ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Υλοποίηση εικονικού διαχειριστή πόρων για MPI εφαρμογές σε υπερυπολογιστικά συστήματα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξιος Παπαβασιλείου

Επιβλέπων: Γεώργιος Γκούμας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Υλοποίηση εικονικού διαχειριστή πόρων για MPI εφαρμογές σε υπερυπολογιστικά συστήματα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξιος Παπαβασιλείου

Επιβλέπων: Γεώργιος Γκούμας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Απριλίου 2022

(*Υπογραφή*)	(*Υπογραφή*)	(*Υπογραφή*)
....	....	....
Γεώργιος Γκούμας Αν. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2022

....
Αλέξιος Παπαβασιλείου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Αλέξιος Παπαβασιλείου, 2022. Εθνικό Μετσόβιο Πολυτεχνείο. Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς το συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν το συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Οι σύγχρονοι υπερυπολογιστές (high performance computers) εμφανίζονται με τη μορφή συστάδων (clusters) μηχανημάτων. Συνδυάζουν τα επιμέρους resources τους, ώστε να σχηματίσουν μια οντότητα αυξημένων δυνατοτήτων ικανή να επιλύσει σύνθετα προβλήματα. Η αποδοτική τους χρήση στηρίζεται στην προσεκτική δρομολόγηση των εργασιών που υποβάλλονται, ώστε διασφαλιστεί η βελτίωση της επίδοσης τους, χωρίς να καταναλωθούν αλόγιστα ποσά ενέργειας.

Το σύστημα διαχείρισης πόρων είναι υπεύθυνο για τη χρονοδρομολόγηση των εφαρμογών και την κατανομή των αναγκαίων πόρων σε αυτές. Και οι δύο διαδικασίες γίνονται με κριτήρια καθορισμένα, που αφορούν τον αλγόριθμο χρονοδρομολόγησης και την πολιτική διαμοιρασμού πόρων αντίστοιχα. Στις μέρες μας, υπάρχει ποικιλία εργαλείων διαχείρισης πόρων που μπορούν να ενσωματωθούν σε υπάρχοντα clusters. Η επιλογή τους είναι στην ευχέρεια των ιδιοκτητών των συστημάτων αυτών.

Αν και πολύτιμοι, οι διαχειριστές πόρων είναι δύσκολα τροποποιήσιμοι. Οι παράμετροι λειτουργίας τους μπορούν να αλλάξουν μόνο με παρέμβαση των administrators του συστήματος. Το γεγονός αυτό δυσχεραίνει τους χρήστες που θέλουν να εκτελέσουν τις εφαρμογές τους με τρόπο διαφορετικό από τον προκαθορισμένο.

Στόχος της παρούσας εργασίας είναι η παράκαμψη αυτού του προβλήματος μέσα από τη δημιουργία ενός εικονικού διαχειριστή πόρων. Πρόκειται για ένα Python πρόγραμμα που, ρυθμιζόμενο από το χρήστη, αναθέτει MPI εργασίες στα μηχανήματα που δεσμεύει. Μέσα από το εργαλείο αυτό, δοκιμάζουμε διαφορετικά σενάρια δρομολόγησης, χρησιμοποιώντας ουρές εργασιών, που απαρτίζονται από εφαρμογές των Nas Parallel Benchmarks (NPB).

Λέξεις κλειδιά

HPC, Συστάδες Υπολογιστών, Διαχειριστής Πόρων, Χρονοδρομολόγηση, Κατανομή Πόρων, Python, MPI

Abstract

Modern supercomputers (high performance computers) come in the form of machine clusters. They are able to combine their individual resources into forming a larger unit, whose augmented capabilities make it suitable for solving complex problems. Their efficient usage depends upon the careful scheduling of submitted jobs, so that we can achieve an increase in app performance without consuming excessive amounts of energy.

A resource management system is responsible for both scheduling and providing sufficient resources to a job. It does so by applying a predefined job scheduling algorithm and a resource allocation policy. Nowadays, there is a variety of resource management tools that can be integrated into existing clusters. Cluster owners can choose as they see fit.

Though valuable, resource managers are hard to modify. Their settings can only be altered through the intervention of the system administrators. Employing a different configuration can, therefore, be challenging for the users who wish to do so.

We attempt to bypass this issue by creating an adjustable resource manager that, once submitted as a batch script, assigns MPI tasks across a number of bound nodes. This Python tool is then used to apply different scheduling scenarios to job queues consisting of applications from the Nas Parallel Benchmarks (NPB) suite.

Keywords

HPC, Cluster, Resource Manager, Job Scheduling, Resource Allocation, Python, MPI

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή κ. Γεώργιο Γκούμα για την εποπτεία αυτής της εργασίας, αλλά και για τις γνώσεις που μου προσέφερε με τη διδασκαλία των μαθημάτων του.

Οφείλω, επιπλέον, ένα ευχαριστώ στον υποψήφιο διδάκτορα Νικόλαο Τριανταφύλλη για το χρόνο και τις ιδέες που αφιέρωσε. Χωρίς την υπομονή και τη μεθοδική του καθοδήγηση η εκπόνηση της εργασίας αυτής δεν θα ήταν εφικτή.

Ακόμα, ευχαριστώ τους ανθρώπους του CSLab, αλλά και της GRNET, επειδή παρείχαν τα αναγκαία μέσα, ώστε να γίνει το πειραματικό σκέλος αυτής της διπλωματικής.

Επίσης, ευχαριστώ τους φίλους και συμφοιτητές μου για τη συνεργασία στα πλαίσια της σχολής, μα, κυρίως, για την παρέα τους.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου για τη βοήθεια και στήριξη που προσέφεραν, όχι μόνο κατά τα φοιτητικά μου χρόνια, αλλά στη ζωή μου γενικότερα.

Περιεχόμενα

1 Εισαγωγή **16**
1.1 Θέμα . . . 16
1.2 Δομή εργασίας . . . 16

2 HPC **18**
2.1 Εισαγωγή . . . 18
2.2 Ταξινόμηση του Flynn . . . 18
2.3 Πολυεπεξεργαστικά συστήματα . . . 19
2.3.1 Συστήματα κοινής μνήμης . . . 19
2.3.2 Συστήματα κατανεμημένης μνήμης . . . 20
2.3.3 Συστάδες υπολογιστών . . . 21
2.4 Στροφή προς τις συστάδες υπολογιστών . . . 22
2.5 Εφαρμογές . . . 22

3 Resource management **23**
3.1 Εισαγωγή . . . 23
3.2 Εργασίες . . . 23
3.3 Διαχειριστής πόρων . . . 24
3.3.1 Υπολογιστικοί κόμβοι . . . 25
3.3.2 Επεξεργαστές . . . 25
3.3.3 Interconnects . . . 26
3.3.4 Αποθηκευτικά μέσα . . . 26
3.3.5 Επιταχυντές . . . 26
3.4 Δρομολόγηση . . . 27
3.5 Αλγόριθμοι Χρονοδρομολόγησης . . . 27
3.6 Backfilling . . . 28
3.7 Επιλογή resource manager . . . 29

4 Slurm **29**
4.1 Εισαγωγή . . . 29
4.2 Αρχιτεκτονική . . . 29
4.2.1 Slurmd . . . 30
4.2.2 Slurmctld . . . 31
4.3 Αλληλεπίδραση με το χρήστη . . . 32

5 MPI **33**
5.1 Εισαγωγή . . . 33
5.2 Αρχιτεκτονική . . . 34
5.3 Μεταγλώττιση . . . 35
5.4 Mpirun . . . 36

6 NAS Parallel Benchmarks **38**
6.1 Εισαγωγή . . . 38
6.2 Benchmarks . . . 39
6.3 Κλάσεις . . . 40

7 Εικονικός διαχειριστής πόρων **40**
7.1 Εισαγωγή . . . 40
7.2 Αρχιτεκτονική . . . 41
7.2.1 Job Class . . . 41
7.2.2 Socket Class . . . 41
7.2.3 Node Class . . . 41
7.2.4 Manager Class . . . 42
7.3 Χρονοδρομολόγηση . . . 42
7.4 Backfilling . . . 43
7.5 Κατανομή Πόρων . . . 44
7.6 Επικοινωνία με το cluster . . . 46
7.7 Παραμετροποίηση . . . 48
7.8 Εποπτεία εκτέλεσης . . . 49
7.9 Ουρές εργασιών . . . 50
7.10 Παράδειγμα χρήσης . . . 52
7.10.1 Πριν την εκτέλεση . . . 52
7.10.2 Κατά την εκτέλεση . . . 52
7.10.3 Μετά την εκτέλεση . . . 54

8 Πειραματική Αξιολόγηση **56**
8.1 Σενάρια δρομολόγησης . . . 56
8.2 CSLab Server Room . . . 57
8.2.1 Πρώτη ουρά . . . 58
8.2.2 Δεύτερη ουρά . . . 59
8.2.3 Τρίτη ουρά . . . 60
8.2.4 Μέσοι όροι . . . 61
8.3 ARIS . . . 62
8.3.1 Πρώτη ουρά (Κλάση C) . . . 63
8.3.2 Δεύτερη ουρά (Κλάση C) . . . 64
8.3.3 Μέσοι Όροι (Κλάση C) . . . 65
8.3.4 Πρώτη ουρά (Κλάση D) . . . 66
8.3.5 Δεύτερη ουρά (Κλάση D) . . . 67
8.3.6 Μέσοι όροι (Κλάση D) . . . 68
8.4 Παρατηρήσεις . . . 69

9 Μελλοντικές επεκτάσεις **70**

A Παράρτημα **71**
A.1 Nas Parallel Benchmarks (NPB) . . . 71
A.2 CSLab Server Room . . . 72
A.3 ARIS . . . 73

Εικόνες

2.1 Η ταξινόμηση του Flynn 19
2.2 Συστήματα κοινής μνήμης 20
2.3 Συστήματα κατανεμημένης μνήμης 21
2.4 Αρχιτεκτονική συστάδων υπλογιστών 21
3.5 Αναπαράσταση resource manager 24
3.6 Κατανομή πυρήνων σε compact, spare και strip πολιτικές 26
4.7 Οντότητες του Slurm 30
4.8 Οι daemons του Slurm 32
4.9 Αρχιτεκτονική του Slurm 33
5.10 Αναπαράσταση MCA 34
5.11 Wrapper compilers 35
5.12 Compiler flags 35
6.13 Αναλυτικά specifications των NPB 40
7.14 Θεωρούμε το benchmark ep.B.x, που τρέχει στο τρίτο κατά σειρά node. Η εφαρμογή έχει id ίσο με 3 47
7.15 Παρατηρούμε πως έχει δημιουργηθεί το σχετικό dummy αρχείο . . 47
7.16 Πλέον, το αρχείο δεν υφίσταται. Συνεπώς, η εκτέλεση έχει ολοκληρωθεί 47
7.17 Πράγματι, τα slots που βρισκόταν η εφαρμογή είναι άδεια 48
7.18 Πληροφορίες για τις εργασίες του συστήματος 50
7.19 Πληροφορίες για τα nodes 50
7.20 Working directory πριν την εκτέλεση 52
7.21 Working directory κατά την εκτέλεση 53
7.22 Κατάλογος pids 53
7.23 Περιεχόμενα του φακέλου rankfiles 53
7.24 Παράδειγμα rankfile 53
7.25 Πληροφορίες για τις εφαρμογές 54
7.26 Πληροφορίες για τα μηχανήματα 54
7.27 Working directory μετά την εκτέλεση 54
7.28 Αρχείο .err 55
7.29 Αρχείο .out 55
7.30 Δεδομένα εξόδου 56
8.31 Specifications ενός μηχανήματος 57
8.32 Specifications του Xeon E5335 57
8.33 Διαγράμματα για την πρώτη ουρά 58
8.34 Διαγράμματα για τη δεύτερη ουρά 59
8.35 Διαγράμματα για την τρίτη ουρά 60
8.36 Διαγράμματα μέσων όρων 61
8.37 Specifications για το σύστημα ARIS και τα nodes της διαμέρισης compute 62
8.38 Διαγράμματα για την πρώτη ουρά 63
8.39 Διαγράμματα για τη δεύτερη ουρά 64
8.40 Διαγράμματα μέσων όρων 65
8.41 Διαγράμματα για την πρώτη ουρά 66

8.42 Διαγράμματα για τη δεύτερη ουρά 67
8.43 Διαγράμματα μέσων όρων 68

Πίνακες

3.1 Αλγόριθμοι χρονοδρομολόγησης 28
7.2 Heatmap 46
8.3 Μέθοδοι δρομολόγησης 56
8.4 Compact πολιτική για την πρώτη ουρά 58
8.5 Strip πολιτική για την πρώτη ουρά 58
8.6 Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς 58
8.7 Compact πολιτική για τη δεύτερη ουρά 59
8.8 Strip πολιτική για τη δεύτερη ουρά 59
8.9 Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς 59
8.10 Compact πολιτική για την τρίτη ουρά 60
8.11 Strip πολιτική για την τρίτη ουρά 60
8.12 Ποσοστιαία μεταβολή των μετρήσεων της τρίτης ουράς 60
8.13 Μέσοι όροι για compact πολιτική 61
8.14 Μέσοι όροι για strip πολιτική 61
8.15 Ποσοστιαία μεταβολή των μέσων όρων 61
8.16 Compact πολιτική για την πρώτη ουρά 63
8.17 Strip πολιτική για την πρώτη ουρά 63
8.18 Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς 63
8.19 Compact πολιτική για τη δεύτερη ουρά 64
8.20 Strip πολιτική για τη δεύτερη ουρά 64
8.21 Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς 64
8.22 Μέσοι όροι για compact πολιτική 65
8.23 Μέσοι όροι για strip πολιτική 65
8.24 Ποσοστιαία μεταβολή των μέσων όρων 65
8.25 Compact πολιτική για την πρώτη ουρά 66
8.26 Strip πολιτική για την πρώτη ουρά 66
8.27 Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς 66
8.28 Compact πολιτική για τη δεύτερη ουρά 67
8.29 Strip πολιτική για την δεύτερη ουρά 67
8.30 Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς 67
8.31 Μέσοι όροι για compact πολιτική 68
8.32 Μέσοι όροι για strip πολιτική 68
8.33 Ποσοστιαία μεταβολή των μέσων όρων 68

Κώδικες

7.1 Χρονοδρομολόγηση . . . 42
7.2 Χρονικό κατώφλι για backfilling . . . 43
7.3 Backfilling . . . 43
7.4 Υποβολή εργασίας για εκτέλεση . . . 44
7.5 Εύρεση μηχανημάτων . . . 44
7.6 Ανάθεση διεργασιών . . . 45
7.7 Εκτέλεση από φλοιό . . . 45
7.8 Configuration αρχείο . . . 49
7.9 Γεννήτορας εργασιών . . . 51
7.10 Bash script . . . 52

1 Εισαγωγή

1.1 Θέμα

Οι προκλήσεις που γεννήθηκαν μέσα από το πεδίο της έρευνας σε διάφορες επιστήμες (π.χ. βιοϊατρική, μετεωρολογία), σε συνδυασμό με τη ραγδαία ανάπτυξη σε θέματα αρχιτεκτονικής και λογισμικού υπολογιστών αποτέλεσαν καταλυτικό παράγοντα για το σχεδιασμό και την υλοποίηση μαζικά παράλληλων πολυεπεξεργαστικών μηχανημάτων, των οποίων η συνδυαστική υπολογιστική ισχύς αυξάνεται με αμείωτους ρυθμούς.

Εξαιτίας της κλίμακάς τους, τα συστήματα αυτά έχουν υψηλές απαιτήσεις τόσο σε επίπεδο συντήρησης, όσο και σε επίπεδο ενεργειακής κατανάλωσης. Εύλογα, λοιπόν, δίνεται μεγάλη έμφαση στο πως μπορεί να βελτιστοποιηθεί η χρήση των πόρων τους. Την εποπτεία των τελευταίων επωμίζονται ειδικά εργαλεία λογισμικού που καλούνται resource managers.

Ένας διαχειριστής πόρων είναι υπεύθυνος, κατά βάση, για τη δρομολόγηση και την κατανομή πόρων στις εφαρμογές εντός της ουράς εργασιών. Η πρώτη, ακολουθεί έναν συγκεκριμένο αλγόριθμο και καθορίζει τη σειρά εκτέλεσης των εν αναμονή εφαρμογών. Η δεύτερη, που γίνεται βάσει καθορισμένης πολιτικής, συνίσταται στη διανομή των απαραίτητων, και μόνο, resources στις εργασίες, με τρόπο τέτοιο, ώστε να αποφεύγεται ο μεταξύ τους ανταγωνισμός. Συμπληρωματικά, ο resource manager παρέχει μια διεπαφή, ώστε να επιτρέπει στον χρήστη να υποβάλλει δουλειές και να παρακολουθεί την εξελιξή τους.

Οι προηγούμενες λειτουργίες ορίζονται με αυστηρές προδιαγραφές, που δεν μπορούν να τροποποιηθούν, παρά μόνο από τους admins του συστήματος. Ως συνέπεια, οι ερευνητές που θέλουν να πειραματιστούν με τις εν λόγω παραμέτρους δυσκολεύονται. Η εργασία μας επιχειρεί να δώσει λύση σε αυτό το θέμα με τη δημιουργια ενός εικονικού διαχειριστή πόρων (virtual resource manager). Πρόκειται για ένα πρόγραμμα, που υποβάλλεται ως εργασία στο πραγματικό σύστημα και δημιουργεί ένα cluster με τα μηχανήματα που δεσμεύει. Για το χρόνο που τα ελέγχει, μπορεί να τα διαχειριστεί κατά βούληση. Είναι, επομένως, τροποποιήσιμο με βάση τις προτιμήσεις του χρήστη. Αφού γίνει η ανάλυση των αρχών του εργαλείου αυτού, προχωράμε σε πειραματικές δοκιμές χρήσης του με MPI εφαρμογές.

1.2 Δομή εργασίας

Η εργασία περιλαμβάνει εννιά συνολικά ενότητες. Στην *Ενότητα 2* ασχολούμαστε με τους υπερυπολογιστές και τις βασικές μορφές των μηχανημάτων που τους απαρτίζουν.

Η *Ενότητα 3* πραγματεύεται την έννοια και τη λειτουργία του διαχειριστή πόρων.

Η *Ενότητα 4* μελετά την περίπτωση του Slurm ως ένα παράδειγμα σύστηματος διαχείρισης πόρων.

Η *Ενότητα 5* δίνει μια εικόνα για τα βασικά στοιχεία της διεπαφής MPI, η οποία έχει καθιερωθεί σαν μοντέλο ανταλλαγής μηνυμάτων για συστήματα κατανεμημένης μνήμης.

Η *Ενότητα 6* αφορά τα Nas Parallel Benchmarks και τα χαρακτηριστικά τους.

Στην *Ενότητα 7* αναλύουμε τις βασικές λειτουργίες του εργαλείου που αναπτύχθηκε, τις οποίες και παρουσιάζουμε με τη μορφή προγραμματιστικής υλοποίησης. Παρέχουμε, επιπλέον, και ένα παράδειγμα χρήσης του προγράμματός μας.

Η *Ενότητα 8* συγκεντρώνει τα αποτελέσματα και τις παρατηρήσεις των πειραμάτων που κάναμε, με σκοπό να δείξουμε τη λειτουργικότητα του εργαλείου μας.

Στην *Ενότητα 9* κάνουμε μια σύντομη αναφορά σε πιθανές προσεγγίσεις για τη βελτίωση της εργασίας.

2 HPC

2.1 Εισαγωγή

Η πολυπλοκότητα των σύγχρονων επιστημονικών εφαρμογών αποτυπώνεται από την ανάγκη υπολογισμού εκατομμυρίων αριθμητικών πράξεων ή την επεξεργασία μεγάλου όγκου δεδομένων. Οι απαιτήσεις αυτές καθιστούν απαγορευτική την εκτέλεση τέτοιων προβλημάτων από έναν κοινό υπολογιστή γραφείου, είτε εξαιτίας του μεγάλου χρόνου ολοκλήρωσής τους, είτε εξαιτίας των περιορισμένων πόρων του συστήματος (π.χ. κύρια μνήμη). Οι υπερυπολογιστές ξεπερνούν τα εμπόδια αυτά, χρησιμοποιώντας εξειδικευμένο υλικό τελευταίας τεχνολογίας και επιστρατεύοντας μεγάλο αριθμό επιμέρους υπολογιστικών μονάδων που παρέχουν μεγάλη ισχύ στο σύνολο[1]. Στην πράξη, ένας υπερυπολογιστής αποτελείται από εκατοντάδες ή χιλιάδες υπολογιστές, που, μέσω ενός πολύ γρήγορου δικτύου, επικοινωνούν μεταξύ τους, ώστε να λύσουν μεγάλα προβλήματα σε εύλογο χρόνο.

