Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klavdij Oberstar

Večnitno paralelno procesiranje v spletnih brskalnikih z uporabo jezika JavaScript

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Peter Peer
Asistent: as. Bojan Klemenc, univ. dipl. inž.

Ljubljana 2011

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00148/2011
Datum: 02.09.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **KLAVDIJ OBERSTAR**

Naslov: **VEČNITNO PARALELNO PROCESIRANJE V SPLETNIH BRSKALNIKIH Z UPORABO JEZIKA JAVASCRIPT**

MULTITHREADED PARALLEL PROCESSING IN WEB BROWSERS USING JAVASCRIPT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Proučite možnosti večnitnega paralelnega procesiranja v spletnih brskalnikih z uporabo jezika JavaScript. Naredite spletno stran, ki bo na več testnih primerih preverjala, kako dobro spletni brskalniki podpirajo paralelno procesiranje. Testirajte na večjedrnih procesorjih in primerjajte posamezne brskalnike med seboj.

Mentor:
doc. dr. Peter Peer

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Klavdij Oberstar, z vpisno številko **63080118**, sem avtor diplomskega dela z naslovom:

Večnitno paralelno procesiranje v spletnih brskalnikih z uporabo jezika JavaScript

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Petra Peera in asistenta Bojana Klemenca, univ. dipl. inž.,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

Ljubljana, 27. september 2011 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Petru Peeru in še posebej asistentu Bojanu Klemencu za vse nasvete in pomoč pri izdelavi diplomske naloge. Hvala tudi staršem in sestri za vso podporo in potrpljenje tekom študija.

Kazalo

Seznam uporabljenih kratic

SMP (Symmetric Multiprocessing) arhitektura strojne opreme, kjer si dva ali več procesorjev delijo skupen pomnilnik

AMP (Asymmetric Multiprocessing) arhitektura strojne opreme, kjer lahko na vsakem procesorju teče druga aplikacija

API (Application Programming Interface) programski vmesnik

HTML (Hypertext Markup Language) označevalni jezik za izgradnjo spletnih strani

XHTML (eXtensible HyperText Markup Language) razširjena verzija HTML jezika

XML (Extensible Markup Language) označevalni jezik za grajenje strukturiranih dokumentov

CSS (Cascading Style Sheets) jezik namenjen določanju sloga spletnih strani

DOM (Document Object Model) dogovor, kako predstaviti in upravljati z objekti v spletnih dokumentih

WHATWG (Web Hypertext Application Technology Working Group) skupnost ljudi, ki skrbijo za razvoj HTML

W3C (World Wide Web Consortium) mednarodna organizacija za spletne standarde

Povzetek

Paralelno procesiranje je v računalništvu prisotno že zelo dolgo. S prihodom HTML5 pa se je pojavila možnost paralelnega izvajanja tudi v spletnem brskalniku s pomočjo niti v jeziku JavaScript. V diplomski nalogi naredimo splošen pregled paralelnega procesiranja. V nadaljevanju podrobneje pogledamo nove funkcionalnosti HTML5, ki nam pomagajo pri paralelnem izvajanju spletnih aplikacij. Predvsem se osredotočimo na način izdelave niti v JavaScriptu (t.i. Web Workers) in komunikacije med njimi. Za testiranje podpore paralelnemu izvajanju programov v brskalniku smo naredili testno spletno stran v HTML5. Po končanem testu spletna stran prikaže rezultate v obliki grafov in omogoča primerjavo med testi v različnih okoljih. Teste smo opravili na različnih brskalnikih in pri različnem številu niti ter s tem ocenili trenutno podprtost paralelnemu izvajanju programov v brskalnikih.

Ključne besede

paralelno procesiranje, večnitenje, primerjalni testi brskalnikov, JavaScript, Web Workers, HTML5

Abstract

Parallel computing has been present on desktop computers for some time. With the development of HTML5 it is also possible to run parallel algorithms in web browsers with JavaScript threads. In this thesis we make a general review of parallel processing. We look at new functionalities of HTML5 that help us create parallel web applications. We focus on the creation of JavaScript Web Workers and communication between them. For the purpose of testing the support of parallel processing inside web browsers we made a test web site in HTML5. The results of the test are shown on the web page as graphs and enable us to compare our results to the results acquired on different platforms. We performed the tests in different web browsers and with different number of threads to evaluate the current state of JavaScript parallel computing support in web browsers.

Key words:

parallel computing, multithreading, browser benchmark, JavaScript, Web Workers, HTML5

Poglavje 1

Uvod

Spletne aplikacije so iz dneva v dan bolj priljubljene. Uporabljamo jih za druženje s prijatelji, pošiljanje spletne pošte, igranje iger, nakupovanje prek spleta itd. Z vse večjimi potrebami narašča tudi zahtevnost spletnih aplikacij. Posledično se spletne tehnologije temu primerno razvijajo in nadgrajujejo. Zato smo se odločili raziskati, kaj nam ponujajo tehnologije, ki bodo zaznamovale prihodnost spleta. Poglobili smo se v paralelno procesiranje, ki je ena izmed novih funkcionalnosti aplikacij, ki tečejo v spletnem brskalniku.

Razvoj sodobnih procesorjev, tako tistih za namizne računalnike kot tistih v manjših napravah, poteka v smeri paralelnega procesiranja. S prihodom najnovejših različic so spletni brskalniki dobili podporo paralelnemu izvajanju programov v jeziku JavaScript, ki tečejo znotraj spletnega brskalnika. Naš namen je bil ugotoviti, kaj nam ponuja spletna tehnologija naslednje generacije. Usmerili smo se predvsem v teste hitrosti izvajanja različnih algoritmov na brskalnikih in podporo brskalnikov novim funkcionalnostim spleta.

