ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και προγραμματιστική υλοποίηση αλγορίθμων χρονοδρομολόγησης σε χρονοδρομολογητή εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Στεφ. Γιαλίδης

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

Αθήνα, Οκτώβριος 2019

Εθνικο Μετσοβιο Πολυτεχνειο
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ
ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και προγραμματιστική υλοποίηση αλγορίθμων χρονοδρομολόγησης σε χρονοδρομολογητή εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Στεφ. Γιαλίδης

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Οκτωβρίου 2019

(*Υπογραφή*)	(*Υπογραφή*)	(*Υπογραφή*)
.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου
Επ. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

.....
Αλέξανδρος Στεφ. Γιαλίδης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα υπολογιστικά συστήματα μεγάλης κλίμακας και συγκεκριμένα τα υπολογιστικά συστήματα υψηλής επίδοσης (High Performance Computing clusters) είναι ευρέως διαδεδομένα και συχνά χρησιμοποιούνται για την επίλυση πολύπλοκων προβλημάτων. Όσο εξαπλώνεται η εφαρμογή τους, η κατάλληλη χρονοδρομολόγηση των εργασιών σ΄ αυτά, αποτελεί καθοριστικό παράγοντα για την αποτελεσματική χρήση αλλά και την εξοικονόμηση στην κατανάλωση ενέργειας. Το σύστημα διαχείρισης πόρων αποτελεί αναπόσπαστο κομμάτι του λογισμικού ενός HPC cluster, καθώς αναλαμβάνει να διαμοιράσει τους υπολογιστικούς πόρους στις αντίστοιχες εργασίες.

Το Slurm είναι ένα σύστημα διαχείρισης πόρων ανοιχτού κώδικα το οποίο έχει βασιστεί σε αρχές όπως η απλότητα, η επεκτασιμότητα, η φορητότητα και η κλιμακωσιμότητα. Το σύνολο αυτών των χαρακτηριστικών το καθιστούν ένα απο τα δημοφιλέστερα συστήματα σε αυτόν τον τομέα καθώς χρησιμοποιείται από την πλειοψηφία των υπερυπολογιστών που βρίσκονται στη λίστα Top500.

Σκοπός της παρούσας διπλωματικής εργασίας είναι η ανάλυση του Slurm, η μελέτη μερικών τεχνικών χρονοδρομολόγησης εργασιών που υποστηρίζει και η προγραμματιστική υλοποίηση μιας επέκτασης χρονοδρομολόγησης για το σύστημα. Εκτελείται σενάριο υποβολής ενός συνόλου εργασιών ώστε να γίνει ποιοτική σύγκριση μεταξύ του ενσωματωμένου χρονοδρομολογητή του Slurm και της επέκτασης που υλοποιήθηκε.

Λέξεις Κλειδιά

Slurm, Σύστημα Διαχείρισης Πόρων, Χρονοδρομολόγηση Εργασιών, Συστοιχίες Υπολογιστών, Συστήματα Μεγάλης Κλίμακας, Εικονικοποίηση, Docker

Abstract

Large scale computing systems and specifically High Performance Computing clusters are of wide extent and often used for solving complex problems. As they become widespread, proper job scheduling constitutes a key factor for resource efficiency and minimization of energy consumption. The resource management system plays a vital role in an HPC cluster, as it is responsible for allocating resources to the corresponding jobs.

Slurm is an open-source resource management system with core values such as simplicity, extensibility, portability and scalability. This set of characteristics makes it one of the most popular systems in this field, being used by the majority of the supercomputers included in the Top500 list.

The purpose of this diploma thesis is the analysis of Slurm, the study of some job scheduling techniques that it offers as well as the implementation of a scheduling plugin for the system. A scenario is performed in which a set of jobs is submitted to the system in order to make a quality-based comparison between the embedded job scheduling mechanism of Slurm and the implemented plugin.

Keywords

Slurm, Resource Management System, Job Scheduling, Cluster, HPC, Virtualization, Docker

Ευχαριστίες

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

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Υποψήφιο Διδάκτορα Νικόλαο Τριανταφύλλη για την υπομονή, επιμονή και συνεχή καθοδήγησή του κατά τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και για την ενθάρρυνσή του και το χρόνο που αφιέρωσε. Χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δεν θα ήταν εφικτή.

Επιπλέον, επιβάλλεται να ευχαριστήσω τους φίλους μου και τους συμφοιτητές μου για την πολύτιμη βοήθειά τους σε επιστημονικό και προσωπικό επίπεδο.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου, τους γονείς μου και τις δύο αδερφές μου για την αγάπη, την αμέριστη υποστήριξή τους όλα αυτά τα χρόνια και την εμπιστοσύνη τους σε κάθε μου επιλογή.

Αλέξανδρος Γιαλίδης

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κώδικες

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της Διπλωματικής

Οι υπερυπολογιστές αποτελούν ίσως τον σημαντικότερο τομέα ανάπτυξης της σύγχρονης εποχής, με ασυναγώνιστη επίδραση σε τομείς όπως οι βιοεπιστήμες, οι γεωεπιστήμες, η μετεωρολογία αλλά και στην έρευνα γενικότερα. Καμία άλλη τεχνολογία δεν έχει ακολουθήσει παρόμοιο ρυθμό ανάπτυξης. Από τις 1000 πράξεις ανά δευτερόλεπτο στα τέλη του 1940 στις 10^{15} πράξεις (PetaFlops) σήμερα, η ταχύτητα των υπερυπολογιστών βελτιώνεται σταθερά κατά περίπου 200 φορές κάθε δεκαετία με τη βοήθεια μιας σειράς από επιτεύγματα στην τεχνολογία, στην αρχιτεκτονική, στους αλγορίθμους και στο λογισμικό συστημάτων.

Τα παραπάνω συστήματα, όπως είναι λογικό, καταναλώνουν υψηλά επίπεδα ενέργειας. Το κόστος λειτουργίας τους πολλές φορές είναι της τάξης των εκατομμυρίων δολαρίων ετησίως. Για το λόγο αυτό, δεν είναι τυχαίο που οι ιδιοκτήτες υπερυπολογιστών ερευνούν με μεγάλη προσοχή πώς χρησιμοποιούνται οι πόροι του συστήματος και με ποιο τρόπο η χρήση τους μπορεί να μεγιστοποιηθεί. Σε αυτή την εργασία κινούμαστε προς αυτή την κατεύθυνση, μελετώντας τα συστήματα διαχείρισης πόρων και κατ΄ επέκταση τις τεχνικές χρονοδρομολόγησης εργασιών που εφαρμόζουν.

Ένα σύστημα διαχείρισης πόρων (resource management system) διαχειρίζεται τον υπολογιστικό φόρτο εμπδοδίζοντας τις εργασίες να ανταγωνίζονται η μία την άλλη για ένα περιορισμένο σύνολο υπολογιστικών πόρων. Ένα τέτοιο σύστημα αποτελείται από έναν διαχειριστή πόρων (resource manager) και έναν χρονοδρομολογητή εργασιών (job scheduler). Συνήθως, ο διαχειριστής πόρων εκτελεί προγράμματα στους διαφορετικούς κόμβους (nodes) του συστήματος ώστε να πληροφορείται για την κατάστασή τους. Επίσης, δημιουργεί τις κατάλληλες ουρές εργασιών ώστε οι χρήστες να είναι σε θέση να υποβάλλουν εργασίες σε αυτό. Οι χρήστες μπορούν να ενημερώνονται για την κατάσταση των εργασιών αλλά και για τη διαθεσιμότητα του συστήματος. Ο χρονοδρομολογητής εργασιών, ο οποίος μπορεί να είναι είτε ενσωματωμένος στον διαχειριστή πόρων είτε εξωτερικός,

ενημερώνεται περιοδικά από το διαχειριστή πόρων σχετικά με τις ουρές και τους διαθέσιμους υπολογιστικούς πορους και φτιάχνει ένα χρονοδιάγραμμα με τη σειρά εκτέλεσής τους, βάσει της πολιτικής χρονοδρομολόγησης που έχει οριστεί από τον διαχειριστή του συστήματος.

Σκοπός αυτή της διπλωματικής εργασίας είναι αρχικά η ανάλυση του συστήματος διαχείρισης πόρων που ονομάζεται Slurm. Στη συνέχεια, γίνεται μελέτη μερικών τεχνικών χρονοδρομολόγησης εργασιών που υποστηρίζει ανάλογα με τις ανάγκες. Βασιζόμενοι στο γεγονός ότι το σύστημα αυτό είναι εν γένει επεκτάσιμο, υλοποιείται μια επέκταση χρονοδρομολόγησης για αυτό. Τέλος, γίνεται ποιοτική σύγκριση της σειράς χρονοδρομολόγησης των εργασιών μεταξύ της επέκτασης που δημιουργήθηκε και του ενσωματωμένου αλγορίθμου χρονοδρομολόγησης μέσω υποβολής ενός συνόλου εργασιών στο σύστημα.

1.2 Δομή της Εργασίας

Η εργασίας αυτή είναι χωρισμένη σε επτά κεφάλαια και δύο παραρτήματα. Στο Κεφάλαιο 2 γίνεται αναφορά στους υπερυπολογιστές και ειδικότερα στις συστοιχίες υπολογιστών (clusters) που χρησιμοποιούνται ολοένα και περισσότερο σε σχέση με αρχιτεκτονικές ειδικού σκοπού.

Στο Κεφάλαιο 3 αναφέρονται οι λόγοι που ένα σύστημα διαχείρισης πόρων έχει κομβικό ρόλο σε έναν υπερυπολογιστή για την επίτευξη αποτελεσματικής αξιοποίησης των υπολογιστικών πόρων.

Στο Κεφάλαιο 4 αναλύεται ο Slurm Workload Manager, ένα σύστημα διαχείρισης πόρων που χρησιμοποιείται από την πλειοψηφία των υπολογιστών που βρίσκονται στη λίστα Top500.

Στο Κεφάλαιο 5 γίνεται μελέτη ορισμένων τεχνικών χρονοδρομολόγησης εργασιών που προσφέρει το Slurm.

Στο Κεφάλαιο 6 γίνεται ποιοτική σύγκριση μεταξύ του ενσωματωμένου χρονοδρομολογητή εργασιών και μιας νέας παραμετροποιήσιμης επέκτασης χρονοδρομολόγησης εκτελώντας ένα σενάριο τεχνητού φόρτου εργασίας του συστήματος.

Το Κεφάλαιο 7 περιλαμβάνει τα συμπεράσματα της εργασίας, ενώ συζητούνται πιθανές κατευθύνσεις επέκτασης και βελτιστοποίησης.

Το Παράρτημα Α αναφέρεται στην εικονικοποίηση και συγκεκριμένα τη χρήση containers μέσω Docker.

Στο Παράρτημα Β δίνονται οδηγίες για την εγκατάσταση του περιβάλλοντος ανάπτυξης αυτής της εργασίας και παρουσιάζονται οι αλλαγές στον πηγαίο κώδικα του Slurm που χρειάστηκε για την ανάπτυξη της επέκτασης χρονοδρομολόγησης.

Κεφάλαιο 2

Υπερυπολογιστές

2.1 Εισαγωγή

Η επίλυση σε εύλογο χρονικό διάστημα μεγάλων υπολογιστικών προβλημάτων, που προκύπτουν σε πολλούς τομείς της έρευνας και των εφαρμογών, έχει υψηλότερες απαιτήσεις σε υπολογιστική ισχύ και μνήμη, από ότι μπορεί να προσφέρει σήμερα ο καλύτερος σειριακός υπολογιστής, της τάξης των Pflops (Floating-point Operations per Second). Τέτοιοι υπολογισμοί είναι εφικτοί με τη χρήση υπολογιστών με πολλούς επεξεργαστές σε συνδυασμό με μεθόδους παράλληλης επεξεργασίας. Αυτές οι μέθοδοι στοχεύουν στην ταυτόχρονη εκμετάλλευση πολλών επεξεργαστών για την αντιμετώπιση μιας μεγάλης υπολογιστικής διεργασίας (task), χωρίζοντάς την σε μικρότερες ανεξάρτητες υποδιεργασίες (subtasks). Αυτές οι διατάξεις είναι γνωστές ως υπερυπολογιστές (supercomputers) ή υπολογιστές υψηλής επίδοσης (high performance computers).

2.2 Ταξινόμηση του **Flynn**

Ανάλογα με τον τρόπο παροχής των εντολών και των δεδομένων στις μονάδες επεξεργασίας διακρίνονται τέσσερις κατηγορίες (Flynn's Taxonomy [5]) αρχιτεκτονικής υπολογιστών 2.1, από τις οποίες η μία είναι σειριακή και οι υπόλοιπες τρεις έχουν δυνατότητες παράλληλης επεξεργασίας:

i *Αρχιτεκτονική Μοναδικής Εντολής Μοναδικού Δεδομένου (Single Instruction Single Data, SISD)*: μια μονάδα επεξεργασίας εκτελεί ακολουθιακά τις εντολές (μια προς μια) πάνω σε μια σειρά δεδομένων. Πρόκειται για το κλασικό μοντέλο σειριακής αρχιτεκτονικής Von Neumann. Παραδείγματα τέτοιων υπολογιστών αποτελούν οι προσωπικοί υπολογιστές (PC) και οι σταθμοί εργασίας (workstations).

ii *Αρχιτεκτονική Μοναδικής Εντολής Πολλαπλών Δεδομένων (Single Instruction Multiple Data, SIMD)*: οι μονάδες επεξεργασίας εκτελούν

συγχρονισμένα μια κοινή ακολουθία εντολών πάνω σε διαφορετικά δεδομένα. Οι υπολογιστές της κατηγορίας αυτής, ανάλογα με τον τρόπο διαχείρισης των δεδομένων, αναφέρονται με τον όρο διανυσματικοί υπολογιστές (vector computers) ή υπολογιστές πινάκων (array computers). Σε αυτήν την κατηγορία, ανήκουν οι υπολογιστές Cray, ILLIAC IV, Connection Machine.

iii *Αρχιτεκτονική Πολλαπλών Εντολών Μοναδικού Δεδομένου (Multiple Instruction Single Data, MISD)*: ελάχιστοι υπολογιστές μπορούν να ενταχθούν σε αυτήν την κατηγορία. Η αρχιτεκτονική αυτή δεν χρησιμοποιήθηκε για επιστημονικούς υπολογισμούς και σήμερα έχει εκλείψει. Παράδειγμα αποτελεί ο υπολογιστής ελέγχου πτήσης του Space Shuttle.

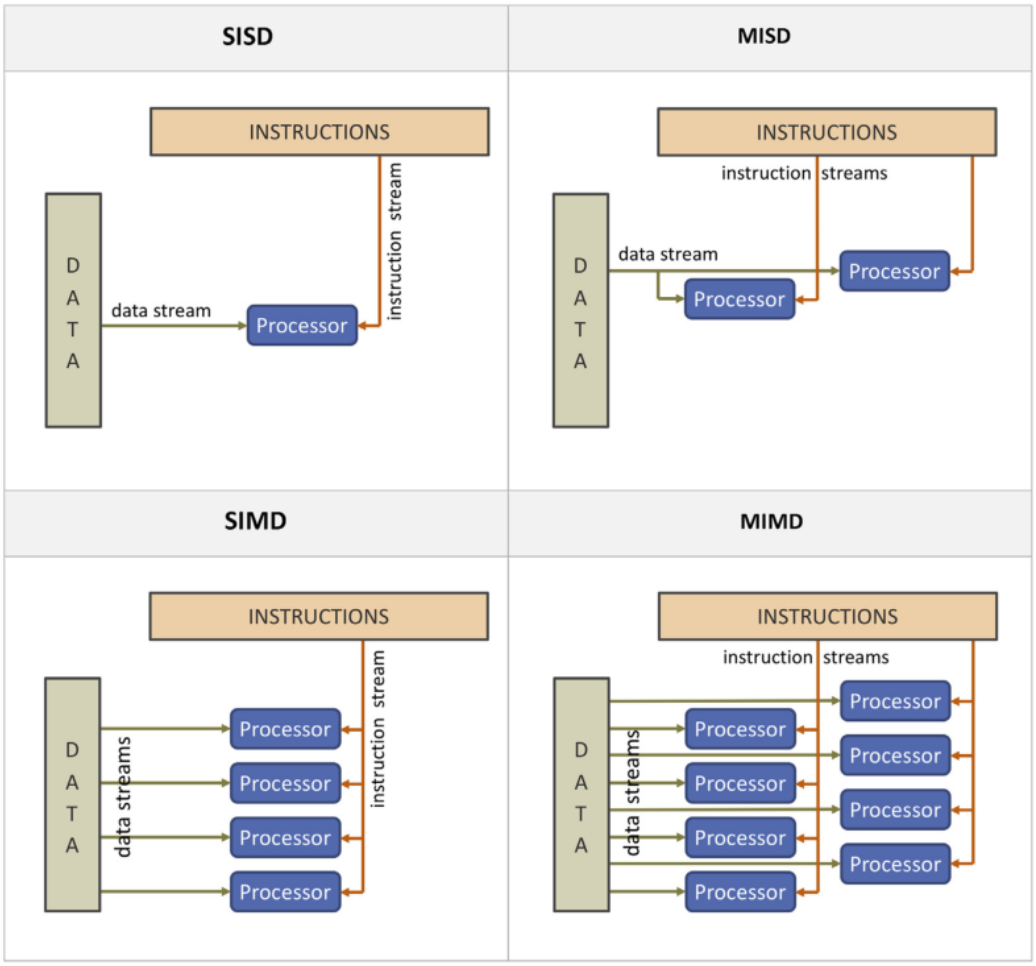
iv *Αρχιτεκτονική Πολλαπλών Εντολών Πολλαπλών Δεδομένων (Multiple Instruction Multiple Data, MIMD)*: οι μονάδες επεξεργασίας είναι ελεύθερες να εκτελούν, ανεξάρτητα η μία από την άλλη οποιαδήποτε εντολή με οποιαδήποτε δεδομένα. Οι υπολογιστές αυτής της αρχιτεκτονικής αναφέρονται ως υπολογιστές πολυεπεξεργαστών (multiprocessor computers). Υπολογιστές που ανήκουν στην κατηγορία αυτή είναι οι Intel Paragon, Intel Xeon Phi.

2.3 Πολυεπεξεργαστικά Συστήματα

Οι παράλληλοι υπολογιστές που βασίζονται σε πολυεπξεργαστές (multiprocessors) είναι η κύρια μορφή υπερυπολογιστών αυτή τη στιγμή. Σε γενικές γραμμές, πρόκειται για οποιοδήποτε σύστημα αποτελείται από ένα σύνολο αυτοδύναμων υπολογιστών συνδεδεμένων μέσω ενός δικτύου και οργανωμένα ώστε να εκτελούν κοινό φόρτο εργασίας. Όπως αναφέραμε προηγουμένως, οι multiprocessors ανήκουν στην κατηγορία μηχανημάτων MIMD.

Η σημασία των πολυεπεξεργαστικών συστημάτων για τον τομέα των υπερυπολογιστών εκτοξεύτηκε με την μεγάλη ανάπτυξη της τεχνολογίας VLSI και της αρχιτεκτονικής των μικροεπεξεργαστών. Αυτά τα δύο σηματοδότησαν μια σημαντική αλλαγή στις τάσεις και την κατεύθυνση των υπερυπολογιστών. Η μείωση του κόστους μέσω της εκμετάλλευσης της μαζικής αγοράς μικροεπεξεργαστών γενικής χρήσης καθόρισαν την επόμενη γενιά υπολογιστών υψηλών επιδόσεων. Η ενσωμάτωση μικροεπεξεργαστών που προέρχονται από τις ευρύτερες αγορές σταθμών εργασίας, προσωπικών υπολογιστών και servers ως βασικές υπολογιστικές μηχανές των υπερυπολογιστών είχε δραματικό αντίκτυπο στην αρχιτεκτονική συστημάτων μεγάλης κλίμακας, εκτοπίζοντας σε μεγάλο βαθμό τις προηγούμενες εξειδικευμένες λύσεις.

Οι πιο βασικές μορφές με τις οποίες εμφανίζονται τα πολυεπεξεργαστικά συστήματα είναι τρεις [6]: πολυεπεξεργαστικά συστήματα μοιραζόμενης μνήμης, συστήματα με μαζικά παράλληλους επεξεργαστές και συστοιχίες υπολογιστών



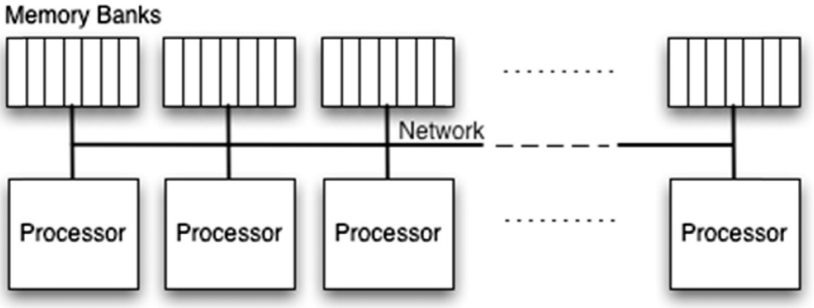
Σχήμα 2.1: Ταξινόμηση κατά Flynn [1]

(clusters). Αυτές οι τρεις οικογένειες πολυεπεξεργαστικών συστημάτων θα αναλυθούν στις επόμενες υποενότητες.

2.3.1 Πολυεπεξεργαστές μοιραζόμενης μνήμης

Ένας πολυεπεξεργαστής μοιραζόμενης μνήμης αποτελεί μια αρχιτεκτονική που απαρτίζεται από σχετικά μικρό αριθμό επεξεργαστών οι οποίοι έχουν άμεση (μέσω υλικού) πρόσβαση σε όλη την κύρια μνήμη του συστήματος, δηλαδή ακολουθούν την αρχιτεκτονική «shared-everything». Αυτό δίνει τη δυνατότητα στον κάθε επεξεργαστή να έχει πρόσβαση σε δεδομένα που έχουν δημιουργηθεί από οποιονδήποτε άλλο επεξεργαστή. Το κλειδί σε αυτού του είδους την αρχιτεκτονική με πολυεπεξεργαστές είναι το δίκτυο διασύνδεσης που ενώνει τους επεξεργαστές με

όλες τις διαθέσιμες μνήμες. Οι πολυεπεξεργαστές μοιραζόμενης μνήμης μπορούν να διαχωριστούν σε δύο κατηγορίες ανάλογα με τον σχετικό χρόνο προσπέλασης των επεξεργαστών στις επιμέρους μνήμες: ομοιόμορφη πρόσβαση στη μνήμη (Uniform Memory Access - UMA) και ανομοιόμορφη πρόσβαση στη μνήμη (Non-Uniform Memory Access - NUMA).



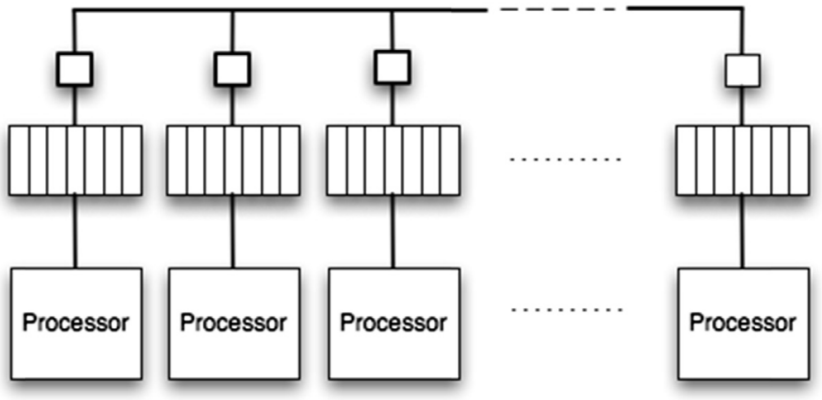
Σχήμα 2.2: Αρχιτεκτονική πολυεπεξεργαστών μοιραζόμενης μνήμης με ομοιόμορφη πρόσβαση

Τα συστήματα με ομοιόμορφη πρόσβαση μνήμης (σχήμα 2.2), αποτελούν μια αρχιτεκτονική στην οποία απαιτείται όμοιος χρόνος για την πρόσβαση οποιασδήποτε μνήμης από έναν επεξεργαστή. Τέτοια συστήματα συχνά αποκαλούνται συμμετρικοί πολυεπεξεργαστές (Symmetric Multiprocessors - SMP). Οι συμμετρικοί πολυεπεξεργαστές διαχειρίζονται από ένα μοναδικό λειτουργικό σύστημα και ένα κεντρικό δίαυλο επικοινωνίας που συνδέει τους επεξεργαστές με τις μνήμες. Οι χρόνοι προσπέλασης των δεδομένων ίσως να διαφέρουν μερικές φορές, το οποίο οφείλεται στις διαμάχες για πρόσβαση στην ίδια μνήμη (contention), αλλά όλοι οι επεξεργαστές θεωρούνται ισάξιοι και έχουν τις ίδιες πιθανότητες προσπέλασης.

