ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

XML Parsing με χρήση Web Workers

Διπλωματική Εργασία

του

Μυλώση Κωνσταντίνου

Θεσσαλονίκη, Ιούνιος 2020

XML PARSING ΜΕ ΧΡΗΣΗ WEB WORKERS

Μυλώσης Κωνσταντίνος

Εφαρμοσμένη Πληροφορική, ΠΑΜΑΚ 2018

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κασκάλης Θεόδωρος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30/6/2020

Κασκάλης Θεόδωρος	Μαργαρίτης Κωνσταντίνος	Σακελλαρίου Ηλίας
.....	.....	.....

Μυλώσης Κωνσταντίνος

.....

Περίληψη

Οι απαιτήσεις της σύγχρονης κοινωνίας για διαμερισμό μεγάλου όγκου πληροφοριών σε πραγματικό χρόνο, οδήγησε στην ανάγκη ανάπτυξης νέων προτύπων επεξεργασίας και μετάδοσης πληροφοριών ή βελτιστοποίηση των ήδη υπαρχόντων, τόσο για ελάφρυνση του φόρτου επεξεργασίας των κεντρικών διακομιστών όσο και για βελτίωση της εμπειρίας του χρήστη. Σκοπός της παρούσας διατριβής ήταν η εξεύρεση της κατάλληλης στρατηγικής για τη βελτιστοποίηση της επεξεργασίας αρχείων σε διαδικτυακές εφαρμογές με τη χρήση web workers, μέσω της συγκριτικής μελέτης προτύπων ανταλλαγής δεδομένων και μοντέλων επεξεργασίας δεδομένων. Για αυτόν τον σκοπό, δημιουργήθηκε μια διαδικτυακή εφαρμογή αξιοποίησης των web workers σε περιβάλλον περιηγητή δικτύου με γλώσσα προγραμματισμού JavaScript. Η συμπεριφορά των web workers και η αποτελεσματικότητα της χρήσης τους εξετάστηκαν σε τρία διαφορετικά υπολογιστικά συστήματα με διαφορετικές παραμέτρους. Από τα αποτελέσματα προέκυψαν τα εξής συμπεράσματα: (α) όσο μεγαλύτερος είναι ο όγκος των προς επεξεργασία δεδομένων, τόσο πιο εμφανές είναι και το όφελος από τη χρήση web workers, (β) η αποδοτικότητα των web workers αυξάνεται με τη χρήση περισσότερων λογικών επεξεργαστών, αλλά μέχρι το 80% περίπου των διαθέσιμων, για αυτό θα πρέπει να τηρείται ένα όριο στη χρήση των διαθέσιμων πόρων. Με βάση τα συμπεράσματα και τους περιορισμούς της έρευνας, προτείνονται θέματα για μελλοντική διερεύνηση.

Λέξεις Κλειδιά*:* πρότυπα, ανταλλαγή δεδομένων, XML, parsing, DOM, παράλληλη επεξεργασία, κατάτμηση, JavaScript, περιηγητές, web workers, νέφος

Abstract

Nowadays, the demands for sharing large volumes of data in real-time, have led to the need to develop new standards for processing and transmitting information or optimizing existing ones so as to alleviate the processing load of central servers and improve the user's experience at the same time. The purpose of this dissertation was to find the appropriate strategy for optimizing file processing in web applications by using web workers, through the comparative study of data exchange models and data processing models. For this purpose, a web application was implemented in JavaScript, in order to utilize web workers in a network browser environment. The behavior of web workers and the effectiveness of their use were examined in three different computer systems with different parameters. The results showed the following: (a) the larger the volume of data to be processed, the more obvious the benefit of using web workers, (b) the efficiency of web workers is increased by the use of more logical processors, but up to about 80% of the available ones, so it is suggested that there should be a limit on the use of available resources. Taking all the conclusions and limitations of the research into account, there are some issues which are proposed for future investigation.

Keywords*:* prototypes, data exchange, XML, parsing, DOM, parallel processing, chunking, JavaScript, browsers, web workers, cloud

Πρόλογος – Ευχαριστίες

Ολοκληρώνοντας τη συγγραφή της παρούσας Μεταπτυχιακής Διατριβής, αισθάνομαι την ανάγκη να εκφράσω τις ευχαριστίες μου σε όλους όσοι συνέβαλαν στην ολοκλήρωση αυτής της προσπάθειας. Οφείλω να ευχαριστήσω ιδιαίτερα τον Αναπληρωτή Καθηγητή κ. Κασκάλη Θεόδωρο για την εμπιστοσύνη που μου έδειξε αναθέτοντάς μου την παρούσα διατριβή και για την υποστήριξη και την καθοδήγησή του κατά τον σχεδιασμό, την οργάνωση και την υλοποίηση της. Επίσης, ευχαριστώ θερμά τον Καθηγητή κ. Μαργαρίτη Κωνσταντίνο και τον Επίκουρο Καθηγητή κ. Σακελλαρίου Ηλία της τριμελούς επιτροπής, για την υποστήριξή τους. Η συνεργασία μου με τα μέλη της συμβουλευτικής επιτροπής ήταν ουσιαστική και αποτελεσματική ως προς την ολοκλήρωση της διατριβής. Ιδιαίτερα οφείλω να ευχαριστήσω τον θείο μου Μυλώση Κωνσταντίνο, για την άμεση, κρίσιμη, ουσιαστική και καθοριστική του υποστήριξη, όταν και όπου χρειάστηκε. Τέλος, ευχαριστώ την οικογένεια μου για την παρακίνηση, την ενθάρρυνση και την ενίσχυση, όσον αφορά στις σπουδές μου και κατά επέκταση και στην παρούσα διατριβή. Παράλληλα, οφείλω να αναφέρω ότι παρά την επισταμένη καθοδήγηση από τον κ. Κασκάλη Θεόδωρο και από τα υπόλοιπα μέλη της τριμελούς επιτροπής, κατά τη διαδικασία της εκπόνησης της παρούσας διατριβής, οποιαδήποτε αδυναμία ή παράλειψη είναι αποκλειστικά δική μου ευθύνη.

Περιεχόμενα

Κατάλογος Εικόνων (αν υπάρχουν)

Κατάλογος Πινάκων (αν υπάρχουν)

Συμβολισμοί

API	Application Program Interface
ARPANET	Advanced Research Projects Agency Network
COP	Common Operational Picture
CORS	Cross-Origin Resource Sharing
CSV	Comma Separated Values
DFA	Deterministic Finite Automaton
DOM	Document Object Model
ECMA	European Computer Manufacturers
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FST	Finite State Transducer
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
IEFT	Internet Engineering Task Force
IoT	Internet of Things
JSON	JavaScript Objective Notation
MIME	Multipurpose Internet Mail Extensions
PXP	Parallel XML Parsing
RAM	Random Access Memory
SAX	Simple API for XML
SFT	Simultaneous Finite Transducer
SGML	Standard Generalized Markup Language
StAX	Streaming API for XML
VTD	Virtual Token Descriptor
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
XML	eXtensible Markup Language
YAML	YAML Ain’t a Markup Language

1 Εισαγωγή

Η αυξανόμενη ανάγκη για εκτέλεση απλών ή πολύ σύνθετων διεργασιών αποτελεσματικότερα, ευκολότερα, με ακρίβεια και πολύ μεγάλη ταχύτητα, είχε ως συνέπεια την διαρκή προσπάθεια για αναζήτηση, ανάπτυξη και βελτίωση νέων, αποδοτικότερων μεθόδων, συστημάτων και προτύπων, με την αξιοποίηση των Η/Υ και των δικτύων. Η ανάγκη δε για διαμερισμό πόρων σε παγκόσμια κλίμακα, οδήγησε στην ανάπτυξη του παγκόσμιου διαδικτύου (Internet), που αποτελεί τη διασύνδεση τοπικών δικτύων και δικτύων ευρείας περιοχής (Tim Berners-Lee, 1980). Με την διαρκή εξέλιξη της τεχνολογίας, παρέχεται η δυνατότητα διαρκούς σύνδεσης, επικοινωνίας και ανταλλαγής πληροφοριών στο διαδίκτυο και με άλλες συσκευές εκτός από τους Η/Υ, όπως κινητά, κάμερες, αυτοκίνητα, παιχνίδια, ιατρικές συσκευές, έξυπνες συσκευές σπιτιού, ακόμα και μικρό αισθητήρες πάνω σε δέντρα (Gupta et al., 2010).

Η διασύνδεση αυτών των συσκευών σε ένα δίκτυο επικοινωνίας και αλληλεπίδρασης, δημιούργησε το Internet of Things (IoT). Οι αισθητήρες και το λογισμικό συνδυάζονται άψογα με το περιβάλλον και οι πληροφορίες διακινούνται σε διαφορετικές πλατφόρμες-κόμβους εύκολα και αποτελεσματικά. Η ραγδαία εξέλιξη της διασύνδεσης διαφορετικών πλατφορμών, δημιούργησε την ανάγκη τυποποίησης του τρόπου ανταλλαγής πληροφοριών. Κατά καιρούς χρησιμοποιήθηκαν διάφορα πρότυπα, με πιο διαδεδομένα το CSV, το XML και το JSON. Τα πρότυπα αυτά, χρησιμοποιούνται για να ελέγξουν τη δομή πριν την αποστολή, αλλά και για να αναλύσουν τις χρήσιμες πληροφορίες κατά την παραλαβή. Καθορίζουν τη μορφή και τη σειρά που θα έχουν τα δεδομένα μέσα στη ροή πληροφοριών. Έτσι, δίνεται η δυνατότητα στον παραλήπτη, να μπορεί να τα επεξεργαστεί με μεγάλη ευκολία και να ελέγξει την ακεραιότητα της μεταδιδόμενης πληροφορίας.

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Παρόλο που αυτά τα πρότυπα έχουν συμβάλλει αρκετά για τη διευκόλυνση ανταλλαγής πληροφοριών, ο όγκος της πληροφορίας που περιέχεται σε τέτοιου είδους αρχεία, έχει αρχίσει και αυξάνεται εκθετικά στις μέρες μας. Οι απαιτήσεις της σύγχρονης κοινωνίας για επικοινωνία αυξάνονται συνεχώς, απαιτώντας νέους τύπους, καλύτερη ποιότητα (ήχος, εικόνα) και φυσικά ταχύτητα. Οι σύγχρονες εφαρμογές απαιτούν

επεξεργασία μεγάλου όγκου πληροφοριών σε πραγματικό χρόνο. Γίνεται αντιληπτό, ότι η επεξεργασία αρχείων που περιέχουν τις παραπάνω πληροφορίες, καθίσταται κοστοβόρα και πολύ χρονοβόρα διαδικασία για ένα υπολογιστικό σύστημα.

Γνωρίζοντας ότι οι επεξεργαστική ισχύς σε ένα σύστημα διπλασιάζεται ανά δύο χρόνια, σύμφωνα με τον πρώτο νόμο του Moore (Mack, 2011), δημιουργείται η ανάγκη αναπροσαρμογής των αλγορίθμων. Οι αλγόριθμοι, χρησιμοποιούνται για να προσφέρουν λύση σε διάφορα προβλήματα που παρουσιάζονται. Θα πρέπει να ανανεώνονται και να αξιοποιούν την επεξεργαστική ισχύ που αποκτάται με το πέρασμα του χρόνου, για ταχύτερη επίλυση των προβλημάτων. Η αύξηση του όγκου των ανταλλασσόμενων πληροφοριών και των απαιτήσεων για καλύτερη ποιότητα και ταχύτητα επικοινωνίας, οδήγησε στην ανάγκη ανάπτυξης νέων αλγορίθμων επεξεργασίας ή βελτιστοποίηση των ήδη υπαρχόντων, που θα αξιοποιούν πιο αποτελεσματικά και αποδοτικά την παράλληλη αύξηση της επεξεργαστικής ισχύος.

Τα τελευταία χρόνια η πρόσβαση των χρηστών στο διαδίκτυο αυξάνεται με ραγδαίους ρυθμούς με παράλληλη αύξηση των απαιτήσεων για ποιότητα και ταχύτητα. Αυτό οδηγεί στην ανάγκη για συνεχή αναβάθμιση της δυναμικότητας των διακομιστών και του εύρους του χρησιμοποιούμενου δικτύου. Παράλληλα με την εξέλιξη της τεχνολογίας, η δυναμικότητα των υπολογιστών των χρηστών τείνει να συναγωνίζεται αυτή των διακομιστών, χωρίς να αξιοποιείται πλήρως.

Η διευθέτηση των παραπάνω, θα βοηθήσει στην ελάφρυνση των διακομιστών, στην καλύτερη εκμετάλλευση της δυναμικότητας των υπολογιστών των χρηστών και τελικά στην βελτίωση της εμπειρίας του χρήστη, μεταφέροντας χρονοβόρες διαδικασίες στο παρασκήνιο. Η αξιολόγηση της δυνατότητας των περιηγητών για παράλληλη επεξεργασία μέσω web workers, θα οδηγήσει στην εξεύρεση κατάλληλων στρατηγικών, προκειμένου να βελτιστοποιηθούν χρονοβόρες διαδικασίες.

1.2 Σκοπός – Στόχοι

Στις προηγούμενες παραγράφους, παρουσιάστηκε η χρησιμότητα των προτύπων για την εύκολη και βέλτιστη ανταλλαγή μεγάλου όγκου δεδομένων και η ανάγκη για εύρεση νέων προτύπων ή βελτιστοποίηση των ήδη υπαρχόντων. Πρωταρχικός σκοπός της παρούσας έρευνας, ήταν η μελέτη προτύπων ανταλλαγής και μοντέλων επεξεργασίας δεδομένων. Αναφέρθηκαν τα στάδια που εκτελούνται στην επεξεργασία για να

αντλήσουν την χρήσιμη πληροφορία κατά την παραλαβή δεδομένων. Επιπρόσθετα, μελετήθηκαν στρατηγικές και αλγόριθμοι που βελτιστοποιούν την επεξεργασία δεδομένων. Ο στόχος της έρευνας ήταν η εύρεση της κατάλληλης στρατηγικής για τη βελτιστοποίηση της επεξεργασίας αρχείων σε διαδικτυακές εφαρμογές με τη χρήση web workers.

1.3 Συνεισφορά

Η συνεισφορά της μελέτης συνοψίζεται στα παρακάτω. Έγινε μια εκτενής βιβλιογραφική ανασκόπηση. Παρουσιάστηκαν και συγκρίθηκαν βασικά πρότυπα ανταλλαγής δεδομένων. Σταχυολογήθηκαν αναφορές από έρευνες και εξηγήθηκαν βασικοί αλγόριθμοι επεξεργασίας. Αναφέρθηκαν διάφορα στάδια του parsing και παρουσιάστηκαν τεχνικές που χρησιμοποιήθηκαν ανά στάδιο. Εξετάστηκε η χρήση παράλληλης επεξεργασίας σε web εφαρμογές και τα προβλήματα της, καθώς είναι σχετικά νέα τεχνική. Σχεδιάστηκε και αναπτύχθηκε κώδικας, που υλοποιεί XML Parsing, με έναν υβριδικό αλγόριθμο παράλληλης επεξεργασίας με web workers. Αξιολογήθηκε η απόδοσή του σε διαφορετικές πλατφόρμες και παρουσιάστηκαν τα αποτελέσματα.

1.4 Βασική Ορολογία

Parse/Parsing: η διαδικασία ανάλυσης ενός εγγράφου με σκοπό και αποτέλεσμα την απορροή χρήσιμων πληροφοριών.

Partitioning: μια λογική διαίρεση ενός αντικειμένου, σε ξεχωριστά ανεξάρτητα κομμάτια.

Αλγόριθμος: μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος.

Περιηγητής ιστοσελίδων (browser): το βασικότερο πρόγραμμα για την πλοήγηση και εκτέλεση διαδικτυακών εφαρμογών.

Script: ένα κομμάτι κώδικα γραμμένο σε JavaScript που εκτελείται σε έναν περιηγητή στη μεριά του χρήστη.

JavaScript: είναι μια γλώσσα προγραμματισμού και αποτελεί μια από τις βασικές τεχνολογίες στον παγκόσμιο ιστό. Ενεργοποιεί διαδραστικές ιστοσελίδες και αποτελεί ουσιαστικό μέρος των εφαρμογών ιστού.

Νήματα (threads): Διεργασίες που εκκινούνται από το κύριο πρόγραμμα και εκτελούνται παράλληλα με αυτό, βελτιώνοντας την απόδοση ενός συστήματος.

Web workers: μηχανισμός για την εκτέλεση διεργασιών σε ξεχωριστά νήματα για μια διαδικτυακή εφαρμογή.

Cloud: είναι η διάθεση υπολογιστικών πόρων μέσω διαδικτύου (π.χ. servers, applications κ.λ.π.), από κεντρικά συστήματα που βρίσκονται σε τοποθεσίες απομακρυσμένες και άγνωστες στον τελικό χρήστη (νέφος).

1.5 Διάρθρωση της μελέτης

Αρχικά, στο Κεφάλαιο 2, γίνεται μια βιβλιογραφική επισκόπηση, όπου περιγράφονται έννοιες και αναλύονται θέματα, που κρίνονται απαραίτητα για τη διαμόρφωση ενός επαρκούς θεωρητικού υπόβαθρου αναφορικά με το περιεχόμενο της παρούσας μελέτης. Εξετάζονται διάφορα πρότυπα ανταλλαγής δεδομένων και ιδιαίτερα το XML. Αναφέρονται διάφορα μοντέλα επεξεργασίας δεδομένων. Πιο εξειδικευμένα, εξετάζονται τα στάδια επεξεργασίας κατά την παραλαβή δεδομένων και η χρήση σειριακής ή παράλληλης προσέγγισης για την ολοκλήρωση της. Στη συνέχεια, παρουσιάζονται διάφορες στρατηγικές και αλγόριθμοι, που χρησιμοποιούνται σε αυτά τα πρότυπα για την άντληση πληροφοριών. Έπειτα, εξετάζεται η επεξεργασία αρχείων σε διαδικτυακές εφαρμογές και αναφέρεται ότι το ρόλο των νημάτων για παράλληλη επεξεργασία σε μια τέτοια εφαρμογή τον έχουν οι web workers. Τέλος, γίνεται μια παρουσίαση των web workers. Στο κεφάλαιο 3, με βάση την παραπάνω μελέτη, για την επεξεργασία αρχείων XML, αναπτύχθηκε μία εφαρμογή η οποία χρησιμοποιεί σειριακή επεξεργασία και μια υβριδική (συνδυασμός σειριακής και παράλληλης), που προσαρμόστηκε για διαδικτυακή εφαρμογή με χρήση web workers. Παρουσιάζονται τα αποτελέσματα και συγκριτικά στοιχεία. Τέλος, στο Κεφάλαιο 4, γίνεται μια σύνοψη και παρατίθενται τα συμπεράσματα που προέκυψαν, κάποια όρια και περιορισμοί της παρούσας μελέτης και προτείνονται μελλοντικές επεκτάσεις για περαιτέρω βελτιστοποίηση.

2 Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο

Η μετάδοση πληροφοριών και ενημέρωσης αποτελεί μια σημαντική ανάγκη επικοινωνίας, προκειμένου να επιλυθούν προβλήματα, να ληφθούν αποφάσεις, να σχεδιαστούν δράσεις, να τεθούν στόχοι επίτευξης, να ανατεθούν καθήκοντα, να συντονιστούν οι ενέργειες των ατόμων που εμπλέκονται, να αξιολογηθούν τα αποτελέσματα και να δοθεί η ανατροφοδότηση (Μακράτζη, 2016). Με την εξέλιξη της πληροφορικής και των υπολογιστικών συστημάτων, αξιοποιήθηκε η επιστήμη των μαθηματικών και της συμβολικής λογικής, με σκοπό την ανάπτυξη σύγχρονων δυναμικών επικοινωνιακών προτύπων.

Είναι εντυπωσιακό να σημειωθεί ότι η ανταλλαγή πληροφοριών μέσω διαδικτύου ξεκίνησε στα μέσα της δεκαετίας του 1960 (Campbell-Kelly, 1987). Το πρώτο μήνυμα στάλθηκε από τον καθηγητή Leonard Kleinrock, το 1969, από το πανεπιστήμιο της Καλιφόρνιας στο Ινστιτούτο Έρευνας του Στάνφορντ, μέσω του δικτύου Advanced Research Projects Agency Network (ARPANET). Στην επερχόμενη δεκαετία, άρχισαν να δημιουργούνται διάφορα δίκτυα, όχι μόνο για την αποστολή απλών μηνυμάτων, αλλά και πιο σύνθετων δομών δεδομένων. Αργότερα, τη δεκαετία του 1980 ο Tim Berners-Lee κατασκεύασε το World Wide Web, συνδέοντας δεδομένα σε ένα πληροφοριακό σύστημα, προσβάσιμο από οποιονδήποτε κόμβο στο δίκτυο.