2.2 Ταξινόμηση του Flynn

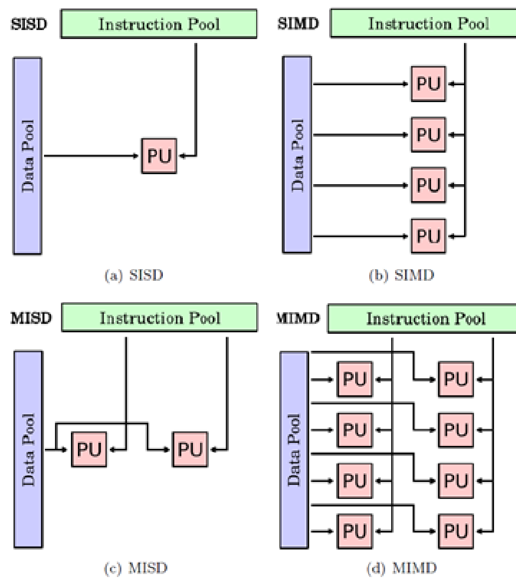
Η πρόταση αυτή αφορά την ταξινόμηση των αρχιτεκτονικών υπολογιστών, και εισήχθη από τον Michael J. Flynn το 1966. Το κριτήριο διαχωρισμού είναι ο αριθμός των διαθέσιμων, ταυτόχρονων ροών εντολών και δεδομένων στην εκάστοτε αρχιτεκτονική. Στο πλαίσιο αυτό, διακρίνονται τέσσερις κατηγορίες[2]:

1. *Single Instruction Single Data stream (SISD)*: Πρόκειται για έναν ακολουθιακό υπολογιστή που εκτελεί μία εντολή τη φορά, όπως αυτή έρχεται από τη μνήμη. Τέτοια αρχιτεκτονική χρησιμοποιούσαν οι παλαιοί προσωπικοί ηλεκτρονικοί υπολογιστές.

2. *Single Instruction Multiple Data stream (SIMD)*: Ένας υπολογιστής που εκτελεί, παράλληλα, την ίδια εντολή σε διαφορετικά δεδομένα, όπως, για παράδειγμα, μία GPU.

3. *Multiple Instructions Single Data stream (MISD)*: Σε αυτήν την περίπτωση έχουμε διαφορετικές εντολές που ενεργούν στο ίδιο δεδομένο. Η αρχιτεκτονική αυτή είναι ασυνήθιστη και βρίσκει εφαρμογή σε περιπτώσεις ελέγχου ανοχής σφαλμάτων.

4. *Multiple Instructions Multiple Data stream (MIMD)*: Πολλαπλοί, αυτόνομοι επεξεργαστές εκτελούν, ταυτόχρονα, διαφορετικές εντολές σε διαφορετικά δεδομένα. Στην κατηγορία αυτή εμπίπτουν τα κατανεμημένα συστήματα.



Εικόνα 2.1: Η ταξινόμηση του Flynn[2]

2.3 Πολυεπεξεργαστικά συστήματα

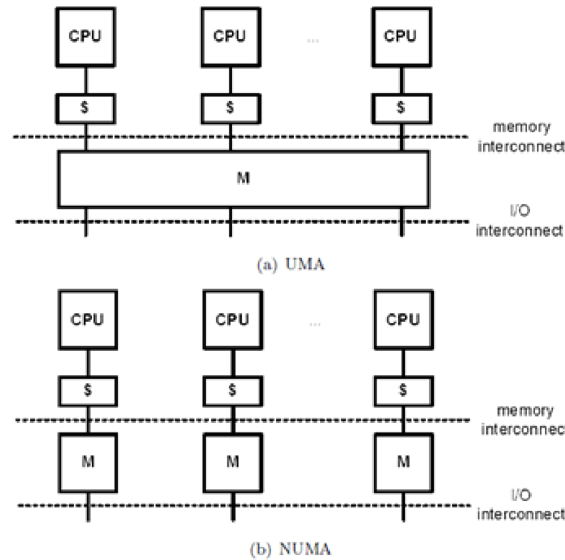
Η ταξινόμηση του Flynn μας είναι χρήσιμη, διότι εισάγει την έννοια των MIMD αρχιτεκτονικών. Μηχανήματα στην κατηγορία αυτή είναι τα πλέον κατάλληλα για να υποστηρίξουν high performance computing, αφού εμφανίζουν αυξημένες δυνατότητες παράλληλης επεξεργασίας. Με την πάροδο των χρόνων έχει εμφανιστεί πληθώρα τέτοιων συστημάτων, που μπορούν να διαχωριστούν ως εξής[3]:

- Συστήματα κοινής μνήμης.
- Συστήματα κατανεμημένης μνήμης.
- Συστάδες (clusters) υπολογιστών.

2.3.1 Συστήματα κοινής μνήμης

Σε μία τέτοια δομή, ο χώρος διευθύνσεων της μνήμης είναι κοινός για όλες τις CPUs και, κατ' επέκταση, για τα προγράμματα που τρέχουν σε αυτές. Συνεπώς, οι επεξεργαστές μοιράζονται, άμεσα, δεδομένα μεταξύ τους με εντολές load/store στην μνήμη. Η τελευταία, μπορεί είτε να ισαπέχει από όλες τις επεξεργαστικές μονάδες, οπότε η αρχιτεκτονική καλείται Uniform Memory Architecture (UMA), είτε όχι, οπότε έχουμε Non Uniform Memory Architecture (NUMA)[2].

Η παραπάνω απόσταση είναι σημαντική, επειδή καθορίζει το χρόνο προσπέλασης στη μνήμη. Έτσι, σε UMA συστήματα, ο χρόνος προσπέλασης είναι ίδιος για κάθε CPU. Το γεγονός αυτό διευκολύνει τον προγραμματιστή, εφόσον αυτός δεν



Εικόνα 2.2: Συστήματα κοινής μνήμης[2]

χρειάζεται να λάβει υπόψη του την τοπικότητα των δεδομένων. Ωστόσο, ένα τέτοιο σύστημα δεν είναι κλιμακώσιμο, αφού η μνήμη μπορεί να εξυπηρετήσει μία CPU τη φορά.

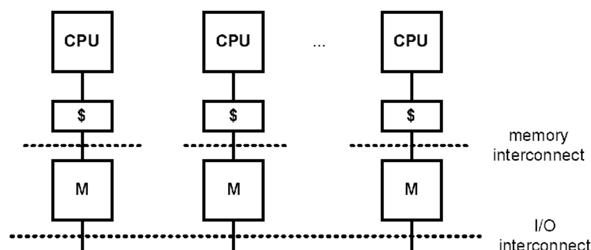
Αντίθετα, ένα NUMA σύστημα κλιμακώνει ευκολότερα, διότι υποστηρίζει περισσότερους πυρήνες ταυτόχρονα (λόγω ετεροχρονισμού). Πλέον, όμως, χρειάζεται προσοχή στην τοπικότητα των δεδομένων, ώστε να εκμεταλλευτούμε μικρότερους χρόνους προσπέλασης που οδηγούν σε καλύτερη επίδοση.

Σε κάθε περίπτωση, η επικοινωνία μεταξύ των στοιχείων του συστήματος επιτυγχάνεται μέσω διαύλου. Με τον τρόπο αυτό, διευκολύνεται τόσο το broadcasting, όσο και η υλοποίηση cache coherence protocols που διασφαλίζουν τη συνέπεια της μνήμης. Παρόλα αυτά, το περιορισμένο bandwidth του διαύλου, δεν επιτρέπει την παρουσία πολλών κόμβων στο δίκτυο και, επομένως, περιορίζει την κλιμακωσιμότητα του συστήματος.

2.3.2 Συστήματα κατανεμημένης μνήμης

Σε αυτά τα συστήματα δεν υπάρχει ενιαία όψη όλης της μνήμης. Δηλαδή, ο κάθε πυρήνας έχει απευθείας πρόσβαση σε ένα κομμάτι φυσικής μνήμης που του αντιστοιχεί. Έμμεσα, ωστόσο, μέσω δικτύου, οι επεξεργαστές ανακτούν πληροφορίες και από τα υπόλοιπα τμήματα μνήμης (βλ. memory interconnect).

Πλέον, εφόσον τα δεδομένα δεν είναι κοινά, η ανταλλαγή τους μπορεί να γίνει μόνο μέσω μηνυμάτων (message passing) από τους σχετικούς επεξεργαστές κατ' εντολή του προγραμματιστή. Για το σκοπό αυτό, χρησιμοποιούνται ειδικά διαμορφωμένες βιβλιοθήκες που καλούνται εντός του προγράμματος. Η message passing

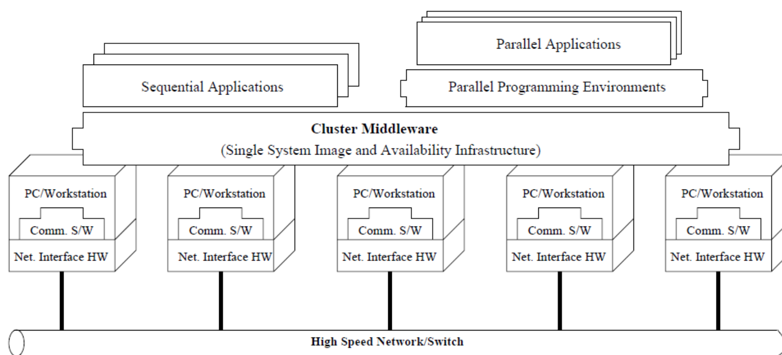


Εικόνα 2.3: Συστήματα κατανεμημένης μνήμης[2]

μεθοδολογία, αν και αρκετά δύσκολη και χρονοβόρα σε υλοποίηση, παρακάμπτει όλα εκείνα τα προβλήματα που εμφανίζονται στην επικοινωνία μέσω διαύλου. Για το λόγο αυτό, εμφανίζει υψηλή επίδοση και κυριαρχεί στους σύγχρονους πολυεπεξεργαστές.

2.3.3 Συστάδες υπολογιστών

Ως συστάδα[3] λογίζεται μία συλλογή workstations ή ατομικών υπολογιστών (κόμβοι) που συνδέονται μεταξύ τους με χρήση κοινού δικτύου. Ένας κόμβος μπορεί να είναι μονοεπεξεργαστικό ή πολυεπεξεργαστικό μηχάνημα, που υποστηρίζει I/O operations και περιλαμβάνει τη δική του μνήμη και το δικό του λειτουργικό σύστημα. Οι κόμβοι συνδέονται μέσα από ένα LAN, οπότε το συνολικό cluster στέκεται σαν ένα μηχάνημα στο επίπεδο δικτύου.



Εικόνα 2.4: Αρχιτεκτονική συστάδων υπλογιστών[3]

Πέρα από τις υπολογιστικές μονάδες, ένα cluster απαρτίζεται από τα ακόλουθα στοιχεία:

- Σύγχρονα λειτουργικά συστήματα (layered ή micro-kernel based).
- Network/switches υψηλών ταχυτήτων (π.χ. Gigabit Ethernet, Infiniband).

- Network Interface Cards (NIC).
- Πρωτόκολλα επικοινωνίας μεγάλων ταχυτήτων (π.χ. Active and Fast messages).
- Middleware:
  - Hardware support (π.χ. DSM).
  - OS support (π.χ. GLUnix).
  - Runtime systems (π.χ. parallel file system).
  - Λογισμικό για resource management και job scheduling.
- Εργαλεία παράλληλου προγραμματισμού (π.χ. compilers, PVMs).
- Applications.

2.4 Στροφή προς τις συστάδες υπολογιστών

Μέχρι και τη δεκαετία του 1990 οι προσπάθειες για βελτίωση της υπολογιστικής επίδοσης εστίαζαν στην ανάπτυξη ταχύτερων και αποδοτικότερων επεξεργαστών. Η τάση αυτή επικράτησε και στον τομέα του supercomputing, μέσα από τη δημιουργία παράλληλων συστημάτων ειδικού σκοπού (π.χ. Cray/SGI T3E). Οι πλατφόρμες αυτές έμειναν στο προσκήνιο, ωσότου το μεγάλο κόστος τους δεν δικαιολογούσε την παρατηρούμενη βελτίωση των εκτελούμενων εφαρμογών. Έκτοτε, χάρη στη δημοφιλία των εξαρτημάτων τους στην αγορά, παρατηρείται μια προτίμηση στα clusters μηχανημάτων. Στο φαινόμενο αυτό συνέβαλε και η σταδιακή τυποποίηση των εργαλείων παράλληλου προγραμματισμού (π.χ. MPI), η οποία διευκόλυνε την ανάπτυξη προγραμμάτων στις εν λόγω συστοιχίες. Η μετάβαση αυτή δικαιολογείται περαιτέρω αν αναλογιστούμε και τα υπόλοιπα προτερήματα των clusters:

- Τα επιμέρους workstations γίνονται ισχυρότερα.
- Χάρη σε νέες τεχνολογίες, το εύρος ζώνης των LAN δικτύων μεγαλώνει διαρκώς.
- Η ενσωμάτωσή τους στα υπάρχοντα δίκτυα είναι ευκολότερη από αυτή των υπερυπολογιστών ειδικού σκοπού.
- Η επέκταση τους γίνεται εύκολα, με την προσθήκη μονάδων hardware (π.χ. μνήμη) ή επεξεργαστών στους κόμβους.

2.5 Εφαρμογές

Τα σημερινά υπερυπολογιστικά συστήματα αξιοποιούνται σε ένα μεγάλο εύρος εφαρμογών, όπως οι παρακάτω:

- *Βιοχημεία*: Μελέτη βιολογικών διεργασιών και πιθανών τρόπων παρέμβασης σε αυτές (π.χ. εφεύρεση νέων φαρμάκων).

- *Χημεία*: Μελέτη ιδιοτήτων ατόμων και ενώσεων (π.χ. για δημιουργία νέων υλικών).
- Φυσική: Προσομοίωση φαινομένων σε διάφορα επίπεδα, από τα υπο-ατομικά σωματίδια ως τους αστέρες και το σύμπαν.
- *Κλιματολογία*: Μελέτη των κλιματολογικών αλλαγών σε μια περιοχή και των παραγόντων που οδηγούν σε αυτές.
- *Μετεωρολογία*: Βελτίωση των μοντέλων πρόβλεψης που χρησιμοποιούνται.
- *Μηχανική*: Μελέτη μηχανικής ρευστών.

3 Resource management

3.1 Εισαγωγή

Η λειτουργία ενός υπερυπολογιστικού συστήματος είναι μια δαπανηρή διαδικασία. Και αυτό, επειδή οι ιδιοκτήτες πρέπει να μεριμνήσουν για τη συντήρηση του υλικού, αλλά και για την κάλυψη της ενέργειας που αυτό χρειάζεται. Για το λόγο αυτό, δίνεται μεγάλη έμφαση στους τρόπους με τους οποίους μπορεί να μεγιστοποιηθεί η χρήση των πόρων και να περιοριστεί το κόστος λειτουργίας τους. Ταυτόχρονα, όμως, δεν πρέπει να θυσιάζεται η επίδοση των εκτελούμενων εφαρμογών. Η ισορροπία μεταξύ κόστους, ρυθμού χρήσης και επίδοσης είναι το ζητούμενο μέσα από την αξιοποίηση διαχειριστών πόρων (resource managers).

3.2 Εργασίες

Με τον όρο εργασία (job) αναφερόμαστε σε ένα σύνολο παραμέτρων που προσδιορίζουν το πρόγραμμα που υποβάλλει ο χρήστης στο σύστημα. Ειδικότερα, μια εργασία μπορεί να περιλαμβάνει[4]:

- Το όνομα του εκτελέσιμου που έχει προέλθει από τη μεταγλώττιση.
- Τα δεδομένα εισόδου/εξόδου του προγράμματος.
- Output directives.
- Μεταβλητές περιβάλλοντος.
- Στοιχεία σχετικά με τον αριθμό των πόρων (π.χ. αριθμός μηχανημάτων, μνήμη) που χρειάζονται για την εκτέλεση του προγράμματος.
- Επιμέρους εργασίες που την απαρτίζουν.

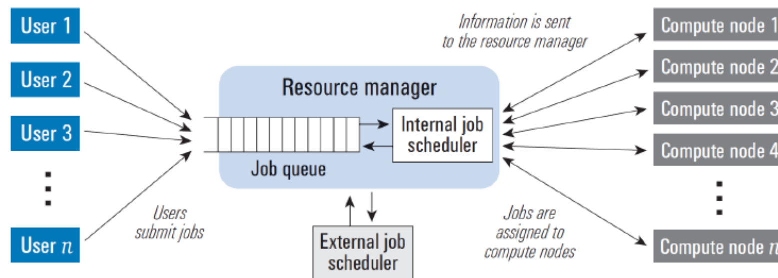
Στα σύγχρονα συστήματα, συνηθίζεται η εκτέλεση των εργασιών να γίνεται στο background (batch mode). Με τον τρόπο αυτό, αποδεσμεύεται ο χρήστης, πράγμα ιδιαίτερα σημαντικό αν αναλογιστεί κανείς πως, εξαιτίας της πολυπλοκότητάς τους, οι εφαρμογές ενδέχεται να τρέχουν για ώρες. Ταυτόχρονα, παρέχεται

ευελιξία στον διαχειριστή πόρων, αφού μπορεί να προσαρμόσει τις ανάγκες της εκάστοτε εργασίας με την κατάσταση του cluster (π.χ. αναμονή μέχρι να ελευθερωθούν κόμβοι). Βέβαια, σε αυτό το mode, οι παράμετροι που αναφέραμε παραπάνω πρέπει να δίνονται πριν την εκτέλεση. Για το λόγο αυτό, ο χρήστης καλείται να γράψει ένα script εντός του οποίου ορίζει αυτές ακριβώς τις προδιαγραφές. Στην πράξη, λοιπόν, μια δουλειά δεν είναι παρά ένα τέτοιο script.

3.3 Διαχειριστής πόρων

Ο διαχειριστής πόρων[5] είναι ένα εργαλείο λογισμικού, αρμόδιο για την εξυπηρέτηση εφαρμογών και την εποπτεία των πόρων του συστήματος. Συγκεκριμένα:

- Δρομολογεί τις εφαρμογές των χρηστών.
- Παρέχει ένα administrative interface για τα μηχανήματα.
- Παρέχει κοινό user interface για τα μηχανήματα.
- Καταγράφει και παρακολουθεί την κατάσταση όλων των μέσων που βρίσκονται υπό την ευθύνη του (π.χ. κόμβων, δίσκων, μνήμης, δικτύου).
- Δέχεται τις εργασίες που υποβάλλονται στο σύστημα.



Εικόνα 3.5: Αναπαράσταση resource manager[6]

Η διαχείριση των πόρων του συστήματος είναι μία σημαντική πρόκληση, που οφείλεται στις αυξημένες απαιτήσεις των παράλληλων εφαρμογών και στις διαφορές που παρουσιάζουν τα μηχανήματα μεταξύ τους. Προκειμένου να γίνεται με τρόπο αποδοτικό, πρέπει να λαμβάνονται υπόψη[7]:

- *Η ετερογένεια των υπολογιστών*: Πρέπει να υποστηρίζονται κόμβοι διαφορετικών ρυθμίσεων σε RAM, CPUs και GPUs.
- *Η κλίμακα των εφαρμογών*: Ανάλογα με το μέγεθος τους, τα προβλήματα εμφανίζουν σημαντικές διαφορές στις ανάγκες τους. Ένας καλός resource manager, θα πρέπει να είναι σε θέση, μέσω της πολιτικής κατανομής πόρων

(resource allocation policy), να θέτει κανόνες σχετικούς με τη χρήση του υλικού, ώστε κάθε εφαρμογή να έχει πρόσβαση μόνο στα μέσα που πραγματικά χρειάζεται.

- *Η κατάσταση των κόμβων*: Υπάρχουν δυναμικές παράμετροι, που επιδρούν στη διαθεσιμότητα του μηχανήματος (π.χ. host/node status).
- *Η τοπολογία του δικτύου*: Η ανάθεση jobs σε υπολογιστικούς κόμβους βασίζεται και στην τοπολογία του επιλεγμένου δικτύου. Ο σκοπός είναι να ελαχιστοποιηθεί η απόσταση εργασιών που πρέπει να επικοινωνούν μεταξύ τους στα πλαίσια εκτέλεσης παράλληλων εφαρμογών.

Στη συνέχεια, γίνεται αναφορά στα τυπικά μέσα που αναγνωρίζει ένας resource manager[8].