V prvem delu diplomske naloge si bomo pogledali zgodovino paralelnega procesiranja, opisali in razložili bomo pomembnejše pojme, ki so povezani s paralelnim procesiranjem. V drugem delu se bomo osredotočili na paralelno izvajanje aplikacij znotraj brskalnika v jeziku Javascript. Tako bomo predstavili novosti, ki jih prinaša splet naslednje generacije. Za praktični preizkus smo izdelali testno stran napisano z označevalnim jezikom HTML5. Možnost

paralelnega izvajanja v brskalniku je tesno povezana s HTML5, zato bomo bolj podrobno predstavili nekatere njegove značilnosti, ki smo jih uporabili za izdelavo testne strani. Na spletni strani so nanizani različni testi paralelnega izvajanja algoritmov. Testna spletna stran omogoča uporabniku, da preveri podporo svojega brskalnika paralelnemu izvajanju programa v spletnem brskalniku.

V zadnjem delu diplomske naloge si bomo pogledali rezultate testiranja. S testi smo želeli preveriti kakšno je trenutno stanje na področju paralelizma spletnih aplikacij. Teste smo opravili na različnih brskalnikih. Prvi test preverja računanje porazdeljeno med več nitmi, drugi pa preverja izris animacije s pomočjo paralelnosti. Rezultate smo predstavili z grafi. V zaključku so podane ugotovitve o prihodnosti paralelnih spletnih aplikacij.

Poglavje 2

Paralelno procesiranje

2.1 Osnove

Paralelno procesiranje je način procesiranja, kjer se več ukazov izvaja hkrati [12]. Deluje na principu, da se večje operacije razdelijo na manjše, ki se potem izvajajo vzporedno. Preučevanje paralelizma poteka že vrsto let, največ na področju visoko učinkovitega računanja. Zanimanje za paralelno računaje je v zadnjih letih zelo naraslo, saj so paralelni procesorji dostopni tudi širšim množicam uporabnikom. Zaradi omejitve zviševanja frekvence in posledično prevelike porabe energije ter previsokih temperatur osrednjih procesnih enot, so se velika podjetja odločila za izdelavo dveh ali več jeder v procesor z enim podnožjem.

Glede na to kako strojna oprema podpira paralelnost, lahko paralelne računalnike razvrstimo v dve skupini. V prvi skupini so tisti, ki imajo več procesnih enot v enem sistemu: to so večjedrni in večprocesorski računalniki. V drugo skupino pa sodijo tisti, ki imajo več različnih računalnikov povezanih skupaj in opravljajo enake naloge. Programiranje paralelnih programov je zahtevnejše kot programiranje programov, ki se izvajajo zaporedno. Pri paralelnih programih se lahko potencialno pojavijo nove programske napake, ki jih pri vzporednih programih ni. Najpogostejša napaka je sočasen dostop do istega vira, kar privede do napačnega rezultata. Prav tako pa je

komunikacija in sinhronizacija med dvema različnima ukazoma velika omejitev pri doseganju večje učinkovitosti pri paralelnem izvajanju. Največjo možno pohitritev pri vzporednem izvajanju opisuje Amdahlov zakon (več o tem v refsec:amdahl).

2.2 Razvoj

Tradicionalni računalniški programi so bili napisani za zaporedno procesiranje. Algoritem je bil sestavljen iz zaporedja ukazov. Ukazi so se izvajali na eni centralno procesno enoti. Izvajali so se tako, da se po koncu enega ukaza začne izvajati drugi in tako vse do konca algoritma [1].

Paralelno računanje sočasno uporablja več računskih enot za rešitev problema. Tako računanje lahko dosežemo le, če problem razbijemo na več neodvisnih delov, ki jih lahko posamezna računska enota izvede sočasno z ostalimi. Računske enote so lahko različne in lahko uporabljajo različne vire za računanje. Ti viri so lahko računalnik z večjedrnim procesorjem, računalnik z več procesorji, več računalnikov povezanih v mrežo, specializirana strojna oprema ali pa kombinacija vseh naštetih [1].

Z vidika strojne opreme je bilo zviševanje frekvence glavni razlog za napredek v hitrosti računanja od sredine osemdesetih do leta 2004. Poenostavljeno je čas izvajanja programa enak številu ukazov pomnoženo s povprečnim časom potrebnim za dokončanje ukaza. S povečevanjem frekvence oziroma zmanjšanjem časa potrebnega za dokončanje posameznega ukaza, se zmanjša tudi čas izvajanja celotnega programa, pod pogojem, da vse ostalo ostane konstantno [4].

Poraba električne energije se za posamezen čip izračuna po enačbi $P = C \times V^2 \times F$, kjer je P električna energija, C je kapaciteta, V je napetost in F je frekvenca [7]. S povečanjem frekvence se povečuje poraba električne energije. Pojavljajo se tudi problemi zaradi hitrosti električnega signala. Zaradi teh takšno zviševanje frekvence ni bilo več mogoče, zato so morala podjetja leta 2004 začeti izdelovati procesorje z večjedrno arhitekturo.

