρετο αριθμό επεξεργαστών. Για την επίλυση αυτού του προβλήματος προτάθηκε από τον Hu [1961] ένας αλγόριθμος γραμμικού χρόνου.

Η δεύτερη περίπτωση αναφέρεται στον προγραμματισμό ενός DAG με αυθαίρετη δομή και ομοιόμορφα βάρη κόμβων σε δύο επεξεργαστές. Για την επίλυση αυτού του προβλήματος οι Coffman και Graham [1972] επινόησαν έναν αλγόριθμο τετραγωνικού χρόνου. Οι αλγόριθμοι των Hu και Coffman κ.α. βασίζονται σε μεθόδους node-labeling που οδηγούν επίσης σε βέλτιστα χρονοδιαγράμματα. Η διαδικασία node-labeling αποδίδει σε κάθε κόμβο μια τιμή σήμανσης (σε συνάρτηση με την διαδρομή από τον κόμβο αυτό στον κόμβο εξόδου) και στη συνέχεια χωρίζει τον γράφο σε επίπεδα εργασιών, τα οποία ανατίθενται στους διαθέσιμους επεξεργαστές. Ο Sethi [1976], στη συνέχεια, βελτίωσε την χρονική πολυπλοκότητα του αλγορίθμου Coffman κ.α. σε σχεδόν γραμμικό χρόνο, υποδεικνύοντας μια πιο αποτελεσματική διαδικασία σήμανσης κόμβων.

Η τρίτη περίπτωση αναφέρεται στον προγραμματισμό ενός DAG διατεταγμένου με διαστήματα και ομοιόμορφα βάρη κόμβων σε έναν αυθαίρετο αριθμό επεξεργαστών. Ένα DAG ονομάζεται διατεταγμένο με διαστήματα (interval-ordered DAG), εάν κάθε δύο κόμβοι που έχουν προτεραιότητα μπορούν να αντιστοιχιστούν σε δύο μη αλληλεπικαλυπτόμενα διαστήματα στην πραγματική ευθεία. Για την αντιμετώπιση του προβλήματος, οι Παπαδημητρίου και Γιαννακάκης [1979] σχεδίασαν έναν αλγόριθμο γραμμικού χρόνου.

Ενώ σε όλες τις παραπάνω τρεις περιπτώσεις, το κόστος επικοινωνίας μεταξύ των εργασιών παραβλέπεται, οι Ali και El-Rewini [1993] έδειξαν ότι ένα DAG διατεταγμένο με διαστήματα με μοναδιαία βάρη ακμών και μοναδιαία βάρη κόμβων, μπορεί επίσης να προγραμματιστεί κατά βέλτιστο τρόπο σε πολυωνυμικό χρόνο.

Ο Ullman [1975] έδειξε ότι ο προγραμματισμός ενός DAG με μοναδιαίο κόστος υπολογισμού σε p επεξεργαστές είναι NP-πλήρης. Έδειξε επίσης ότι ο προγραμματισμός ενός DAG με μια ή δύο τιμές κόστους υπολογισμού σε δύο επεξεργαστές είναι NP-πλήρης. Οι Παπαδημητρίου και Γιαννακάκης [1979] έδειξαν ότι ο προγραμματισμός ενός

DAG διατεταγμένου με διαστήματα με αυθαίρετο κόστος υπολογισμού σε δύο επεξεργαστές είναι NP-πλήρης. Ο Garey κ.α. [1983] έδειξαν ότι ο προγραμματισμός ενός opposing forest με μοναδιαίο κόστος υπολογισμού σε p επεξεργαστές είναι NP-πλήρης. Τέλος, οι Παπαδημητρίου και Γιαννακάκης [1990] έδειξαν ότι ο προγραμματισμός ενός DAG με μοναδιαίο κόστος υπολογισμού σε p επεξεργαστές, ενδεχομένως με την μέθοδο task-duplication, είναι επίσης NP-πλήρης (Kwok και Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999).

Διευκρινίζεται ότι η κλάση NP περιλαμβάνει όλα τα υπολογιστικά προβλήματα που επιλύονται σε μη ντετερμινιστικό πολυωνυμικό χρόνο. Ωστόσο, αν δίνεται μια πιθανή λύση για ένα πρόβλημα της κλάσης NP, μπορεί να ελεγχθεί σε πολυωνυμικό χρόνο αν αυτή πράγματι αποτελεί μια λύση του προβλήματος.

5. ΒΑΣΙΚΕΣ ΕΥΡΕΤΙΚΕΣ ΤΕΧΝΙΚΕΣ

Η NP-πληρότητα του προβλήματος και το γεγονός, ότι ο αποτελεσματικός προγραμματισμός είναι απαραίτητος για την επίτευξη μιας ουσιαστικής επιτάχυνσης σε ένα παράλληλο ή κατανεμημένο σύστημα, έχει ωθήσει τους ερευνητές να προτείνουν μια πληθώρα ευρετικών αλγορίθμων. Οι ευρετικοί αλγόριθμοι μπορούν να βρουν καλές λύσεις εντός εύλογου χρονικού διαστήματος. Ωστόσο, ενώ οι περισσότεροι από αυτούς τους αλγορίθμους αναφέρονται ως αποτελεσματικοί (όπως αποδεικνύεται από πειραματικές μελέτες), συνήθως δεν μπορούν να δημιουργήσουν βέλτιστες λύσεις και η μεταξύ τους σύγκριση είναι ένα σύνθετο έργο.

Ο προγραμματισμός λίστας και η συσταδοποίηση αποτελούν βασικές κατηγορίες ευρετικών τεχνικών για τον χρονοπρογραμματισμό των κόμβων ενός γράφου.

5.1 Προγραμματισμός λίστας (list scheduling)

Οι περισσότεροι αλγόριθμοι χρονοπρογραμματισμού βασίζονται σε μια τεχνική που ονομάζεται list scheduling. Η βασική ιδέα αυτής της τεχνικής είναι η δημιουργία μιας ακολουθίας κόμβων με ανάθεση ορισμένων προτεραιοτήτων και η επαναλαμβανόμενη εκτέλεση δύο βημάτων μέχρι τον πλήρη χρονοπρογραμματισμό των κόμβων του γράφου. Τα βήματα αυτά αναφέρονται στην απομάκρυνση του εκάστοτε κόμβου με την υψηλότερη προτεραιότητα και την ανάθεσή του σε έναν επεξεργαστή, ο οποίος υποστηρίζει τον σχετικά προγενέστερο χρόνο έναρξης της εκτέλεσής του (earliest start-time). Ωστόσο, σε ορισμένες εξελιγμένες ευρετικές τεχνικές, ο κατάλληλος επεξεργαστής δεν καθορίζεται με αυτό το κριτήριο.

Ο καθορισμός των προτεραιοτήτων των κόμβων υποστηρίζεται από διάφορες τεχνικές, όπως οι HLF (Highest level First), LP (Longest Path), LPT (Longest Processing Time) και CP (Critical Path).

Δύο παράμετροι που χρησιμοποιούνται συχνά για την αντιστοίχιση προτεραιότητας στους ελεύθερους κόμβους είναι τα t-level και b-level. Ορισμένοι αλγόριθμοι αποδίδουν υψηλότερη προτεραιότητα σε έναν κόμβο με μικρότερο t-level, ενώ άλλοι αποδίδουν

υψηλότερη προτεραιότητα σε έναν κόμβο με μεγαλύτερο b-level. Επί πλέον, ορισμένοι αλγόριθμοι αποτελούν συνδυασμό των προηγούμενων, αποδίδοντας υψηλότερη προτεραιότητα σε έναν κόμβο στον οποίο το μέγεθος της διαφοράς ((b-level) – (t-level)) είναι μεγαλύτερο. Γενικά, ο προγραμματισμός σε φθίνουσα σειρά ως προς το b-level θεωρεί ότι όσο μεγαλύτερο είναι το b-level, τόσο πιο επείγουσα είναι η αντίστοιχη εργασία και τείνει να προγραμματίζει πρώτα τους κόμβους της κρίσιμης διαδρομής.

Όσον αφορά τον προσδιορισμό του χρόνου έναρξης για την εκτέλεση ενός κόμβου, σύμφωνα με την πιο συνηθισμένη τεχνική, ένας κόμβος ανατίθεται σε έναν δεδομένο επεξεργαστή, όταν ολοκληρωθεί η εκτέλεση όλων των κόμβων, οι οποίοι έχουν ήδη προγραμματιστεί σε αυτόν τον επεξεργαστή. Μια τεχνική που ονομάζεται insertion heuristic ή insertion approach ελέγχει για χρονικές περιόδους – χρονοθυρίδες (holes) κατά τις οποίες ένας επεξεργαστής βρίσκεται σε αδράνεια καθώς δεν εκτελεί κάποια εργασία και προσπαθεί να τις συμπληρώσει με ανάθεση εργασιών, εφόσον είναι εφικτό. Το μέγεθος της χρονοθυρίδας αδράνειας (δηλαδή η διαφορά μεταξύ χρόνου έναρξης και χρόνου τερματισμού εκτέλεσης αντίστοιχα για δύο εργασίες, που εκτελούνται διαδοχικά στον ίδιο επεξεργαστή) θα πρέπει να είναι κατ’ ελάχιστο αντίστοιχο του υπολογιστικού κόστους της εργασίας, η οποία πρόκειται να παρεμβληθεί. Παρότι η χρήση της τεχνικής insertion approach στον προγραμματισμό λίστας έχει τη δυνατότητα να μειώσει το μήκος του χρονοδιαγράμματος, δεν υπάρχει γενικά καμία εγγύηση για βελτίωση. Αυτό γίνεται σαφές, όταν αντιληφθούμε ότι η τεχνική insertion approach μπορεί να οδηγήσει σε διαφορετικές κατανομές κόμβων σε επεξεργαστές, με ενδεχόμενο αποτέλεσμα ένα μεγαλύτερο χρονοδιάγραμμα (Sinnen 2007).

Πολλοί αλγόριθμοι υποστηρίζουν μια δυναμική προσέγγιση της τεχνικής προγραμματισμού λίστας, η οποία ονομάζεται dynamic list scheduling. Οι αλγόριθμοι αυτοί, μετά από κάθε ανάθεση κόμβου σε επεξεργαστή, επαναπροσδιορίζουν τις προτεραιότητες όλων των μη προγραμματισμένων κόμβων και στη συνέχεια αναδιατάσσουν την αλληλουχία των κόμβων της λίστας (Kwok and Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999). Οι αλγόριθμοι προγραμματισμού, οι οποίοι χρησιμοποιούν αυτήν την τεχνική, μπορούν δυνητικά να δημιουργήσουν καλύτερα χρονοδιαγράμματα. Ωστόσο, μια

δυναμική προσέγγιση ενδέχεται να αυξήσει την πολυπλοκότητα του αλγορίθμου (Planeta 2015).

5.1.1 Start Time Minimization

Η πιο κοινή επιλογή επεξεργαστή για την ανάθεση ενός κόμβου είναι εκείνος ο οποίος επιτρέπει τον ενωρίτερο χρόνο έναρξης (earliest start-time) του κόμβου. Ο επεξεργαστής αυτός βρίσκεται απλά υπολογίζοντας τον χρόνο έναρξης μιας εργασίας σε καθένα από τους επεξεργαστές και επιλέγοντας εκείνον για τον οποίο ο χρόνος έναρξης είναι ελάχιστος. Συνήθως, ο προγραμματισμός λίστας συνεπάγεται την ελαχιστοποίηση του χρόνου έναρξης και έχουν προταθεί πολλοί αλγόριθμοι για αυτό το είδος προγραμματισμού.

Ο προγραμματισμός λίστας με την ελαχιστοποίηση του χρόνου έναρξης ανήκει στην τάξη άπληστων αλγορίθμων. Σε κάθε βήμα, προσπαθεί να δημιουργήσει ένα νέο μερικό πρόγραμμα μικρού μήκους, με την εικασία ότι αυτό τελικά θα οδηγήσει σε ένα γρήγορο τελικό πρόγραμμα. Ωστόσο, ένα λάθος σε πρώιμο στάδιο δεν μπορεί να διορθωθεί αργότερα.

Έχει δειχτεί ότι, για γραφήματα εργασιών χωρίς κόστος επικοινωνίας, το χειρότερο μήκος ενός χρονοδιαγράμματος, που παράγεται από τον προγραμματισμό λίστας με ελαχιστοποίηση του χρόνου έναρξης, είναι το διπλάσιο του βέλτιστου μήκους. Ωστόσο, για γραφήματα εργασιών με κόστος επικοινωνίας δεν υπάρχει κάποια εγγύηση για το μήκος του χρονοδιαγράμματος, που παράγεται από έναν τέτοιο αλγόριθμο (Sinnen 2007).

5.1.2 HLFET (highest levels first with estimated times)

Ο αλγόριθμος HLFET (highest levels first with estimated times) είναι ένας από τους πιο απλούς αλγορίθμους προγραμματισμού λίστας. Καθορίζει την προτεραιότητα στους κόμβους σύμφωνα με το s-level, το οποίο εξ ορισμού δεν εξετάζει τα κόστη επικοινωνίας μεταξύ των κόμβων. Τα βάρη των κόμβων θεωρούνται γνωστά και περιγράφονται από

τους εκτιμώμενους χρόνους (estimated times). Ο αλγόριθμος περιγράφηκε από τους Kwok και Ahmad (Kwok και Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors December 1999) ως εξής:

1. Υπολογίζεται το s-level κάθε κόμβου.
2. Οι κόμβοι τοποθετούνται σε μια έτοιμη ουρά αναμονής σε φθίνουσα σειρά με βάση τις τιμές των s-level. Αρχικά περιέχεται μόνο ο κόμβος εισόδου. Εάν οι κόμβοι έχουν το ίδιο s-level, η προτεραιότητα ορίζεται τυχαία.
3. Γίνεται ανάθεση του κόμβου με την υψηλότερη προτεραιότητα σε έναν επεξεργαστή, ο οποίος υποστηρίζει τον σχετικά προγενέστερο χρόνο εκκίνησης (earliest start-time). Δεν χρησιμοποιείται insertion approach.
4. Ενημερώνεται η ουρά με την εισαγωγή των κόμβων που είναι πλέον έτοιμοι (δηλαδή ικανοποιούνται οι εξαρτήσεις τους).
5. Εφόσον στην έτοιμη ουρά υπάρχουν διαθέσιμοι κόμβοι, γίνεται μετάβαση στο βήμα 2.

5.1.3 MCP (Modified Critical Path)

Ο αλγόριθμος MCP (Modified Critical Path) θεωρεί μία κρίσιμη διαδρομή (CP) στο DAG και ορίζει την υψηλότερη προτεραιότητα χρησιμοποιώντας το μέγεθος ALAP (As-Late-As-Possible start-time) ενός κόμβου, χωρίς να εξετάζει τα κόστη επικοινωνίας μεταξύ των κόμβων. Οι μεγαλύτεροι χρόνοι έναρξης των κόμβων μιας κρίσιμης διαδρομής είναι ίσοι με τα t-level τους (Yuyi Jiang, Zhiqing Shao and Yi Guo 2014). Ο αλγόριθμος MCP, αρχικά, υπολογίζει την τιμή του ALAP όλων των κόμβων και τους διατάσσει σε μια λίστα κόμβων σε αύξουσα σειρά με βάση τους χρόνους ALAP. Στη συνέχεια προγραμματίζει τους κόμβους της λίστας έναν προς έναν, έτσι ώστε κάθε κόμβος να ανατίθεται σε έναν επεξεργαστή, ο οποίος επιτρέπει τον σχετικά προγενέστερο χρόνο εκκίνησης της εκτέλεσής του (earliest start-time). Ο αλγόριθμος MCP χρησιμοποιεί insertion approach και έχει υψηλότερη ταχύτητα σε σύγκριση με τον αλγόριθμο HLFET (Samriti, et al. 2012). Ο αλγόριθμος MCP περιγράφεται ως εξής (Darte, Robert και Vivien 2000, Planeta 2015):

1. Υπολογίζεται ο χρόνος ALAP κάθε κόμβου.

2. Για κάθε κόμβο, δημιουργείται μια λίστα που αποτελείται από τους χρόνους ALAP του ίδιου του κόμβου και όλων των θυγατρικών του σε φθίνουσα σειρά. Εάν θυγατρικοί κόμβοι έχουν ίσους ελάχιστους χρόνους ALAP μεταξύ τους, η προτεραιότητα ορίζεται αυθαίρετα.
3. Οι κόμβοι τοποθετούνται σε μια ουρά αναμονής σε αύξουσα σειρά με βάση τις τιμές των ALAP.
4. Γίνεται ανάθεση του κόμβου με την υψηλότερη προτεραιότητα σε έναν επεξεργαστή, ο οποίος υποστηρίζει τον σχετικά προγενέστερο χρόνο έναρξης (earliest start-time) χρησιμοποιώντας insertion approach.
5. Εφόσον στην ουρά προτεραιότητας υπάρχουν κόμβοι οι οποίοι δεν έχουν προγραμματιστεί, γίνεται μετάβαση στο βήμα 4.

5.1.4 HEFT (Heterogeneous earliest time first)

Ο αλγόριθμος HEFT (Heterogeneous earliest time first) θεωρείται από τους σημαντικότερους της κατηγορίας list scheduling heuristics, καθώς είναι απλός και πολύ γρήγορος (Planeta 2015). Ο αλγόριθμος HEFT αντιπροσωπεύει μια ευρέως διαδεδομένη οικογένεια αλγορίθμων για ετερογενή συστήματα.

Λόγω της ετερογένειας των επεξεργαστών, ο υπολογισμός των b-levels και t-levels είναι άνευ σημασίας. Ο αλγόριθμος HEFT, για την διάταξη των κόμβων σε μια ουρά προτεραιότητας, χρησιμοποιεί για κάθε κόμβο ένα μέγεθος εξαρτώμενο τόσο από τα κόστη υπολογισμού σε διαφορετικούς επεξεργαστές, όσο και από τα κόστη επικοινωνίας μεταξύ των εργασιών. Το μέγεθος αυτό χαρακτηρίζεται ως upward rank. Η ταξινόμηση με βάση το upward rank διασφαλίζει και τοπολογική ταξινόμηση, έτσι ώστε κάθε γονικός κόμβος να προηγείται των θυγατρικών του. Το upward rank δίνεται από τον τύπο:

$$r_u(v_i) = \overline{w_i} + \max_{v_j \in succ(v_i)} (\overline{c_{i,j}} + r_u(v_j)),$$

όπου $\overline{w_i}$ εκφράζει το μέσο υπολογιστικό κόστος του κόμβου v_i στους διαθέσιμους επεξεργαστές (πλήθους p):

$$\overline{w_i} = \frac{\sum_{j=1}^{p} w(v_i, j)}{p}$$

και $\overline{c_{i,j}}$ εκφράζει το μέσο κόστος επικοινωνίας του κόμβου v_i με τον κόμβο v_j, το οποίο χαρακτηρίζει την αντίστοιχη ακμή πριν την ανάθεση εργασιών σε συγκεκριμένους επεξεργαστές:

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}}$$

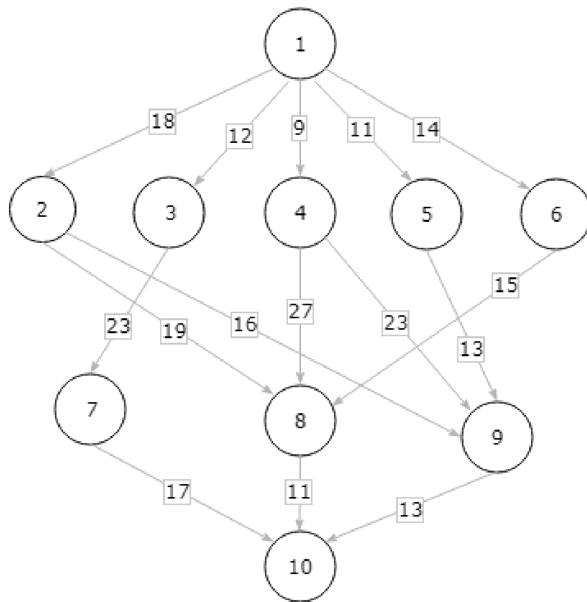
όπου $\overline{B}$ ο μέσος ρυθμός μετάδοσης του συστήματος και $\overline{L}$ η μέση χρονική υστέρηση επικοινωνίας (Topcuoglu, Hariri and Wu 2002).