Τα συστήματα με ανομοιόμορφη πρόσβαση μνήμης (σχήμα 2.3), ενώ εξακολουθεί να αποτελεί μια «shared-everything» αρχιτεκτονική, δεν προσφέρουν ισάξια πρόσβαση των επεξεργαστών στη συνολικά διαθέσιμη μνήμη του συστήματος. Η ονομασία τους οφείλεται στα ανομοιόμορφα χρονικά διαστήματα που απαιτούνται για την προσπέλαση των κοντινότερων τμημάτων φυσικής μνήμης συγκριτικά με τα πιο απομακρυσμένα. Οι NUMA αρχιτεκτονικές κλιμακώνουν πιο εύκολα, επιτρέποντας την ενσωμάτωση μεγαλύτερου αριθμού επεξεργαστών σε σχέση με τα συστήματα SMP. Ωστόσο, λόγω αυτής της ανομοιομορφίας στους χρόνους προσπέλασης μνήμης, ο προγραμματισμός σε αυτά τα συστήματα γίνεται πιο περίπλοκος καθώς θα πρέπει να λαμβάνεται σε μεγάλο βαθμό υπόψη η τοπικότητα δεδομένων (locality) έτσι ώστε να επιτυγχάνονται οι καλύτερες δυνατές επιδόσεις.

2.3.2 Μαζικά παράλληλοι επεξεργαστές

Οι αρχιτεκτονικές που βασίζονται σε μαζικά παράλληλους επεξεργαστές (massively parallel processors - MPP) είναι αυτές που μπορούν να κλιμακώσουν σε

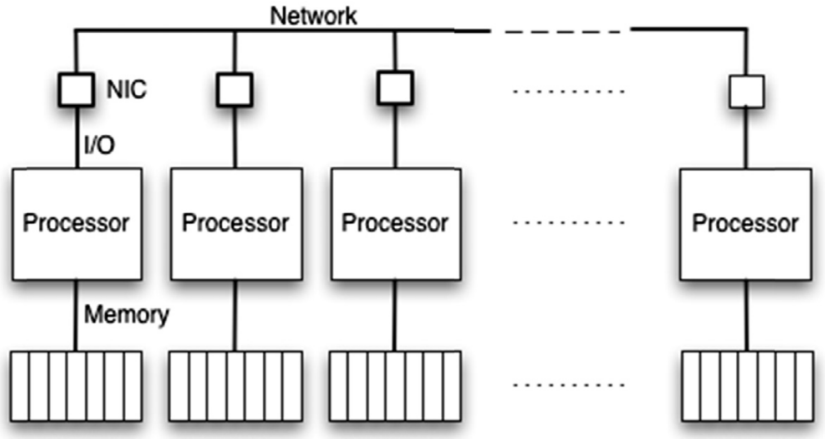


Σχήμα 2.3: Αρχιτεκτονική πολυεπεξεργαστών μοιραζόμενης μνήμης με ανομοιόμορφη πρόσβαση

εκατοντάδες ή και χιλιάδες επεξεργαστές με τρομερές επιδόσεις. Σε αντίθεση με τα συστήματα μοιραζόμενης μνήμης, οι μαζικά παράλληλοι επεξεργαστές βασίζονται στο μοντέλο «shared-nothing» με χρήση κατανεμημένης μνήμης. Οι μεγαλύτεροι και ταχύτεροι υπερυπολογιστές σήμερα, οι οποίοι αποτελούνται από εκατομμύρια πυρήνες επεξεργασίας, ανήκουν σε αυτήν την κατηγορία πολυεπεξεργαστικών συστημάτων. Ένα MPP, αποτελείται από ξεχωριστές ομάδες επεξεργαστών οι οποίες είναι απευθείας συνδεδεμένες μόνο με τη δικιά τους φυσική μνήμη. Αυτή η λογική απλοποιεί τη σχεδίαση και εξαλείφει τους παράγοντες που εμποδίζουν την κλιμακωσιμότητα. Με την έλλειψη κοινής μνήμης, θα πρέπει να χρησιμοποιηθεί διαφορετικός μηχανισμός ώστε να επιτραπεί η επικοινωνία και η οργάνωση των επεξεργαστών διαφορετικών ομάδων (σχήμα 2.4). Οι ομάδες επεξεργαστών συνδέονται με ένα δίκτυο περιοχής αποθήκευσης (Storage Area Network - SAN) μέσω κάρτας δικτύου (Network Interface Controller - NIC) και χρησιμοποιούν μεθόδους ανταλλαγής μηνυμάτων, όπως είναι το MPI (Message Passing Interface) για τη συνεργασία τους.

2.3.3 Συστοιχίες υπολογιστών

Ενώ όλοι οι υπερυπολογιστές σήμερα επωφελούνται από τη χρήση μικροεπεξεργαστών VLSI και μνήμης DRAM τα οποία παράγονται μαζικά στο εμπόριο, τα προηγούμενα συστήματα βασίζονται σε σχεδιασμό ειδικού σκοπού όπου υπάρχει στενή συσχέτιση (tight coupling) μεταξύ των επεξεργαστών για μέγιστες επιδόσεις. Ωστόσο, η κυρίαρχη κλάση χρησιμοποιούμενων υπερυπολογιστών είναι οι συστοιχίες υπολογιστών (clusters ή αλλιώς commodity clusters), οι οποίοι εκμεταλλεύονται ακόμη περισσότερο την οικονομία των κοινών «εξαρτημάτων» (components). Commodity cluster ορίζεται μια συστοιχία υπολογιστών όπου τα επιμέρους δίκτυα και υπολογιστικοί κόμβοι είναι προϊόντα μαζικής παραγωγής τα



Σχήμα 2.4: Αρχιτεκτονική μαζικά παράλληλων επεξεργαστών

οποία οργανισμοί προμηθεύονται και χρησιμοποιούν με τρόπο διαφορετικό από τον αρχικό σκοπό του κατασκευαστή [7].

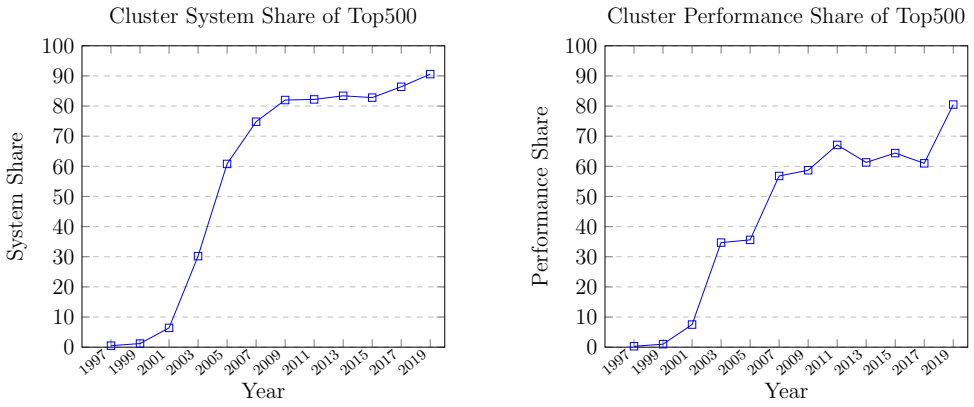
Από τη δεκαετία του ’90 παρατηρήθηκαν τα οφέλη των commodity clusters όσον αφορά την επίδοση σε σχέση με το κόστος. Το 1997, το Network Of Workstations (NOW) cluster [8], φτιαγμένο από εμπορικούς σταθμούς εργασίας (workstations), ήταν η πρώτη συστοιχία υπολογιστών που κατάφερε να κάνει την εμφάνισή της στη λίστα του Top500 [9], ενώ το Beowulf cluster [10], φτιαγμένο από προσωπικούς υπολογιστές, ήταν το πρώτο που τιμήθηκε με το βραβείο Gordon Bell Prize [1].

2.4 Μετάβαση σε συστοιχίες υπολογιστών

Σήμερα, οι γρηγορότεροι υπερυπολογιστές του κόσμου απαρτίζονται από MPPs και clusters. Όπως φαίνεται στο σχήμα 2.5 η χρήση των clusters παρουσίασε εκθετική άνοδο την περίοδο 2001-2007 και η επίδοσή τους συνεχίζει να βελτιώνεται τα τελευταία χρόνια [9]. Εξαιτίας της ειδικής σχεδίασής τους για high performance computing, τα MPPs παραμένουν στην κορυφή των υπερυπολογιστών όσον αφορά την επίδοση.

Μερικοί από τους λόγους για τους οποίους οι συστοιχίες υπολογιστών (commodity clusters) προτιμώνται έναντι των υπερυπολογιστών ειδικής κατασκευής είναι οι ακόλουθοι [11]:

* Οι επιμέρους σταθμοί εργασίας γίνονται ολοένα και πιο ισχυροί.
* Το εύρος ζώνης (bandwidth) των δικτύων διασύνδεσης μεταξύ των σταθμών εργασίας αυξάνεται συνεχώς, καθώς νέες τεχνολογίες και πρωτόκολλα υλοποιούνται στα περιβάλλοντα τοπικών δικτύων.
* Η ενσωμάτωση των συστοιχιών υπολογιστών σε υπάρχοντα δίκτυα είναι πιο εύκολη σε σχέση με ειδικά σχεδιασμένους υπερυπολογιστές.



Σχήμα 2.5: Ποσοστά χρήσης των clusters στη λίστα Top500

* Οι συστοιχίες υπολογιστών προσφέρουν εύκολη επεκτασιμότητα. Οι δυνατότητες των επιμέρους κόμβων μπορούν να βελτιωθούν εύκολα, προσθέτοντας επιπλέον μνήμη ή επεξεργαστές.

* Οι συστοιχίες υπολογιστών είναι μια φθηνή και έτοιμη προς χρήση εναλλακτική των εξειδικευμένων πλατφορμών μεγάλης υπολογιστικής επίδοσης (High Performance Computing - HPC).

* Η ανάπτυξη λογισμικού για τους απλούς σταθμούς εργασίας είναι πιο ώριμη συγκριτικά με τις κλειστές (proprietary) λύσεις των παράλληλων υπολογιστών κάτι το οποίο οφείλεται κυρίως στη μη-προτυποποιημένη φύση των παράλληλων συστημάτων.

2.5 Εφαρμογές

Η παγκόσμια αγορά των HPC συστημάτων και κατ΄ επέκταση των υπερυπολογιστών παρουσιάζει ραγδαία ανάπτυξη. Τα συστήματα αυτά χρησιμοποιούνται κυρίως από τη βιομηχανία, τις ακαδημαϊκές κοινότητες και τις κυβερνήσεις. Αναλυτικότερα, μερικές από τις σημαντικότερες περιοχές εφαρμογής των ανωτέρω συστημάτων είναι οι ακόλουθες [12]:

– **Βιοεπιστήμες**: Ανακάλυψη φαρμάκων, ανίχνευση και πρόληψη ασθενειών

– **Χημική μηχανική**: Επεξεργασία και μοριακός σχεδιασμός

– **Οικονομία**: Ανάλυση ρίσκου, αυτόματες αγοραπωλησίες μετοχών

– **Ηλεκτρονική**: Σχεδιασμός και επαλήθευση ηλεκτρονικών στοιχείων

– **Γεωεπιστήμες**: Εξερεύνηση πετρελαίου και μοντελοποίηση δεξαμενών

- **Άμυνα και Ενέργεια**: Διαχείριση πυρηνικής ενέργειας και έρευνα
- **Πανεπιστήμια**: Βασική και εφαρμοσμένη έρευνα
- **Μετεωρολογία**: Βραχυπρόθεσμη πρόγνωση καιρού, μοντελοποίηση κλίματος
- **Κυβερνητικά εργαστήρια**: Βασική και εφαρμοσμένη έρευνα
- **Μηχανολογικός σχεδιασμός**: Δισδιάστατη και τρισδιάστατη σχεδίαση, επαλήθευση και μοντελοποίηση

Κεφάλαιο 3

Διαχείριση Πόρων

Η απόκτηση και εγκατάσταση ενός υπερυπολογιστή συχνά αντιπροσωπεύει μια σημαντική οικονομική επένδυση από το ίδρυμα στο οποίο θα ανήκει. Ωστόσο, οι δαπάνες δεν σταματούν μόλις αποκτηθεί το υλικό και ολοκληρωθεί η εγκατάστασή του. Το ίδρυμα θα πρέπει να χρησιμοποιεί ειδικούς διαχειριστές συστήματος, να πληρώνει για υποστήριξη και να καλύπτει το ποσό της ενέργειας που δαπανάται για τη λειτουργία και ψύξη των μηχανημάτων. Όλα αυτά, συνολικά, αναφέρονται ως «κόστος ιδιοκτησίας». Για μεγάλες εγκαταστάσεις, το κόστος είναι υπέρογκο καθώς ένα megawatt ενέργειας κοστίζει περίπου 1 εκατομμύριο δολάρια ανά χρόνο στις Ηνωμένες Πολιτείες της Αμερικής. Για το λόγο αυτό, δεν είναι τυχαίο που οι ιδιοκτήτες υπερυπολογιστών ερευνούν με μεγάλη προσοχή πώς χρησιμοποιούνται οι πόροι του συστήματος και με ποιο τρόπο η χρήση τους μπορεί να μεγιστοποιηθεί.

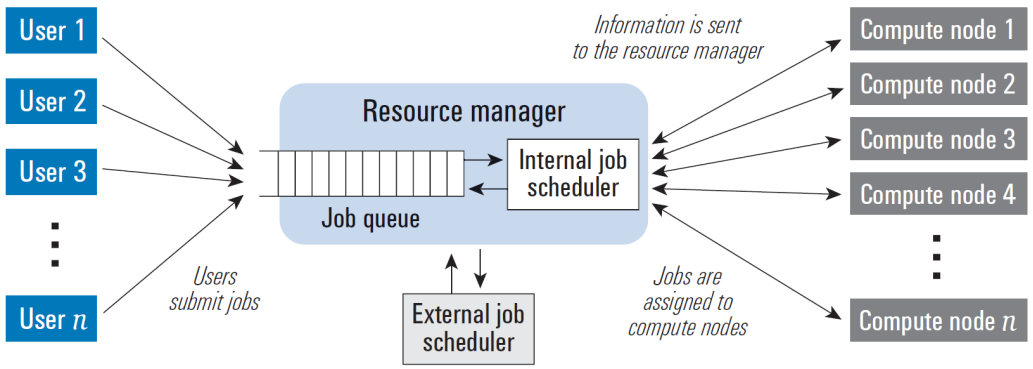
3.1 Συστήματα Διαχείρισης Πόρων

Λύση στις παραπάνω ανησυχίες έρχονται να δώσουν τα συστήματα διαχείρισης πόρων (resource management systems) καθώς παίζουν κομβικό ρόλο για τον τρόπο ανάθεσης πόρων ενός υπερυπολογιστή σε εφαρμογές χρηστών. Ένα σύστημα διαχείρισης πόρων δε βοηθάει απλά στην κατανομή διαφορετικών φορτίων εργασίας και διάρκειας, αλλά παρέχει επίσης κοινές διεπαφές (interfaces) μεταξύ διαφορετικών τύπων μηχανημάτων και παραμετροποιήσεών τους, απλοποιώντας την πρόσβαση σε αυτά και ανακουφίζοντας από ανησυχίες για προβλήματα φορητότητας. Προσφέρει μηχανισμούς με τους οποίους υπολογιστικά συστήματα γίνονται προσβάσιμα σε διάφορες κατηγορίες χρηστών (συμπεριλαμαβανομένου των χρηστών που δεν ανήκουν στο ίδρυμα που φιλοξενεί το cluster, μέσω ειδικού περιβάλλοντος συνεργασίας όπως είναι το National Science Foundation XSEDE [13]) με ακριβή καταγραφή χρήσης πόρων και χρέωσής της κατάλληλα.

Τα συστήματα διαχείρισης πόρων αποτελούν αναπόσπαστο κομμάτι του λογισμικού των HPC clusters. Μία υψηλού επιπέδου αναπαράσταση ενός τέτοιου συστήματος παρουσιάζεται στο σχήμα 3.1. Πραγματοποιούν τρεις βασικές

λειτουργίες: κατανομή πόρων, χρονοδρομολόγηση εργασιών και παρακολούθηση της εκτέλεσής τους (monitoring). Η κατανομή πόρων αναλαμβάνει την ανάθεση φυσικού υλικού (hardware), το οποίο μπορεί να είναι ένα μικρό κομμάτι του μηχανήματος ή και όλο το σύστημα, σε εργασίες χρηστών ανάλογα με τις ανάγκες τους.

Στις επόμενες υποενότητες, αναλύονται τα διαφορετικά ήδη πόρων που αναγνωρίζονται συνήθως από ένα τυπικό σύστημα διαχείρισης πόρων [1].



Σχήμα 3.1: Αφαιρετική αναπαράσταση ενός συστήματος διαχείρισης πόρων [2]

3.1.1 Υπολογιστικοί κόμβοι

Η αύξηση του αριθμού των υπολογιστικών κόμβων που είναι ανατεθειμένοι σε ένα «embarassingly parallel» πρόγραμμα είναι ο πιο απλός τρόπος για να κλιμακώσει το μέγεθος των δεδομένων εισόδου ή να μειωθεί ο χρόνος εκτέλεσης ενός σταθερού φορτίου. Ο αριθμός των κόμβων είναι επομένως μία από τις πιο σημαντικές παραμέτρους που ζητούν οι χρήστες για να εκτελέσουν ένα παράλληλο πρόγραμμα. Οι κόμβοι διαφοροποιούνται μεταξύ τους σε χαρακτηριστικά όπως το μέγεθος της μνήμης, είδη πυρήνων επεξεργασίας (CPU) και συχνότητας, διαθέσιμες διασυνδέσεις, τοπικοί χώροι αποθήκευσης κ.α. Σωστά διαμορφωμένα συστήματα διαχείρισης πόρων επιτρέπουν την επιλογή των πιο κατάλληλων κόμβων που καλύπτουν τις ανάγκες της εκάστοτε εργασίας, αποφεύγοντας την ανάθεση πόρων που πιθανόν να μη χρειαστούν τελικά για την εκτέλεσή της.

3.1.2 Πυρήνες επεξεργασίας

Οι νεότεροι κόμβοι υπερυπολογιστών έχουν ένα ή και περισσότερα πολυπύρηνα socket επεξεργασίας, προσφέροντας τοπική παραλληλοποίηση προγραμμάτων που το υποστηρίζουν μέσω πολυνηματισμού (multithreading) ή χρησιμοποιώντας

πολλαπλές παράλληλες διεργασίες του ενός νήματος. Γι΄ αυτό το λόγο, τα συστήματα διαχείρισης πόρων προσφέρουν την επιλογή προσδιορισμού κοινής (*shared*) ή αποκλειστικής (*exclusive*) ανάθεσης κόμβων σε εργασίες. Η κοινή χρήση κόμβων είναι χρήσιμη για περιπτώσεις στις οποίες μια εργασία δεν χρησιμοποιεί εξ ολοκλήρου τους πόρους που της έχουν ανατεθεί. Μέσω του (coscheduling), δηλαδή βάζοντας διαφορετικές εργασίες να εκτελεστούν στους ίδιους πόρους, μπορεί να επιτευχθεί καλύτερη αξιοποίηση των διαθέσιμων υπολογιστικών πόρων. Ωστόσο, αυτό οδηγεί τα προγράμματα που τρέχουν στον ίδιο κόμβο να μοιράζονται, επιπλέον, και τα υπόλοιπα φυσικά μέρη του συγκεκριμένου κόμβου όπως είναι η μνήμη, το δίκτυο, οι δίαυλοι εισόδου/εξόδου κάτι που μπορεί να οδηγήσει σε ανταγωνισμό των εργασιών για τους κοινούς πόρους. Στην περίπτωση που μας ενδιαφέρει η απόδοση των προγραμμάτων τότε είναι λογικό να προτιμάται η ανάθεση κόμβων σε exclusive mode ώστε να ελαχιστοποιούνται οι καθυστερήσεις που αναπόφευκτα δημιουργούνται λόγω του φαινομένου αυτού.

3.1.3 Διασυνδέσεις

Πολλά συστήματα είναι φτιαγμένα να λειτουργούν μόνο με έναν τύπο δικτύου, άλλα ωστόσο υποστηρίζουν διαφόρων ειδών διασυνδέσεων και έχουν φτιαχτεί έτσι ώστε να εκμεταλλεύονται σύγχρονες τεχνολογίες διασύνδεσης όπως είναι το GigE και το InfiniBand συνδυαστικά. Η επιλογή της κατάλληλης σύνδεσης εξαρτάται από τα χαρακτηριστικά και τις ανάγκες της εκάστοτε εφαρμογής. Για παράδειγμα, το πρόγραμμα έχει ανάγκη από χαμηλές τιμές καθυστέρησης επικοινωνίας (latency) ή χρειάζεται όσο το δυνατόν περισσότερο εύρος επικοινωνίας (bandwidth); Συχνά, η απάντηση στο παραπάνω ερώτημα είναι στενά συνδεδεμένη με την βιβλιοθήκη επικοινωνίας που χρησιμοποιεί η εφαρμογή που εκτελείται. Για παράδειγμα, συχνά παρατηρείται εγκατάσταση του message-passing interface (MPI) με διαφορετικές βιβλιοθήκες για Ethernet και InfiniBand αν είναι διαθέσιμα και τα δύο είδη δικτύων. Η επιλογή «λάθους» τύπου δικτύου πιθανά θα στοιχίσει στην απόδοση της εκτέλεσης.

3.1.4 Μέσα αποθήκευσης

Πληθώρα clusters βασίζεται στη χρήση κοινών συστημάτων αρχείων (file system) τα οποία είναι προσβάσιμα από όλους τους κόμβους του συστήματος. Αυτό είναι βολικό, καθώς ένα πρόγραμμα που μεταγλωττίζεται στον κύριο κόμβο και στη συνέχεια αποθηκευτεί σε ένα κοινό σύστημα αρχείων, τότε γίνεται άμεσα διαθέσιμο σε όλους τους υπολογιστικούς κόμβους. Επίσης, μπορεί να χρησιμοποιηθεί κοινό σύνολο δεδομένων (dataset) το οποίο μπορεί να μεταβάλλεται κατά την εκτέλεση μιας εφαρμογής και να αναγνωρίζουν την αλλαγή όλοι οι κόμβοι ταυτόχρονα. Ωστόσο, ενδέχεται τα συστήματα αρχείων που χρησιμοποιούνται από ένα cluster να μην προσφέρουν το απαραίτητο εύρος ζώνης το οποίο θα είναι κλιμακώσιμο με τον αριθμό των μηχανημάτων ή τις ταυτόχρονες προσβάσεις από χρήστες. Για

προγράμματα που εκτελούν μεγάλο αριθμό I/O θα ήταν προτιμώτερη η χρήση τοπικών δίσκων στον εκάστοτε κόμβο. Το μειονέκτημα σε αυτή την πρακτική είναι ότι το σύνολο δεδομένων που παράγεται από τις εφαρμογές, θα πρέπει να μεταφέρεται στο κύριο σύστημα αρχείων κατά τον τερματισμό του στην περίπτωση που απαιτείται περαιτέρω ανάλυση ή επεξεργασία αυτών.