2.3 Pomnilnik

2.3.1 Deljen pomnilnik

Deljen pomnilnik je skupen pomnilnik z enim naslovnim prostorom, ki je razdeljen med računske enote. Uporablja se v arhitekturi SMP. Prednost takega pomnilnika je relativno lahko programiranje, saj vsi procesorji vidijo enake podatke. Komunikacija med procesorji poteka tako, da eden od procesorjev zapiše podatke na pomnilnik, drugi pa jih prebere. Ker so dostopi do deljenega pomnilnika relativno hitri, tudi komunikacija med procesorji poteka zelo hitro. Slabost takega pomnilnika je ozko grlo pri prenosu podatkov iz pomnilnika v procesorje in obratno, zato imajo običajno takšni sistemi deset ali manj procesorjev. Problematična je tudi ažurnost podatkov. Eden izmed procesorjev spremeni vrednost neke spremenljivke, toda le-ta je shranjena v registrih procesorja. Ostali procesorji morajo za to spremembo vedeti, zato se morajo uskladiti tudi registri ostalih procesorjev [2].

2.3.2 Porazdeljen pomnilnik

Pri porazdeljenem pomnilniku ima vsaka računska enota svoj lokalni pomnilnik. Porazdeljeni pomnilnik je lahko ločen fizično in ne samo logično. Ker ima vsak procesor svoj lokalni pomnilnik, lahko zasede vso pasovno širino in ni potrebe po skladnosti registrov. Zaradi neomejene pasovne širine ima lahko tak sistem neomejeno število procesorjev. Največja slabost takih sistemov je medprocesorska komunikacija. Ker komunikacija poteka po mreži, je potrebno sestaviti sporočilo in ga poslati. Ko procesor prejme sporočilo se njegovo delo prekine z namenom pošiljanja odgovora, kar podaljša čas izvajanja ukaza [2].

2.3.3 Porazdeljeno deljen pomnilnik

Obstaja tudi kombinacija obeh vrst pomnilnikov v navezi z virtualizacijo pomnilnika, pri katerem ima računska enota svoj lokalni pomnilnik in dostop

do pomnilnika drugih procesorjev. Tak sistem ima tako prednosti deljenega in tudi porazdeljenega pomnilnika. Slabe lastnosti takega sistema so nekoliko počasnejše delovanje in potreba po zaklepanju podatkov, da ne pride do hkratnega spreminjanja podatkov [13].

2.3.4 Registri

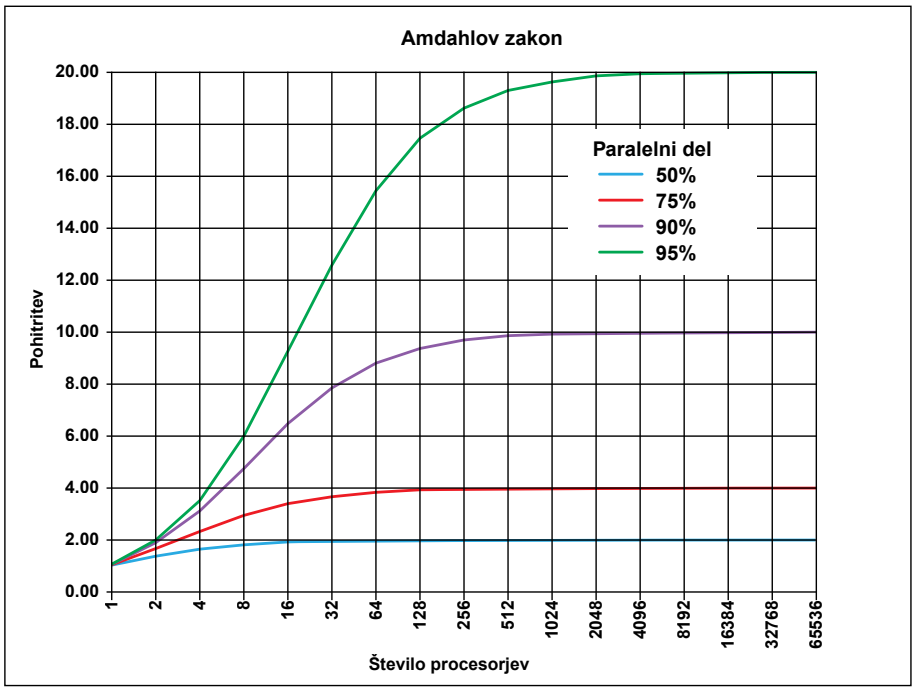
Poleg glavnega pomnilnika imajo računalniški sistemi tudi registre. Registri so majhen, hiter pomnilnik blizu procesorja, v katerem so shranjeni deli glavnega pomnilnika. Podatki se shranijo v registre z namenom hitrejšega dostopa. Algoritem poskuša predvidevati, kateri podatki se bodo v prihodnosti najkasneje potrebovali in jih zamenja s podatki iz glavnega pomnilnika, ki so trenutno potrebni. Pri paralelnih računalnikih se lahko pojavi problem, saj so lahko enake vrednosti shranjene večkrat na različnih lokacijah v registrih, posledica tega pa je nepravilno izvajanje programa. Takšne probleme lahko rešimo s sistemom za skladnost registrov. Sistem sledi zapisom v registrih in jih strateško razvršča ter s tem poskrbi za pravilno izvajanje programa [12].

2.4 Amdahlov zakon

Optimalna pohitritev pri vzporednosti bi bila linearna. Pri podvojitvi računskih elementov bi se moral čas izvajanja prepoloviti. Toda v praksi ima zelo malo algoritmov optimalno pohitritev. Večina je takih, ki se le približujejo linearni pohitritvi in še to le pri majhnem številu računskih enot. Pri določenem številu enot dodajanje nove enote ne pohitri izvajanja.