Για τον κόμβο εξόδου ισχύει:

$$r_u(v_{exit}) = \overline{w_{exit}}$$

Τα βήματα του αλγορίθμου είναι τα εξής:

1. Οι κόμβοι τοποθετούνται σε μια ουρά προτεραιότητας σε φθίνουσα σειρά με βάση τις τιμές upward rank.
2. Για τον κόμβο με την υψηλότερη προτεραιότητα υπολογίζεται το EFT (earliest finish-time) σε κάθε επεξεργαστή, χρησιμοποιώντας insertion approach.
3. Γίνεται ανάθεση του κόμβου στον επεξεργαστή με το μικρότερο EFT.
4. Εφόσον στην ουρά προτεραιότητας υπάρχουν κόμβοι οι οποίοι δεν έχουν προγραμματιστεί, γίνεται μετάβαση στο βήμα 2.



(α) Γράφος 10 εργασιών με κόστη ακμών

κόμβος	PU0	PU1	PU2	μέσο κόστος κόμβου
1	14	16	9	13,00
2	13	19	18	16,67
3	11	13	19	14,33
4	3	8	17	9,33
5	12	13	10	11,67
6	13	16	9	12,67
7	7	15	11	11,00
8	5	11	14	10,00
9	18	12	20	16,67
10	21	7	16	14,67

(β) Παράμετροι ετερογενούς συστήματος – υπολογιστικά κόστη ανά επεξεργαστή

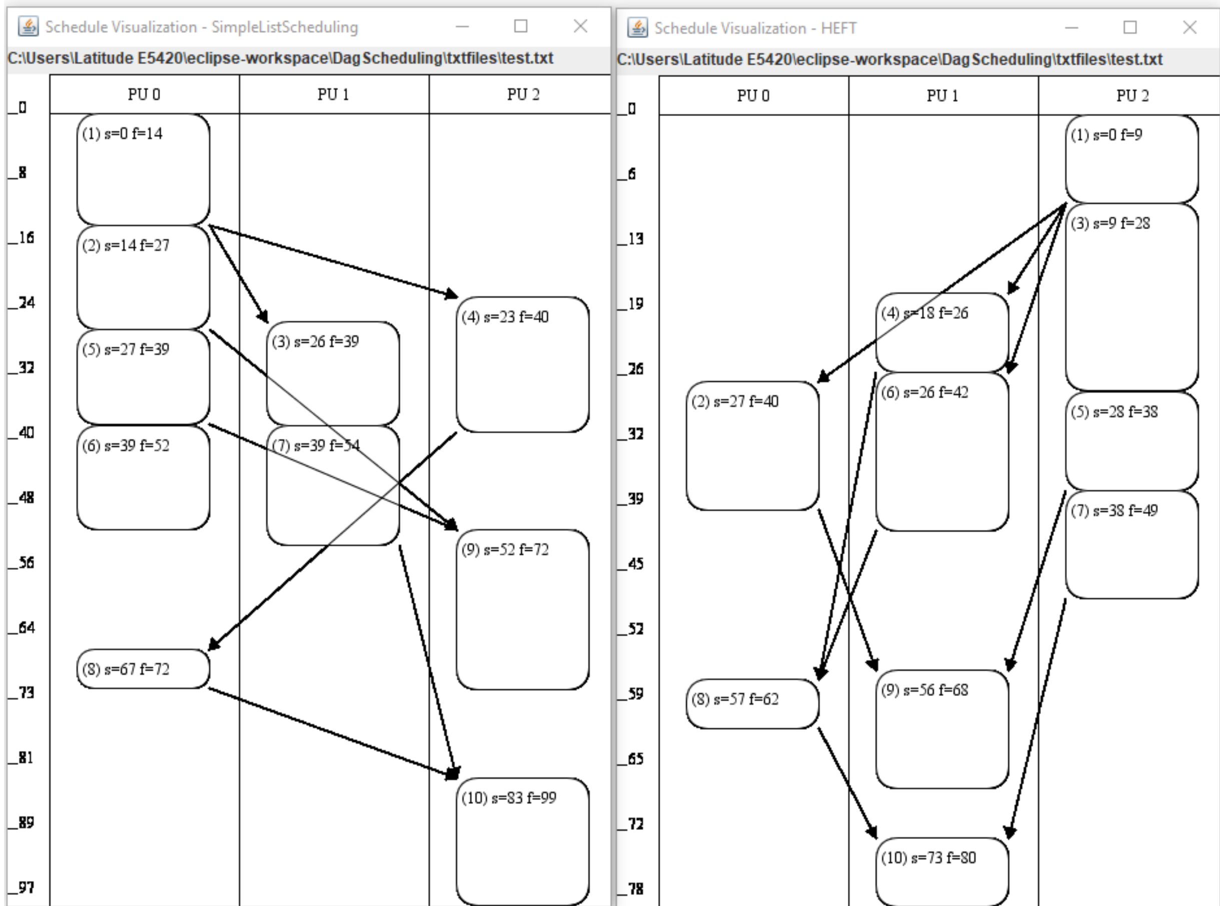
Σχήμα 5.1. Παράδειγμα παράλληλης εφαρμογής σε ετερογενές σύστημα (Valouxis, et al. 2013)

$\boldsymbol{r_u(v_{10})}$	w(10)	**14,67**	=14,67
$\boldsymbol{r_u(v_9)}$	w(9) + c(9,10) + $r_u(v_{10})$	**44,34**	=16,67+13+14,67
$\boldsymbol{r_u(v_8)}$	w(8) + c(8,10) + $r_u(v_{10})$	**35,67**	=10+11+14,67
$\boldsymbol{r_u(v_7)}$	w(7) + c(7,10) + $r_u(v_{10})$	**42,67**	=11+17+14,67
$\boldsymbol{r_u(v_6)}$	w(6) + c(6,8) + $r_u(v_8)$	**63,34**	=12,67+15+35,67
$\boldsymbol{r_u(v_5)}$	w(5) + c(5,9) + $r_u(v_9)$	**69,01**	=11,67+13+44,34
$\boldsymbol{r_u(v_4)}$	w(4) + max(c(4,8) + $r_u(v_8)$, c(4,9) + $r_u(v_9)$)	**76,67**	=9,33+MAX(27+35,67;23+44,34)
$\boldsymbol{r_u(v_3)}$	w(3) + c(3,7) + $r_u(v_7)$	**80,00**	=14,33+23+42,67
$\boldsymbol{r_u(v_2)}$	w(2) + max(c(2,8) + $r_u(v_8)$, c(2,9) + $r_u(v_9)$)	**77,01**	=16,67+MAX(19+35,67;16+44,34)
$\boldsymbol{r_u(v_1)}$	w(1) + max(c(1,2) + $r_u(v_2)$, c(1,3) + $r_u(v_3)$, c(1,4) + $r_u(v_4)$, c(1,5) + $r_u(v_5)$, c(1,6) + $r_u(v_6)$)	**108,01**	=13+MAX(18+77,01;12+80;9+76,67;11+69,01;14+63,34)

Πίνακας 5.1. Υπολογισμός των τιμών upward rank για τον προσδιορισμό της ουράς προτεραιότητας

	$\boldsymbol{r_u(v_1)}$	$\boldsymbol{r_u(v_3)}$	$\boldsymbol{r_u(v_2)}$	$\boldsymbol{r_u(v_4)}$	$\boldsymbol{r_u(v_5)}$	$\boldsymbol{r_u(v_6)}$	$\boldsymbol{r_u(v_9)}$	$\boldsymbol{r_u(v_7)}$	$\boldsymbol{r_u(v_8)}$	$\boldsymbol{r_u(v_{10})}$
	108,01	80,00	77,01	76,67	69,01	63,34	44,34	42,67	35,67	14,67
EFT0	14	32	**40**	43	52	47	69	70	**62**	108
EFT1	16	34	46	**26**	39	**42**	**68**	106	79	**80**
EFT2	**9**	**28**	46	45	**38**	53	76	**49**	73	97
επεξεργαστής	***PU2***	***PU2***	***PU0***	***PU1***	***PU2***	***PU1***	***PU1***	***PU2***	***PU0***	***PU1***

Πίνακας 5.2. Τα βήματα προγραμματισμού σύμφωνα με τον αλγόριθμο HEFT



(α) Αλγόριθμος Start Time Minimization

(β) Αλγόριθμος HEFT

Σχήμα 5.2. Οπτικοποίηση προγραμματισμού ενός DAG

5.1.5 Lookahead HEFT

Οι Bittencourt, Sakellariou και Madeira πρότειναν μια επέκταση του κλασικού αλγορίθμου HEFT. Σύμφωνα με την πρόταση αυτή, οι τοπικά βέλτιστες αποφάσεις δεν βασίζονται μόνο σε εκτιμήσεις που αφορούν μόνο ένα κόμβο v_i, αλλά εξετάζουν επιπλέον και το χρονοδιάγραμμα και λαμβάνουν υπόψη πληροφορίες σχετικές με τον αντίκτυπο αυτής της απόφασης στις θυγατρικές εργασίες του v_i. Ο στόχος είναι να ελαχιστοποιηθεί το μέγιστο EFT μεταξύ όλων των θυγατρικών σε όλους τους επεξεργαστές στους οποίους εκτιμάται ο κόμβος v_i (Bittencourt, Sakellariou και Madeira 2010).

Τα βήματα του αλγορίθμου είναι τα εξής:

1. Οι κόμβοι τοποθετούνται σε μια ουρά προτεραιότητας σε φθίνουσα σειρά με βάση τις τιμές upward rank, όπως και στον αλγόριθμο HEFT.
2. Ο κόμβος με την υψηλότερη προτεραιότητα (έστω v_i) ανατίθεται δοκιμαστικά στον επεξεργαστή p_j.
3. Για κάθε θυγατρικό κόμβο του v_i υπολογίζεται το *EFT* (earliest finish-time) στον επεξεργαστή p_j και προσδιορίζεται η μεγαλύτερη μεταξύ αυτών των τιμών:

$$CFT_{p_j}(v_i) = max(\{EFT(v_k) | v_k \in succ(v_i)\})$$

4. Εφόσον υπάρχουν επεξεργαστές στους οποίους δεν έχει ανατεθεί δοκιμαστικά ο κόμβος v_i, γίνεται μετάβαση στο βήμα 2.
5. Ο κόμβος v_i ανατίθεται τελικά σε εκείνον τον επεξεργαστή p_j, για τον οποίον ισχύει:

   $$CFT_{p_j}(v_i) < CFT_{p_t}(v_i), \qquad \forall p_t \in P, \qquad p_j \neq p_t$$

   δηλαδή, στον επεξεργαστή στον οποίο το μέγιστο *EFT* μεταξύ των θυγατρικών του v_i έχει την ελάχιστη τιμή.

Στον παραπάνω αλγόριθμο, ο υπολογισμός του *CFT* δεν εξετάζει το ενδεχόμενο μη προγραμματισμένοι κόμβοι (οι οποίοι έπονται στην ουρά προτεραιότητας) να δημιουργούν εξαρτήσεις για τους θυγατρικούς του v_i (για τον οποίο υποθέσαμε ότι έχει

την υψηλότερη προτεραιότητα) με αποτέλεσμα να προκύπτουν επιπλέον καθυστερήσεις. Επίσης, κάποιοι θυγατρικοί κόμβοι ενδέχεται να έχουν χαμηλότερη προτεραιότητα έναντι άλλων θυγατρικών που ανήκουν στο κρίσιμο μονοπάτι.

Οι Bittencourt, Sakellariou και Madeira πρότειναν μια δεύτερη προσέγγιση, με ανάθεση του κόμβου v_i στον επεξεργαστή ο οποίος ελαχιστοποιεί τον σταθμισμένο μέσο όρο (weighted average) των *EFT* όλων των θυγατρικών του v_i. Ο σταθμισμένος μέσος όρος ενός κόμβου v_i που εκτελείται σε έναν επεξεργαστή p_j δίνεται από τον τύπο:

$$WCFT_{p_j}(v_i) = \frac{\sum_{v_k \in succ(v_i)} r(v_k) \cdot EFT(v_k)}{\sum_{v_k \in succ(v_i)} EFT(v_k)}$$

Επιπλέον, με δεδομένο ότι μικρές αλλαγές στη σειρά προτεραιότητας μπορούν επίσης να επιφέρουν βελτίωση στο makespan της εφαρμογής, πρότειναν την παρακάτω επέκταση του αλγορίθμου Lookahead HEFT. Με την προϋπόθεση ότι οι δύο πρώτοι μη προγραμματισμένοι έτοιμοι κόμβοι (όπως ταξινομούνται από το HEFT) είναι ανεξάρτητοι, εξετάζονται ο καθένας με τη σειρά του για ανάθεση, χρησιμοποιώντας και τις πληροφορίες που αφορούν τις θυγατρικές του εργασίες (όπως προβλέπεται από τον αλγόριθμο Lookahead HEFT). Στη συνέχεια δίνεται προτεραιότητα στον κόμβο που δίνει το καλύτερο συνολικό αποτέλεσμα (Bittencourt, Sakellariou και Madeira 2010).

5.1.6 Ο προτεινόμενος αλγόριθμος CPHEFT

Ο προτεινόμενος αλγόριθμος ανήκει στην κατηγορία list scheduling heuristics και αποδίδει καθοριστικό ρόλο στους κόμβους της κρίσιμης διαδρομής ενός γράφου, με την κατά προτεραιότητα ανάθεσή τους στους πόρους ενός ετερογενούς συστήματος.

Ωστόσο, κάθε κόμβος της κρίσιμης διαδρομής έχει συνήθως έναν ή περισσότερους γονικούς. Αυτοί οι γονικοί κόμβοι -λόγω των εξαρτήσεων- ενδέχεται να καθυστερήσουν την εκτέλεση ενός κόμβου της κρίσιμης διαδρομής και, στην περίπτωση αυτή, κρίνονται σημαντικοί κατά την κατανομή σε ουρά προτεραιότητας.

Ο προτεινόμενος αλγόριθμος, CPHEFT, αρχικά διατάσσει τους κόμβους του DAG σε μια ουρά προτεραιότητας, όπως και ο HEFT, σε φθίνουσα σειρά με βάση το μέγεθος που

χαρακτηρίζεται ως upward rank, διασφαλίζοντας και τοπολογική ταξινόμηση. Το upward rank δίνεται από τον τύπο:

$$r_u(v_i) = \overline{w_i} + \max_{v_j \in succ(v_i)} (\overline{c_{i,j}} + r_u(v_j)),$$

όπου $\overline{w_i}$ εκφράζει το μέσο υπολογιστικό κόστος του κόμβου v_i στους διαθέσιμους επεξεργαστές (πλήθους p) και $\overline{c_{i,j}}$ εκφράζει το μέσο κόστος επικοινωνίας του κόμβου v_i με τον κόμβο v_j.

Για τον υπολογισμό της κρίσιμης διαδρομής, για κάθε κόμβο v_i υπολογίζεται το μέγεθος downward rank, το οποίο δίνεται από τον τύπο:

$$r_d(v_i) = \max_{v_j \in pred(v_i)} (r_d(v_j) + \overline{w_j} + \overline{c_{j,i}}),$$

ενώ για τον κόμβο εισόδου ισχύει:

$$r_d(v_{entry}) = 0.$$

Σημειώνεται ότι ο κόμβος εισόδου v_{entry} ανήκει εξ ορισμού στην κρίσιμη διαδρομή. Επιπλέον, για κάθε κόμβο v_i, ο οποίος ανήκει στη κρίσιμη διαδρομή, ισχύει:

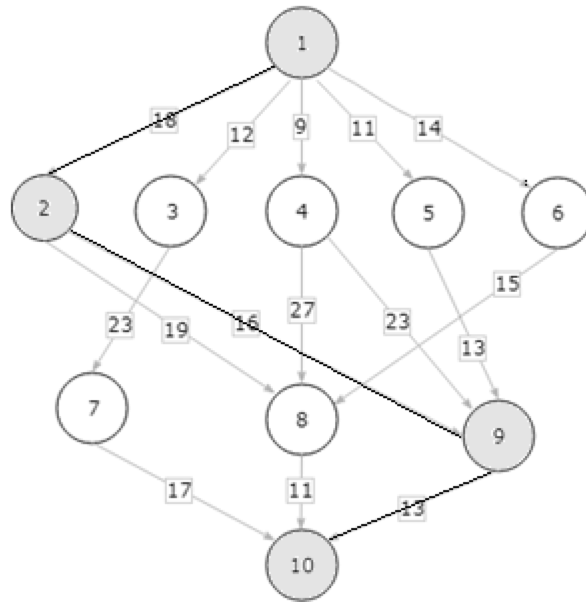
$$r_u(v_i) + r_d(v_i) = r_u(v_{entry}),$$

η οποία είναι και η μέγιστη τιμή για τα αθροίσματα $r_u(v_i) + r_d(v_i)$, για κάθε κόμβο v_i του DAG.

Τα βήματα του αλγορίθμου είναι τα εξής:

1. Οι κόμβοι της κρίσιμης διαδρομής τοποθετούνται σε μια ουρά προτεραιότητας σε φθίνουσα σειρά με βάση τις τιμές upward rank.
2. Ορίζεται μια νέα ουρά προτεραιότητας, στην οποία εισάγεται κάθε κόμβος της κρίσιμης διαδρομής, αφότου όλοι οι γονικοί του έχουν εισαχθεί. Η προτεραιότητα μεταξύ των γονικών ενός κόμβου καθορίζεται σε φθίνουσα σειρά, με βάση τις τιμές upward rank. Στην περίπτωση κόμβων με ίσες τιμές upward rank, αυτοί διατάσσονται σε αύξουσα σειρά με βάση το πλήθος των άμεσων γονικών τους.

3. Για τον κόμβο με την υψηλότερη προτεραιότητα υπολογίζεται το EFT (earliest finish-time) για κάθε επεξεργαστή, χρησιμοποιώντας insertion approach.
4. Γίνεται ανάθεση του κόμβου στον επεξεργαστή με το μικρότερο EFT.
5. Εφόσον στην ουρά προτεραιότητας υπάρχουν κόμβοι οι οποίοι δεν έχουν προγραμματιστεί, γίνεται μετάβαση στο βήμα 3.