3.3.1 Υπολογιστικοί κόμβοι

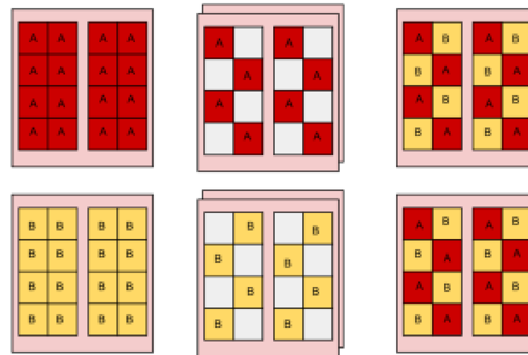
Η αύξηση του αριθμού των κόμβων που ανατίθενται σε μια εφαρμογή οδηγεί σε κλιμάκωση των δεδομένων εισόδου ή σε μείωση του χρόνου εκτέλεσης. Συνεπώς, το πλήθος αυτό είναι σημαντικό σε ό,τι αφορά την εκτέλεση προγραμμάτων. Οι κόμβοι μπορεί να παρουσιάζουν διαφορετικά χαρακτηριστικά (π.χ. μέγεθος μνήμης, interconnects). Ένα αξιόπιστο σύστημα διαχείρισης πόρων επιλέγει τα πιο κατάλληλα μηχανήματα για μια εργασία.

3.3.2 Επεξεργαστές

Στα περισσότερα σύγχρονα μηχανήματα, οι επιμέρους πυρήνες ομαδοποιούνται σε sockets. Έτσι, υποστηρίζεται τοπικός παραλληλισμός για εφαρμογές που τον στηρίζουν μέσω νηματών ή με χρήση πολλαπλών παράλληλων εργασιών ενός νήματος. Οι διαχειριστές πόρων παρέχουν τη δυνατότητα αποκλειστικής ή από κοινού κατανομής cores σε εφαρμογές, μέσα από τις παρακάτω πολιτικές[9]:

- *Compact*: Ένα socket n πυρήνων κατανέμεται σε n εργασίες της ίδιας εφαρμογής. Κατ' αναλογία, το εύρος ζώνης της μνήμης, αλλά και η LLC μοιράζονται σε n τμήματα.
- *Spare/Spread*: Ένα socket n πυρήνων κατανέμεται σε $\frac{n}{2}$ εργασίες της ίδιας εφαρμογής. Τώρα, σε κάθε δουλειά, αντιστοιχούν διπλάσιο bandwidth και cache space από ότι στην compact μέθοδο. Ωστόσο, οι μισοί συνολικά πυρήνες μένουν ανενεργοί, με συνέπεια να μειώνεται σημαντικά το utilization του συστήματος. Τις περισσότερες φορές, η πτώση αυτή δεν αναλογεί με τη βελτίωση της επίδοσης των προγραμμάτων. Ως αποτέλεσμα, η πολιτική αυτή δεν προτιμάται.
- *Strip (co-scheduling)*: Ένα socket n πυρήνων κατανέμεται εξημισίας σε δύο εφαρμογές. Η ιδέα είναι να συνδυάσουμε τα πλεονεκτήματα των προηγούμενων επιλογών: πλήρη απασχόληση των επεξεργαστών με μικρότερο ανταγωνισμό μεταξύ των εργασιών. Στην πράξη, το co-scheduling μπορεί να

λειτουργήσει καλύτερα αν γίνει προσεκτικά το ταίριασμα των εφαρμογών. Για παράδειγμα, ένα memory intensive πρόγραμμα να συνοδεύεται από ένα compute intensive πρόγραμμα.



Εικόνα 3.6: Κατανομή πυρήνων σε compact, spare και strip πολιτικές[9]

3.3.3 Interconnects

Υπάρχουν συστήματα που μπορούν να λειτουργήσουν με διαφορετικό τύπο διασύνδεσης κάθε φορά (π.χ. InfiniBand, GigE). Η επιλογή της κατάλληλης σύνδεσης υπαγορεύεται από τις ανάγκες της εφαρμογής (π.χ. μικρό latency, μεγάλο bandwidth) και έχει σημαντική επίπτωση στην απόδοση της εκτέλεσης.

3.3.4 Αποθηκευτικά μέσα

Πολλά clusters βασίζονται σε συστήματα αρχείων προσβάσιμα σε όλα τα μηχανήματα. Διευκολύνουν, έτσι, τον χρήστη, αφού δεν χρειάζεται μεταφορά των προγραμμάτων από κόμβο σε κόμβο. Επιπλέον, διαχειρίζονται καλύτερα μεγάλα dataset, διότι οποιαδήποτε μεταβολή σε αυτά είναι άμεσα φανερή στα μηχανήματα. Ενδέχεται, όμως, το διαθέσιμο σύστημα αρχείων να μην κλιμακώνει για μεγάλο αριθμό μηχανημάτων ή να μην υποστηρίζει ταυτόχρονες προσβάσεις χρηστών. Για εφαρμογές πολλών I/O operations είναι σκόπιμη η αξιοποίηση των τοπικών δίσκων που έχουν οι κόμβοι. Βέβαια, σε μια τέτοια περίπτωση, τα δεδομένα που παράγονται από τις εφαρμογές πρέπει να μεταφερθούν στο κύριο σύστημα αρχείων αν πρόκειται να υποστούν επεξεργασία.

3.3.5 Επιταχυντές

Η χρήση επιταχυντών (π.χ. GPUs) ή many integrated cores (MICs) αποτελεί μια κοινή και οικονομική, από θέμα ενέργειας, πρακτική για τη βελτίωση της υπολογιστικής επίδοσης. Παρόλα αυτά, δυσχαιρένει τη διαχείριση των μηχανημάτων, καθώς διαφορετικοί κόμβοι μπορεί να έχουν διαφορετικούς επιταχυντές (ορισμένοι

μπορεί να μην έχουν καθόλου). Μοντέρνοι διαχειριστές πόρων επιτρέπουν τον ορισμό, από τους χρήστες, παραμέτρων που δηλώνουν ποιοί κόμβοι είναι προτιμέροι για την εκτέλεση μιας εργασίας.

3.4 Δρομολόγηση

Η δρομολόγηση γίνεται σε δύο διακριτά βήματα[10]. Το πρώτο, αφορά τον καθορισμό της σειράς εξυπηρέτησης των εργασίων που έχουν υποβληθεί (time-specific). Η σειρά αυτή υπαγορεύεται από έναν επιλεγμένο αλγόριθμο. Ασχολούμαστε εκτενέστερα με τους αλγορίθμους αυτούς στην επόμενη ενότητα. Το δεύτερο βήμα, σχετίζεται με την κατανομή εργασιών σε επεξεργαστές του συστήματος (space-specific), όπως αναπτύχθηκε στην ενότητα 3.3.2. Πρόκειται για μια διαδικασία που εξαρτάται τόσο από τα χαρακτηριστικά της εκάστοτε εργασίας (π.χ. requested nodes), όσο και από την τρέχουσα κατάσταση του cluster (π.χ. δουλειές που ήδη τρέχουν). Εφόσον σχεδιαστεί με επιτυχία, η δρομολόγηση εξασφαλίζει:

- *Αποδοτικότητα*: Βελτιώνει την επίδοση των προς δρομολόγηση εφαρμογών, χωρίς να εισάγει σημαντικό overhead.
- *Δικαιοσύνη*: Οι πόροι πρέπει να μοιράζονται με τρόπο δίκαιο, προκειμένου να ικανοποιούνται όλοι οι χρήστες και να αποφεύγονται φαινόμενα starvation.
- *Ευελιξία*: Οι αλγόριθμοι που καθορίζουν που τρέχει ένα job πρέπει να αποκρίνονται δυναμικά στις αλλαγές φορτίου και να εκμεταλλεύονται όλο το φάσμα των διαθέσιμων resources.
- *Διαφάνεια*: Η συμπεριφορά και τα αποτελέσματα της εκτέλεσης μιας εργασίας πρέπει να είναι ανεξάρτητα από τα μηχανήματα στα οποία αυτή τρέχει. Πιο συγκεκριμένα, δεν πρέπει να υπάρχουν διαφοροποιήσεις μεταξύ remote και local execution, ενώ ο κώδικας των εφαρμογών θα πρέπει να υποστηρίζεται απαράλλαχτος από το σύστημα.

3.5 Αλγόριθμοι Χρονοδρομολόγησης

Με την υποβολή της στο cluster, κάθε εργασία εισάγεται σε μια ουρά, όπου και παραμένει μέχρι να έρθει η σειρά της να τρέξει. Η σειρά αυτή προκύπτει από έναν επιλεγμένο, από τον scheduler, αλγόριθμο. Κατά βάση, όλοι οι αλγόριθμοι χρονοδρομολόγησης λειτουργούν με τρόπο όμοιο: ταξινομούν τις δουλειές της ουράς ως προς μια δοθείσα προτεραιότητα και επιλέγουν την κεφαλή (head). Οι αλγόριθμοι, διαφοροποιούνται, ωστόσο, στο κριτήριο με το οποίο αποδίδουν προτεραιότητα σε κάθε δουλειά. Επειδή, πολλές φορές, τα κριτήρια αυτά είναι σύνθετα, προτιμάται η έκφραση τους με μαθηματικές συναρτήσεις (utility functions), μέσω των οποίων μπορούμε, απλά, να συνυπολογίσουμε πολλούς παράγοντες στη διαδικασία της ταξινόμησης. Οι παράγοντες αυτοί έχουν να κάνουν με τις ιδιότητες των εργασιών και είναι, βασικά, οι εξής[11]:

- t_j : Η εκτίμηση για τη διάρκεια μιας εργασίας (walltime).
- q_j: Ο χρόνος αναμονής στην ουρά.
- n_j: Ο αριθμός των απαιτούμενων μηχανημάτων.

Στη συνέχεια παρουσιάζουμε ορισμένους αλγόριθμους χρονοδρομολόγησης και τον τρόπο που εκφράζονται μέσω utility functions:

Αλγόριθμος	Utility Function
FCFS	q_j
FCSJ	q_j/t_j
WFP1	$(q_j/t_j) \cdot n_j$
WFP3	$(q_j/t_j)^3 \cdot n_j$

Πίνακας 3.1: Αλγόριθμοι χρονοδρομολόγησης

Οι ίδιοι αλγόριθμοι μπορούν να αποδοθούν και περιγραφικά:

- *FCFS*: First Come First Served.
- *FCSJ*: First Come Shortest Job. Ευνοούνται παλιές/σύντομες εργασίες, χωρίς να συνεκτιμάται το μέγεθος τους.
- *WFP1*: Ευνοούνται παλιές/σύντομες εργασίες, χωρίς να παραμελούνται αυτές που έχουν μεγαλύτερη κλίμακα.
- *WFP3*: Ευνοούνται παλιές/σύντομες εργασίες (ακόμα περισσότερο), χωρίς να παραμελούνται αυτές που έχουν μεγαλύτερη κλίμακα.

3.6 Backfilling

Οι προηγούμενοι αλγόριθμοι εφαρμόζονται συνοδεία τεχνικών βελτιστοποίησης, που αποσκοπούν στην καλύτερη αξιοποίηση των πόρων του συστήματος και στην αύξηση του σχετικού throughput. Μια τέτοια μέθοδος είναι αυτή του backfilling, με την οποία εργασίες χαμηλής προτεραιότητας επιτρέπεται να παρακάμψουν εργασίες υψηλότερης προτεραιότητας εφόσον δεν καθυστερούν την εκτέλεσή τους. Υπάρχουν δύο βασικές μορφές backfilling[12][13]:

- *Easy Backfilling*: Μεταγενέστερες δουλειές μπορούν να δρομολογηθούν, αν δεν εμποδίζουν την εκτέλεση της κεφαλής της ουράς.
- *Conservative Backfilling*: Μεταγενέστερες δουλειές μπορούν να δρομολογηθούν, αν δεν εμποδίζουν την εκτέλεση όλων των προηγούμενων διεργασιών στην ουρά.

3.7 Επιλογή resource manager

Στην ενότητα αυτή αναφέρουμε μερικά συστήματα διαχείρισης πόρων[5]:

- *Slurm Workload Manager*: Ένα εργαλείο ανοιχτού κώδικα (σε γλώσσα C), συμβατό με LINUX clusters.
- *Torque Resource Manager*: Ένας διαχειριστής που μπορεί να ενσωματώσει έναν εξωτερικό scheduler (π.χ. Maui). Αποτελεί επέκταση του PBS.
- *Moab Cluster Suite*: Ένας ιδιωτικός διαχειριστής που διανέμεται, όπως και ο Torque, από την Adaptive Computing.
- *Oracle Grid Engine*: Συντηρείται από την Oracle (παλαιότερα από την Sun Microsystems) και είναι ένα από τα πρώτα συστήματα public cloud computing.
- *HTCondor*: Ένας open source διαχειριστής που αναπτύχθηκε από το πανεπιστήμιο του Wisconsin. Διαθέτει ένα grid component, χάρη στο οποίο δρομολογεί εργασίες σε idle κόμβους. Η Red Hat έχει βασίσει το MRG Grid product στον HTCondor.

4 Slurm

4.1 Εισαγωγή

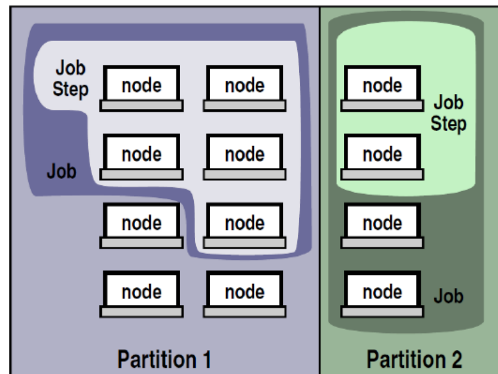
Το Slurm είναι ένα σύστημα διαχείρισης πόρων για LINUX clusters. Έχει σχεδιαστεί με σκοπό να επιτελεί τρεις βασικές λειτουργίες[14]:

1. *Ανάθεση πόρων*: Προσφέρει, για ένα διάστημα, αποκλειστική (ή μη) πρόσβαση σε πόρους του συστήματος, ώστε οι χρήστες που τους αιτούνται να ολοκληρώσουν τη δουλειά τους.
2. *Αλληλεπιδραση με το χρήστη*: Επιτρέπει την υποβολή, εκτέλεση και παρακολούθηση παράλληλων, συνήθως, εργασιών μέσω του πληκτρολογίου.
3. *Διευθέτηση συγκρούσεων*: Χειρίζεται πιθανά αντικρουόμενα αιτήματα για πόρους, διατηρώντας μια ουρά με όλες τις εκκρεμείς εφαρμογές του συστήματος.

4.2 Αρχιτεκτονική

Ο σχεδιασμός του Slurm στηρίζεται στη λογική ερμηνεία των βασικών συστατικών του διαχειριζόμενου cluster μέσα από τις λεγόμενες οντότητες. Η πρώτη από αυτές, το *node*, είναι το βασικό υπολογιστικό μηχάνημα της συστοιχίας. Τα nodes ομαδοποιούνται, ανάλογα με τα χαρακτηριστικά τους, σε *partitions* (διαμερίσεις), απ' όπου και μοιράζονται στα *jobs* που τους αιτούνται. Οι περιορισμοί στα πλαίσια μιας διαμέρισης (π.χ. αριθμός nodes, ομάδες χρηστών) επιβάλλουν την

ιεράρχηση των εργασιών που αναμένουν να εξυπηρετηθούν. Μόλις μια δουλειά αποκτήσει τους αναγκαίους γι'αυτή πόρους, ο χρήστης μπορεί να την εκτελέσει σε ένα ή περισσότερα βήματα (*job steps*), χρησιμοποιώντας, κάθε φορά, ένα υποσύνολο των διαθέσιμων nodes.



Εικόνα 4.7: Οντότητες του Slurm[14]

Η διαχείριση των οντοτήτων αυτών γίνεται από κατάλληλα προγράμματα, τους daemons, που εκτελούνται στο background, χωρίς, δηλαδή, να γίνονται αντιληπτά από τους χρήστες του συστήματος. Στο Slurm ξεχωρίζουμε δύο daemons: τον slurmctld και τον slurmd. Με αυτούς θα ασχοληθούμε στις επόμενες ενότητες.

4.2.1 Slurmd

Ο slurmd είναι ένας πολυνηματικός δαίμονας που τρέχει σε κάθε node του cluster. Ο κύριος ρόλος του είναι η εκτέλεση των εργασιών που φτάνουν στο σύστημα. Παράλληλα, όμως, ο slurmd επικοινωνεί με τον slurmctld: ανακοινώνει ότι είναι ενεργός και παρέχει πληροφορίες σχετικές με την εργασία και τον κόμβο που του αντιστοιχεί. Επειδή εκτελεί δουλειές εκ μέρους πολλών χρηστών, είναι απαραίτητο να τρέχει με δικαιώματα root. Συνολικά, αποτελείται από πέντε συστατικά στοιχεία:

1. *Machine and Job Status Services*: Απαντάνε σε αιτήματα του controller που αφορούν την κατάσταση ενός μηχανήματος και των jobs που εκτελούνται σε αυτό.

2. *Remote Execution*: Εκκινά, παρακολουθεί και αποδεσμεύει διεργασίες, όπως ορίζεται από τον slurmctld ή τις εντολές srun και scancel. Η αρχικοποίηση μίας εργασίας μπορεί να περιλαμβάνει την εκτέλεση ενός prolog προγράμματος, τη θέσπιση ορίων χρήσης, την εξαγωγή μεταβλητών περιβάλλοντος, τη διανομή interconnect resources και τη διαχείριση ομάδων διεργασιών. Αντίστοιχα, ο τερματισμός της δουλειάς, προϋποθέτει την εκτέλεση ενός epilog προγράμματος και την λήξη όλων των επιμέρους διεργασιών ενός group.

3. *Stream Copy Service*: Επιτρέπει το χειρισμό των stderr, stdout και stdin για μεμονωμένα tasks. Τα δεδομένα εισόδου μίας εργασίας μπορεί να προέρχονται από ένα ή περισσότερα αρχεία, από μία srun εντολή ή από το /dev/null. Τα δεδομένα εξόδου αποθηκεύονται σε τοπικά αρχεία ή επιστρέφονται στην εντολή srun. Ανεξάρτητα από την τοποθεσία του stdout/err, η προκύπτουσα έξοδος αποθηκεύεται σε μία ενδιάμεση μνήμη, ώστε να μην μπλοκαριστούν τοπικά tasks.

4. *Job Control*: Επιτρέπει την επικοινωνία με το περιβάλλον του Remote Execution, προωθώντας σήματα ή αιτήσεις τερματισμού σε τοπικά εκτελούμενες εργασίες.

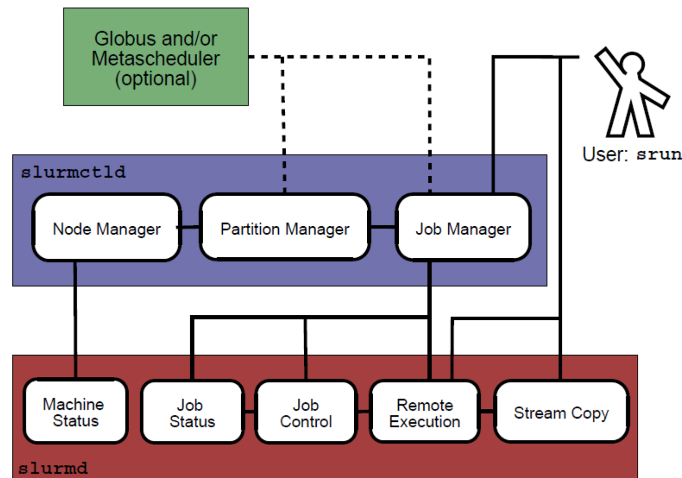
4.2.2 Slurmctld

Οι περισσότερες πληροφορίες σχετικές με την κατάσταση του Slurm βρίσκονται στον slurmctld, τον επονομαζόμενο και controller. Κατά την εκκίνησή του, ο slurmctld διαβάζει τις προδιαγραφές και την πιο πρόσφατη κατάσταση του συστήματος. Περιοδικά, αποθηκεύει την κατάστασή του στο δίσκο, ενώ το ίδιο γίνεται και σε περίπτωση σημαντικών αλλαγών, ώστε να υπάρχει ανεκτικότητα σε σφάλματα. Ο controller λειτουργεί σε master ή stanby mode, ανάλογα με την κατάσταση του εφεδρικού του διδύμου. Σημειώνεται, πως η εκτέλεση του δεν απαιτεί δικαιώματα root. Καλό είναι, όμως, να γίνεται από έναν μόνο χρήστη, ο οποίος σημειώνεται ως SlurmUser. Τον δαίμονα συνθέτουν τρεις επιμέρους δομές:

1. *Node manager*: Παρακολουθεί την κατάσταση όλων των nodes του cluster, είτε ρωτώντας τους σχετικούς slurmd δαίμονες, είτε λαμβάνοντας, ασύγχρονα, ενημερώσεις από μέρους τους. Σε κάθε περίπτωση, διασφαλίζει πως ένας κόμβος έχει τις ρυθμίσεις που τον καθιστούν κατάλληλο για χρήση.

2. *Partition manager*: Εντάσσει τα μηχανήματα σε partitions. Αφού δεχθεί ένα αίτημα για την έναρξη μιας εργασίας από τον job manager, κατανέμει σε αυτή τα απαραίτητα nodes, έχοντας λάβει υπόψη την κατάσταση της εκάστοτε διαμέρισης. Η δουλειά του μπορεί να γίνει και κατ' υπόδειξη των διαχειριστών, με τη βοήθεια της εντολής scontrol.