3.1.5 Επιταχυντές

Ετερογενείς αρχιτεκτονικές που χρησιμοποιούν επιταχυντές (accelerators), για παράδειγμα κάρτες γραφικών (GPUs) ή many integrated cores (MICs), σε συνδυασμό με τις κύριες μονάδες επεξεργασίας (CPUs) αποτελούν μία συνήθη επιλογή για την αύξηση της συνολικής υπολογιστικής απόδοσης με ταυτόχρονη μείωση της ενέργειας που καταναλώνεται. Όμως, η ύπαρξη επιταχυντών σε ένα cluster περιπλέκει την δουλειά του συστήματος διαχείρισης πόρων καθώς πλέον μερικοί κόμβοι μπορεί να έχουν επιταχυντές ενός τύπου, κάποιοι άλλοι ενός διαφορετικού τύπου και κόμβοι οι οποίοι δεν έχουν καθόλου. Μοντέρνα συστήματα διαχείρισης πόρων επιτρέπουν στους χρήστες να ορίζουν παραμέτρους στις εργασίες που υποβάλουν που να υποδεικνύουν ποιο είδος κόμβου να είναι το καταλληλότερο για την εκτέλεση. Προγράμματα τα οποία δεν είναι γραμμένα για την αξιοποίηση επιταχυντών θα πρέπει να περιορίζονται σε απλούς κόμβους έτσι ώστε να επιτυγχάνεται καλύτερη αξιοποίηση των υπολογιστικών πόρων του συστήματος σε βάθος χρόνου.

3.2 Εργασίες (Jobs)

Τα συστήματα διαχείρισης πόρων κατανέμουν τους διαθέσιμους υπολογιστικούς πόρους σε *εργασίες* (*jobs*) που έχουν ορίσει χρήστες. Μια εργασία αποτελεί ένα αυτόνομο μερίδιο δουλειάς το οποίο σχετίζεται με μία είσοδο και κατά την εκτέλεσή της παράγει μία έξοδο. Η έξοδος μπορεί να είναι τόσο μικρή όσο μια γραμμή εμφανιζόμενη στην κονσόλα, ένα σύνολο δεδομένων πολλών terabyte αποθηκευμένο σε πολλαπλά αρχεία ή μία ροή δεδομένων η οποία θα μεταδίδεται μέσω ενός δικτύου ευρείας ζώνης σε ένα άλλο μηχάνημα. Μία εργασία μπορεί να διαιρείται σε μικρότερα κομμάτια, τα tasks. Συνήθως, κάθε task σχετίζεται με την εκτέλεση ενός συγκεκριμένου προγράμματος. Σε γενικές γραμμές, δεν είναι απαραίτητο τα επιμέρους tasks μιας εργασίας να έχουν κοινά χαρακτηριστικά όσον αφορά χρησιμοποιούμενους πόρους και διάρκεια εκτέλεσης.

Οι εργασίες μπορούν να εκτελούνται διαδραστικά (*interactively*), κατά την οποία είναι δυνατή η συμμετοχή του χρήστη μέσω της κονσόλας, ενδεχομένως για την παροχή επιπλέον εισόδου στην εφαρμογή κατά την εκτέλεση, ή να επεξεργάζονται κατά δεσμίδες (*batch processing*) όπου όλη η απαραίτητη πληροφορία για την ολοκλήρωση της εργασίας είναι διαθέσιμη κατά την υποβολή της. Ο δεύτερος τρόπος, (batch processing), παρέχει μεγαλύτερη ευελιξία στα συστήματα διαχείρισης πόρων, καθώς μπορεί να αποφασίσει πότε είναι η ιδανική στιγμή να ξεκινήσει η κάθε εργασία σύμφωνα με την κατάσταση του cluster. Στη λήψη αυτής της απόφασης

σημαντικό ρόλο παίζει ο χρονοδρομολογητής που χρησιμοποιεί το σύστημα διαχείρισης πόρων (σχήμα 3.1), ο οποίος μπορεί να είναι είτε ενσωματωμένος στον resource manager είτε εξωτερικός για μεγαλύτερη ευελιξία.

3.3 Χρονοδρομολόγηση

Εργασίες που περιμένουν να εκτελεστούν τοποθετούνται σε *ουρές* (*queues*). Η ουρά καθορίζει την σειρά με την οποία επιλέγονται οι εργασίες από το σύστημα διαχείρισης πόρων για την εκκίνησή τους. Όπως υπονοείται απο τον ορισμό της «ουράς» στην επιστήμη υπολογιστών, πολλές φορές ο τρόπος επιλογής είναι «First In First Out - FIFO», αν και η χρήση καλών χρονοδρομολογητών προσφέρει μεθόδους για την καλύτερη επιλογή εργασιών με στόχο την μέγιστη αξιοποίηση των υπολογιστικών πόρων. Τα περισσότερα συστήματα χρησιμοποιούν πολλαπλές ουρές εργασιών, με την καθεμία να έχει ένα συγκεκριμένο σκοπό και ένα σύνολο περιορισμών χρονοδρομολόγησης. Για παράδειγμα, μπορεί να υπάρχει μία ουρά συγκεκριμένα για διαδραστικές εργασίες, ενώ άλλες με συγκεκριμένο όριο σε χρόνο εκτέλεσης, μνήμης ή αριθμό κόμβων.

Οι διαχειριστές του συστήματος, όταν επιλέγουν έναν εξωτερικό χρονοδρομολογητή, στην περίπτωση που δεν τους καλύπτει αυτός που πιθανά να είναι ενσωματωμένος στο σύστημα διαχείρισης πόρων, θα πρέπει να λαμβάνουν υπόψη διαφορετικούς παράγοντες για την αξιολόγησή τους. Τα βασικά χαρακτηριστικά ενός καλού χρονοδρομολογητή παρουσιάζονται εν συντομία παρακάτω [2]:

- *Ευρύ πεδίο*: υπάρχουν διάφορα είδη εργασιών που υποβάλλονται σε ένα σύστημα, οπότε ο χρονοδρομολογητής θα πρέπει να υποστηρίζει εξίσου αποδοτικά εργασίες τύπου batch, παράλληλες, σειριακές, κατανεμημένες, διαδραστικές και μη.

- *Κλιμακωσιμότητα*: ο χρονοδρομολογητής θα πρέπει να είναι σε θέση να κλιμακώνει σε κόμβους της τάξης των χιλιάδων και να διαχειρίζεται χιλιάδες εργασίες ταυτόχρονα.

- *Υποστήριξη πληθώρας αλγορίθμων*: ο χρονοδρομολογητής οφείλει να υποστηρίζει αρκετούς αλγορίθμους χρονοδρομολόγησης και να έχει τη δυνατότητα εφαρμογής διαφορετικού ανά ουρά εργασιών.

- *Δυναμικότητα*: η προσθήκη ή αφαίρεση υπολογιστικών πόρων σε μια εργασία κατά την εκτέλεσή της θα πρέπει να υποστηρίζεται από τον χρονοδρομολογητή.

- *Δικαιοσύνη*: υπό οποιεσδήποτε συνθήκες, ο χρονοδρομολογητής θα πρέπει να κατανέμει δίκαια τους πόρους του συστήματος (fair-share).

- *Αποδοτικότητα*: στην περίπτωση που χρησιμοποιείται ένας περίπλοκος αλγόριθμος χρονοδρομολόγησης είναι λογικό να υπάρχει ένα overhead.

Ωστόσο, αυτό θα πρέπει να είναι μηδαμινό ή τουλάχιστον να καλύπτεται από την εξοικονόμηση χρόνο λόγω της χρήσης του.

- *Δυνατότητα ενσωμάτωσης*: ο χρονοδρομολογητής θα πρέπει να συνδυάζεται αποτελεσματικά με τυπικά συστήματα διαχείρισης πόρων προσφέροντας τους κατάλληλες προγραμματιστικές διεπαφές (interfaces) για την χρήση του.
- *Ευαισθησία αρχιτεκτονικής*: ανάλογα με τη συνδεσμολογία του cluster και τα χαρακτηριστικά των κόμβων του, ο χρονοδρομολογητής θα πρέπει να χρησιμοποιήσει το κατάλληλο σύνολο πόρων που ταιριάζουν περισσότερο στις απαιτήσεις της εκάστοτε δουλειάς (παραλληλοποίηση, χρήση επιταχυντών κ.ο.κ.).
- *Υποστήριξη προεκχώρισης*: ο χρονοδρομολογητής θα πρέπει να έχει λειτουργίες που να του επιτρέπουν την προεκχώριση εργασιών (job preemption), δηλαδή την παύση μιας εκτελούμενης εργασίας για την εκτέλεση μιας άλλης.

3.4 Κατηγοριοποίηση

Βάσει του τρόπου με τον οποίο χρονοδρομολογούνται οι εργασίες, τα συστήματα διαχείρισης πόρων μπορούν να διαχωριστούν σε δύο μεγάλες κατηγορίες [4]: *queuing* και *planning* συστήματα. Συγκερκριμένα, το κριτήριο διαφοροποίησης ανάμεσα στις δύο κατηγορίες είναι το χρονικό πλαίσιο στο οποίο ορίζεται η χρονοδρομολόγηση. Τα queuing συστήματα προσπαθούν να χρησιμοποιήσουν τους υπολογιστικούς πόρους που είναι διαθέσιμοι εκείνη τη δεδομένη στιγμή με εργασίες οι οποίες περιμένουν στην ουρά. Δηλαδή τους ενδιαφέρει αποκλειστικά το «παρόν» και δεν δημιουργούν μελλοντικό πλάνο κατανομής πόρων για όλες τις εργασίες της ουράς. Έτσι, οι εργασίες που αναμένουν, δεν έχουν εκτιμώμενη ώρα εκκίνησης. Αντίθετα, τα planning συστήματα νοιάζονται τόσο για το «παρόν» όσο και για το «μέλλον» καθώς εκτιμώμενοι χρόνοι εκκίνησης υπολογίζονται για όλες τις εργασίες. Δημιουργείται, λοιπόν, ένα ολοκληρωμένο πρόγραμμα με την μελλοντική χρήση πόρων του συστήματος το οποίο είναι διαθέσιμο και στους χρήστες. Στον πίνακα 3.1 συγκρίνονται συνοπτικά οι δύο κατηγορίες.

3.5 Επιλογή Διαχειριστή Πόρων (Resource Manager)

Έχουν αναπτυχθεί και είναι διαθέσιμα πολλά διαφορετικά συστήματα διαχείρισης πόρων. Μερικά από αυτά αναφέρονται παρακάτω:

- *Slurm Workload Manager* [14] είναι ένα ευρέως διαδεδομένο πακέτο ανοιχτού κώδικα.

	Queuing	**Planning**
χρονικό πλαίσιο σχεδιασμού	παρόν	παρόν και μέλλον
υποβολή νέας εργασίας	προσθήκη σε ουρά	επανασχεδιασμός
εκτίμηση χρόνου εκκίνησης	καθόλου	σε όλες τις εργασίες
εκτίμηση διάρκειας εργασίας	όχι απαραίτητα	υποχρεωτικά
κρατήσεις πόρων	χωρίς δυνατότητα	υποστηρίζεται
backfilling	προαιρετικά	υποστηρίζεται

Πίνακας 3.1: Διαφορές μεταξύ Queuing και Planning συστημάτων [4]

- *Portable Batch System (PBS)* [15] ήταν αρχικά διαθέσιμος ως ιδιωτικός κώδικας (proprietary) αλλά και ως υλοποιήσεις με συμβατές διεπαφές και εντολές.
- *Moab Cluster Suite* [16], βασισμένο στον ανοιχτού κώδικα Maui Custer Scheduler, πρόκειται για έναν επίσης proprietary resource manager προσφερόμενο από την Adaptive Computing Inc.
- *IBM Tivoli Workload Scheduler* [17] είναι προιόν της IBM που αρχικά προοριζόταν αποκλειστικά για συστήματα με λειτουργικό σύστημα AIX αλλά πλέον υποστηρίζει POWER και x86-based Linux πλατφόρμες.
- *Univa Grid Engine* [18] είναι ένα batch-queuing σύστημα που χρησιμοποιεί τεχνολογίες που αρχικά είχαν υλοποιηθεί απο τις Sun Microsystems και Oracle.
- *Hadoop Yet Another Resource Negotiator (YARN)* [19] πρόκειται για έναν resource manager ειδικά σχεδιασμένο για MapReduce εφαρμογές.

Το Slurm Workload Manager, όντας ίσως το πιο διαδεδομένο, ανοιχτού κώδικα και ενεργά αναπτυσσόμενο σύστημα διαχείρισης πόρων, επιλέχθηκε να αναλυθεί στο επόμενο κεφάλαιο.

Κεφάλαιο 4

Slurm Workload Manager

4.1 Εισαγωγή

Το Slurm είναι ένα ανοιχτού κώδικα, παραμετροποιήσιμο, επεκτάσιμο, και κλιμακώσιμο σύστημα διαχείρισης πόρων (resource management system) κατάλληλο για χρήση σε μεγάλα αλλά και μικρά συστήματα μεγάλης κλίμακας (clusters) βασισμένα σε Linux ή Unix-compatible λειτουργικά συστήματα. Η προέλευσή του χρονολογείται το 2001, όταν μια μικρή ομάδα προγραμματιστών με πρωτοπόρο τον Morris Jette στο Lawrence Livermore National Laboratory ξεκίνησε να ασχολείται με την έρευνα πάνω σε προηγμένα συστήματα χρονοδρομολόγησης για συστήματα μεγάλης κλίμακας. Από τότε, η ανάπτυξη του Slurm έχει προχωρήσει εντυπωσιακά, αφού η ομάδα πλέον αποτελείται από περίπου 200 συνεργάτες καθώς επίσης και πολλαπλά ιδρύματα, συμπεριλαμβανομένου του SchedMD LLC (η κύρια εταιρία που είναι υπεύθυνη για την ανάπτυξη, υποστήριξη και εκπαίδευση πάνω στο Slurm), Linux NetworX, HewlettPackard, Groupe Bull, Cray, Barcelona Supercomputing Center, Oak Ridge National Laboratory, Los Alamos National Laboratory, Intel, Nvidia και πολλών ακόμα. Τον Ιούνιο του 2014 το Slurm βρισκόταν ανάμεσα στα πιο διαδεδομένα συστήματα διαχείρισης πόρων, χρησιμοποιούμενο από περίπου το 60% των μηχανημάτων στην λίστα Top 500 [9].
Τα γενικά χαρακτηριστικά στα οποία θεμελιώθηκε το Slurm είναι τα ακόλουθα:

- *Απλότητα*: Το Slurm είναι αρκετά απλό ώστε να επιτρέπει στους χρήστες του να κατανοούν την υλοποίησή του διαβάζοντας τον κώδικα και να τον επεκτείνουν.
- *Λογισμικό ανοιχτού κώδικα*: Το Slurm είναι και θα παραμείνει ελεύθερο και προσβάσιμο στο ευρύ κοινό.
- *Φορητότητα*: Το Slurm είναι γραμμένο στη γλώσσα C και παραμετροποιήσιμο μέσω του GNU *autoconf*. Ενώ έχει γραφτεί συγκεκριμένα για το Linux, δεν θα είναι δύσκολο να υποστηριχθεί και σε άλλα Unix-like λειτουργικά συστήματα. Επίσης διαθέτει ένα γενικό μηχανισμό για την ανάπτυξη και χρήση διαφόρων επεκτάσεων (plugins) κάτι το οποίο επιτρέπει την υποστήριξη ευρείας ποικιλίας

συστημάτων. Ένα configuration αρχείο του Slurm υποδεικνύει ποια plugins θα χρησιμοποιηθούν.

– *Ανεξαρτησία διασύνδεσης*: Το Slurm υποστηρίζει επικοινωνία βασισμένη στα πρωτόκολλα UDP και IP, καθώς και τη διασύνδεση Quadrics Elan3. Ωστόσο, η προσθήκη κι άλλων πιο σύνθετων διασυνδέσεων είναι εύκολη μέσω του μηχανισμού plugin.

– *Κλιμακωσιμότητα*: Το Slurm είναι σχεδιασμένο έτσι ώστε να κλιμακώνει σε συστοιχίες με χιλιάδες κόμβους. Ο controller του Slurm (τον οποίο θα αναλύσουμε σύντομα) σε ένα σύστημα με 1000 κόμβους καταναλώνει μνήμη της τάξης των 2 MB, δίχως να παρουσιάζεται μείωση της απόδοσης. Οι εργασίες που υποβάλλονται στο σύστημα μπορούν να προσδιορίζουν τις ανάγκες τους όσον αφορά τους πόρους με διάφορους τρόπους.

– *Ανεκτικότητα σφαλμάτων*: Το Slurm μπορεί να διαχειριστεί ποικλία σφαλμάτων στα επιμέρους αρχιτεκτονικά μέρη του (ακόμη και του controller) δίχως να θέσει σε κίνδυνο το φόρτο εργασίας. Μέσω παραμετροποίησης, δίνεται η δυνατότητα στις εργασίες των χρηστών να συνεχίσουν την εκτέλεσή τους στην περίπτωση αποτυχίας των κόμβων στους οποίους εκτελούνταν. Οι κόμβοι που είχαν ανατεθεί σε μία εργασία είναι διαθέσιμοι για επαναχρησιμοποίηση μόλις τερματίσει η συγκεκριμένη εργασία. Αν για κάποιο λόγο κάποιοι κόμβοι αποτύχουν να ολοκληρώσουν τον τερματισμό μιας εργασίας εγκαίρως, τότε θα επηρεαστεί μόνο η χρονοδρομολόγηση σε αυτούς και μόνο προβληματικούς κόμβους.

– *Ασφάλεια*: Το Slurm χρησιμοποιεί μεθόδους κρυπτογράφησης για τον έλεγχο ταυτότητας (authentication) χρήστη προς υπηρεσία αλλά και υπηρεσία προς υπηρεσία, κάτι που μπορεί να παραμετροποιηθεί μέσω του μηχανισμού των plugins. Το Slurm δεν υποθέτει ότι το δίκτυο στο οποίο βρίσκονται οι κόμβοι είναι ασφαλές, ωστόσο θεωρεί ότι ολόκληρη η συστοιχία βρίσκεται εντός ενός ενιαίου τομέα διαχείρισης με μία κοινή βάση χρηστών.

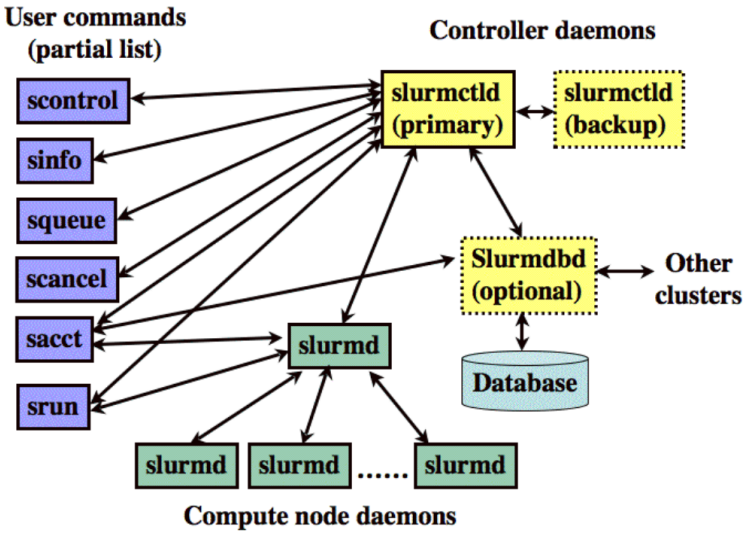
– *Ευκολία διαχείρισης*: Το Slurm χρησιμοποιεί ένα απλό configuration αρχείο για την διαχείρηση και παραμετροποίησή του το οποίο είναι δυνατό να αλλάξει ανά πάσα στιγμή χωρίς να επηρεάζεται η εκτέλεση των εργασιών. Ετερογενείς κόμβοι μέσα στη συστοιχία μπορούν να διαχειριστούν με ευκολία. Οι διεπαφές (interfaces) που προσφέρει το Slurm μπορούν να χρησιμοποιηθούν από scripts και η συμπεριφορά του είναι εξαιρετικά ντετερμινιστική.

Ως ένα σύστημα διαχείρισης πόρων, το Slurm παρέχει τρεις βασικές λειτουργίες. Αρχικά, αναθέτει σε χρήστες αποκλειστική ή/και μη αποκλειστική πρόσβαση σε πόρους (υπολογιστικούς κόμβους) για ένα συγκεκριμένο χρονικό διάστημα έτσι ώστε να ολοκληρώνουν εργασίες. Επίσης, προσφέρει ένα μέσο για την υποβολή, εκτέλεση και παρακολούθηση εργασιών. Τέλος, διευθετεί τυχόν συγκρουόμενα αιτήματα για πόρους διατηρώντας μία ουρά με τις εργασίες προς εκτέλεση.

Οι χρήστες αλληλεπιδρούν με το Slurm κυρίως μέσω τεσσάρων προγραμμάτων γραμμής εντολών: *srun* για την υποβολή μιας εργασίας και προαιρετικά τον έλεγχό της διαδραστικά (interactive mode), *scancel* για την ακύρωση μιας εργασίς που ήδη εκτελείται ή αναμένει στην ουρά για να εκτελεστεί, *squeue* και *sinfo* για την παρακολούθηση της ουράς εργασιών και της συνολικής κατάστασης του συστήματος αντίστοιχα. Οι διαχειριστές του συστήματος πραγματοποιούν προνομιούχες (privileged) ενέργειες μέσω της εντολής *scontrol*.

4.2 Αρχιτεκτονική

Όπως απεικονίζεται στην εικόνα 4.1, το Slurm απαρτίζεται από διάφορα αρχιτεκτονικά μέρη. Κύριο συστατικό του συστήματος αποτελεί ο δαίμονας (daemon) *slurmctld* ή αλλιώς controller, ο οποίος αντιπροσωπεύει τον «εγκέφαλο» που διαχειρίζεται κεντρικές λειτουργικότητες του συστήματος. Αν ο κόμβος στον οποίο τρέχει ο συγκεκριμένος δαίμονας υποστεί αποτυχία, τότε τον έλεγχο αναλαμβάνει ένας εφεδρικός κόμβος (fail-over twin), αν αυτός υπάρχει καθώς η παρουσία του είναι προαιρετική. Ο *slurmctld* controller μέσω διεπαφών προσφέρει εντολές όπως scontrol, sinfo, squeue, scancel, sacct, srun και άλλες. Αυτές οι εντολές, οι οποίες μπορούν να εκτελεστούν από οποιονδήποτε κόμβο της συστοιχίας, επιτρέπουν σε έναν χρήστη να υποβάλλει και να παρακολουθεί εργασίες καθώς επίσης και να διαχειρίζεται πόρους του συστήματος εφόσον βέβαια έχει επιτραπεί σε αυτόν να προβεί σε αυτήν την ενέργεια.



Σχήμα 4.1: Αρχιτεκτονική του Slurm

Όταν αποκαλούμε τον *slurmctld* «εγκέφαλο» του συστήματος, τότε θα μπορούσαμε να αποκαλέσουμε τους *slurmd* δαίμονες ως τα «χέρια» του, αφού αυτοί

εκτελούνται σε κάθε κόμβο του συστήματος, φέρνοντας εις πέρας εισερχόμενες εντολές. Καθήκον τους αποτελεί η εκτέλεση, παρακολούθηση και σηματοδότηση εργασιών, ενώ είναι υπεύθυνοι για την συλλογή πληροφοριών του μηχανήματος στο οποίο τρέχουν και την αποστολή τους στον controller. Όπως φαίνεται και στην εικόνα 4.1 υπάρχουν κάποιες εντολές οι οποίες εκτελούνται απευθείας στους *slurmd* δαίμονες. Αυτό συμβαίνει απλά για να αποφορτίσουν τον controller από τα πολλαπλά αιτήματα. Για τον ίδιο λόγο, όταν ο *slurmctld* χρειάζεται να μεταφέρει μία εντολή στους *slurmd* δαίμονες, επικοινωνεί μόνο με έναν απο αυτούς ο οποίος θα αποτελέσει την ρίζα ενός κατανεμημένου δέντρου επικοινωνίας.