Potencialno pohitritev algoritmov na paralelnih računalnikih opisuje Amdahlov zakon [14]. Zakon preko formule $\frac{1}{1-P+\frac{P}{N}}$ dokazuje največjo pohitritev, pri paralelizaciji programa pri čemer je P odstotek programa, ki se ga lahko paralelizira. Iz tega sledi, da je $(1 - P)$ del programa, ki se ga ne da paralizirati, N pa število uporabljenih procesorjev. Če se število procesorjev približuje neskončnosti, potem se največja pohitritev približuje $\frac{1}{1-P}$, kar v

praksi pomeni, da ob povečevanju števila procesorjev, tudi pri majhnem neparalelnem delu učinkovitost zelo hitro pada.



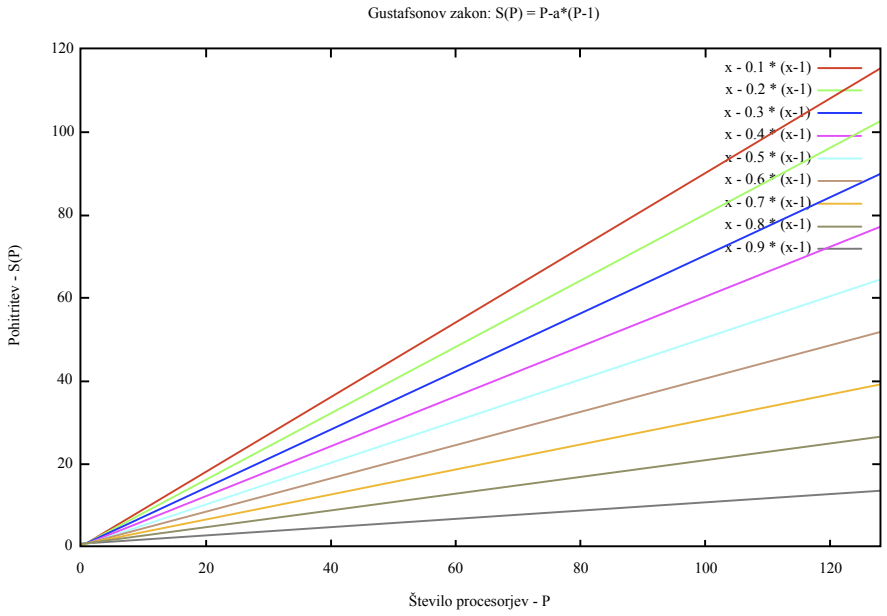
Slika 2.1: Graf, ki prikazuje pohitritve programa po Amdahlovem zakonu [14].

Če imamo algoritem, katerega se lahko 90% paralelizira in 10% ostane neparaleliziranega, lahko pohitrimo delovanje največ za 10 krat ne glede na to koliko procesorjev imamo. Iz grafa na sliki 2.1 je prav tako razvidno, kakšne so možne pohitritve pri različnih velikostih paralelnega dela algoritma. Iz tega lahko sklepamo, da je paralelizacija uporabna le za majhno število procesorjev.

2.5 Gustafsonov zakon

Gustafsonov zakon govori o tem, da se veliki ponavljajoči se podatki zelo lahko paralelizirajo. Zakon nasprotuje Amdahlovem zakonu, ki opisuje ome-

jitve pri pohitritvi, ki jih prinaša paralelizacija [15]. Zakon je opisan z enačbo $S(P) = P - \alpha \times (P - 1)$, pri čemer je P število procesorjev, α ne paraliziran del in S dobljena pohitritev. Z razliko Amdahlovega zakona, le-ta upošteva povečanje računske moči pri povečavi števila računalnikov. V praksi to pomeni da je pohitritev linearna, kar lahko vidimo na sliki 2.2. Vzemimo za primer algoritem, ki se ga da 10% paralelizirati, potem dobimo po Gustafsonovem zakonu pri 16 procesorjih $2,5\times$ pohitritev in pri Amdahlovem zakonu približno 1,1× pohitritev. Ob povečevanju procesorjev se po Amdahlovem zakonu pohitritev približuje vrednosti $\frac{1}{0,9}$ $(1, \bar{1})$, pri Gustafsonovem zakonu pa se pohitritev konstantno povečuje.



Slika 2.2: Graf pohitritev programa po Gustafsonovem zakonu [15].

Izpeljava pohitritve po Gustafsonu:
Pri velikosti problema n razdelimo program na del, ki ga lahko paraleliziramo $b(n)$ in na del, ki ga ne moremo $a(n)$.

$$a(n) + b(n) = 1 \tag{2.1}$$

Če uporabimo P procesorjev, lahko paralelni del izvajamo na vseh P procesorjih, zato ga pomnožimo s P. $S(P)$ je pohitritev zaradi uporabe P procesorjev.

$$S(P) = a(n) + P \times b(n) \tag{2.2}$$

Sedaj paralelni $b(n)$ zapišemo s pomočjo neparalelnega.

$$S(P) = a(n) + P \times (1–a(n)) \tag{2.3}$$

Če privzamemo, da se ob povečavi problema n poveča predvsem zmogljivost procesiranja paralelnega dela, potem ima a(n) vse manjši pomen in je pohitritev enaka P, ko gre n proti neskončnosti. Gustafsonov zakon pravi, da je neparalelni del lahko predstavljen kot konstanta, in ne vpliva na končno pohitritev programa. S tem zakon rešuje vprašanje smisel paralelnega računanja.