3. *Job manager*: Δέχεται αιτήματα για εργασίες και τοποθετεί εν αναμονή εφαρμογές σε μια ουρά προτεραιότητας. Παραμένει αδρανής, μέχρι να ανιχνεύσει κάποια αλλαγή του συστήματος, που θα επιτρέψει την εκτέλεση μίας νέας δουλειάς. Τέτοιες μεταβολές αφορούν την υποβολή ή ολοκλήρωση εργασιών και την ενεργοποίηση διαμερίσεων ή κόμβων. Σε τέτοιες περιπτώσεις, ο job manager επιλέγει τις εργασίες με την υψηλότερη προτεραιότητα σε κάθε partition και αιτείται την εκκίνηση τους στον partition manager, παρέχοντας, ταυτόχρονα, απαραίτητα δεδομένα για την εκτέλεση. Με το πέρας της τελευταίας, εκκαθαρίζει τους μέχρι πρότινος δεσμευμένους κόμβους, για να ξεκινήσει ένα νέο κυκλο δρομολόγησης.

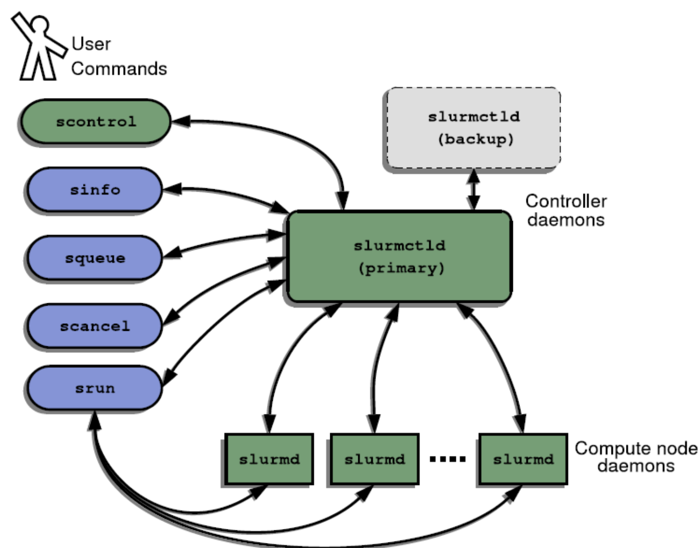


Εικόνα 4.8: Οι daemons του Slurm[14]

4.3 Αλληλεπίδραση με το χρήστη

Το Slurm παρέχει ένα σύνολο command line εντολών που επιτρέπει την εκτέλεση εργασιών και, για διαπιστευμένους χρήστες, τη μεταβολή του configuration του συστήματος. Οι σημαντικότερες από αυτές είναι οι εξής:

- *scancel*: Ακύρωση μίας εργασίας ή ενός βήματος αυτής.
- *scontrol*: Διαχειριστικές για το σύστημα λειτουργίες (π.χ. απενεργοποίηση ενός μηχανήματος για συντήρηση), που παρέχονται σε επιλεγμένους users.
- *sinfo*: Εμφάνιση πληροφοριών (φιλτραρισμένων ή μη) για τα nodes και τα partitions του συστήματος.
- *squeue*: Εμφάνιση πληροφοριών (φιλτραρισμένων ή μη) για running και waiting εργασίες.
- *sbatch*: Υποβολή εργασιών με τη μορφή script, όπως αναφέρεται στην ενότητα 3.2.
- *srun*: Έναρξη των παράλληλων tasks (job steps) μίας εργασίας. Σε περίπτωση interactive εκτέλεσης, η εντολή αυτή μένει ενεργή, με σκοπό να προωθήσει δεδομένα στο stdin.



Εικόνα 4.9: Αρχιτεκτονική του Slurm[14]

5 MPI

5.1 Εισαγωγή

Το MPI (Message Passing Interface) λειτουργεί ως προδιαγραφή για μία διεπαφή βιβλιοθήκης ανταλλαγής μηνυμάτων[15]. Ο ρόλος του είναι ο χειρισμός της μετάδοσης μηνυμάτων σε ένα μοντέλο παράλληλου προγραμματισμού, όπου δύο διεργασίες ανταλλάσσουν δεδομένα μεταξύ των χώρων διευθύνσεών τους μέσω συντονισμένων εντολών. Βέβαια, πέρα από την point-to-point επικοινωνία διεργασιών, το MPI υποστηρίζει επιπλέον λειτουργίες, όπως συλλογική επικοινωνία, RDMA, δυναμική δημιουργία διεργασιών και παράλληλο I/O. Τονίζεται πως το MPI είναι ένα πρότυπο και όχι μια υλοποίηση: υπάρχουν πολλές διαφορετικές υλοποιήσεις του. Επίσης, το MPI δεν είναι κάποια γλώσσα: περιλαμβάνει bindings που εκφράζουν τις λειτουργίες του μέσω συναρτήσεων και μεθόδων. Έτσι, μπορεί να φορτωθεί ως βιβλιοθήκη σε γνωστές γλώσσες προγραμματισμού (C, C++, Fortran 77 και F90). Η ευκολία του αυτή στη χρήση, το κατέστησε βασικό μοντέλο σε διαφορετικά επίπεδα επικοινωνίας. Κάτι τέτοιο φαίνεται στα σύγχρονα κατανεμημένα συστήματα, όπου η high-level επικοινωνία στηρίζεται σε τυποποιημένες low-level ρουτίνες ανταλλαγής μηνυμάτων. Ταυτόχρονα, λόγω του σαφούς ορισμού τους, οι συγκεκριμένες ρουτίνες μπορούν να υλοποιηθούν πολύ αποδοτικά (ακόμα και με hardware υποστήριξη σε ορισμένες περιπτώσεις), προσφέροντας σημαντικά βελτιωμένη κλιμακωσιμότητα στο επίπεδο της εφαρμογής. Η διάδοσή του MPI εξασφάλισε τον βασικό του στόχο, τη δημιουργία ενός αποδοτικού, ασφαλούς και εύχρηστου μοντέλου για την υλοποίηση message-passing προγραμμάτων.

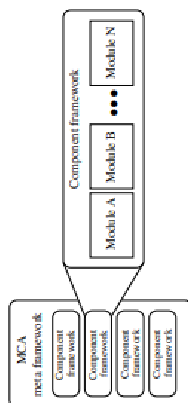
5.2 Αρχιτεκτονική

Ο σχεδιασμός του MPI βασίζεται στην MPI Component Architecture (MCA), σύμφωνα με την οποία κάθε λειτουργικότητα της διεπαφής αντιστοιχεί σε μια back-end υλοποίηση σε επίπεδο λογισμικού. Στο πλαίσιο της αρχιτεκτονικής αυτής, διακρίνονται τρία επιμέρους συστατικά[16]:

1. *Framework*: Δρα ως διαχειριστής των components κατά τον χρόνο εκτέλεσης. Όταν του ζητηθεί, παραμετροποιεί και εγκαθιστά τα κατάλληλα components για ένα task. Επιπλέον, προωθεί σε αυτά run-time παραμέτρους higher-level εντολών (π.χ. mpirun), χρήσιμες για την επιλογή των τελικών modules.

2. *Component*: Μία προγραμματιστική υλοποίηση της διεπαφής ενός μόνο framework. Όταν καλείται, βρίσκει, φορτώνει και χρησιμοποιεί, ανάλογα με τις προδιαγραφές του, ένα ή περισσότερα modules.

3. *Module*: Πρόκειται για αυτοδιαχειριζόμενες μονάδες λογισμικού που συνιστούν στιγμιότυπο ενός component (όμοιο με το στιγμιότυπο μιας κλάσης στη C++). Εν τέλει, είναι τα στοιχεία που εκτελούνται με την κλήση κάποιου framework.



Εικόνα 5.10: Αναπαράσταση MCA[16]

5.3 Μεταγλώττιση

Για την μεταγλώττιση των MPI προγραμμάτων έχουν αναπτυχθεί, για κάθε υποστηριζόμενη γλώσσα, wrapper compilers[17]:

Language	Wrapper compiler name
C	mpicc
C++	mpiCC, mpicxx, or mpic++ (note that mpiCC will not exist on case-insensitive filesystems)
Fortran	mpifort (for v1.7 and above) mpif77 and mpif90 (for older versions)

Εικόνα 5.11: Wrapper compilers[17]

Οι μεταγλωττιστές αυτοί προσθέτουν τα σχετικά compiler/linker flags προκειμένου να καλέσουν τον back-end compiler από τη γραμμή εντολών (π.χ. ο mpicc τελικά καλεί τον gcc). Δημιουργούν, έτσι, command lines της μορφής:

[**compiler**] [x**CPPFLAGS**] [x**FLAGS**] user_arguments [x**LDFLAGS**] [x**LIBS**]

Τα πεδία αυτά αντιστοιχούν στον default back-end μεταγλωττιστή, ως και, κατά σειρά, στα σχετικά preprocesssor, compiler, linker και linker library flags. Κατά βάση, λαμβάνουν τιμές από ορισμένες preset μεταβλητές περιβάλλοντος, τις οποίες ο χρήστης έχει την ευχέρεια να αλλάξει, κάνοντας export τις παραμέτρους που υποδεικνύονται στον παρακάτω πίνακα:

Wrapper Compiler	Compiler	Preprocessor Flags	Compiler Flags	Linker Flags	Linker Library Flags
Generic		OMPI_CPPFLAGS OMPI_CXXPPFLAGS OMPI_F77PPFLAGS OMPI_F90PPFLAGS	OMPI_CFLAGS OMPI_CXXFLAGS OMPI_F77FLAGS OMPI_F90FLAGS	OMPI_LDFLAGS	OMPI_LIBS
mpicc	OMPI_MPICC	OMPI_MPICC_CPPFLAGS	OMPI_MPICC_CFLAGS	OMPI_MPICC_LDFLAGS	OMPI_MPICC_LIBS
mpicxx	OMPI_MPICXX	OMPI_MPICXX_CXXPPFLAGS	OMPI_MPICXX_CXXFLAGS	OMPI_MPICXX_LDFLAGS	OMPI_MPICXX_LIBS
mpif77	OMPI_MPIF77	OMPI_MPIF77_F77PPFLAGS	OMPI_MPIF77_F77FLAGS	OMPI_MPIF77_LDFLAGS	OMPI_MPIF77_LIBS
mpif90	OMPI_MPIF90	OMPI_MPIF90_F90PPFLAGS	OMPI_MPIF90_F90FLAGS	OMPI_MPIF90_LDFLAGS	OMPI_MPIF90_LIBS

Εικόνα 5.12: Compiler flags[17]

Για παράδειγμα, τα linker flags ορίζονται μέσω της OMPI_MPICC_LDFLAGS. Αν η μεταβλητή αυτή δεν χρησιμοποιηθεί, επιλέγεται η generic OMPI_LDFLAGS. Σημειώνεται, πως σε νεότερες εκδόσεις του MPI, οι μεταβλητές αυτές διαβάζονται μέσα από ένα text file που δημιουργείται αυτόματα στον φάκελο $pkgdatadir.

5.4 Mpirun

Η εκτέλεση MPI προγραμμάτων γίνεται με χρήση των εντολών mpirun, orterun και mpiexec. Οι εντολές αυτές έχουν την ίδια συμπεριφορά. Στην εργασία αυτή αναφερόμαστε στην mpirun, ωστόσο η ίδια ανάλυση ισχύει και για τις υπόλοιπες. Σε SIMD μοντέλο, η σύνταξη της εντολής έχει ως εξής[18]:

mpirun [options] <**program**> [<args>]

Αντίστοιχα, σε MIMD η εντολή γίνεται:

mpirun [global_options] [local_options1] <**program1**> [<args1>] :
[local_options2] <**program2**> [<args2>] :
⋮
[local_optionsN] <**programN**> [<argsN>]

Θα επικεντρωθούμε στην περίπτωση του SIMD, αφού ότι γράψουμε εδώ ισχύει και για MIMD. Εξετάζουμε ξεχωριστά κάθε scope της εντολής. Έχουμε:

1. <***program***>: Η εφαρμογή που θα εκτελεστεί. Το MPI την ξεχωρίζει ως το πρώτο άγνωστο argument στη γραμμή εντολών.
2. *[**options**]*: Optional arguments για τη γραμμή εντολών, όπως τα παρακάτω:
   - ***–h,–help***: Εμφάνιση ενός εγχειριδίου για τη σύνταξη της εντολής mpirun.
   - ***–q,–quiet***: Απόκρυψη μηνυμάτων κατά την εκτέλεση της εφαρμογής.
   - ***–v,–verbose***: Εμφάνιση αναλυτικών πληροφοριών κατά την εκτέλεση.
   - ***–V,–version***: Η τρέχουσα έκδοση του MPI.
   - ***-timestamp-output,–timestamp-output***: Timestamp κάθε γραμμής της εξόδου των stdout και stderr.
   - ***-report-bindings,–report-bindings***: Καταγραφή των bindings μεταξύ logical cores και διεργασιών.
3. *[**args**]*: Command line arguments που παίρνουν τιμές από τον χρήστη. Αυτά που, κυρίως, μας απασχολούν είναι:
   - *[**-H,-host**]* <$host_1$,$host_2$,...,$host_n$>: Μια ακολουθία με τα μηχανήματα στα οποία θα τρέξουν οι διεργασίες. Για παράδειγμα, με την παρακάτω εντολή δημιουργούνται δύο εργασίες στο μηχάνημα aa και μία στο bb:

     mpirun -H aa,aa,bb ./a.out
   - *[**-hostfile,–hostfile**]* <$hostfile$>: Ένα αρχείο με τα μηχανήματα που θα τρέξουν εργασίες. Έστω, για παράδειγμα, το παρακάτω:

aa slots=2
bb slots=2
cc slots=2

Εκτελούμε την εντολή:

mpirun -hostfile myhostfile ./a.out

Τωρα, κάθε μηχάνημα τρέχει και από δύο εργασίες.

- ***[-c,-n,-np,-N]*** <#>: Ο αριθμός των αντιγράφων του προγράμματος που θα εκτελεστούν στα επιλεγμένα nodes. Αν η παράμετρος αυτή μείνει ακαθόριστη, δημιουργείται ένα αντίγραφο σε κάθε διαθέσιμο slot του μηχανήματος (βλ. παράδειγμα hostfile). Η τιμή της παραμέτρου αυτής καθορίζει και τα αναγνωριστικά (ranks) που αποδίδονται σε κάθε διεργασία. Έτσι, αν επιλέξουμε n αντίγραφα, καθένα από αυτά αποκτά ένα μοναδικό rank μεταξύ του 0 και του $n-1$.

- ***–map-by*** <foo>: Η αντιστοίχιση μιας διεργασίας σε ένα κομμάτι hardware. Ιδιαίτερο ενδιαφέρον έχει ο τρόπος που ένα task βρίσκει θέση σε ένα μηχάνημα. Διακρίνουμε τρεις περιπτώσεις:

  i. ***–map-by*** *node*: Οι διεργασίες μοιράζονται με round robin τρόπο στα διαθέσιμα nodes.

  ii. ***–map-by*** *socket*: Οι διεργασίες μοιράζονται με round robin τρόπο στα sockets ενός node. Μόλις αυτά καλυφθούν, ο αλγόριθμος προχωρά στο επόμενο διαθέσιμο μηχάνημα.

  iii. ***–map-by*** *core*: Για ένα node, οι διεργασίες μοιράζονται με round robin τρόπο στα logic cores ενός socket του. Μόλις το socket γεμίσει, ο αλγόριθμος συνεχίζει με το επόμενο. Αν δεν υπάρχει κάποιο slot ελεύθερο, επιλέγεται ένα διαφορετικό μηχάνημα.

- ***[-mca,–mca]*** <*key*> <*value*>: Πέρασμα παραμέτρων σε MCA modules, με μορφή key-value pairs. Η τιμή του κλειδιού αντιστοιχεί σε ένα MCA module, το οποίο αποκτά τιμή value. Σημειώνεται πως σε ένα key μπορούν να ανατεθούν πολλά values. Έστω η εντολή:

  mpirun -mca btl tcp,self -np 1 foo

  Με βάση αυτή, το MPI θα χρησιμοποιήσει τα tcp και self BTL's, ώστε να τρέξει ένα αντίγραφο του foo στον δεσμευμένο κόμβο. Η παράμετρος mca χρησιμοποιείται πολλές φορές για τον προσδιορισμό διαφορετικών key-value pairs. Πρακτικά, το argument αυτό υποκαθιστά τη διαδικασία εξαγωγής των αντίστοιχων environment variables. Έτσι, ισοδύναμα, μπορούμε να γράψουμε: OMPI_MCA_<key>=<value>.

- ***[-rf,–rankfile]*** <$rankfile$>: Ένα αρχείο με το οποίο ο χρήστης καθορίζει, με explicit τρόπο, το mapping των διεργασιών στους πυρήνες κάθε node, δημιουργώντας custom πολιτικές διαμοιρασμού πόρων. Το περιεχόμενό του μοιάζει ως εξής:

rank 0=aa slot=1:0-2
rank 1=bb slot=0:0,1
rank 2=cc slot=1-2

Με βάση αυτό, η διεργασία με rank 0 τρέχει στους πυρήνες 0,1 και 2 του socket 1 στο node aa. Αντίστοιχα, η διεργασία με rank 1 απασχολεί τους πυρήνες 0 και 1 του socket 0 του μηχανήματος bb. Τέλος, το τρίτο task (rank 2) τρέχει στα cores 1 και 2 του node cc. Εδώ, τονίζεται πως δεν χρειάζεται να αναφέρουμε το κάθε μηχάνημα με το όνομά του. Αντ' αυτού, χρησιμοποιούμε ένα offset το οποίο ταυτοποιεί ένα node με βάση τη σειρά που δεσμεύτηκε (μέσω των επιλογών -H,-hostfile ή ενός job scheduler). Δηλαδή, μια γραμμή σαν την ακόλουθη, περιγράφει ένα task που δεσμεύει cores στο πρώτο κατά σειρά επιλεγμένο μηχανημα:

rank 0=+n0 slot=1:0-2

Μέχρι στιγμής, στα προηγούμενα αρχεία, χρησιμοποιείται η λογική αρίθμηση των cores (σε συνάρτηση με το slot που ανήκουν). Υπάρχει, όμως, η επιλογή, μέσω της παραμέτρου rmaps_rank_file_physical, να χρησιμοποιηθεί η φυσική αρίθμηση των πυρήνων. Σε μία τέτοια περίπτωση έχουμε ένα αρχείο σαν το παρακάτω:

rank 0=aa slot=1
rank 1=bb slot=8
rank 2=cc slot=6

Εδώ, η διεργασία με rank 0 τρέχει στην physical unit 1 του μηχανήματος aa. Όμοια, το δεύτερο job δεσμεύεται στην physical unit 8 του node bb, ενώ το τρίτο στον πυρήνα με αριθμό 6 του cc.

6 NAS Parallel Benchmarks

6.1 Εισαγωγή

Τα NAS Parallel Benchmarks (NPB)[19] συνιστούν μία σουίτα μετροπρογραμμάτων που διανέμονται από τον κλάδο Advanced Supercomputing της NASA (NAS). Η ανάγκη για τη δημιουργία τους, το 1991, προήλθε, αφενός, από τη δυσκολία μεταφοράς full scale εφαρμογών στα τότε παράλληλα συστήματα και, αφετέρου, από την ασυμβατότητα των ήδη υπαρχόντων benchmarks με τα ολοένα και αναπτυσσόμενα μηχανήματα της εποχής. Προκειμένου η εφαρμογή τους να έχει νόημα, τα προγράμματα έχουν τις εξής ιδιότητες:

- Είναι προσαρμόσιμα στις μεταβολές σε επίπεδο αλγορίθμων και λογισμικού.
- Δεν βασίζονται σε συγκεκριμένη μορφή αρχιτεκτονικής.
- Η ορθότητα των αποτελεσμάτων μπορεί να εξακριβωθεί εύκολα. Κοινώς, τα δεδομένα εισόδου/εξόδου είναι μικρά σε μέγεθος. Ταυτόχρονα, η διαδικασία για τον υπολογισμό της εξόδου πρέπει να είναι σαφώς ορισμένη.

- Είναι προσαρμόσιμα σε απαιτήσεις μνήμης και χρόνου εκτέλεσης.
- Μπορούν να διανεμηθούν εύκολα.

Τα benchmarks που επιλέχθηκαν προέρχονται από εφαρμογές υπολογιστικής δυναμικής ρευστού. Η αρχική τους έκδοση περιλαμβάνει πέντε βασικούς πυρήνες και τρεις ψευδο-εφαρμογές. Στην πορεία, έχουν προστεθεί επιπλέον περιπτώσεις για adaptive meshes, παράλληλα I/O, εφαρμογές πολλαπλών ζωνών και υπολογιστικά πλέγματα.