Σχήμα 5.3. Γράφος εργασιών (DAG) – Κρίσιμο μονοπάτι

$r_d(v_{10})$	max($r_d(v_7)$ + w(7) + c(7,10), $r_d(v_8)$ + w(8) + c(8,10), $r_d(v_9)$ + w(9) + c(9,10))	**93,33**	= MAX(62,33+11,00+17; 66,67+10,00+11; 63,67+16,67+13)
$r_d(v_9)$	max($r_d(v_2)$ + w(2) + c(2,9), $r_d(v_4)$ + w(4) + c(4,9), $r_d(v_5)$ + w(5) + c(5,9))	**63,67**	= MAX(31,00+16,67+16; 22,00+9,33+23; 24,00+11,67+13)
$r_d(v_8)$	max($r_d(v_2)$ + w(2) + c(2,8), $r_d(v_4)$ + w(4) + c(4,8), $r_d(v_6)$ + w(6) + c(6,8))	**66,67**	= MAX(31,00+16,67+19; 22,00+9,33+27; 27,00+12,67+15)
$r_d(v_7)$	$r_d(v_3)$ + w(3) + c(3,7)	**62,33**	=25,00 +14,33+23
$r_d(v_6)$	$r_d(v_1)$ + w(1) + c(1,6)	**27,00**	=0+13+14
$r_d(v_5)$	$r_d(v_1)$ + w(1) + c(1,5)	**24,00**	=0+13+11
$r_d(v_4)$	$r_d(v_1)$ + w(1) + c(1,4)	**22,00**	=0+13+9
$r_d(v_3)$	$r_d(v_1)$ + w(1) + c(1,3)	**25,00**	=0+13+12
$r_d(v_2)$	$r_d(v_1)$ + w(1) + c(1,2)	**31,00**	=0+13+18
$r_d(v_1)$	0	**0,00**	=0

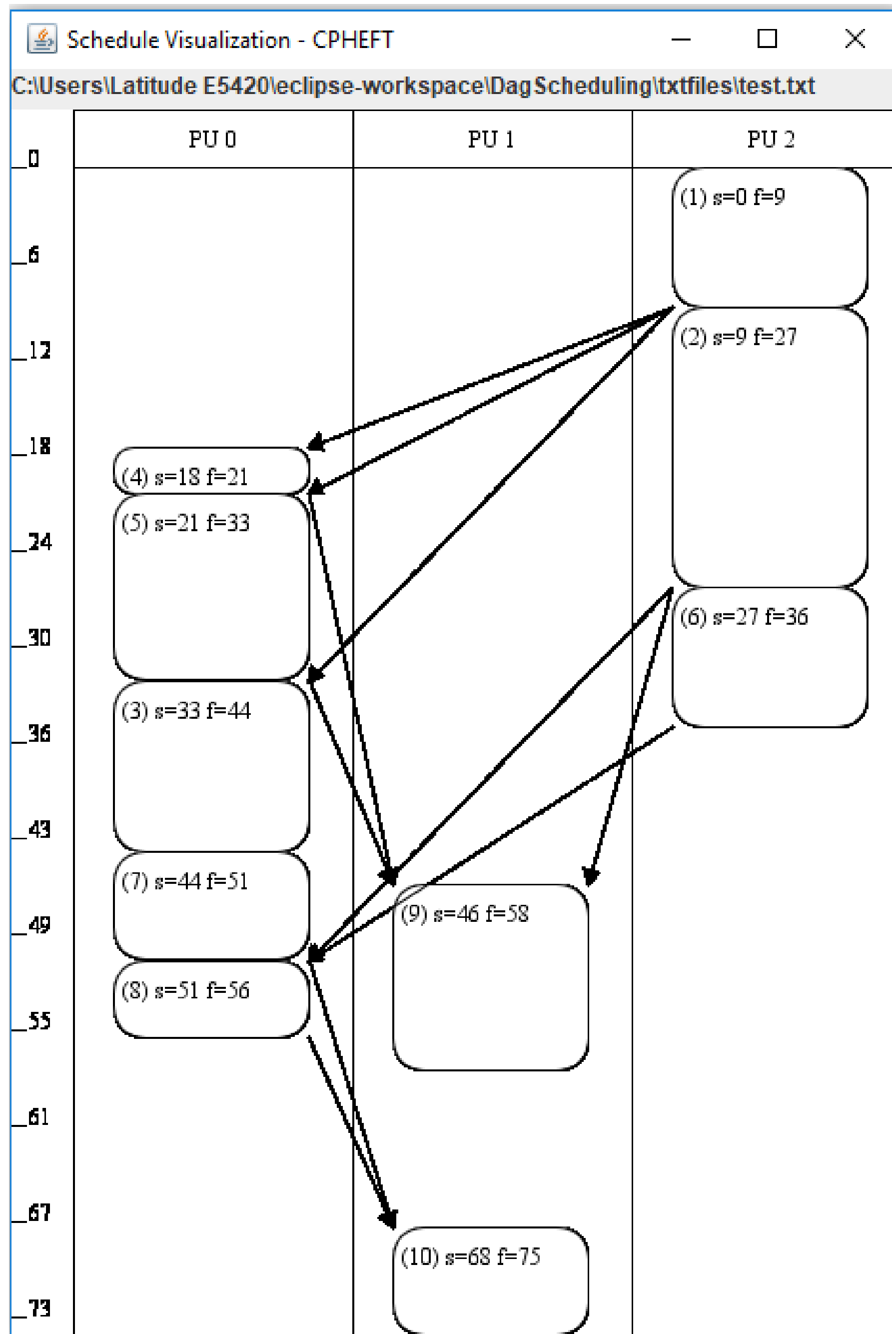
Πίνακας 5.3. Υπολογισμός των τιμών downward rank για τον προσδιορισμό του κρίσιμου μονοπατιού

κόμβοι	$r_u(v_i)$	$r_d(v_i)$	$r_u(v_i) + r_d(v_i)$	πλήθος γονικών	προτεραιότητες
v_1	108,01	0,00	**108,01**	0	1
v_2	77,01	31,00	**108,01**	1	2
v_3	80,00	25,00	**105,00**	1	6
v_4	76,67	22,00	**98,67**	1	3
v_5	69,01	24,00	**93,01**	1	4
v_6	63,34	27,00	**90,34**	1	8
v_7	42,67	62,33	**105,00**	1	7
v_8	35,67	66,67	**102,34**	3	9
v_9	44,34	63,67	**108,01**	3	5
v_{10}	14,67	93,33	**108,01**	3	10

Πίνακας 5.4. Τιμές για τον καθορισμό της ουράς προτεραιότητας στον αλγόριθμο CPHEFT

	v_1	v_2	v_4	v_5	v_9	v_3	v_7	v_6	v_8	v_{10}
EFT0	14	40	**21**	**33**	61	**44**	**51**	64	**56**	92
EFT1	16	46	26	33	**58**	71	82	74	69	**75**
EFT2	**9**	**27**	44	37	66	46	78	**36**	62	87
επεξεργαστής	***PU2***	***PU2***	***PU0***	***PU0***	***PU1***	***PU0***	***PU0***	***PU2***	***PU0***	***PU1***

Πίνακας 5.5. Τα βήματα προγραμματισμού σύμφωνα με τον αλγόριθμο CPHEFT



Σχήμα 5.4. Οπτικοποίηση προγραμματισμού με τον προτεινόμενο αλγόριθμο CPHEFT

5.2 Συσταδοποίηση (clustering)

Η συσταδοποίηση (clustering) είναι μια σημαντική κατηγορία προγραμματισμού εργασιών, η οποία χαρακτηρίζεται από δύο φάσεις. Στην πρώτη, οι κόμβοι ομαδοποιούνται σε συστάδες (clusters) με σκοπό την ανάθεση όλων των κόμβων μιας

συστάδας στον ίδιο επεξεργαστή. Στη δεύτερη φάση, οι συστάδες συγχωνεύονται έτσι ώστε το πλήθος τους να ισούται με τον αριθμό των διαθέσιμων επεξεργαστών, χαρτογραφούνται σε συγκεκριμένους επεξεργαστές και τελικά γίνεται ο προγραμματισμός των κόμβων κάθε συστάδας (Valouxis, et al. 2013).

Οι εργασίες που ανήκουν σε μια συστάδα μπορούν να επικοινωνούν με εργασίες άλλων συστάδων αμέσως μετά την ολοκλήρωση της εκτέλεσης τους. Η συσταδοποίηση είναι NP-πλήρης ως προς την ελαχιστοποίηση του παράλληλου χρόνου εκτέλεσης (Gerasoulis and Yang 1993). Με βάση τη συσταδοποίηση έχουν προταθεί πολλοί αλγόριθμοι για τον προγραμματισμό εργασιών για συστήματα πολλαπλών επεξεργαστών, όπως οι Linear Clustering Method (LCM), αλγόριθμος του Sarkar και Dominant Sequence Clustering (DSC).

Ο αλγόριθμος του Sarkar χρησιμοποιεί την τεχνική του μηδενισμού των ακμών για την συσταδοποίηση των εργασιών. Οι κόμβοι, που συνδέονται με μεγάλο κόστος επικοινωνίας, ομαδοποιούνται και εκτελούνται στον ίδιο επεξεργαστή, ελαχιστοποιώντας έτσι τον χρόνο εκτέλεσης για τον γράφο εργασιών. Για το σκοπό αυτό, αρχικά κάθε εργασία εντάσσεται σε μία ξεχωριστή συστάδα και οι ακμές ταξινομούνται σε φθίνουσα σειρά με βάση το βάρος τους. Στη συνέχεια εξετάζονται όλες οι ακμές μία προς μία, μηδενίζονται εφόσον δεν αυξάνεται το makespan και οι κόμβοι που συνδέουν εντάσσονται στην ίδια συστάδα.

Ο αλγόριθμος DSC (Dominant Sequence Clustering) βασίζεται στη συσταδοποίηση και ομαδοποιεί τους κόμβους του γραφήματος των εργασιών καθορίζοντας τις προτεραιότητές τους με βάση τα μεγέθη b-level και t-level. Επιπλέον, έχει διατυπωθεί από τον Δικαιάκο κ.α. και η άπληστη εκδοχή του αλγορίθμου DSC (Ashish Kumar Maurya and Anil Kumar Tripathi 2018).

5.2.1 Γραμμική συσταδοποίηση

Η ιδέα της γραμμικής συσταδοποίησης ορίστηκε από τους Kim και Browne και αφορά την εύρεση της κρίσιμης διαδρομής στον γράφο εργασιών και την συγχώνευση όλων των κόμβων αυτής στην ίδια συστάδα (Ashish Kumar Maurya and Anil Kumar Tripathi 2018).

Ο αλγόριθμος LC (Linear Clustering algorithm) είναι μια παραδοσιακή μέθοδος γραμμικής συσταδοποίησης που βασίζεται στην κρίσιμη διαδρομή. Αναγνωρίζει την κρίσιμη διαδρομή, αφαιρεί από το DAG τους κόμβους που ανήκουν σ' αυτή και τους εκχωρεί σε μια γραμμική συστάδα. Η διαδικασία επαναλαμβάνεται μέχρις-ότου αφαιρεθούν από το DAG όλοι οι κόμβοι. Έπειτα, κάθε συστάδα ανατίθεται σε κάποιον επεξεργαστή (Park, Shirazi and Marquis, Mapping of Parallel Tasks to Multiprocessors with Duplication 1998).

Ο αλγόριθμος ECP (Effective Critical Path) εφαρμόζει την ιδέα της συσταδοποίησης για τον χρονοπρογραμματισμό εργασιών σε σύστημα πολυεπεξεργαστών απεριόριστου πλήθους. Αρχικά, κατανέμει κάθε εργασία του γράφου εργασιών σε μία ξεχωριστή συστάδα. Σε κάθε βήμα, ο αλγόριθμος προσπαθεί να βελτιώσει την προηγούμενη συσταδοποίηση συνδυάζοντας κατάλληλες συστάδες σε μία. Η πράξη συγχώνευσης πραγματοποιείται με το μηδενισμό κόστους μιας ακμής που συνδέει δύο συστάδες, οπότε επικοινωνία τους καθίσταται τοπική. Ο κύριος στόχος του αλγορίθμου ECP είναι να ελαχιστοποιήσει το makespan, λαμβάνοντας υπόψη τους περιορισμούς προτεραιότητας. Αυτός ο αλγόριθμος χρησιμοποιεί την ιδέα του μηδενισμού ακμών στην κρίσιμη διαδρομή του γράφου εργασιών για την συσταδοποίηση των εργασιών. Υπολογίζεται επανειλημμένα μια κρίσιμη διαδρομή του γράφου εργασιών και εκτελούνται τα βήματα μηδενισμού ακμών μόνο στις ακμές της κρίσιμης διαδρομής, με στόχο τη μείωση του συνολικού χρόνου εκτέλεσης του γράφου εργασιών (makespan). Ο αλγόριθμος έχει τα ακόλουθα χαρακτηριστικά:

- μηδενίζει μια ακμή μιας κρίσιμης διαδρομής σε κάθε βήμα, παράγοντας μια νέα κρίσιμη διαδρομή,

- χρησιμοποιεί την οπισθοδρόμηση (backtracking), μετά το μηδενισμό μιας ακμής, στην περίπτωση που το τρέχον makespan είναι μεγαλύτερο από το makespan του προηγουμένου βήματος,

- ενημερώνει δυναμικά το χρονοδιάγραμμα για κάθε επεξεργαστή μέχρις ότου το makespan του γράφου εργασιών να μην αυξάνεται, και

- δίνει έμμεσα το εφικτό χρονοδιάγραμμα που προκύπτει σε κάθε βήμα της συσταδοποίησης.

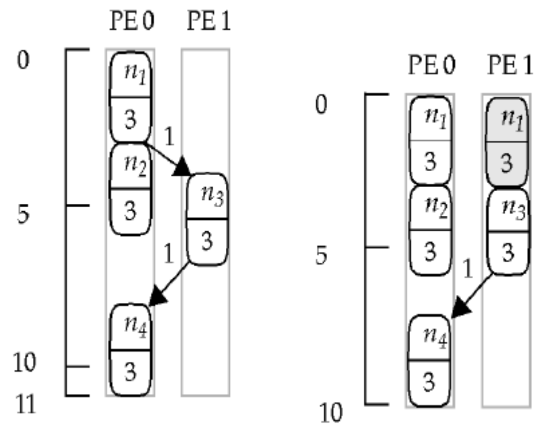
Διαφορετικά, δεν καθορίζεται το τρέχον makespan, επί του οποίου βασίζονται οι αποφάσεις συσταδοποίησης (Ashish Kumar Maurya and Anil Kumar Tripathi 2018).

5.3 Duplication

Κατά την εκτέλεση ενός γράφου εργασιών, συχνά πρέπει ένας κόμβος να περιμένει τις εισροές από τους γονικούς του κόμβους οι οποίες καθυστερούν λόγω του κόστους επικοινωνίας, ενώ παράλληλα ο επεξεργαστής εκτέλεσής του παραμένει αδρανής. Μια λύση που έχει αξιοποιηθεί για τη μείωση του κόστους επικοινωνίας και την αποφυγή μακρών διαστημάτων αναμονής, χαρακτηρίζεται ως node-duplication. Η μέθοδος duplication είναι μια σχετικά νέα προσέγγιση για την επίλυση προβλημάτων προγραμματισμού εκτέλεσης εργασιών σε σύστημα πολλαπλών επεξεργαστών. Σε αυτήν την προσέγγιση, ορισμένοι κόμβοι ενός γράφου εργασιών ανατίθενται σε περισσότερους από έναν επεξεργαστές.

Δεδομένου ότι το κόστος επικοινωνίας δύο εξαρτημένων εργασιών που εκτελούνται στο ίδιο μηχάνημα θεωρείται μηδενικό, οι περισσότερες ευρετικές μέθοδοι με βάση τη μέθοδο duplication προσπαθούν να χρησιμοποιήσουν τον χρόνο αδράνειας, που έχει απομείνει σε έναν επεξεργαστή. Αναθέτουν σ’ αυτόν τον επεξεργαστή μια εργασία, η οποία έχει ήδη προγραμματιστεί σε κάποιο άλλο πόρο, εφ’ όσον η θυγατρική της εκτελείται στον επεξεργαστή αυτό (Genez, Sakellariou, et al., Scheduling Scientific Workflows on Clouds Using a Task Duplication Approach 2018, IBM Knowledge Center n.d.). Ο στόχος είναι να μειωθεί ο ενωρίτερος χρόνος έναρξης (EST) για την θυγατρική εργασία.

Οι αλγόριθμοι προγραμματισμού, οι οποίοι βασίζονται σε duplication, δημιουργούν βραχύτερα χρονοδιαγράμματα χωρίς να θίγεται η αποτελεσματικότητα, αλλά υπερφορτώνουν τους υπολογιστικούς πόρους. Ωστόσο, μπορεί να είναι χρήσιμοι σε συστήματα, όπως δίκτυα σταθμών εργασίας, που έχουν υψηλές καθυστερήσεις επικοινωνίας και χαμηλό εύρος ζώνης, καθώς δείχνουν σημαντική βελτίωση της απόδοσης για DAGs με υψηλές τιμές CCR (Park, Shirazi and Marquis, Mapping of Parallel Tasks to Multiprocessors with Duplication 1998).



Σχήμα 5.5. Duplication (Ahmad and Kwok 1998)

6. ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΕΡΓΑΣΙΩΝ ΜΕ ΑΚΡΙΒΕΙΣ ΤΕΧΝΙΚΕΣ

6.1 Ακέραιος προγραμματισμός

Αν και οι ευρετικοί αλγόριθμοι και οι αλγόριθμοι προσέγγισης είναι ο πιο συνηθισμένος τρόπος αντιμετώπισης του προβλήματος χρονοπρογραμματισμού εργασιών, ο μαθηματικός προγραμματισμός μπορεί να αποτελέσει ανταγωνιστική εναλλακτική λύση και έχει προταθεί για την επίλυση του προβλήματος.

Ο ακέραιος προγραμματισμός είναι ένας κλάδος του μαθηματικού προγραμματισμού στον οποίο διερευνώνται προβλήματα βελτιστοποίησης (μεγιστοποίησης ή ελαχιστοποίησης) συναρτήσεων πολλών μεταβλητών. Σε ένα πρόβλημα ακέραιου προγραμματισμού ορισμένες ή όλες οι μεταβλητές περιορίζονται σε ακέραιους αριθμούς. Σε πολλές περιπτώσεις ο όρος αναφέρεται σε ακέραιο γραμμικό προγραμματισμό (ILP), στον οποίο η αντικειμενική συνάρτηση και οι περιορισμοί είναι γραμμικοί.