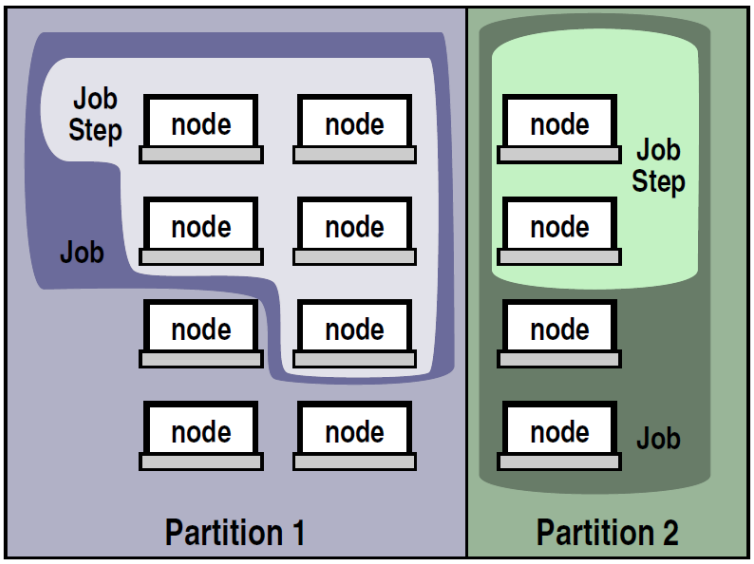
Τέλος, το τελευταίο και προαιρετικό συστατικό του Slurm αποτελεί ο *slurmdbd* δαίμονας, που πιθανά να είναι συνδεδεμένος με περισσότερα από ένα Slurm clusters, ο οποίος προσφέρει μία διεπαφή για μία βάση δεδομένων. Αυτό είναι ιδιαίτερα χρήσιμο για την αποθήκευση και πρόσβαση σε πληροφορίες για προηγούμενες εργασίες που έιχαν υποβληθεί στο σύστημα.

4.2.1 Οντότητες

Κάθε πολύπλοκο τεχνητό σύστημα έχει τις δικές τους αφηρημένες έννοιες που αποκαλούνται οντότητες, οι οποίες αντιπροσωπεύουν κομμάτια του πραγματικού κόσμου, και το Slurm δεν γίνεται να αντιβαίνει τον κανόνα. Μία από τις βασικότερες οντότητες του συστήματος είναι ο *κόμβος* (*node*) και σέβεται τον ορισμό που δόθηκε σε προηγούμενο κεφάλαιο. Ο εκάστοτε κόμβος θα πρέπει να συμπεριλαμβάνεται σε τουλάχιστον μια *διαμέριση* (*partition*) η οποία συγκεντρώνει πόρους σε μία λογική δομή. Δεν είναι απαραίτητο, λοιπόν, οι διαμερίσεις να είναι ανεξάρτητα σύνολα μεταξύ τους. Οι διαμερίσεις χρησιμοποιούνται συνήθως για τη συσχέτιση κόμβων που έχουν ισχυρή ομοιότητα στα χαρακτηριστικά τους, δηλαδή παρόμοιες λειτουργίες σε hardware ή/και software. Φυσικά θα πρέπει να υπάρχουν και οντότητες οι οποίες θα δίνουν λόγο ύπαρξης στις προηγούμενες: η *εργασία* (*job*) ή μερίδιο πόρων ανατεθειμένων σε ένα χρήστη για ένα συγκεκριμένο χρονικό διάστημα και το *βήμα εργασίας* (*job step*) το οποίο αποτελεί ένα σύνολο από (πιθανά παράλληλα) tasks μέσα σε μία εργασία. Κάθε διαμέριση μπορεί να θεωρηθεί σαν μια ξεχωριστή ουρά εργασιών έχοντας η καθεμία ένα σύνολο από περιορισμούς όπως μέγιστος αριθμός εργασιών, χρόνος εκτέλεσης, επιτρεπόμενοι χρήστες κ.α. Οι εργασίες κατανέμονται σε κόμβους μίας διαμέρισης βάσει προτεραιότητας έως ότου οι πόροι της διαμέρισης να εξαντληθούν. Από τη στιγμή που ένα σύνολο πόρων έχει ανατεθεί σε μία εργασία, ο χρήστης είναι σε θέση να εκκινήσει παράλληλες δουλειές σε μορφή βημάτων με οποιαδήποτε παραμετροποίηση είναι εντός της ανάθεσης. Για παράδειγμα, ένα και μόνο βήμα μπορεί να ξεκινήσει χρησιμοποιώντας όλους τους πόρους που έχουν κατανεμηθεί στην εργασία, ή πολλαπλά βήματα μπορούν να ξεκινήσουν ταυτόχρονα χρησιμοποιώντας μέρος της ανάθεσης το καθένα.

Στο σχήμα 4.2 απεικονίζεται η συσχέτιση των διαφορετικών οντοτήτων που ορίζει το Slurm. Ένα σύνολο από κόμβους έχει χωριστεί σε δύο ξεχωριστές διαμερίσεις. Στη διαμέριση 1 εκτελείται μία εργασία, με ένα βήμα να καταναλώνει το σύνολο των

πόρων που της έχουν ανατεθεί. Η εργασία στην διαμέριση 2 έχει ένα βήμα το οποίο καταναλώνει μόνο τους μισούς πόρους απο αυτούς που έχουν ανατεθεί συνολικά στην εργασία. Αυτή η εργασία μπορεί μελλοντικά να εκκινήσει επιπλέον βήματα τα οποία θα καλύψουν τους αδρανείς κόμβους της ανάθεσής της.



Σχήμα 4.2: Οντότητες του Slurm [3]

4.2.2 Δαίμονας Slurmd

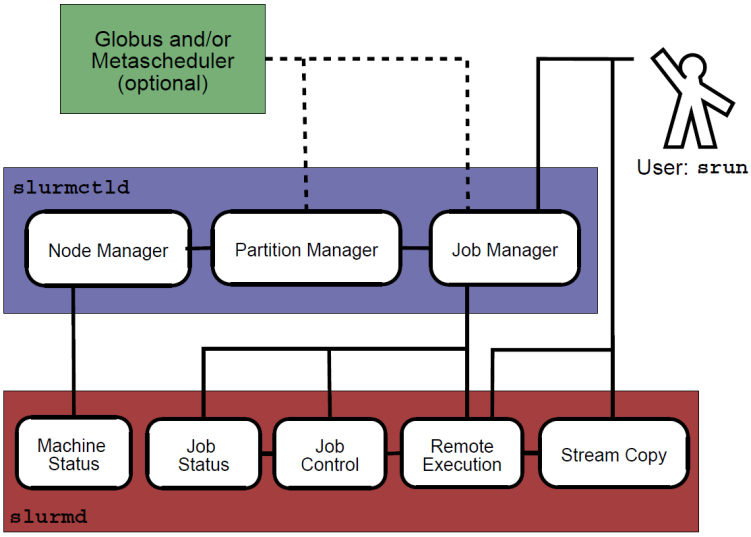
Το *slurmd* πρόκειται για έναν πολυνηματικό δαίμονα ο οποίος τρέχει στον κάθε υπολογιστικό κόμβο και σκοπός του είναι να διαβάζει την παραμετροποίηση του Slurm από το configuration αρχείο, ειδοποιεί τον controller ότι είναι ενεργός, περιμένει για εργασίες, εκτελεί τις εργασίες, επιστρέφει το αποτέσμα της εκτέλεσης, περιμένει για περισσότερη εργασία. Επειδή εκκινεί εργασίες εκ μέρους άλλων χρηστών, είναι αναγκαίο να τρέχει με δικαιώματα *root*. Επίσης ανταλλάσει ασύγχρονα πληροφορία με τον *slurmctld* σχετικά με την κατάσταση της εκτελούμενης εργασίας και τον κόμβο στον οποίο βρίσκεται. Η μοναδική πληροφορία που διατηρεί οποιαδήποτε στιγμή αφορά αποκλειστικά στις εκτελούμενες εργασίες εκείνη τη στιγμή. Τα κύρια δομικά στοιχεία του *slurmd* όπως απεικονίζονται και στο σχήμα 4.3 είναι τα εξής:

- `Machine and Job Status Services`: Απαντά σε αιτήματα προερχόμενα από τον controller σχετικά με την κατάσταση του host μηχανήματος και των εργασιών που εκτελούνται τη δεδομένη χρονική στιγμή. Κάποιες ενημερώσεις όπως για παράδειγμα η εκκίνηση του δαίμονα στέλνονται ασύγχρονα.
- `Remote Execution`: Ξεκινά, παρακολουθεί και καθαρίζει διεργασίες (οι οποίες συνήθως ανήκουν σε μία παράλληλη εργασία) όπως διατάσσει ο

slurmctld ή οι εντολές srun και scancel. Η εκκίνηση μίας διεργασίας μπορεί να περιλαμβάνει εκτός των άλλων την εκτέλεση του prolog προγράμματος, την ρύθμιση ορίων διεργασίας, τη ρύθμιση του πραγματικού και ισχύων uid, την αρχικοποίηση του stdio και τη διαχείριση ομάδων διεργασιών (group processes). Ο τερματισμός μιας διεργασίας από την άλλη μπορεί να περιλαμβάνει τον τερματισμό όλων των μελών του αντίστοιχου process group και την εκτέλεση του epilog προγράμματος.

– `Stream Copy Service`: Επιτρέπει τον χειρισμό των stdin, stdout και stderr απομακρυσμένων tasks. Η είσοδος της εργασίας μπορεί να προέλθει από ένα αρχείο ή πολλαπλά αρχεία (ένα για κάθε task), από μία εντολή srun ή από το /dev/null. Η έξοδος της εργασίας είναι δυνατό να αποθηκευθεί σε τοπικά αρχεία του κόμβου ή να επιστραφεί στην εντολή srun. Ανεξαρτήτως από το που έχει κατευθυνθεί το stdout/stderr, όλη η έξοδος που παράγεται από μία εργασία αποθηκεύεται σε μία ενδιάμεση μνήμη (buffering) για να αποφευχθεί το μπλοκάρισμα τοπικών tasks.

– `Job Control`: Επιτρέπει την ασύγχρονη επικοινωνία με το περιβάλλον του Remote execution προωθώντας σήματα ή αιτήματα τερματισμού προς οποιαδήποτε διεργασία εκτελείται τοπικά.



Σχήμα 4.3: Υποσυστήματα του Slurm [3]

4.2.3 Δαίμονας Slurmctld

Η πλειοψηφία της πληροφορίας της κατάστασης του Slurm διατηρείται στον *slurmctld*. Ο *slurmctld* αποτελεί επίσης έναν πολυνηματικό δαίμονα με ανεξάρτητους

μηχανισμούς κλειδώματος για read και write operations έχοντας κατά νου την κλιμακωσιμότητα που αναφέρθηκε προηγουμένως. Κατά την εκκίνησή του, ο controller (όπως αλλιώς αναφέρεται), διαβάζει την παραμετροποίηση του συστήματος από το configuration file και οποιαδήποτε τυχόν αποθηκευμένη πληροφορία κατάστασης. Ολόκληρη η κατάσταση του controller σώζεται στο δίσκο ανά τακτά χρονικά διαστήματα, ή κατευθείαν αν αλλάξει η παραμετροποίηση για λόγους ανεκτικότητας σφαλμάτων. Ο *slurmctld* έχει δύο λειτουργίες εκτέλεσης (mode): master ή standby ανάλογα την κατάσταση στην οποία βρίσκεται ο εφεδρικός controller (fail-over twin), αν αυτός υπάρχει. Σε αντίθεση με τον *slurmd*, αυτός ο δαίμονας δεν είναι απαραίτητο να εκτελείται με root δικαιώματα. Προτείνεται ένας μοναδικός user του συστήματος να εκτελεί τον *slurmctld*, ο οποίος θα πρέπει να αναφέρεται στο configuration αρχείο του Slurm ως `SlurmUser`. Τα κύρια δομικά στοιχεία του *slurmctld* όπως απεικονίζονται και στο σχήμα 4.3 είναι τα εξής:

- `Node Manager`: Παρακολουθεί την κατάσταση του κάθε κόμβου που ανήκει στο cluster. Αυτό επιτυγχάνεται είτε ρωτώντας περιοδικά τους *slurmd*s για την κατάστασή τους είτε λαμβάνοντας ασύγχρονες ενημερώσεις από αυτούς. Διασφαλίζει ότι οι κόμβοι έχουν την αναμενόμενη παραμετροποίηση πριν τους θεωρήσει κατάλληλους για ανάθεση κάποιας εργασίας.

- `Partition Manager`: Ομαδοποιεί τους κόμβους σε σύνολα που όπως αναφέραμε ονομάζονται διαμερίσεις (partitions). Αναθέτει κόμβους σε εργασίες βάσει της κατάστασής τους και της παραμετροποίησης. Αιτήματα που αφορούν εκκίνηση εργασιών προέρχονται από τον Job Manager. Μεταβάλλει την παραμετροποίηση κόμβων και διαμερίσεων σύμφωνα με εντολές scontrol.

- `Job Manager`: Δέχεται αιτήματα για εργασίες και τοποθετεί όσες εκκρεμούν σε μία ουρά προτεραιότητας. Ενεργοποιείται είτε περιοδικά είτε στην περίπτωση που υπάρχει κάποια αλλαγή κατάστασης η οποία πιθανώς να επιτρέπει την εκκίνηση μιας εργασίας που βρίσκεται στην ούρα. Μία τέτοια αλλαγή μπορεί να είναι η ολοκλήρωση μιας εργασίας, η υποβολή μιας νέας εργασίας, η μετάβαση της κατάστασης μίας διαμέρισης ή ενός κόμβου σε ενεργή κ.α. Τότε, λοιπόν, ο Job Manager επιλέγει μία εργασία από την ουρά (θα αναφέρουμε σε λίγο πως) για την κάθε διαθέσιμη διαμέριση και τις αναθέτει πόρους αν αυτό είναι δυνατόν. Εφόσον ανατεθούν πόροι σε μία εργασία, τότε ο Job Manager είναι υπεύθυνος να μεταφέρει την απαραίτητη πληροφορία στους συγκεκριμένους κόμβους για της εκτέλεση αυτής.

4.2.4 Δαίμονας Slurmdbd

Το Slurm προσφέρει τη δυνατότητα συλλογής πληροφοριών για την εκάστοτε εργασία και τα επιμέρους βήματά τους. Τα δεδομένα αυτά μπορούν να καταγραφούν είτε σε ένα απλό αρχείο είτε σε μια βάση δεδομένων. Η αποθήκευση δεδομένων ιστορικότητας σε βάση δεδομένων απαιτεί την ύπαρξη ενός επιπλέον δαίμονα, τον

slurmdbd daemon έτσι ώστε η πρόσβαση στα δεδομένα να γίνεται με ασφάλεια και ελεγχόμενο τρόπο. Στην περίπτωση που δεν υπήρχε ο δαίμονας αυτός, τότε οποιαδήποτε εντολή του Slurm που μπορεί να χρησιμοποιηθεί από έναν απλό χρήστη και σχετίζεται με την ανάκτηση δεδομένων, θα έπρεπε να παρέχει και τα αντίστοιχα αναγνωριστικά (credentials) για την πρόσβαση στη βάση δεδομένων. Κάτι τέτοιο όπως είναι λογικό δημιουργεί θέματα ασφαλείας. Με την ύπαρξη του slurmdbd δαίμονα, η ταυτοποίηση των χρηστών γίνεται με κεντρικό τρόπο στο Slurm (περισσότερα στην επόμενη ενότητα) και πρόσβαση στη βάση δεδομένων έχει αποκλειστικά ο συγκεκριμένος δαίμονας αυξάνοντας το επίπεδο ασφαλείας του συστήματος. Έχει, δηλαδή, το ρόλο του διαμεσολαβητή. Εκτός όμως από ασφάλεια, το slurmdbd παρέχει και αυξημένη επίδοση στην ανάκτηση δεδομένων καθώς χρησιμοποιεί τεχνικές caching.

Η παράμετρος που καθορίζει τον τύπο αποθήκευσης των πληροφοριών καθορίζεται από την παράμετρο *AccountingStorageType* του κεντρικού αρχείου παραμετροποίησης του Slurm (*slurm.conf*). Οι πιθανές τιμές που μπορεί να πάρει είναι οι εξής:

* `accounting_storage/none` - δεν αποθηκεύονται δεδομένα ιστορικότητας για τις εργασίες. Αποτελεί προεπιλογή.

* `accounting_storage/filetxt` - τα δεδομένα ιστορικότητας αποθηκεύονται στο αρχείο που καθορίζεται από την παράμετρο *AccountingStorageLoc*. Επιτρέπει την αποθήκευση περιορισμένου αριθμού πεδίων για την εκάστοτε εργασία.

* `accounting_storage/slurmdbd` - τα δεδομένα ιστορικότητας αποθηκεύονται μέσω του slurmdbd daemon ο οποίος διαχειρίζεται μια βάση δεδομένων MySQL.

4.3 Ασφάλεια

Το Slurm έχει ένα από μοντέλο ασφαλείας: οποιοσδήποτε χρήστης του cluster μπορεί να υποβάλει εργασίες προς εκτέλεση και να ακυρώσει τις δικές του εργασίες. Οποιοσδήποτε χρήστης έχει πρόσβαση για να δει την παραμετροποίηση του Slurm και πληροφορίες κατάστασης. Μόνο χρήστες με αναβαθμισμένα δικαιώματα (privileged users) είναι σε θέση να μεταβάλλουν την παραμετροποίηση, να ακυρώσουν οποιαδήποτε εργασία, ή να προβούν σε άλλες ενέργειες περιορισμένης πρόσβασης. Ως priviliged θεωρούνται μόνο ο χρήστης `root` και `SlurmUser` που έχει οριστεί στο στο configuration αρχείο του Slurm. Στην περίπτωση που υπάρχει ανάγκη για μεταβολές της παραμετροποίησης από άλλους χρήστες τότε θα πρέπει να χρησιμοποιηθούν Set-UID προγράμματα (π.χ. *setuid*, *setgid*) για να οριστούν συγκεκριμένα δικαιώματα σε συγκεκριμένους χρήστες.

4.3.1 Επαλήθευση επικοινωνίας (Communication Authentication)

Ιστορικά, η επαλήθευση ταυτότητας μεταξύ διαφορετικών κόμβων (node-to-node) επιτυγχανόταν χρησιμοποιώντας δεσμευμένες πόρτες και Set-UID προγράμματα. Στο πλαίσιο αυτό, οι δαίμονες έλεγχαν αν η πόρτα προέλευσης του αιτήματος βρισκόταν κάτω από ένα συγκεκριμένο αριθμό οι οποίες είναι διαθέσιμες μόνο στους χρήστες root. Η επικοινωνία μέσω αυτής της σύνδεσης, λοιπόν, θεωρούνταν έμμεσα αξιόπιστες. Επειδή ο αριθμός των δεσμευμένων πορτών είναι περιορισμένος και καθώς τα Set-UID προγράμματα αποτελούν πιθανόν θέμα ασφαλείας, το Slurm αποφάσισε να χρησιμοποιήσει ένα σχήμα επαλήθευσης που δεν βασίζεται στις αναφερθείσες μεθόδους. Συγκεκριμένα, σε κάθε μήνυμα που ανταλλάσσεται σε ένα Slurm cluster, υπάρχει ένα αναγνωριστικό βάσει του οποίου μπορεί να προσδιοριστεί έγκυρα το uid και gid του αποστολέα. Εφόσον οι παραλήπτες του μηνύματος επιβεβαιώσουν την γνησιότητα του αναγνωριστικού, μπορούν να χρησιμοποιήσουν το uid και gid για να κρίνουν αν το μήνυμα προέρχεται από χρήστη με κατάλληλη εξουσιοδότηση.

Η υλοποίηση του παραπάνω μηχανισμού είναι καθήκον του αντίστοιχου «auth» plugin. Υπάρχουν δύο authentication plugins: *none* και *Munge*. Το *none* χρησιμοποιεί ως τιμή του αναγνωριστικού την τιμή `null` και είναι κατάλληλο μόνο για δοκιμές και δίκτυα όπου η ασφάλεια δεν είναι απαραίτητη. Το *Munge* χρησιμοποιεί μέθοδο κρυπτογράφησης για την παραγωγή του αναγνωριστικού του αποστολέα. Φυσικά, μέσω του μηχανισμού των επεκτάσεων είναι εύκολο να αναπτυχθεί και να χρησιμοποιηθεί οποιαδήποτε άλλη υλοποίηση.

4.3.2 Επαλήθευση εργασιών (Job Authentication)

Από τη στιγμή που ο controller αναθέσει ορισμένους πόρους σε έναν χρήστη, τότε για το κάθε βήμα εργασίας που θα εκτελέσει ο χρήστης δημιουργείται ένα αναγνωριστικό συνδυάζοντας το uid του χρήστη, το job id της εργασίας, το step id του συγκεκριμένου βήματος, τη λίστα των ανατεθειμένων πόρων και την διάρκεια ζωής του αναγνωριστικού. Στη συνέχεια, ο controller το κρυπτογραφεί με το ιδιωτικό του κλειδί. Αυτό το αναγνωριστικό ορίζει στον χρήστη πρόσβαση στους πόρους που του ανατέθηκαν και ο *slurmd* δεν χρειάζεται πλέον να επικοινωνήσει με τον *slurmctld* για να επαληθεύσει την ορθότητα του αιτήματος για εργασία. Αρκεί να χρησιμοποιήσει το δημόσιο κλειδί του controller για να εξακριβώσει τη γνησιότητα του αναγνωριστικού και αν τα επιμέρους στοιχεία αυτού ταυτίζονται με το αίτημα. Χρησιμοποιείται, λοιπόν, κρυπτογραφία δημόσιου κλειδιού.

4.3.3 Εξουσιοδότηση (Authorization)

Υπάρχει η δυνατότητα να περιορίζεται η πρόσβαση σε μία διαμέριση μέσω του *RootOnly* (flag). Αν είναι ενεργοποιημένη αυτή η ρύθμιση, τότε σε αυτή τη

διαμέριση γίνονται δεκτές εργασίες και αιτήματα ανάθεσης πόρων μόνο αν το ισχύον uid (effective uid) του αποστολέα ανήκει σε χρήστη με ανεβασμένα δικαιώματα. Όπως έχουμε αναφέρει, ένας τέτοιος χρήστης μπορεί να υποβάλλει εργασίες εκ μέρους οποιουδήποτε άλλου χρήστη. Επίσης, υπάρχει η δυνατότητα περιορισμού πρόσβασης σε διαμερίσεις βάσει του group στο οποίο ανήκουν χρησιμοποιώντας τη ρύθμιση *AllowGroups* στο configuration αρχείο του Slurm.

4.4 Εντολές

Σε αυτήν την υποενότητα θα αναφερθούν κάποιες βασικές εντολές του Slurm που δίνουν τη δυνατότητα σε έναν απλό χρήστη του συστήματος να εκτελέσει μία νέα εργασία (*srun, sbatch, salloc*), να ακυρώσει μια εργασία (*scancel*), να δει πληροφορίες του συστήματος και την κατάσταση στην οποία αυτό βρίσκεται (*sinfo, sacct, squeue*). Ενδιαφέρον αποτελεί το γεγονός ότι όλες οι εντολές προς το Slurm ξεκινάνε με τον χαρακτήρα «s».

`srun`

Η εντολή *srun* χρησιμοποείται για την εκκίνηση παράλληλων εργασιών ή βήματα εργασιών στο cluster. Στην περίπτωση που οι πόροι για την εκτέλεση της εργασίας δεν έχουν ήδη ανατεθεί, για παράδειγμα όταν η εντολή εκτελείται απευθείας στο τερματικό ενός κόμβου, τότε θα γίνει πρώτα η απαραίτητη κατανομή πόρων. Αντίθετα, αν η εντολή κληθεί από μία εργασία που ήδη εκτελείται, για πράδειγμα μέσω ενός batch script (μέσω της εντολής *sbatch*), τότε ξεκινάει ένα καινούριο βήμα (job step). Αν δεν υπάρχουν διαθέσιμοι πόροι για την εκκίνηση της εργασίας, τότε η εντολή μπλοκάρει μέχρι να υπάρξουν διαθέσιμοι, διαφορετικά ξεκινά αμέσως.