Η ραγδαία εξέλιξη του δικτύου αυτού, με τη διασύνδεση ετερογενών συστημάτων και τη διόγκωση των διακινούμενων πληροφοριών, δημιούργησε την ανάγκη αποτελεσματικής οργάνωσης, διαχείρισης και επεξεργασίας μεγάλου όγκου δεδομένων. Έπρεπε λοιπόν, να τεθούν κάποιοι κανόνες που θα καθόριζαν έναν αποδεκτό τρόπο επικοινωνίας με κοινό συντακτικό, οι οποίοι θα εξασφάλιζαν τη δομή και την ακεραιότητα της μεταδιδόμενης πληροφορίας. Για το σκοπό αυτό, αναπτύχθηκαν συγκεκριμένα πρότυπα, που θέτουν συγκεκριμένους συντακτικούς και εννοιολογικούς κανόνες.

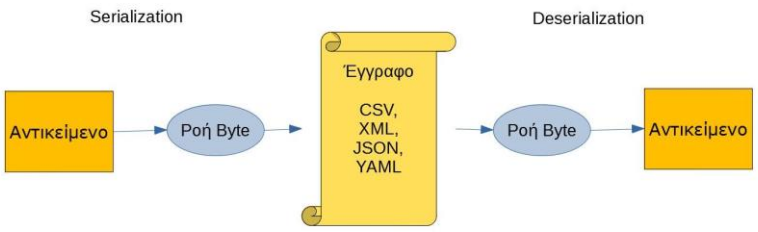
2.1 Πρότυπα ανταλλαγής δεδομένων

Τα συστήματα υπολογιστών ενδέχεται να ποικίλλουν ανάλογα με την αρχιτεκτονική hardware, το λειτουργικό σύστημα και τους μηχανισμούς κωδικοποίησης (encoding). Οι εσωτερικές δυαδικές αναπαραστάσεις δεδομένων ποικίλλουν σε κάθε

περιβάλλον. Η αποθήκευση και η ανταλλαγή δεδομένων μεταξύ αυτών των διαφορετικών περιβαλλόντων, απαιτεί μορφή δεδομένων ουδέτερης πλατφόρμας και γλώσσας, έτσι ώστε να μπορούν να γίνουν κατανοητά από όλα τα συστήματα.

Τα δεδομένα υπολογιστών οργανώνονται σε δομές δεδομένων, όπως πίνακες, δέντρα και αντικείμενα (objects). Για απλά, γραμμικά δεδομένα, έναν αριθμό ή μια συμβολοσειρά, δεν γίνεται λόγος για δομές δεδομένων. Όταν οι δομές δεδομένων πρέπει να αποθηκευτούν ή να μεταδοθούν σε άλλη τοποθεσία, όπως σε ένα δίκτυο, πρέπει να μετασχηματιστούν σε απλή ροή χαρακτήρων (serialization) (*Data Deserialization*, 2016).

Το serialization δεδομένων, είναι η διαδικασία μετατροπής αντικειμένων που υπάρχουν σε πολύπλοκες δομές δεδομένων, σε ροή bytes για σκοπούς αποθήκευσης, μεταφοράς και διανομής σε φυσικές συσκευές. Το serialization καθίσταται πολύπλοκο για εμφωλευμένες δομές δεδομένων και αναφορές αντικειμένων. Όταν τα αντικείμενα διπλώνονται σε πολλαπλά επίπεδα, όπως συμβαίνει στα δέντρα, συρρικνώνονται σε μια σειρά από bytes και υπάρχουν αρκετές πληροφορίες, όπως η σειρά μετάβασης, για να βοηθήσουν στην ανασυγκρότηση της αρχικής διάρθρωσης δέντρου στην πλευρά του προορισμού. Μόλις μεταδοθούν τα δεδομένα από την πηγαία μηχανή στη μηχανή προορισμού, πραγματοποιείται η αντίστροφη διαδικασία από αυτή της δημιουργίας ροών, που ονομάζεται deserialization. Τα ανασυγκροτημένα αντικείμενα είναι κλώνοι του αρχικού αντικειμένου (Εικόνα 2-1).



Εικόνα 2-1: Serialization και deserialization δεδομένων

Το σύνολο των κανόνων που καθορίζουν με λεπτομέρεια όλες τις παραπάνω διαδικασίες ονομάζονται πρότυπα. Στην πορεία του χρόνου δημιουργήθηκαν αρκετά πρότυπα ανάλογα με τους κατασκευαστές και τις ανάγκες που κάλυπταν.

Από τα πιο διαδεδομένα πρότυπα ανταλλαγής δεδομένων είναι το CSV (Comma-Separated Values), XML (eXtensible Markup Language), JSON (JavaScript Object Notation) και YAML (YAML Ain't Markup Language). Η επιλογή του καταλληλότερου προτύπου για μια εφαρμογή εξαρτάται από διάφορους παράγοντες, όπως η πολυπλοκότητα των δεδομένων, η ανάγκη για αναγνωσιμότητα από τον άνθρωπο και οι περιορισμοί ταχύτητας και αποθηκευτικού χώρου.

Σε μια σύντομη χρονική αναδρομή (Arvindpdmn, 2019), το πρότυπο CSV ξεκίνησε να χρησιμοποιείται στα μηχανήματα IBM Fortran από το 1972. Παρόλα αυτά το ίδιο το όνομα CSV δόθηκε στην εποχή των σύγχρονων υπολογιστών, περίπου το 1983. Ο όρος serialization χρησιμοποιείται για πρώτη φορά στο πλαίσιο των δεδομένων το 1997. Την επόμενη χρονιά (1998), έγιναν προτάσεις από το World Wide Web Consortium (W3C) σχετικά με το XML 1.0. Το XML, όπως και το HyperText Markup Language (HTML), προέρχεται από την αρχική γλώσσα σήμανσης Standard Generalized Markup Language (SGML), η οποία δημιουργήθηκε στην εποχή πριν το Internet της δεκαετίας του 1980. Αργότερα, το 2001, ο Douglas Crockford, πήρε τα εύσημα για την εφεύρεση του JSON. Αποκαλύφθηκε από τον ίδιο όμως, ότι χρησιμοποιούταν ήδη από τη Netscape από το 1996. Τέλος, η YAML είναι επέκταση της JSON και η πρώτη έκδοση της κυκλοφόρησε το 2004.

2.1.1 Comma-Separated Values

Τα αρχεία CSV, είναι ένα πρότυπο ανταλλαγής δεδομένων μεταξύ διαφόρων συστημάτων και πλατφορμών. Οι χρήστες εκτιμούν την απλότητα και τη δυνατότητα ανάγνωσης και επεξεργασίας σχεδόν σε κάθε περιβάλλον. Ωστόσο, δεδομένου ότι η μορφή αρχείου CSV δεν υπήρξε ποτέ επίσημα τυποποιημένη, είναι συχνά δύσκολο για έναν αναγνώστη να διαβάσει ένα αρχείο CSV από άγνωστες πηγές. Αυτό οφείλεται στο γεγονός ότι οι πληροφορίες ζωτικής σημασίας για την ανάλυση, όπως η κωδικοποίηση χαρακτήρων, ο τύπος οριοθέτησης, η παρουσία μιας γραμμής κεφαλίδας και οι τύποι δεδομένων στήλης, δεν δίδονται ρητά στον αναγνώστη (Döhmen, 2016).

Το 2005, η Internet Engineering Task Force (IEFT), δημοσιοποίησε ένα πρότυπο με κάποιες προαπαιτήσεις, έτσι ώστε όλα τα αρχεία CSV να έχουν κοινή δομή για να μπορούν να αναλυθούν και να κατανοηθούν με μεγαλύτερη ευκολία και να υπάρχει μια υποτυπώδης συνεννόηση μεταξύ μηχανής αποστολής και μηχανής παραλήπτη.

- Η πρώτη γραμμή του αρχείου, η κεφαλίδα, είναι προαιρετική. Ωστόσο, η παρουσία ή η απουσία της γραμμής κεφαλίδας πρέπει να υποδεικνύεται μέσω του προαιρετικού στοιχείου παράμετρο κεφαλίδας αυτού του τύπου Multipurpose Internet Mail Extensions (MIME) (Shafranovich, 2005).
- Κάθε εγγραφή βρίσκεται σε μια γραμμή, οριοθετημένη με αλλαγή γραμμής (line break).
- Τα πεδία μιας εγγραφής διαχωρίζονται με κόμμα ή ελληνικό ερωτηματικό
- Κάθε γραμμή περιέχει τον ίδιο αριθμό πεδίων.

Ένα μικρό παράδειγμα που δείχνει τη σύνταξη CSV αρχείων φαίνεται παρακάτω, όπου το CSV παρουσιάζει πληροφορίες ανθρώπων:

1| Name,Surname,Age

2| Kostas,Milo,26

3| Dimitris,Milo,56

4| …

Ένα αρχείο CSV, μετά την τυποποίησή του, είναι εύκολο για να διαβαστεί από τον άνθρωπο και να τροποποιηθεί με το χέρι. Μπορεί με απλό τρόπο να υλοποιηθεί και να αναλυθεί. Υπάρχουν πάρα πολλές εφαρμογές που μπορούν να επεξεργαστούν CSV αρχεία. Τέλος, παρέχει εξαρχής πληροφορίες για το σχήμα του, είναι μικρό σε μέγεθος και μπορεί να γίνει αρκετά γρήγορα η διαχείρισή του.

Παρόλα αυτά, τα CSV αρχεία επιτρέπουν μόνο τη μεταφορά βασικών δεδομένων. Οι πολύπλοκες διαμορφώσεις δεν μπορούν να εισαχθούν και να εξαχθούν με αυτό το πρότυπο. Επιπρόσθετα, δεν υπάρχει διαχωρισμός μεταξύ κειμένου και αριθμητικών τιμών. Δεν υπάρχει τυποποιημένος τρόπος αναπαραγωγής δεσμευμένων χαρακτήρων. Υπάρχει δυσκολία εισαγωγής ενός CSV αρχείου σε SQL βάση δεδομένων, γιατί δεν υπάρχει διαχωρισμός μεταξύ κενής τιμής NULL και κενού διαστήματος “”. Ένα άλλο μειονέκτημα, είναι ότι δεν υπάρχει τυποποιημένος τρόπος αναπαραγωγής δυαδικών δεδομένων. Είναι γνωστό ότι υπάρχει μικρή υποστήριξη για ειδικούς χαρακτήρες. Τέλος, παρατηρείται ότι ενώ υπάρχει τυποποιημένο πρότυπο, όπως προαναφέρθηκε, μπορεί να υπάρξουν πολλές διαφοροποιήσεις από CSV σε CSV, που πληρούν τις προαπαιτήσεις. Συμπερασματικά λοιπόν, μπορεί να ειπωθεί ότι τα CSV αρχεία προτείνονται για χρήση με απλή και γρήγορη υλοποίηση, ανάλυση και επεξεργασία.

2.1.2 eXtensible Markup Language

Το XML είναι πρότυπο ανταλλαγής δεδομένων, που περιέχει ένα σύνολο κανόνων για την ηλεκτρονική κωδικοποίηση κειμένων. Ορίζεται, κυρίως, στην προδιαγραφή XML 1.0, που δημιούργησε ο διεθνής οργανισμός προτύπων World Wide Web Consortium (W3C, 2008), αλλά και σε διάφορες άλλες σχετικές προδιαγραφές ανοιχτών προτύπων.

Το XML σχεδιάστηκε δίνοντας έμφαση στην απλότητα, τη γενικότητα και τη χρησιμότητα της στο διαδίκτυο (Nurseitov et al., 2009). Είναι μία μορφοποίηση δεδομένων κειμένου, με ισχυρή υποστήριξη κωδικοποίησης Unicode για όλες τις γλώσσες του κόσμου. Κατά τους σχεδιαστικούς στόχους του XML, το W3C ορίζει ότι η XML θα πρέπει να χρησιμοποιείται εύκολα μέσω του διαδικτύου και ότι τα XML αρχεία θα πρέπει να είναι ευανάγνωστα από τον άνθρωπο και με λογική. Αν και η σχεδίαση του XML εστιάζει στα κείμενα, χρησιμοποιείται ευρέως και για την αναπαράσταση ακανόνιστων δομών δεδομένων, που προκύπτουν στις υπηρεσίες ιστού.

Το XML κατά κόρον, εξυπηρετεί δύο βασικούς σκοπούς. Χρησιμοποιείται για να δομήσει δεδομένα και να διευκολύνει την αποθήκευση και τη μεταφορά τους. Αυτό το καταφέρνει με την ιδιαίτερη και χαρακτηριστική του σύνταξη. Είναι σημαντικό να αναφερθεί ότι το XML είναι μια μορφή ιεραρχικής μορφής δεδομένων καθορισμένη από το χρήστη.

Μέσα σε ένα αρχείο XML υπάρχουν ετικέτες και κείμενο. Η χρήση ετικετών είναι αναγκαία για τη σωστή δόμηση των δεδομένων. Το κείμενο που πρόκειται να αποθηκευτεί, περιβάλλεται από ετικέτες που ακολουθούν συγκεκριμένες οδηγίες σύνταξης. Αυτό που κάνει το XML διαφορετικό από την HTML, είναι το γεγονός ότι το XML χρησιμοποιείται για τη μεταφορά πληροφοριών ενώ η HTML χρησιμοποιείται για την προβολή πληροφοριών.

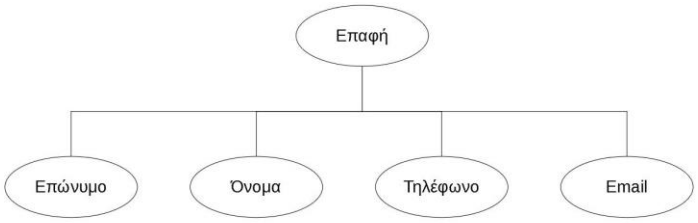
Ένα μικρό παράδειγμα που δείχνει τη σύνταξη XML αρχείων φαίνεται παρακάτω, όπου το XML παρουσιάζει πληροφορίες ανθρώπων με ετικέτες: όνομα, επίθετο, ηλικία.

```
<people>
    <person>
        <first>Kostas</first>
        <last>Milo<last>
        <age>26</age>
```

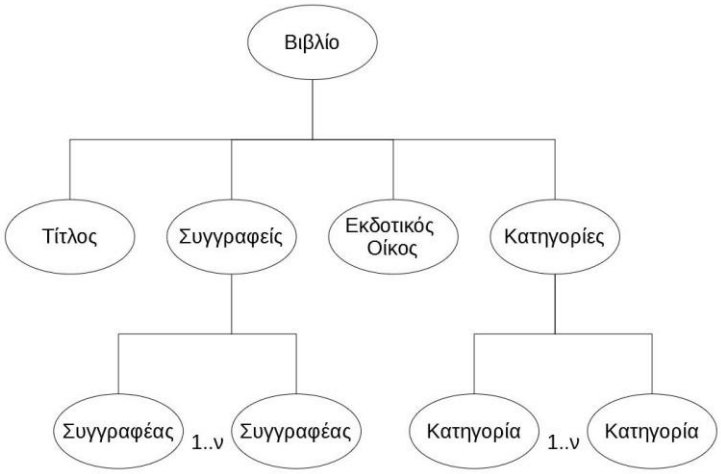
```
    </person>
    <person>
        <first>Dimitris</first>
        <last>Milo<last>
        <age>56</age>
    </person>
</people>
```

Υπάρχει μία ποικιλία εφαρμογών Application Programming Interface (API), που μπορούν να χρησιμοποιούν οι προγραμματιστές, για να προσπελαύνουν δεδομένα XML. Στην παρούσα εργασία θα εξεταστεί η διαχείριση δεδομένων XML σε περιηγητές, δηλαδή σε εφαρμογές διαδικτύου.

Τα αρχεία XML μπορούν να αναπαραχθούν με δομή πίνακα (array), η οποία είναι η πιο απλή και υπεύθυνη για το μεγαλύτερο μέρος των περισσότερων εγγράφων XML (Εικόνα 2-2). Παρόλα αυτά υπάρχει και η ακανόνιστη-ασύμμετρη μορφή (irregular-unbalanced), η οποία μπορεί να περιλαμβάνει εμφωλιασμένα πεδία σε διαφορετικό βάθος κάθε φορά (Εικόνα 2-3).



Εικόνα 2-2: Παράδειγμα XML με δομή πίνακα



Εικόνα 2-3: Παράδειγμα XML με ακανόνιστη δομή

2.1.3 JavaScript Object Notation

Το JSON είναι ένα ανάλαφρο πρότυπο ανταλλαγής δεδομένων. Σχεδιαστικά υλοποιήθηκε για να διευκολύνει τον άνθρωπο τόσο στην ανάγνωση όσο και στη σύνταξη τέτοιων εγγράφων. Θεωρείται ότι είναι εξίσου εύκολο και για τα μηχανήματα να το αναλύσουν και να το παράγουν. Βασίζεται σε ένα υποσύνολο προτύπου γλώσσας προγραμματισμού JavaScript ECMA-262 3ης Έκδοσης - Δεκέμβριος 1999. Το JSON είναι μια μορφή κειμένου που είναι πλήρως ανεξάρτητη από τη γλώσσα, αλλά χρησιμοποιεί συμβατικές πρακτικές που είναι γνωστές στους προγραμματιστές της οικογένειας γλωσσών C, συμπεριλαμβανομένων των C, C++, C#, Java, JavaScript, Perl, Python και πολλές άλλες.

Όπως υποστηρίζει ο Douglas Crockford, ανακάλυψε το JSON, δεν το εφηύρε (Severance, 2012). Ισχυρίζεται ότι η καινοτομία του δεν ήταν κάτι άλλο παρά ένα πρότυπο μέσα σε πολλά άλλα που υπήρχαν ήδη. Τεχνικά ισοδύναμο, αλλά συντακτικά διαφορετικό, με φυσικούς τρόπους για αναπαράσταση διάφορων δομών δεδομένων που χρησιμοποιούν οι γλώσσες προγραμματισμού.

Οι πιο συνηθισμένες δομές που χρησιμοποιούμε στον προγραμματισμό, είναι οι κλιμακωτές μεταβλητές, οι γραμμικές λίστες και τα ζεύγη κλειδιών-τιμών. Το JSON αναπαριστά αυτές τις δομές με το πιο φυσικό και άμεσο serialization, μειώνοντας σημαντικά την αναντιστοιχία μεταξύ της αναπαράστασης των δομών στη μνήμη, στις

εφαρμογές και κατά την επικοινωνία. Το JSON δεν είναι μόνο βολικό, αλλά είναι και αποτελεσματικό.

Ένα μικρό παράδειγμα που δείχνει τη σύνταξη JSON αρχείων φαίνεται παρακάτω, όπου το JSON παρουσιάζει πληροφορίες ανθρώπων: όνομα, επίθετο, ηλικία.

```
{
  "people" : [
    {
      "Name" : "Kostas",
      "Surname" : "Milo",
      "Age" : 26
    },
    {
      "Name" : "Dimitris",
      "Surname" : "Milo",
      "Age" : 56
    }
  ]
}
```

2.1.4 YAML Ain't a Markup Language

Το YAML είναι ακόμη ένα πρότυπο που ασχολείται με την αποθήκευση και τη μεταφορά δεδομένων. Είναι σχεδιασμένο για να είναι φιλικό προς τον άνθρωπο και για να λειτουργεί καλά με σύγχρονες γλώσσες προγραμματισμού για κοινές καθημερινές διεργασίες (Ben-Kiki & Evans, 2007). Υποστηρίζει κωδικοποίηση Unicode, για όλες τις γλώσσες του κόσμου και μερικοί από τους χαρακτήρες δεσμεύονται για να παρέχουν δομικές πληροφορίες.

Μπορεί να χρησιμοποιηθεί για όλες τις δομές δεδομένων, αλλά χειρίζεται με μεγάλη ευκολία τις παρακάτω: mappings (hashes/dictionaries), sequences (arrays/lists) και scalars (strings/numbers). Το YAML αξιοποιεί τις δομές αυτές, δημιουργώντας ένα απλό σύστημα, για το serialization οποιασδήποτε δομής δεδομένων. Ενώ οι περισσότερες γλώσσες προγραμματισμού μπορούν να χρησιμοποιήσουν το YAML για επεξεργασία των δεδομένων, το YAML λειτουργεί καλύτερα με τις γλώσσες

προγραμματισμού που είναι βασικά κατασκευασμένες γύρω από αυτές τις τρεις δομές δεδομένων. Τέτοιες γλώσσες, σύγχρονες και ευέλικτες, είναι η Perl, Python, PHP, Ruby και Javascript.

Το YAML δημιουργήθηκε ειδικά για να λειτουργήσει καλά σε απλές περιπτώσεις χρήσης, όπως: αρχεία ρύθμισης παραμέτρων (configuration files), αρχεία καταγραφής (log), ανταλλαγή δεδομένων σε διαφορετικά συστήματα και εντοπισμός σφαλμάτων σε πολύπλοκες δομές δεδομένων. Έχει σαν στόχο, να είναι ευανάγνωστη στους ανθρώπους, να παρέχει φορητότητα δεδομένων μεταξύ διαφορετικών συστημάτων, να είναι επεκτάσιμη και να είναι εύκολη στην υλοποίηση και στη χρήση.