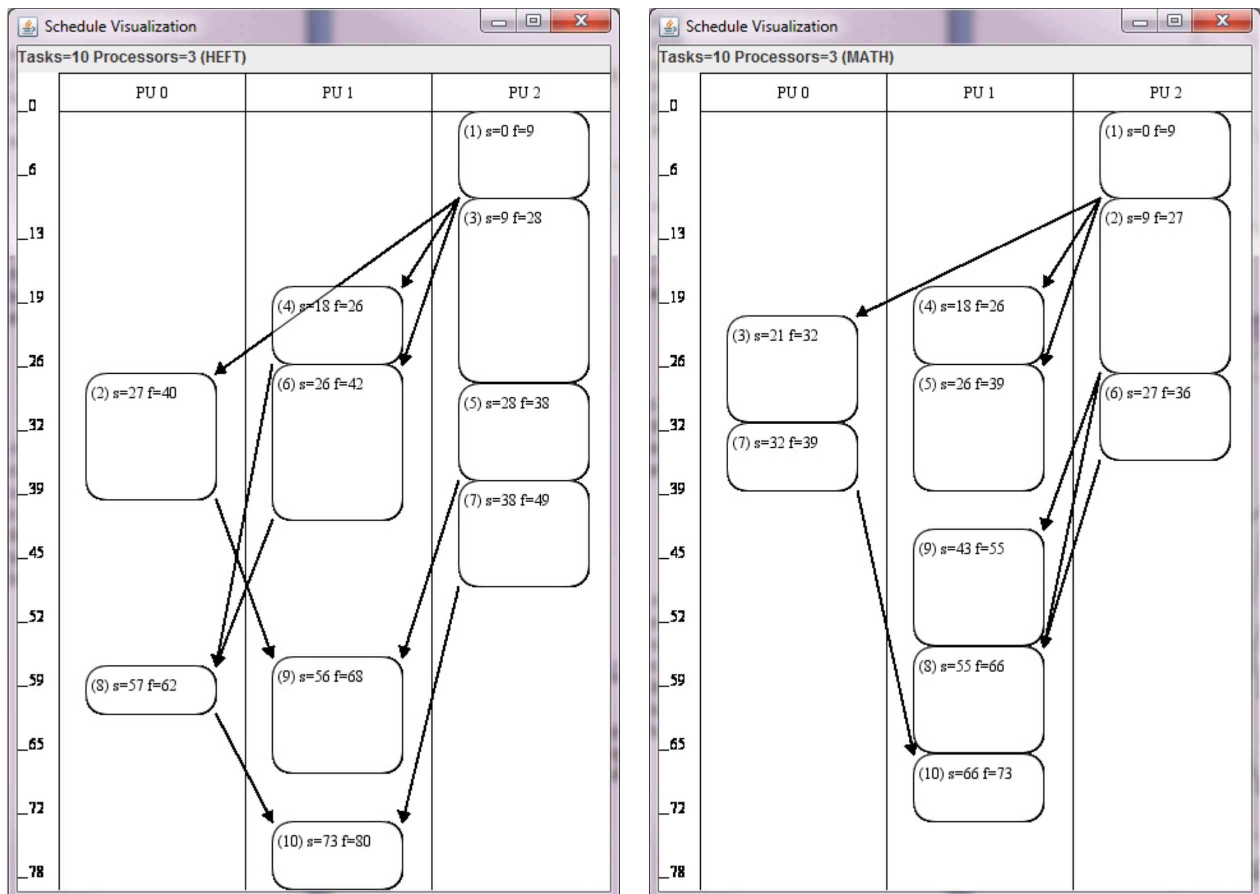
Έχουν αναπτυχθεί αρκετές μέθοδοι επίλυσης για προβλήματα ακέραιου προγραμματισμού, αλλά η απόδοση οποιασδήποτε συγκεκριμένης τεχνικής φαίνεται να εξαρτάται σε μεγάλο βαθμό από το συγκεκριμένο πρόβλημα. Οι μέθοδοι αυτές (όπως cutting planes, dynamic programming, branch and bound και άλλες) βασίζονται στην μείωση του αριθμού εφικτών λύσεων. Στην περίπτωση που οι εφικτές λύσεις είναι πολλές, μια προσέγγιση, η οποία συνίσταται σε πλήρη απαρίθμηση όλων των εφικτών λύσεων, απαιτεί έναν όγκο υπολογιστικής εργασίας, ο οποίος αναπτύσσεται εκθετικά με τον αριθμό των μεταβλητών. Μια τέτοια προσέγγιση αποδεικνύεται μη πρακτική.

Επομένως, το κεντρικό θεωρητικό πρόβλημα στον ακέραιο προγραμματισμό είναι αν μπορεί να αποφευχθεί η πλήρης απαρίθμηση εφικτών λύσεων κατά την επίλυση σχετικών προβλημάτων. Μια από τις μαθηματικές διατυπώσεις αυτού του προβλήματος είναι η ταύτιση των κλάσεων P και NP. Οι κλάσεις P και NP αποτελούνται από όλα τα προβλήματα απόφασης που μπορούν να λυθούν σε πολυωνυμικό χρόνο σε μια ντετερμινιστική (για τις κλάσεις P) ή μη ντετερμινιστική (για τις κλάσεις NP) μηχανή Turing. Ο πολυωνυμικός χρόνος αναφέρεται σε έναν αριθμό υπολογιστικών λειτουργιών

που εξαρτάται ως πολυώνυμο από το λεγόμενο "μέγεθος εισόδου" του προβλήματος. Η κλάση NP περιλαμβάνει όλες τις περιπτώσεις απόφασης για προβλήματα ακέραιου προγραμματισμού τα οποία έχουν έναν εκθετικό αριθμό εφικτών λύσεων (σε σχέση με το "μέγεθος εισόδου" του προβλήματος). Το πρόβλημα "P = NP" παραμένει προς το παρόν ανοιχτό (1988).

6.1.1 MATH

Στα πλαίσια του μαθηματικού προγραμματισμού έχουν προταθεί δύο βασικοί τύποι προσέγγισης, η δρομολόγηση (sequencing) και η επικάλυψη (overlap). Στην προσέγγιση, που χαρακτηρίζεται ως δρομολόγηση, απαιτείται ένας σημαντικός αριθμός περιορισμών για εξειδικευμένες μεταβλητές προκειμένου να καθοριστούν οι προτεραιότητες μεταξύ των εργασιών για κάθε επεξεργαστή. Στην προσέγγιση, που χαρακτηρίζεται ως επικάλυψη, οι μεταβλητές που αντιπροσωπεύουν την αλληλεπικαλυπτόμενη εκτέλεση εργασιών ενσωματώνονται στο μοντέλο.



(α) Αλγόριθμος HEFT (β) Αλγόριθμος MATH

Σχήμα 6.1. Προγραμματισμός DAG με τους HEFT και MATH

Μια τρίτη προσέγγιση, που εφαρμόζεται σε μικρού μεγέθους προβλήματα και χαρακτηρίζεται ως MATH, αναπτύχθηκε από τους Βαλουξή, Γκόγκο, Αλεφραγκή, Γούλα και Χούσο [2013] και μπορεί να θεωρηθεί ως μια παραλλαγή του μοντέλου επικάλυψης. Στο μοντέλο αυτό, δεν υπάρχουν ρητές μεταβλητές που προσδιορίζουν την επικάλυψη. Εργασίες, οι οποίες δεν συνδέονται με σχέση γονέα-παιδιού στον γράφο εργασιών, αντιμετωπίζουν το θέμα της επικάλυψης στον ίδιο επεξεργαστή μέσω εξειδικευμένων περιορισμών. Παράλληλα, εργασίες με σχέση γονέα-παιδιού δεσμεύονται να μην εκτελούνται ταυτόχρονα, μέσω περιορισμών προτεραιότητας που υπάρχουν στο μοντέλο.

Το μοντέλο MATH θεωρεί δύο σύνολα αντικειμένων. Το πρώτο είναι το σύνολο των εργασιών, V και το δεύτερο ένα σύνολο πλήρως συνδεδεμένων ετερογενών επεξεργαστών, P. Ο χρόνος εκτέλεσης κάθε εργασίας v_i σε έναν επεξεργαστή p_j δίνεται από την παράμετρο $w(v_i, p_j)$. Το κόστος επικοινωνίας μεταξύ των εργασιών v_i και v_j, αν δεν προγραμματιστούν στον ίδιο επεξεργαστή, δίνεται από την παράμετρο $c_{i,j}$. Τις μεταβλητές απόφασης του προβλήματος αποτελούν οι δυαδικές μεταβλητές $y(v_i, p_j)$, για κάθε $v_i \in V$ και $p_j \in P$ και οι ακέραιες μεταβλητές $x(v_i)$, για κάθε $v_i \in V$. Η μεταβλητή $y(v_i, p_j)$ παίρνει την τιμή 1 στην περίπτωση που η εργασία v_i ανατίθεται στον επεξεργαστή p_j και 0 σε κάθε άλλη περίπτωση. Από την άλλη πλευρά, η μεταβλητή $x(v_i)$ παίρνει ως τιμή την τιμή του χρόνου έναρξης της εκτέλεσης της εργασίας v_i στον επεξεργαστή p_j στην περίπτωση που $y(v_i, p_j) = 1$. Επιπλέον, αφού το πρόβλημα περιγράφεται ως κατευθυνόμενος μη κυκλικός γράφος (DAG), θα πρέπει να σημειωθεί, ότι το σύνολο των ακμών E αντιπροσωπεύει τους περιορισμούς προτεραιότητας μεταξύ των εργασιών, ενώ τα βάρη των ακμών αντιπροσωπεύουν τα κόστη επικοινωνίας.

Η αντικειμενική συνάρτηση του μοντέλου έχει ως στόχο την ελαχιστοποίηση του συνολικού μήκους του προγράμματος και περιγράφεται από την εξίσωση:

$$minimize\ x(v_{exit})$$

Για τους περιορισμούς ορίζονται τρεις ομάδες. Η πρώτη ορίζει ότι κάθε εργασία θα πρέπει να αντιστοιχιστεί σε έναν ακριβώς επεξεργαστή:

$$\sum_{p \in P} y(v_i, p) = 1, \quad \forall v_i \in V$$

Η δεύτερη ομάδα περιορισμών ορίζει ότι κάθε εργασία v_i εξαρτώμενη από ένα σύνολο V' άλλων εργασιών, θα πρέπει να ξεκινήσει την εκτέλεσή της όταν θα έχει ολοκληρωθεί η εκτέλεση όλων των εργασιών που ανήκουν στο V'. Επιπλέον, στην περίπτωση που οι εργασίες v_i και v_j έχουν σχέση γονέα-παιδιού και ανατίθενται σε διαφορετικούς επεξεργαστές, πρέπει επίσης να ληφθεί υπόψη το μεταξύ τους κόστος επικοινωνίας. Για να μοντελοποιηθεί αυτός ο περιορισμός εισάγονται τρεις νέες μεταβλητές:

- Η μεταβλητή $e(v_i)$ είναι ακέραια και εκφράζει τον χρόνο που απαιτείται για την εκτέλεση της εργασίας v_i, όπως προκύπτει από την εξίσωση:

$$e(v_i) = \sum_{p \in P} w(v_i, p) \cdot y(v_i, p), \quad \forall v_i \in V$$

- Η μεταβλητή $k(v_i, v_j, p)$ είναι το γινόμενο των δυαδικών μεταβλητών $y(v_i, p)$ και $y(v_j, p)$, όπου v_i και v_j έχουν σχέση γονέα-παιδιού, δηλαδή $ακμή(v_i, v_j) \in E$. Δεδομένου ότι το γινόμενο μεταξύ των μεταβλητών παραβιάζει τη γραμμικότητα του μοντέλου, η τιμή της μεταβλητής $k(v_i, v_j, p)$ προσδιορίζεται μέσω των παρακάτω σχέσεων:

$$k(v_i, v_j, p) \le y(v_i, p),$$

$$k(v_i, v_j, p) \le y(v_j, p),$$

$$k(v_i, v_j, p) \ge y(v_i, p) + y(v_j, p) - 1,$$

$$\forall p \in P, \qquad \forall v_i, v_j \in V: ακμή(v_i, v_j) \in E$$

- Η μεταβλητή $z_{i,j}$ είναι δυαδική και παίρνει την τιμή 1 στην περίπτωση που οι εργασίες v_i και v_j ανατίθενται στον επεξεργαστή p και την τιμή 0 όταν αυτές ανατίθενται σε διαφορετικούς επεξεργαστές:

$$z_{i,j} = \sum_{p \in P} k(v_i, v_j, p), \quad \forall v_i, v_j \in V: ακμή(v_i, v_j) \in E$$

Η διαφορά μεταξύ των χρόνων έναρξης της εκτέλεσης των εργασιών v_i και v_j, όταν οι εργασίες v_i και v_j έχουν σχέση γονέα-παιδιού ($ακμή(v_i, v_j) \in E$), δεν θα πρέπει να είναι μικρότερη από τον χρόνο έναρξης της εκτέλεσης της εργασίας v_i, $x(v_i)$, στον καθορισμένο επεξεργαστή αυξημένο κατά τον επιπλέον χρόνο, $c_{i,j}$, που απαιτείται για

την επικοινωνία των v_i και v_j, όταν αυτές δεν ανατίθενται στον ίδιο επεξεργαστή ($z_{i,j} = 0$). Ο περιορισμός αυτός περιγράφεται από την ανισότητα:

$$x(v_j) - x(v_i) \geq e(v_i) + c_{i,j} \cdot (1 - z_{i,j}), \qquad \forall v_i, v_j \in V: ακμή(v_i, v_j) \in E$$

Η τρίτη ομάδα περιορισμών εγγυάται ότι δύο εργασίες v_i και v_j, οι οποίες δεν έχουν σχέση γονέα-παιδιού ($ακμή(v_i, v_j) \notin E$), δεν επικαλύπτονται έτσι ώστε να μπορούν να ανατεθούν στον ίδιο επεξεργαστή ($z_{i,j} = 1$). Ο περιορισμός αυτός περιγράφεται από τις ανισότητες:

$$(1 - z_{i,j}) \cdot M + x(v_j) - x(v_i) \geq e(v_j) - M \cdot (1 - m_{i,j}), \qquad \forall v_i, v_j \in V: ακμή(v_i, v_j) \notin E$$

$$(1 - z_{i,j}) \cdot M + x(v_i) - x(v_j) \geq e(v_j) - M \cdot m_{i,j}, \qquad \forall v_i, v_j \in V: ακμή(v_i, v_j) \notin E$$

όπου $m_{i,j}$ δυαδική μεταβλητή, η οποία εξασφαλίζει την ισχύ ακριβώς μιας από τις παραπάνω ανισότητες (Valouxis, και συν. 2013).

6.1.2 MATHL

Για τα DAGs που έχουν περισσότερες από μερικές δεκάδες κόμβους, το παραγόμενο μοντέλο γίνεται πολύ μεγάλο για να λυθεί με τη χρήση ακέραιου προγραμματισμού. Σύμφωνα με τον αλγόριθμο MATHL για DAGs μεγάλου μεγέθους, οι κόμβοι ομαδοποιούνται σε επίπεδα ανάλογα με τα βήματα που απαιτούνται μέχρι τον καθένα, ξεκινώντας από τον κόμβο εισόδου. Η επίλυση γενικεύεται βαθμιαία, προγραμματίζοντας σε κάθε βήμα κόμβους που ανήκουν στο ίδιο επίπεδο. Αρχικά, παράγεται ένα υποπρόβλημα, το οποίο περιέχει μόνο κόμβους που ανήκουν στο μηδενικό επίπεδο. Στον γράφο προστίθεται προσωρινά ένας εικονικός κόμβος και συνδέεται με όλους τους κόμβους του μηδενικού επιπέδου. Σε αυτό το βήμα, η τιμή που πρέπει να ελαχιστοποιηθεί είναι η ώρα έναρξης αυτού του εικονικού κόμβου. Επομένως, κατασκευάζεται ένα μαθηματικό μοντέλο που περιλαμβάνει μεταβλητές οι οποίες αναφέρονται μόνο στους κόμβους του μηδενικού επιπέδου. Με την επίλυση αυτού του μοντέλου διαμορφώνεται μια μερική λύση που προγραμματίζει βέλτιστα όλους τους κόμβους αυτού του επιπέδου. Στη συνέχεια, προγραμματίζονται οι κόμβοι του επιπέδου 1, υπό την προϋπόθεση ότι ο προγραμματισμός των κόμβων που ανήκουν στο

προηγούμενο επίπεδο δεν μπορεί να αλλάξει. Για το επίπεδο 1 και κάθε επόμενο επίπεδο, το μαθηματικό μοντέλο το οποίο θα δημιουργηθεί ορίζει μεταβλητές μόνο για τους κόμβους που ανήκουν στο υπό εξέταση επίπεδο, ενώ οι μεταβλητές που θα αναφέρονται σε προηγούμενα επίπεδα έχουν συγκεκριμένες τιμές. Η διαδικασία συνεχίζεται μέχρι να προσπελαστεί ο κόμβος εξόδου.

Η διαδικασία που περιγράφεται παραπάνω δεν εγγυάται τη βέλτιστη λύση στο πρόβλημα. Αυτό είναι εμφανές από το γεγονός ότι, ενώ επιλύεται κάθε επίπεδο, δεν λαμβάνεται υπόψη η ύπαρξη κόμβων που ανήκουν σε επόμενα επίπεδα. Παρόλα αυτά, τυπικά παράγονται ικανοποιητικές λύσεις σε αποδεκτό χρόνο, δεδομένου ότι τα υποπροβλήματα που επιλύονται από τον μαθηματικό επιλυτή έχουν σημαντικά μικρότερα μεγέθη και η επεξεργασία τους μπορεί να ολοκληρωθεί σε λίγα δευτερόλεπτα. Όπως αποδεικνύεται για τυχαίους γράφους μέχρι 1000 κόμβων η παραπάνω προσέγγιση (MATHL) υπερβαίνει κατά πολύ σε απόδοση τους αλγορίθμους HEFT και HEFTLA (Valouxis, et al. 2013).

6.2 Προγραμματισμός με περιορισμούς

Ο προγραμματισμός με περιορισμούς (CP) είναι μια ισχυρή προσέγγιση για την επίλυση συνδυαστικών προβλημάτων αναζήτησης, που βασίζονται σε ένα ευρύ φάσμα τεχνικών από την τεχνητή νοημοσύνη, την επιχειρησιακή έρευνα, τη θεωρία γράφων κλπ.

Στον προγραμματισμό με περιορισμούς, ο χρήστης δηλώνει τους περιορισμούς, μοντελοποιώντας το πρόβλημα ως ένα πρόβλημα ικανοποίησης περιορισμών (CSP) και χρησιμοποιείται ένας σχετικός επιλυτής γενικού σκοπού για την επίλυσή του. Συγκεκριμένα, σε ένα πρόβλημα ικανοποίησης περιορισμών (CSP) δηλώνονται ένα σύνολο μεταβλητών απόφασης, ένα σύνολο τιμών για κάθε μεταβλητή και ένα σύνολο περιορισμών που ορίζονται μεταξύ αυτών των μεταβλητών. Οι επιλυτές προβλημάτων ικανοποίησης περιορισμών δέχονται ένα πραγματικό πρόβλημα, που περιγράφεται από τα όρια των μεταβλητών απόφασης και τους περιορισμούς, και επιστρέφουν μια τιμή για κάθε μεταβλητή, τέτοια ώστε να ικανοποιούνται οι περιορισμοί.

Οι επεκτάσεις αυτού του πλαισίου μπορεί να περιλαμβάνουν, για παράδειγμα, την εύρεση όλων των λύσεων, την εύρεση βέλτιστων λύσεων ως προς ένα ή περισσότερα κριτήρια βελτιστοποίησης, την αντικατάσταση (μερικών ή όλων) των περιορισμών με προτιμήσεις, και την θεώρηση μιας ρύθμισης όπου οι περιορισμοί κατανέμονται μεταξύ διαφόρων παραγόντων (Rossi, Van Beek and Walsh 2006).

Μία από τις βασικές ιδέες του προγραμματισμού με περιορισμούς είναι ότι οι περιορισμοί μπορούν να χρησιμοποιηθούν "ενεργά" για να μειώσουν το υπολογιστικό έργο που απαιτείται για την επίλυση συνδυαστικών προβλημάτων. Συνεπώς, οι περιορισμοί δεν χρησιμοποιούνται μόνο για τη δοκιμή της εγκυρότητας μιας λύσης, όπως συμβαίνει στις συμβατικές γλώσσες προγραμματισμού, αλλά και σε έναν ενεργό τρόπο για να αφαιρεθούν τιμές από τα αντίστοιχα σύνολα τιμών, να προκύψουν νέοι περιορισμοί και να εντοπιστούν ασυνέπειες. Αυτή η διαδικασία της ενεργητικής χρήσης των περιορισμών για να φτάσουμε σε συγκεκριμένα συμπεράσματα ονομάζεται constraint propagation.