6.2 Benchmarks

Τα NBP περιλαμβάνουν πέντε βασικά προβλήματα:

1. *EP*: Ένας embarrassingly parallel πυρήνας που υπολογίζει ένα ολοκλήρωμα με ψευδο-τυχαίες δοκιμές. Παρέχει μία εκτίμηση για το άνω όριο σε επίδοση floating πράξεων, σε περιπτώσεις, δηλαδή, με ελάχιστη επικοινωνία μεταξύ των επεξεργαστών.

2. *MG*: Ένας απλοποιημένος multigrid πυρήνας. Απαιτεί ένα καλά δομημένο πρωτόκολλο long distance communication, ενώ ελέγχει την επικοινωνία δεδομένων μικρής και μεγάλης απόστασης.

3. *CG*: Ένας πυρήνας που κάνει χρήση της μεθόδου συζυγών κλίσεων για να υπολογίσει μια προσέγγιση της μικρότερης ιδιοτιμής ενός αραιού, συμμετρικού, θετικού πίνακα. Προσομοιώνει, έτσι, υπολογισμούς σε ένα αδόμητο πλέγμα, καθώς δοκιμάζει άτακτη επικοινωνία μεγάλων αποστάσεων, μέσω πολλαπλασιασμού πινάκων-διανυσμάτων.

4. *FT*: Μία τρισδιάστατη λύση μερικής διαφορικής εξίσωσης με εφαρμογή γρήγορου μετασχηματισμού Fourier. Η εφαρμογή αυτή προσεγγίζει την φασματική μέθοδο επίλυσης διαφορικών εξισώσεων και είναι ιδιαίτερα απαιτητική σε επίπεδο long-distance επικοινωνίας.

5. *IS*: Πρόκειται για μία ταξινόμηση ακεραίων που βρίσκει εφαρμογή σε particle codes. Ελέγχει τόσο την ταχύτητα υπολογισμού ακεραίων, όσο και την επίδοση στο κομμάτι της επικοινωνίας.

Υπάρχουν, όμως, και τρεις επιπλέον εφαρμογές: οι LU,SP και BT. Αυτές πραγματεύονται το ίδιο πρόβλημα, την επίλυση ενός συστήματος μη γραμμικών μερικών διαφορικών εξισώσεων. Ωστόσο, η καθεμία το κάνει με διαφορετικό τρόπο:

1. *LU*: Γίνεται χρήση ενός Lower-Upper Gauss-Seidel solver με περιορισμένο παραλληλισμό.

2. *SP*: Αξιοποιείται ένας Scalar Penta-Diagonal solver.

3. *BT*: Εφαρμόζεται ένας Block Tri-Diagonal solver.

6.3 Κλάσεις

Οι ιδιότητες που αναφέραμε προηγουμένως, σε συνδυασμό με την ανάγκη για την αξιολόγηση συστημάτων διαφορετικών δυνατοτήτων, επιβάλλουν την ύπαρξη πολλαπλών μεγεθών στα NPB. Καθένα από τα μεγέθη αυτά ορίζει μία κλάση. Συνολικά, υπάρχουν οκτώ κλάσεις: οι S,W,A,B,C,D,E,F όπως προκύπτουν με σειρά μεγέθους. Η S χρησιμεύει για γρήγορα test μικρής έκτασης, ενώ η W αντιστοιχεί σε μέγεθος ενός 90's workstation. Οι A,B,C λογίζονται ως μια ομάδα μικρών κλάσεων, η καθεμιά τέσσερις φορές μεγαλύτερη από την προηγούμενη (σε επίπεδο προβλήματος). Τέλος, οι D,E,F, είναι οι μεγαλύτερες τάξεις. Σε αυτές, μετάβαση από τη μία στην άλλη ισοδυναμεί με πολλαπλασιασμό του μεγέθους του προβλήματος με έναν παράγοντα x16. Αναλυτικότερα, οι προδιαγραφές των benchmarks σε κάθε κλάση φαίνονται στον πίνακα που ακολουθεί[20]:

Benchmark	Parameter	Class S	Class W	Class A	Class B	Class C	Class D	Class E	Class F
CG	no. of rows	1400	7000	14000	75000	150000	1500000	9000000	54000000
	no. of nonzeros	7	8	11	13	15	21	26	31
	no. of iterations	15	15	15	75	75	100	100	100
	eigenvalue shift	10	12	20	60	110	500	1500	5000
EP	no. of random-number pairs	2^{24}	2^{25}	2^{28}	2^{30}	2^{32}	2^{36}	2^{40}	2^{44}
FT	grid size	64 x 64 x 64	128 x 128 x 32	256 x 256 x 128	512 x 256 x 256	512 x 512 x 512	2048 x 1024 x 1024	4096 x 2048 x 2048	8192 x 4096 x 4096
	no. of iterations	6	6	6	20	20	25	25	25
IS	no. of keys	2^{16}	2^{20}	2^{23}	2^{25}	2^{27}	2^{31}	2^{35}	
	key max. value	2^{11}	2^{16}	2^{19}	2^{21}	2^{23}	2^{27}	2^{31}	
MG	grid size	32 x 32 x 32	128 x 128 x 128	256 x 256 x 256	256 x 256 x 256	512 x 512 x 512	1024 x 1024 x 1024	2048 x 2048 x 2048	4096 x 4096 x 4096
	no. of iterations	4	4	4	20	20	50	50	50
BT	grid size	12 x 12 x 12	24 x 24 x 24	64 x 64 x 64	102 x 102 x 102	162 x 162 x 162	408 x 408 x 408	1020 x 1020 x 1020	2560 x 2560 x 2560
	no. of iterations	60	200	200	200	200	250	250	250
	time step	0.01	0.0008	0.0008	0.0003	0.0001	0.00002	0.000004	0.0000006
(BT-IO)	write interval	5	5	5	5	5	5	5	
	Gbytes written	0.0008	0.022	0.42	1.7	6.8	135.8	2122.4	
LU	grid size	12 x 12 x 12	33 x 33 x 33	64 x 64 x 64	102 x 102 x 102	162 x 162 x 162	408 x 408 x 408	1020 x 1020 x 1020	2560 x 2560 x 2560
	no. of iterations	50	300	250	250	250	300	300	300
	time step	0.5	0.0015	2.0	2.0	2.0	1.0	0.5	0.2
SP	grid size	12 x 12 x 12	36 x 36 x 36	64 x 64 x 64	102 x 102 x 102	162 x 162 x 162	408 x 408 x 408	1020 x 1020 x 1020	2560 x 2560 x 2560
	no. of iterations	100	400	400	400	400	500	500	500
	time step	0.015	0.0015	0.0015	0.001	0.00067	0.0003	0.0001	0.000015

Εικόνα 6.13: Αναλυτικά specifications των NPB[20]

7 Εικονικός διαχειριστής πόρων

7.1 Εισαγωγή

Το πρόγραμμα που κατασκευάστηκε χρησιμεύει για το στήσιμο ενός εικονικού resource manager επί ενός πραγματικού συστήματος. Αυτό σημαίνει πως υποβάλλεται σαν ένα bash script που δεσμεύει, για καθορισμένο χρόνο, ορισμένα μηχανήματα του cluster. Άπαξ και δεσμευτούν, τα συγκεκριμένα nodes λειτουργούν ανεξάρτητα από τα υπόλοιπα, αφού δεν επηρεάζονται από εξωτερικές παρεμβάσεις. Σχηματίζουν, δηλαδή, ένα υπο-cluster, που, με την προσθήκη ενός λογισμικού διαχείρισης πόρων, έχει παρόμοια συμπεριφορά με το αρχικό. Το ρόλο αυτού του λογισμικού καλύπτει το πρόγραμμά μας, που βρίσκεται στην σελίδα που ακολουθεί: `https://github.com/alexispapabil/thesis`.

7.2 Αρχιτεκτονική

Προκειμένου να λειτουργεί σωστά, ο διαχειριστής πρέπει να γνωρίζει τι συμβαίνει στα μηχανήματα που βρίσκονται υπό τον έλεγχό του. Ένας τρόπος να το πετύχει αυτό, είναι να διατηρεί μια λογική αναπαράσταση τους. Έτσι, μπορεί να κρίνει αν υπάρχουν οι διαθέσιμοι πόροι για τη δρομολόγηση μιας εργασίας, ανάλογα με το ποιοι πυρήνες είναι ελεύθεροι και ποιοι κατηλειμμένοι. Η αντιστοίχιση πραγματικών και εικονικών μηχανημάτων, γίνεται μέσω python classes. Ειδικότερα, κάθε μηχάνημα (node) αποδίδεται σαν ένα στιγμιότυπο μιας κατάλληλα διαμορφωμένης κλάσης. Η τελευταία έχει σχεδιαστεί έτσι, ώστε να ενθυλακώνει όλα τα επιμέρους δομικά στοιχεία ενός node. Στα πλαίσια της εργασίας, έχουν αναπτυχθεί τέσσερις κλάσεις:

- Job Class.
- Socket Class.
- Node Class.
- Manager Class.

7.2.1 Job Class

Ένα στιγμιότυπο αυτής της κλάσης αντιστοιχεί σε μία εργασία της ουράς του συστήματος. Ενσωματώνει όλες τις πληροφορίες που αφορούν την εκτέλεσή της (π.χ. όνομα εφαρμογής, αριθμός parallel tasks, walltime), τις οποίες και γνωστοποιεί στα μηχανήματα που θα την αναλάβουν. Κάθε εργασία μπορεί να βρίσκεται σε μία από δύο καταστάσεις: queuing, οπότε και παραμένει στην ουρά ή running, όταν εκτελείται.

7.2.2 Socket Class

Τα μηχανήματα που χρησιμοποιούμε εμφανίζουν τα λογικά τους cores ομαδοποιημένα σε sockets. Κάθε socket περιλαμβάνει έναν καθορισμένο αριθμό πυρήνων, ο οποίος οριοθετεί και το πλήθος των παράλληλων διεργασιών που γίνεται να υποστηριχθούν. Είναι υπεύθυνο για την εκτέλεση και εποπτεία των tasks που του έχουν ανατεθεί, καθώς και για την απελευθέρωση των πυρήνων που ολοκληρώνουν το έργο τους.

7.2.3 Node Class

Με τον όρο node αναφερόμαστε σε έναν υπολογιστή από αυτούς που απαρτίζουν το cluster. Στη δική μας προσέγγιση, ένα τέτοιο μηχάνημα συνίσταται από ένα σταθερό αριθμό όμοιων μεταξύ τους sockets. Σε λογικό, επομένως, επίπεδο, ένα node λειτουργεί ως framework για ένα σύνολο από socket instances. Δηλαδή, μία κλήση σε κάποια μέθοδο της κλάσης node ισοδυναμεί με μία κλήση στην αντίστοιχη μέθοδο όλων των socket objects.

7.2.4 Manager Class

Η κλάση αυτή περιλαμβάνει τις βασικές λειτουργίες του διαχειριστή πόρων. Υπάρχει μία μέθοδος που ελέγχει την ουρά, προκειμένου να βρει εργασίες που μπορούν να δρομολογηθούν. Μία δεύτερη, που αναθέτει σε αυτές τους αναγκαίους πόρους. Και, τέλος, μία τρίτη, που, κάθε φορά, διαθέτει την κατάσταση των nodes στο χρήστη. Αυτές θα μας απασχολήσουν στις επόμενες ενότητες.

7.3 Χρονοδρομολόγηση

Η χρονοδρομολόγηση ακολουθεί ένα συγκεκριμένο μοτίβο. Αρχικά, ανάλογα με τον επιλεγμένο αλγόριθμο, ταξινομείται η ουρά εργασιών. Ύστερα, προσπαθούμε να δρομολογήσουμε την κεφαλή αυτής. Αν η διαδικασία αυτή επιτύχει, αφαιρούμε την εν λόγω εργασία από την ουρά. Διαφορετικά, αν έχει υποδειχθεί, δοκιμάζουμε να κάνουμε backfilling με δουλειές μικρότερης προτεραιότητας. Επαναλαμβάνουμε τα προηγούμενα βήματα όσο η ουρά δεν είναι κενή και ολοκληρώνουμε την όλη διαδικασία μόλις αδειάσει το cluster (δηλ. όταν έχει τελειώσει η εκτέλεση όλων των εφαρμογών).

```
def scheduler(self, policy, fun, bf):
    flag = 0
    index = 0
    os.mkdir(self.piddir)
    p = init.makeqfile()
    while 1:
        self.free()
        index = self.readqueue(index)
        if self.queue:
            self.queue.sort(key = lambda job: (fun)(job), reverse = True)
            if self.submit(self.queue[0], policy, bf):
                job = self.queue.pop(0)
                job.startedat(datetime.datetime.now())
                self.scheduled.add(job)
            elif bf:
                self.backfill(policy)
        elif self.empty() and p.poll()==0:
            os.rmdir(self.piddir)
            flag = 1
        self.snapshot()
        if flag:
            break
        time.sleep(10)
```

Κώδικας 7.1: Χρονοδρομολόγηση

Σημειώνουμε πως για την αναπαράσταση των αλγορίθμων αρκούν τα σχετικά utility functions. Δεν χρειάζεται, δηλαδή, ξεχωριστή υλοποίηση για τον καθένα. Κρίθηκε προτιμότερο να υπάρχει ένα ενιαίο template, η εκτέλεση του οποίου διαφοροποιείται ανάλογα με το utility function που παρέχεται. Για τις ανάγκες της εργασίας, υποστηρίζονται οι FCFS και WFP3. Ωστόσο, μπορούν εύκολα να

προστεθούν και άλλοι αλγόριθμοι, αν εισαχθούν οι συναρτήσεις που τους εκφράζουν.

```
'FCFS': lambda j: aux.elapsed(j.enter)
'WFP3': lambda j: math.pow(aux.elapsed(j.enter)/j.duration,3)*j.procs
```

7.4 Backfilling

Είδαμε προηγουμένως πως, σε περίπτωση που δεν μπορέσουμε να δρομολογήσουμε το head της ουράς, ενδέχεται να χρειαστεί να κάνουμε backfilling. Και εδώ, η μέθοδος που ακολουθούμε είναι σχετικά τυποποιημένη. Για αρχή, πρέπει να υπολογίσουμε μετά από πόσο χρόνο θα μπορεί να εκτελεστεί η κεφαλή. Αυτό μπορεί να γίνει σχετικά απλά, ως εξής:

1. Βρίσκουμε τον αριθμό των μηχανημάτων που στερείται η συγκεκριμένη εργασία. Έστω x ο αριθμός αυτός.

2. Για κάθε απασχολημένο μηχάνημα του cluster, υπολογίζουμε τον χρόνο μετά από τον οποίο θα είναι ελεύθερο. Κάτι τέτοιο είναι εφικτό, επειδή ξέρουμε τις χρονικές διάρκειες των εργασίων που τρέχουν σε αυτό. Ταξινομούμε τους χρόνους που προκύπτουν σε σειρά αύξουσα.

3. Παρατηρούμε οτι ο x στη σειρά χρόνος είναι ο ελάχιστος μετά από τον οποίο θα έχουν ελευθερωθεί x ακριβώς μηχανήματα. Άρα, είναι ο χρόνος μετά από τον οποίο θα τρέξει η εργασία.

```
def backlog(self, job, nodes, i):
    occupied = [node for node in self.nodes if node not in nodes]
    window = map(lambda node: node.remaining(), occupied)
    job.runafter(sorted(window)[i-1])
```

Κώδικας 7.2: Χρονικό κατώφλι για backfilling

Η τιμή που βρήκαμε σηματοδοτεί το χρονικό περιθώριο για το backfilling. Φανερώνει, δηλαδή, το άνω όριο της διάρκειας των εφαρμογών που μπορούμε να δρομολογήσουμε.

```
def backfill(self, policy):
    i = 1
    while i < len(self.queue) and not self.full(policy):
        if self.queue[i].duration <= self.queue[0].interval:
            if self.submit(self.queue[i], policy, 1):
                job = self.queue.pop(i)
                job.startedat(datetime.datetime.now())
                self.scheduled.add(job)
        i += 1
```

Κώδικας 7.3: Backfilling

7.5 Κατανομή Πόρων

Κάθε απόπειρα δρομολόγησης επιτυγχάνει ή αποτυγχάνει, ανάλογα με τη διαθεσιμότητα των αναγκαίων πόρων.

```
def submit(self, job, policy, bf):
    policy = 'spare' if policy == 'strip' and job.exclusive else policy
    nodes = self.findnodes(job, policy, bf)
    if nodes and policy == 'compact':
        rankfile = self.bind(nodes, job, self.cores)
    elif nodes and policy == 'spare':
        rankfile = self.bind(nodes, job, self.cores//2)
    elif nodes and policy == 'strip':
        rankfile = self.bind(nodes, job, self.cores//2)
    else:
        return 0
    names = [node.name for node in nodes]
    hosts = '+' + ',+'.join(names)
    self.mpirun(hosts, job, rankfile)
    return 1
```

Κώδικας 7.4: Υποβολή εργασίας για εκτέλεση

Στην υλοποίησή μας, η διαδικασία ανάθεσης πόρων σε μια εργασία γίνεται σε τρία βήματα:

1. *Εύρεση μηχανημάτων*: Αναζητούμε nodes τα οποία λογίζονται ελεύθερα. Αν ο αριθμός τους δεν επαρκεί, δηλώνουμε αποτυχία.

```
def findnodes(self, job, policy, bf):
    if policy == 'compact':
        nodes = [node for node in self.nodes if not node.occupied(policy)]
        num = math.ceil(job.procs/(self.cores*self.sockets))
    elif policy == 'spare':
        nodes = [node for node in self.nodes if not node.occupied(policy)]
        num = math.ceil(2*job.procs/(self.cores*self.sockets))
    elif policy == 'strip':
        nodes = [node for node in self.nodes if not (node.occupied(policy)
                 or node.exclusive)]
        nodes.sort(key = lambda: node, node.precedence(self.heatmap, job))
        num = math.ceil(2*job.procs/(self.cores*self.sockets))
    if num > len(nodes):
        if bf and job == self.queue[0]:
            self.backlog(job, set(nodes), num-len(nodes))
        return []
    return nodes[:num]
```

Κώδικας 7.5: Εύρεση μηχανημάτων

2. *Ανάθεση διεργασιών*: Δεσμεύουμε τα cores των προηγούμενων μηχανημάτων με τα parallel tasks της προς δρομολόγηση εφαρμογής. Σε πρώτη φάση, το binding είναι εικονικό. Γίνεται, δηλαδή, στα python objects που έχουμε

φτιάξει. Σημειώνεται, ωστόσο, και σε ένα rankfile που θα χρειαστεί στην εκτέλεση με mpirun.

```
def bind(self, nodes, job, pps):
    if not os.path.exists(self.rankdir):
        os.mkdir(self.rankdir)
    rem = job.procs
    rf = open(os.path.join(self.rankdir, job.app + '.' +
            str(job.id) + '.' + 'rf'), 'w')
    for num,node in enumerate(nodes):
        node.exclusive = job.exclusive
        for socket in node.sockets:
            while rem:
                i = socket.freecore()
                socket.jobs[i] = job
                rf.write('rank' + ' ' + str(job.procs-rem) + '=+n' +
                        str(num) + ' ' + 'slot=' + str(socket.num) +
                        ':' + str(i) + '\n')
                rem -= 1
                if not rem%pps:
                    break
    return os.path.abspath(rf.name)
```

Κώδικας 7.6: Ανάθεση διεργασιών

3. *Εκτέλεση από φλοιό (shell)*: Τρέχουμε με mpirun στα nodes που έχουμε βρει παραπάνω, με χρήση και του σχετικού rankfile. Στο σημείο αυτό, έχουμε αξιοποιήσει τη βιβλιοθήκη subprocess [21], που μας επιτρέπει να κάνουμε spawn διεργασίες που τρέχουν στο shell, έξω από το κύριο νήμα εκτέλεσης του python κώδικα για τον resource manager.

```
def mpirun(self, hosts, job, rankfile):
    out = job.app + '.' + str(job.id) + '.' + 'o'
    err = job.app + '.' + str(job.id) + '.' + 'e'
    subprocess.call('echo '' > ' + self.piddir + '/' + \
                str(job.id), shell = True)
    mpirun = 'mpirun -H ' + hosts + ' -np ' + str(job.procs) +
            ' -v --report-bindings --timestamp-output ' + \
            '-rf ' + rankfile + ' ' + \
            self.appdir + '/' + job.app + ' 2>> ' + \
            self.logdir + '/' + err + ' 1>> ' + \
            self.logdir + '/' + out + ' && ' + \
            'rm' + ' ' + self.piddir + '/' + str(job.id) + '\n'
    subprocess.Popen(mpirun, shell = True)
```

Κώδικας 7.7: Εκτέλεση από φλοιό

Όπως φαίνεται και στα σχετικά αποσπάσματα κώδικα, η πολιτική κατανομής πόρων επηρεάζει το πως διαμορφώνονται τα προηγούμενα βήματα. Ο resource

manager μας υποστηρίζει τις τακτικές compact, strip και spare. Από αυτές, ασχολούμαστε μόναχα με τις πρώτες δύο:

- *Compact*: Ψάχνουμε μηχανήματα που είναι άδεια, προκειμένου να δεσμεύσουμε το σύνολο των λογικών τους cores (ή όσους χρειάζεται).