Αξίζει να αναφερθεί ότι δεν υπάρχει συσχετισμός μεταξύ του YAML και του XML. Το XML σχεδιάστηκε για να είναι συμβατό με το πρότυπο γενικευμένης γλώσσας σήμανσης (SGML) και να υποστηρίζει τη δομημένη τεκμηρίωση. Επομένως, το XML είχε εξαρχής πολλούς περιορισμούς σχεδιασμού, που το YAML δεν έχει. Το YAML επειδή είναι μεταγενέστερο πρότυπο, έχει διδαχτεί από το XML και άλλα τέτοια πρότυπα.

Φαίνεται ότι το YAML έχει αρκετά κοινά με το JSON, όσον αφορά τους στόχους και την παρόμοια σύνταξή τους. Παρόλα αυτά είναι πιο πολύπλοκο και με περισσότερη πληροφορία. Μπορεί να θεωρηθεί το YAML ένα υπερσύνολο του JSON. Κάθε JSON αρχείο δηλαδή, είναι επίσης ένα έγκυρο αρχείο YAML.

Ένα μικρό παράδειγμα που δείχνει τη σύνταξη YAML αρχείων φαίνεται παρακάτω, όπου το YAML παρουσιάζει πληροφορίες ανθρώπων: όνομα, επίθετο, ηλικία.

```
People:
    - Name: “Kostas”
      Surname: “Milo”
      Age: 26
    - Name: “Dimitris”
      Surname: “Milo”
      Age: 56
```

2.1.5 Σύγκριση Προτύπων

Τα πρότυπα που χρησιμοποιούνται περισσότερο σε διαδικτυακές εφαρμογές, είναι τα XML και JSON. Το CSV εξακολουθεί να χρησιμοποιείται σε άλλες εφαρμογές κυρίως λόγω της απλότητάς του. Το YAML είναι σχετικά νέο πρότυπο, δεν έχει χρησιμοποιηθεί αρκετά.

Παρότι παλαιότερα φαινόταν να υπερισχύει η χρήση του XML, σήμερα φαίνεται να χρησιμοποιούνται εξίσου και τα δύο. Υπάρχουν όμως κάποιες βασικές διαφορές ανάμεσα σε αυτά τα δύο πρότυπα (Πίνακας 2-1).

Πίνακας 2-1: Σύγκριση XML και JSON

	XML	JSON
Ευανάγνωστο	Σχετικά εύκολο να διαβαστεί, καθώς τα δεδομένα περιέχονται με ετικέτες σήμανσης	Πολύ εύκολο να διαβαστεί, καθώς βασίζεται στον ορισμό αντικειμένων και τιμών
Μέγεθος κώδικα	Περισσότερος κώδικας	Λιγότερος κώδικας
Ταχύτητα ανάλυσης	Πιο αργό, καθώς το κείμενο πρέπει να παρθεί μέσα από τις ετικέτες	Πιο γρήγορο, τα δεδομένα είναι σαφώς καθορισμένα ως αντικείμενο και τιμή
Δημιουργία	Ελάχιστα δυσκολότερη σύνταξη	Ευκολία στη δημιουργία, καθώς το συντακτικό είναι πιο απλό
Ευελιξία και Επεκτασιμότητα	Αρκετά συναφής με τον προγραμματισμό, επομένως χρειάζεται περισσότερη γνώση, αλλά υπάρχει μεγαλύτερη ευελιξία	Λειτουργεί με περιορισμένο τύπο δεδομένων, το οποίο μπορεί να μην είναι επαρκές για όλες τις εφαρμογές
Ασφάλεια	Το XML είναι περισσότερο ασφαλές	Επειδή το JSON είναι υποσύνολο της JavaScript, μπορεί χρησιμοποιηθεί για

		να τρέξει κακόβουλο κώδικα

Σημείωση: Τα παραπάνω απορρέουν από την εμπειρία του γράφοντος.

Προκύπτει λοιπόν, ότι εκεί που υστερεί το XML υπερτερεί το JSON και το αντίστροφο. Για αυτόν το λόγο χρησιμοποιούνται και τα δύο, ανάλογα με τις ανάγκες του προγραμματιστή. Σε όλα τα παραπάνω πρότυπα, απαιτείται επεξεργασία η οποία περιλαμβάνει διάφορα στάδια μέχρι να εξαχθεί η χρήσιμη πληροφορία.

Επειδή το XML είναι συνδεδεμένο με την HTML, που είναι η γλώσσα παρουσίασης διαδικτυακών εφαρμογών, χρησιμοποιείται κατά κόρον για την ανταλλαγή δεδομένων μεταξύ τέτοιων εφαρμογών και αποτελεί αντικείμενο της παρούσας έρευνας. Τα αρχεία XML προσφέρουν αρκετή ευελιξία και μπορεί να αποβούν αρκετά σύνθετα. Η τελική πληροφορία περικλείεται από λέξεις κλειδιά προκειμένου να εξασφαλιστεί η ακεραιότητά της και ο παραλήπτης θα πρέπει να την αποσπάσει από αυτές. Αυτό, γίνεται με επεξεργασία που περιλαμβάνει διάφορα στάδια, είναι γνωστή με την ορολογία parsing και είναι αρκετά χρονοβόρα.

Στην συγκεκριμένη έρευνα, θα εξεταστεί σε βάθος η διαδικασία του parsing, προκειμένου να εντοπιστούν τα σημεία που απαιτούν περισσότερο χρόνο. Θα εξεταστούν διάφοροι υπάρχοντες αλγόριθμοι, θα εντοπιστούν ενδεχόμενα προβλήματα και θα μελετηθούν τρόποι βελτιστοποίησης για τη μείωση του χρόνου parsing ενός XML αρχείου.

2.2 Μοντέλα Επεξεργασίας

Υπάρχουν ήδη έτοιμα μοντέλα επεξεργασίας, που έχουν υλοποιηθεί για να προσπελαύνουν και να διαχειρίζονται έγγραφα, να κατασκευάζουν και να αναπαριστούν τα δεδομένα τους (Πίνακας 2-2). Η διαδικασία προσπέλασης μπορεί να γίνει μετά την ολοκλήρωση της λήψης του εγγράφου ή κατά τη διάρκεια λήψης (streaming) (Oliveira et al., 2013). Στην πρώτη κατηγορία, κατά γενική ομολογία, χρησιμοποιούνται δομές δεδομένων και αποθηκεύονται στην μνήμη, είτε σαν αντικείμενα σε δεντρικές δομές, είτε σαν πίνακες. Στην δεύτερη κατηγορία, streaming, χρησιμοποιούνται τα μοντέλα push και pull. Διάφορα μοντέλα που ανήκουν στην πρώτη κατηγορία είναι το DOM,

JDOM, dom4j, XOM, VTD κ.α., ενώ στην δεύτερη επικρατούν το Simple API for XML (SAX) και το Streaming API for XML (StAX).

Πίνακας 2-2: Ανάλυση APIs (Oliveira et al., 2013)

API	Μοντέλο Parsing
DOM	Μνήμη: δεντρική δομή αντικειμένου
JDOM	Μνήμη: δεντρική δομή αντικειμένου
Dom4j	Μνήμη: δεντρική δομή αντικειμένου
XOM	Μνήμη: δεντρική δομή αντικειμένου
VTD	Μνήμη: δομή πίνακα
SAX	Κατά τη ροή: μοντέλο push
StAX	Κατά τη ροή: μοντέλο pull

Η παρούσα έρευνα αφορά εφαρμογές σε περιβάλλον περιηγητών διαδικτύου και θα χρησιμοποιηθεί το μοντέλο ανάλυσης Document Object Model (DOM). Το DOM είναι μια διεπαφή ανεξάρτητη πλατφόρμας και γλώσσας προγραμματισμού (*DOM Standard*, 2020), που επιτρέπει στα προγράμματα να έχουν δυναμική πρόσβαση και να ενημερώνουν το περιεχόμενο, τη δομή και το στυλ ενός εγγράφου.

2.3 Document Object Model

Στο DOM, ένα έγγραφο XML, αναπαρίσταται σαν ένα δέντρο, όπου αρχή του είναι το βασικό στοιχείο αναφοράς ή αλλιώς ρίζα (root) και κάτω από αυτό υπάρχουν άλλα στοιχεία τα οποία με τη σειρά τους μπορεί να περιέχουν άλλα στοιχεία (elements). Κάθε στοιχείο περικλείεται από ετικέτες (tags). Κάθε tag αποτελεί και έναν κόμβο του δέντρου, το οποίο αντίστοιχα αν έχει άλλα εμφωλευμένα tags θα διαθέτει κι άλλους κόμβους κάτω από αυτό. Μπορούμε να το παρομοιάσουμε με γενεαλογικό δέντρο, όπου ένας γονιός έχει παιδιά, τα παιδιά του γονιού γίνονται γονείς για τα δικά τους παιδιά κ.ο.κ.

Το DOM διευκολύνει την πλοήγηση στο δέντρο. Για κάθε κόμβο υπάρχουν αναφορές με βάση την θέση του μέσα στο δέντρο. Πιο συγκεκριμένα:

- **parentNode**

  Είναι ο κόμβος-πατέρας από τον κόμβο στον οποίο βρισκόμαστε την εκάστοτε

στιγμή. Αν βρισκόμαστε σε ένα στοιχείο που δεν έχει πατέρα, όπως το βασικό σημείο αναφοράς (root), τότε το parentNode είναι κενό, null.

- **childNodes**

  Όλα τα αντικείμενα τα οποία είναι παιδιά του εκάστοτε κόμβου.

- **firstChild, lastChild**

  Το πρώτο ή το τελευταίο παιδί ενός κόμβου που βρισκόμαστε εκείνη την στιγμή. Αν κόμβος δεν έχει παιδιά εμφανίζεται η τιμή null.

- **nextSibling, previousSibling**

  Ο επόμενος ή ο προηγούμενος αδερφός ενός κόμβου. Δυο ή και παραπάνω παιδιά του ίδιου κόμβου είναι αδέρφια.

- **nodeType**

  Ο τύπος του κόμβου που εξετάζουμε εκείνη την στιγμή. Κόμβοι εγγράφου, κόμβοι στοιχείων, κόμβοι κειμένου, κόμβοι σχολίων και οι κόμβοι τεμαχισμένου εγγράφου.

- **nodeValue**

  Το περιεχόμενο (τιμή) ενός κόμβου.

- **nodeName**

  Το όνομα tag ενός στοιχείου που αποτελεί τον κόμβο.

Για να χρησιμοποιήσει κάποιος μία από τις παραπάνω ιδιότητες θα πρέπει να βρίσκεται ήδη σε ένα συγκεκριμένο κόμβο-στοιχείο.

2.4 Parsing

Με τον όρο parsing εννοείται η ειδική επεξεργασία με σκοπό την ανάκτηση της χρήσιμης πληροφορίας από ένα πολύπλοκο έγγραφο. Αυτό χωρίζεται σε διάφορα στάδια. Προεπεξεργασία, εξαγωγή, επικύρωση και εκμετάλλευση δεδομένων.

Ιδανικά, κάθε αλγόριθμος, πριν προβεί σε parsing, ακολουθεί βήματα προεπεξεργασίας, preparsing. Απαιτείται έλεγχος για να αποφασιστεί αν μπορεί να γίνει το parsing ή αν η δομή του αρχείου XML είναι λανθασμένη. Αυτό συμβαίνει για να διαπιστωθεί αν τα δεδομένα που περιέχονται στο έγγραφο μπορούν να ανακτηθούν σωστά, χωρίς να χρειαστεί να ολοκληρωθεί η επεξεργασία του, σπαταλώντας άσκοπα τους διαθέσιμους πόρους του υπολογιστή.

Εφόσον δεν παρατηρηθεί πρόβλημα στη διαδικασία της προεπεξεργασίας, ο αλγόριθμος συνεχίζει στη διαδικασία του parsing. Εδώ πρέπει να παρθεί η χρήσιμη πληροφορία. η οποία βρίσκεται μέσα σε tags. Ενδεχομένως να υπάρχουν και εμφωλευμένα tags, δηλαδή tag μέσα σε tag. Επομένως, θα πρέπει να γίνει «ξεδίπλωμα», μέχρις ότου να παρθεί το πεδίο με τη χρήσιμη πληροφορία. Κατά τη διάρκεια του parsing, ο αλγόριθμος θα πρέπει ιδανικά πάλι να εκτελεί και τους απαραίτητους ελέγχους προκειμένου να εξακριβώνεται ότι η χρήσιμη πληροφορία δεν έχει αλλοιωθεί.

2.4.1 Στάδια parsing

Το parsing XML αρχείων στο DOM, γίνεται σε δύο βασικά στάδια. Στο πρώτο στάδιο, γίνεται η ανάγνωση του αρχείου και η δημιουργία ενός δένδρου στη μνήμη, επικυρώνοντας ταυτόχρονα την ορθότητα των δεδομένων. Στο δεύτερο στάδιο προσπελαύνεται το δένδρο που δημιουργήθηκε και γίνεται ουσιαστική επεξεργασία των δεδομένων (καταχώρηση σε βάση κ.ο.κ.).

Κατά το πρώτο στάδιο, χρησιμοποιούνται διάφορες τεχνικές, όπως στοίβες Last In First Out - LIFO, πίνακες χαρτογράφησης - Maps κ.α., προκειμένου να δημιουργηθεί μια συνδεδεμένη λίστα (linked list) για την κατασκευή του δέντρου DOM. Η δημιουργία ενός DOM δέντρου γίνεται με την απόσπαση tags, δηλαδή των ετικετών έναρξης και λήξης. Κατά την ανάγνωση ενός XML αρχείου, ό,τι περικλείεται ανάμεσα σε ετικέτα αρχής και λήξης, προστίθεται σαν ένα κλαδί-κόμβο (node) στο δέντρο.

Όταν μια ετικέτα στοιχείου έναρξης <e> διαβάζεται, ένας κόμβος DOM δημιουργείται για το στοιχείο e και κάθε ζεύγος χαρακτηριστικό – τιμή (key - value) που συσχετίζεται με το στοιχείο, αναλύεται και αποθηκεύεται, δημιουργώντας τα απαραίτητα DOM nodes. Η σειρά των παιδιών ακολουθεί τη σειρά με την οποία εμφανίζονται τα στοιχεία στο έγγραφο. Όταν συναντάται κλείσιμο μιας ετικέτας </e>, γίνεται έλεγχος με την ετικέτα έναρξης. Εάν τα ονόματα δεν ταιριάζουν, η ανάλυση τερματίζει καθώς το έγγραφο είναι λάθος διαμορφωμένο. Όταν αναγνωσθεί ο τελευταίος χαρακτήρας του εγγράφου, δεν θα πρέπει να υπάρχει ετικέτα που δεν έχει κλείσει, αλλιώς η διαδικασία απορρίπτεται.

Στο δεύτερο στάδιο γίνεται η ανάκτηση, parse, των τιμών από τους κόμβους του δέντρου, που δημιουργήθηκε στην μνήμη από το προηγούμενο στάδιο. Μετά την

ανάκτηση, ακολουθεί η επεξεργασία ή οποιαδήποτε άλλη εκμετάλλευσή τους. Ταυτόχρονα, βεβαιώνεται ότι η χρήσιμη πληροφορία δεν έχει αλλοιωθεί.

2.4.2 Σειριακή Προσέγγιση

Στην σειριακή επεξεργασία, ξεκινάει η διαδικασία του πρώτου σταδίου, η οποία εκτελείται από ένα μοναδικό νήμα. Διαβάζεται σειριακά το αρχείο, γίνεται έλεγχος των ετικετών και δημιουργείται το δέντρο στη μνήμη. Στη συνέχεια, στο δεύτερο στάδιο, που εκτελείται από το ίδιο νήμα διατρέχεται το δένδρο που έχει δημιουργηθεί στη μνήμη, σε κάθε κόμβο διανύονται όλα τα παρακλάδια του, ανακτώνται οι τιμές και αξιοποιούνται οι πληροφορίες.

Τα τελευταία χρόνια οι πιο βασικοί, απλοί και συνηθισμένοι αλγόριθμοι βασίζονταν στη σειριακή επεξεργασία του εγγράφου. Κατά τη σειριακή επεξεργασία, διαβάζεται ένα έγγραφο γραμμή-γραμμή και κατόπιν γίνεται η προσπάθεια για ανάκτηση της χρήσιμης πληροφορίας. Είναι εύκολα αντιληπτό, ότι η σειριακή επεξεργασία σε ένα πολύπλοκο αρχείο, με πολλά εμφωλευμένα tags ή με κενά στοιχεία (elements), υστερεί. Το μέγεθος του αρχείου επιτείνει το πρόβλημα. Το βασικότερο μειονέκτημα είναι ότι είναι χρονοβόρα διαδικασία και δεν εκμεταλλεύεται τη διαθέσιμη υπολογιστική ισχύ (ακόμα και οι οικιακοί υπολογιστές διαθέτουν πολλαπλούς πυρήνες).

Μελετώντας αλγορίθμους που χρησιμοποιούν σειριακή επεξεργασία, προκύπτει ότι καθυστερούν πολύ σε μεγάλα και σε πολύπλοκα αρχεία. Λύση σε τέτοιου είδους προβλήματα, στον χώρο των υπολογιστών και της πληροφορίας έδινε ανέκαθεν η χρήση της παράλληλης επεξεργασίας.

2.4.3 Παράλληλη Προσέγγιση

Στην παράλληλη επεξεργασία, ένα πολύπλοκο πρόβλημα διασπάται σε μικρότερα προβλήματα που ανατίθενται σε διαφορετικές διεργασίες-νήματα. Με την ολοκλήρωση των διεργασιών πρέπει να συντεθούν τα αποτελέσματα και να σχηματιστεί η τελική λύση.

Το κυριότερο πρόβλημα της παράλληλης επεξεργασίας είναι ο συγχρονισμός των νημάτων. Σε πολλές περιπτώσεις ένα νήμα χρειάζεται πληροφορίες από άλλα νήματα, επομένως θα πρέπει να βεβαιώνεται ότι έχει ολοκληρωθεί η επεξεργασία δεδομένων από αυτά. Επίσης, θα πρέπει να εξασφαλίζεται, ότι διαφορετικά νήματα δεν επεξεργάζονται

τα ίδια δεδομένα, κλειδώνοντας τις περιοχές της μνήμης που χρησιμοποιούν. Αυτό πρέπει να γίνεται με πολύ προσοχή προκειμένου να αποφευχθεί κλείδωμα της ίδιας περιοχής από δύο ή περισσότερα νήματα, με αποτέλεσμα να μπουν σε ατέρμονη αναμονή και να παγώσει η επεξεργασία (deadlock).

Κατά τη σχεδίαση παράλληλης επεξεργασίας, πρέπει να ληφθούν υπόψη δύο παράγοντες, η στρατηγική διαμοιρασμού και το βάθος στο οποίο θα σταματήσει ο διαμοιρασμός (Rao & Kumar, 1987). Ο διαμοιρασμός του προβλήματος και η ανάθεση του σε νήματα θα πρέπει να γίνει με τέτοιο τρόπο, που να περιορίζονται προβλήματα συγχρονισμού ή προστασίας δεδομένων. Επίσης, πρέπει να δοθεί προσοχή και στο πλήθος των νημάτων που θα εκκινηθούν σε σχέση με τους διαθέσιμους πόρους αλλά και την πολυπλοκότητα του προβλήματος. Οι παραπάνω παράγοντες κρίνονται πολύ βασικοί προκειμένου να είναι αποδοτική η παράλληλη επεξεργασία. Ένας κακός σχεδιασμός μπορεί να επιφέρει αντίθετα αποτελέσματα από τα προσδοκόμενα.

Στην επεξεργασία XML, για τη δημιουργία του δέντρου, στο πρώτο στάδιο, με παράλληλη επεξεργασία, εκτελείται αλγόριθμος που ακολουθεί δύο βήματα:

1. Διάσπαση του αρχείου XML σε κομμάτια και κατασκευή δομών δέντρου (κλαδιά) στη μνήμη.
2. Σύνδεση των δομών του προηγούμενου βήματος και κατασκευή του DOM δέντρου.

Θεωρείται ότι n πυρήνες επεξεργαστών είναι διαθέσιμοι. Σε κάθε πυρήνα μπορεί να ανατεθεί ένα σύνολο από κομμάτια. Έπειτα, κάθε νήμα επεξεργάζεται ένα κομμάτι ανά φορά και δημιουργεί συνδέσεις γονέα-παιδιού, όπως απαιτείται. Για την επεξεργασία των δεδομένων στο δεύτερο στάδιο, μπορούμε να ξεκινήσουμε n νήματα, ανάλογα με τους διαθέσιμους πυρήνες, με κάθε νήμα να επεξεργάζεται ένα κλαδί του δέντρου μέχρι το τέλος του.

2.4.3.1 Partitioning

Ο διαμοιρασμός-partitioning είναι μια λογική διαίρεση ενός αντικειμένου (αρχείου, δομών δεδομένων κ.τ.λ.), σε ξεχωριστά ανεξάρτητα κομμάτια. Η κατάτμηση ενδείκνυται για πολλούς λόγους, όπως ευκολότερη διαχείριση μεγάλου όγκου δεδομένων, βελτιστοποίηση της απόδοσης των συστημάτων ή για καλύτερη κατανομή φορτίου σε αυτά.