Με άλλα λόγια, αυτό που ονομάζουμε constraint propagation είναι μια διαδικασία αλληλεπίδρασης μεταξύ της μείωσης του πεδίου τιμών μιας μεταβλητής απόφασης και όλων των περιορισμών που αναφέρονται σε αυτήν τη μεταβλητή. Αυτή η διαδικασία μπορεί να οδηγήσει σε περισσότερες μειώσεις στο πεδίο τιμών της μεταβλητής. Αυτές οι μειώσεις στο πεδίο τιμών, με τη σειρά τους, κοινοποιούνται στους συσχετιζόμενους περιορισμούς. Η διαδικασία συνεχίζεται έως ότου δεν μπορούν πλέον να μειωθούν τα πεδία τιμών των μεταβλητών ή συμβεί το πεδίο τιμών μιας μεταβλητής να εξισωθεί με το κενό σύνολο, οπότε παρουσιάζεται αποτυχία. Αν ένα κενό πεδίο τιμών μεταβλητής εμφανιστεί στην αρχή της διαδικασίας, αυτό σημαίνει ότι το πρόβλημα δεν έχει λύση (IBM Knowledge Center n.d.).

Στη συνολική συμπεριφορά ενός συστήματος προγραμματισμού με περιορισμούς υπογραμμίζεται το γεγονός ότι ο ορισμός του προβλήματος, η διάδοση των περιορισμών και η αναζήτηση λύσεων με ευρετικές τεχνικές και στρατηγική οπισθοδρόμησης (backtracking) διαχωρίζονται με σαφήνεια. Αυτός ο διαχωρισμός βασίζεται σε θεμελιώδεις αρχές του προγραμματισμού με περιορισμούς, που ορίστηκαν στα τέλη της δεκαετίας του 1970 και στις αρχές της δεκαετίας του 1980. Η διάκριση μεταξύ της

λογικής αναπαράστασης των περιορισμών και του ελέγχου της χρήσης τους είναι σύμφωνη με την εξίσωση του Kowalski για τον λογικό προγραμματισμό, Αλγόριθμος = Logic + Control. Μια άλλη σημαντική αρχή που χρησιμοποιείται στον προγραμματισμό με περιορισμούς είναι η αποκαλούμενη "αρχή τοπικότητας", η οποία δηλώνει ότι η διαδικασία constraint propagation πρέπει να έχει κατά το δυνατόν τοπικό χαρακτήρα, ανεξάρτητα από το αν υπάρχουν ή δεν υπάρχουν άλλοι περιορισμοί.

Ενώ το γενικό πρόβλημα ικανοποίησης περιορισμών (CSP) είναι NP-πλήρες, η διάδοση περιορισμών είναι συνήθως μη-πλήρης. Ειδικότερα, η διάδοση των περιορισμών δεν μπορεί να ανιχνεύσει όλες τις ασυνέπειες. Συνεπώς, πρέπει να εκτελεστεί κάποιο είδος αναζήτησης για να προσδιοριστεί εάν το επικείμενο στιγμιότυπο CSP έχει λύση ή όχι. Συχνά, η αναζήτηση πραγματοποιείται μέσω ενός αλγορίθμου αναζήτησης δέντρου. Τα δύο βασικά στοιχεία ενός αλγορίθμου αναζήτησης δέντρου είναι (i) ο τρόπος να προχωρήσουμε «προς τα εμπρός», δηλαδή ο ορισμός για το ποιες αποφάσεις λαμβάνονται και σε ποιο σημείο της αναζήτησης και (ii) ο τρόπος να πάμε «προς τα πίσω», δηλαδή ο ορισμός της στρατηγικής οπισθοδρόμησης (backtracking) που δηλώνει την συμπεριφορά του αλγορίθμου κατά την ανίχνευση μιας αντίφασης.

Η περιγραφή των αποφάσεων που πρέπει να ληφθούν και σε ποιο σημείο της αναζήτησης αναφέρεται συχνά ως ευρετική αναζήτηση. Γενικά, οι αποφάσεις που λαμβάνονται αντιστοιχούν στην προσθήκη επιπλέον περιορισμών. Ως εκ τούτου, κατά τη διάρκεια της έρευνας, η διάδοση των περιορισμών (constraint propagation) κρίνει τον συνδυασμό των αρχικών περιορισμών και των περιορισμών που απορρέουν από τις αποφάσεις που λαμβάνονται. Κατά την ανίχνευση μιας αντίφασης, αποδεικνύεται ότι δεν υπάρχει εφικτή εκχώρηση τιμών στις μεταβλητές σύμφωνη με τα αρχικά δεδομένα του CSP και τις ευρετικές αποφάσεις που έχουν ληφθεί. Στις περιπτώσεις αυτές, η πιο συχνά χρησιμοποιούμενη στρατηγική οπισθοδρόμησης είναι η depth first chronological backtracking, σύμφωνα με την οποία η τελευταία απόφαση αίρεται και επιβάλλεται ένας άλλος περιορισμός (Baptiste, Le Pape and Nuijten 2001).

7. ΠΕΙΡΑΜΑΤΑ

7.1 Θέματα υλοποίησης

7.1.1 Scheduling Length Ratio (SLR)

Το μέγεθος, το οποίο χρησιμοποιείται συχνότερα για την αξιολόγηση ενός προγράμματος για την εκτέλεση των εργασιών ενός DAG, είναι το makespan και ορίζεται από την εξίσωση:

$$makespan = max(AFT(v_{exit}))$$

Ωστόσο, το makespan είναι ένα μέγεθος το οποίο για διαφορετικά DAGs παρουσιάζει μεγάλη διαφοροποίηση, καθιστώντας δύσκολη τη σύγκριση. Προκειμένου να συγκρίνουμε προγράμματα για την εκτέλεση εργασιών, οι οποίες ορίζουν DAGs με διαφορετικές τοπολογίες, απαιτούνται μεγέθη τέτοια ώστε τα αποτελέσματα να μην είναι ευαίσθητα ως προς το μέγεθος του DAG. Το μέγεθος Schedule Length Ratio (SLR) πληροί αυτή την προϋπόθεση και αποτελεί την εξομάλυνση του makespan στον προσδιορισμό του βέλτιστου-εφικτού μήκους προγράμματος για την εκτέλεση ενός DAG σε δεδομένο ετερογενές περιβάλλον. Θεωρητικά, θα μπορούσε να χρησιμοποιηθεί η βέλτιστη λύση γι αυτή την αξιολόγηση. Καθώς όμως η εύρεση του βέλτιστου makespan είναι NP-πλήρης, συνήθως χρησιμοποιείται σε αντικατάσταση το εκτιμώμενο μήκος της κρίσιμης διαδρομής (Zhang, Koelbel και Kennedy 2007). Ορίζουμε:

$$SLR = makespan/CPIC,$$

όπου CPIC (Critical Path Including Communication) είναι το κόστος της κρίσιμης διαδρομής συμπεριλαμβανομένου του κόστους επικοινωνίας μεταξύ των αντίστοιχων κόμβων. Μια σχετικά μικρότερη τιμή SLR είναι ενδεικτική ενός καλύτερου χρονοδιαγράμματος.

7.1.2 Speedup

Το speedup ορίζεται ως ο λόγος του χρόνου που απαιτείται για την σειριακή εκτέλεση όλων των εργασιών, προς τον χρόνο που απαιτείται από τον παράλληλο αλγόριθμο για

την εκτέλεση των ίδιων εργασιών (makespan). Η τιμή του αριθμητή προσδιορίζεται με την υπόθεση ότι όλες οι εργασίες του γράφου ανατίθενται για σειριακή εκτέλεση σε μοναδικό πόρο, ο οποίος ελαχιστοποιεί το συνολικό υπολογιστικό κόστος:

$$speedup = \min_{p_j \in P}\left(\sum_{v_i \in V} w(v_i, p_j)\right)\Big/ makespan$$

7.1.3 Efficiency

Ως αποδοτικότητα ορίζεται ο λόγος:

$$efficiency \ = speedup/p$$

όπου p είναι το πλήθος των επεξεργαστών.

7.1.4 Processor Utilization

Σε ένα σύστημα πολλαπλών επεξεργαστών σημαντικό ρόλο παίζει η κατανομή των εργασιών στους διάφορους επεξεργαστές. Το μέγεθος utilization μετρά για κάθε επεξεργαστή το ποσοστό του χρόνου κατά τον οποίο απασχολήθηκε ο επεξεργαστής, προς τον συνολικό χρόνο εκτέλεσης του αλγορίθμου (makespan):

$$utilization(p_j) = \left(\sum w(v_i, p_j)/makespan\right) * 100,$$

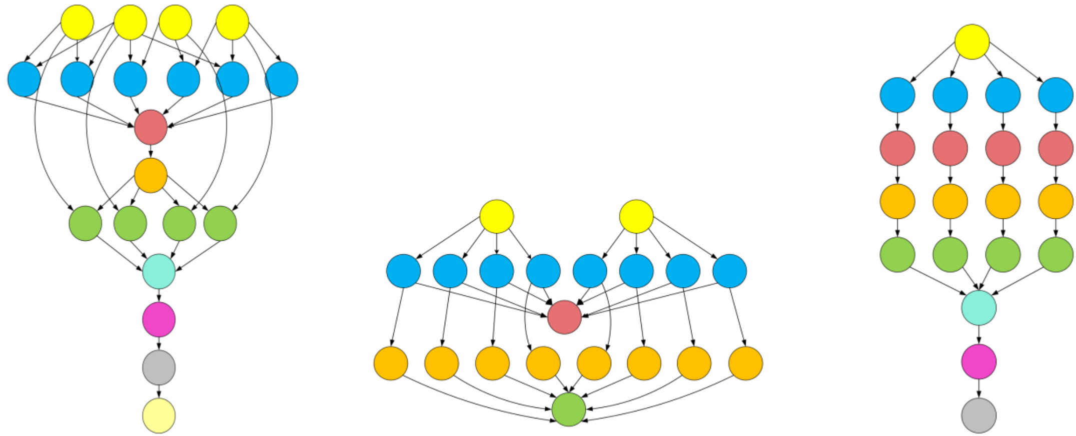
όπου $p_j \in P$ και v_i οι εργασίες που έχουν ανατεθεί στον p_j.

7.2 Γραφήματα επιστημονικών ροών εργασίας (scientific workflows)

Για την αξιολόγηση αλγορίθμων, που διαχειρίζονται την βέλτιστη εκτέλεση εργασιών σε παράλληλα ετερογενή περιβάλλοντα, μπορούν να χρησιμοποιηθούν γνωστές εφαρμογές που χρησιμοποιούνται σε πραγματικά προβλήματα. Τέτοιες εφαρμογές είναι η απαλοιφή Gauss (ένας αλγόριθμος για την επίλυση συστημάτων γραμμικών εξισώσεων),

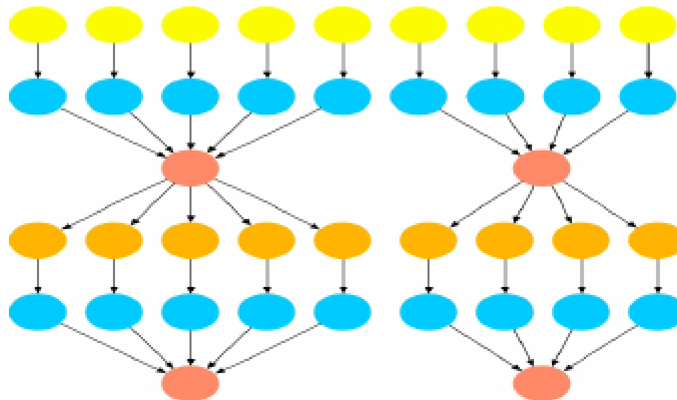
ο μετασχηματισμός Fast Fourier και οι συστολικοί πίνακες. Επιπλέον, μπορούν να χρησιμοποιηθούν σύνθετα δεδομένα προερχόμενα από επιστημονικές ροές πραγματικού κόσμου, οι οποίες είναι διαθέσιμες σε διάφορα επιστημονικά πεδία όπως στην αστρονομία, την βιολογία κλπ. Οι επιστημονικές ροές εργασιών επιτρέπουν σύνθετες επιστημονικές αναλύσεις με τις οποίες μπορούν να επικυρωθούν τα αποτελέσματα. Οι μεγάλης κλίμακας επιστημονικές ροές εργασιών εκτελούνται σε εξίσου σύνθετους, παράλληλους και κατανεμημένους πόρους, οι οποίοι πρέπει να παρακολουθούνται σε πραγματικό χρόνο, για την ανίχνευση και αυτόματη επίλυση προβλημάτων (Genez, Sakellariou, et al., Scheduling Scientific Workflows on Clouds Using a Task Duplication Approach 2018) (Deelman, et al. 2011). Ενδεικτικά αναφέρονται:

- Από την αστρονομία, η εφαρμογή Montage ως ένα εργαλείο ανοιχτού κώδικα, που δημιουργήθηκε από τη NASA/IPAC για τη παραγωγή προσαρμοσμένων ψηφιδωτών του ουρανού. Οι ροές εργασιών του Montage περιστρέφουν και εξομαλύνουν τα επίπεδα πολλών εικόνων εισόδου, οι οποίες είναι στη μορφή FITS (Flexible Image Transport System), και τις επανασχεδιάζουν σε σφαιρική και υπερβολική επιφάνεια για να σχηματίσουν το ψηφιδωτό εξόδου, με στόχο τη βαθύτερη κατανόηση του τμήματος του ουρανού που μελετάται. Η εφαρμογή Montage, ως ροή εργασιών, μπορεί να εκτελεστεί σε περιβάλλοντα Grid όπως το TeraGrid.
- Από την βιοπληροφορική, η ροή εργασιών Epigenomics, ένας αγωγός επεξεργασίας δεδομένων, που χρησιμοποιεί το σύστημα διαχείρισης ροής εργασιών Pegasus για την αυτοματοποίηση της εκτέλεσης ακολουθίας λειτουργιών σε σχέση με το γονιδίωμα. Τα δεδομένα αλληλουχίας DNA, που απαιτούνται για τη ροή εργασιών, παράγονται από το σύστημα γενετικού αναλυτή Illumina-Solexa και χωρίζονται σε πολλά τμήματα που μπορούν να λειτουργούν παράλληλα. Τα δεδομένα σε κάθε κομμάτι μετατρέπονται σε μορφή αρχείου, που μπορεί να χρησιμοποιηθεί από το σύστημα Maq για την χαρτογράφηση της επιγενετικής κατάστασης των ανθρώπινων κυττάρων.



Σχήμα 7.1. Montage, Cybershake, Epigenomics

- Από τη Σεισμολογία, η ροή εργασιών Cybershake, η οποία χρησιμοποιείται από το Σεισμολογικό Κέντρο της Νότιας Καλιφόρνιας (SCEC) με την τεχνική Probabilistic Seismic Analysis Risk (PSHA), για να διεξάγει υπολογισμούς πιθανής σεισμικής επικινδυνότητας. Δεδομένης μιας περιοχής ενδιαφέροντος, υπολογίζονται συνθετικά σεισμογραφήματα βάσει δονήσεων που έχουν προβλεφθεί.



Σχήμα 7.2. LIGO

- Από την αστροφυσική, το LIGO (Laser Interferometer Gravitational Wave Observatory), το οποίο προσπαθεί να ανιχνεύσει τα βαρυτικά κύματα που προβλέπονται από τη θεωρία γενικής σχετικότητας του Αϊνστάιν. Η ροή εργασιών Inspiral Analysis της LIGO χρησιμοποιείται για τη δημιουργία και την ανάλυση βαρυτικών κυματομορφών από δεδομένα, που προκύπτουν από την συσσωμάτωση συμπαγών δυαδικών συστημάτων, όπως δυαδικά αστέρια νετρονίων και μαύρες τρύπες (Juve, Chervenak, et al., Characterizing and Profiling Scientific Workflows 2014).

7.3 Αυτόματη δημιουργία συνθετικών γράφων

Η αξιολόγηση με τυχαία παραγόμενους γράφους είναι ένας κοινός τρόπος σύγκρισης σχετικών αλγορίθμων στην έρευνα. Η προσέγγιση αυτή δεν μπορεί να δείξει πώς θα συμπεριφερόταν ένας αλγόριθμος σε σενάρια πραγματικού κόσμου. Ωστόσο, τα πειράματα με τυχαία παραγόμενους γράφους μπορούν να εφαρμόζονται σε μια μεγαλύτερη ποικιλία δομών DAG, γεγονός το οποίο συμβάλλει στην πληρότητα της αξιολόγησης.

Στα πλαίσια αυτής της εργασίας αναπτύχθηκε μια γεννήτρια γράφων εργασιών. Η ανάπτυξη της γεννήτριας βασίστηκε στην γεννήτρια τυχαίων γράφων για την παραγωγή χρονοδιαγραμμάτων με βέλτιστες λύσεις του Kwok (Kwok και Ahmad, Benchmarking and comparison of the task graph scheduling algorithms 1999).

Αρχικά, η γεννήτρια παραγωγής DAGs διαθέτει σε κάθε επεξεργαστή τμήματα ίσου εύρους, τα λεγόμενα timelines. Στη συνέχεια, κάθε timeline χωρίζεται σε τυχαίο αριθμό τμημάτων τυχαίου μεγέθους. Το μήκος κάθε τμήματος υποδηλώνει το κόστος υπολογισμού της αντίστοιχης εργασίας. Έπειτα, οι εργασίες συνδέονται τυχαία σε αυστηρά διατεταγμένα ζεύγη. Το κόστος επικοινωνίας για κάθε ζεύγος εργασιών, με σχέση γονέα-παιδιού, υπολογίζεται ως το γινόμενο του μέσου υπολογιστικού κόστους των εργασιών του ζεύγους στους διαθέσιμους επεξεργαστές του συστήματος επί μια τυχαία παραγόμενη τιμή του λόγου CCR (Communication to Computation Ratio).

Οι παράμετροι εισόδου της γεννήτριας, οι οποίες επηρεάζουν τη δομή του προκύπτοντος DAG, είναι:

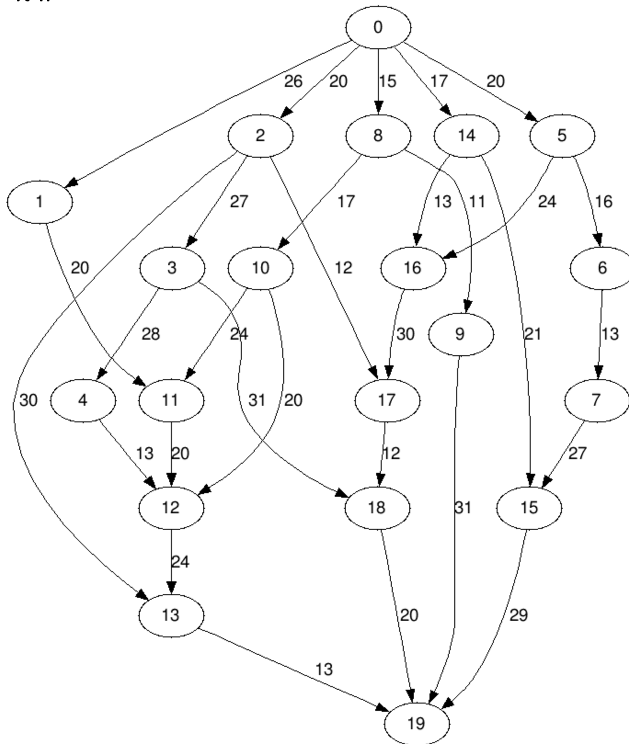
- το πλήθος των επεξεργαστών (processors), το οποίο καθορίζει και τον αρχικό αριθμό των timelines
- το πλήθος των εργασιών (tasks), το οποίο διαιρούμενο με το πλήθος των επεξεργαστών ορίζει το άνω όριο για την παραγωγή τυχαίων αριθμών· οι αριθμοί αυτοί εκφράζουν το πλήθος τμημάτων για κάθε timeline
- η παράμετρος fanout, η οποία παριστάνει το πλήθος εξαρτήσεων στον γράφο εργασιών

- η παράμετρος maxdist, η οποία ρυθμίζει το επίπεδο παραλληλισμού (υψηλότερες τιμές επιτρέπουν υψηλότερο παραλληλισμό του DAG).