- *Strip*: Ψάχνουμε, επιπλέον, μηχανήματα που τρέχουν ήδη μία εργασία, ώστε να κάνουμε bind ως και τους μισούς τους πυρήνες. Προκειμένου να μειώσουμε φαινόμενα ανταγωνισμού, προσπαθούμε να ταιριάξουμε τις εφαρμογές μεταξύ τους. Η διαδικασία αυτή γίνεται με τη συνδρομή ενός heatmap σαν αυτό του σχήματος. Όσο μικρότερη η τιμή του κυτίου (i,j), τόσο πιο συμβατές είναι μεταξύ τους οι εφαρμογές της γραμμής i και της στήλης j .

	BT	*CG*	*EP*	*FT*	*IS*	*LU*	*MG*	*SP*
BT	6	5	1	8	4	2	7	3
CG	2	7	1	5	8	3	6	4
EP	4	7	8	5	6	3	2	1
FT	6	3	1	8	4	2	5	7
IS	3	7	1	6	8	2	5	4
LU	2	4	1	8	3	6	7	5
MG	2	4	1	7	6	3	8	5
SP	6	2	1	7	3	4	5	8

Πίνακας 7.2: Heatmap

7.6 Επικοινωνία με το cluster

Τονίσαμε νωρίτερα, πως το κλειδί στη λειτουργία του εργαλείου μας είναι η συνέπεια του state του αληθινού cluster με την εικονά του αντιγράφου που υπάρχει σε επίπεδο προγράμματος. Αναπόφευκτα, λοιπόν, πρέπει να υπάρχει αμφίδρομη επικοινωνία, ώστε η κάθε πλευρά να ενημερώνεται για τις αλλαγές της άλλης. Στα πλαίσια αυτής της εργασίας μας ενδιαφέρουν μεταβολές σε θέματα διαθεσιμότητας πυρήνων, που γνωστοποιούνται με δύο τρόπους:

1. *Εικονικό cluster* → *Πραγματικό cluster*: Μέσα από την εντολή mpirun κατά την εκτέλεση. Κάθε φορά, το rankfile δείχνει ποιες θέσεις του μηχανήματος θα δεσμευτούν.

2. *Πραγματικό cluster* → *Εικονικό cluster*: Με δεδομένο πως το εικονικό binding προηγείται του κανονικού, μας ενδιαφέρει η αποδέσμευση πόρων με το πέρας της εκτέλεσης μιας εργασίας. Στο πραγματικό σύστημα αυτό γίνεται αυτόματα. Το ζήτημα είναι πως θα επικαιροποιήσουμε τα μηχανήματα σε επίπεδο κώδικα. Ένας τρόπος είναι να διατηρούμε ένα pool με όλες τις εργασίες που έχουν δρομολογηθεί χωρίς να έχουν ολοκληρωθεί. Πρόκειται, ουσιαστικά, για ένα directory με dummy αρχεία στο εσωτερικό του. Τα

αρχεία φέρουν σαν όνομα το id της εργασίας στην οποία αντιστοιχούν. Δημιουργούνται από τον εικονικό manager πριν την εκτέλεση της εφαρμογής και διαγράφονται από τον πραγματικό μετά το τέλος αυτής. Κάθε εργασία της οποίας το id δεν απαντάται στο εν λόγω directory θεωρείται περατωμένη και οι θέσεις που καταλάμβανει στους πυηρήνες των εικονικών nodes μπορούν να εκκαθαριστούν.

Node	Socket	Taken	Status	Core 0	Core 1	Core 2	Core 3
n0	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n0	1	0/4	Empty	.	.	.	.
n1	0	4/4	Full	lu.B.x	lu.B.x	lu.B.x	lu.B.x
n1	1	4/4	Full	lu.B.x	lu.B.x	lu.B.x	lu.B.x
n2	0	4/4	Full	ep.B.x	ep.B.x	ep.B.x	ep.B.x
n2	1	0/4	Empty	.	.	.	.
n3	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n3	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n4	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n4	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n5	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n5	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n6	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n6	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n7	0	0/4	Empty	.	.	.	.
n7	1	0/4	Empty	.	.	.	.

Εικόνα 7.14: Θεωρούμε το benchmark ep.B.x, που τρέχει στο τρίτο κατά σειρά node. Η εφαρμογή έχει id ίσο με 3

Εικόνα 7.15: Παρατηρούμε πως έχει δημιουργηθεί το σχετικό dummy αρχείο

Εικόνα 7.16: Πλέον, το αρχείο δεν υφίσταται. Συνεπώς, η εκτέλεση έχει ολοκληρωθεί

Node	Socket	Taken	Status	Core 0	Core 1	Core 2	Core 3
n0	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n0	1	0/4	Empty	.	.	.	.
n1	0	4/4	Full	lu.B.x	lu.B.x	lu.B.x	lu.B.x
n1	1	4/4	Full	lu.B.x	lu.B.x	lu.B.x	lu.B.x
n2	0	0/4	Empty	.	.	.	.
n2	1	0/4	Empty	.	.	.	.
n3	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n3	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n4	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n4	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n5	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n5	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n6	0	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n6	1	4/4	Full	ft.B.x	ft.B.x	ft.B.x	ft.B.x
n7	0	0/4	Empty	.	.	.	.
n7	1	0/4	Empty	.	.	.	.

Εικόνα 7.17: Πράγματι, τα slots που βρισκόταν η εφαρμογή είναι άδεια

7.7 Παραμετροποίηση

Ο διαχειριστής λειτουργεί βάσει προδιαγραφών που καθορίζονται από τον χρήστη, μέσα από ένα yaml αρχείο. Εντός του αρχείου αυτού υπάρχουν πληροφορίες που αφορούν:

- Την τοποθεσία (absolute path) των εκτελέσιμων εφαρμογών στο μηχάνημα.
- Την τοποθεσία της ουράς του cluster.
- Τον επιλεγμένο αλγόριθμο χρονοδρομολόγησης.
- Τον αριθμό των διαθέσιμων μηχανημάτων στο cluster.
- Τον αριθμό των sockets ανά μηχάνημα και των πυρήνων ανά socket.
- Την επιλεγμένη πολιτική διαμοιρασμού πόρων.
- Την τοποθεσία των output αρχείων, όπως αυτά προκύπτουν από την εκτέλεση των εφαρμογών.
- Την τοποθεσία που αποθηκεύεται το state του cluster.

```
Version: 1.0

Applications:
    Path: /home/users/apapavas/test/NPB3.4/NPB3.4-MPI/bin
    Queue: /home/users/apapavas/rmanager/queue

Scheduling:
    Algorithm: FCFS
    Backfilling : 0

Nodes: 8

Sockets: 2

Cores: 4

Allocation:
    Policy: compact

Log  : /home/users/apapavas/rmanager/log/fcfs

State: /home/users/apapavas/rmanager/state
```

Κώδικας 7.8: Configuration αρχείο

7.8 Εποπτεία εκτέλεσης

Το κυρίως πρόγραμμα έχει σχεδιαστεί ώστε να δέχεται δύο παραμέτρους από την γραμμή εντολών:

1. <***-c,-config***>: Υποχρεωτική παράμετρος για το configuration αρχείο που θα δώσει ο χρήστης. Σε περίπτωση που δεν δοθεί κάποια τιμή, χρησιμοποιείται ένα file που έχει προεπιλεχθεί.

2. <***-i,-info***>: Προαιρετική παράμετρος που παρέχει πληροφορίες για την εκτέλεση. Οι πληροφορίες αυτές αντλούνται από μια τοποθεσία που προσδιορίζεται στο config αρχείο. Μπορεί να λάβει δύο τιμές:

   i. ***queue***: Δίνονται πληροφορίες για τις εργασίες εντός του συστήματος. Ειδικότερα, μπορούμε να δούμε:
      - Ποιες βρίσκονται στην ουρά.
      - Ποιες τρέχουν.
      - Πότε ξεκίνησε η εκτέλεσή τους.
      - Μετά από πόσο χρόνο θα ολοκληρωθούν.

   ii. ***state***: Δίνονται πληροφορίες για τα μηχανήματα. Πιο συγκεριμένα:
      - Ποιο είναι το τρέχον μηχάνημα.
      - Ποιο είναι το τρέχον socket του μηχανήματος.

```
apapavas@clone1:~/rmanager$ python3 ./main.py -c config/compact.yaml -i queue
+--------------------------------------------------------------+
|Id        App       Procs     Status    Started    Remaining |
+--------------------------------------------------------------+
|0         ft.B.x    32        R         13:39:40   00:00:33  |
|1         lu.B.x    32        R         13:39:50   00:00:28  |
|2         mg.B.x    8         Q         -          00:00:10  |
|3         lu.B.x    8         Q         -          00:02:00  |
|4         is.B.x    8         Q         -          00:00:05  |
+--------------------------------------------------------------+
```

Εικόνα 7.18: Πληροφορίες για τις εργασίες του συστήματος

- Πόσα cores είναι κατηλειμμένα σε κάθε socket.
- Ποιο task, αν υπάρχει, βρίσκεται σε κάθε επεξεργαστή.

```
apapavas@clone1:~/rmanager$ python3 ./main.py -c config/compact.yaml -i state
+----------------------------------------------------------------------------------+
|Node      Socket    Taken     Status    Core 0    Core 1    Core 2    Core 3    |
+----------------------------------------------------------------------------------+
|n0        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n0        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                                  |
|n1        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n1        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                                  |
|n2        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n2        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                                  |
|n3        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n3        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                                  |
|n4        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n4        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                                  |
|n5        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n5        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                                  |
|n6        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n6        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                                  |
|n7        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n7        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
+----------------------------------------------------------------------------------+
```

Εικόνα 7.19: Πληροφορίες για τα nodes

7.9 Ουρές εργασιών

Σε αντίθεση με έναν πραγματικό διαχειριστή, το πρόγραμμά μας δεν αλληλεπιδρά με τον χρήστη μέσω command line arguments. Για το λόγο αυτό, η ουρά δεν μπορεί να τροφοδοτηθεί από το πληκτρολόγιο (π.χ. με μια εντολή τύπου sbatch όπως στον Slurm). Αντίθετα, η κατασκευή της βασίζεται στην ανάγνωση ενός αρχείου, κάθε γραμμή του οποίου αντιστοιχεί σε μία εφαρμογή προς εκτέλεση. Η τυχαία εγγραφή του αρχείου έχει την ακόλουθη μορφή:

ft.B.x 8 60 0, όπου:

- Η πρώτη στήλη αντιστοιχεί στο όνομα της εφαρμογής (εκτελέσιμο).
- Η δεύτερη στήλη αντιστοιχεί στον αριθμό των παράλληλων διεργασιών αυτής.
- Η τρίτη στήλη αναγράφει, σε δευτερολεπτα, την εκτιμώμενη διάρκεια εκτέλεσης.
- Η τέταρτη στήλη δείχνει αν πρόκειται για exclusive εργασία. Αν, δηλαδή, η συγκεκριμένη εφαρμογή μπορεί να γίνει co-schedule με κάποια άλλη (τιμή *0*) ή όχι (τιμή *1*).

Η εν λόγω μέθοδος κατασκευής της ουράς δεν είναι ρεαλιστική, διότι προϋποθέτει πως γνωρίζουμε εκ των προτέρων ποιες εργασίες θα καταφτάσουν στο σύστημα. Προφανώς, σε έναν πραγματικό resource manager, κάτι τέτοιο δεν ισχύει, αφού οι δουλειές υποβάλλονται τυχαία. Μπορούμε, όμως, να διορθώσουμε αυτό το πρόβλημα, αν φροντίσουμε η κατασκευή του αρχείου να γίνεται δυναμικά, ταυτόχρονα με τη δρομολόγηση των εφαρμογών εντός αυτού. Στην πράξη, αρκεί μια συνάρτηση που λειτουργεί ως γεννήτορας εργασιών, εισάγοντας, μετά από κάθε εγγραφή, έναν παράγοντα αναμονής μέχρι την επόμενη. Σε python, η τιμή της χρονοκαθυστέρησης ρυθμίζεται εύκολα μέσω της βιβλιοθήκης random[22]. Η συνάρτηση αυτή εκτελείται παράλληλα με τον κυρίως κώδικα του resource manager, που, με τη σειρά του, διαβάζει ανά τακτά διαστήματα το αρχείο, ώστε να συγκεντρώσει τις νέες δουλειές που έχουν καταγραφεί.

```
def generator(num):
    apps = ['bt','cg','ep','ft','is','lu','mg','sp']
    procs = [2**i for i in range(1,7)]
    classes = ['B']
    walltime = {
            'bt':[550, 230, 230, 80, 90, 80, 80],
            'cg':[120, 90, 70, 60, 90, 150, 120],
            'ep':[70, 40, 20, 10, 5, 3, 2],
            'ft':[100, 70, 60, 50, 60, 50, 80],
            'is':[10, 5, 5, 5, 10, 20, 10],
            'lu':[260, 170, 120, 60, 45, 50, 50],
            'mg':[20, 20, 10, 10, 10, 5, 5],
            'sp':[640, 470, 470, 160, 160, 130, 130]
            }
    f = open('queue','a')
    for i in range(num):
        app = random.choice(apps)
        c = random.choice(classes)
        p = random.choice(procs)
        w = walltime[app][int(math.log(p,2))-1]
        interval = random.randint(3,10)
        f.write(app + '.' + c + '.' + 'x' + ' ' + str(p) + ' ' + str(w)
                + ' ' + str(0) + '\n')
        f.flush()
        time.sleep(interval)
```

Κώδικας 7.9: Γεννήτορας εργασιών

7.10 Παράδειγμα χρήσης

Έστω πως θέλουμε να δρομολογήσουμε πέντε εργασίες της κλάσης B των NPB, με το configuration που προσδιορίζεται στο yaml της ενότητας 7.7.

7.10.1 Πριν την εκτέλεση

Υποβάλλουμε το πρόγραμμά μας σαν bash script στην ουρά που επιθυμούμε. Αυτό γίνεται μέσω του αρμόδιου command line argument που προσφέρει ο διαχειριστής του συστήματος (π.χ. qsub για Torque).

```
#!/bin/bash

# Give the job a descriptive name
#PBS -N scheduler

## Output and error files
#PBS -o demo.out
#PBS -e demo.err

## Limit memory, runtime etc.
#PBS -l walltime=00:20:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8

export PATH=/various/common_tools/gcc-4.5.2/bin:$PATH
export LD_LIBRARY_PATH=/various/common_tools/gcc-4.5.2/lib64/

module load openmpi/1.8.3

cd /home/users/apapavas/rmanager

python3 main.py -c config/compact.yaml
```

Κώδικας 7.10: Bash script

Σημειώνεται πως, αρχικά, το working directory έχει ως εξής:

```
apapavas@scirouter:~/rmanager$ ls
aux.py  config  demo.sh  generator.py  heatmap  init.py  job.py  log  main.py  manager.py  node.py  __pycache__  shared.py  slot.py  stats.py
```

Εικόνα 7.20: Working directory πριν την εκτέλεση

7.10.2 Κατά την εκτέλεση

Όσο η εργασία τρέχει, το directory έχει την παρακάτω εικόνα:
Παρατηρούμε πως έχουν δημιουργηθεί δύο νέοι φάκελοι:

1. Ο φάκελος *pids*: Πρόκειται για το directory που καταγράφονται οι εν ενεργεία εφαρμογές κατά τον τρόπο που παρουσιάστηκε στην ενότητα 7.6.

```
apapavas@scirouter:~/rmanager$ ls
aux.py  config  demo.sh  generator.py  heatmap  init.py  job.py  log  main.py  manager.py  node.py  pids  __pycache__  queue  rankfiles  rewind  shared.py  slot.py  state  stats.py
```

Εικόνα 7.21: Working directory κατά την εκτέλεση

```
apapavas@scirouter:~/rmanager/pids$ ls
0  1
```

Εικόνα 7.22: Κατάλογος pids

2. Ο φάκελος rankfiles: Είναι η τοποθεσία για τα rankfiles που συνοδεύουν την εκτέλεση των εφαρμογών.

```
apapavas@scirouter:~/rmanager/rankfiles$ ls
ft.B.x.0.rf  is.B.x.4.rf  lu.B.x.1.rf  lu.B.x.3.rf  mg.B.x.2.rf
```

Εικόνα 7.23: Περιεχόμενα του φακέλου rankfiles

Ένα τέτοιο rankfile φαίνεται παρακάτω:

```
apapavas@scirouter:~/rmanager/rankfiles$ cat lu.B.x.3.rf
rank 0=+n0 slot=0:0
rank 1=+n0 slot=0:1
rank 2=+n0 slot=0:2
rank 3=+n0 slot=0:3
rank 4=+n0 slot=1:0
rank 5=+n0 slot=1:1
rank 6=+n0 slot=1:2
rank 7=+n0 slot=1:3
```

Εικόνα 7.24: Παράδειγμα rankfile

Με βάση τα αρχεία αυτά γίνεται αντιστοίχιση cores με parallel tasks στα πλαίσια της εκτέλεσης με την εντολή mpirun.

Βλέπουμε, επίσης, δύο νέα αρχεία:

1. Το αρχείο *queue*: Το αποθετήριο των υποβληθέντων εφαρμογών. Είναι το αρχείο που διαβάζει περιοδικά ο scheduler προκειμένου να συλλέξει νέες εργασίες.

2. Το αρχείο *state*: Εκεί καταγράφεται η τρέχουσα κατάσταση του cluster, που, μέσω CLI και των εντολών της ενότητας 7.8, προβάλλεται και στην οθόνη του χρήστη.

```
apapavas@clone1:~/rmanager$ python3 ./main.py -c config/compact.yaml -i queue
+----------------------------------------------------------+
|Id        App       Procs     Status    Started   Remaining |
+----------------------------------------------------------+
|0         ft.B.x    32        R         13:39:40  00:00:33  |
|1         lu.B.x    32        R         13:39:50  00:00:28  |
|2         mg.B.x    8         Q         -         00:00:10  |
|3         lu.B.x    8         Q         -         00:02:00  |
|4         is.B.x    8         Q         -         00:00:05  |
+----------------------------------------------------------+
```

Εικόνα 7.25: Πληροφορίες για τις εφαρμογές

```
apapavas@clone1:~/rmanager$ python3 ./main.py -c config/compact.yaml -i state
+------------------------------------------------------------------------------+
|Node      Socket    Taken     Status    Core 0    Core 1    Core 2    Core 3    |
+------------------------------------------------------------------------------+
|n0        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n0        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                              |
|n1        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n1        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                              |
|n2        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n2        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                              |
|n3        0         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|n3        1         4/4       Full      ft.B.x    ft.B.x    ft.B.x    ft.B.x    |
|                                                                              |
|n4        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n4        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                              |
|n5        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n5        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                              |
|n6        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n6        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|                                                                              |
|n7        0         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
|n7        1         4/4       Full      lu.B.x    lu.B.x    lu.B.x    lu.B.x    |
+------------------------------------------------------------------------------+
```

Εικόνα 7.26: Πληροφορίες για τα μηχανήματα

7.10.3 Μετά την εκτέλεση

Βλέπουμε πως ο φάκελος pids έχει διαγραφεί, συνεπώς δεν υπάρχουν active jobs.

```
apapavas@scirouter:~/rmanager$ ls
aux.py  config  demo.err  demo.out  demo.sh  generator.py  heatmap  init.py  job.py  log  main.py  manager.py  node.py  __pycache__  queue  rankfiles  rewind  shared.py  slot.py  state  stats.py
```

Εικόνα 7.27: Working directory μετά την εκτέλεση

Σε κάθε εφαρμογή αντιτοιχούν δύο output αρχεία:

1. Αρχείο *.err*: Σε αυτό καταγράφονται τα bindings μεταξύ πυρήνων και MPI processes. Χρησιμοποιείται για να εξακριβωθεί αν η εκτέλεση έγινε στα πρότυπα του αντίστοιχου rankfile. Αν για κάποιο λόγο η εκτέλεση αποτύχει, τα error logs θα αποθηκευτούν εδώ.

```
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 2 bound to socket 0[core 2[hwt 0]]: [././B/.][./././.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 3 bound to socket 0[core 3[hwt 0]]: [./././B][./././.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 4 bound to socket 1[core 4[hwt 0]]: [./././.][B/././.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 5 bound to socket 1[core 5[hwt 0]]: [./././.][./B/./.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 6 bound to socket 1[core 6[hwt 0]]: [./././.][././B/.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 7 bound to socket 1[core 7[hwt 0]]: [./././.][./././B]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 0 bound to socket 0[core 0[hwt 0]]: [B/././.][./././.]
Wed Apr  6 13:41:01 2022<stderr>:[clone1:10008] MCW rank 1 bound to socket 0[core 1[hwt 0]]: [./B/./.][./././.]
```