Η επιλογή τεχνικής διαμοιρασμού είναι ο πιο σημαντικός παράγοντας στην παράλληλη επεξεργασία. Ίσως είναι και το πιο σημαντικό βήμα κατά την προετοιμασία ενός προβλήματος που θα λυθεί με παράλληλη επεξεργασία. Η τεχνική που θα επιλεχθεί είναι αυτή που συνήθως καθορίζει και την απόδοση της λύσης (Balasundaram et al., 1991).

Για να είναι αποτελεσματική οποιαδήποτε τεχνική, πρέπει να επιτευχθεί εξισορρόπηση φορτίου ανά κομμάτι, για να αποφευχθούν τυχόν αδρανή νήματα. Πρέπει επίσης να δοθεί προσοχή στο που θα σταματήσει ο διαμοιρασμός και στο πλήθος των νημάτων που θα ξεκινήσουν. Στην ιδανική περίπτωση, θα πρέπει να δημιουργηθούν τόσα κομμάτια, έτσι ώστε σε κάθε νήμα να αντιστοιχεί ίσος αριθμός κομματιών και κάθε νήμα να αναλώνει τον ίδιο χρόνο για επεξεργασία, χωρίς να υπάρχουν αναμονές ή κλειδώματα. Επειδή αυτό είναι δύσκολο να επιτευχθεί, ανάλογα με τη φύση του προβλήματος, χρησιμοποιούνται διαφορετικές τεχνικές, το στατικό partitioning, το δυναμικό partitioning ή συνδυασμός και των δύο (Lu et al., 2006).

2.4.1.3.1 Στατικό Partitioning

Κατά το στατικό partitioning, διαχωρίζεται ένα αντικείμενο σε ισομεγέθη κομμάτια. Το πλεονέκτημα της τεχνικής αυτής είναι η ευκολία και η ταχύτητα διαμοιρασμού. Αυτό δε σημαίνει απαραίτητα ότι θα δημιουργηθεί ισορροπημένο φορτίο για κάθε νήμα και θα οδηγήσει σε καλό παραλληλισμό, γιατί δεν γνωρίζουμε την ανάγκη επεξεργασίας κάθε κομματιού.

Στην επεξεργασία αρχείων XML, η διαδικασία αυτή υλοποιείται πριν από το parsing, επομένως γνωρίζει λίγα για το περιεχόμενο του parsing (π.χ. namespace declarations). Για έγγραφα XML με δομή πίνακα, το στατικό partitioning παρέχει τον καλύτερο δυνατό παραλληλισμό. Αυτό συμβαίνει επειδή ένας γραμμικός πίνακας μπορεί εύκολα να χωριστεί σε ίσα κομμάτια. Το partitioning γίνεται σειριακά από την αρχή προς το τέλος, έτσι ώστε κάθε κομμάτι να είναι συνεχόμενο στο έγγραφο XML. Αξίζει να σημειωθεί ότι τέτοια δομή έχει η πλειοψηφία των μεγάλων εγγράφων XML.

Αντίθετα, η στρατηγική στατικού partitioning δεν είναι πρακτική για έγγραφα XML με ακανόνιστες δομές δέντρων, λόγω της ισχυρής εξάρτησης μεταξύ των διαφορετικών βημάτων επεξεργασίας. Αυτό μπορεί να δημιουργήσει ένα επιπλέον

στάδιο επεξεργασίας, κατά το οποίο θα πρέπει να ενωθούν τα επιμέρους κλαδιά που δημιουργήθηκαν, προκειμένου να σχηματιστεί το δέντρο.

2.4.1.3.2 Δυναμικό Partitioning

Σε αντίθεση με το στατικό partitioning, η τεχνική αυτή διαμοιράζει ένα αντικείμενο και παράγει δυναμικά κομμάτια, όχι με βάση το μέγεθος, αλλά κάνοντας μια προεπεξεργασία, προσπαθεί να κατανείμει καλύτερα το φορτίο στις διεργασίες που θα ξεκινήσει, με βάση τη φύση του προβλήματος.

Στο πρώτο στάδιο επεξεργασίας ενός XML αρχείου, είναι δύσκολο να επιτευχθεί πλήρης ισοκατανομή φορτίου στα νήματα, γιατί η προεπεξεργασία που θα απαιτηθεί, θεωρείται δυσανάλογη του κέρδους. Παρόλα αυτά, μπορεί να αποφασιστεί να χρησιμοποιηθεί δυναμικό partitioning σε μια προσπάθεια να κατακερματιστεί σωστότερα το XML αρχείο, ούτως ώστε να χρειαστεί λιγότερη επεξεργασία κατά σύνδεση των κλαδιών, για τη δημιουργία του δέντρου DOM.

Στο δεύτερο στάδιο μπορεί να χρησιμοποιηθεί ευκολότερα το δυναμικό partitioning. Αυτό, γιατί το δέντρο έχει κατασκευαστεί ήδη από το πρώτο στάδιο στη μνήμη και μπορεί με ευκολία να προσπελαστεί με διάφορους αλγορίθμους (π.χ. Depth-First Search, DFS), να ελεγχθεί η επεξεργασία που απαιτείται για κάθε κομμάτι, με σκοπό να διαμοιραστεί ισότιμα στα διαθέσιμα νήματα.

2.4.4 Υβριδική Προσέγγιση

Κατά την υβριδική προσέγγιση χρησιμοποιείται και η σειριακή και η παράλληλη. Στόχος είναι να χρησιμοποιηθούν και οι δύο προσεγγίσεις σε όποιο στάδιο της επεξεργασίας κρίνεται απαραίτητο, με σκοπό τη βελτιστοποίησή της.

Σε περιπτώσεις που ο όγκος των δεδομένων είναι μικρός, η κατάτμηση και η εκκίνηση νημάτων μάλλον επιβαρύνει τη διαδικασία, οπότε η σειριακή επεξεργασία είναι μονόδρομος. Αντίθετα σε περιπτώσεις όπου ο όγκος των προς επεξεργασία δεδομένων είναι μεγάλος, το κέρδος από την παράλληλη επεξεργασία θεωρείται δεδομένο.

Σε ενδιάμεσες περιπτώσεις, όπου ο όγκος δεν είναι ούτε πολύ μικρός, ούτε πολύ μεγάλος, η υβριδική επεξεργασία μπορεί να αποτελεί τη βέλτιστη λύση,

χρησιμοποιώντας τη σειριακή επεξεργασία για τις απλές διαδικασίες και την παράλληλη για τις συνθετότερες.

2.4.5 Πλεονεκτήματα-Μειονεκτήματα

Κλασικό πλεονέκτημα της σειριακής επεξεργασίας, είναι η απλότητα των αλγορίθμων και η ευκολία ελέγχου της όλης διαδικασίας. Βασικό μειονέκτημα, είναι η μη αξιοποίηση της σύγχρονης επεξεργασίας, αφού ακόμα και οι οικιακοί υπολογιστές διαθέτουν αρκετούς πυρήνες.

Αυτό οδηγεί στην αναζήτηση νέων μεθόδων για την πλήρη αξιοποίηση της σύγχρονης τεχνολογίας, με χρήση αλγορίθμων παράλληλης επεξεργασίας. Αρκετά σημαντικό για να αποφασιστεί αν θα χρησιμοποιηθεί παράλληλη επεξεργασία, είναι το μέγεθος του υπό επεξεργασία αρχείου σε συνδυασμό με την πολυπλοκότητα του, σε σχέση με το βάθος εμφωλιασμού.

Ένας πρώτος έλεγχος μπορεί να γίνει ως προς το μέγεθος του αρχείου, οπότε αν το αρχείο έχει σημαντικό μέγεθος μπορεί να εκτελεστεί και το πρώτο στάδιο με παράλληλη επεξεργασία, χωρίς όμως να μπορεί να εκτιμηθεί η πολυπλοκότητά του. Διαφορετικά, αν είναι μικρό το έγγραφο, εκτελείται το πρώτο στάδιο σειριακά και κατά τη δημιουργία του δένδρου, ελέγχεται η πολυπλοκότητα, ούτως ώστε να αποφασιστεί αν αξίζει να γίνει το δεύτερο στάδιο με παράλληλη επεξεργασία.

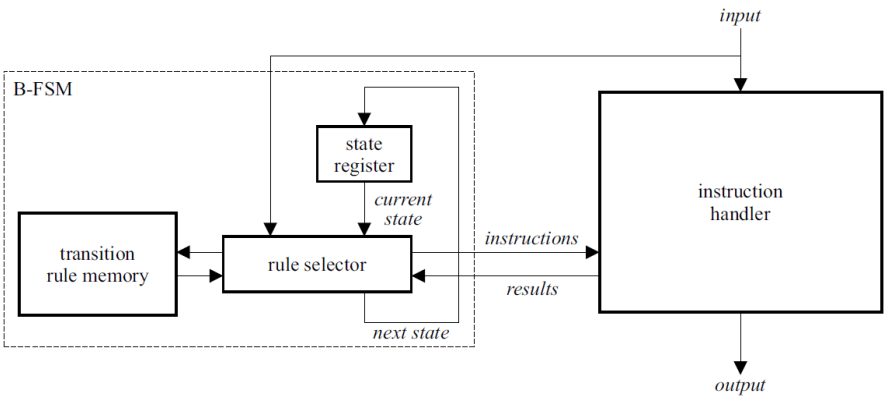
Σε περίπτωση που η διαδικασία σύνδεσης των κομματιών γίνεται ταυτόχρονα με τη δημιουργία δέντρων DOM, ο συγχρονισμός είναι απαραίτητος για να βεβαιώσει ότι οι συνδέσεις γονέα-παιδιού έχουν ενημερωθεί σωστά. Εδώ χρειάζεται να σημειωθεί ότι σύμφωνα με το πρότυπο XML, υπάρχει μια διάταξη μεταξύ αδερφών ανάλογα με τη σχετική θέση κατά την εισαγωγή τους στο έγγραφο.

Συνοψίζοντας, αν το αρχείο έχει μικρό σχετικά μέγεθος, ενδείκνυται το πρώτο στάδιο να γίνεται σειριακά και ανάλογα με την πολυπλοκότητα του, να αποφασίζεται αν το δεύτερο στάδιο θα γίνει σειριακά ή παράλληλα. Αν το αρχείο είναι πολύ μεγάλο, ενδείκνυται πλήρης παραλληλισμός.

2.4.6 Ερευνητικά Δεδομένα

2.4.6.1 Parallel XML Parsing με Finite State Machine

Για αρκετά χρόνια, η επεξεργασία XML αρχείων γινόταν σειριακά. Ήταν χρονοβόρα διαδικασία και άρχισαν μελέτες για τη βελτιστοποίηση της. Ένα από τα πρώτα ευρήματα των (Lunteren et al., 2004), το Zurich XML Accelerator (ZuXA), προορίστηκε για να ξεπεράσει τα εμπόδια απόδοσης που εμφανίζονται κατά την επεξεργασία ενός XML αρχείου. Η αρχιτεκτονική του ZuXA βασίστηκε σε μια μηχανή πεπερασμένης κατάστασης (Finite State Machine, FSM). Δηλαδή ένα μοντέλο αυτοματοποίησης, που καλείται να αλλάξει κατάσταση ανάλογα με τις παραμέτρους που λαμβάνει και ανάλογα με την κατάσταση εκτελεί τον αντίστοιχο αλγόριθμο. Αυτή η εναλλαγή καταστάσεων ονομάζεται μετάβαση (J. Wang, 2019) και συντελεί στην επιτάχυνση της συνολικής επεξεργασίας. Στην Εικόνα 2-4, απεικονίζεται η αρχιτεκτονική του ZuXA.



Εικόνα 2-4: Αρχιτεκτονική ZuXA

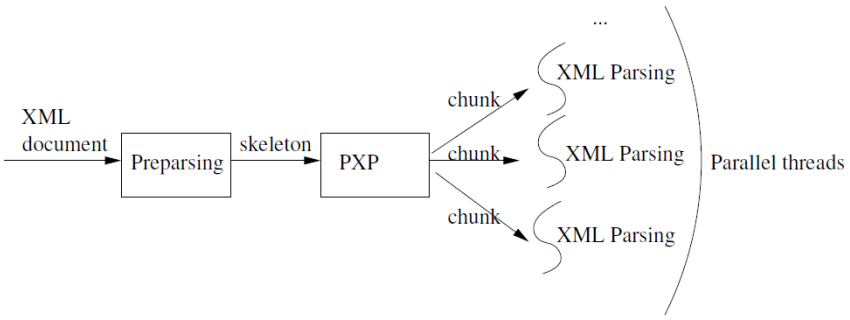
2.4.6.2 Parallel XML Parsing (2006)

Στη δημοσίευση των (Lu et al., 2006), παρουσιάστηκε η στρατηγική τους, PXP (Parallel XML Parsing) για το parsing ενός XML αρχείου, με υβριδική προσέγγιση. Αρχικά, εκτελείται σειριακά μια προεπεξεργασία στο αρχείο (preparsing), για τη δημιουργία σκελετού (skeleton) στη μνήμη. Σκελετός, ορίζεται ένα δέντρο το οποίο έχει μόνο τις απαραίτητες πληροφορίες δομής του αρχείου και είναι μικρότερο και απλούστερο από αυτό που δημιουργήθηκε από έναν πλήρη XML parser. Στο δεύτερο στάδιο, χρησιμοποιείται αυτό το δέντρο και ξεκινάει η διαδικασία του partitioning.

Διαχωρίζεται σε κομμάτια και κάθε κομμάτι διαμοιράζεται σε ένα νήμα ξεκινώντας τη διαδικασία του παράλληλου parsing.

Εξετάζεται η χρήση στατικού ή δυναμικού partitioning, ανάλογα με την πολυπλοκότητα της δομής του XML αρχείου. Έτσι, αν πρόκειται για απλές δομές (π.χ. array) προτείνει στατικό partitioning, ενώ για πιο σύνθετες δομές (π.χ. unbalanced tree) θεωρείται αποδοτικότερο το παράλληλο partitioning.

Στην Εικόνα 2-5, παρουσιάζεται η αρχιτεκτονική του αλγορίθμου.



Εικόνα 2-5: Αρχιτεκτονική PXP (Lu et al, 2006).

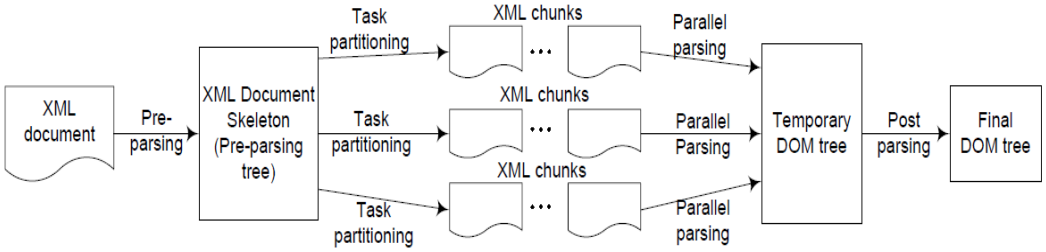
Επιβεβαιώνεται ότι η προεπεξεργασία, είναι πολύ σημαντική για τη βελτιστοποίηση της συνολικής διαδικασίας parsing, γιατί προετοιμάζει το δεύτερο στάδιο να γίνει με μεγαλύτερη ευκολία και ταχύτητα. Επίσης, δίνεται μεγάλη βαρύτητα στο σημείο όπου διαχωρίζεται ένα XML σε κομμάτια.

2.4.6.3 Parallel XML Parsing (2007)

Χρησιμοποιώντας τα θεμέλια της παραπάνω δημοσίευσης, μελετήθηκε περεταίρω η αρχιτεκτονική PXP (Y. Pan, Lu, et al., 2007).

Ξεκινώντας, προεπεξεργάζεται ένα αρχείο XML και δημιουργείται ο σκελετός του. Στη συνέχεια διαμοιράζεται το έγγραφο σε κομμάτια και ανατίθεται μια διαδικασία σε κάθε νήμα. Σε αυτή την έρευνα, εξετάζεται η δυνατότητα να εκκινεί ένα νήμα, νέα νήματα, ανάλογα με το μέγεθος του τμήματος που έχει αναλάβει, χωρίς όμως αυτό να αποβεί σε βάρος της απόδοσης. Κατά το παράλληλο parsing, δημιουργείται ένα προσωρινό DOM δέντρο και στη συνέχεια αφαιρούνται οι προσωρινοί κόμβοι και δημιουργείται το τελικό δέντρο.

Στην Εικόνα 2-6, φαίνεται η αρχιτεκτονική του αλγορίθμου PXP.

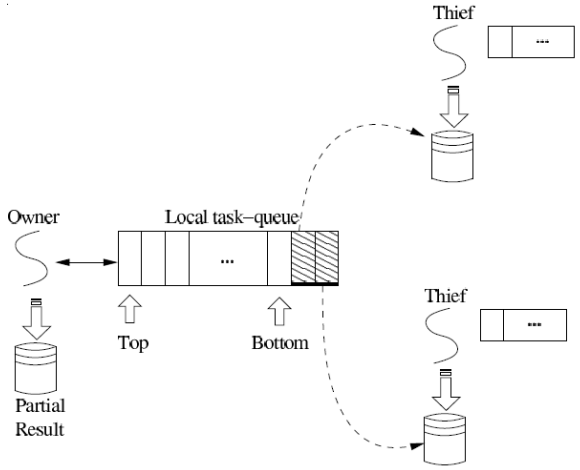


Εικόνα 2-6: Αρχιτεκτονική PXP (Y. Pan, Lu, et al., 2007).

2.4.6.4 Parallel XML Parsing με Work Stealing

Σε μια άλλη δημοσίευση (Lu & Gannon, 2007), παρουσιάστηκε ένα μοντέλο παράλληλης προσέγγισης. Το μοντέλο βασίζεται σε ένα μηχανισμό δυναμικής εξισορρόπησης φορτίου με τη μέθοδο work stealing, στο οποίο τα νήματα που τελειώνουν γρηγορότερα, «κλέβουν» δουλειά από νήματα που καθυστερούν. Η μέθοδος αυτή είναι μια γενικότερη μέθοδος η οποία μπορεί να εφαρμοστεί σε όλα τα στάδια επεξεργασίας ενός XML αρχείου.

Στην Εικόνα 2-7, απεικονίζεται η αρχιτεκτονική του work stealing.



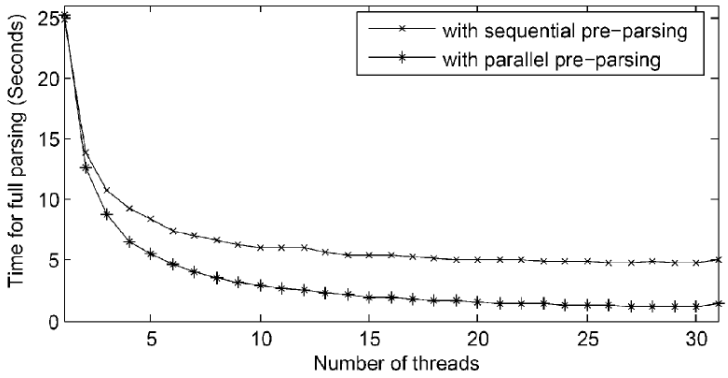
Εικόνα 2-7: Δυναμικό partitioning με χρήση work stealing (Lu & Gannon, 2007).

2.4.6.5 Parallel XML Parsing με Deterministic Finite Automaton

Στο τέλος της ίδιας χρονιάς, στη δημοσίευση των (Y. Pan, Zhang, et al., 2007), εξετάστηκε ο παραλληλισμός στο πρώτο στάδιο με τη χρήση meta-Deterministic Finite Automaton (meta-DFA) και αποδείχθηκε ότι μπορούν να επιτευχθούν επιπρόσθετες

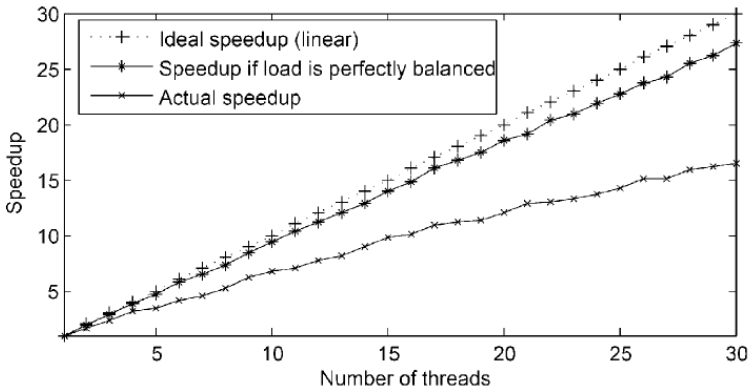
βελτιώσεις στην απόδοση. Το meta-DFA είναι μια μεταγενέστερη και βελτιωμένη έκδοση του Finite State Machine, το Deterministic FSM.

Στην Εικόνα 2-8, παρουσιάζεται το κέρδος που επιτυγχάνεται για όλη την επεξεργασία με τη χρήση παράλληλης προεπεξεργασίας σε σχέση με τη σειριακή.