Φυσικά η εντολή *srun* δέχεται πληθώρα επιλογών. Μερικές από αυτές που θα χρησιμοποιηθούν και αργότερα στο κεφάλαιο 6 είναι οι εξής:

- `-N` / `--nodes`: καθορίζει τον απαιτούμενο αριθμό κόμβων για την εκτέλεση της εργασίας
- `-n` / `--ntasks`: καθορίζει τον αριθμό των διεργασιών (tasks) που θα εκτελεστούν και ζητά τον αντίστοιχο αριθμό κόμβων που να ικανοποιεί αυτή την απαίτηση
- `-c` / `--cpus-per-task`: καθορίζει τον μέγιστο αριθμό επεξεργαστών που κατανέμονται στην κάθε διεργασία (task). Στην περίπτωση που δεν οριστεί αυτή η επιλογή, τότε ως προεπιλογή η κάθε διεργασία εκτελείται σε έναν κόμβο.

Για παράδειγμα, η εντολή `srun -n4 -c8 my_program` δημιουργεί 4 διεργασίες οι οποίες περιορίζονται σε 8 επεξεργαστές η καθεμία. Οπότε αν το cluster αποτελείται από κόμβους των 16 επεξεργαστών, τότε 2 κόμβοι θα χρειαστούν για την εκτέλεση της εργασίας.

salloc

Η εντολή *salloc* χρησιμοποιείται για τη δημιουργία μιας ανάθεσης κόμβων, πιθανώς βάσει κάποιων περιορισμών (π.χ. αριθμός επεξεργαστών ανά κόμβο). Όταν αποκτηθούν οι αιτούμενοι πόροι τότε εκτελείται η εντολή που έχει δοθεί σαν όρισμα στην *salloc*, και αφού ολοκληρωθεί η εκτέλεση απελευθερώνονται οι πόροι.

Για παράδειγμα, η εντολή `salloc -N5 srun -n10 my_program` αποκτά πρώτα 5 κόμβους και στη συνέχεια εκτελεί ένα παράλληλο πρόγραμμα. Όμοια, η εντολή `salloc -N16 xterm` αποκτά τους κομβους και εκτελεί ένα τερματικό μέσω του οποίου στη συνέχεια μπορούν να εκτελεστούν διαδραστικά άλλες εντολές.

sbatch

Η εντολή *sbatch* χρησιμοποιείται για την υποβολή batch scripts στο σύστημα. Αυτός είναι και ο προτιμότερος τρόπος εκτέλεσης εργασιών στο Slurm, καθώς επιτρέπει στο χρονοδρομολογητή του συστήματος, βάσει της πολιτικής χρονοδρομολόγησης, να διαλέξει την κατάλληλη στιγμή εκκίνησής τους για να διατηρήσει υψηλά ποσοστά χρήσης του συστήματος και job throughput.

Ένα batch script μπορεί να περιέχει επιλογές που έπονται του λεκτικού «`#SBATCH`» αντίστοιχες με αυτές της εντολής *srun* που αναλύσαμε παραπάνω. Η εντολή *sbatch* δεν μπλοκάρει μέχρι να εκτελεστεί η εργασία, αντιθέτως επιστρέφει μόλις η εργασία υποβληθεί με επιτυχία στο σύστημα και εισέλθει στην ουρά εργασιών. Όταν οι απαραίτητοι πόροι για την εκτέλεση γίνουν διαθέσιμοι, το Slurm εκτελεί ένα αντίγραφο του batch script στον πρώτο κόμβο την ανάθεσης. Η έξοδος που παράγει η εργασία αποθηκεύεται σε αρχείο ώστε να είναι προσβάσιμη από το χρήστη σε δευτερεύων χρόνο.

sacct

Η εντολή *sacct* δίνει τη δυνατότητα ανάκτησης πληροφοριών που σχετίζονται με τις εργασίες είτε από logs του Slurm είτε από τη βάση δεδομένων (αν χρησιμοποιείται). Οι επιλογές που προσφέρονται μπορούν να καθορίσουν τα πεδία που εμφανίζονται (επιλογή `--format=`) αλλά και να φιλτράρουν την αναζήτηση. Ως προεπιλογή, η εντολή *sacct* δείχνει πληροφορία για τις εργασίες, τα βήματά τους, την κατάστασή τους και αποτέλεσμα της εκτέλεσής τους (exit code).

Παρόμοια εντολή είναι η `squeue` η οποία όμως περιορίζεται σε εργασίες των οποίων η εκτέλεσή τους δεν έχει ολοκληρωθεί και βρίσκονται ακόμη στο σύστημα.

sinfo

Η εντολή *sinfo* παρέχει πληροφορίες που σχετίζονται με τις διαμερίσεις (partitions) του Slurm, την κατάστασή τους και τους κόμβους από τους οποίους απαρτίζονται. Μέσω επιλογών μπορεί να διαφοροποιηθεί η παρουσίαση και η ομαδοποίηση των αποτελεσμάτων. Για παράδειγμα, μέσω της εντολής `sinfo`

`-pbatch` λαμβάνουμε πληροφορία σχετικά με τη διαθεσιμότητα της διαμέρισης με όνομα batch, την κατάστασή της, τον περιορισμό σε όριο εκτέλεσης εργασιών και τους κόμβους της.

```
$ sinfo -pbatch
PARTITION AVAIL TIMELIMIT NODES STATE  NODELIST
batch     up        40:00     8 alloc  nodes[1-8]
```

scancel

Η εντολή *scancel* σταματά ή μεταφέρει σήματα σε εργασίες ή επιμέρους βήματα εργασιών που είτε εκτελούνται είτε αναμένουν στην ουρά για χρονοδρομολόγηση. Μπορεί να εκτελεστεί δίνοντας σαν όρισμα συγκεκριμένα αναγνωριστικά (job id, step id), για παράδειγμα: `scancel 17 42`, όπου τερματίζονται οι εργασίες με αριθμό 17 και 42. Επίσης, μπορεί να εκτελεστεί ορίζοντας φίλτρα εργασιών που μπορεί να αφορούν το όνομα της εργασίας, τη διαμέριση, την κατάσταση που βρίσκεται ή τον χρήστη που την υπέβαλε. Για παράδειγμα, η εντολή `scancel -tPENDING -ualex` θα τερματίσει όλες τις εργασίες που έχει υποβάλλει ο χρήστης alex και βρίσκονται σε κατάσταση αναμονής στην ουρά.

4.5 Επεκτάσεις

Μία από τις βασικές αρχές του Slurm και ένας από τους λόγους που είναι ευρέως διαδεδομένο αποτελεί η επεκτασιμότητα. Πρόκειται για προγραμματιστική υλοποίηση μιας αυστηρά ορισμένης διεπαφής (API) η οποία σχετίζεται άμεσα με διαφορετικούς τομείς του συστήματος όπως είναι η χρονοδρομολόγηση εργασιών, η ταυτοποίηση χρηστών, η καταγραφή ιστορικών δεδομένων κ.α. Το κάθε plugin έχει το δικό του μοναδικό αναγνωριστικό το οποίο μοιάζει με έναν τύπο MIME [20] <είδος>/<αναγνωριστικό>, για παράδειγμα sched/builtin, sched/backfill, select/linear, select/cons_res κ.ο.κ. Τα ενεργά plugins καθορίζονται από το configuration αρχείο *slurm.conf*.

Κεφάλαιο 5

Χρονοδρομολόγηση στο Slurm

5.1 Εισαγωγή

Το Slurm χρησιμοποιεί σχετικά απλούς προεπιλεγμένους αλγορίθμους χρονοδρομολόγησης ώστε να είναι σε συμφωνία με τους στόχους του αρχικού σχεδιασμού: απλότητα και αποδοτικότητα. Γεγονότα όπως η ολοκλήρωση μιας εκτελούμενης εργασίας, η υποβολή μιας νέας, ή μια αλλαγή στην παραμετροποίηση του συστήματος πυροδοτούν τη διαδικασία χρονοδρομολόγησης η οποία λαμβάνει υπόψη της περιορισμένο και καθορισμένο αριθμό εργασιών (*default_queue_depth* με προεπιλογή το 100) που βρίσκονται στην αρχή της ουράς προτεραιότητας. Πρόκειται δηλαδή για χρονοδρομολόγηση βάσει γεγονότων (event-triggered). Ωστόσο, ο προηγούμενος αλγόριθμος έχει τη δυνατότητα να λάβει υπόψη του όλες τις εργασίες που περιμένουν στην ουρά. Κάτι τέτοιο, όπως είναι λογικό, κοστίζει σε χρόνο και γι΄ αυτό εκτελείται σε πιο αραία χρονικά διαστήματα βάσει μιας περιόδου (*sched_interval*).

Σε αυτό το κεφάλαιο, παρουσιάζονται ενδεικτικές παράμετροι του Slurm που επηρεάζουν άμεσα την χρονοδρομολόγηση εργασιών, τα scheduler plugins που έρχονται ενσωματωμένα με τον κώδικα του Slurm ενώ στη συνέχεια γίνεται μελέτη ορισμένων προηγμένων τεχνικών χρονοδρομολόγησης που προσφέρει ως σύστημα διαχείρισης πόρων.

5.2 Παραμετροποίηση

Πλέον είναι γνωστό ότι το Slurm είναι παραμετροποιήσιμο από πολλές πτυχές, και η χρονοδρομολόγηση εργασιών είναι μία από αυτές. Μερικές από τις πιο βασικές παραμέτρους του configuration αρχείου του Slurm (*slurm.conf*) που επηρεάζουν άμεσα την χρονοδρομολόγηση παρουσιάζονται παρακάτω:

– *SchedulerType*: Προσδιορίζει ποιο scheduler plugin θα χρησιμοποιηθεί για την χρονοδρομολόγηση εργασιών. Αυτή η επιλογή θα αναλυθεί επιπλέον σε

επόμενη ενότητα.

- *SchedulerParameters*: Αποτελεί μια οικογένεια άλλων παραμέτρων η ερμηνεία των ποίων εξαρτάται από το scheduler plugin που είναι ενεργό.

- *defer*: Με την ενεργοποίηση αυτής της επιλογής, αποφεύγεται η εκτέλεση της διαδικασίας χρονοδρομολόγησης κάθε φορά που υποβάλλεται μια νέα εργασία στο σύστημα.

- *default_queue_depth*: Ο προεπιλεγμένος αριθμός εργασιών για τις οποίες θα γίνει προσπάθεια χρονοδρομολόγησης σε κάθε γεγονός του συστήματος, σε αντίθεση με την παραπάνω που αφορά όλες τις εργασίες που βρίσκονται στην ουρά αναμονής.

- *sched_max_job_start*: Ο μέγιστος αριθμός εργασιών που ο βασικός μηχανισμός χρονοδρομολόγησης θα προσπαθήσει να ξεκινήσει. Ως προεπιλογή είναι ορισμένο το 0 το οποίο δεν θέτει τέτοιο όριο.

- *batch_sched_delay*: Ο χρόνος, σε δευτερόλεπτα, που μπορεί να καθυστερήσει η χρονοδρομολόγηση των εργασιών τύπου batch. Χρήσιμο σε ένα περιβάλλον όπου υποβάλλονται εργασίες batch με μεγάλο ρυθμό και θέλουμε να επιτύχουμε μείωση του κόστους χρονοδρομολόγησης.

- *sched_min_interval*: Πόσο συχνά, σε μs, ο βασικός μηχανισμός χρονοδρομολόγησης θα εκτελείται και θα λαμβάνει υπόψη εργασίες ίσες με την τιμή της παραμέτρου *default_queue_depth*. Κανονικά, όπως αναφέρθηκε και στην προηγούμενη ενότητα, ο μηχανισμός εκτελείται σε κάθε γεγονός που συμβαίνει στο σύστημα. Αν αυτά τα γεγονότα είναι πολύ συχνά, η χρονοδρομολόγηση είναι λογικό να δαπανά σχετικά μεγάλο όγκο πόρων. Θέτοντας την επιλογή αυτή, κάτι τέτοιο μπορεί να αποφευχθεί.

- *sched_interval*: Πόσο συχνά, σε δευτερόλεπτα, ο βασικός μηχανισμός χρονοδρομολόγησης θα εκτελείται και θα λαμβάνει υπόψη όλες τις εργασίες της ουράς.

5.3 Επεκτάσεις Χρονοδρομολόγησης

Για τον εμπλουτισμό της διαδικασίας χρονοδρομολόγησης, το Slurm προσφέρει τη δυνατότητα ορισμού μίας επέκτασης. Για την παραμετροποίηση και τον προσδιορισμό της επέκτασης χρονοδρομολόγησης χρησιμοποιείται το πεδίο *SchedulerType* του configuration αρχείου *slurm.conf*. Να σημειωθεί ότι υπάρχει η δυνατότητα αλλαγής του shceduling plugin, αλλά για να αναγνωριστεί από το Slurm θα πρέπει να γίνει επανεκκίνηση του slurmctld δαίμονα. Τα plugins που υπάρχουν αυτή τη στιγμή ενσωματωμένα στον κώδικα του Slurm (έκδοση 18.08) είναι τα εξής:

* `sched/builtin`
  Πρόκειται για τον κλασικό FIFO scheduler ο οποίος ξεκινά εργασίες βάσει προτεραιότητας. Όπως έχουμε πει υπάρχουν διαφορετικές ουρές προτεραιότητας για την εκάστοτε διαμέριση. Αν μια εργασία δεν μπορεί να χρονοδρομολογηθεί στη διαμέριση που έχει ζητήσει, τότε και καμία άλλη με χαμηλότερη προτεραιότητα δεν θα ξεκινήσει. Εξαίρεση αποτελεί η περίπτωση κατά την οποία μια εργασία δεν είναι δυνατόν να ξεκινήσει ποτέ στη συγκεκριμένη διαμέριση λόγω περιορισμών (π.χ. λιγότεροι κόμβοι από τους επιθυμητούς), οπότε χαμηλότερης προτεραιότητας jobs παίρνουν τη θέση της για χρονοδρομολόγηση.

* `sched/hold`
  Στην περίπτωση που υπάρχει το αρχείο «/etc/slurm.hold» τότε οι εργασίες που καταφθάνουν στο σύστημα απλώς «κρατώνται» σε αυτό και δεν γίνεται καθόλου η διαδικασία χρονοδρομολόγησης εργασιών. Αν, ωστόσο, δεν υπάρχει το συγκεκριμένο αρχείο, τότε χρησιμοποιείται ο ενσωματωμένος scheduler του Slurm.

* `sched/backfill`
  Ενισχύει τον ενσωματωμένο FIFO scheduler αυξάνοντας αισθητά τον ρυθμό διεκπεραίωσης (throughput) του συστήματος. Αυτό επιτυγχάνεται με την χρονοδρομολόγηση εργασιών χαμηλότερης προτεραιότητας στην περίπτωση που αυτό δεν προκαλεί καθυστέρηση στην εκκίνηση εργασιών υψηλότερης προτεραιότητας. Φυσικά, για να έχει βάση αυτή η στρατηγική, θα πρέπει οι χρήστες που υποβάλλουν εργασίες να ορίζουν και το αντίστοιχο χρονικό όριο (job time limit) γιατί διαφορετικά όλες οι εργασίες θα είχαν το ίδιο όριο και δε θα ήταν εφικτή η εφαρμογή του backfilling. Αποτελώντας το προεπιλεγμένο scheduling plugin του Slurm αρμόζει να μελετήσουμε τις επιμέρους παραμετροποιήσεις που υποστηρίζονται.

Επέκταση Backfill

Όπως αναφέρθηκε και προηγουμένως, το Slurm προσφέρει ένα *backfill* scheduler plugin η χρήση του οποίου μπορεί να βελτιώσει ιδιαιτέρως την διαδικασία της χρονομολόγησης σε σύγκριση με το FIFO scheduling που χρησιμοποιεί εσωτερικά το σύστημα. Δίχως backfill scheduling, οι εργασίες της εκάστοτε διαμέρισης χρονοδρομολογούνται αποκλειστικά και μόνο βάσει των προτεραιοτήτων τους που έχουν οριστεί από το σύστημα (μέσω priority plugin), με αποτέλεσμα την μείωση του ποσοστού χρησιμοποίησης πόρων του cluster η οποία θα μπορούσε να αποφευχθεί με λίγο εξυπνότερη χρονοδρομολόγηση. Όταν είναι ενεργό το backfill scheduler plugin, το σύστημα έχει τη δυνατότητα να κατανέμει πόρους σε εργασίες χαμηλότερης προτεραιότητας αν και μόνο αν δεν θα καθυστερήσει η εκκίνηση οποιασδήποτε άλλης εργασίας υψηλότερης προτεραιότητας. Ο αλγόριθμος βασίζεται στα χρονικά όρια που έχουν ορισθεί στις εργασίες από του χρήστες που τις

υποβάλλουν και για να είναι αποτελεσματικός θα πρέπει αυτά να είναι ακριβή. Υπάρχουν ορισμένοι παράμετροι του configuration αρχείου που σχετίζονται με τα χρονικά όρια εργασιών:

- *DefaultTime*: ορίζει το προεπιλεγμένο χρονικό όριο εργασίας ανά διαμέριση σε περίπτωση που ο χρήστης δεν το έχει θέσει κατά την υποβολή της
- *MaxTime*: ορίζει το μέγιστο χρονικό όριο εργασίας ανά διαμέριση
- *OverTimeLimit*: ορίζει τον αριθμό των λεπτών που μπορεί να ξεπεράσει η εκτέλεση μιας εργασίας σε σχέση με το χρονικό της όριο (μπορεί να είναι και *UNLIMITED*) προτού τερματιστεί από το σύστημα.

Η χρήση του backfill scheduling αποτελεί μια χρονοβόρα διαδικασία καθώς επεξεργάζεται τα δεδομένα όλων των εργασιών που βρίσκονται στην ουρά. Για να έχει πρόσβαση στη δομή της ουράς εργασιών, θα πρέπει να έχει αποκτήσει τα απαραίτητα locks, που σημαίνει ότι εκείνη τη στιγμή οι υπόλοιπες ενέργειες που χρειάζονται πρόσβαση σε αυτή μπλοκάρουν. Αναγκαστικά, λοιπόν, τα σχετιζόμενα locks απελευθερώνονται περίπου κάθε δύο δευτερόλεπτα ώστε να είναι δυνατή η επεξεργασία άλλων αιτημάτων όπως είναι η υποβολή μιας νέας εργασίας στο σύστημα [3]. Προσφέρεται η δυνατότητα μέσω παραμετροποίησης (bf_continue) να συνεχίσει ο αλγόριθμος του backfilling χωρίς την επαναπόκτηση των locks αγνοώντας βέβαια τις πιθανές νέες εργασίες που εισέρχονται στο σύστημα μέχρι την ολοκλήρωση του κύκλου του. Υπάρχει πληθώρα επιλογών που μπορούν να μεταβάλλουν τη συμπεριφόρα του αλγορίθμου backfill στο Slurm. Ενδεικτικά, μέσω παραμετροποίησης μπορεί να ορισθεί: η περίοδος με την οποία εκτελείται ο αλγόριθμος, ο μέγιστος αριθμός backfilled εργασιών ανά διαμέριση ή ανά χρήστη, το χρονικό παράθυρο μελλοντικών εργασιών κ.α.

5.4 Προηγμένες τεχνικές χρονοδρομολόγησης

Συνδυάζοντας τα scheduler plugins και την ευρεία παραμετροποίηση που προσφέρει το Slurm, είναι δυνατή η υποστήριξη προηγμένων τεχνικών χρονοδρομολόγησης εργασιών. Οι διαχειριστές του συστήματος, ανάλογα με τον στόχο που θέλουν να επιτύχουν κάθε φορά, μπορούν να εναλλάσσουν μεταξύ αυτών. Στις επόμενες υποενότητες μελετώνται μερικές από αυτές τις τεχνικές.

5.4.1 Gang Scheduling

Το Gang Scheduling αποτελεί μια πρακτική χρονοδρομολόγησης κατά την οποία δύο ή περισσότερες εργασίες με παρόμοια χαρακτηριστικά κατανέμονται στο ίδιο σύνολο πόρων του συστήματος. Αυτές οι εργασίες στη συνέχεια εκτελούνται εναλλάξ, με τέτοιο τρόπο που οποιαδήποτε στιγμή μόνο μία εργασία έχει

αποκλειστική πρόσβαση στους υπολογιστικούς πόρους. Το χρονικό διάστημα για το οποίο η εκάστοτε εργασία έχει τον πλήρη έλεγχο των πόρων ονομάζεται *timeslice* και μπορεί να παραμετροποιηθεί. Χρήση του Gang Scheduling στο Slurm επιτυγχάνει αύξηση του ρυθμού διεκπεραίωσης (throughput), καθώς επιτρέπει σε «μικρότερες» εργασίες να ξεκινήσουν νωρίτερα από άλλες «μεγαλύτερες», με ταυτόχρονη εκτέλεσή τους μέσω oversubscribing αντί να περιμένουν στην ουρά προτεραιότητας.

Το Slurm δημιουργεί ένα αυτόνομο timeslicer νήμα το οποίο αποτρέπει το starvation των εργασιών που υπόκεινται σε gang scheduling. Ο timeslicer ξυπνάει περιοδικά, στην αρχή του κάθε timeslice, και ελέγχει σε κάθε διαμέριση για εργασίες οι οποίες είναι σταματημένες. Αν υπάρχουν, τότε τοποθετεί όλες τις εκτελούμενες εργασίες στο τέλος της ουράς και επαναφέρει την εργασία που ήταν σε σταματημένη κατάσταση (suspended state) την περισσότερη ώρα. Στη συνέχεια, ελέγχει αν στα χαρακτηριστικά της εργασίας αυτής ταιριάζει κάποια ακόμη για να χρονοδρομολογηθούν στους ίδιους πόρους.

Μέσω παραμέτρων μπορεί να ορισθεί ο μεγιστος αριθμός εργασιών που θα μοιράζονται υπολογιστικούς πόρους (*OverSubscribe*), το μέγεθος τους timeslice σε δευτερόλεπτα (*SchedulerTimeSlice*). Η συνολική παραμετροποίηση που απαιτείται για την ενεργοποίηση του Gang Scheduling είναι προσβάσιμη στο [21].

5.4.2 Preemption

Στενά συνηφασμένο με το Gang Scheduling είναι το Preemption, δηλαδή η αναστολή εργασιών χαμηλότερης προτεραιότητας για να επιτραπεί η εκτέλεση εργασιών με υψηλότερη προτεραιότητα. Στην πραγματικότητα, το job preemption αποτελεί μια διαφοροποίηση της γενικότερης λογικής του Gang Scheduling. Όταν σε μια εργασία υψηλής προτεραιότητας κατανεμηθούν υπολογιστικοί πόροι που εκείνη τη στιγμή χρησιμοποιούνται από μία ή περισσότερες άλλες εργασίες χαμηλότερης προτεραιότητας, τότε αυτές αναστέλλονται. Θα συνεχίσουν την εκτέλεσή τους, στους ίδιους πόρους, όταν ολοκληρωθεί η εκτέλεση της εργασίας υψηλής προτεραιότητας ή μέσω παραμετροποίησης σε νεότερες εκδόσεις του Slurm μπορούν να επαναποθετηθούν στην ουρά προτεραιότητας για να περάσουν απο τη διαδικασία χρονοδρομολόγησης από την αρχή.

Οι δύο βασικές παράμετροι που ρυθμίζουν τον τρόπο με τον οποίο εκτελείται το preemption των εργασιών είναι οι εξής:

* **PreemptType**: προσδιορίζει τον μηχανισμό με τον οποίο θα αναγνωρίζονται από το σύστημα οι εργασίες που μπορούν να προκαλέσουν preemption. Προσφέρονται τρια plugins που αναλαμβάνουν τον υπολογισμό.
  - *preempt/none* δεν γίνεται καθόλου preemption.
  - *preempt/partition_prio* όπου εργασίες που ανήκουν σε μια διαμέριση μπορούν να κάνουν preempt εργασίες μιας διαμέρισης χαμηλότερης προτεραιότητας.

  - *preempt/qos* υποδεικνύει ότι εργασίες ενός QOS μπορούν να κάνουν preempt εργασίες χαμηλότερου QOS είτε στην ίδια είτε σε διαφορετική διαμέριση.