Για κάθε DAG, που παράγεται από την γεννήτρια, δημιουργούνται δύο αρχεία τυποποιημένης μορφής, όπως φαίνεται στα σχήματα 7.3. και 7.4. Με την online εκτέλεση του πρώτου αρχείου, το DAG μπορεί να απεικονιστεί, όπως επίσης φαίνεται στο σχήμα 7.5. Στο δεύτερο αρχείο, περιγράφονται τα στοιχεία του παραγόμενου DAG, δηλαδή το πλήθος εργασιών, οι διαθέσιμοι πόροι στους οποίους θα ανατεθούν οι εργασίες, το υπολογιστικό κόστος κάθε εργασίας ανά επεξεργαστή, καθώς και οι εξαρτήσεις μεταξύ των εργασιών και τα αντίστοιχα κόστη επικοινωνίας ως λίστα ακμών. Το αρχείο αυτό προσπελάζεται από τον κώδικα που υλοποιεί τον αλγόριθμο CPHEFT.

```
digraph t20p4 {
0 -> 1 [ label = 26 ];
0 -> 2 [ label = 20 ];
0 -> 5 [ label = 20 ];
0 -> 8 [ label = 15 ];
0 -> 14 [ label = 17 ];
1 -> 11 [ label = 20 ];
2 -> 3 [ label = 27 ];
2 -> 13 [ label = 30 ];
2 -> 17 [ label = 12 ];
3 -> 4 [ label = 28 ];
3 -> 18 [ label = 31 ];
4 -> 12 [ label = 13 ];
5 -> 6 [ label = 16 ];
5 -> 16 [ label = 24 ];
6 -> 7 [ label = 13 ];
7 -> 15 [ label = 27 ];
8 -> 9 [ label = 11 ];
8 -> 10 [ label = 17 ];
9 -> 19 [ label = 31 ];
10 -> 11 [ label = 24 ];
10 -> 12 [ label = 20 ];
11 -> 12 [ label = 20 ];
12 -> 13 [ label = 24 ];
13 -> 19 [ label = 13 ];
14 -> 15 [ label = 21 ];
14 -> 16 [ label = 13 ];
15 -> 19 [ label = 29 ];
16 -> 17 [ label = 30 ];
17 -> 18 [ label = 12 ];
18 -> 19 [ label = 20 ];
 }
```

Σχήμα 7.3.



Σχήμα 7.5.

```
Processors:4
Tasks:20
# task computation cost on each processor
0 14 9 8 12
1 30 36 14 10
2 39 25 33 28
3 15 7 17 4
4 14 7 10 6
5 6 10 10 9
6 12 14 15 4
7 45 75 54 39
8 12 16 27 27
9 7 10 12 14
10 6 10 10 8
11 5 4 13 12
12 14 32 32 17
13 25 22 20 9
14 6 3 15 13
15 10 7 4 14
16 38 28 17 51
17 13 23 11 19
18 4 5 13 11
19 2 3 2 15
Dependencies:30
# from_task_id to_task_id weight
0 1 26
0 2 20
0 5 20
0 8 15
0 14 17
1 11 20
2 3 27
2 13 30
2 17 12
3 4 28
3 18 31
4 12 13
5 6 16
5 16 24
6 7 13
7 15 27
8 9 11
8 10 17
9 19 31
10 11 24
10 12 20
11 12 20
12 13 24
13 19 13
14 15 21
14 16 13
15 19 29
16 17 30
17 18 12
18 19 20
```

Σχήμα 7.4.

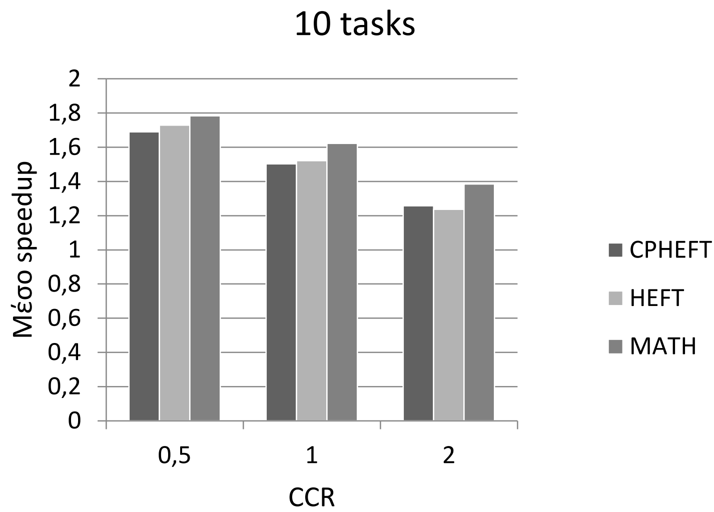
7.4 Αποτελέσματα πειραμάτων

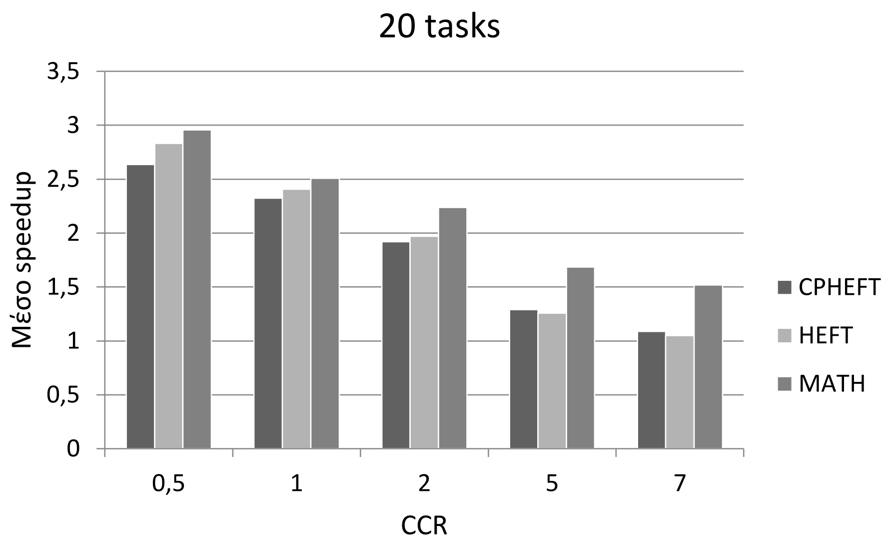
Στα πλαίσια προσομοίωσης με την γεννήτρια τυχαίων γράφων δημιουργήθηκε, αρχικά, ένα μεγάλο σύνολο DAGs με διαφορετικά χαρακτηριστικά. Στη συνέχεια, καθένα από τα παραγόμενα DAGs εκτελέστηκε από τους επιλυτές MATH, HEFT και CPHEFT και υπολογίστηκαν μεγέθη για την αξιολόγηση της σχετικής τους απόδοσης.

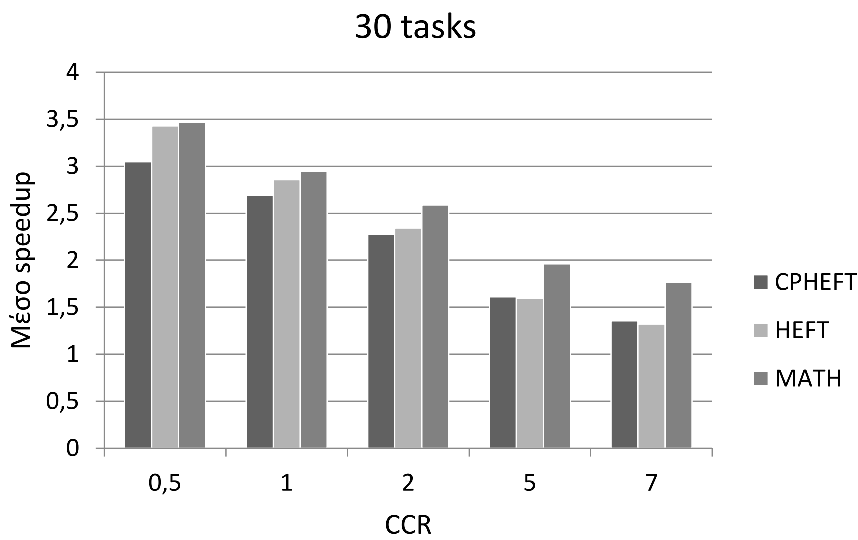
Τα διαφορετικά χαρακτηριστικά στα σύνολα των γράφων, που παρήχθησαν, ήταν το πλήθος των εργασιών σε κάθε γράφο, το πλήθος των ετερογενών επεξεργαστών στους οποίους ανατέθηκαν, καθώς και ο λόγος CCR (communication cost to computation cost ratio). Οι διαφορετικές τιμές για το πλήθος εργασιών ανήκουν στο σύνολο {10, 20, 30, 40}, το πλήθος των διαθέσιμων πόρων περιορίστηκαν στο σύνολο {2, 3, 4, 5, 6, 7, 8}, ενώ οι τιμές του CCR για τις οποίες εξετάστηκε η απόδοση των αλγορίθμων ανήκουν στο σύνολο {0,5, 1, 2, 5, 7}. Για κάθε συνδυασμό διαφορετικών χαρακτηριστικών, παρήχθησαν και εκτελέστηκαν 20 ως προς το πλήθος DAGs και υπολογίστηκαν οι μέσοι όροι των μεγεθών που επέστρεψαν οι επιλυτές.

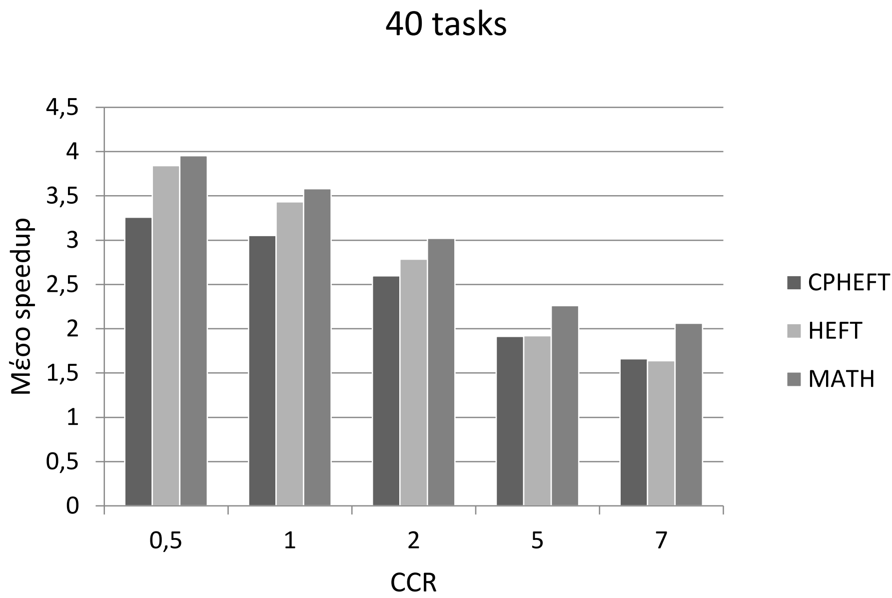
Παρότι, το μέγεθος το οποίο χρησιμοποιείται συχνότερα για την αξιολόγηση ενός αλγορίθμου για τον προγραμματισμό των εργασιών ενός DAG είναι το makespan, στην παρούσα εργασία, ως βασικό κριτήριο αξιολόγησης επιλέχτηκε το speedup. Το speedup είναι ένα μέγεθος, το οποίο δεν παρουσιάζει μεγάλη διαφοροποίηση για διαφορετικά DAGs. Μια σχετικά μεγαλύτερη τιμή του speedup είναι ενδεικτική μιας καλύτερης επίλυσης. Επίσης, εξετάστηκε το μέγεθος utilization, για τον προσδιορισμό της κατανομής των εργασιών στους διάφορους επεξεργαστές από τους διαφορετικούς αλγορίθμους.

Επιπλέον, ο αλγόριθμος MATH, ο οποίος συμμετέχει στη σύγκριση, διασφαλίζει την βέλτιστη λύση για ένα μεγάλο αριθμό από τα τυχαία προβλήματα και συγκεκριμένα εκείνα που χαρακτηρίζονται από μικρό αριθμό εργασιών και πόρων.

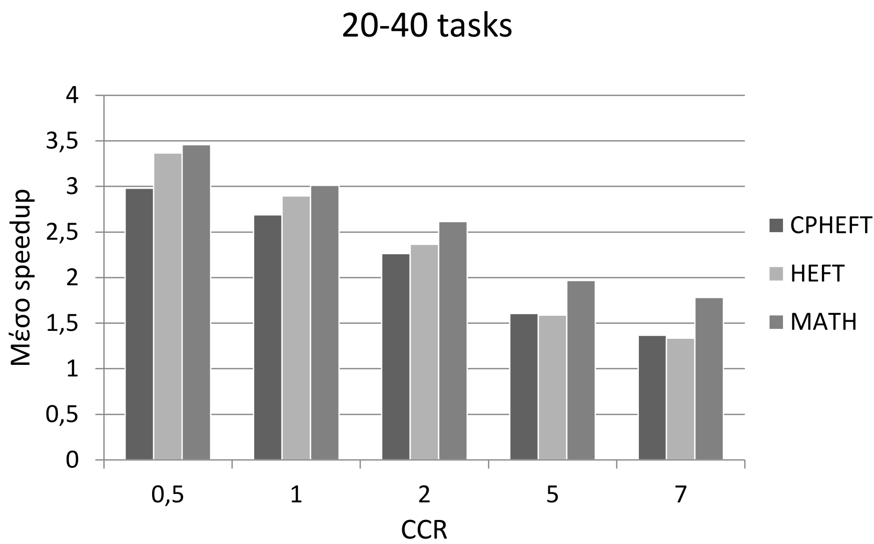

10 tasks
Μέσο speedup
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0,5
1
2
CCR
CPHEFT
HEFT
MATH


20 tasks
Μέσο speedup
3,5
3
2,5
2
1,5
1
0,5
0
0,5
1
2
5
7
CCR
CPHEFT
HEFT
MATH

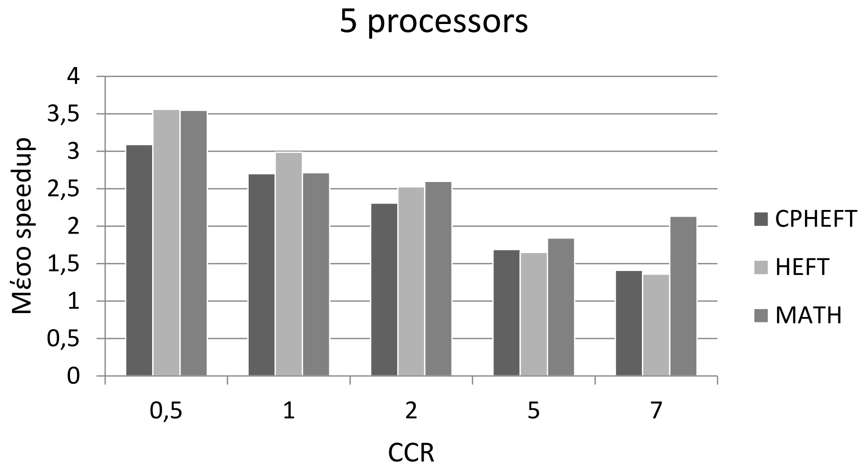

30 tasks
Μέσο speedup
4
3,5
3
2,5
2
1,5
1
0,5
0
0,5
1
2
5
7
CCR
CPHEFT
HEFT
MATH

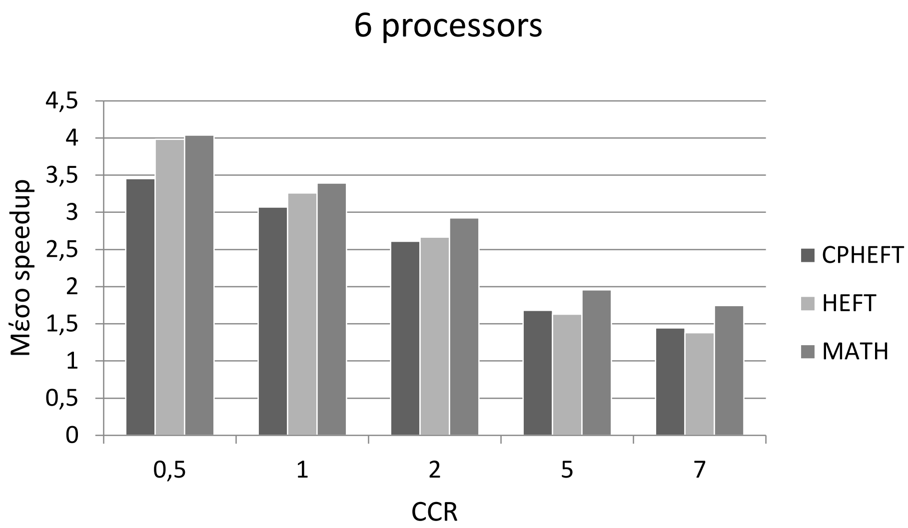


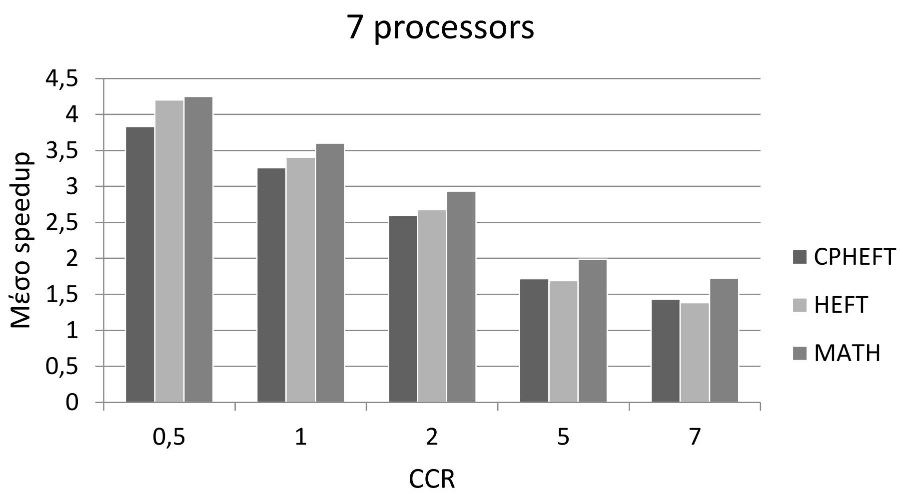
Γραφήματα 7.1. Μέσο speedup για συστήματα 2-8 επεξεργαστών, σε συνάρτηση με το CCR και το πλήθος εργασιών