Εικόνα 2-8: Σύγκριση χρόνου σειριακής και παράλληλης προεπεξεργασίας (Y. Pan, Zhang, et al., 2007).

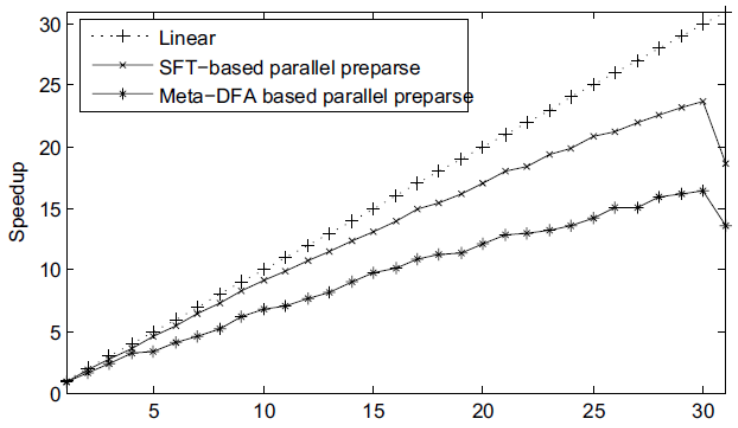
Στην Εικόνα 2-9, παρουσιάζεται το ιδανικό σενάριο με γραμμική επιτάχυνση (Ideal linear speedup), ένα υποθετικό σενάριο επιτάχυνσης με τέλεια ισορροπημένο διαμοιρασμό φορτίου (Speedup if load is perfectly balanced) και αυτό που επιτεύχθηκε στην παραπάνω έρευνα (Actual speedup).



Εικόνα 2-9: Σύγκριση της επιτευχθείσης επιτάχυνσης σε σχέση με ιδανικά σενάρια (Y. Pan, Zhang, et al., 2007).

2.4.6.6 Parallel XML Parsing με Simultaneous Finite Transducer

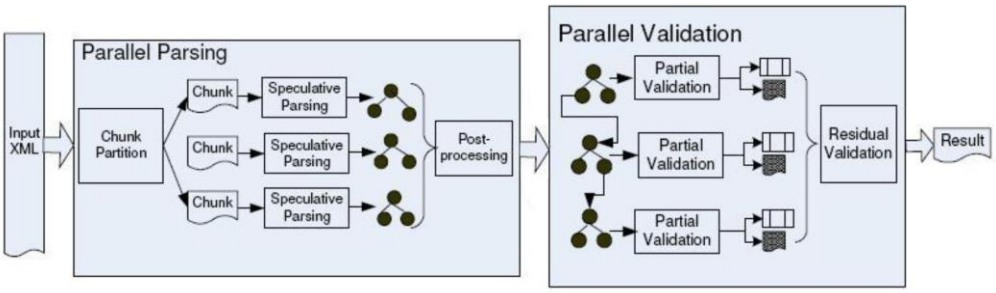
Αργότερα (Pan et al., 2008), παρουσιάστηκε μια παράλληλη προσέγγιση. Εκτελείται παράλληλη προεπεξεργασία, με τη χρήση ενός μετατροπέα Simultaneous Finite Transducer (SFT), που στην πραγματικότητα είναι μια βελτιωμένη εκδοχή του Finite State Transducer (FST) που είναι μια παραλλαγή του FSM. Συμπεραίνεται ότι επιφέρει μεγαλύτερη επιτάχυνση σε σχέση με τη στρατηγική meta-DFA, όπως φαίνεται στην Εικόνα 2-10.



Εικόνα 2-10: Σύγκριση επιτάχυνσης SFT και meta-DFA (Pan et al., 2008).

2.4.6.7 Parallel XML Parsing (2008)

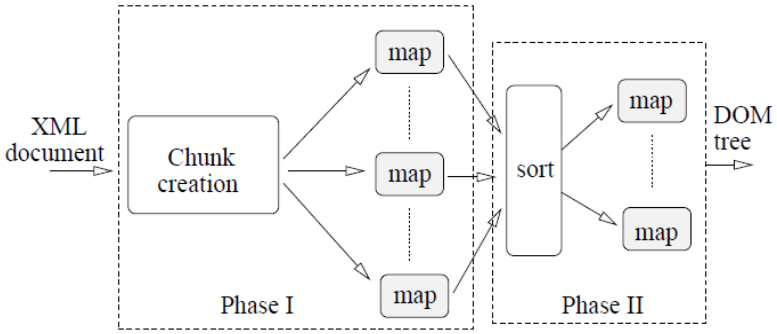
Η επόμενη δημοσίευση των (Wu et al., 2008), έχει στόχο να εκμεταλλευτεί περισσότερο τη χρήση της παράλληλης επεξεργασίας. Χωρίζει την επεξεργασία σε δύο στάδια, το parsing και την επικύρωση του αποτελέσματος (validation). Αρχικά, εκτελείται διαχωρισμός του XML αρχείου σε κομμάτια και ξεκινάει το παράλληλο parsing. Ύστερα, χρησιμοποιώντας πάλι παράλληλη επεξεργασία, βεβαιώνεται ότι έχει δημιουργηθεί σωστά το δέντρο και ότι δεν έχει χαθεί η χρήσιμη πληροφορία από το αρχικό έγγραφο XML. Στην Εικόνα 2-11, φαίνεται η ροή της επεξεργασίας, όπου αρχικά διαχωρίζεται το XML σε κομμάτια, τα οποία διαμοιράζονται σε νήματα και εκτελείται παράλληλο parsing. Στη συνέχεια ελέγχεται η ορθότητα του αποτελέσματος με παράλληλες διεργασίες.



Εικόνα 2-11: Parsing XML αρχείου (Wu et al., 2008).

2.4.6.8 Parallel XML Parsing με ParDOM

Αργότερα, σε μια άλλη έρευνα (Shah et al., 2009), μελετήθηκε η αρχιτεκτονική του PXP και εξετάστηκε η βελτιστοποίηση δημιουργίας δέντρου στη μνήμη, με την υλοποίηση του αλγορίθμου ParDOM (Partial-DOM). Στην πρώτη φάση, ο αλγόριθμος, ξεκινάει διαχωρίζοντας το XML αρχείο σε κομμάτια. Χρησιμοποιώντας παράλληλη επεξεργασία, κάθε νήμα δημιουργεί ένα ολοκληρωμένο τμήμα του δέντρου, το οποίο περιέχει σημαντικές πληροφορίες για τη θέση που πρέπει να έχει μέσα στο τελικό δέντρο DOM. Στην επόμενη φάση, γίνεται ταξινόμηση και ένωση των κομματιών για τη δημιουργία του δέντρου DOM. Στην Εικόνα 2-12, παρουσιάζεται η αρχιτεκτονική του ParDOM.

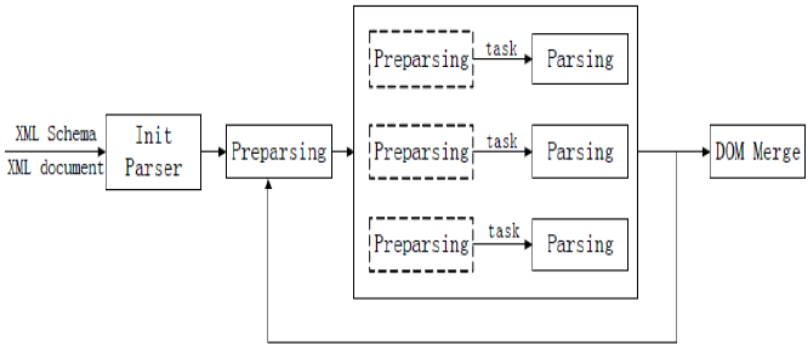


Εικόνα 2-12: Ροή διεργασιών που εκτελούνται στο ParDOM (Shah et al., 2009).

2.4.6.9 Parallel XML Parsing με KEPT

Βασισμένοι στους αλγορίθμους PXP και SFT, υλοποιήθηκε ο αλγόριθμος KEPT (Key Element Parse Tracing) που παραλληλίζει το preparsing και το parsing σε επίπεδο

στοιχείου (element) (Li et al., 2009). Το KEPT είναι ένα υβριδικό μοντέλο. Διαχωρίζει τα δεδομένα σε elements και μετά τα επεξεργάζεται, χρησιμοποιώντας παράλληλη επεξεργασία. Τέλος, συγχωνεύει τα αποτελέσματα σε ένα δέντρο DOM σειριακά. Στην Εικόνα 2-13, παρουσιάζεται η αρχιτεκτονική του αλγορίθμου KEPT.

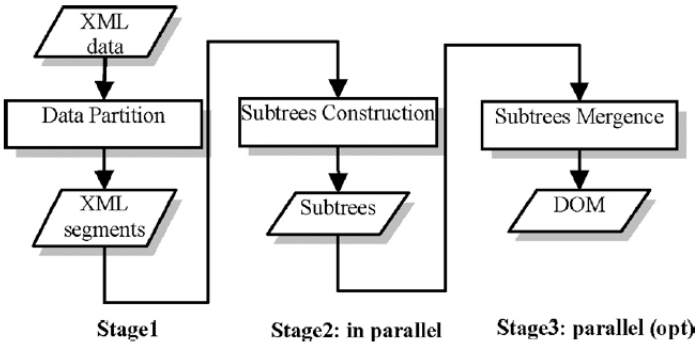


Εικόνα 2-13: Αρχιτεκτονική KEPT

2.4.6.10 Parallel XML Parsing με ParaParse

Σε μια άλλη δημοσίευση (Chen & Liao, 2011), παρουσιάζεται ο αλγόριθμος ParaParse, για το παράλληλο parsing ενός αρχείου XML. Υποστηρίζει το διαχωρισμό τμημάτων του αρχείου και την κατασκευή υποδέντρων, (κλαδιά τμήματα του αρχικού δέντρου) με παράλληλο τρόπο. Στη συνέχεια, τα υποδέντρα συγχωνεύονται για να δημιουργήσουν ολόκληρο το δέντρο DOM. Δεδομένου ότι το στάδιο κατασκευής των υποδέντρων απαιτεί αρκετή υπολογιστική ισχύ, ο παραλληλισμός αυτού του σταδίου μπορεί να βελτιώσει σημαντικά το συνολικό parsing. Τα πειράματα δείχνουν ότι το ParaParse είναι αρκετά αποτελεσματικό σε πολλαπλούς πυρήνες.

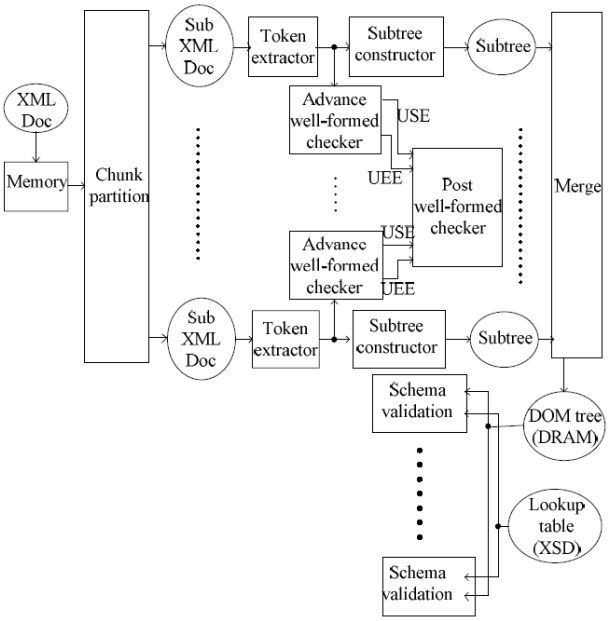
Στην Εικόνα 2-14, φαίνεται η ροή του αλγορίθμου ParaParse.



Εικόνα 2-14: Αρχιτεκτονική ParaParse (Chen & Liao, 2011).

2.4.6.11 Parallel Speculative DOM-based XML Parser

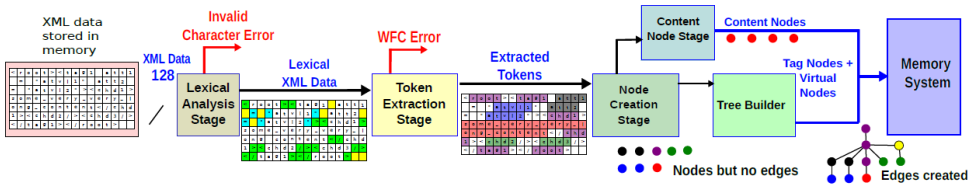
Αργότερα, παρουσιάστηκε μια ακόμη παράλληλη προσέγγιση, η Parallel Speculative DOM-based XML Parser (PSDXP) (Jianliang et al., 2012). Αρχικά το XML έγγραφο διαχωρίζεται σε κομμάτια, τα οποία ανατίθενται σε νήματα που κάνουν έλεγχο της σωστής δομής και δημιουργούν υποδέντρο. Ακολουθεί η δημιουργία του δέντρου DOM από τα υποδέντρα και τέλος γίνεται έλεγχος της ορθότητας. Στην Εικόνα 2-15, φαίνεται η αρχιτεκτονική του αλγορίθμου PSDXP.



Εικόνα 2-15: Αρχιτεκτονική PSDXP (Jianliang et al., 2012).

2.4.6.12 Highly Parallel XML Accelerator

Σε μια άλλη δημοσίευση (Ahmad et al., 2018), παρουσιάστηκε ένας αλγόριθμος παράλληλου parsing, ο Highly Parallel XML Accelerator (HPXA), για τη βελτιστοποίηση parsing XML αρχείων. Για να επιτευχθεί αυτό, χρησιμοποιείται ένας σύνθετος μηχανισμός FSM, προκειμένου να αναγνωριστούν οι χαρακτήρες αρχής και τέλους των ετικετών, χαρακτηριστικών (attributes) και περιεχομένου, τη δημιουργία προσωρινών κόμβων δέντρου και τελικά τη δημιουργία του ολοκληρωμένου δέντρου DOM. Στην Εικόνα 2-16, φαίνεται η αρχιτεκτονική του HPXA.

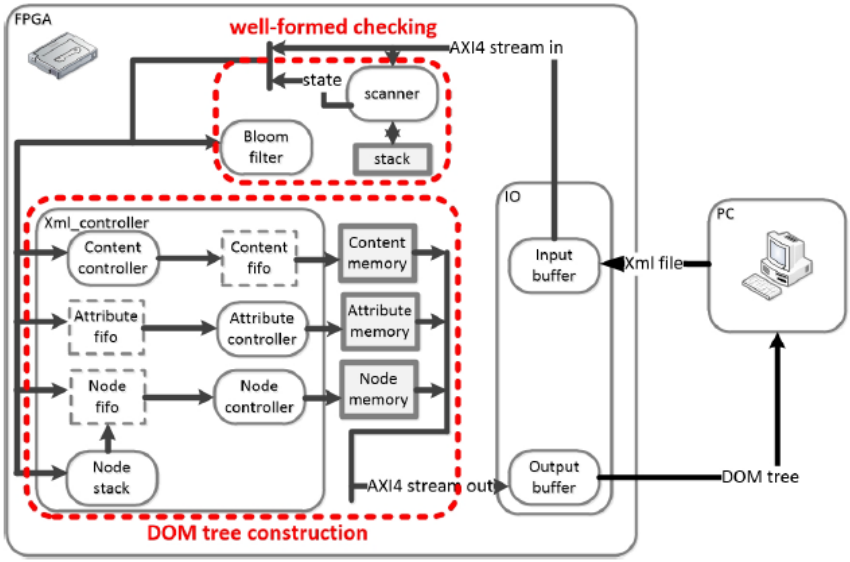


Εικόνα 2-16: Αρχιτεκτονική HPXA (Ahmad et al., 2018).

2.4.6.13 Parallel XML Parsing με FPGA

Αργότερα, σε μια δημοσίευση των (Z. Pan et al., 2019), αναφέρεται ότι το XML έχει αρκετά σημαντικό ρόλο για την μετάδοση δεδομένων σε web servers και σε βάσεις δεδομένων και είναι πολύ σημαντικό το parsing. Το διαφορετικό σε αυτή την έρευνα είναι ότι χρησιμοποιείται μια εξειδικευμένη συσκευή (hardware) Field-Programmable Gate Array (FPGA), για τη βελτιστοποίηση παράλληλου parsing, που δείχνει τις τάσεις της σύγχρονης εποχής για αναζήτηση όλο και πιο γρήγορων μεθόδων επεξεργασίας XML.

Η συσκευή αυτή, ενσωματώνεται στον υπολογιστή και είναι αυτή που εκτελεί προσαρμοσμένο λογισμικό, ανεξάρτητα από την κεντρική μονάδα του υπολογιστή. Παίρνει σαν είσοδο το XML αρχείο, το επεξεργάζεται ελέγχοντας παράλληλα και την ορθότητα, δημιουργεί το δέντρο DOM στην μνήμη της και τελικά το επιστρέφει έτοιμο στον υπολογιστή. Με τη χρήση αυτής της συσκευής, επιτυγχάνεται γρηγορότερο parsing γιατί έχουμε μια υπολογιστική μονάδα να ασχολείται μόνο με αυτό. Στην Εικόνα 2-17, φαίνεται η αρχιτεκτονική του αλγόριθμου που εκτελείται στη συσκευή FPGA.



Εικόνα 2-17: XML Parsing με χρήση συσκευής FPGA (Z. Pan et al., 2019).

2.5 Web Workers

Η παράλληλη επεξεργασία υλοποιείται με νήματα (threads) που ξεκινά η κύρια διαδικασία (process). Στις διαδικτυακές εφαρμογές, που εκτελούνται σε κάποιον περιηγητή (browser), η παράλληλη επεξεργασία υλοποιείται με τη χρήση των web workers. Σύμφωνα με το W3C και το Web Hypertext Application Technology Working Group (WHATWG), web worker είναι ένα script, το οποίο δημιουργείται από μια HTML σελίδα και τρέχει στο παρασκήνιο.

Οι web workers είναι ένα πολύ ισχυρό εργαλείο στην HTML5 και τα τελευταία χρόνια έχει αρχίσει να τραβάει την προσοχή. Παρέχει ένα API, που επιτρέπει την εκτέλεση scripts σε ξεχωριστά νήματα, παράλληλα με τη διαδικτυακή εφαρμογή, χωρίς να απασχολεί τη διεπαφή του χρήστη. Αυτό δίνει τη δυνατότητα σε μεγάλες και χρονοβόρες διεργασίες, να εκτελούνται χωρίς να καθιστούν την ιστοσελίδα μη ανταποκρινόμενη.

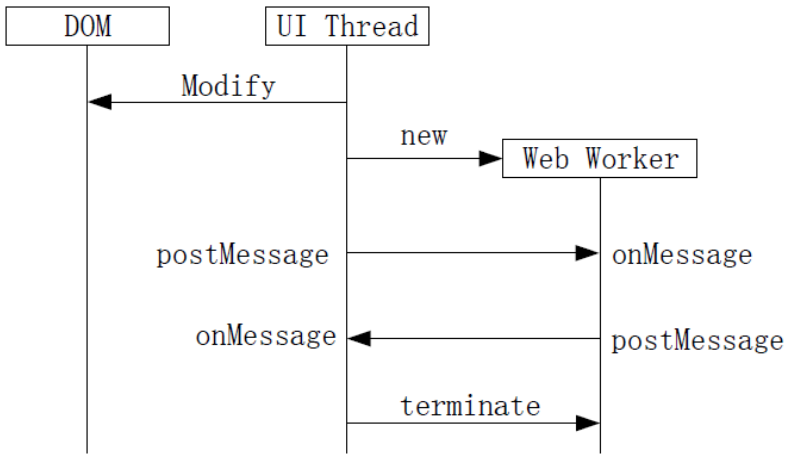
Οι web workers χρησιμοποιούνται για να διαχειριστούν μεγάλες διεργασίες που απαιτούν επεξεργαστική ισχύ, εύρος ζώνης δικτύου και άλλα. Έχουν σκοπό να βελτιστοποιήσουν την περιήγηση του χρήστη, δίνοντας του την αίσθηση ότι η σελίδα εμφανίζεται ομαλά χωρίς καθυστερήσεις.

Οι παράλληλες διεργασίες που εκτελούνται με νήματα, απαιτούν αρκετή επεξεργαστική ισχύ για την προετοιμασία τους και συνεπώς επιπλέον φόρτο για το

σύστημα. Επομένως, χρειάζεται σύνεση στον αριθμό των νημάτων που θα χρησιμοποιηθούν, καθώς το καθένα από αυτά καταλαμβάνει αρκετούς πόρους του συστήματος.

Ιδιαίτερα, σε περιβάλλον περιηγητή, όπου οι διεργασίες εκτελούνται σε εικονική μηχανή, η εκκίνηση ενός web worker είναι αργή και κατά την δημιουργία του δεσμεύονται πόροι του συστήματος (κυρίως μνήμη RAM) που μπορεί και να μη χρησιμοποιηθούν ποτέ. Για το λόγο αυτό, η χρήση τους θα πρέπει να περιορίζεται σε εφαρμογές που έχουν βαριές διεργασίες και χρονοβόρες.