2.6 Flynntova taksonomija

Flynnt je razvrstil sisteme po tem ali uporabljajo enega ali več ukazov na enem ali več podatkih [2]. Razdelil jih je v štiri skupine:

- SISD (single-instruction-single-data) popolnoma zaporedno procesiranje; enoprocesorski računalniki (von Neumannova arhitektura),
- SIMD (single-instruction-multiple-data) ponavljenje operacije na večih podatkih; možnost paralelizacija; grafični procesor,
- MISD (multiple-instruction-single-data) le redko uporabljen način,
- MIMD (miltiple-instruction-multiple-data) najpogosteje uporabljen način v paralelnih računalnikih.

Velika večina sodobnih računalnikov spada v klasifikacijo MIMD. Prav tako spada tudi naše testno okolje, na katerem bomo opravili teste paralelnega procesiranja spletnih aplikacij.

2.7 Načini procesiranja večjedrnih sistemov

Obstajajo trije različni načini delovanja operacijskih sistemov v povezavi z arhitekturo strojne opreme [10]. Ti načini so SMP (symmetric multiprocessing), AMP (asymmetric multiprocessing) in BMP (bound multiprocessing). Vsi imajo prednosti in slabosti ter potrebujejo nekakšno komunikacijsko shemo.

2.7.1 Simetrično multiprocesiranje

Pri simetričnem multiprocesiranju (SMP) sta povezana dva ali več identičnih procesorjev, ki imajo skupen glavni pomnilnik ter so nadzorovani z eno instanco operacijskega sistema [10]. Vsi najpogosteje uporabljeni sistemi dandanes uporabljajo SMP arhitekturo. Glavna prednost tega načina je zelo enostavno upravljanje z obremenitvijo posameznega jedra ter dostopnost podatkov za vsa jedra in ukaze. Vsako posamezno jedro je obravnavano kot ločen procesor. SMP ne potrebuje nobenega dodatnega komunikacijskega mehanizma. Komunikacija poteka preko pomnilnika. Edina slabost, ki se pojavi, je nezmožnost delovanja v heterogenih sistemih.

2.7.2 Asimetrično multiprocesiranje

Asimetrično multiprocesiranje (AMP) za razliko od SMP lahko deluje v heterogenih sistemih, prav tako ima lahko več instanc operacijskega sistema, kar pomeni, da ima lahko eno jedro več operacijskih sistemov. Pri tem načinu se pojavi potreba po mehanizmu za komunikacijo med jedri, saj le tako lahko izrabimo ves potencial. Problemi se pojavijo pri večjedrnih sistemih zaradi skalarnosti [10].

2.7.3 Vezano multiprocesiranje

Način BMP pozna vse prednosti simetričnega multiprocesiranja, možnost zaklepanja ukazov na specifična jedra ter zmožnost neodvisnega poganjanja

aplikacij, ki si delijo iste podatke [10].

2.8 Niti

Nit je najmanjša enota procesiranja, ki jo lahko operacijski sistem razvrsti [16]. Niti in procesi se v operacijskem sistemu razlikujejo. Nit je del nekega procesa, ki ima navadno več niti, ki si delijo vire (npr. pomnilnik), ukaze in vrednosti spremenljivk.

Niti so se pojavile že v času enojedrnih procesorjev. Takrat so se niti razvrščale v časovnih intervalih. Vsaka nit je dobila svoj košček procesorskega časa. Zaradi hitrega izmenjevanja izvajanja med različnimi nitmi je uporabnik zaznal izvajanje kot istočasno. Ob prihodu večjedrnih procesorjev se niti izvajajo sočasno vsaka na svojem jedru. Operacijski sistem skrbi za časovno razvrščanje niti na posameznem jedru.

Programerji lahko upravljajo z nitmi preko sistemskih klicev. Implementacije niti se delijo na jedrne in uporabniške niti. Uporabniške niti se uporabljajo pri paralelnem izvajanju časovnikov, signalov ali drugih metod, ki lahko prekinejo izvajanje.

2.8.1 Razlika med nitmi in procesi

Niti in procesi se razlikujejo v več podrobnostih. Procesi so navadno neodvisni, niti pa so del procesa. Nit in proces si delita stanje in vire, medtem ko ima vsak proces svoje stanje in vire. Naslovni prostor se deli med nitmi, procesi pa imajo vsak svojega. Procesi komunicirajo preko sistemske medprocesne komunikacije. Izmenjava podatkov o stanju je hitrejša med nitmi kot med procesi in ne zahteva posredovanja jedra operacijskega sistema. Ustvarjanje in prekinitev niti je veliko hitrejše. V sistemih UNIX celo do desetkrat hitrejše kot pa pri procesih. Preklop med nitmi je hitrejši kot med procesi [16].

2.8.2 Stanja niti

Nit je lahko v stanju pripravljenosti, teku ali pa je blokirana, za razliko od procesa, ki je lahko poleg tega še v stanju mirovanja. Pri nitih takšno stanje ni smiselno, saj je tako stanje koncept procesnega nivoja. Če je proces odstranjen, so skupaj z njim odstranjene tudi vse niti procesa, saj si delijo skupni naslovni prostor [9]. Za spreminjanje stanja niti uporabljamo naslednje štiri operacije:

- Ustvari: Ko ustvarimo nov proces, se ustvari tudi nova nit procesa. Posledično lahko nit v procesu ustvari nove niti. Nova nit ima dodeljen register in prostor na skladu ter je postavljena v vrsto pripravljenih niti.