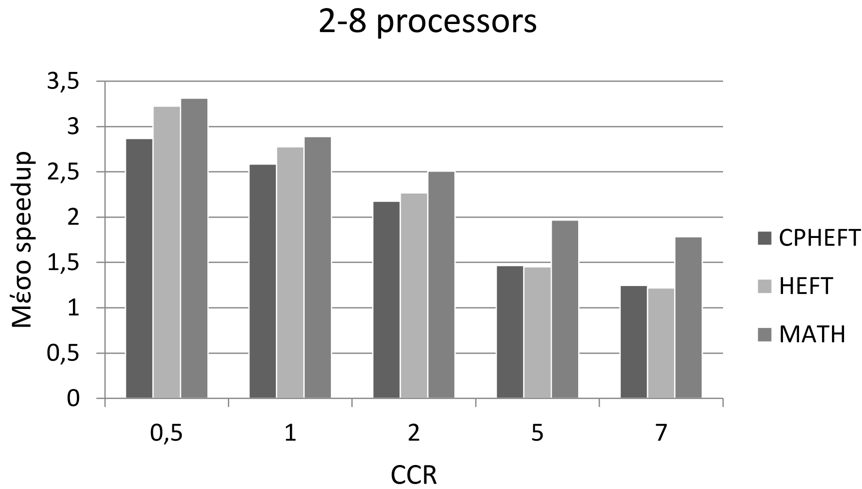
Γράφημα 7.2. Μέσο speedup για τυχαία παραγόμενους γράφους 20-40 εργασιών και συστήματα 2-8 επεξεργαστών, σε συνάρτηση με το CCR



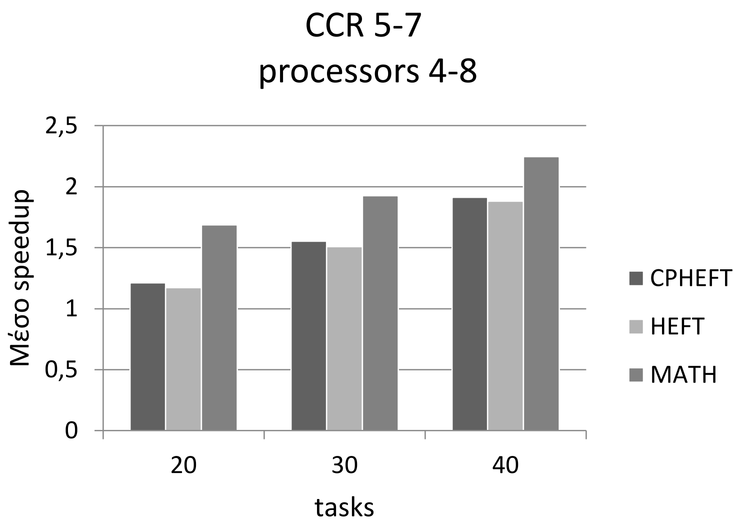




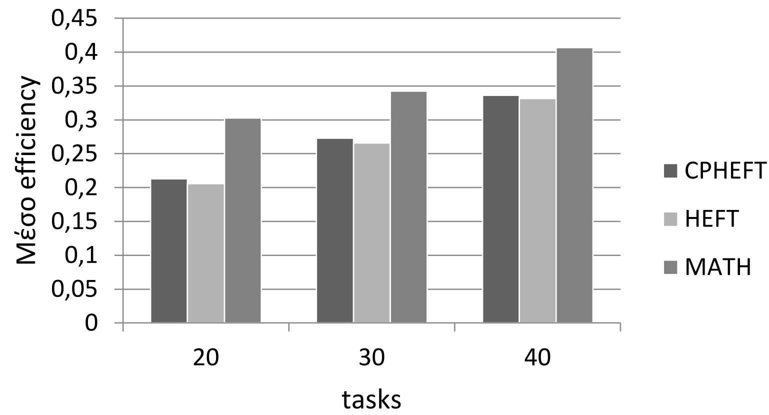
Γραφήματα 7.3. Μέσο speedup για τυχαία παραγόμενους γράφους 10, 20, 30, 40 εργασιών, σε συνάρτηση με το CCR και το πλήθος επεξεργαστών



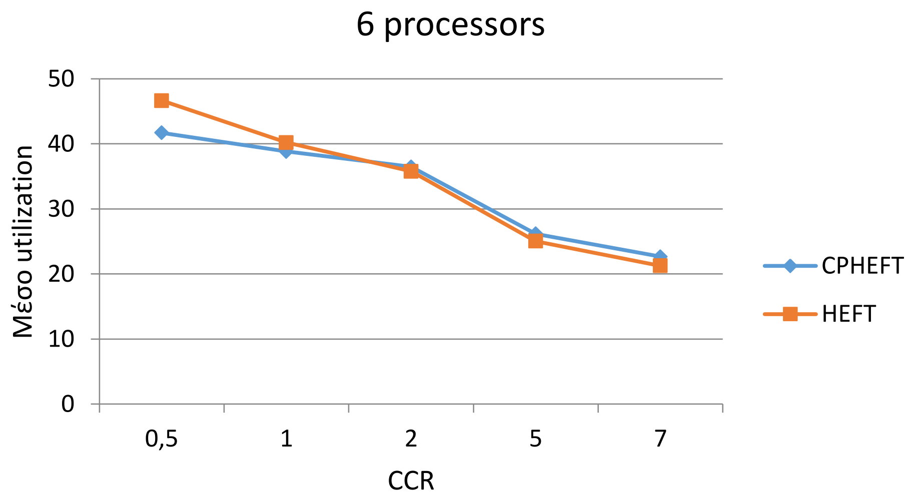
Γράφημα 7.4. Μέσο speedup για συστήματα 2-8 επεξεργαστών και τυχαία παραγόμενους γράφους 10, 20, 30, 40 εργασιών, σε συνάρτηση με το CCR

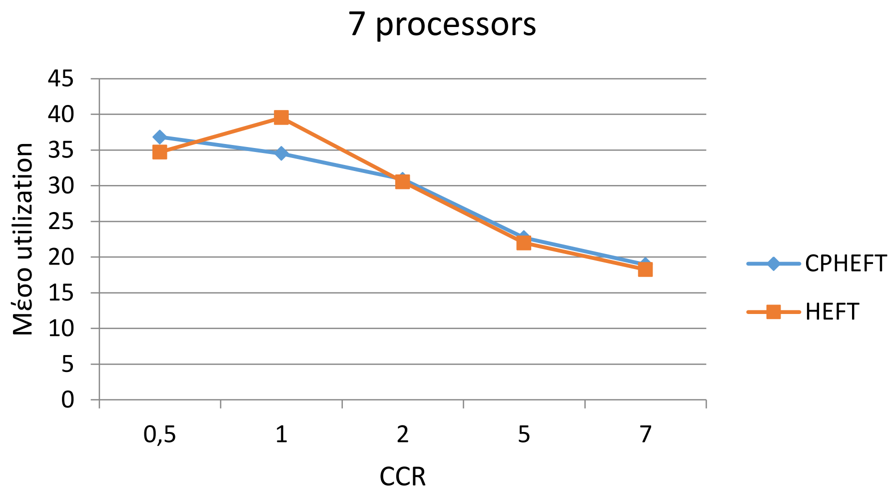


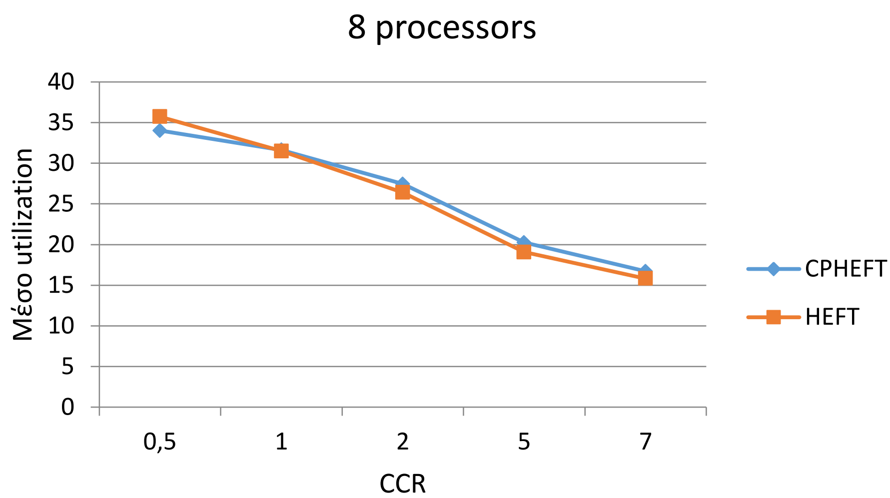
Γράφημα 7.5. Μέσο speedup για συστήματα 4-8 επεξεργαστών και CCR 5-7, σε συνάρτηση με το πλήθος εργασιών των τυχαία παραγόμενων γράφων



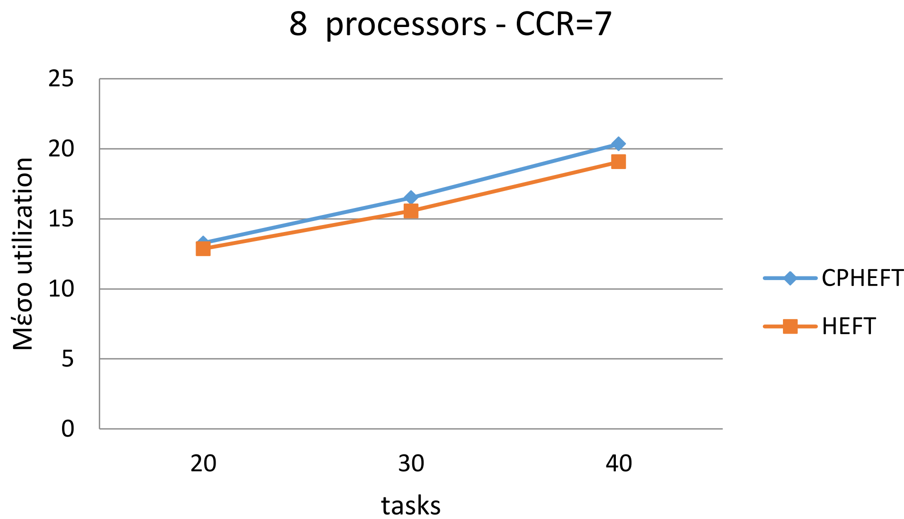
Γράφημα 7.6. Μέσο efficiency για συστήματα 4-8 επεξεργαστών και CCR 5-7, σε συνάρτηση με το πλήθος εργασιών των τυχαία παραγόμενων γράφων

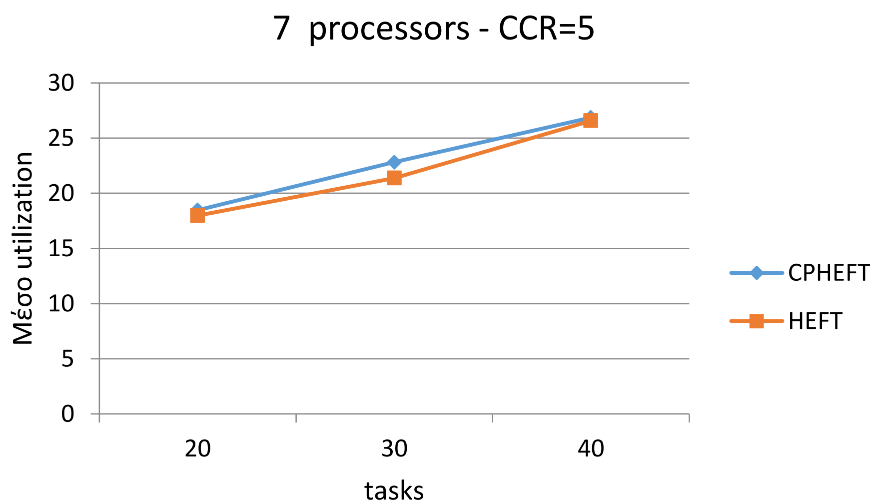






Γραφήματα 7.7. Μέσο utilization των διαθέσιμων επεξεργαστών για τυχαία παραγόμενους γράφους 10, 20, 30, 40 εργασιών σε συνάρτηση με τον λόγο CCR





Γραφήματα 7.8. Μέσο utilization των διαθέσιμων επεξεργαστών για τυχαία παραγόμενους γράφους και CCR σταθερό σε συνάρτηση με το πλήθος εργασιών

7.5 Ερμηνεία αποτελεσμάτων

Αναλύοντας τα αποτελέσματα για κάθε DAG παρατηρούμε, ότι ο αλγόριθμος CPHEFT εμφανίζει οριακά καλύτερα αποτελέσματα όταν υπάρχουν περισσότεροι πόροι και όταν αυξάνεται ο CCR. Συγκεκριμένα, στην επίλυση μικρών προβλημάτων 10 κόμβων μπορούν να αξιοποιηθούν μόνο δύο ή τρεις επεξεργαστές, ενώ μια μεγάλη τιμή CCR ενθαρρύνει την εκτέλεση σε μοναδικό κόμβο. Ωστόσο, λόγω της διάταξης των κόμβων σε συνάρτηση με την κρίσιμη διαδρομή, ο αλγόριθμος CPHEFT διαχειρίζεται καλύτερα

την μεταφορά δεδομένων στις περιπτώσεις που το κόστος επικοινωνίας μεταξύ των κόμβων είναι αυξημένο.

Τα γραφήματα 7.1 περιγράφουν το μέσο speedup για συστήματα 2 - 8 επεξεργαστών, σε συνάρτηση με το CCR και τις διαφορετικές τιμές του πλήθους εργασιών των DAGs που παρήχθησαν. Αντίθετα, τα γραφήματα 7.3 περιγράφουν το μέσο speedup για DAGs με πλήθος εργασιών από 20 έως 40, σε συνάρτηση με το CCR και για συγκεκριμένο πλήθος επεξεργαστών. Τα γραφήματα 7.2 και 7.4. περιγράφουν συγκεντρωτικά τα παραπάνω, ενώ στα γραφήματα 7.5 και 7.6 γίνεται εμφανής η απόδοση του προτεινόμενου αλγορίθμου CPHEFT σε ετερογενή συστήματα με 4 - 8 επεξεργαστές και προβλήματα με CCR$\geq$5.

Τα γραφήματα 7.7 και 7.8 εξετάζουν το μέσο utilization, δηλαδή την αξιοποίηση των πόρων του συστήματος από τους αλγόριθμους CPHEFT και HEFT κατά την διάρκεια εκτέλεσης του προγράμματος (makespan). Είναι εμφανές, ότι το αυξημένο κόστος επικοινωνίας δεν βοηθάει στην παράλληλη εκτέλεση. Ωστόσο, αυτό είναι το χαρακτηριστικό του προβλήματος στο οποίο ο CPHEFT εμφανίζει κάποια υπεροχή – αξιοποιήσιμη ενδεχομένως, με παρέμβαση στην διαδικασία ανάθεσης των κόμβων η οποία εκτελείται με την τεχνική earliest finish time first.

Ο συσχετισμός αυτός περιορίζεται, ασφαλώς, σε σχέση με τον αλγόριθμο HEFT, καθώς για προβλήματα αυτού του μεγέθους ο αλγόριθμος MATH έχει αδιαμφισβήτητη υπεροχή.

8. ΣΥΜΠΕΡΑΣΜΑΤΑ

Σ’ αυτή την εργασία παρουσιάστηκε ο αλγόριθμος CPHEFT, ένας ευρετικός αλγόριθμος για την εκτέλεση εργασιών σε παράλληλα ετερογενή περιβάλλοντα, ο οποίος αποτελεί μια διαφοροποίηση του αλγορίθμου HEFT. Η διαφοροποίηση αυτή έγκειται στην ταξινόμηση των εργασιών με βάση την κρίσιμη διαδρομή. Με την ταξινόμηση αυτή, ορίζεται η προτεραιότητα για την ανάθεση των εργασιών σε καθέναν από τους διαθέσιμους επεξεργαστές διαφορετικής υπολογιστικής ισχύος του συστήματος. Οι εισροές που χρησιμοποιήθηκαν για σύγκριση ήταν DAGs τυχαία παραγόμενα από γεννήτρια, η οποία αναπτύχθηκε γι αυτό το σκοπό. Παρατηρήθηκε, ότι ο αλγόριθμος CPHEFT είναι αποδοτικός για τιμές CCR, οι οποίες περιγράφουν προβλήματα με αυξημένο το κόστος επικοινωνίας μεταξύ των εργασιών μάλλον, παρά το υπολογιστικό τους κόστος. Προβλήματα μεγάλου CCR αποτελούνται από ροές πολλών εργασιών, οι οποίες εκτελούνται σε μεγάλης κλίμακας παράλληλα/κατανεμημένα συστήματα υπολογιστών, συμπεριλαμβανομένων των clouds. Κατά τη διάρκεια εκτέλεσης της ροής εργασιών αυτών των προβλημάτων, εάν τα αρχεία που απαιτούνται για την εκτέλεση μιας εργασίας δεν είναι ήδη διαθέσιμα στον πόρο όπου θα εκτελεστεί η εργασία, ο πόρος παραμένει αδρανής περιμένοντας το τέλος της μετάδοσης αυτών των αρχείων. Με τη μίσθωση πηγών cloud pay-as-you-go (π.χ. εικονικών μηχανών), οι μεταφορές δεδομένων που απαιτούνται μπορεί να είναι εξαιρετικά δαπανηρές, δεδομένου ότι τέτοιες μεταφορές είναι επιρρεπείς σε καθυστερήσεις και μπορεί να επιμηκύνουν την περίοδο ενοικίασης των πόρων (Genez, Sakellariou, et al., Scheduling Scientific Workflows on Clouds Using a Task Duplication Approach 2018). Καθώς οι μεταφορές δεδομένων διαδραματίζουν ουσιαστικό ρόλο στην εκτέλεση ροών εργασιών, ο αλγόριθμος CPHEFT μπορεί να επεκταθεί με τη δημιουργία μηχανισμού duplication για την μείωση του κόστους επικοινωνίας και αυτό μπορεί να αποτελέσει μια μελλοντική εργασία.

ΑΝΑΦΟΡΕΣ

Ahmad, Ishfaq, and Yu-Kwong Kwok. "On exploiting task duplication in parallel program scheduling." *Transactions On Parallel And Distributed Systems.* Ieee Transactions On Parallel And Distributed Systems, 1998. 872-892.

Ajwani, Deepak, Adan Cosgaya-Lozano, and Norbert Zeh. "A Topological Sorting Algorithm for Large Graphs." *Experimental Algorithmics*, 2012: 21.

Ashish Kumar Maurya, and Anil Kumar Tripathi. "ECP: a novel clustering-based technique to schedule precedence constrained tasks onmultiprocessor computing systems." *Computing*, 2018: 1-25.

Baptiste, Philippe, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling : applying constraint programming to scheduling problems.* Kluwer Academic Publishers, 2001.

Bittencourt, Luiz , Rizos Sakellariou, and Edmundo Madeira. "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm." Pisa, Italy, 2010.

Cormen, Thomas, and Devin Balkcom. *Khan Academy.* Dartmouth Computer Science. n.d. https://www.khanacademy.org (accessed 10 4, 2018).

Darte, Alain , Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization.* New York: Birkhauser Boston, 2000.

Deelman, Ewa, et al. "Online workflow management and performance analysis with Stampede." 2011.

Genez, Thiago , Rizos Sakellariou, Luiz Bittencourt, Edmundo Madeira, and Torsten Braun. "Scheduling Scientific Workflows on Clouds Using a Task Duplication Approach." *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC).* 2018.

Genez, Thiago , Rizos Sakellariou, Luiz Bittencourt, Edmundo Madeira, and Torsten Braun. "Scheduling Scientific Workflows on Clouds Using a Task Duplication Approach." 2018.

Gerasoulis, Apostolos , and Tao Yang. *Scheduling Program Task Graphs on MIMD Architectures.* Vol. 231, in *Parallel Algorithm Derivation and Program Transformation. The Springer International Series In Engineering and Computer Science*, 153-186. Boston: Springer, Boston, MA, 1993.