Παρακάτω στην Εικόνα 2-18, φαίνεται ο μηχανισμός του web worker. Όπου UI Thread η διεπαφή (User Interface) που βλέπει ο χρήστης και το κύριο νήμα της HTML διαδικτυακής σελίδας και DOM η δομή μιας HTML σελίδας, σε μια διαδικτυακή εφαρμογή. Ένας web worker, δημιουργείται και τερματίζεται από το κεντρικό νήμα UI, με τη χρήση ενός JavaScript αρχείου. Μπορεί να εκτελέσει υπολογιστικές διεργασίες, αλλά δεν μπορεί να αλλάξει τη δομή της διαδικτυακής εφαρμογής. Δεν έχει άμεση πρόσβαση στο DOM, όμως επικοινωνεί μέσω της κύριας σελίδας, με το πέρασμα μηνυμάτων-Messages (Z. Wang et al., 2018).



Εικόνα 2-18 Αρχιτεκτονική Web Worker (Green, 2012)

Για παράδειγμα, η εφαρμογή «Oliver's JS Raytracer», όπου ασχολείται με την προετοιμασία και τη σύνθεση (render) 3D εικόνας, παρουσιάζεται για να εξεταστεί η χρησιμότητα των web workers. Σε αυτή την εφαρμογή, φαίνεται η διαφορά μεταξύ σειριακής επεξεργασίας, με ένα μοναδικό νήμα και της παράλληλης, με πολλούς web workers, με σκοπό το rendering μιας εικόνας. Η εφαρμογή μπορεί να βρεθεί στον σύνδεσμο «https://nerget.com/rayjs-mt/rayjs.html». Στον Πίνακα 2-3, φαίνεται πως η

χρήση web workers, συμβάλλει στη μείωση χρόνου εκτέλεσης. Οι χρόνοι που παρουσιάζονται, αφορούν μέσους όρους από πολλαπλές εκτελέσεις στον υπολογιστή που αναπτύχθηκε η εργασία.

Πίνακας 2-3: Σύγκριση σειριακής και παράλληλης επεξεργασίας με χρήση web workers

Αριθμός Web Workers	Χρόνος Εκτέλεσης σε seconds (s)
0	1.463s
1	0.435s
2	0.281s
4	0.199s
8	0.222s
12	0.271s
16	0.343s

Είναι εμφανές ότι με τη σειριακή προσέγγιση (0 web workers), ο χρόνος εκτέλεσης είναι αρκετά μεγάλος. Παρατηρείται σημαντική βελτίωση στο χρόνο με τη χρήση μόλις ενός web worker. Προκύπτει επίσης, ότι η χρήση περισσότερων από 4 web workers επιβαρύνουν τελικά τη διαδικασία και χειροτερεύουν την απόδοση. Αυτό δεν είναι αποδεκτό, με δεδομένο ότι λόγος για τον οποίον χρησιμοποιείται η παράλληλη επεξεργασία, είναι η μείωση του χρόνου. Η έρευνα των (Verdu & Pajuelo, 2016) επιβεβαιώνει τα πορίσματα, λέγοντας ότι στις περισσότερες περιπτώσεις, λίγοι web workers παρουσιάζουν παρόμοια ή και ελαφρώς υψηλότερη απόδοση από πολλούς.

Επομένως, ένας προγραμματιστής θα πρέπει να γνωρίζει όχι μόνο αν θα χρησιμοποιήσει παράλληλη επεξεργασία, αλλά και πόσα νήματα απαιτούνται για να αποφέρουν το βέλτιστο αποτέλεσμα, λαμβάνοντας υπόψη την πολυπλοκότητα του προβλήματος και τους διαθέσιμους πόρους.

Όταν χρησιμοποιείται δυναμικό partitioning, δεν είναι άμεσα ελέγξιμος ο αριθμός των νημάτων που θα εκκινηθούν. Είναι ευθύνη του προγραμματιστή όμως, να ελέγξει το μέγιστο πλήθος των νημάτων που θα χρησιμοποιηθούν. Αντίθετα, στο στατικό partitioning, αποφασίζεται εκ των προτέρων το πλήθος των νημάτων, ανάλογα με τη φύση του προβλήματος και τους διαθέσιμους πόρους.

Οι web εφαρμογές, χρησιμοποιούν τοπικό χώρο αποθήκευσης, local storage, προκειμένου να αποθηκεύουν προσωρινά δεδομένα που θα ξαναχρειαστούν, με σκοπό να αποφύγουν τις καθυστερήσεις του δικτύου. Ο χώρος αυτός, μπορεί να χρησιμοποιηθεί από τις παράλληλες διεργασίες της εφαρμογής (web workers).

Οι web workers δεν είναι συμβατοί με όλους τους περιηγητές. Παρακάτω, στην Εικόνα 2-19, εμφανίζεται η πρώτη έκδοση των προγραμμάτων περιήγησης που άρχισε η πλήρης υποστήριξη των web workers. Σαφώς, όλες οι μεταγενέστερες εκδόσεις, τους υποστηρίζουν. Επιπρόσθετα, φαίνεται και αν οι συναρτήσεις των web worker είναι συμβατές με τους περιηγητές.

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
`Worker`	4	12	3.5	10	10.6	4	4	18	4	11	5.1	1.0
`Worker()` constructor	4	12	3.5	10	10.6	4	4	18	4	11	5.1	1.0
`message` event	4	12	3.5	10	10.6	4	4	18	4	11.5	5.1	1.0
`messageerror` event	60	18	57	?	47	?	60	60	57	47	?	8.0
`onmessage`	4	12	3.5	10	10.6	4	4	18	4	11	5.1	1.0
`onmessageerror`	60	18	57	?	47	?	60	60	57	44	?	8.0
`postMessage`	Yes	12	Yes	10 ★	47	Yes	Yes	Yes	Yes	44	Yes	Yes
`terminate`	4	12	3.5	10	10.6	4	4	18	4	11	5.1	1.0

Full support Compatibility unknown

Εικόνα 2-19: Υποστήριξη web workers από περιηγητές πηγή (*Worker*, 2020)

Η χρήση web workers και local storage επεκτείνεται ήδη σε cloud περιβάλλον (Zhang et al., 2015). Μπορούν δηλαδή να χρησιμοποιηθούν εκτός από την παράλληλη επεξεργασία και με κατανεμημένο τρόπο. Δίνεται η δυνατότητα να διαμοιραστούν διεργασίες σε περισσότερους από έναν υπολογιστές, οπουδήποτε στον κόσμο, με διάφανο για τον προγραμματιστή τρόπο. Αυτό συνεπάγεται ατελείωτους πόρους και υπολογιστική ισχύ, παρέχοντας ένα πιο γρήγορο και αποτελεσματικό περιβάλλον επεξεργασίας για τις web εφαρμογές.

3 Μεθοδολογία

Στην παρούσα διπλωματική, εξετάζεται η αξιοποίηση της παράλληλης επεξεργασίας στην πλευρά του χρήστη, για parsing XML αρχείων. Για το σκοπό αυτό δημιουργήθηκε μια διαδικτυακή εφαρμογή που αξιοποιεί τους web workers σε περιβάλλον περιηγητή δικτύου με γλώσσα προγραμματισμού JavaScript.

3.1 Εξοπλισμός

Για την ανάπτυξη της εφαρμογής χρησιμοποιήθηκε το Visual Studio Code σε έναν υπολογιστή με οχταπύρηνο επεξεργαστή με 16 λογικούς πυρήνες και μνήμη 8GB.

Για τα τεστ χρησιμοποιήθηκαν τρεις υπολογιστές με την παρακάτω σύνθεση:

1. 8 πυρήνες – 16 λογικοί επεξεργαστές (νήματα): AMD Ryzen 2700 Eight-Core Processor ~3.2 GHz, 8GB RAM, με λειτουργικό σύστημα Windows 10.
2. 4 πυρήνες – 8 λογικοί επεξεργαστές: Intel i7-4790~3.6GHz, 24GB RAM, με λειτουργικό σύστημα Windows 10.
3. 2 πυρήνες – 4 λογικοί επεξεργαστές: Intel i7-7500U 2.70GHz ~2.9GHz, 8GB RAM, με λειτουργικό σύστημα Windows 10.

Στον εξυπηρετητή έτρεχε Apache Tomcat 10.0 Web Server και η δοκιμή στην πλευρά του χρήστη έγινε σε περιβάλλον περιηγητή Chrome 83.0.4103.61.

Δημιουργήθηκαν XML αρχεία 20 έως 160 Mega Bytes (MB), με επαναλαμβανόμενα δεδομένα, τα οποία ήταν αποθηκευμένα στους σκληρούς δίσκους του κάθε χρήστη.

3.2 Διαδικασία

Πριν ξεκινήσει η ανάπτυξη της εφαρμογής προετοιμάστηκε το περιβάλλον (software/hardware) που χρειάζεται για ανάπτυξη/ δοκιμές διαδικτυακών εφαρμογών με web workers.

3.2.1 Προεργασία

Προέκυψε ότι για να εκτελεστεί η εφαρμογή που εκκινεί web workers, πρέπει να φιλοξενείται υποχρεωτικά σε web server, επειδή θεωρείται ότι ζητά πόρους που φιλοξενούνται σε άλλη πηγή Cross-Origin Resource Sharing (*Cross-Origin Resource Sharing (CORS)*, 2020). Εγκαταστάθηκε το λογισμικό Apache Tomcat 10.0, το οποίο χρησιμοποιήθηκε σαν εξυπηρετητής εφαρμογών ιστοσελίδων.

Μελετήθηκε η συμπεριφορά των εφαρμογών που χρησιμοποιούν web workers και διαπιστώθηκε ότι λόγω της εγγενούς αδυναμίας να χειριστούν δεδομένα του κυρίως προγράμματος και της αναγκαστικής επικοινωνίας μέσω μηνυμάτων, υπάρχει μεγάλη καθυστέρηση, όσο μεγαλώνει ο όγκος των δεδομένων που πρέπει να μεταφερθούν από και προς αυτούς. Συγκεκριμένα για την αποστολή ενός κενού μηνύματος μέχρι τη λήψη απάντησης από τον web worker, απαιτούνται περίπου 50 χιλιοστά του δευτερολέπτου (ms). Ο χρόνος αυτός μεγαλώνει εκθετικά, όσο αυξάνεται ο όγκος των δεδομένων.

Για να αντιμετωπιστεί το πρόβλημα, χρησιμοποιήθηκαν Transferable Objects (*Transferable*, 2020), μια μέθοδος η οποία αντί να στέλνει αντίγραφο των δεδομένων στο web worker χρησιμοποιεί αναφορά τους. Δεν παρατηρήθηκε σημαντική βελτίωση και αποτελεί ένα θέμα το οποίο απαιτεί περεταίρω διερεύνηση.

Με βάση τις έρευνες που έχουν παρουσιαστεί, αποφασίστηκε να χρησιμοποιηθεί ένα υβριδικό μοντέλο επεξεργασίας, όπου το αρχείο επεξεργάζεται σε δύο στάδια. Το πρώτο στάδιο εκτελείται στο κύριο νήμα και σκοπός του είναι ο έλεγχος εγκυρότητας του αρχείου και η δημιουργία ενός σκελετού δέντρου με τις ετικέτες που συναντώνται. Στο δεύτερο στάδιο, ανάλογα με τη δομή του σκελετού που σχηματίστηκε, αποφασίζεται ο τρόπος που θα γίνει ο διαμοιρασμός του αρχείου σε κομμάτια και η απόδοσή τους σε αντίστοιχους web workers για επεξεργασία και κατασκευή του αντίστοιχου τμήματος του δέντρου. Με την ολοκλήρωση των web workers, το κύριο νήμα συνενώνει τα τμήματα που επέστρεψαν αυτοί και κατασκευάζει το τελικό δέντρο DOM.

Κατά την ανάπτυξη της εφαρμογής, έγιναν διάφορες δοκιμές, με XML αρχεία απλής δομής (μορφή array). Διαπιστώθηκε ότι η επεξεργασία στο κύριο νήμα είχε τελειώσει πριν καν ολοκληρωθεί η προεργασία για εκκίνηση παράλληλων νημάτων. Αυτό οφείλεται στα εγγενή προβλήματα της υλοποίησης παράλληλης επεξεργασίας σε περιβάλλον περιηγητή (χρόνοι αρχικοποίησης, μεταφορά δεδομένων από και προς web worker κ.α.), ιδιαίτερα όταν πρόκειται για αρχεία μικρού μεγέθους και απλής δομής. Αφού ο σκοπός της εργασίας ήταν να εξεταστεί η βελτιστοποίηση XML parsing με

χρήση web worker, δημιουργήθηκαν XML αρχεία διαφόρων μεγεθών από 20 έως 160 MB, με ακανόνιστη μορφή δέντρου.

3.2.2 Ανάπτυξη εφαρμογής

Η εφαρμογή αποτελείται από ένα html αρχείο, και 3 javascripts (Παράρτημα Α). Στο html αρχείο είναι η διεπαφή του χρήστη και στα υπόλοιπα είναι οι λειτουργικές ενότητες (functions) που χρησιμοποιούνται. Επειδή, για λόγους ασφαλείας δεν επιτρέπεται σε διαδικτυακές εφαρμογές η πρόσβαση στο σύστημα αρχείων του χρήστη, η εφαρμογή ζητά από το χρήστη να επιλέξει το αρχείο που θα επεξεργαστεί.

Εάν το άνοιγμα του αρχείου είναι επιτυχές, καλείται η συνάρτηση parseSerial(), η οποία κάνει την επεξεργασία του XML, χρησιμοποιώντας την ενσωματωμένη στους περιηγητές λειτουργία XML Parsing (DOMParser) και καταγράφεται ο απαιτούμενος χρόνος σαν «σειριακή» επεξεργασία (από το κύριο νήμα). Στη συνέχεια καλείται η συνάρτηση parseHybrid(), η οποία είναι υπεύθυνη για την ολοκλήρωση της επεξεργασίας με χρήση web workers.

Εάν υπάρχουν διαθέσιμοι πυρήνες, υλοποιείται το XML parsing στα στάδια που αναφέρθηκαν παραπάνω, διαφορετικά γίνεται parsing στο τρέχον νήμα. Ξεκινά και προσπελαύνει το αρχείο ανά χαρακτήρα, με σκοπό να ανιχνεύσει ετικέτες αρχής και τέλους. Εάν εντοπιστεί αναντιστοιχία ετικετών ή λάθη στη σύνταξη, σταματά τη διαδικασία εμφανίζοντας το ανάλογο μήνυμα. Ολοκληρώνοντας, έχει σχηματιστεί ένας σκελετός δέντρου στον οποίο περιέχονται οι ετικέτες αρχής και η θέση τους στο αρχείο.

Στη συνέχεια, ανάλογα με τη δομή του σκελετού που σχηματίστηκε, τους διαθέσιμους πυρήνες και το μέγεθος του αρχείου, αποφασίζεται ο τρόπος κατάτμησης και επεξεργασίας. Εάν το μέγεθος του αρχείου είναι πολύ μικρό ή η δομή του είναι απλή (διάταξη πίνακα), γίνεται η επεξεργασία στο κύριο νήμα.

Εάν από το σκελετό προκύψουν περισσότερες από μια ετικέτες πρώτου επιπέδου, μοιράζονται στους διαθέσιμους πυρήνες. Ο κάθε worker που θα ξεκινήσει, επεξεργάζεται τις ετικέτες που θα του αντιστοιχηθούν, γνωρίζοντας τη θέση τους στο αρχείο από το σκελετό που έχει ήδη σχηματιστεί. Ευθύνη του είναι να σχηματίσει ένα υποδέντρο με τα κλαδιά και τα attributes που αντιστοιχούν στις συγκεκριμένες ετικέτες. Όταν ολοκληρώσει, επιστρέφει το υποδέντρο στο κύριο νήμα.

Δεν χρησιμοποιήθηκε κάποια από τις υπάρχουσες βιβλιοθήκες για XML parsing, αλλά αναπτύχθηκε εξαρχής ένας αλγόριθμος, προσαρμοσμένος κατά το δυνατόν στους περιορισμούς που τίθενται από το περιβάλλον των περιηγητών και τη λειτουργία των web workers. Προκειμένου να βελτιστοποιηθεί η απόδοσή του χρησιμοποιήθηκε τεχνική State Machine. Καθώς ο αλγόριθμος διατρέχει το XML αρχείο, μπορεί να βρεθεί σε ετικέτα αρχής, σε τιμή ή σε ετικέτα τέλους, τα οποία αποτελούν τρεις διακριτές καταστάσεις. Ανάλογα με την κατάσταση που βρίσκεται ανά πάσα στιγμή, εκτελείται ο αντίστοιχος αλγόριθμος.

Το κύριο νήμα όταν λάβει τα αποτελέσματα από όλους τους web workers, συνθέτει το πλήρες δέντρο που αντιστοιχεί στο XML αρχείο, καταγράφει και εμφανίζει το χρόνο που απαιτήθηκε.

3.3 Αποτελέσματα

Μετά την ολοκλήρωση της εφαρμογής, έγιναν δοκιμές σε τρία διαφορετικά συστήματα και με διαφορετικές παραμέτρους, προκειμένου να εξεταστεί η συμπεριφορά των web workers και η αποτελεσματικότητα της χρήσης τους. Θα πρέπει να σημειωθεί ότι εκ των πραγμάτων οι περιηγητές, στο περιβάλλον των οποίων εκτελούνται οι web workers, δεν μπορούν να εκμεταλλευτούν όλους τους διαθέσιμους πόρους και τις δυνατότητες του υποκείμενου συστήματος. Κατά τις δοκιμές παρατηρήθηκε απόκλιση των αποτελεσμάτων, η οποία οφείλονταν στην κατάσταση του υποκείμενου συστήματος (λειτουργικό σύστημα, εκτέλεση άλλων προγραμμάτων κ.τ.λ.). Για το σκοπό αυτό έγιναν επαναλαμβανόμενες δοκιμές ανά περίπτωση και υπολογίστηκε ο μέσος όρος. Καταγράφηκαν τα δεδομένα που προέκυψαν και παρουσιάζονται σε πίνακες και γραφήματα.

Σχεδιάστηκαν δύο σενάρια δοκιμών, προκειμένου να εξεταστεί κατά πόσο επηρεάζει το προσδοκώμενο κέρδος ο όγκος της επεξεργαζόμενης πληροφορίας και ο αριθμός των παράλληλων διεργασιών. Από τις πρώιμες δοκιμές διαπιστώθηκε ότι η απόδοση αρχίζει να γίνεται εμφανής όταν το μέγεθος των αρχείων ξεπερνά τα 20 MB. Επειδή τα συστήματα που χρησιμοποιήθηκαν διέθεταν σχετικά μικρή μνήμη RAM, διαπιστώθηκε ότι η χρήση υπερβολικά μεγάλων αρχείων προκαλούσε χρήση εικονικής μνήμης (εναλλαγή των περιεχομένων της μνήμης σε αρχείο του σκληρού δίσκου) και

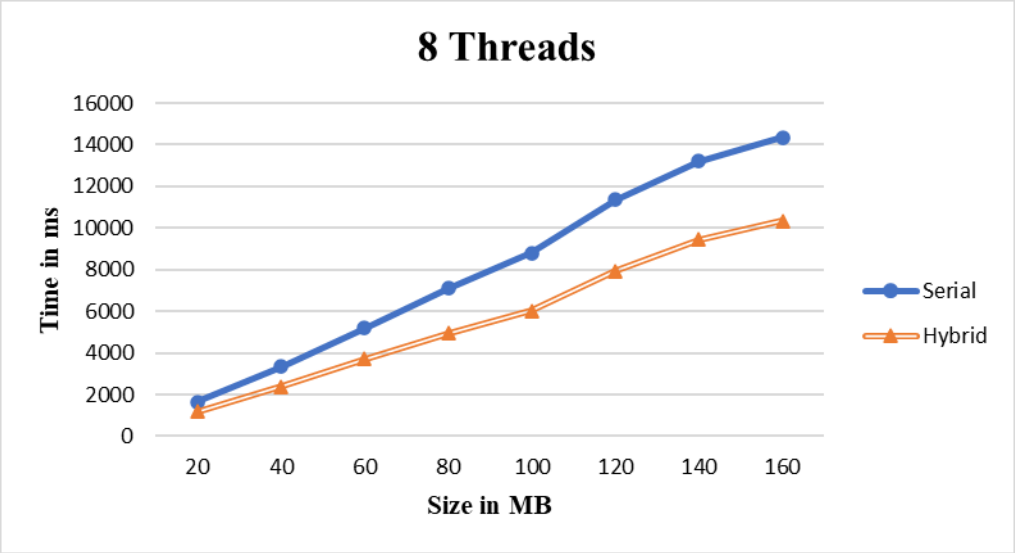
συνεπώς καθυστέρηση επεξεργασίας, αποφασίστηκε να χρησιμοποιηθεί αρχείο 120 MB για τις δοκιμές με μεταβαλλόμενο αριθμό νημάτων.