* **PreemptMode**: θέτει τον μηχανισμό με τον οποίο εργασίες χαμηλής προτεραιότητας θα γίνονται preempt. Αξίζει να σημειωθεί ότι στην περίπτωση που το PreemptType είναι το preempt/partition_prio, μπορεί να ορισθεί διαφορετικός μηχανισμός ανά διαμέριση.
  - *CANCEL* η preempted εργασία θα ακυρώνεται.
  - *CHECKPOINT* θα λαμβάνεται ένα «στιγμιότυπο» της preempted εργασίας (αν αυτό είναι δυνατό, διαφορετικά θα ακυρώνεται) το οποίο μπορεί να χρησιμοποιηθεί για τη συνέχιση της εκτέλεσής της.
  - *REQUEUE* η preempted εργασία θα επανατοποθετείται στην ουρά εργασιών (αν αυτό είναι δυνατό, διαφορετικά θα ακυρώνεται) οπότε μπορεί να εκτελεστεί από την αρχή σε διαφορετικούς πόρους ύστερα απο χρονοδρομολόγηση
  - *SUSPEND* η preempted εργασία θα αναστέλλεται και η εκτέλεσή της θα συνεχίζεται αργότερα. Στην περίπτωση που το PreemptType είναι το preempt/qos τότε οι εργασίες θα γίνονται gang scheduled

Η συνολική παραμετροποίηση που απαιτείται για την εφαρμογή του Job Preemption είναι προσβάσιμη στο [22].

5.4.3 Generic Resources

Τα Generic Resources (GRES) στο Slurm αναφέρονται σε άλλες συσκευές υλικού που είναι συνδεδεμένες σε κόμβους, συνήθως επιταχυντές. Το Slurm προς το παρόν υποστηρίζει Nvidia GPUs, CUDA Multi-Process Service (MPS) και Intel MICs μέσω επεκτάσεων. Για να αναγνωρίσει το Slurm το επιπλέον αυτό υλικό, οι διαχειριστές του συστήματος θα πρέπει να ορίσουν ονομασία του πόρου, αριθμό, CPUs που επιτρέπεται πρόσβαση στον πόρο κ.α. Αυτό επιτυγχάνεται μέσω των παραμέτρων GresTypes και Gres. Το GresTypes ορίζεται σε επίπεδο συστήματος όπου αναφέρονται τα ονόματα των generic resources (π.χ. GresTypes=gpu), ενώ η παράμετρος Gres ορίζεται σε επίπεδο κόμβου (π.χ. Gres=gpu:tesla:1,gpu:kepler:1). Οι χρήστες, αν θέλουν κάποιον κόμβο με συγκεκριμένα generic resources θα πρέπει να το ζητήσουν ρητά κατά την υποβολή εργασιών μέσω επιλογών (π.χ. *–gres, –gpu, –gpu-per-node*). Σε αντίθεση με τους άλλους υπολογιστικούς πόρους, τα GRES που έχουν κατανεμηθεί σε μια εργασία δεν γίνονται αυτόματα διαθέσιμα σε άλλες εργασίες αν αυτή αναστασλεί λόγω preemption. Υπάρχει η δυνατότητα το εκάστοτε βήμα εργασίας (job step) να αποκτήσει μέρος των συνολικών GRES που έχουν ανατεθεί στην εργασίας (προεπιλογή είναι το κάθε βήμα να αποκτά όλα τα GRES). Αυτό επιτρέπει ευελιξία στο μοίρασμα των GRES ανάμεσα σε ξεχωριστά job steps [23].

5.4.4 Elastic Computing

Το Slurm έχει τη δυνατότητα να υποστηρίξει ένα cluster το οποίο μεγαλώνει και μικραίνει on demand, βασιζόμενο σε μια υπηρεσία όπως το Amazon Elastic Computing Cloud (Amazon EC2) και το Google Cloud Platform (GCP) για την αξιοποίηση υπολογιστικών πόρων. Αυτοί οι πόροι μπορούν να ενσωματωθούν σε ένα προϋπάρχον cluster για την επεξεργασία του πλεονάζοντος φόρτου εργασίας (cloud bursting) ή μπορούν να λειτουργήσουν ως ένα αυτόνομο και αυτοδύναμο cluster. Με αυτόν τον τρόπο το συνολικό σύστημα οδηγείται σε αύξηση του ρυθμού διεκπεραίωσης εργασιών και ανταποκρισιμότητας ενώ το κόστος αυξομειώνεται ανάλογα με τις ανάγκες σε πόρους.

Το Elastic Computing βασίζεται στον μηχανισμό εξοικονόμησης ενέργειας (power saving) του Slurm [24]. Οι κόμβοι που δεν χρησιμοποιούνται αναστέλλονται έως ότου κάποια εργασία ανατεθεί σε αυτούς. Για την αποφυγή απότομης αύξησης του ρεύματος, που είναι λογικό να συμβαίνει όταν αναστέλλονται και επανεκκινούνται μεγάλες ομάδες κόμβων ταυτόχρονα, το Slurm αυξάνει σταδιακά την ισχύ των μηχανημάτων. Αυτό βέβαια απαιτεί από τον πυρήνα του λειτουργικού συστήματος την υποστήριξη CPU throttling. Η πλήρης παραμετροποίηση που απαιτείται για την υποστήριξη αυτής της μεθόδου είναι προσβάσιμη στο [25].

Κεφάλαιο 6

Υλοποίηση Επέκτασης Χρονοδρομολόγησης

Όπως έχουμε αναφέρει και προηγουμένως, το Slurm προσφέρει πληθώρα δυνατοτήτων παραμετροποίησης ενώ επίσης είναι εύκολα επεκτάσιμο με τον μηχανισμό των plugins. Σε αυτό το κεφάλαιο θα αναπτυχθεί μια επέκταση χρονοδρομολογητή (scheduler plugin) μέσω της οποίας θα γίνει παρέμβαση στη διαδικασία χρονοδρομολόγησης εργασιών.

Αρχικά, αναφέρονται τα χαρακτηριστικά του συστήματος στο οποίο έγινε η αξιολόγηση της χρονοδρομολόγησης. Έπειτα, περιγράφονται τα είδη εργασιών τα οποία χρησιμοποιήθηκαν και ο τρόπος υποβολής τους στο σύστημα ώστε να δημιουργηθεί μια τεχνητή ουρά εργασιών (job queue). Τέλος, εκτελούμε το σενάριο χρονοδρομολόγησης με το `builtin` scheduler plugin και ύστερα με το δικό μας για να γίνει η ποιοτική σύγκριση της σειράς χρονοδρομολόγησης των εργασιών.

6.1 Το σύστημα

Το σύστημα που χρησιμοποιήθηκε για την αξιολόγηση της επέκτασης αποτελείται από έναν κόμβο ο οποίος εκτελεί τον *slurmctld* δαίμονα, έναν που εκτελεί τον *slurmdbd* δαίμονα και 8 κόμβους που εκτελούν τον *slurmd* δαίμονα. Ο καθένας εκ των worker nodes, δηλαδή τους κόμβους που εκτελούν τον *slurmd* και είναι διαθέσιμοι για εκτέλεση εργασιών, απαρτίζεται από 4 επεξεργαστές (CPUs) και 1800 MB μνήμης το οποίο έχει οριστεί μέσω του κεντρικού configuration αρχείου του Slurm (*slurm.conf*). Οι κόμβοι δημιουργήθηκαν με τη χρήση Docker Containers τα οποία αναλύονται στο Παράρτημα Α.

Φυσικά, το σύστημα μπορεί να διαφοροποιηθεί ανάλογα με τις ανάγκες και ο τρόπος με τον οποίο μπορεί να γίνει αυτό περιγράφεται στο Παράρτημα Β.

6.2 Οι εργασίες

Για τη δημιουργία τεχνητού φόρτου εργασίας χρησιμοποιήθηκαν τέσσερα διαφορετικά είδη εργασιών:

- Serial: όπου η εργασία εκτελείται σειριακά σε έναν επεξεργαστή ενός κόμβου.
- OpenMP: όπου η εργασία είναι παράλληλη και μπορεί να χρησιμοποιήσει πολλαπλούς επεξεργαστές οι οποίοι ανήκουν όμως στον ίδιο κόμβο.
- MPI: όπου η εργασία χρησιμοποιεί ανταλλαγή μηνυμάτων για την επίτευξη παραλληλισμού μέσω πολλαπλών tasks τα οποία εκτελούνται σε διαφορετικό επεξεργαστή και ίσως σε διαφορετικό κόμβο.
- Hybrid OpenMP-MPI: όπου το κάθε MPI task μπορεί να χρησιμοποιεί περισσότερους του ενός επεξεργαστές με τη χρήση OpenMP.

Για την αξιολόγηση των σεναρίων εκτέλεσης, θα χρησιμοποιηθεί ένα σύνολο εργασιών που απαρτίζεται από 3 serial jobs, 4 OpenMP jobs, 4 MPI jobs και 9 Hybrid jobs. Στη συνέχεια, περιγράφεται ο τρόπος με τον οποίο θα εκτελεστούν τα σενάρια.

6.3 Μέθοδος εκτέλεσης

Το σύνολο των εργασιών υποβάλλεται στο σύστημα με τη μορφή batch scripts, δηλώνοντας τις απαιτήσεις τους σε πόρους μέσω επιλογών προς την *sbatch* εντολή. Ένα παράδειγμα ενός τέτοιου script είναι το 6.1 όπου γίνεται εμφανές ποιες επιλογές ορίζονται για την εκάστοτε εργασία: ο αριθμός των κόμβων (nodes) και ο αριθμός των διεργασιών (ntasks) που σε συνδυασμό με την επιλογή *–cpus-per-task* καθορίζουν τον συνολικό αριθμό αιτούμενων επεξεργαστών (στο παράδειγμα 16*2 = 32 CPUs). Η σύμβαση για την ονομασία των εργασιών έχει οριστεί να είναι ο συνδυασμός του είδους του προγράμματος εκτέλεσης, του αριθμού των tasks και τον αριθμό των CPUs ανά task.

Για να γίνει σωστή παρατήρηση της σειράς χρονοδρομολόγησης των εργασιών θα πρέπει πρώτα να υποβληθούν όλες οι εργασίες στο σύστημα, με αποτέλεσμα να γεμίσει η ουρά (job queue) και στη συνέχεια να ξεκινήσει ο μηχανισμός της χρονοδρομολόγησης. Για να επιτευχθεί αυτό, θα πρέπει αρχικά να απενεργοποιηθούν οι κόμβοι του συστήματος θέτοντας τις διαμερίσεις στις οποίες ανήκουν σε κατάσταση DOWN [26]. Εφόσον υποβάλλουμε το σύνολο των εργασιών, επαναφέρουμε τις διαμερίσεις σε κατάσταση UP για να γίνουν διαθέσιμοι οι πόροι στο σύστημα και να ξεκινήσει η κατανομή τους. Αξίζει να σημειωθεί ότι η σειρά με την οποία υποβάλλονται οι εργασίες στο σύστημα γίνεται με τυχαίο τρόπο σε κάθε σενάριο εκτέλεσης.

Επίσης, καθώς ο χρόνος υποβολής (SubmitTime) της εκάστοτε εργασίας είναι με ακρίβεια δευτερολέπτου, οι εργασίες υποβάλλονται με διαφορά ενός δευτερολέπτου ώστε να είναι εμφανής η σειρά με την οποία εισήλθαν στο σύστημα.

```
#!/usr/bin/env bash

#SBATCH --job-name=HYBRID_T16_C2

#SBATCH --output=job_%j.out
#SBATCH --error=job_%j.err

#SBATCH --nodes 8

#SBATCH --ntasks 16

#SBATCH --cpus-per-task 2

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mpirun -np $SLURM_NTASKS /home/src/hybrid_openmp_mpi
```

Κώδικας 6.1: Μορφή ενός batch script

6.4 Εκτέλεση με sched/builtin

Η χρήση του **sched/builtin** ως scheduler plugin, οδηγεί σε χρονοδρομολόγηση εργασιών αποκλειστικά βάσει του χρόνου υποβολής τους στο σύστημα (SubmitTime) σε μορφή First In First Out (FIFO) ή ισοσδύναμα First Come First Served (FCFS).

Τα αποτελέσματα της χρονοδρομολόγησης με αυτό το plugin ενεργοποιημένο παρουσιάζονται στον πίνακα 6.1, τα οποία εξήχθησαν με τη βοήθεια της εντολής *sacct* σε συνδυασμό με κατάλληλες επιλογές για την εμφάνιση ιδιοτήτων που μας ενδιαφέρουν. Παρατηρούμε ότι ανεξάρτητα των περιορισμών που έχει θέσει η κάθε εργασία για κόμβους και επεξεργαστές, η σειρά κατανομής πόρων σε αυτές (ισοδύναμα το StartTime) γίνεται βάσει του SubmitTime.

6.5 Εκτέλεση με sched/thesis

Με τη βοήθεια της αυστηρά ορισμένης διεπαφής για επεκτάσεις χρονοδρομολόγησης, κατασκευάστηκε μια επέκταση χρονοδρομολόγησης (scheduler plugin) μέσω της οποίας είναι εφικτή η παρέμβαση στο μηχανισμό επιλογής της επόμενης εργασίας για εκτέλεση στο σύστημα. Η επέκταση ονομάστηκε «*sched/thesis*» τηρώντας τον κανόνα ονοματοδοσίας που επισημαίνεται στην ενότητα 4.5.

Όπως έχει αναφερθεί, το Slurm εκτός της επέκτασης χρονοδρομολόγησης που εκτελείται ανά μία ορισμένη περίοδο (interval) έχει και έναν εσωτερικό μηχανισμό χρονοδρομολόγησης που βασίζεται σε γεγονότα για την εκτέλεσή του. Σκοπός της επέκτασης που δημιουργήθηκε είναι ο πλήρης έλεγχος του μηχανισμού χρονοδρομολόγησης, επομένως ο εσωτερικός μηχανισμός απενεργοποιήθηκε. Για να

JobId	JobName	Nodes	CPUs	SubmitTime	StartTime
2	HYBRID_T16_C2	8	32	12:57:01	13:05:30
3	OPENMP_T1_C4	1	4	12:57:02	13:05:32
4	MPI_T16_C1	4	16	12:57:04	13:05:32
5	HYBRID_T4_C4	4	16	12:57:05	13:05:47
6	MPI_T24_C1	6	24	12:57:06	13:05:48
7	OPENMP_T1_C3	1	3	12:57:08	13:05:48
8	HYBRID_T3_C3	3	9	12:57:09	13:06:11
9	SERIAL_T1_C1	1	1	12:57:11	13:06:11
10	HYBRID_T8_C4	8	32	12:57:12	13:06:39
11	HYBRID_T8_C2	4	16	12:57:13	13:06:39
12	OPENMP_T1_C2	1	2	12:57:15	13:06:39
13	HYBRID_T3_C4	3	12	12:57:16	13:06:39
14	SERIAL_T1_C1	1	1	12:57:18	13:06:40
15	HYBRID_T2_C4	2	8	12:57:19	13:06:40
16	OPENMP_T1_C1	1	1	12:57:20	13:06:40
17	HYBRID_T5_C4	5	20	12:57:22	13:06:41
18	MPI_T32_C1	8	32	12:57:23	13:07:22
19	SERIAL_T1_C1	1	1	12:57:25	13:07:36
20	HYBRID_T5_C2	5	10	12:57:26	13:07:36
21	MPI_T4_C1	4	4	12:57:27	13:07:37

Πίνακας 6.1: Αποτέλεσμα χρονοδρομολόγησης με `sched/builtin`

είναι εμφανής η σειρά με την οποία εκκινήθηκαν οι εργασίες, η επέκταση έχει παραμετροποιηθεί έτσι ώστε να χρονοδρομολογεί μόνο μία εργασία ανά περίοδο εκτέλεσης.

Σε κάθε περίοδο εκτέλεσης της χρονοδρομολόγησης, ανακτάται ολόκληρη η ουρά εργασιών και ταξινομείται βάσει των απαιτήσεών τους σε υπολογιστικούς πόρους. Στον πίνακα 6.2 εμφανίζονται τα αποτελέσματα εκτέλεσης του ίδιου συνόλου εργασιών όπως με το *sched/builtin*. Παρατηρούμε ότι πλέον η χρονοδρομολόγηση δεν γίνεται με πρωταρχικό κριτήριο τον χρόνο υποβολής (SubmitTime), αλλά τους απαιτούμενους υπολογιστικούς κόμβους (nodes). Φυσικά, το κριτήριο αυτό μπορεί να μεταβληθεί μέσω παραμετροποίησης και αναφέρεται αναλυτικά στην επόμενη υποενότητα.

6.5.1 Παραμετροποίηση επέκτασης

Έχει δημιουργηθεί μια νέα παράμετρος στο κεντρικό configuration αρχείο του Slurm (*slurm.conf*), με όνομα *ThesisParameters*, μέσω της οποίας μπορεί να παραμετροποιηθεί το κριτήριο με το οποίο θα χρονοδρομολογηθούν οι εργασίες. Η παράμετρος αυτή διαβάζεται κατά την εκτέλεση της χρονοδρομολόγησης και επηρεάζει την ταξινόμηση της ουράς εργασιών. Η λογική της ταξινόμησης

JobId	JobName	Nodes	CPUs	SubmitTime	StartTime
2	SERIAL_T1_C1	1	1	14:13:12	14:14:42
3	OPENMP_T1_C1	1	1	14:13:13	14:14:44
4	HYBRID_T16_C2	8	32	14:13:14	14:16:34
5	OPENMP_T1_C4	1	4	14:13:16	14:14:54
6	MPI_T32_C1	8	32	14:13:17	14:16:36
7	OPENMP_T1_C3	1	3	14:13:18	14:14:52
8	MPI_T24_C1	6	24	14:13:20	14:16:16
9	HYBRID_T3_C4	3	12	14:13:21	14:15:00
10	SERIAL_T1_C1	1	1	14:13:23	14:14:46
11	HYBRID_T8_C4	8	32	14:13:24	14:16:52
12	OPENMP_T1_C2	1	2	14:13:25	14:14:50
13	HYBRID_T8_C2	4	16	14:13:27	14:15:22
14	HYBRID_T3_C3	3	9	14:13:28	14:14:58
15	HYBRID_T5_C4	5	20	14:13:29	14:16:12
16	SERIAL_T1_C1	1	1	14:13:31	14:14:48
17	MPI_T16_C1	4	16	14:13:32	14:15:24
18	HYBRID_T2_C4	2	8	14:13:34	14:14:56
19	MPI_T4_C1	4	4	14:13:35	14:15:02
20	HYBRID_T4_C4	4	16	14:13:37	14:15:48
21	HYBRID_T5_C2	5	10	14:13:38	14:16:10

Πίνακας 6.2: Αποτέλεσμα χρονοδρομολόγησης με `sched/thesis`

παρουσιάζεται στον κώδικα 6.2. Συγκεκριμένα, ο διαχειριστής μπορεί να ορίσει την προτεραιότητα διαφορετικών περιορισμών των εργασιών χωρίζοντάς τες με το σύμβολο «:». Οι επιλογές που μπορούν να οριστούν είναι οι εξής:

- nodes (n): αριθμός αιτούμενων κόμβων
- cpus (c): αριθμός αιτούμενων επεξεργαστών
- memory (m): αριθμός αιτούμενης μνήμης
- submit time (t): χρόνος υποβολής της εργασίας

Για παράδειγμα, στην περίπτωση που έχει οριστεί `ThesisParameters=n:c:t`, θα χρονοδρομολογηθούν πρώτες οι εργασίες που απαιτούν λιγότερους υπολογιστικούς κόμβους. Αν υπάρξουν ισοπαλίες τότε το επόμενο κριτήριο θα είναι ο αριθμός των απαιτούμενων επεξεργαστών. Στην περίπτωση που υπάρξουν εργασίες με ίδιο αριθμό κόμβων και επεξεργαστών, το τελικό κριτήριο επιλογής θα είναι ο χρόνος υποβολής.

Στον πίνακα 6.3 παρουσιάζονται τα αποτελέσματα εκτέλεσης για διαφορετικές τιμές της παραμέτρου *ThesisParameters*. Στην κάθε περίπτωση, τα αποτελέσματα

είναι ταξινομημένα βάσει του χρόνου εκκίνησης της εργασίας ώστε να είναι εμφανή τα κριτήρια με τα οποία έγινε η χρονοδρομολόγηση.

```
static int composite_comparator(void *x, void *y)
{
  job_queue_rec_t *job_rec1 = *(job_queue_rec_t **) x;
  job_queue_rec_t *job_rec2 = *(job_queue_rec_t **) y;

  char* temp_thesis_params = xmalloc(sizeof(*thesis_params));
  strcpy(temp_thesis_params, thesis_params);

  char *token = strtok(temp_thesis_params, ":");

  int result = 0;

  while (token != NULL)
  {
    switch (token[0])
    {
      case (int) 'n':
        result = node_comparator(job_rec1, job_rec2);
        break;
      case (int) 'c':
        result = cpu_comparator(job_rec1, job_rec2);
        break;
      case (int) 'm':
        result = memory_comparator(job_rec1, job_rec2);
        break;
      case (int) 't':
        result = submit_time_comparator(job_rec1, job_rec2);
        break;
      default:
        result = -1;
    }

    token = strtok(NULL, ":");

    if (result != 0)
      break;
  }

  if (result == 0)
    result = submit_time_comparator(job_rec1, job_rec2);

  xfree(temp_thesis_params);
  return result;
}
```

Κώδικας 6.2: Συγκριτής εργασιών της επέκτασης χρονοδρομολόγησης

JobName	Nodes	CPUs	SubmitTime	JobName	Nodes	CPUs	SubmitTime
SERIAL_T1_C1	1	1	14:13:12	SERIAL_T1_C1	1	1	16:34:33
OPENMP_T1_C1	1	1	14:13:13	SERIAL_T1_C1	1	1	16:34:36
SERIAL_T1_C1	1	1	14:13:23	OPENMP_T1_C1	1	1	16:34:40
SERIAL_T1_C1	1	1	14:13:31	SERIAL_T1_C1	1	1	16:34:52
OPENMP_T1_C2	1	2	14:13:25	OPENMP_T1_C2	1	2	16:34:56
OPENMP_T1_C3	1	3	14:13:18	OPENMP_T1_C3	1	3	16:34:38
OPENMP_T1_C4	1	4	14:13:16	OPENMP_T1_C4	1	4	16:34:30
HYBRID_T2_C4	2	8	14:13:34	MPI_T4_C1	4	4	16:34:31
HYBRID_T3_C3	3	9	14:13:28	HYBRID_T2_C4	2	8	16:34:34
HYBRID_T3_C4	3	12	14:13:21	HYBRID_T3_C3	3	9	16:34:49
MPI_T4_C1	4	4	14:13:35	HYBRID_T5_C2	5	10	16:34:53
HYBRID_T8_C2	4	16	14:13:27	HYBRID_T3_C4	3	12	16:34:48
MPI_T16_C1	4	16	14:13:32	HYBRID_T8_C2	4	16	16:34:45
HYBRID_T4_C4	4	16	14:13:37	MPI_T16_C1	4	16	16:34:47
HYBRID_T5_C2	5	10	14:13:38	HYBRID_T4_C4	4	16	16:34:55
HYBRID_T5_C4	5	20	14:13:29	HYBRID_T5_C4	5	20	16:34:41
MPI_T24_C1	6	24	14:13:20	MPI_T24_C1	6	24	16:34:44
HYBRID_T16_C2	8	32	14:13:14	HYBRID_T16_C2	8	32	16:34:37
MPI_T32_C1	8	32	14:13:17	HYBRID_T8_C4	8	32	16:34:42
HYBRID_T8_C4	8	32	14:13:24	MPI_T32_C1	8	32	16:34:51

JobName	Nodes	CPUs	SubmitTime	JobName	Nodes	CPUs	SubmitTime
SERIAL_T1_C1	1	1	16:55:59	MPI_T16_C1	4	16	16:46:03
SERIAL_T1_C1	1	1	16:56:02	HYBRID_T5_C2	5	10	16:46:04
SERIAL_T1_C1	1	1	16:56:06	HYBRID_T3_C3	3	9	16:46:06
OPENMP_T1_C1	1	1	16:56:14	HYBRID_T16_C2	8	32	16:46:07
OPENMP_T1_C2	1	2	16:56:20	HYBRID_T8_C4	8	32	16:46:08
OPENMP_T1_C3	1	3	16:56:16	SERIAL_T1_C1	1	1	16:46:10
MPI_T4_C1	4	4	16:55:58	HYBRID_T4_C4	4	16	16:46:11
OPENMP_T1_C4	1	4	16:56:22	MPI_T32_C1	8	32	16:46:13
HYBRID_T2_C4	2	8	16:56:03	OPENMP_T1_C1	1	1	16:46:14
HYBRID_T3_C3	3	9	16:56:09	MPI_T24_C1	6	24	16:46:15
HYBRID_T5_C2	5	10	16:56:11	MPI_T4_C1	4	4	16:46:17
HYBRID_T3_C4	3	12	16:56:17	HYBRID_T5_C4	5	20	16:46:18
HYBRID_T8_C2	4	16	16:56:05	HYBRID_T2_C4	2	8	16:46:20
MPI_T16_C1	4	16	16:56:10	SERIAL_T1_C1	1	1	16:46:21
HYBRID_T4_C4	4	16	16:56:21	SERIAL_T1_C1	1	1	16:46:22
HYBRID_T5_C4	5	20	16:56:07	OPENMP_T1_C3	1	3	16:46:24
MPI_T24_C1	6	24	16:56:13	OPENMP_T1_C4	1	4	16:46:25
HYBRID_T16_C2	8	32	16:55:56	HYBRID_T8_C2	4	16	16:46:26
HYBRID_T8_C4	8	32	16:56:00	HYBRID_T3_C4	3	12	16:46:28
MPI_T32_C1	8	32	16:56:18	OPENMP_T1_C2	1	2	16:46:29

Πίνακας 6.3: Σειρά εκτέλεσης βάσει της παραμέτρου *ThesisParameters*

Πάνω αριστερά: `ThesisParameters=n:c:t`
Πάνω δεξιά: `ThesisParameters=c:n:t`
Κάτω αριστερά: `ThesisParameters=c:t`
Κάτω δεξιά: `ThesisParameters=t`

Κεφάλαιο 7

Επίλογος

7.1 Αξιολόγηση

Σε αυτήν τη διπλωματική εργασία έγινε εκτενής αναφορά στο Slurm το οποίο αποτελεί ένα από τα δημοφιλέστερα και ευρέως χρησιμοποιούμενα συστήματα διαχείρισης πόρων. Αναλύθηκε η αρχιτεκτονική του, οι βασικές εντολές του, μερικές από τις προχωρημένες τεχικές χρονοδρομολόγησης εργασιών που προσφέρει, ενώ στο τέλος αναπτύχθηκε μια νέα επέκταση χρονοδρομολόγησης. Η επέκταση αυτή αναλαμβάνει την επιλογή της επόμενης εργασίας για εκτέλεση από την ουρά εργασιών, βάσει κριτηρίων προτεραιότητας που έχουν οριστεί σε παράμετρο του κεντρικού αρχείου παραμετροποίησης του συστήματος (*slurm.conf*). Προσφέρεται η δυνατότητα αλλαγής του κριτηρίου αυτού κατά την εκτέλεση (runtime), δίχως να χρειάζεται επανεκκίνηση του συστήματος. Τέλος, εκτελέστηκε σενάριο με ένα σύνολο εργασιών διαφορετικού είδους ώστε να γίνει μια ποιοτική σύγκριση μεταξύ των `sched/builtin` και `sched/thesis`.