Εικόνα 7.28: Αρχείο .err

2. Αρχείο *.out*: Το αρχείο με τα αποτελέσματα της εκτέλεσης. Αναγράφει τα χαρακτηριστικά του προβλήματος (κλάση, μέγεθος, διεργασίες, iterations) και ενδεικτικές μετρικές επίδοσης (διάρκεια, Mop/s total, Mop/s/process).

```
Wed Apr  6 13:40:51 2022<stdout>:
Wed Apr  6 13:40:51 2022<stdout>:
Wed Apr  6 13:40:51 2022<stdout>: NAS Parallel Benchmarks 3.4 -- IS Benchmark
Wed Apr  6 13:40:51 2022<stdout>:
Wed Apr  6 13:40:51 2022<stdout>: Size:  33554432  (class B)
Wed Apr  6 13:40:51 2022<stdout>: Iterations:   10
Wed Apr  6 13:40:51 2022<stdout>: Number of processes:     8
Wed Apr  6 13:40:52 2022<stdout>:
Wed Apr  6 13:40:52 2022<stdout>:   iteration
Wed Apr  6 13:40:52 2022<stdout>:        1
Wed Apr  6 13:40:52 2022<stdout>:        2
Wed Apr  6 13:40:53 2022<stdout>:        3
Wed Apr  6 13:40:53 2022<stdout>:        4
Wed Apr  6 13:40:53 2022<stdout>:        5
Wed Apr  6 13:40:54 2022<stdout>:        6
Wed Apr  6 13:40:54 2022<stdout>:        7
Wed Apr  6 13:40:54 2022<stdout>:        8
Wed Apr  6 13:40:54 2022<stdout>:        9
Wed Apr  6 13:40:55 2022<stdout>:        10
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>: IS Benchmark Completed
Wed Apr  6 13:40:55 2022<stdout>: Class           =                        B
Wed Apr  6 13:40:55 2022<stdout>: Size            =                 33554432
Wed Apr  6 13:40:55 2022<stdout>: Iterations      =                       10
Wed Apr  6 13:40:55 2022<stdout>: Time in seconds =                     2.73
Wed Apr  6 13:40:55 2022<stdout>: Total processes =                        8
Wed Apr  6 13:40:55 2022<stdout>: Active procs    =                        8
Wed Apr  6 13:40:55 2022<stdout>: Mop/s total     =                   122.83
Wed Apr  6 13:40:55 2022<stdout>: Mop/s/process   =                    15.35
Wed Apr  6 13:40:55 2022<stdout>: Operation type  =              keys ranked
Wed Apr  6 13:40:55 2022<stdout>: Verification    =               SUCCESSFUL
Wed Apr  6 13:40:55 2022<stdout>: Version         =                      3.4
Wed Apr  6 13:40:55 2022<stdout>: Compile date    =              12 Oct 2021
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>: Compile options:
Wed Apr  6 13:40:55 2022<stdout>:    MPICC        = mpicc
Wed Apr  6 13:40:55 2022<stdout>:    CLINK        = $(MPICC)
Wed Apr  6 13:40:55 2022<stdout>:    CMPI_LIB     = (none)
Wed Apr  6 13:40:55 2022<stdout>:    CMPI_INC     = (none)
Wed Apr  6 13:40:55 2022<stdout>:    CFLAGS       = -O3
Wed Apr  6 13:40:55 2022<stdout>:    CLINKFLAGS   = $(CFLAGS)
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>: Please send feedbacks and/or the results of this run to:
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>: NPB Development Team
Wed Apr  6 13:40:55 2022<stdout>: npb@nas.nasa.gov
Wed Apr  6 13:40:55 2022<stdout>:
Wed Apr  6 13:40:55 2022<stdout>:
```

Εικόνα 7.29: Αρχείο .out

Πέρα από αυτά, υπάρχουν τα output και error files, στα οποία αποθηκεύονται, αντίστοιχα, output και error records σχετικά με το script. Στο παράδειγμά

μας, υπολογίζουμε ορισμένες μετρικές επίδοσης, που παρουσιάζονται στην επόμενη ενότητα.

```
apapavas@scirouter:~/rmanager$ cat demo.out
FCFS(0): (30.05, 44.62, 74.67)
```

Εικόνα 7.30: Δεδομένα εξόδου

8 Πειραματική Αξιολόγηση

8.1 Σενάρια δρομολόγησης

Θέλουμε να δούμε πως διαμορφώνεται η δρομολόγηση εφαρμογών των NPB σε σχέση με τις εκάστοτε ρυθμίσεις του διαχειριστή πόρων. Ειδικότερα, δοκιμάζουμε τις περιπτώσεις:

Πολιτική Κατανομής Πόρων	*Αλγόριθμος*	*Backfilling*
Compact	FCFS	Όχι
Compact	FCFS	Ναι
Compact	WFP3	Όχι
Compact	WFP3	Ναι
Strip[1]	FCFS	Όχι
Strip	FCFS	Ναι
Strip	WFP3	Όχι
Strip	WFP3	Ναι

Πίνακας 8.3: Μέθοδοι δρομολόγησης

Οι προηγούμενες παραμετροποιήσεις εφαρμόστηκαν μέσα από δύο πειράματα. Στο πρώτο, που έγινε στο server room του CSLab, επιχειρήσαμε να δρομολογήσουμε τρεις διαφορετικές ουρές με 50 εφαρμογές κλάσης B. Στο δεύτερο, που πραγματοποιήθηκε στον υπερυπολογιστή ARIS του ΕΔΥΤΕ, δοκιμάσαμε να εξυπηρετήσουμε τέσσερις ουρές 200 εφαρμογών των κλάσεων C και D. Η αξιολόγηση των προσομοιώσεών μας βασίζεται στις ακόλουθες παραμέτρους:

- *Average Waiting Time*: Μέσος χρόνος παραμονής μιας εργασίας στην ουρά.
- *Average Running Time*: Μέσος χρόνος εκτέλεσης για μια εργασία.
- *Average Turnaround Time*: Μέσος χρόνος απόκρισης των εργασιών. Ορίζεται ως το άθροισμα των δύο προηγούμενων μεγεθών.

[1] Σε strip πολιτική έγινε χρήση του heatmap του πίνακα 7.2

8.2 CSLab Server Room

Για τις δοκιμές που πραγματοποιήθηκαν αξιοποιήσαμε τα 16 μηχανήματα της ουράς clones. Τα μηχανήματα αυτά χρησιμοποιούν το ίδιο μοντέλο επεξεργαστή, τον Xeon E5335 της Intel. Πρόκειται για ένα πακέτο δύο τετραπύρηνων socket. Οι αναλυτικές προδιαγραφές ενός μηχανήματος και της ενσωματωμένης CPU φαίνονται στις εικόνες που ακολουθούν.

```
apapavas@clone1:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 15
Model name:            Intel(R) Xeon(R) CPU           E5335  @ 2.00GHz
Stepping:              7
CPU MHz:               2000.201
BogoMIPS:              4000.47
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              4096K
NUMA node0 CPU(s):     0-7
```

Εικόνα 8.31: Specifications ενός μηχανήματος

Performance Specifications	
Total Cores	4
Processor Base Frequency	2.00 GHz
Cache	8 MB L2 Cache
Bus Speed	1333 MHz
FSB Parity	Yes
TDP	80 W
VID Voltage Range	1.0000V-1.5000V

(a) Performance Specifications

Advanced Technologies	
Intel® Turbo Boost Technology ‡	No
Intel® Hyper-Threading Technology ‡	No
Intel® Virtualization Technology (VT-x) ‡	Yes
Intel® VT-x with Extended Page Tables (EPT) ‡	No
Intel® 64 ‡	Yes
Instruction Set	64-bit
Idle States	Yes
Enhanced Intel SpeedStep® Technology	Yes
Intel® Demand Based Switching	No
Thermal Monitoring Technologies	Yes

(b) Advanced Technologies

Εικόνα 8.32: Specifications του Xeon E5335

8.2.1 Πρώτη ουρά[2]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	256.83	70.96	327.79
FCFS(BF)	136.33	72.24	208.54
WFP3	249.85	72.3	322.15
WFP3(BF)	153.31	71.58	224.69

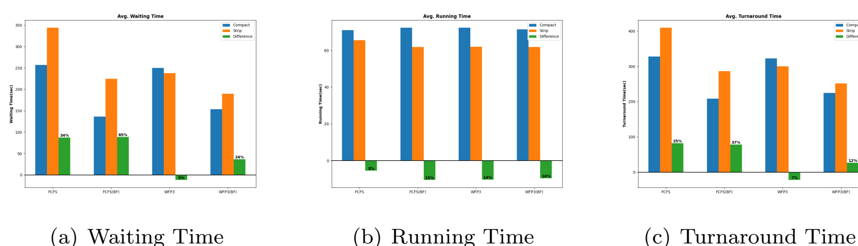
Πίνακας 8.4: Compact πολιτική για την πρώτη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	344.04	65.57	409.6
FCFS(BF)	224.91	61.75	286.66
WFP3	238.29	61.92	300.21
WFP3(BF)	189.86	61.73	251.59

Πίνακας 8.5: Strip πολιτική για την πρώτη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	34	-8	25
FCFS(BF)	65	-15	37
WFP3	-5	-14	-7
WFP3(BF)	24	-14	12

Πίνακας 8.6: Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.33: Διαγράμματα για την πρώτη ουρά

[2] https://github.com/alexispapabil/thesis/blob/main/exp/Class%20B/dataA

8.2.2 Δεύτερη ουρά[3]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	129.27	80.86	210.13
FCFS(BF)	104.82	80.55	185.37
WFP3	137.29	84.41	221.7
WFP3(BF)	132.37	83.84	216.21

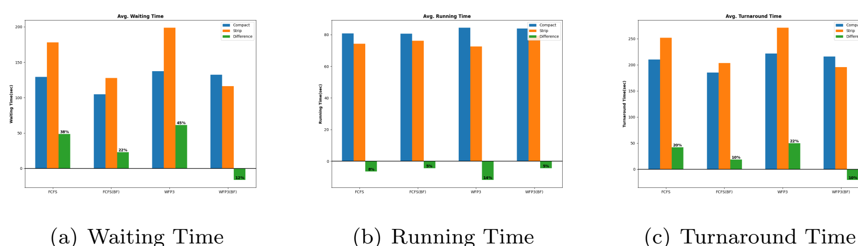
Πίνακας 8.7: Compact πολιτική για τη δεύτερη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	177.91	74.32	252.23
FCFS(BF)	127.6	76.15	203.75
WFP3	198.78	72.59	271.31
WFP3(BF)	116.16	79.47	195.63

Πίνακας 8.8: Strip πολιτική για τη δεύτερη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	38	-8	20
FCFS(BF)	22	-5	10
WFP3	45	-14	22
WFP3(BF)	-12	-5	-10

Πίνακας 8.9: Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.34: Διαγράμματα για τη δεύτερη ουρά

[3] `https://github.com/alexispapabil/thesis/blob/main/exp/Class%20B/dataB`

8.2.3 Τρίτη ουρά[4]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	469.67	121.94	591.61
FCFS(BF)	330.61	122.91	453.52
WFP3	372.24	123.23	495.48
WFP3(BF)	315.99	125.21	441.2

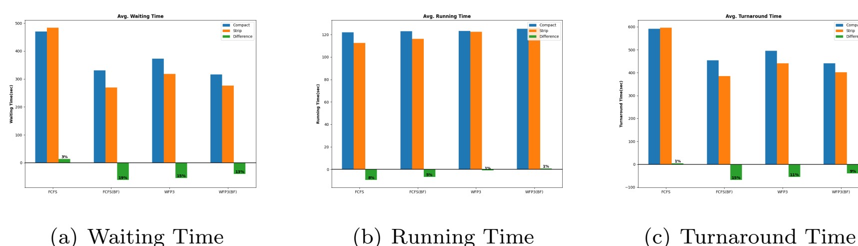
Πίνακας 8.10: Compact πολιτική για την τρίτη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	483.53	112.58	596.11
FCFS(BF)	269.17	116.15	385.32
WFP3	317.71	122.42	440.14
WFP3(BF)	275.88	126.09	401.96

Πίνακας 8.11: Strip πολιτική για την τρίτη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	3	-8	1
FCFS(BF)	-19	-5	-15
WFP3	-15	-1	-11
WFP3(BF)	-13	1	-9

Πίνακας 8.12: Ποσοστιαία μεταβολή των μετρήσεων της τρίτης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.35: Διαγράμματα για την τρίτη ουρά

[4] https://github.com/alexispapabil/thesis/blob/main/exp/Class%20B/dataC

8.2.4 Μέσοι όροι

Υπολογίζουμε τους αριθμητικούς μέσους των μετρήσεων που έχουν παρουσιαστεί, ώστε να έχουμε ενιαία εικόνα για το σύνολο των πειραμάτων.

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	285.26	91.25	376.51
FCFS(BF)	190.59	91.89	282.48
WFP3	253.13	93.31	346.44
WFP3(BF)	200.56	93.47	294.03

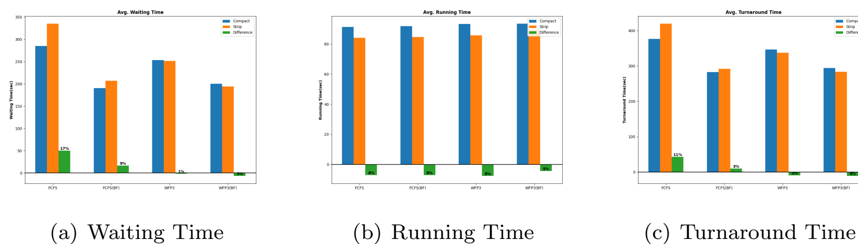
Πίνακας 8.13: Μέσοι όροι για compact πολιτική

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	335.16	84.16	419.31
FCFS(BF)	207.23	84.68	291.91
WFP3	251.59	85.64	337.24
WFP3(BF)	193.97	89.1	283.06

Πίνακας 8.14: Μέσοι όροι για strip πολιτική

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	17	-8	11
FCFS(BF)	9	-8	3
WFP3	-1	-8	-3
WFP3(BF)	-3	-5	-4

Πίνακας 8.15: Ποσοστιαία μεταβολή των μέσων όρων



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.36: Διαγράμματα μέσων όρων

8.3 ARIS

Στην περίπτωση του ARIS (Advanced Research Information System)[23], αποκτήσαμε πρόσβαση σε μηχανήματα της διαμέρισης compute. Πρόκειται για κόμβους αρχιτεκτονικής x86-64, οι οποίοι συνδέονται μεταξύ τους μέσα από ένα δίκτυο Infiniband FDR14 τοπολογίας fat tree. Τα επιμέρους μηχανήματα είναι εξοπλισμένα με μία κοινή επεξεργαστική μονάδα, την E5-2680v2 της Intel, που φέρει δύο sockets των 10 πυρήνων (το καθένα). Σημειώνεται πως οι εφαρμογές κλάσης C εκτελέστηκαν από 16 μηχανήματα, ενώ αυτές της τάξης D από 64.

Architecture	x86-64
Operating System	Redhat/Centos 6.7
Interconnect	
Technology	Infiniband FDR
Topology	Fat tree
Bandwidth [Gb/s]	56
Storage	
Type	IBM GPFS
Size [PByte]	1
Bandwidth [GB/s]	6
System Software	
Operating system	RedHat/Centos Linux 6.7
Batch system	SLURM
System Management	xCat IBM
Monitoring	Nagios, Ganglia

(a) Στοιχεία του ARIS

THIN nodes technical information	
Architecture	x86-64
System	IBM NeXtScale nx360 M4
Total number of nodes	426
Total number of cores	8520
Total amount of RAM [TByte]	27
Total Linpack Performance [TFlop/s]	180
Components	
Processor Type	Ivy Bridge - Intel Xeon E5-2680v2
Nominal Frequency [GHz]	2.8
Processors per Node	2
Cores per Processor	10
Cores per Node	20
Hyperthreading	OFF
Memory	
Memory per Node [GByte]	64

(b) Thin nodes specifications

Εικόνα 8.37: Specifications για το σύστημα ARIS και τα nodes της διαμέρισης compute[23]

8.3.1 Πρώτη ουρά (Κλάση C)[5]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	485.52	112.9	598.42
FCFS(BF)	330.18	112.87	443.02
WFP3	386.1	112.82	498.93
WFP3(BF)	386.25	112.89	499.14

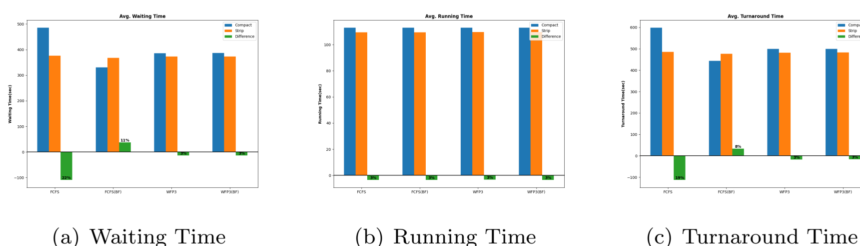
Πίνακας 8.16: Compact πολιτική για την πρώτη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	376.35	109.42	485.76
FCFS(BF)	367.26	109.43	476.69
WFP3	372.6	109.52	482.12
WFP3(BF)	372.8	109.51	482.5

Πίνακας 8.17: Strip πολιτική για την πρώτη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-22	-3	-19
FCFS(BF)	11	-3	8
WFP3	-3	-3	-3
WFP3(BF)	-3	-3	-3

Πίνακας 8.18: Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.38: Διαγράμματα για την πρώτη ουρά

[5] `https://github.com/alexispapabil/thesis/blob/main/exp/Class%20C/dataA`

8.3.2 Δεύτερη ουρά (Κλάση C)[6]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	477.31	94.81	572.18
FCFS(BF)	330.21	94.31	424.52
WFP3	364.26	94.57	458.82
WFP3(BF)	355.43	94.68	450.1

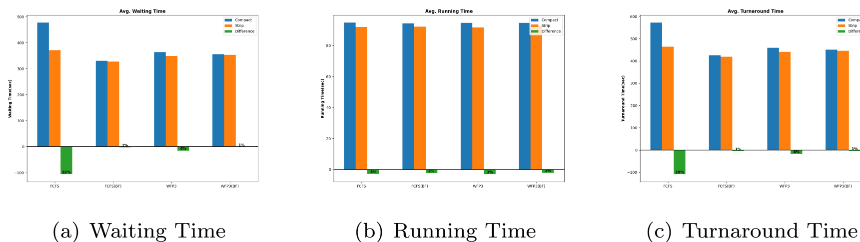
Πίνακας 8.19: Compact πολιτική για τη δεύτερη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	371.51	92.02	463.53
FCFS(BF)	327.25	92.12	419.37
WFP3	349.53	91.6	441.12
WFP3(BF)	353.59	92.66	446.25

Πίνακας 8.20: Strip πολιτική για τη δεύτερη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-22	-3	-19
FCFS(BF)	-1	-2	-1
WFP3	-4	-3	-4
WFP3(BF)	-1	-2	-1

Πίνακας 8.21: Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.39: Διαγράμματα για τη δεύτερη ουρά

[6] `https://github.com/alexispapabil/thesis/blob/main/exp/Class%20C/dataB`

8.3.3 Μέσοι Όροι (Κλάση C)

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	481.42	103.86	585.3
FCFS(BF)	330.2	103.59	433.79
WFP3	375.18	103.7	478.88
WFP3(BF)	370.84	103.79	474.62

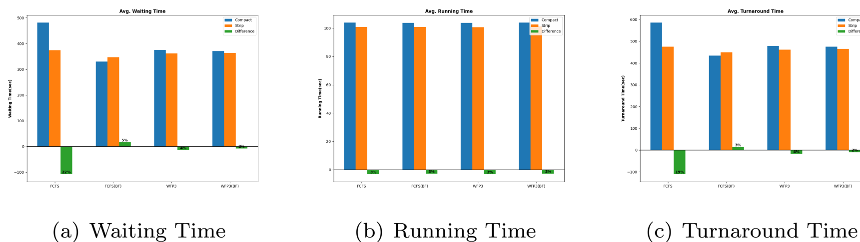
Πίνακας 8.22: Μέσοι όροι για compact πολιτική

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	373.93	100.72	474.65
FCFS(BF)	347.26	100.78	448.08
WFP3	361.07	100.56	461.62
WFP3(BF)	363.2	101.09	464.38

Πίνακας 8.23: Μέσοι όροι για strip πολιτική

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-22	-3	-19
FCFS(BF)	5	-3	3
WFP3	-4	-3	-4
WFP3(BF)	-2	-3	-2

Πίνακας 8.24: Ποσοστιαία μεταβολή των μέσων όρων



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.40: Διαγράμματα μέσων όρων

8.3.4 Πρώτη ουρά (Κλάση D)[7]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	776.75	213.16	989.91
FCFS(BF)	625.48	213.4	838.88
WFP3	578.92	213.69	792.61
WFP3(BF)	586.24	213.48	799.72