- Blokiraj: Za vzpostavljanje stanja, ko mora nit počakati na nek dogodek. Procesor lahko med tem časom izvaja drugo nit iz istega ali drugega procesa.

- Odblokiraj: Ko se zgodi dogodek na katerega nit čaka, se le ta premakne v vrsto pripravljenih niti.

- Končaj: Nit se zaključi, registri in sklad se počistijo.

2.8.3 Večnitenje

O večnitenju govorimo, kadar eden izmed procesov vsebuje več niti [16]. Takšno programiranje je široko zastopano v zadnjem času, saj prinaša veliko prednosti. Niti si delijo iste vire, toda se lahko izvajajo neodvisno. Niti se lahko izvajajo paralelno na več različnih procesorjih ali jedrih procesorja. Večnitni programi se posledično izvajajo veliko hitreje na sistemih z več procesorji. Tu se lahko pojavijo težave, na katere mora biti programer pozoren. Paziti je potrebno na pravilno zaporedje računanja, dostop do datotek, da ne pride do smrtnega objema ter poskrbeti za komunikacijo med posameznimi nitmi.

Še ena izmed velikih prednosti večnitnih aplikacij (tudi na eno procesnem sistemu) je odzivnost na uporabniške vnose. Če imamo aplikacijo z eno nitjo, lahko daljši ukaz navidezno zamrzne izvajanje aplikacije, ki postane neodzivna na uporabniške vnose. Pri dveh ali več nitih razvrščevalnik tako razporedi izvajanje, da ostane aplikacija odzivna na vnose uporabnika.

Razvrščanje niti v operacijskih sistemih poteka na dva načina. Prvi način je, da operacijski sistem zazna, kdaj je potrebno zamenjati izvajanje ene niti z drugo. Slabost tega je, da se lahko nit zaključi ob nepravem času in povzroči inverzijo prioritete. Inverzija prioritete se zgodi takrat, ko nit z višjo prioriteto čaka na vire,ki jih je prej zasegla nit z nižjo prioriteto, med tem časom pa se požene nit s srednjo prioriteto. Tega se lahko rešimo tako, da niti same dobijo nadzor nad tem ustavljanjem. Tu se lahko pojavi težava, če nit čaka na dostopnost vira.

2.8.4 Jedrne niti in Uporabniške niti

Jedrne niti so najosnovnejše enote pri razvrščanju v razvrščevalniku operacijskega sistema [9]. Vsak proces ima vsaj eno nit. Kadar obstaja več niti, si le-te delijo pomnilnik in datotečne vire. Nitke ne posedujejo virov, z izjemo sklada, programskega števca in lokalnega nitnega prostora. Jedro lahko nitke razporedi med jedra procesorja.

Nitke implementirane v uporabniškem prostoru se imenujejo uporabniške nitke. Jedro se ne zaveda takih niti. Niti ureja in razvršča aplikacija v uporabniškem prostoru. Uporabniške niti se zelo hitro ustvarijo in urejajo, toda ne izkoristijo večjedrnosti in blokirajo, če blokirajo njihove sistemske nitke. Takšne nitke implementirane v virtualnih strojih se imenujejo zelene niti [9].

2.8.5 Programiranje niti

Veliko programskih jezikov podpira paralelno programiranje. C in C++ ne moreta neposredno ustvariti niti, toda imata dostop do programskega vme-

snika operacijskega sistema, ki omogoča ustvarjanje niti. Obstaja tudi veliko knjižnic za ustvarjanje niti (npr. OpenMP, Pthread, Cilk, MPI). Nekateri jeziki so posebej zgrajeni za podporo paralelnemu procesiranju (npr. Ateji PX, CUDA) [16].

2.9 Problemi paralelnosti

2.9.1 Tekmovalno stanje

Tekmovalno stanje se pojavi takrat, ko si dve različni niti delita nek vir [12]. Takšno stanje se pojavi, ko želita obe niti hkrati dostopati do istega vira. Poskrbeti moramo, da do takega stanja ne pride in na primer zakleniti vir tako, da ena izmed niti ne more brati vira, ki ga je zasegla že druga nit.

Primer takega stanja dobimo, če imamo dve niti (A in B) in spremenljivko S (vrednost je 0), kateri moramo povečati vrednost za dva. Nit A prebere spremenljivko in jo shrani v register, prav tako jo prebere tudi nit B. Obe niti imata sedaj v svojem registru shranjeno vrednost 0. Niti nato povečata vrednost spremenljivke za ena in jo shranita nazaj v samo spremenljivko. Spremenljivka S ima sedaj vrednost 1, kar pa je narobe, saj bi vrednost morala biti 2. Pravilno bi bilo, da bi prva nit prebrala vrednost in jo povečala. Druga nit bi čakala, da se proces zaključi. Ko bi se povečana vrednost zapisala nazaj v spremenljivko, bi druga nit vrednost prebrala in jo povečala. Tako bi dobili pravilni rezultat.

Pri zaklepanju virov moramo paziti, da ne pride do smrtnega objema. Smrtni objem se pojavi, ko ena nit čaka drugo, da sprosti vir, druga pa čaka prvo, da sprosti svojega. Za to poskrbijo algoritmi, ki preverjajo, kdaj lahko pride do smrtnega objema in ga ustrezno preprečijo ali rešijo.