Στο πρώτο σενάριο, διατηρήθηκε σταθερός ο αριθμός των εκκινούμενων νημάτων, ο οποίος είναι ίσος με τον αριθμό φυσικών πυρήνων και μεταβάλλονταν το μέγεθος (Size) του προς επεξεργασία αρχείου, από 20 έως 160 MB. Στο δεύτερο σενάριο, χρησιμοποιήθηκε αρχείο σταθερού μεγέθους (Size) 120 MB και μεταβάλλονταν ο αριθμός των εκκινούμενων νημάτων, το πλήθος των οποίων δεν υπερέβαινε τον αριθμό των διαθέσιμων λογικών επεξεργαστών αφαιρούμενου ενός, για το κεντρικό νήμα (αριθμός λογικών επεξεργαστών - 1).

Παρουσιάζονται τα αποτελέσματα των δοκιμών στα τρία προαναφερθέντα συστήματα. Για κάθε δοκιμή, παρατίθεται ένα γράφημα και ένας πίνακας στον οποίον φαίνονται οι χρόνοι κάθε δοκιμής σε χιλιοστά του δευτερολέπτου (Time in ms), για κάθε δοκιμή (Serial, Hybrid) και η ποσοστιαία διαφορά τους (Percentage).

3.3.1 Σύστημα δοκιμών 1

Σε αυτό το σύστημα, για την πρώτη δοκιμή, εκκινήθηκαν 8 νήματα όσοι είναι και οι διαθέσιμοι φυσικοί πυρήνες του και αρχεία με μέγεθος από 20 μέχρι 160 MB. Τα αποτελέσματα φαίνονται στην Εικόνα 3-1 και στον Πίνακα 3-1.



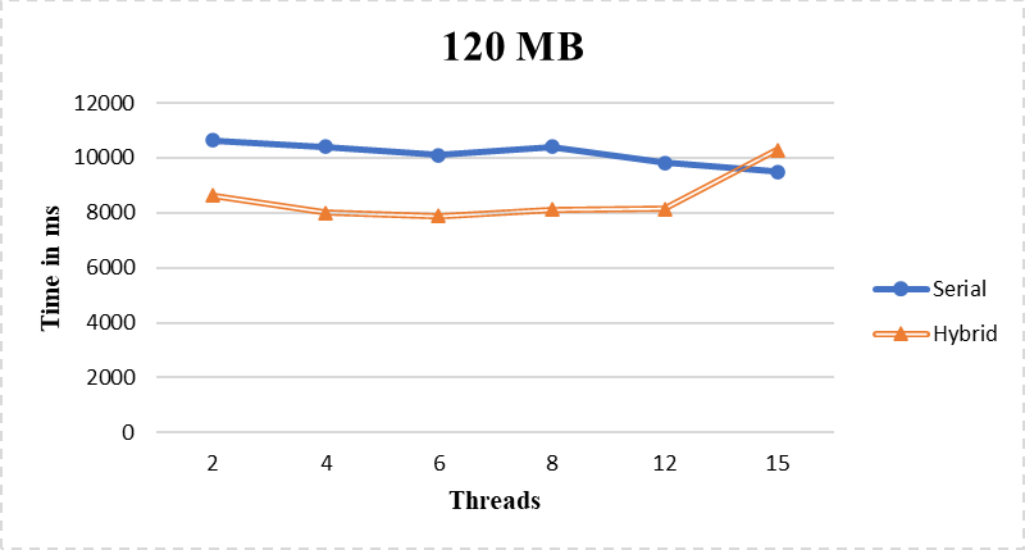
Εικόνα 3-1: 8 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

Πίνακας 3-1: 8 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

Size MB	Threads	Serial	Hybrid	Percentage
20	8	1642,33	1182,83	27,98
40	8	3344,67	2400,33	28,23
60	8	5207,33	3707,50	28,80
80	8	7116,17	4951,83	30,41
100	8	8791,50	6019,17	31,53
120	8	11347,50	7943,00	30,00
140	8	13208,17	9449,33	28,46
160	8	14358,17	10357,67	27,86

Από το γράφημα προκύπτει, ότι όσο μεγαλώνει το μέγεθος του αρχείου, τόσο μεγαλώνει και το κέρδος από τη χρήση web workers. Παρατηρείται ότι μετά τα 100MB ο χρόνος επεξεργασίας και για τις δύο δοκιμές γίνεται μεγαλύτερος, λόγω της ανάγκης χρήσης εικονικής μνήμης. Παρόλα αυτά όμως, η διαφορά μεταξύ των δύο δοκιμών είναι εμφανής.

Στη δεύτερη δοκιμή, χρησιμοποιήθηκε το ίδιο αρχείο XML μεγέθους 120MB και εκκινήθηκαν διαδοχικά από 2 έως 15 νήματα (οι διαθέσιμοι λογικοί επεξεργαστές τους υπολογιστή, μείον έναν για το κύριο νήμα). Τα αποτελέσματα φαίνονται στην Εικόνα 3-2 και στον Πίνακα 3-2.



Εικόνα 3-2: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

Πίνακας 3-2: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

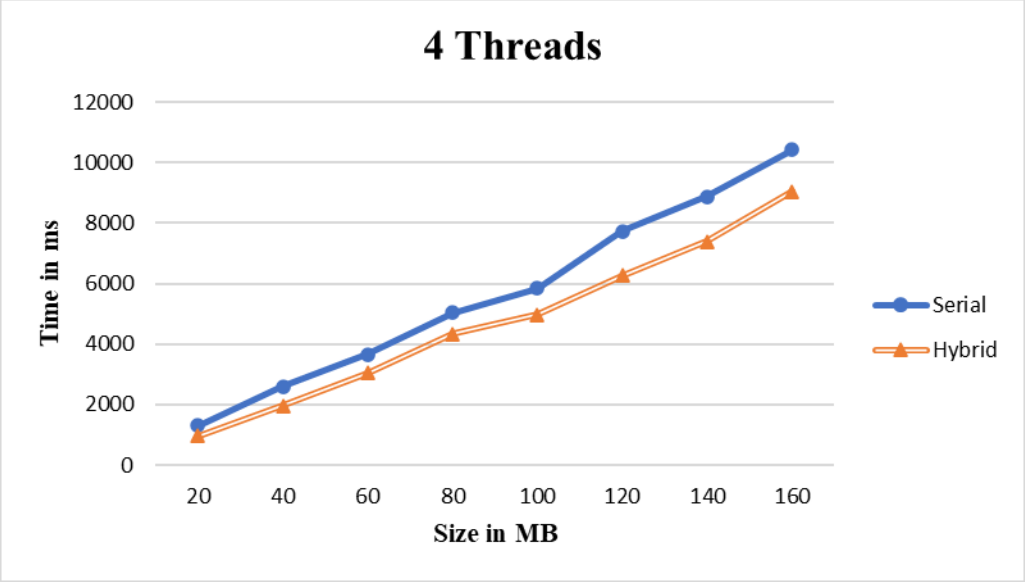
Size MB	Threads	Serial	Hybrid	Percentage
120	2	10635,10	8624,80	18,90
120	4	10405,90	7997,20	23,15

120	6	10101,20	7895,20	21,84
120	8	10407,20	8121,70	21,96
120	12	9816,30	8134,10	17,14
120	15	9498,70	10292,40	-8,36

Από το γράφημα προκύπτει ότι η διαφορά του χρόνου επεξεργασίας είναι εμφανής από τη χρήση μόλις δύο νημάτων. Παρατηρείται ότι μέχρι τη χρήση του 80% περίπου των διαθέσιμων λογικών επεξεργαστών, το κέρδος από την παράλληλη επεξεργασία παραμένει σταθερό. Η χρήση όμως περισσότερων νημάτων, καταλήγει μη αποδοτική. Φαίνεται, ότι πρέπει να τηρείται ένα όριο στη χρήση των διαθέσιμων λογικών επεξεργαστών, πέραν του οποίου δεν υπάρχει όφελος, τουλάχιστον με τη χρήση web workers.

3.3.2 Σύστημα δοκιμών 2

Σε αυτό το σύστημα, για την πρώτη δοκιμή, εκκινήθηκαν 4 νήματα όσοι είναι και οι διαθέσιμοι φυσικοί πυρήνες του και αρχεία με μέγεθος από 20 μέχρι 160 MB. Τα αποτελέσματα φαίνονται στην Εικόνα 3-3 και στον Πίνακα 3-3.



Εικόνα 3-3: 4 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

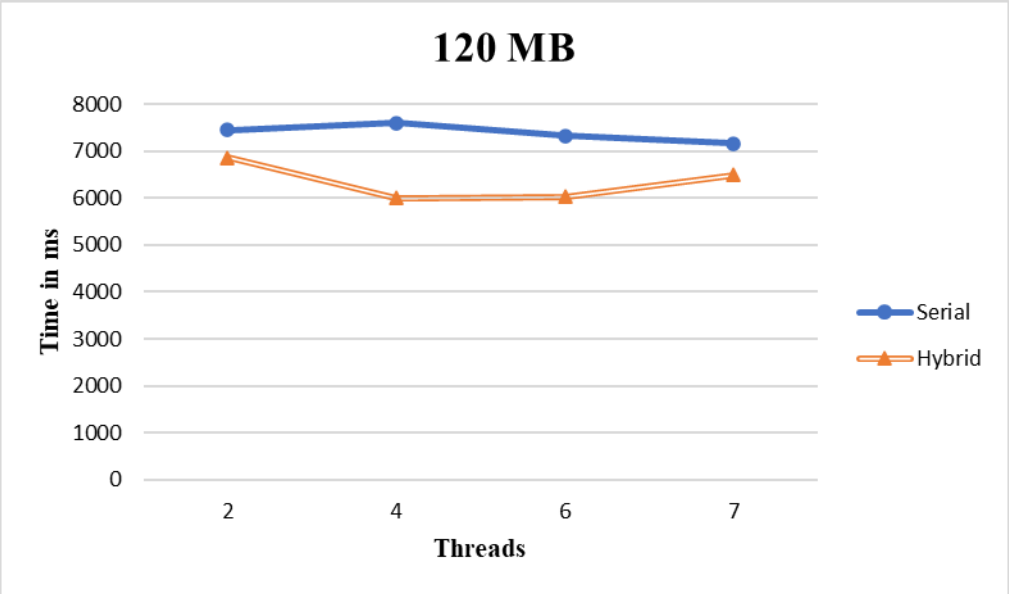
Πίνακας 3-3: 4 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

Size MB	Threads	Serial	Hybrid	Percentage
20	4	1307,17	967,67	25,97
40	4	2611,33	1975,00	24,37

60	4	3662,33	3042,33	16,93
80	4	5042,17	4346,67	13,79
100	4	5852,83	4985,00	14,83
120	4	7737,33	6263,50	19,05
140	4	8867,00	7392,33	16,63
160	4	10420,83	9049,67	13,16

Από το γράφημα προκύπτει, ότι όσο μεγαλώνει το μέγεθος του αρχείου, τόσο μεγαλώνει και το κέρδος από τη χρήση web workers. Παρότι το σύστημα των δοκιμών είναι μικρότερης δυναμικότητας, το κέρδος από την παράλληλη επεξεργασία είναι εμφανές.

Στη δεύτερη δοκιμή, χρησιμοποιήθηκε το ίδιο αρχείο XML μεγέθους 120MB και εκκινήθηκαν διαδοχικά από 2 έως 7 νήματα (οι διαθέσιμοι λογικοί επεξεργαστές του υπολογιστή, μείον έναν για το κύριο νήμα). Τα αποτελέσματα φαίνονται στην Εικόνα 3-4 και στον Πίνακα 3-4.



Εικόνα 3-4: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

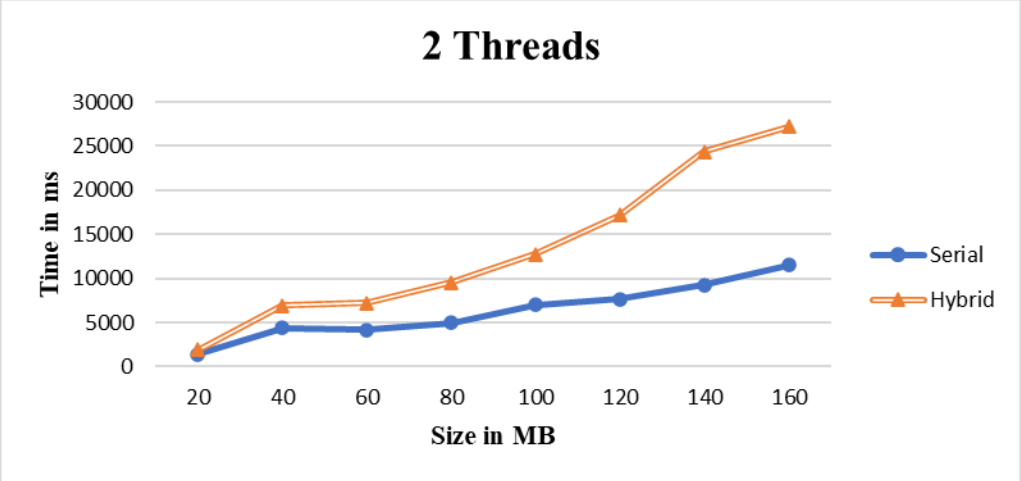
Πίνακας 3-4: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

Size MB	Threads	Serial	Hybrid	Percentage
120	2	7447,50	6863,00	7,85
120	4	7590,83	6001,17	20,94
120	6	7329,67	6025,83	17,79
120	7	7154,00	6488,50	9,30

Από το γράφημα προκύπτει ότι η διαφορά του χρόνου επεξεργασίας είναι εμφανής από τη χρήση μόλις δύο νημάτων. Και σε αυτή τη δοκιμή, παρατηρείται ότι μετά από 80% περίπου των διαθέσιμων λογικών επεξεργαστών, το κέρδος από την παράλληλη επεξεργασία μειώνεται.

3.3.3 Σύστημα δοκιμών 3

Σε αυτό το σύστημα, για την πρώτη δοκιμή, εκκινήθηκαν 2 νήματα όσοι είναι και οι διαθέσιμοι φυσικοί πυρήνες του και αρχεία με μέγεθος από 20 μέχρι 160 MB. Τα αποτελέσματα φαίνονται στην Εικόνα 3-5 και στον Πίνακα 3-5.



Εικόνα 3-5: 2 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

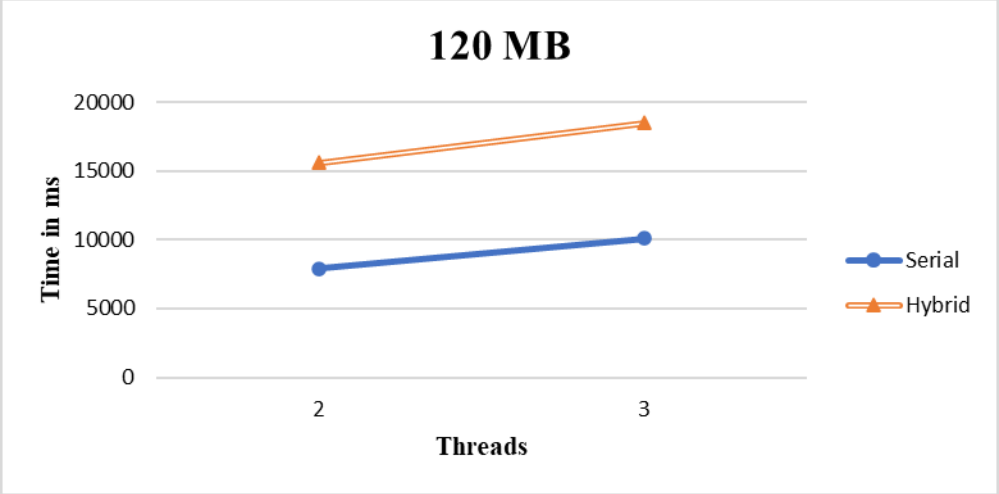
Πίνακας 3-5: 2 σταθερά νήματα - μεταβλητό μέγεθος αρχείου.

Size MB	Threads	Serial	Hybrid	Percentage
20	2	1431,83	1914,17	-33,69
40	2	4411,50	6959,50	-57,76
60	2	4192,33	7193,50	-71,59
80	2	4944,67	9570,00	-93,54
100	2	6974,17	12720,83	-82,40
120	2	7683,67	17150,67	-123,21
140	2	9238,50	24371,00	-163,80
160	2	11516,67	27196,33	-136,15

Επειδή το συγκεκριμένο σύστημα είναι μικρής δυναμικότητας και με μόλις δύο διαθέσιμους φυσικούς πυρήνες, η παράλληλη επεξεργασία με χρήση web workers, δε φαίνεται να αποδίδει. Μάλιστα όσο μεγαλώνει το μέγεθος του αρχείου, χειροτερεύει η

απόδοση της παράλληλης επεξεργασίας. Προφανώς αυτό οφείλεται στο ότι δεν μπορεί να διαχειριστεί ικανοποιητικά την εκκίνηση – επικοινωνία με web workers.

Στη δεύτερη δοκιμή, χρησιμοποιήθηκε το ίδιο αρχείο XML μεγέθους 120MB και εκκινήθηκαν διαδοχικά από 2 έως 3 νήματα (οι διαθέσιμοι λογικοί επεξεργαστές του υπολογιστή, μείον έναν για το κύριο νήμα). Τα αποτελέσματα φαίνονται στην Εικόνα 3-6 και στον Πίνακα 3-6.



Εικόνα 3-6: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

Πίνακας 3-6: Μεταβλητά νήματα - σταθερό μέγεθος αρχείου 120MB.

Size MB	Threads	Serial	Hybrid	Percentage
120	2	7926,10	15601,80	-96,84
120	3	10080,40	18484,70	-83,37

Και εδώ επειδή πρόκειται για σύστημα χαμηλής δυναμικότητας, φαίνεται ότι όχι μόνο δεν προκύπτει όφελος από τη χρήση παράλληλης επεξεργασίας, αλλά επιβάρυνση.

4 Επίλογος

Σε αυτή την εργασία, έγινε έρευνα σχετικά με την ανταλλαγή και την επεξεργασία δεδομένων μεταξύ ετερογενών συστημάτων και τα κυριότερα πρότυπα για αυτό. Εστιάστηκε στη χρήση του διαδεδομένου προτύπου XML και παρουσιάστηκαν διάφορες έρευνες για την κατά το δυνατόν καλύτερη επεξεργασία τους. Μελετήθηκε η δυνατότητα παράλληλης επεξεργασίας στην πλευρά του χρήστη και αναπτύχθηκε εφαρμογή για επεξεργασία XML αρχείων σε περιβάλλον περιηγητή ιστοσελίδων, με χρήση web workers. Έγιναν δοκιμές σε τρία συστήματα με διαφορετική σύνθεση εξοπλισμού (hardware) και παρουσιάστηκαν τα αποτελέσματα.

4.1 Σύνοψη και συμπεράσματα

Στην παρούσα μελέτη κατά την εφαρμογή του πρώτου συστήματος δοκιμών, με την εκκίνηση 8 νημάτων (όσοι ήταν και οι διαθέσιμοι φυσικοί πυρήνες) και χρήση αρχείων με μέγεθος από 20 μέχρι 160 MB, προέκυψε το συμπέρασμα ότι όσο μεγαλώνει το μέγεθος του αρχείου, τόσο μεγαλώνει και το όφελος από τη χρήση web workers. Με την διαδοχική εκκίνηση από 2 έως 15 νημάτων και χρήση του ίδιου αρχείου XML μεγέθους 120 MB, η διαφορά του χρόνου επεξεργασίας ήταν εμφανής από τη χρήση μόλις δύο νημάτων. Ωστόσο, παρατηρήθηκε ότι ενώ μέχρι τη χρήση του 80% περίπου των διαθέσιμων λογικών επεξεργαστών το κέρδος από την παράλληλη επεξεργασία παρέμεινε σταθερό, η χρήση περισσότερων νημάτων, δεν ήταν αποδοτική. Φαίνεται, ότι πρέπει να τηρείται ένα όριο στη χρήση των διαθέσιμων λογικών επεξεργαστών, πέραν του οποίου δεν υπάρχει όφελος, τουλάχιστον με τη χρήση web workers. Ανάλογα ήταν τα ευρήματα από την εφαρμογή της ίδιας μεθοδολογίας σε σύστημα μικρότερης δυναμικότητας με 4 φυσικούς πυρήνες. Τέλος, με την εφαρμογή των δοκιμών σε σύστημα σημαντικά μικρότερης δυναμικότητας μόλις 2 φυσικών πυρήνων δεν προέκυψε όφελος από τη χρήση παράλληλης επεξεργασίας. Αυτό οφείλεται στο ότι και οι δύο πυρήνες χρησιμοποιούνται από το λειτουργικό σύστημα προκειμένου να εξυπηρετηθούν οι ανάγκες του περιηγητή και της απαιτούμενης εικονικής μηχανής για τη λειτουργία των web workers.

Θεωρείται δεδομένο, ότι η χρήση παράλληλης επεξεργασίας για την επίλυση πολύπλοκων προβλημάτων επιφέρει θεαματικά αποτελέσματα. Όταν όμως αυτό πρέπει

να γίνει ανεξάρτητο συστημάτων (hardware - software), παρεμβάλλονται πολλές παράμετροι οι οποίες επηρεάζουν αρνητικά την αποδοτικότητα.