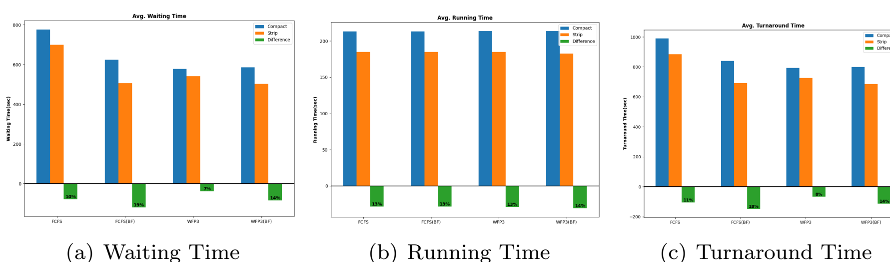
Πίνακας 8.25: Compact πολιτική για την πρώτη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	699.63	184.82	884.45
FCFS(BF)	506.27	185.07	691.34
WFP3	540.78	184.94	725.77
WFP3(BF)	503	182.78	685.78

Πίνακας 8.26: Strip πολιτική για την πρώτη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-10	-14	-11
FCFS(BF)	-19	-13	-18
WFP3	-7	-15	-8
WFP3(BF)	-14	-15	-14

Πίνακας 8.27: Ποσοστιαία μεταβολή των μετρήσεων της πρώτης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.41: Διαγράμματα για την πρώτη ουρά

[7] https://github.com/alexispapabil/thesis/blob/main/exp/Class%20D/dataA

8.3.5 Δεύτερη ουρά (Κλάση D)[8]

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	742.17	171.91	914.09
FCFS(BF)	535.77	172.25	708.03
WFP3	705.44	179.79	885.23
WFP3(BF)	706.15	180.88	887.04

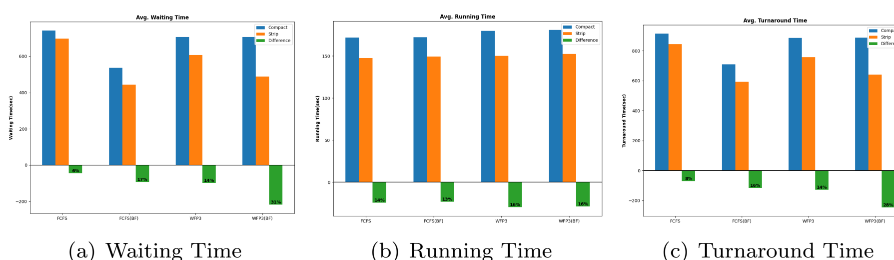
Πίνακας 8.28: Compact πολιτική για τη δεύτερη ουρά

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	696.8	147.61	844.41
FCFS(BF)	443.21	149.18	592.39
WFP3	606.92	150.15	757.07
WFP3(BF)	488.31	152.17	640.49

Πίνακας 8.29: Strip πολιτική για την δεύτερη ουρά

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-6	-14	-8
FCFS(BF)	-17	-13	-16
WFP3	-14	-16	-14
WFP3(BF)	-31	-16	-28

Πίνακας 8.30: Ποσοστιαία μεταβολή των μετρήσεων της δεύτερης ουράς



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.42: Διαγράμματα για τη δεύτερη ουρά

[8] `https://github.com/alexispapabil/thesis/blob/main/exp/Class%20D/dataB`

8.3.6 Μέσοι όροι (Κλάση D)

	Compact		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	759.46	192.54	952
FCFS(BF)	580.63	192.83	773.46
WFP3	642.18	196.74	838.92
WFP3(BF)	646.02	197.18	843.38

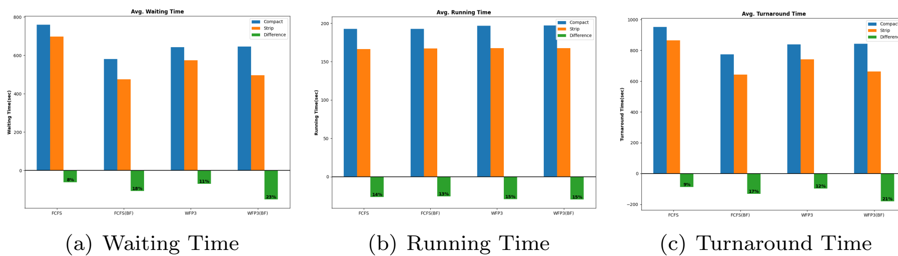
Πίνακας 8.31: Μέσοι όροι για compact πολιτική

	Strip		
	Waiting (sec)	*Running (sec)*	*Turnaround (sec)*
FCFS	698.22	166.22	864.43
FCFS(BF)	474.74	167.13	641.87
WFP3	573.85	167.55	741.42
WFP3(BF)	495.66	167.48	663.14

Πίνακας 8.32: Μέσοι όροι για strip πολιτική

	Ποσοστιαία μεταβολή		
	Waiting (%)	*Running (%)*	*Turnaround (%)*
FCFS	-8	-14	-9
FCFS(BF)	-18	-13	-17
WFP3	-11	-15	-12
WFP3(BF)	-23	-15	-21

Πίνακας 8.33: Ποσοστιαία μεταβολή των μέσων όρων



(a) Waiting Time (b) Running Time (c) Turnaround Time

Εικόνα 8.43: Διαγράμματα μέσων όρων

8.4 Παρατηρήσεις

Τα αποτελέσματα που βρήκαμε καθορίζονται από δύο παράγοντες: τον αλγόριθμο δρομολόγησης και την πολιτική κατανομής πόρων. Στην προσπάθειά μας να τα αξιολογήσουμε, θα απομονώσουμε την επίδραση του καθενός από αυτούς. Η ανάλυσή μας, δηλαδή, γίνεται υπό δύο άξονες:

1. *Σταθερή πολιτική κατανομής πόρων*: Μας ενδιαφέρουν οι ράβδοι ίδιου χρώματος. Παρατηρούμε πως ο WFP3 περιορίζει σημαντικά το waiting time, επειδή προωθεί δουλειές σύντομης διάρκειας. Την ίδια στιγμή, όμως, εργασίες που έχουν ξεμείνει στην ουρά αποκτούν, από ένα σημείο και πέρα, προτεραιότητα. Συνεπώς, οι εφαρμογές που μένουν πίσω επιβαρύνονται κατ' ελάχιστο. Βλέπουμε, επίσης, πως η επιλογή του backfilling συμφέρει και τους δύο αλγορίθμους. Ωστόσο, ο μεγαλύτερος κερδισμένος από μία τέτοια κίνηση είναι ο FCFS. Και αυτό, επειδή, εξ'ορισμού, διαθέτει μεγάλα περιθώρια βελτίωσης, σε αντίθεση με τον WFP3. Στην περίπτωση του τελευταίου, οι σύντομες σε διάρκεια εργασίες, που είναι οι πλέον ενδεδειγμένες για backfilling, έχουν, πιθανότατα, ήδη δρομολογηθεί. Δεν υπάρχουν, δηλαδή, πολλές επιλογές για να γεμίσουν οι άδειες θέσεις του cluster. Κλείνοντας, σημειώνουμε πως οι χρόνοι εκτέλεσης μένουν σταθεροί ανεξάρτητα από το ποιος αλγόριθμος έχει επιλεχθεί. Σε compact πολιτική το γεγονός αυτό είναι αναμενόμενο, διότι κάθε εφαρμογή τρέχει μόνη της στα μηχανήματα που έχει δεσμεύσει. Σε strip πολιτική έχουν σημασία οι αντιστοιχίες των εφαρμογών εντός του κάθε node. Οι τελευταίες δεν μπορούν να προβλεφθούν, εξαρτώνται από την αλληλουχία των εργασιών στην ουρά.

2. *Σταθερός αλγόριθμος δρομολόγησης*: Ασχολούμαστε, τώρα, με τις μπάρες που βρίσκονται στα ίδια σημεία του άξονα x. Διαπιστώνουμε, για αρχή, πως οι χρόνοι εκτέλεσης είναι μικρότεροι στην περίπτωση της strip πολιτικής. Το ταίριασμα των εφαρμογών, δηλαδή, φαίνεται να λειτουργεί ακόμα και όταν, αναπόφευκτα, υπάρχουν περιπτώσεις που οι δύο εργασίες που εκτελούνται δεν είναι απόλυτα συμβατές μεταξύ τους. Μάλιστα, η αξία του αναδεικνύεται περαιτέρω, όσο μετακινούμαστε σε μεγαλύτερα προβλήματα (τάξη D των NPB). Η μείωση του χρόνου εκτέλεσης οδηγεί και σε καλύτερη ρυθμαπόδοση (throughput) για το σύστημα, καθώς οι επιμέρους κόμβοι, που πλέον ελευθερώνονται πιο γρήγορα, υποχρεώνουν τις εργασίες σε μικρότερο χρόνο αναμονής.

Όπως και να έχει, η επιλογή του τρόπου που δρομολογούμε δεν μπορεί παρά να εξαρτάται από το τι θέλουμε να πετύχουμε κάθε φορά. Για τις εναλλακτικές που εξετάζουμε εδώ, εστιάζοντας στα πιο ρεαλιστικά-από πλευράς κλίμακας- πειράματα που έγιναν στο ARIS, φαίνεται πως οι καλύτερες επιλογές μοιάζουν αυτές των WFP3 ή FCFS (με backfilling) σε strip πολιτική.

9 Μελλοντικές επεκτάσεις

Στην παρούσα εργασία θεωρήσαμε πως τα μηχανήματα που διαχειριζόμαστε είναι όμοια. Ο κώδικας θα μπορούσε να τροποποιηθεί, για να μεριμνά για τα ιδιαίτερα χαρακτηριστικά κάθε μηχανήματος (π.χ. ύπαρξη GPUs, αυξημένη/μειωμένη μνήμη). Έτσι, ανάλογα με τις ανάγκες των εργασιών και τις υποδείξεις των χρηστών που τις υποβάλλουν, θα γίνεται η κατάλληλη επιλογή από τα διαθέσιμα nodes.

Μια άλλη προσέγγιση, αφορά την εξομοίωση πραγματικών συστημάτων διαχείρισης πόρων. Πρόκειται, δηλαδή, για την αντιστοίχιση των αρχιτεκτονικών components ενός resource manager στα μηχανήματα του cluster που έχουν δεσμευτεί. Στην περίπτωση του Slurm, για παράδειγμα, το ζητούμενο είναι η δημιουργία δύο νέων daemons που, ρυθμιζόμενοι από το χρήστη, θα μιμούνται τους slurmctld και slurmd. Με τον τρόπο αυτό δημιουργείται ένα πληρέστερο σύστημα, που προσφέρει υψηλές επιδόσεις.

A Παράρτημα

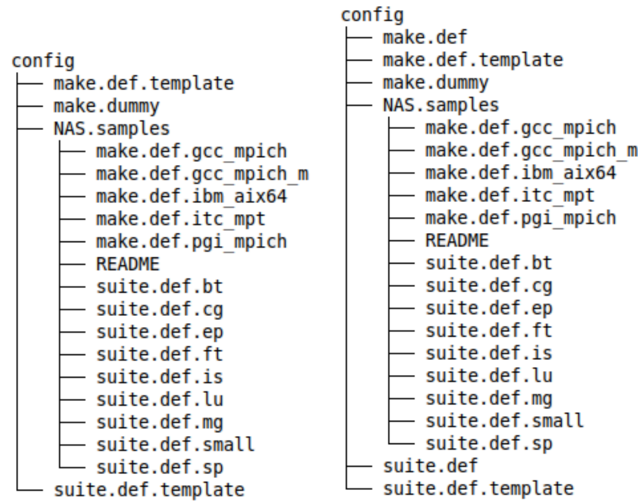
A.1 Nas Parallel Benchmarks (NPB)

Οι εφαρμογές των NPB μπορούν να μεταγλωττιστούν εύκολα ως μία σουίτα προγραμμάτων με τη βοήθεια των *make.def* και *suite.def*.

```
#---------------------------------------------------------------------------
# Parallel C:
#
# For IS, which is in C, the following must be defined:
#
# MPICC      - C compiler
# CFLAGS     - C compilation arguments
# CMPI_INC   - any -I arguments required for compiling MPI/C
# CLINK      - C linker
# CLINKFLAGS - C linker flags
# CMPI_LIB   - any -L and -l arguments required for linking MPI/C
#
# compilations are done with $(MPICC) $(CMPI_INC) $(CFLAGS) or
#                            $(MPICC) $(CFLAGS)
# linking is done with       $(CLINK) $(CMPI_LIB) $(CLINKFLAGS)
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# This is the C compiler used for MPI programs
#---------------------------------------------------------------------------
MPICC = mpicc
# This links MPI C programs; usually the same as ${MPICC}
CLINK   = $(MPICC)

#---------------------------------------------------------------------------
# These macros are passed to the linker to help link with MPI correctly
#---------------------------------------------------------------------------
CMPI_LIB =

#---------------------------------------------------------------------------
# These macros are passed to the compiler to help find 'mpi.h'
#---------------------------------------------------------------------------
CMPI_INC =

#---------------------------------------------------------------------------
# Global *compile time* flags for C programs
#---------------------------------------------------------------------------
CFLAGS  = -O3

#---------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#---------------------------------------------------------------------------
CLINKFLAGS = $(CFLAGS)


#---------------------------------------------------------------------------
```

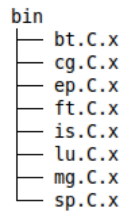
Εικόνα 1.44: Τμήμα του αρχείου *make.def*

Τα αρχεία αυτά περιλαμβάνουν, κατά σειρά, τα flags για τους επιλεγμένους compilers και τις εφαρμογές για τις οποίες θα δημιουργηθούν εκτελέσιμα. Υπάρχουν, μάλιστα, ενδεικτικά templates, τα οποία μπορούν να αντιγραφούν, όπως φαίνεται και στην εικόνα.



Εικόνα 1.45: Το directory config των NPB

Από τη μεριά του, και αφού εισάγει τις προτιμήσεις του, ο χρήστης δεν έχει παρά να τρέξει την εντολή `make suite` και να επιβεβαιώσει πως τα προβλεπόμενα αρχεία βρίσκονται στον φάκελο bin.



Εικόνα 1.46: Εκτελέσιμα για την κλάση C

A.2 CSLab Server Room

Τα μηχανήματα της ουράς clones διαθέτουν την έκδοση 3.4.2 της Python. Σε αυτό το περιβάλλον, χρειάζεται να γίνει εγκατάσταση της βιβλιοθήκης pyyaml, μέσω της εντολής `pip install pyyaml`. Πρέπει, επιπλέον, να κάνουμε export τις κατάλληλες μεταβλητέες περιβάλλοντος. Εδώ, χρησιμοποιούμε την έκδοση 4.5.2 του GNU και το μοντέλο 1.8.3 του Open MPI. Το τελικό script, μέσω του οποίου υποβάλλουμε και το πρόγραμμά μας, είναι το ακόλουθο:

```bash
#!/bin/bash

#Give the job a descriptive name
#PBS -N rmanager

## Output and error files
#PBS -o rmanager.out
#PBS -e rmanager.err

## Limit memory, runtime etc.
#PBS -l walltime=03:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=16:ppn=8

export PATH=/various/common_tools/gcc-4.5.2/bin:$PATH
export LD_LIBRARY_PATH=/various/common_tools/gcc-4.5.2/lib64/

module load openmpi/1.8.3

cd /home/users/apapavas/rmanager

python3 main.py -c config/strip.yaml
```

Κώδικας 1.11: Εκτέλεση στο cluster του CSLab

A.3 ARIS

Όπως και νωρίτερα, πρέπει να ικανοποιούνται οι εξαρτήσεις (dependencies) του κώδικά μας. Η βιβλιοθήκη pyyaml είναι ήδη εγκατεστημένη, άρα απομένει η επιλογή των κατάλληλων environmental variables. Στην περίπτωσή μας, αρκεί η Python 3.7.6, καθώς και οι εκδόσεις 5.5.0 και 4.0.1 των GNU και Open MPI. Τα αντίστοιχα modules φορτώνονται με την εντολή `module load`, όπως φαίνεται και στο script που ακολουθεί. Τονίζεται, πως για να υποστηριχθεί strip πολιτική, πρέπει να προσδιορίσουμε τη μνήμη ανά CPU (`--mem-per-cpu`) και όχι ανά node (`--mem`). Σε αντίθετη περίπτωση, η μνήμη, και άρα ο ίδιος ο κόμβος, δεσμεύεται εξ'ολοκλήρου από μία μόνο εφαρμογή.

```bash
#!/bin/bash -l

#SBATCH --job-name=rmanager
#SBATCH --output=rmanager.%j.out
#SBATCH --error=rmanager.%j.err
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=20
#SBATCH --cpus-per-task=1
#SBATCH --time=02:00:00
#SBATCH --mem-per-cpu=1400
#SBATCH --partition=compute
#SBATCH --account=pa220401
#SBATCH --exclusive

if [ x$SLURM_CPUS_PER_TASK == x ]; then
  export OMP_NUM_THREADS=1
else
  export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
fi

cd /work2/pa22/apapavas/rmanager

module purge

module load gnu/5.5.0
module load python/3.7.6
module load openmpi/4.0.1/gnu

python3 main.py -c config/strip.yaml
```

Κώδικας 1.12: Εκτέλεση στο ARIS

Βιβλιογραφία

[1] GRnet, *Εισαγωγή στους υπερυπολογιστές και το σύστημα Aris*. Available at: https://hpc.grnet.gr/supercomputer

[2] CSLab, *Parallel Processing Systems*, NTUA, Dec 2015, pp. 7-12.

[3] Mark Baker and Rajkumar Buyya, *Cluster Computing at a Glance*, Prentice Hall PTR, 1999, pp. 1-45.

[4] Dr. J. Lakshmi, *Understanding HPC Job Schedulers*, Indian Institute of Science.

[5] Dell, *HPC Scheduling and Resource Management*, 2010. Available at: https://i.dell.com/sites/content/business/solutions/whitepapers/en/Documents/hpc_scheduling_rm_011210.pdf

[6] S. Iqbal, R.Gupta and Y. Fang, *Planning Considerations for Job Scheduling in HPC Clusters*, Dell Power Solutions, Tech. Rep., 2005.

[7] Albert Reuther et al., *Scalable System Scheduling for HPC and Big Data*, 2017, p. 13.

[8] Thomas Sterling, Matthew Anderson and Maciej Brodowicz, *High Performance Computing: Modern Systems and Practices*, Morgan Kaufmann, 2017, pp. 142-144.

[9] Nikolaos Triantafyllis, *HPC Job Scheduling*, NTUA, 2018, pp. 36-38.

[10] Mina Naghshnejad, *Scheduling, Characterization and Prediction of HPC Workloads for Distributed Computing Environments*, University of California, 2019, pp. 6-7.

[11] Wei Tang et al., *Fault Aware, Utility Based Job Scheduling on Blue Gene/P Systems*, IEEE International Conference on Cluster Computing and Workshops, 2009.

[12] Li Yu et al., *System-Wide Tradeoff Modeling of Performance, Power, and Resilience on Petascale Systems*, The Journal of Supercomputing, 2017.

[13] Yuping Fan, *Job Scheduling in High Performance Computing*, Illinois Institute of Technology, 2021, pp. 1-2.

[14] Morris Jette, Mark Grondona, *SLURM: Simple Linux Utility for Resource Management*, Proceedings of ClusterWorld Conference and Expo, San Jose, California, Jun 2003. Available at : https://slurm.schedmd.com/publications.html

[15] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Computer System Laboratory*, Jun 2021. Available at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, pp. 1-2.

[16] Edgar Gabriel ta al., *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, 11th European PVM/MPI Users' Group Meeting, 2004. Available at: `https://www-lb.open-mpi.org/papers/euro-pvmmpi-2004-overview/euro-pvmmpi-2004-overview.pdf`

[17] Open MPI, *FAQ: Compiling MPI jobs*. Available at: `https://www.open-mpi.org/faq/?category=mpi-apps`

[18] Open MPI, *mpirun(1) man page (version 1.8.8)*. Available at: `https://www.open-mpi.org/doc/v1.8/man1/mpirun.1.php`

[19] D.H. Bailey et al., *THE NAS PARALLEL BENCHMARKS*, NASA Publications, 1991.

[20] NASA Advanced Supercomputing (NAS) Division, *NAS Parallel Benchmarks*. Available at: `https://www.nas.nasa.gov/software/npb.html`

[21] Python, *Subprocess management*. Available at: `https://docs.python.org/3/library/subprocess.html`

[22] Python, *Generate pseudo-random numbers*. Available at: `https://docs.python.org/3/library/random.html`

[23] ARIS Documentation, *Hardware Overview*, Available at: `https://doc.aris.grnet.gr/system/hardware/`