2.9.2 Sinhronizacija

Niti morajo biti med seboj sinhronizirane, da ne pride do izvajanja dela programa, ki se že izvaja. Nit mora počakati, da druga zaključi izvajanje.

Da vzpostavimo sinhronizacijo, lahko uporabimo zapreko (angl. barrier) kot enega izmed sinhronizacijskih algoritmov. Obstajajo tudi drugi algoritmi, ki poskrbijo za sinhronizacijo med nitmi [12].

2.9.3 Paralelna upočasnitev

Pri paralelizaciji programa vedno ne dosežemo pohitritve. Ustvarjanje novih niti zahteva svoj procesorski čas. Če ustvarimo preveč niti, lahko te porabijo več časa za komunikacijo med seboj kot za samo reševanje problema. Upočasnitev lahko povzroči tudi razbitje problema na premajhne dele. Število niti je treba skrbno načrtovati in premisliti ali so nove niti resnično potrebne [12].

2.10 Paralelno programiranje

Paralelno procesiranje se je v zadnjem času zelo razširilo največ na račun večjedrnih procesorjev, v zadnjem času pa tudi na račun grafičnih kartic. Te so poleg procesiranja grafike sposobne tudi procesiranja drugih programov. Da pa bi popolnoma izkoristili prednost paralelnosti, moramo prilagoditi način programiranja. Program je potrebno razdeliti na več niti, da se jih lahko izvaja na več jedrih. Tu se pojavi težava, saj je včasih zelo težko najti paralelizem v kodi.

2.10.1 Iskanje paralelizma

Nekatere algoritme je relativno lahko paralelizirati. Kot primer, če za košnjo travnika z eno kosilnico porabimo 4 ure, bomo z dvema kosilnicama rabili zgolj dve uri. Prav tako pa je v računalništvu. Dve niti bosta dvakrat hitreje pretvorili barvno sliko v črno-belo. Vsaka nit bo vzporedno z drugo pretvorila svojo polovico slike. Veliko takih algoritmov je že paraliziranih.

Na drugi strani pa imamo algoritme, ki se lahko vsaj delno paralelizirajo. Dva kuharja ne bosta mogla dvakrat hitreje skuhati, saj bosta morala

občasno čakati en drugega, toda še vedno bosta hitreje opravila delo kot pa en sam. Večina algoritmov je bila napisanih za zaporedno izvajanje in so bili posledično tudi tako optimizirani in prilagojeni. Za pretvorbo takih algoritmov v paralelne, je potrebno algoritem dodobra preučiti in po potrebi odstraniti ali spremeniti določen del. Še posebej moramo biti pozorni kaj lahko odstranimo in spremenimo, da bo algoritem še vedno deloval pravilno in bo dosežena pohitritev [17].

2.10.2 Odpravljanje napak

Odpravljanje napak paralelnih algoritmov je zelo težko delo. Napake se lahko pojavljajo naključno in jih je zelo težko odkriti z običajnimi tehnikami odkrivanja napak. Niti se lahko izvajajo v različnem vrstnem redu. V veliko primerih ni važno kako se niti izvajajo in kdaj se končajo. Določeni algoritmi pa zahtevajo določen vrstni red, pri tem pa lahko pride do napake, če se niti ne izvedejo v pravilnem vrstnem redu. Takrat je potrebno vgraditi določeno logiko, ki prepreči nepravilno izvajanje niti [17].

2.10.3 Optimizacija

Glavni namen paralelizacije je hitrejše delovanje. Če nismo pazljivi, se zelo pogosto zgodi, da je paraleliziran algoritem počasnejši od zaporednega. Največkrat so niti preveč odvisne od zaporedja izvajanja, tako da se večino časa čakajo, ali ne pride do vzporednega izvajanja, ker vse niti zahtevajo iste sistemske vire. Kodo je potrebno skozi več iteracij popraviti in ali spremeniti, da dobimo optimalno delovanje.

Strojna oprema se nenehno nadgrajuje in spreminja. Veliko starih programov, ki so bili napisani na primer za dvojedrne procesorje, je potrebno sedaj prilagoditi za štiri ali večjedrne. Včasih je potrebno že napisane algoritme za večjedrne sisteme popraviti, saj na novi strojni opremi ne tečejo več optimalno.

Poglavje 3

HTML5 in JavaScript

Čeprav so programi, ki izkoriščajo možnosti paralelnega računanja že dalj časa prisotni pri osebnih računalnikih, ni bilo mogoče izkoriščati paralelnega procesiranja (brez uporabe vtičnikov) v programih, ki tečejo neposredno v spletnih brskalnikih. S prihodom novega standarda HTML5 je to postalo možno z uvedbo niti v JavaScriptu – t.i. Web Workers. Za naše testiranje paralelnosti v spletnem brskalniku bomo potrebovali HTML5, zato si bomo v tem poglavju pogledali nekaj ključnih lastnosti HTML5, ki smo jih uporabili pri izdelavi testne strani.

3.1 Razvoj HTML5

HTML5 je peta verzija standarda Hypertext Markup Language in je še vedno v razvoju. Glavni namen te različice je izboljšanje podpore multimedijskim vsebinam in obdržati berljivost za ljudi ter razumljivost za računalnike in naprave. Glavni namen HTML5 ni le zajeti HTML4, ampak tudi eXtensible HyperText Markup Language (XHTML) in Document Object Model (DOM). HTML5 je poskus opredelitve enotnega označevalnega jezika, ki združuje tako HTML kot XHTML. Poleg tega vključuje podrobni procesni model, ki spodbuja bolj interoperabilne implementacije. S tem omogoča izdelavo bolj zapletenih spletnih strani [5].