Αποδείχθηκε ότι το Slurm αποτελεί ένα εύκολα επεκτάσιμο σύστημα, με επεκτάσεις που δύναται να επηρεάζουν διάφορες πτυχές του συστήματος ανάλογα με τις εκάστοτε ανάγκες.

7.2 Μελλοντικές Επεκτάσεις

Ως περαιτέρω έρευνα, θα μπορούσε και θα είχε ενδιαφέρον να αναπτυχθεί μια επέκταση για το Slurm η οποία θα επηρεάζει τον τρόπο με τον οποίο γίνεται η επιλογή των υπολογιστικών πόρων από τους συνολικά διαθέσιμους του συστήματος. Πρόκειται, δηλαδή, για μια επέκταση τύπου `select`. Κατά την υποβολή εργασιών θα μπορούσε ο χρήστης να «μαρκάρει» την εργασία με ένα ειδικό χαρακτηριστικό (memory intensive, cpu intensive κ.α.) για να γίνει η κατάλληλη επιλογή κόμβων. Φυσικά, θα μπορούσε να συνδυαστεί και με ένα ειδικά σχεδιασμένο scheduler plugin το οποίο θα συνεργάζεται με το selection plugin για καλύτερα αποτελέσματα.

Βιβλιογραφία

[1] T. Sterling, M. Anderson, and M. Brodowicz, Eds., *High Performance Computing*. Morgan Kaufmann, 2018.

[2] S. Iqbal, R. Gupta, and Y. Fang, "Planning Considerations for Job Scheduling in HPC Clusters," *Dell Power Solutions, Tech. Rep.*, pp. 133–136, 2005.

[3] "Slurm SchedMD," 2018. [Online]. Available: https://slurm.schedmd.com

[4] M. Hovestadt, O. Kao, A. Keller, and A. Streit, "Scheduling in HPC Resource Management Systems: Queuing vs. Planning," vol. 2862, 06 2003, pp. 1–20.

[5] M. Flynn, *Flynn's Taxonomy*. Boston, MA: Springer US, 2011, pp. 689–697.

[6] Tanenbaum, Andrew S., *Structured Computer Organization (5th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[7] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, "High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions," *Computing in Science & Engineering*, vol. 7, pp. 51– 59, 04 2005.

[8] T. E. Anderson, D. E. Culler, and D. Patterson, "A case for NOW (Networks of Workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54–64, Feb 1995.

[9] "Top 500. The List," 1993-2016. [Online]. Available: https://top500.org

[10] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A Parallel Workstation For Scientific Computation," in *In Proceedings of the 24th International Conference on Parallel Processing*. CRC Press, 1995, pp. 11–14.

[11] M. Baker and R. Buyya, "Cluster computing: the commodity supercomputer," *Software: Practice and Experience*, vol. 29, no. 6, pp. 551–576, 1999.

[12] D. Eadline, *High Performance Computing for Dummies*. Wiley Publishing, Inc., 2009.

[13] "XSEDE: Extreme Science and Discovery Environment," 2011. [Online]. Available: https://xsede.org

[14] M. Jette, A. Yoo, and M. Grondona, "SLURM: Simple linux utility for resource management," *Lecture Notes in Computer Science*, July 2003.

[15] Altair Engineering, "PBS Professional Open Source Project." [Online]. Available: https://www.pbspro.org/

[16] Wikipedia, "Moab Cluster Suite." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Moab_Cluster_Suite

[17] Wikipedia, "IBM Tivoli Workload Scheduler." [Online]. Available: https://en.wikipedia.org/w/index.php?title=IBM_Tivoli_Workload_Scheduler

[18] Univa, "Grid Engine." [Online]. Available: http://www.univa.com/products/

[19] Vavilapalli, Murthy, Douglas, Agarwal, Konar, Evans, Graves, Lowe, Shah, Seth, Saha, Curino, O'Malley, Radia, Reed, and Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[20] N. Freed, J. Klensin, and T. Hansen, "Media Type Specifications and Registration Procedures," Internet Requests for Comments, RFC Editor, BCP 13, January 2013.

[21] "Slurm Gang Scheduling," 2018. [Online]. Available: https://slurm.schedmd.com/gang_scheduling.html

[22] "Slurm Preemption," 2018. [Online]. Available: https://slurm.schedmd.com/preempt.html

[23] "Slurm Generic Resources Scheduling," 2018. [Online]. Available: https://slurm.schedmd.com/gres.html

[24] "Slurm Power Saving," 2018. [Online]. Available: https://slurm.schedmd.com/power_save.html

[25] "Slurm Elastic Computing," 2018. [Online]. Available: https://slurm.schedmd.com/elastic_computing.html

[26] "Slurm FAQ," 2019. [Online]. Available: https://slurm.schedmd.com/faq.html

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[28] T. Bui, "Analysis of Docker Security, arXiv:1501.02967v1," Jan 2011.

[29] A. Grattafiori, "Understanding and hardening linux containers," *Whitepaper, NCC Group*, 2016.

[30] "Docker Hub," 2019. [Online]. Available: https://hub.docker.com/

Παράρτημα Α΄

Docker και Εικονικοποίηση

Το Docker είναι ένα λογισμικό ανοιχτού κώδικα το οποίο αυτοματοποιεί την ανάπτυξη εφαρμογών μέσω των «containers». Τα Docker containers περικλείουν κομμάτια λογισμικού μαζί με ένα πλήρες σύστημα αρχείων που περιέχει ό,τι είναι απαραίτητο για να τρέξει το λογισμικό: αρχεία κώδικα, βιβλιοθήκες αλλά και ολόκληρα συστήματα τα οποία απαιτεί το λογισμικό για να δουλέψει όπως βάση δεδομένων, application server κλπ. Έτσι λοιπόν εγγυάται ότι το λογισμικό μέσα στο container θα εκτελεστεί επιτυχώς ανεξαρτήτως από το περβάλλον στο οποίο θα γίνει αυτό.
Πρωτού όμως αναλύσουμε το οικοσύστημα του Docker θα πρέπει να αναφερθούμε στον πιο γενικό όρο της εικονικοποίησης (Virtualization).

Α΄.1 Εικονικοποίηση

Η εικονικοποίηση, γενικά, αποτελεί μία πρακτική για την υποδιαίρεση υπολογιστικών πόρων (hardware) σε επιμέρους περιβάλλοντα [27]. Παγκόσμιοι κολοσσοί όπως η Amazon, η Microsoft και η Google χρησιμοποιούν τεχνικές εικονικοποίησης για να προσφέρουν στους πελάτες τους υποδομή ως υπηρεσία (Infrastructure as a Service - IaaS) [28]. Η εικονικοποίηση μπορεί να εφαρμοστεί τόσο στην πλευρά του διακομιστή (server side) όσο και στην πλευρά του πελάτη (client side). Η server side εικονικοποίηση μπορεί να διαχωριστεί σε 2 βασικές κλάσεις [28]:

- βασισμένη σε hypervisor (hypervisor based virtualization)
- βασισμένη σε containers (container based virtualization)

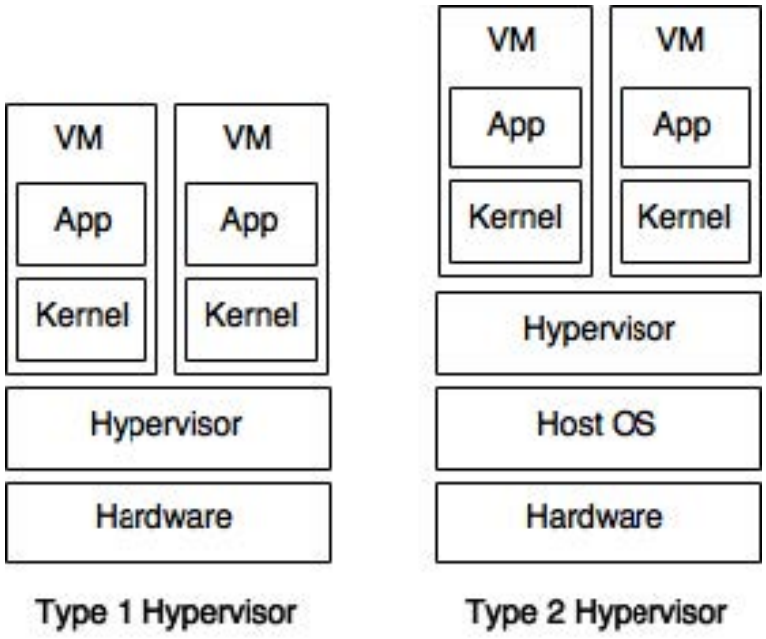
Η εικονικοποίηση με hypervisor στηρίζεται σε ένα λογισμικό που ονομάζεται το ίδιο hypervisor, η δουλειά του οποίου είναι να προσφέρει ένα επίπεδο αφαιρετικότητας στους πόρους του συστήματος μέσω των εικονικών μηχανών (virtual machines). Η εικονικοποίηση με containers ορίζει την έννοια των containers, οι οποίοι

χρησιμοποιούνται για να απομονώσουν την εκτέλεση διαφορετικών εφαρμογών ενώ τρέχουν στον ίδιο πυρήνα λειτουργικού συστήματος (ΛΣ) [28].

Α΄.1.1 Εικονικοποίηση με Hypervisor

Η εικονικοποίηση που βασίζεται σε ένα λογισμικό ελέγχου, τον «επόπτη» (hypervisor) επιτρέπει τη δημιουργία ολοκληρωμένων εικονικών μηχανών (VMs). Τα VMs αυτά περιέχουν το δικό τους λειτουργικό σύστημα, με το δικό τους πυρήνα, εξαρτήσεις και εφαρμογές [28]. Ανάλογα με τον τρόπο που εκτελείται ο hypervisor τους διαχωρίζουμε σε δύο μεγάλες κατηγορίες: *Τύπου 1* και *Τύπου 2*.

Οι τύπου 1 ή όπως αποκαλούνται *bare metal hypervisors* εκτελούνται απευθείας στο υλικό του συστήματος. Παραδείγματα τέτοιου τύπου αποτελούν οι VMWare ESX, XEN και Microsoft Hyper-V. Οι τύπου 2 ή αλλιώς *hosted hypervisors* χρειάζονται ένα υπάρχων λειτουργικό σύστημα (host) για να εκτελεστούν.



Σχήμα Α΄.1: Τύποι Hypervisor

Χρησιμοποιώντας hypervisors τύπου 2 υπάρχουν δύο διαφορετικές τεχνικές εικονικοποίησης οι οποίες αναλύονται παρακάτω:

- **Πλήρης εικονικοποίηση**: Το βασικότερο χαρακτηριστικό αυτής της τεχνικής είναι ότι το λειτουργικό σύστημα μπορεί να τρέξει μέσα στην εικονική μηχανή χωρίς καμία τροποποίηση, ακριβώς όπως τρέχει και στο πραγματικό μηχάνημα. Η εικονική μηχανή δεν ξέρει ότι τρέχει σε εικονικό περιβάλλον και ότι το υλικό της δεν είναι πραγματικό αλλά προσομοιώνεται

μέσω κατάλληλου λογισμικού. Το κυριότερο πλεονέκτημα είναι η ευκολία στη δημιουργία εικονικών μηχανών, αλλά το γεγονός ότι τα πάντα προσομοιώνονται μέσω λογισμικού καθιστά αυτήν την τεχνική αργή.

- **Παραεικονικοποίηση**: Σε αυτή την περίπτωση η εικονική μηχανή έχει επίγνωση ότι τρέχει σε εικονικό περιβάλλον και συνεργάζεται με τον host ώστε να επιτευχθεί καλύτερη επίδοση. Γνωρίζει ότι το υλικό της δεν είναι πραγματικό. Για το λόγο αυτό δε μπορεί να χρησιμοποιηθεί ο ίδιος πυρήνας του λειτουργικού συστήματος που εκτελείται στο πραγματικό μηχάνημα. Χρειάζεται να γίνουν κάποιες τροποποιήσεις ώστε ένα λειτουργικό σύστημα να εκτελεστεί σε μία εικονική μηχανή που χρησιμοποιεί παραεικονικοποίηση. Ωστόσο η επίδοση είναι αρκετά καλύτερη σε σχέση με την πλήρη εικονικοποίηση.

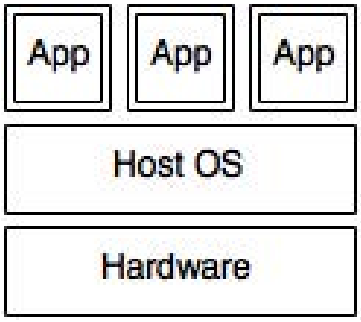
Α΄.1.2 Εικονικοποίηση με Containers

Η εικονικοποίηση με containers ή αλλιώς *containerization* χρησιμοποιεί ορισμένες λειτουργίες του πυρήνα του λειτουργικού συστήματος για να απομονώσει ομάδες από διεργασίες μεταξύ τους [29] κάτι που καθιστά τη βάση για τεχνολογίες όπως το Docker. Αυτά τα απομονωμένα περιβάλλοντα αποκαλούνται *containers*. Δημιουργούνται χρησιμοποιώντας ένα συνδυασμό από επιμέρους χαρακτηριστικά του πυρήνα όπως είναι τα kernel namespaces, τα cgroups καθώς επίσης και τα root capabilities που αυτός προσφέρει. Σε αντίθεση με τις εικονικές μηχανές, τα containers μοιράζονται ένα κοινό λειτουργικό σύστημα οπότε δε χρειάζεται η ύπαρξη ενός hypervisor. Η εικόνα Α΄.2 δείχνει πολλαπλές εφαρμογές που η καθεμία εκτελείται στον δικό της container και μοιράζονται το ίδιο υλικό αλλά και λειτουργικό σύστημα.Τεχνολογίες containers όπως τα Docker, Rkt και LXC (Linux Containers) είναι μερικές απο τις υλοποιήσεις αυτής της μεθόδου εικονικοποίησης.

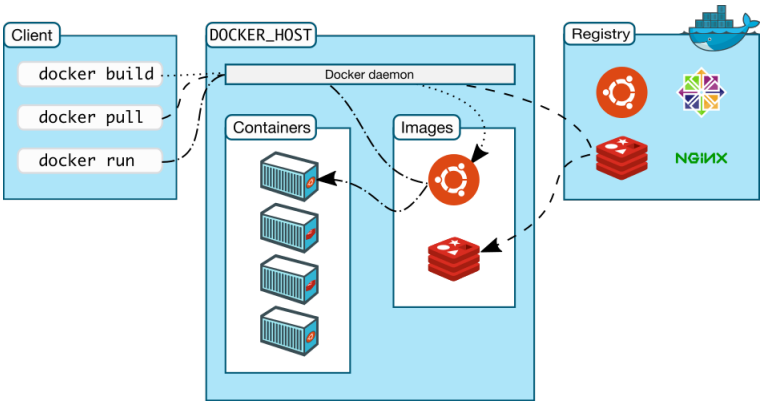
Συγκριτικά με την εικονικοποίηση με hypervisor, η χρήση των containers οδηγεί στη δημιουργία μικρότερων σε μέγεθος εφαρμογών εφόσον δε χρειάζεται η παρουσία ενός ολόκληρου ΛΣ. Αυτό φυσικά οφελεί και τις επιδόσεις της εφαρμογής και τους χρόνους εκκίνησής της αφού τρέχει απευθείας στον πυρήνα του υποκείμενου συστήματος και δε χρειάζεται η ενδιάμεση μετάφραση των κλήσεων συστήματος που αναλάμβανε ο hypervisor. Γι΄ αυτό λόγο σε περιπτώσεις όπου η επίδοση είναι καθοριστικός παράγοντας προτιμάται η εικονικοποίηση μέσω containers [29, 28].

Α΄.2 Αρχιτεκτονική του Docker

Υπάρχουν τέσσερα βασικά στοιχεία από τα οποία αποτελείται το Docker: Docker Client και Server, Docker Images, Docker Registries και Docker Containers. Πάνω σε αυτά τα βασικά στοιχεά βασίζεται η συνολική αρχιτεκτονική του Docker η οποία παρουσιάζεται γραφικά στο Σχήμα Α΄.3.



Σχήμα Α΄.2: Εικονικοποίηση μέσω Containers



Σχήμα Α΄.3: Αρχιτεκτονική του Docker

Docker Client και Server

Το Docker ακολουθεί μία Client-Server αρχιτεκτονική. Ο Docker Server ή αλλιώς Daemon δέχεται αιτήματα απο τους Docker Clients μέσω μίας διεπαφής προγραμματισμού εφαρμογών (API) και δρα αναλόγως. Αυτό επιτρέπει στον εκάστοτε Docker Client να τρέχει σε διαφορετικό μηχάνημα από το οποίο εκτελείται ο Docker Server χωρίς αυτό φυσικά να είναι απαραίτητο. Ο Daemon είναι αυτός που διαχειρίζεται τις επιμέρους οντότητες όπως είναι τα containers, images και δίκτυα.

Docker Image

Μια εικόνα Docker (Docker Image) αποτελεί ένα read-only πρότυπο το οποίο μπορεί να χρησιμοποιηθεί για να δημιουργηθεί ένα container. Για να δημιουργηθεί μία τέτοια εικόνα χρησιμοποιείται ένα Dockerfile το οποίο δουλεύει σαν «συνταγή» με οδηγίες σε συνδυασμό με την εντολή *docker build*. Για παράδειγμα μια εικόνα θα μπορούσε να περιέχει ένα λειτουργικό σύστημα Ubuntu, μαζί με έναν Apache Web Server για να τρέξει μία web εφαρμογή. Το Docker προσφέρει έναν εύκολο τρόπο να φτιάχνει κανείς τις δικές του εικόνες, αλλά και να κατεβάζει έτοιμες εικόνες από

κάποιο Docker Registry καθώς επίσης και να τις τροποποιεί για τις δικές του ανάγκες.

Docker Registry και Containers

Τα Docker registries αποτελούν τις «αποθήκες» εικόνων, οι οποίες μπορεί να είναι είτε δημόσιες είτε ιδιωτικές και μπορεί κανείς να ανεβάσει ή να κατεβάσει εικόνες Docker. Το Docker Hub [30] είναι το δημόσιο Docker registry και εκεί ένας χρήστης μπορεί να βρει πληθώρα εικόνων που έχουν δημιουργηθεί από άλλους χρήστες και να τις χρησιμοποιήσει για να δημιουργήσει ένα docker container.

Για παράδειγμα, σε ένα project μπορεί κάποιος να χρειάζεται μια βάση δεδομένων. Χρησιμοποιώντας το Docker Hub μπορεί να βρει μια εικόνα βάσης δεδομένων όπως η MySQL, να την κατεβάσει με την εντολή `docker pull mysql` και να δημιουργήσει έτσι ένα container από αυτήν την εικόνα.

Τα Docker containers μοιάζουν με ένα φάκελο που περιέχει όλα όσα χρειάζονται για την εκτέλεση της εφαρμογής που φιλοξενεί και δημιουργείται από μια εικόνα (Docker Image). Τα containers μπορεί κανείς να τα τρέξει, να τα ξεκινήσει, να τα σταματήσει, να τα μεταφέρει ή να τα διαγράψει. Κάθε container είναι ένα απομονωμένο και ασφαλές περιβάλλον εφαρμογής.

Dockerfile

Το Dockerfile είναι ένα αρχείο κειμένου το οποίο περιέχει οδηγίες για την κατασκευή μιας Docker εικόνας. Τέτοιες οδηγίες μπορεί να είναι ο ορισμός της βασικής εικόνας (`FROM`), η εκτέλεση μιας εντολής μέσα στην εικόνα (`RUN`), η έκθεση μιας πόρτας στο δίκτυο (`EXPOSE`), η προσάρτηση ενός μέσου αποθήκευσης (`VOLUME`) ή ο ορισμός της εντολής που θα εκτελεστεί κατά την εκκίνηση ενός container με αυτήν την εικόνα (`CMD`).

Για παράδειγμα, στον κώδικα Α΄.1 παρουσιάζεται ένα Dockerfile για μια εφαρμογή Python η οποία βασίζεται στην επίσημη εικόνα python με το tag *3.4-alpine*. Αρχικά, αντιγράφει τον φάκελο εργασίας στο φάκελο /code της εικόνας και τον θέτει ως ενεργό. Στη συνέχεια, εκτελεί την εντολή `pip install -r requirements.txt` μέσα στην εικόνα για να εγκαταστήσει τα προαπαιτούμενα πακέτα, θεωρώντας ότι υπάρχει το αρχείο *requirements.txt* στον φάκελο /code. Τέλος, ορίζει την εντολή που θα εκτελεστεί όταν ξεκινήσει ένα container από αυτήν την εικόνα.

```
FROM python:3.4-alpine

ADD . /code
WORKDIR /code

RUN pip install -r requirements.txt

CMD ["python", "app.py"]
```

Κώδικας Α.1: Παράδειγμα ενός Dockerfile

Παράρτημα Β΄

Περιβάλλον Ανάπτυξης

Στο παράρτημα αυτό αναλύεται ο τρόπος με τον οποίο μπορεί κάποιος να δοκιμάσει σε τοπικό περιβάλλον το Slurm με χρήση του Docker και να αναπαράγει τα αποτελέσματα που παρουσιάστηκαν στο Κεφάλαιο 6.