Στην έρευνα αυτή, μελετήθηκε η υλοποίηση παράλληλης επεξεργασίας XML αρχείων με χρήση web workers, σε περιβάλλον περιηγητή ιστοσελίδων. Οι web workers εκτελούνται σε εικονική μηχανή που ενεργοποιείται από τον περιηγητή ιστοσελίδων και φιλοξενείται στο κύριο λειτουργικό σύστημα. Αυτό σημαίνει ότι απαιτούνται επιπλέον χρόνοι για την αρχικοποίηση και την εκκίνηση κάθε νήματος (web worker). Ένας άλλος περιορισμός που υπάρχει στο σύστημα περιηγητή – web workers, είναι ότι δεν υπάρχει δυνατότητα πρόσβασης σε κοινά δεδομένα μεταξύ κυρίου προγράμματος και νημάτων και η επικοινωνία γίνεται μόνο μέσω μηνυμάτων. Όλα αυτά επιβαρύνουν σημαντικά το χρόνο εκτέλεσης και περιορίζουν τα οφέλη της παράλληλης επεξεργασίας.

Παρόλα αυτά, αν χρησιμοποιηθούν σωστά, λαμβάνοντας υπόψη τους παραπάνω περιορισμούς, τις δυνατότητες των υποκείμενων συστημάτων, τη σωστή κατάτμηση του προβλήματος και διαμοιρασμό του σε αντίστοιχα νήματα, σαφώς ενδείκνυται η χρήση τους. Ακόμη και αν το κέρδος σε χρόνο εκτέλεσης δεν είναι μεγάλο, υπάρχει το σαφές πλεονέκτημα της απελευθέρωσης του κυρίου προγράμματος, που είναι υπεύθυνο για το περιβάλλον που βλέπει ο χρήστης, με αποτέλεσμα να μην καταλαβαίνει τις καθυστερήσεις για πολύπλοκες και χρονοβόρες διαδικασίες και να έχει καλύτερη διάδραση με τις εφαρμογές.

4.2 Όρια και περιορισμοί της έρευνας

Παρά τον προσεκτικό σχεδιασμό της παρούσας μελέτης, για την εγκυρότερη ερμηνεία και αξιοποίηση των αποτελεσμάτων της, κρίνεται σημαντική η αναφορά και η αιτιολόγηση κάποιων ορίων και περιορισμών, όσον αφορά στην εφαρμογή της. Συγκεκριμένα, δεν εξετάστηκαν αρχεία μεγαλύτερου όγκου λόγω του περιορισμού του διαθέσιμου συστήματος και πιο συγκεκριμένα της διαθέσιμης μνήμης RAM. Ωστόσο, μπορεί να υποτεθεί ότι σε μεγαλύτερα αρχεία θα ήταν εμφανέστερη η ωφέλεια από τη χρήση web workers. Επίσης, στην παρούσα έρευνα, ήταν αντικειμενικά και πρακτικά δύσκολο να οργανωθεί και να εκτελεστούν δοκιμές σε κατανεμημένα συστήματα.

Οι web workers είναι σχετικά νέα τεχνολογία και δεν έχουν γίνει αρκετές εμπεριστατωμένες μελέτες που να παρέχουν στοιχεία για την καλύτερη αξιοποίησή τους. Επιπρόσθετα, τα διαθέσιμα συστήματα που χρησιμοποιήθηκαν για την εκπόνηση της

μελέτης και τις δοκιμές δεν ήταν υψηλών προδιαγραφών (οικιακοί υπολογιστές), καθώς έπρεπε να προσομοιώνουν και περιβάλλον χρήστη και διακομιστή, γιατί οι web workers δεν επιτρέπεται να εκτελεστούν κατευθείαν από το τοπικό σύστημα αρχείων. Τέλος, στις δοκιμές χρησιμοποιήθηκαν αρχεία, τα οποία περιείχαν κατασκευασμένα δεδομένα και όχι πραγματικά. Αυτό ενδεχομένως να είχε διαφορετική επίδραση στην επίδοση των web workers.

4.3 Μελλοντικές Επεκτάσεις

Με βάση τα αποτελέσματα της παρούσας μελέτης και λαμβάνοντας υπόψη τα όρια και τους περιορισμούς της, όπως παρουσιάστηκαν παραπάνω, προτείνεται μελλοντικά να εξεταστούν: (α) ο παραλληλισμός με web workers και από το πρώτο στάδιο (preparse, validation), ίσως και κατά τη συρραφή (merge) των αποτελεσμάτων μετά την ολοκλήρωση του parsing, (β) η παράλληλη επεξεργασία σε πραγματικό χρόνο, κατά τη διάρκεια λήψης αρχείου (streaming), με τη χρήση web workers, (γ) η παράλληλη επεξεργασία με κατανεμημένο τρόπο, με χρήση web worker σε περιβάλλον σε απομακρυσμένα συστήματα (cloud) και (δ) εκμετάλλευση των δυνατοτήτων υποκείμενων καρτών γραφικών για παράλληλη επεξεργασία (webGPU).

Βιβλιογραφία

Ahmad, I., Patil, S., & Sarangi, S. R. (2018). HPXA: A highly parallel XML parser. *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 249–252. https://doi.org/10.23919/DATE.2018.8342012

Arvindpdmn, A. (2019, February 27). *Data Serialization*. Devopedia. https://devopedia.org/data-serialization

Balasundaram, V., Fox, G., Kennedy, K., & Kremer, U. (1991). A static performance estimator to guide data partitioning decisions. *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 213–223. https://doi.org/10.1145/109625.109647

Ben-Kiki, O., & Evans, C. (2007). YAML Ain't Markup Language (YAML™) Version 1.2. *2007*, 84.

Campbell-Kelly, M. (1987). Data Communications at the National Physical Laboratory (1965-1975). *Annals of the History of Computing*, *9*(3/4), 221–247. https://doi.org/10.1109/MAHC.1987.10023

Chen, R., & Liao, H. (2011). ParaParse: A parallel method for XML parsing. *2011 IEEE 3rd International Conference on Communication Software and Networks*, 81–85. https://doi.org/10.1109/ICCSN.2011.6014223

Cross-Origin Resource Sharing (CORS). (2020). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Data Deserialization. (2016, July 29). CIS. https://www.cisecurity.org/blog/data-deserialization/

Döhmen, T. R. (2016). *Multi-Hypothesis Parsing of Tabular Data in Comma-Separated Values (CSV) Files*. 113.

DOM Standard. (2020). DOM. https://dom.spec.whatwg.org/

Green, I. (2012). *Web Workers*. O'Reilly Media. http://shop.oreilly.com/product/0636920024446.do

Gupta, V., Udupi, P., & Poursohi, A. (2010). Early lessons from building Sensor.Network: an open data exchange for the web of things. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 738–744. https://doi.org/10.1109/PERCOMW.2010.5470530

Jianliang, M., Zhang, S., Hu, T., Wu, M., & Chen, T. (2012). Parallel Speculative Dom-based XML Parser. *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, 33–40. https://doi.org/10.1109/HPCC.2012.15

Li, X., Wang, H., Liu, T., & Li, W. (2009). Key Elements Tracing Method for Parallel XML Parsing in Multi-Core System. *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 439–444. https://doi.org/10.1109/PDCAT.2009.64

Lu, W., Chiu, K., & Pan, Y. (2006). A Parallel Approach to XML Parsing. *2006 7th IEEE/ACM International Conference on Grid Computing*, 223–230. https://doi.org/10.1109/ICGRID.2006.311019

Lu, W., & Gannon, D. (2007). Parallel XML processing by work stealing. *Proceedings of the 2007 Workshop on Service-Oriented Computing Performance: Aspects, Issues, and Approaches - SOCP '07*, 31–38. https://doi.org/10.1145/1272457.1272462

Lunteren, J. V., Engbersen, T., Bostian, J., Carey, B., & Larsson, C. (2004). XML Accelerator Engine. *In The First International Workshop on High Performance XML Processing*.

Mack, C. A. (2011). Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing*, *24*(2), 202–207. https://doi.org/10.1109/TSM.2010.2096437

Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). *Comparison of JSON and XML Data Interchange Formats: A Case Study*. 6.

Oliveira, B., Santos, V., & Belo, O. (2013). Processing XML with Java – A Performance Benchmark. *International Journal of New Computer Architectures and Their Applications*, 15.

Pan, Y., Lu, W., Zhang, Y., & Chiu, K. (2007). A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 351–362. https://doi.org/10.1109/CCGRID.2007.14

Pan, Y., Zhang, Y., Chiu, K., & Lu, W. (2007). Parallel XML Parsing Using Meta-DFAs. *Third IEEE International Conference on E-Science and Grid Computing (e-Science 2007)*, 237–244. https://doi.org/10.1109/E-SCIENCE.2007.55

Pan, Z., Jiang, X., Wu, J., & Li, X. (2019). Hybrid XML Parser Based on Software and Hardware Co-design. *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 325–325. https://doi.org/10.1109/FCCM.2019.00066

Rao, V. N., & Kumar, V. (1987). Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, *16*(6), 479–499. https://doi.org/10.1007/BF01389000

Severance, C. (2012). Discovering JavaScript Object Notation. *Computer*, *45*(4), 6–8. https://doi.org/10.1109/MC.2012.132

Shafranovich, Y. (2005). *Common Format and MIME Type for Comma-Separated Values (CSV) Files* (No. RFC4180; p. RFC4180). RFC Editor. https://doi.org/10.17487/rfc4180

Shah, B., Rao, P. R., Moon, B., & Rajagopalan, M. (2009). A Data Parallel Algorithm for XML DOM Parsing. In Z. Bellahsène, E. Hunt, M. Rys, & R. Unland (Eds.), *Database and XML Technologies* (Vol. 5679, pp. 75–90). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-03555-5_7

Transferable. (2020). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Transferable

Verdu, J., & Pajuelo, A. (2016). Performance Scalability Analysis of JavaScript Applications with Web Workers. *IEEE Computer Architecture Letters*, *15*(2), 105–108. https://doi.org/10.1109/LCA.2015.2494585

W3C. (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. https://www.w3.org/TR/REC-xml/

Wang, J. (2019). *Formal Methods in Computer Science*. Chapman and Hall/CRC. https://doi.org/10.1201/9780429184185

Wang, Z., Deng, H., Hu, L., & Zhu, X. (2018). HTML5 Web Worker Transparent Offloading Method for Web Applications. *2018 IEEE 18th International Conference on Communication Technology (ICCT)*, 1319–1323. https://doi.org/10.1109/ICCT.2018.8600046

Worker. (2020). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Worker

Wu, Y., Zhang, Q., Yu, Z., & Li, J. (2008). *A Hybrid Parallel Processing for XML Parsing and Schema Validation*. Balisage: The Markup Conference 2008, Montréal, Canada. https://doi.org/10.4242/BalisageVol1.Wu01

Yinfei Pan, Ying Zhang, & Chiu, K. (2008). Simultaneous transducers for data-parallel XML parsing. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–12. https://doi.org/10.1109/IPDPS.2008.4536240

Zhang, X., Jose, S., Gibbs, S. J., Jose, S., Kunjithapatham, A., & Jeong, S. (2015). *(54) CLOUD-BASED WEB WORKERS AND*. 20.

Μακράτζη, Α. (2016). Ενεργητική ακρόαση και μάθηση για μια ανθρωποκεντρική Παιδαγωγική. *Ανθρωποκεντρικό*, *1*(1), 34–49.

Παράρτημα Α - Κώδικας εφαρμογής

openXML.html

```html
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <input type="file" id="file-input" />
    <h1>Log Window:</h1>
    <div>
        <pre>
            <div id='log'></div>
        </pre>
    </div>
    <pre id="file-content"></pre>
    <script type="module" src="parsingXML.js"></script>
    <script type="module">
        import { initializeContent, parseHybrid, parseSerial, parseAll, showLog } from './parsingXML.js';
        document.getElementById('file-input')
            .addEventListener('change', readXMLFile, false);
        function readXMLFile(e) {
            var file = e.target.files[0];
            if (!file) {
                return;
            }
            var reader = new FileReader();
            reader.onload = function (e) {
                var content = e.target.result;
                initializeContent(content);
                parseSerial();
                if (navigator.hardwareConcurrency > 2) {
                    parseHybrid();
                }
                else
                    showLog("No available threads for parallel processing");
            };
            reader.readAsText(file);
        }
        function displayContents(content) {
            var element = document.getElementById('file-content');
            element.textContent = content;
        }
    </script>
</body>
</html>
```

parseXML.js

```js
"use strict";

import { wwTree, wwNode } from './classes.js';
import { parseChunk } from './parseChunk.js';


var fullTree = new wwTree();
var threadFinished = 1;
var timeStart;
var xmlDoc;
var content;
var skeleton = new wwTree();
var lvlCount = 0;
const minXMLSize = 1_000;
const threadsIdealFactor = 0.8;

export function initializeContent(cnt) {
    content = cnt;
}

export function showLog(text) {
    document.getElementById('log').textContent += '\n' +
        window.performance.now() + '-' + text;
}

export function parseSerial() {
    timeStart = window.performance.now();
    var parser = new DOMParser();
    xmlDoc = parser.parseFromString(content, "text/xml");
    showLog("Time for serial parsing: " + (window.performance.now() - timeStart));
}

export function parseAll() {
    timeStart = window.performance.now();
    preparse();
    for (var [k, v] of skeleton.get()) {
        var tr2 = parseChunk(content, k, v);
        mergeTree(fullTree, tr2);
    }
    showLog("Time for all parsing: " + (window.performance.now() - timeStart));
}

```

```
function splitToChunks(array, parts) {
    let result = [];
    for (let i = parts; i > 0; i--) {
        result.push(array.splice(0, Math.ceil(array.length / i)));
    }
    return result;
}

export function parseHybrid() {
    skeleton.clear();
    timeStart = window.performance.now();
    if (!preparse()) {
        showLog('Preparse failed');
        return;
    }
    var numThreads = navigator.hardwareConcurrency;
    var threadFactor = (numThreads - 1) * threadsIdealFactor;
    if (threadFactor > 1) {
        if (skeleton.size() > 1) {
            var widx = 0;
            for (var i in skeleton) {
                for (var [sk, v] of skeleton[i]) {
                    var ww = new Worker('parseChunk.js', { type: "module" });
                    ww.postMessage({ dd: { cnt: (content, [content]), lvlORsize: 'lvl', widx: widx++, tag: sk, startsArr: v } });
                    ww.onmessage = function (e) {
                        processMessage(e, skeleton.size());
                    }
                }
            }
        }
        else if (content.length > minXMLSize && skeleton.size() == 1) {
            var maxThreads = Math.floor(content.length / (content.length / threadFactor));
            var tagsPerThread = Math.floor(lvlCount / maxThreads);
            var startsArr;
            var tag = '';
            for (var [k, v] of skeleton["nodes"]) {
                tag = k;
                startsArr = v;
                if (v.length > threadFactor)
                    startsArr = splitToChunks([...v], threadFactor);
                else
                    startsArr = splitToChunks([...v], 2);
            }
            var widx = 0;

            for (var i = 0; i < startsArr.length; i++) {
                var ww = new Worker('parseChunk.js', { type: "module" });
                ww.postMessage({ dd: { cnt: (content, [content]), lvlORsize: 'lvl', widx: widx++, tag: tag, startsArr: startsArr[i] } });
                ww.onmessage = function (e) {
                    processMessage(e, startsArr.length);
                }
            }
        }
    }
}

function processMessage(e, threadsStarted) {
    mergeTree(fullTree, e.data.result.stree);
    if (threadFinished++ == threadsStarted) {
        console.log("All threads finished: ", threadFinished, threadsStarted);
        showLog("Time for full parallel parsing: " + (window.performance.now() - timeStart));
    }
}

function mergeTree(tr1, tr2) {
    for (var [x, y] of Object.entries(tr2)) {
        for (var [k, v] of y) {
            for (var z in v)
                tr1.addNode(k, v[z]);
        }
    }
}

function preparse() {
    var startTagsCounter = 0;
    var endTagsCounter = 0;
    var depth = 0;
    var startTag, endTag;
    var i = 0;
    var tagsStack = [];
    while (i < content.length) {
        if (content.charAt(i) == '<') {
            var c1 = content.charAt(i + 1);
            if ('<>\\'.includes(c1)) {
                showLog("Start tag invalid");
                return false;
            }

            if (c1 != '/' && c1 != '?' && c1 != '!') {
                var z = i + 1;
                startTag = '';
                startTagsCounter++;
                while (content.charAt(z) != '>') {
                    startTag += content.charAt(z++);
                }
                if ('>\\'.includes(content.charAt(z + 1)) || '</\\'.includes(content.charAt(z - 1))) {
                    showLog("Start tag invalid char at end");
                    return false;
                }
                tagsStack.push(startTag);
                while (content.charAt(++z) < '0');
                if (content.charAt(z) == '<' && depth == 1) {
                    skeleton.addNode(startTag, i);
                }
                i = z;
                if (depth == 1)
                    lvlCount++;
                depth++;
            }
            else if (c1 == '/') {
                var k = i + 2;
                endTag = '';
                endTagsCounter++;
                while (content.charAt(k) != '>') {
                    endTag += content.charAt(k++);
                }
                if (endTag != tagsStack.pop()) {
                    showLog("Tag not found --" + endTag)
                    return false;
                }
                if (k + 1 < content.length && '>\\'.includes(content.charAt(k + 1))) {
                    showLog("End tag invalid char at end " + k + ", " + content.charAt(k + 1));
                    return false;
                }
                i = k;
                depth--;
            }
            else if (c1 == '?' || c1 == '!') {
                var y = i + 2;
                while (content.charAt(y++) != '>');
                i = y;
            }

        }
        else {
            i++;
        }
    }
    if (tagsStack.length > 0) {
        showLog("Not valid XML, stack not empty");
        return false;
    }
    return true;
}
```

parseChunk.js

```
import { wwTree, wwNode } from './classes.js';

onmessage = function (e) {
    if (e.data.dd.lvlORsize == 'lvl') {
        postMessage({ result: { widx: e.data.dd.widx, stree: parseChunk(e.data.dd.cnt[0], e.data.dd.tag, e.data.dd.startsArr) } });
    }
    else {
        postMessage({ result: { widx: e.data.dd.widx, stree: null } });
        console.log("Something went wrong");
    }
}

export function parseChunk(content, tag, arr) {
    var subTree = new wwTree();
    for (var k = 0; k < arr.length; k++) {
        var i = arr[k] + tag.length + 2;
        var startTag = '', endTag = '', value = '';
        var state = -1, prevState = 0;
        var node = new wwNode();
        var prevTag = '';
        var c = '>';
        while (true) {
            while (c == '>' && content.charAt(i) < '0') i++;
            c = content.charAt(i);
            if (c == '<') {
                if (content.charAt(i + 1) == '/') {
                    state = 2;
                    endTag = '';
                    i += 2;
                    prevState = state;
                } else {
                    if (prevState == 1) {
                        node.addNode(startTag);
                        prevTag = startTag;
                    }
                    else {
                        if (prevTag == endTag) prevTag = '';
                    }
                    state = 1;
                    startTag = '';
                    i++;
                    prevState = state;
                }
                continue;
            }
            else if (c == '>') {

                if (state == 1) {
                    value = '';
                    state = 3;
                }
                if (state == 2) {
                    if (endTag == tag) break;
                    if (endTag == startTag) {
                        if (prevTag != '') {
                            node.addNodeItem(prevTag, value);
                        }
                        else {
                            node.addItem(startTag, value);
                        }
                    }
                }
                i++;
                continue;
            }
            if (state == 1) {
                startTag += c;
            }
            else if (state == 2) {
                endTag += c;
            }
            else {
                value += c;
            }
            i++;
        }
        subTree.addNode(tag, node.getNode());
    }
    return subTree;
}
```

classes.js

```
export class wwTree {
    constructor() {
        this.nodes = new Map();
    }
    clear() {
        this.nodes.clear();
    }
    get() {
        return this.nodes;
    }
    addNode(key, node = null) {
        var tmpArr;
        if (this.nodes.has(key)) {
            tmpArr = this.nodes.get(key);
        }
        else {
            tmpArr = [];
        }
        if (node) {
            tmpArr[tmpArr.length] = node;
        }
        this.nodes.set(key, tmpArr);
    }
    size() {
        return this.nodes.size;
    }
}

```

```
export class wwNode {
    constructor() {
        this.nodes = new Map();
    }
    getNode() {
        return this.nodes;
    }
    addItem(key, value) {
        if (!this.nodes.has(key)) {
            this.nodes.set(key, value);
        }
        else {
        }
    }
    addNode(key) {
        if (!this.nodes.has(key)) {
            this.nodes.set(key, []);
        }
        else {
        }
    }
    addNodeItem(key, item) {
        var tmpArr;
        if (this.nodes.has(key)) {
            tmpArr = this.nodes.get(key);
        }
        else {
            tmpArr = [];
        }
        tmpArr[tmpArr.length] = item;
        this.nodes.set(key, tmpArr);
    }
}
```