IBM Knowledge Center. *IBM Knowledge Center.* n.d. https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.5.1/ilog.odms.cpo.help/CP_Optimizer/User_manual/topics/propagate_propagate.html (accessed September 30, 2018).

Ilavarasan, E., and P. Thambidurai. "Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments." *Journal of Computer Sciences 3 (2)*, 2007: 94-103.

Juve, Gideon, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. "Characterizing and Profiling Scientific Workflows." 2014.

Juve, Gideon, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. "Characterizing and Profiling Scientific Workflows." 2014.

Kalvin, Alan, and Yaakov Varol. "On the Generation of All Topological Sortings." *JOURNAL OF ALGORITHMS*, 1983: 150-162.

Khan, Minhaj Ahmad. "Scheduling for heterogeneous Systems using constrained critical paths." *Parallel Computing*, April 2012: 175-193 .

Kwok, Yu-Kwong, and Ishfaq Ahmad. "Benchmarking and comparison of the task graph scheduling algorithms." *Journal of Parallel and Distributed Computing 59.3*, 1999: 381–422.

—. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors." *ACM Computing Surveys, Vol. 31, No. 4.* December 1999. 406–471.

Miller, Ade, Ralph Johnson, Kurt Keutzer, Stephen Toub, and Rick Molloy. "http://www.ademiller.com/tech/reports/paraplop_2010_the_task_graph_pattern_workshop_submission.pdf." n.d. http://www.ademiller.com (accessed 10 3, 2018).

Park, Gyung-Leen, Behrooz Shirazi, and Jeff Marquis. "Mapping of Parallel Tasks to Multiprocessors with Duplication." *The Thirty-First Hawaii International Conference on System Sciences.* Kohala Coast, HI, USA: IEEE, 1998.

Park, Gyung-Leen, Behrooz Shirazi, and Jeff Marquis. "Mapping of Parallel Tasks to Multiprocessors with Duplication." 1998.

Planeta, Maksym. "Simulation of a Scheduling Algorithm for DAG-based Task Models." Dresden, 2015.

Robert, Yves. "Task Graph Scheduling." Chap. 733 in *Encyclopedia of Parallel Computing*, by University of Illinois at Urbana-ChampaignUrbanaUSA. Boston: Springer, Boston, MA, 2011.

Rossi, Francesca , Peter Van Beek, and Toby Walsh. "Introduction." In *Handbook of Constraint Programming*. Elsevier Science, 2006.

Samriti, Sandeep Gill, Ankur Bharadwaj, Navpreet Singh, Harsimran Singh, and Jashwinder Singh. "Analysis of HLFET and MCP Task Scheduling Algorithms." *International Journal of Modern Engineering Research (IJMER)* 2, no. 3 (2012): 1176-1180.

Sinnen, Oliver. *TASK SCHEDULING FOR PARALLEL SYSTEMS.* Hoboken, New Jersey: JohnWiley & Sons, Inc, 2007.

Topcuoglu, Haluk, Salim Hariri, and Min-you Wu. *"Performance-effective and low-complexity task scheduling for heterogeneous computing".* Parallel and Distributed Systems, IEEE Transactions, 2002, 260--274.

Valouxis, Christos, Christos Gogos, Panayiotis Alefragis, George Goulas, Nikolaos Voros, and Efthymios Housos. "DAG Scheduling using Integer Programming in heterogeneous parallel execution environments." *Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2013).* Ghent, Belgium, 2013. 392--401.

Wikipedia, the free encyclopedia. *Topological sorting.* August 10, 2018. https://en.wikipedia.org/wiki/Topological_sorting.

Yuyi Jiang, Zhiqing Shao, and Yi Guo. "A DAG Scheduling Scheme on Heterogeneous Computing Systems Using Tuple-Based Chemical Reaction Optimization." Edited by Academic Editor: Yu-Bo Yuan. *The Scientific World Journal* (Hindawi Publishing Corporation) 2014 (2014): 23.

Zhang, Yang, Charles Koelbel, and Ken Kennedy. "Relative Performance of Scheduling Algorithms in Grid Environments." *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007).* Rio de Janeiro, 2007.

ΠΑΡΑΡΤΗΜΑ

```
package generator;

import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;

import utils.Pair;

public class DAGGenerator {

	private double length;
	private int processors;
	private int tasks;
	private int fanout;
	private int maxdist;

	private List<List<Double>> timelines;
	private List<GTask> allTasks;
	private Set<Pair> dependencies;
	private long seed = 1234567890L;
	private Random randomGenerator;

	private List<List<Integer>> computation_costs;
	private List<Integer> communication_costs;

	public DAGGenerator(double l, int p, int t, int f, int m) {
		length = l;
		processors = p;
		tasks = t;
		fanout = f;
		maxdist = m;
		timelines = new ArrayList<>();
		for (int i = 0; i < p; i++) {
			List<Double> aTimeline = new ArrayList<Double>();
			aTimeline.add(0.0);
			aTimeline.add(length);
			timelines.add(aTimeline);
		}
		allTasks = new ArrayList<>();
		dependencies = new HashSet<>();
		randomGenerator = new Random(seed);
	}

	public void setSeed(long seed) {
		this.seed = seed;
		randomGenerator = new Random(seed);
```

```
        }

        public void generate() {
              int number_of_tasks[] = new int[processors];
          int sum = tasks - 2 * processors - 2;

          for (int i = 0; i < processors-1; ++i) {
               number_of_tasks[i] = randomGenerator.nextInt(sum);
          }
          number_of_tasks[processors-1] = sum;

          java.util.Arrays.sort(number_of_tasks);
          for (int i = processors-1; i > 0; --i) {
               number_of_tasks[i] = number_of_tasks[i] - number_of_tasks[i-1];
          }
          for (int i = 0; i < processors; ++i) {
               ++number_of_tasks[i];
          }

              int t_id = 1;
              for (int p = 0; p < processors; p++) {
                     for (int j = 0; j < number_of_tasks[p]; j++) {
                            double time = randomGenerator.nextDouble() * length;
                            timelines.get(p).add(time);
                     }
                     Collections.sort(timelines.get(p));
                     for (int i = 0; i < timelines.get(p).size() - 1; i++) {
                            GTask aTask = new GTask();
                            aTask.id = t_id;
                            t_id++;
                            aTask.cpu = p;
                            aTask.start = timelines.get(p).get(i);
                            aTask.end = timelines.get(p).get(i + 1);
                            allTasks.add(aTask);
                     }
              }

              for (GTask task1 : allTasks) {
                     int task1_i = allTasks.indexOf(task1);
                     int fo = randomGenerator.nextInt(fanout) + 1;
                     for (int i = 0; i < fo; i++) {
                            boolean retry = true;
                            int task2_i = -1;
                            GTask task2 = null;
                            while (retry) {
                                   task2_i =
randomGenerator.nextInt(allTasks.size());
                                   task2 = allTasks.get(task2_i);
                                   if (task2.start >= task1.end || task1.start >=
task2.end) {
                                          double d1 = Math.abs(task2.start -
task1.end);
                                          double d2 = Math.abs(task1.start -
task2.end);
                                          double min = d1;
                                          if (d2 < min)
                                                 min = d2;
                                          if (min > length / maxdist) {
                                                 continue;
```

```
					}
					retry = false;
				}
			}
			if (task1_i < task2_i) {
				Pair pair = new Pair(task1_i + 1, task2_i +
1);
				dependencies.add(pair);
				task1.addSuccessor(task2);
				task2.addPredecessor(task1);
			} else {
				Pair pair = new Pair(task2_i + 1, task1_i +
1);
				dependencies.add(pair);
				task2.addSuccessor(task1);
				task1.addPredecessor(task2);
			}
		}
	}

	// add start and end tasks;
	GTask start = new GTask();
	start.id = 0;
	start.cpu = 0;
	start.start = 0.0;
	start.end = 0.0;
	GTask end = new GTask();
	end.id = allTasks.size() + 1;
	end.cpu = 0;
	end.start = length;
	end.end = length;
	for (GTask task1 : allTasks) {
		if (task1.successors.isEmpty()) {
			task1.addSuccessor(end);
			dependencies.add(new Pair(task1.id, end.id));
		}
		if (task1.predecessors.isEmpty()) {
			task1.addPredecessor(start);
			dependencies.add(new Pair(start.id, task1.id));
		}
	}
	allTasks.add(end);
	allTasks.add(0, start);

	// task computation costs on each processor
	int total_computation_cost = 0;
	computation_costs = new ArrayList<>();
	for (GTask t : allTasks) {
		List<Integer> computation_cost_of_task = new
ArrayList<Integer>();
		double computation_cost = t.end - t.start;
		if (computation_cost < 10) {
			computation_cost =
randomGenerator.nextInt((int)(length * processors / tasks)) + 10;
//			System.out.print(computation_cost + "    ");
		}
		for (int p = 0; p < processors; p++) {
			int temp;
			if (t.cpu == p) {
```

```
                                temp = (int) computation_cost;
                                computation_cost_of_task.add(temp);
                        } else {
                                temp = (int) (randomGenerator.nextDouble() *
computation_cost + 0.25 * computation_cost);
                                computation_cost_of_task.add(temp);
                        }
                        total_computation_cost += temp;
                        double total_mean_computation_cost =
total_computation_cost / processors;
                }
                System.out.print(t + " ");
                for (Integer c : computation_cost_of_task)
                        System.out.printf("%d ", c);
                System.out.println();
                computation_costs.add(computation_cost_of_task);
        }

        double CCR = 5;
        double total_communication_cost = (total_computation_cost /
processors) * CCR;
        int actual_total_communication_cost = 0;
        int number_of_dependencies = dependencies.size();
        double average_communication_cost = total_communication_cost /
number_of_dependencies;
        List<Pair> sorted_dependencies = new ArrayList<>(dependencies);
        Collections.sort(sorted_dependencies);
        communication_costs = new ArrayList<Integer>();
        for (Pair pair : sorted_dependencies) {
                int temp = (int) (average_communication_cost / 2.0
                                + randomGenerator.nextDouble() *
average_communication_cost);
                communication_costs.add(temp);
                actual_total_communication_cost += temp;
        }

        double actualCCR =  (double) actual_total_communication_cost /
(double) (total_computation_cost / processors);
        System.out.println("Actual CCR " + actualCCR);

    }

    public void printDetails() {
        StringBuffer sb = new StringBuffer();
        sb.append("Tasks(" + allTasks.size() + "):\n");
        for (GTask t : allTasks) {
                sb.append(t.toString() + " ");
        }
        sb.append("\nDependencies(" + dependencies.size() + "):\n");
        List<Pair> deps = new ArrayList<>(dependencies);
        Collections.sort(deps);
        for (Pair pair : deps) {
                sb.append(String.format("%d->%d ", pair.x, pair.y));
        }
        System.out.println(sb.toString());
    }

    public void exportToTXT(String name) {
        StringBuffer sb = new StringBuffer();
```

```
		sb.append("#\n");
		sb.append("Processors:" + processors + "\n");
		sb.append("Tasks:" + allTasks.size() + "\n");
		sb.append("# task computation cost on each processor");
		for (GTask t : allTasks) {
			sb.append(String.format("\n%d ", t.id));
			for (int p = 0; p < processors; p++) {
				sb.append(String.format("%d ",
computation_costs.get(t.id).get(p)));
			}
		}
		sb.append("\nDependencies:" + dependencies.size() + "\n");
		sb.append("# from_task_id to_task_id weight");
		List<Pair> deps = new ArrayList<>(dependencies);
		Collections.sort(deps);
		int i = 0;
		for (Pair pair : deps) {
			i++;
			sb.append(String.format("\n%d %d %d", pair.x, pair.y,
communication_costs.get(i - 1)));
		}
		// System.out.println(sb.toString());
		PrintWriter out;
		try {
			out = new PrintWriter("txtfiles\\" + name + ".txt");
			out.println(sb.toString());
			out.close();
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
	}

	public void exportToDOT(String name) {
		StringBuilder sb = new StringBuilder();
		sb.append("digraph ");
		sb.append(name);
		sb.append(" {\n");
		List<Pair> deps = new ArrayList<>(dependencies);
		Collections.sort(deps);
		int i = 0;
		for (Pair p : deps) {
			i++;
			sb.append(String.format("%d -> %d [ label = %d ];\n", p.x,
p.y, communication_costs.get(i - 1)));
		}
		sb.append(" }\n");
		// System.out.println(sb.toString());
		PrintWriter out;
		try {
			out = new PrintWriter("dotfiles\\" + name + ".dot");
			out.println(sb.toString());
			out.close();
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
	}

	private static void small_instance(String name) {
		double length = 100;
```

```
		int processors = 6;
		int tasks = 30;
		int fanout = 3;
		int maxDistanceFactor = 10;
		DAGGenerator generator = new DAGGenerator(length, processors,
tasks, fanout, maxDistanceFactor);
		generator.setSeed(new Random().nextLong());
		generator.generate();
		generator.printDetails();
		generator.exportToDOT(name);
		generator.exportToTXT(name);
	}

	private static void big_instance() {
		double length = 1000000.0;
		int processors = 4;
		int tasks = 100;
		int fanout = 6;
		int maxDistanceFactor = 7;
		DAGGenerator generator = new DAGGenerator(length, processors,
tasks, fanout, maxDistanceFactor);
		generator.generate();
		generator.printDetails();
		generator.exportToDOT("big");
		generator.exportToTXT("datasets\\data_b");
	}

	public static boolean log = false;

	private static void writeFile(String contents, String FileName) {
		try {
			OutputStream stream = new FileOutputStream(FileName);
			BufferedOutputStream output = new
BufferedOutputStream(stream, 4096);
			for (int i = 0; i < contents.length(); i += 2048) {
				if (i + 2048 < contents.length())
					output.write(contents.substring(i, i +
2048).getBytes());
				else

	output.write(contents.substring(i).getBytes());
			}
			if (log)
				System.out.println("Write Finished, Closing Stream");
			output.close();
			if (log)
				System.out.println("Stream closed");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		for(int i = 0; i <  20; i++) {
			small_instance("t30p6r5_"+i);
		}
//		small_instance("t30p6_");
//	     big_instance();
	}
```

```
}
```

```
package solver;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.jgrapht.graph.DefaultWeightedEdge;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import model.Problem;
import model.Task;

public class CPHeft extends BaseSolver {
	final Logger logger = LoggerFactory.getLogger(CPHeft.class);
	// rank is a Map having key the task_id and value the rank
	protected Map<String, Integer> rank, cprank;

	public CPHeft(Problem aProblem) {
		super(aProblem);
		rank = new HashMap<String, Integer>();
		for (Task aTask : aProblem.getTasks()) {
			rank.put(aTask.getId(), -1);
		}
		cprank = new HashMap<String, Integer>();
	}

	public void solve() {
		sortByRankUpwardValuesDesc();
		int T = aProblem.getTasks().size();
		for (int i = 0; i < T; i++) {
			String current_task = findTaskHavingPriority(i);
			int min_eft = Integer.MAX_VALUE;
			int r_min_eft = -1;
			for (int r = 0; r < aProblem.getNumberOfProcessors(); r++) {
				int earliest_finish_time_in_r =
getEarliestFinishTime(
						current_task, r);
				// logger.info(String.format(
				// "Earliest finish time of task %s in resource %d is
%d",
				// current_task, r, earliest_finish_time_in_r));
				if (earliest_finish_time_in_r < min_eft) {
					min_eft = earliest_finish_time_in_r;
					r_min_eft = r;
				}
			}
			solution.scheduleFinishTime(current_task, r_min_eft,
min_eft);
		}
		solution.exportToSOL("cpheftsolfile");
		solution.exportMetrics("cpheftmetrics");
		solution.display();
		//System.out.println( solution.computemakespan());
	}

	protected void sortByRankUpwardValuesDesc() {
```

```
			int T = aProblem.getTasks().size();
			String[] taskIds = new String[T];
			double[] c_ranku = new double[T];
			double[] c_rankd = new double[T];
			double[] c_p = new double[T];
			int k = 0;
			for (Task aTask : aProblem.getTasks()) {
				taskIds[k] = aTask.getId();
				c_ranku[k] = aTask.getRankUpward();
				c_rankd[k] = aTask.getRankDownward();
				c_p[k] = c_ranku[k] + c_rankd[k];
				k++;
			}

			for (int i = 1; i < T; i++) {
				for (int j = T - 1; j >= i; j--) {
					if (c_ranku[j - 1] < c_ranku[j]) {
						String temp = taskIds[j];
						taskIds[j] = taskIds[j - 1];
						taskIds[j - 1] = temp;
						double temp2 = c_ranku[j];
						c_ranku[j] = c_ranku[j - 1];
						c_ranku[j - 1] = temp2;
						double temp3 = c_p[j];
						c_p[j] = c_p[j - 1];
						c_p[j - 1] = temp3;
					}
				}
			}

			List<String> cplist=new ArrayList<String>();
			for (int i = 0; i < T; i++) {
				rank.put(taskIds[i], i);
				aProblem.getTask(taskIds[i]).setRank(i);
				if (Math.abs(c_p[i] - c_p[0])<0.001) {
					cplist.add(taskIds[i]);
				}
			}


			for (String task_id : cplist) {
				update_cprank(task_id);
			}
		}
		//Sort the tasks in scheduling queue by critical task, decreasing order
of upward rank value and increasing order of predecessors.
		protected void update_cprank(String t_id) {
			int length = aProblem.getTask(t_id).getDependedOnTasks().size();
			int[] ptur = new int[length];  //the order of rank upward values
			String[] ptids = new String[length];
			int j=0;
			for (String parentTask_id :
aProblem.getTask(t_id).getDependedOnTasks()) {
				if (cprank.containsKey(parentTask_id)){
					length--;
				}
				else {
					ptur[j] = aProblem.getTask(parentTask_id).getRank();
					ptids[j] = parentTask_id;
```

```
                    j++;
                }
            }

            for (int p = 1; p < length; p++) {
                for (int q = length - 1; q >= p; q--) {
                    if ((ptur[q - 1] > ptur[q]) || ((ptur[q - 1] ==
ptur[q])  && (aProblem.getTask(ptids[q - 1]).getDependedOnTasks().size() >
aProblem.getTask(ptids[q]).getDependedOnTasks().size()))){
                        String temp = ptids[q];
                        ptids[q] = ptids[q - 1];
                        ptids[q - 1] = temp;
                        int temp2 = ptur[q];
                        ptur[q] = ptur[q - 1];
                        ptur[q - 1] = temp2;
                    }
                }
            }

            for (int p = 0; p < length; p++) {
                update_cprank(ptids[p]);
            }

            int t=0;
            for(Integer x: cprank.values()) {
                if (x>=t)
                    t=x+1;
            }
            cprank.put(t_id, t);
        }

        protected String findTaskHavingPriority(int priority) {
            for (String task_id : cprank.keySet()) {
                if (cprank.get(task_id) == priority)
                    return task_id;
            }
            throw new IllegalStateException();
        }

        protected int getEarliestFinishTime(String task_id, int resource_id) {
            Set<DefaultWeightedEdge> edges =
aProblem.getFullGraph().incomingEdgesOf(task_id);
            int max = solution.getAllTasksEarliestTimeInResource(resource_id);
            for (DefaultWeightedEdge dwe : edges) {
                String source_task_id =
aProblem.getFullGraph().getEdgeSource(dwe);
                int sft = solution.getFinishTime(source_task_id);
                int rs = solution.getProcessor(source_task_id);
                if (resource_id != rs)
                    sft = sft + (int)
aProblem.getFullGraph().getEdgeWeight(dwe);
                if (sft > max)
                    max = sft;

            }
            max = max + aProblem.getTask(task_id).getDemandIn(resource_id);
            return max;
        }
}
```