Ο κώδικας της επέκτασης που υλοποιήθηκε είναι διαθέσιμος και στο [`https://github.com/alexgialidis/slurm-scheduler-plugin`].

Αρχικά, πρέπει να γίνει εγκατάσταση του Docker. Σε λειτουργικό σύστημα Ubuntu αυτό γίνεται εύκολα με την εντολή:

```
$ snap install docker
```

Για να μη χρειάζεται η χρήση `sudo` για την εκτέλεση εντολών `docker`, μπορεί να δημιουργηθεί ένα νέο UNIX group «*docker*» και να προστεθεί σε αυτό ο χρήστης του συστήματος. Προκειμένου να λάβουν χώρα οι αλλαγές, θα πρέπει να γίνει επανεκκίνηση του συστήματος.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Από εδώ και στο εξής, θεωρούμε ότι ο φάκελος εργασίας ονομάζεται `workspace`. Στη συνέχεια, κατεβάζουμε την έκδοση 18.08.6.2 του Slurm:

```
[~/workspace] $ git clone https://github.com/SchedMD/slurm.git
[~/workspace] $ cd slurm
[~/workspace/slurm] $ git checkout tags/slurm-18-08-6-2
```

Στον φάκελο `workspace`, δημιουργούμε το αρχείο Dockerfile (μπορεί να βρεθεί στο repository) για την κατασκευή της εικόνας (image) βάσει της οποίας θα φτιαχτούν τα containers του Slurm cluster. Κατά το χτίσιμο της εικόνας, λαμβάνεται υπόψη ο κώδικας του Slurm που βρίσκεται στο `workspace/slurm`.

```
[~/workspace] $ docker build -t slurm-docker-cluster:18.08.6 .
```

Στο Dockerfile έχουν προστεθεί τα απαραίτητα πασκαγες για την υποστήριξη προγραμμάτων OpenMP και MPI. Επίσης, έχει οριστεί το UNIX group «*slurm*» στο οποίο ανήκει ο χρήστης `user`.

Για να είναι δυνατή η δημιουργία και διατήρηση των αρχείων μέσα στα Docker containers, είναι απαραίτητο να δημιουργηθεί ένας φάκελος που ονομάσαμε `home`, στον οποίο δικαίωμα εγγραφής θα έχουν όλοι οι χρήστες που ανήκουν στο UNIX group «*slurm*» το οποίο επίσης δημιουργούμε.

```
[~/workspace] $ mkdir home
[~/workspace] $ sudo groupadd slurm
[~/workspace] $ sudo chgrp slurm home
[~/workspace] $ sudo chmod g+w home
```

Πλέον, έχει στηθεί όλο το απαραίτητο περιβάλλον και αρκεί να δημιουργηθούν τα containers έχοντας ως «συνταγή» το παραπάνω docker image. Για τη δημιουργία τους χρησιμοποιείται το *docker-compose* με το παρακάτω docker-compose.yml αρχείο. Δημιουργείται ένα container για το *slurmctld*, ένα για το *slurmdbd* σε συνδυασμό με τη βάση mysql και όσα *slurmd* επιθυμούμε. Να σημειωθεί πως στο περιβάλλον του *slurmctld* έχει προστεθεί η μεταβλητή `DISPLAY` και το volume `/tmp/.X11-unix` για να επιτρέπεται η εκτέλεση εφαρμογών με γραφικό περιβάλλον εσωτερικά των containers. Για να γίνει αυτό, θα πρέπει επίσης να εκτελεστεί η εντολή `xhost local:docker`. Παράδειγμα τέτοιας εφαρμογής είναι το πρόγραμμα *sview* του Slurm, που εμφανίζει την κατάσταση των κόμβων του συστήματος και της ουράς εργασιών.

```
1  version: "2.2"
2  services:
3    mysql:
4      image: mysql:5.7
5      hostname: mysql
6      container_name: mysql
7      environment:
8        MYSQL_RANDOM_ROOT_PASSWORD: "yes"
9        MYSQL_DATABASE: slurm_acct_db
10       MYSQL_USER: slurm
11       MYSQL_PASSWORD: password
12     volumes:
13       - var_lib_mysql:/var/lib/mysql
14
15   slurmdbd:
16     image: slurm-docker-cluster:18.08.6
17     command: ["slurmdbd"]
18     container_name: slurmdbd
19     hostname: slurmdbd
20     volumes:
21       - etc_munge:/etc/munge
22       - etc_slurm:/etc/slurm
23       - var_log_slurm:/var/log/slurm
24       - ./home:/home
25     expose:
26       - "6819"
```

```
27      depends_on:
28        - mysql
29
30    slurmctld:
31      image: slurm-docker-cluster:18.08.6
32      command: ["slurmctld"]
33      container_name: slurmctld
34      hostname: slurmctld
35      volumes:
36        - etc_munge:/etc/munge
37        - etc_slurm:/etc/slurm
38        - slurm_jobdir:/data
39        - var_log_slurm:/var/log/slurm
40        - ./home:/home
41        - /tmp/.X11-unix:/tmp/.X11-unix
42      expose:
43        - "6817"
44      depends_on:
45        - "slurmdbd"
46      environment:
47        - DISPLAY=$DISPLAY
48
49    c1:
50      image: slurm-docker-cluster:18.08.6
51      command: ["slurmd"]
52      hostname: c1
53      container_name: c1
54      volumes:
55        - etc_munge:/etc/munge
56        - etc_slurm:/etc/slurm
57        - slurm_jobdir:/data
58        - var_log_slurm:/var/log/slurm
59        - ./home:/home
60      expose:
61        - "6818"
62      depends_on:
63        - "slurmctld"
64    ...
65    (c2, c3, c4, c5, c6, c7, c8)
66    ...
67  volumes:
68    etc_munge:
69    etc_slurm:
70    slurm_jobdir:
71    var_lib_mysql:
72    var_log_slurm:
```

Κώδικας Β.1: `docker_compose.yml`

Για τη δημιουργία όλων των απαραίτητων containers, αρκεί να εκτελεστεί η παρακάτω εντολή:

```
[~/workspace] $ docker-compose up -d
```

Για την καταγραφή των δεδομένων των εργασιών στο Slurm (job accounting) χρειάζεται να εκτελεστεί το παρακάτω script το οποίο γνωστοποιεί στο slurmdbd την ονομασία του cluster που έχει ορισθεί στο αρχείο *slurm.conf* (ClusterName).

```
[~/workspace] $ docker exec slurmctld bash -c
   "/usr/bin/sacctmgr␣--immediate␣add␣cluster␣name=linux"
[~/workspace] $ docker-compose restart slurmdbd slurmctld
```

Στην περίπτωση που γίνει αλλαγή στην παράμετρο *ThesisParameters* του αρχείου *slurm.conf* που βρίσκεται στο φάκελο `/etc/slurm/`, είναι απαραίτητη η εκτέλεση της εντολής `scontrol reconfigure` στον container που εκτελεί τον slurmctld δαίμονα ώστε να ενημερωθεί η επέκταση χρονοδρομολόγησης.

Για την καταστροφή όλων των containers, των δικτύων και των volumes Που δημηιουργήθηκαν, εκτελούμε την εντολή:

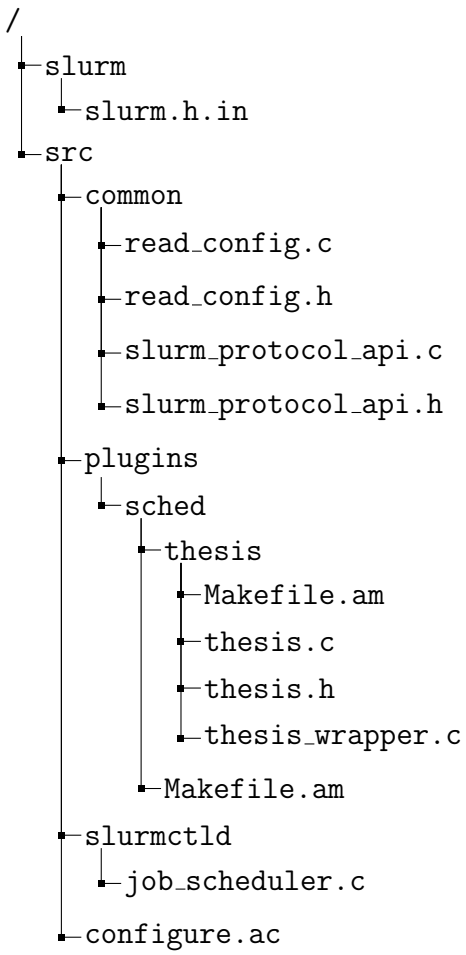
```
[~/workspace] $ docker-compose down -v
```

Αλλαγές αρχείων

Στο σχήμα Β′.1 παραουσιάζεται μια απεικόνιση των αρχείων που μεταβήθηκαν στον κώδικα του Slurm. Ενδεικτικά:

- Στους φακέλους /src/common και /slurm γίνονται αλλαγές για την υποστήριξη της νέας παραμέτρου *ThesisParameters*.
- Ο φάκελος /src/plugins/sched/thesis περιέχει όλα τα αρχεία σχετικά με την νέα επέκταση που δημιουργήθηκε.
- Το αρχείο /src/slurmctld/job_scheduler.c τροποποιήθηκε έτσι ώστε να απενεργοποιηθεί ο εσωτερικός μηχανισμός χρονοδρομολόγησης του Slurm που βασίζεται σε γεγονότα του συστήματος.

Στις επόμενες σελίδες βρίσκεται ο πηγαίος κώδικας των αρχείων αυτών. Όπως αναφέρθηκε και προηγουμένως, τα αρχεία που δημιουργήθηκαν είναι διαθέσιμα και στο [`https://github.com/alexgialidis/slurm-scheduler-plugin`].



Σχήμα Β′.1: Ιεραρχία αρχείων που τροποποιήθηκαν/δημιουργήθηκαν

```
1
...
...
723 extern bool replace_batch_job(slurm_msg_t * msg, void
       *fini_job, bool locked)
724 {
725
...     ...
765     while (0) {

...     ...

987 }
988
...
...
1181 static int _schedule(uint32_t job_limit)
1182 {
1183
...     ...
1564    while (0) {
1565
...     ...
2134 }
2135
...
...
```

Κώδικας Β.2: Τροποποίηση του job_scheduler.c

```
typedef struct slurm_ctl_conf {
  ...
  char *thesis_params;
  ...
} slurmd_status_t;
```

Κώδικας Β.3: Προσθήκη στο slurm.h.in

```
#define DEFAULT_THESIS_PARAMS       "n:c:m:t"
```

Κώδικας Β.4: Προσθήκη στοread_config.h

```
...
s_p_options_t slurm_conf_options[] = {
  ...
  {"ThesisParameters", S_P_STRING},
  ...
};
...
extern void free_slurm_conf (slurm_ctl_conf_t *ctl_conf_ptr,
    bool purge_node_hash)
```

```
{
  ...
  xfree(ctl_conf_ptr->thesis_params);
  ...
}
...
void init_slurm_conf (slurm_ctl_conf_t *ctl_conf_ptr)
{
  ...
  xfree(ctl_conf_ptr->thesis_params);
  ...
}
...
static int _validate_and_set_defaults(slurm_ctl_conf_t
   *conf, s_p_hashtbl_t *hashtbl)
{
  ...
  if (!s_p_get_string(&conf->thesis_params,
     "ThesisParameters", hashtbl))
    conf->thesis_params = xstrdup(DEFAULT_THESIS_PARAMS);
  ...
}
...
```

Κώδικας Β.5: Προσθήκη στο `read_config.c`

```
char *slurm_get_thesis_params(void)
{
  char *thesis_params = NULL;
  slurm_ctl_conf_t *conf;

  if (slurmdbd_conf) {
  } else {
    conf = slurm_conf_lock();
    thesis_params = xstrdup(conf->thesis_params);
    slurm_conf_unlock();
  }
  return thesis_params;
}
```

Κώδικας Β.6: Προσθήκη στο `slurm_protocol_api.c`

```
char *slurm_get_thesis_params(void);
```

Κώδικας Β.7: Προσθήκη στο `slurm_protocol_api.h`

Κώδικας της επέκτασης

```
/*
*  thesis_wrapper.c - plugin for Slurm's internal scheduler.
*/

#include ...

const char plugin_name[] = "Alex's Thesis Scheduler plugin";
const char plugin_type[] = "sched/thesis";
const uint32_t plugin_version = SLURM_VERSION_NUMBER;

static pthread_t thesis_thread = 0;
static pthread_mutex_t thread_flag_mutex =
    PTHREAD_MUTEX_INITIALIZER;

int init(void)
{
  sched_verbose("Thesis scheduler plugin loaded");

  slurm_mutex_lock( &thread_flag_mutex );
  if ( thesis_thread ) {
    debug2( "Thesis scheduler thread already running, "
    "not starting another" );
    slurm_mutex_unlock( &thread_flag_mutex );
    return SLURM_ERROR;
  }

  slurm_thread_create(&thesis_thread, thesis_agent, NULL);

  slurm_mutex_unlock( &thread_flag_mutex );

  return SLURM_SUCCESS;
}

void fini(void)
{
  slurm_mutex_lock( &thread_flag_mutex );
  if ( thesis_thread ) {
    verbose( "Thesis scheduler plugin shutting down" );
    stop_thesis_agent();
    pthread_join(thesis_thread, NULL);
    thesis_thread = 0;
  }
  slurm_mutex_unlock( &thread_flag_mutex );
}

int slurm_sched_p_reconfig(void)
{
```

```
47   thesis_reconfig();
48   return SLURM_SUCCESS;
49 }
50
51 uint32_t slurm_sched_p_initial_priority(uint32_t last_prio,
52 struct job_record *job_ptr)
53 {
54   return priority_g_set(last_prio, job_ptr);
55 }
```

Κώδικας Β.8: thesis_wrapper.c

```
1  /*
2  *  thesis.c - Implementation of thesis sched plugin
3  */
4
5  #include ...
6
7  #ifndef THESIS_INTERVAL
8  #  define THESIS_INTERVAL   2
9  #endif
10
11 #ifndef DEFAULT_THESIS_PARAMS
12 #  define DEFAULT_THESIS_PARAMS       "n:c:m:t"
13 #endif
14
15 /****** local variables ******/
16 static bool stop_thesis = false;
17 static pthread_mutex_t term_lock = PTHREAD_MUTEX_INITIALIZER;
18 static pthread_cond_t  term_cond = PTHREAD_COND_INITIALIZER;
19 static bool config_flag = false;
20 static int thesis_interval = THESIS_INTERVAL;
21 char *thesis_params = DEFAULT_THESIS_PARAMS;
22 static int max_sched_job_cnt = 1;
23 static int sched_timeout = 0;
24
25 /****** local functions ******/
26 static void _begin_scheduling(void);
27 static void _load_config(void);
28 static void _my_sleep(int secs);
29
30 /* Terminate thesis_agent */
31 extern void stop_thesis_agent(void)
32 {
33   slurm_mutex_lock(&term_lock);
34   stop_thesis = true;
35   slurm_cond_signal(&term_cond);
36   slurm_mutex_unlock(&term_lock);
37 }
```

```
38
39 static void _my_sleep(int secs)
40 {
41   struct timespec ts = {0, 0};
42   struct timeval now;
43
44   gettimeofday(&now, NULL);
45   ts.tv_sec = now.tv_sec + secs;
46   ts.tv_nsec = now.tv_usec * 1000;
47   slurm_mutex_lock(&term_lock);
48   if (!stop_thesis)
49     slurm_cond_timedwait(&term_cond, &term_lock, &ts);
50   slurm_mutex_unlock(&term_lock);
51 }
52
53 static void _load_config(void)
54 {
55   char *sched_params, *select_type, *tmp_ptr;
56
57   sched_timeout = slurm_get_msg_timeout() / 2;
58   sched_timeout = MAX(sched_timeout, 1);
59   sched_timeout = MIN(sched_timeout, 10);
60
61   sched_params = slurm_get_sched_params();
62
63   if (sched_params && (tmp_ptr=strstr(sched_params,
       "interval=")))
64     thesis_interval = atoi(tmp_ptr + 9);
65   if (thesis_interval < 1) {
66     error("Invalid SchedulerParameters interval: %d",
67     thesis_interval);
68     thesis_interval = THESIS_INTERVAL;
69   }
70
71   if (sched_params && (tmp_ptr=strstr(sched_params,
       "max_job_bf=")))
72     max_sched_job_cnt = atoi(tmp_ptr + 11);
73   if (sched_params && (tmp_ptr=strstr(sched_params,
       "bf_max_job_test=")))
74     max_sched_job_cnt = atoi(tmp_ptr + 16);
75   if (max_sched_job_cnt < 1) {
76     error("Invalid SchedulerParameters bf_max_job_test: %d",
77     max_sched_job_cnt);
78     max_sched_job_cnt = 50;
79   }
80   xfree(sched_params);
81
82   select_type = slurm_get_select_type();
83   if (!xstrcmp(select_type, "select/serial")) {
```

```
84      max_sched_job_cnt = 0;
85      stop_thesis_agent();
86    }
87    xfree(select_type);
88
89    thesis_params = slurm_get_thesis_params();
90    debug2("THESIS: loaded thesis params: %s", thesis_params);
91  }
92
93  static int node_comparator(job_queue_rec_t *job_rec1,
        job_queue_rec_t *job_rec2)
94  {
95    return (job_rec1->job_ptr->details->min_nodes -
          job_rec2->job_ptr->details->min_nodes);
96  }
97
98  static int cpu_comparator(job_queue_rec_t *job_rec1,
        job_queue_rec_t *job_rec2)
99  {
100   return (job_rec1->job_ptr->details->min_cpus -
          job_rec2->job_ptr->details->min_cpus);
101 }
102
103 static int memory_comparator(job_queue_rec_t *job_rec1,
        job_queue_rec_t *job_rec2)
104 {
105   return (job_rec1->job_ptr->details->pn_min_memory -
          job_rec2->job_ptr->details->pn_min_memory);
106 }
107
108 static int submit_time_comparator(job_queue_rec_t *job_rec1,
        job_queue_rec_t *job_rec2)
109 {
110   return
          SLURM_DIFFTIME(job_rec1->job_ptr->details->submit_time,
          job_rec2->job_ptr->details->submit_time);
111 }
112
113 static int composite_comparator(void *x, void *y)
114 {
115   job_queue_rec_t *job_rec1 = *(job_queue_rec_t **) x;
116   job_queue_rec_t *job_rec2 = *(job_queue_rec_t **) y;
117
118   int result = 0;
119
120   char* temp_thesis_params = xmalloc(sizeof(*thesis_params));
121   strcpy(temp_thesis_params, thesis_params);
122
123   char *token = strtok(temp_thesis_params, ":");
```

```
124
125   while (token != NULL)
126   {
127     switch (token[0])
128     {
129       case (int) 'n':
130       result = node_comparator(job_rec1, job_rec2);
131       break;
132       case (int) 'c':
133       result = cpu_comparator(job_rec1, job_rec2);
134       break;
135       case (int) 'm':
136       result = memory_comparator(job_rec1, job_rec2);
137       break;
138       case (int) 't':
139       result = submit_time_comparator(job_rec1, job_rec2);
140       break;
141       default:
142       result = -1;
143     }
144
145     token = strtok(NULL, ":");
146
147     if (result != 0)
148       break;
149   }
150
151   if (result == 0)
152     result = submit_time_comparator(job_rec1, job_rec2);
153
154   xfree(temp_thesis_params);
155
156   return result;
157 }
158
159 static void _begin_scheduling(void)
160 {
161   int j, rc = SLURM_SUCCESS, job_cnt = 0;
162   List job_queue;
163   job_queue_rec_t *job_queue_rec;
164   struct job_record *job_ptr;
165   struct part_record *part_ptr;
166   bitstr_t *alloc_bitmap = NULL, *avail_bitmap = NULL;
167   bitstr_t *exc_core_bitmap = NULL;
168   uint32_t max_nodes, min_nodes;
169   time_t now = time(NULL), sched_start;
170   bool resv_overlap = false;
171   sched_start = now;
172   alloc_bitmap = bit_alloc(node_record_count);
```

```
173   job_queue = build_job_queue(true, false);
174
175   /**
176   * Sort the list based on thesis_params configuration
177   */
178   debug2("THESIS: current thesis params: %s", thesis_params);
179   list_sort(job_queue, composite_comparator);
180
181   while ((job_queue_rec = (job_queue_rec_t *)
          list_pop(job_queue))) {
182     job_ptr  = job_queue_rec->job_ptr;
183     part_ptr = job_queue_rec->part_ptr;
184     xfree(job_queue_rec);
185     if (part_ptr != job_ptr->part_ptr)
186       continue; /* Only test one partition */
187
188     if (++job_cnt > max_sched_job_cnt) {
189       debug2("scheduling loop exiting after %d jobs",
190       max_sched_job_cnt);
191       break;
192     }
193
194     /* Determine minimum and maximum node counts */
195     min_nodes = MAX(job_ptr->details->min_nodes,
196     part_ptr->min_nodes);
197
198     if (job_ptr->details->max_nodes == 0)
199       max_nodes = part_ptr->max_nodes;
200     else
201       max_nodes = MIN(job_ptr->details->max_nodes,
             part_ptr->max_nodes);
202
203     max_nodes = MIN(max_nodes, 500000);      /* prevent
          overflows */
204
205     if (min_nodes > max_nodes) {
206       /* job's min_nodes exceeds partition's max_nodes */
207       continue;
208     }
209
210     j = job_test_resv(job_ptr, &now, true, &avail_bitmap,
211     &exc_core_bitmap, &resv_overlap, false);
212
213     if (j != SLURM_SUCCESS) {
214       FREE_NULL_BITMAP(avail_bitmap);
215       FREE_NULL_BITMAP(exc_core_bitmap);
216       continue;
217     }
218
```

```
219     // actual resource allocation
220     rc = select_nodes(job_ptr, false, NULL, NULL, false);
221
222     if (rc == SLURM_SUCCESS) {
223       /* job initiated */
224       last_job_update = time(NULL);
225       debug2("THESIS: Started JobId %d on %s",
226       job_ptr->job_id, job_ptr->nodes);
227       if (job_ptr->batch_flag == 0)
228         srun_allocate(job_ptr);
229       else if (!IS_JOB_CONFIGURING(job_ptr))
230         launch_job(job_ptr);
231     }
232
233     FREE_NULL_BITMAP(avail_bitmap);
234     FREE_NULL_BITMAP(exc_core_bitmap);
235
236     if ((time(NULL) - sched_start) >= sched_timeout) {
237       debug2("scheduling loop exiting after %d jobs",
238       max_sched_job_cnt);
239       break;
240     }
241   }
242   FREE_NULL_LIST(job_queue);
243   FREE_NULL_BITMAP(alloc_bitmap);
244 }
245
246 /* Note that slurm.conf has changed */
247 extern void thesis_reconfig(void)
248 {
249   config_flag = true;
250 }
251
252 /* thesis_agent - detached thread */
253 extern void *thesis_agent(void *args)
254 {
255   time_t now;
256   double wait_time;
257   static time_t last_sched_time = 0;
258   /* Read config, nodes and partitions; Write jobs */
259   slurmctld_lock_t all_locks = {
260     READ_LOCK, WRITE_LOCK, READ_LOCK, READ_LOCK, READ_LOCK };
261
262   _load_config();
263   last_sched_time = time(NULL);
264   while (!stop_thesis) {
265     _my_sleep(thesis_interval);
266     if (stop_thesis)
267       break;
```

```
268     if (config_flag) {
269       config_flag = false;
270       _load_config();
271     }
272     now = time(NULL);
273     wait_time = difftime(now, last_sched_time);
274     if ((wait_time < thesis_interval))
275       continue;
276
277     lock_slurmctld(all_locks);
278     _begin_scheduling();
279     last_sched_time = time(NULL);
280     (void) bb_g_job_try_stage_in();
281     unlock_slurmctld(all_locks);
282   }
283   return NULL;
284 }
```

Κώδικας Β.9: `thesis.c`