V praksi HTML5 doda veliko novih spletnih komponent. To so na primer `<video>`, `<audio>` in `<canvas>`. Kot tudi podporo za vektorsko grafiko zapisa SVG. Nove komponente omogočajo lažjo izdelavo multimedijske in grafične podobe spletne strani, predvsem brez uporabe dodatnega programskega vmesnika ali vtičnikov. Ostale nove značke, kot so `<section>`, `<article>` in `<nav>` pa so tu z namenom obogatiti izgled besedila. Prav tako je uvedenih nekaj novih atributov, medtem ko so bili nekateri stari atributi in elementi odstranjeni. Nekatere stare značke (`<a>`, `<cite>`, `<menu>`) so spremenjene, posodobljene ali pa standardizirane. Programski vmesnik in DOM so postali osnovni del HTML5 specifikacije. HTML5 opredeli potrebno procesiranje nepravilnih značk tako, da so sintaktične napake obravnavane enotno pri vseh spletnih brskalnikih in drugih uporabniških agentih.

Standardizacija

WHATWG je začela z delom na specifikaciji v juniju 2004 pod imenom Web Applications 1.0. Od januarja 2011 naprej pa je specifikacija kot osnutek standarda pri WHATWG in v delovnem osnutku pri W3C. Zaradi dolgega procesa standardizacije bo verjetno HTML5 popolnoma implementiran v brskalnikih, še preden bo dokončana standardizacija. Osnutek HTML5 je bil preložen za osem mesecev, čeprav bi moral biti standard potrjen že leta 2010. Po Hicksonu naj bi HTML5 kandidiral za standardizacijo leta 2012. Končni standard naj bi postal leta 2014. Ne glede na vse pa je veliko komponent HTML5 dokončanih in so že implementirane v spletnih brskalnikih [5].

Podpora brskalnikov

V zadnjem času večji spletni brskalniki tekmujejo, kateri bo imel boljšo podporo HTML5. Posledično je velika večina novosti že implementiranih in pripravljenih za uporabo. Najboljšo podporo ima brskalnik Google Chrome, sledita mu Mozilla Firefox in Opera, najslabše podprt pa je brskalnik Internet Explorer [6]. Podpora iz meseca v mesec narašča in vse več razvijalcev

spletnih strani se odloča za izdelavo spletne strani v standardu HTML5.

3.2 Pregled novih elementov v HTML5

HTML 5 je vpeljal veliko novih elementov in dodal nove atribute, ki spreminjajo lastnosti elementov. Novi elementi nam omogočajo lažjo izgradnjo moderne spletne strani. Prav tako pa odstranijo potrebo po uporabi vtičnikov in drugih dodatkov, saj nove funkcionalnosti omogočajo izdelavo multimedijske strani. Nove elemente bomo potrebovali pri izgradnji testne strani. Za izrisovanje grafov bomo uporabili element canvas. Zato si bomo v nadaljevanju pogledali, kako se uporablja nove elemente in kaj nam novi elementi omogočajo.

Strukturni elementi

HTML5 je vpeljal elemente, ki določajo strukturo spletne strani [6]. Spletne strani imajo navadno navigacijo, glavo, nogo, stranske menije in ostale sekcije. Ker se sedaj v glavnem uporabljata elementa `<div>` ali `<span>`, so v HTML5 dodani novi elementi, ki olajšajo izdelavo in boljšo razčlenitev spletne strani. Ti elementi so:

- `<nav>` navigacija na spletni strani,
- `<section>` odsek strani,
- `<header>` glava spletne strani,
- `<footer>` noga spletne strani,
- `<article>` članek ali pa primarna vsebina strani,
- `<aside>` stranska vrstica ali pa dodatna vsebina in
- `<figure>` slike, ki so del članka.

Vrstni elementi

Vrstni elementi so elementi, ki omogočajo označevanje teksta in strojno razumevanje označene vsebine [6]. Označen teksta lahko na primer preberemo s skriptnim jezikom. Med te elemente spadajo:

- `<mark>` za označevanje teksta,
- `<time>` za označevanje dela vsebine, ki je datum ali čas,
- `<meter>` za označevanje vrednosti na lestvici (npr. 2 od 10),
- `<progress>` za prikaz napredka, nekega ukaza.

Podpora dinamičnim stranem

HTML5 je bil ustvarjen z namenom podpore razvijalcev spletnih aplikacij. Zato je zelo poenostavljeno izdelovanje dinamičnih strani [6]. HTML5 podpira menije, ki so bili odstranjeni iz HTML4. Atribut '`href`' iz elementa za povezavo je neobvezen, kar omogoča enostavnejše klice skriptnih funkcij. Element `<script>` ima dodaten atribut '`async`', ki pove brskalniku, da se lahko skripta nalaga asinhrono. Pri asinhronem nalaganju se vsebina spletne strani naloži neodvisno od skripte. Tak način prepreči zakasnitve pri prikazu spletne vsebine, saj se ta naloži pred skripto. Dodatni elementi za podporo dinamičnim stranem pa so:

- `<details>` prikaže podrobnosti o elementu,
- `<datagrid>` ustvari tabelo, ki ima podatke iz baze ali drugega dinamičnega vira,
- `<menu>` za ustvarjanje sistema menijev,
- `<command>` definira akcijo, ki se bo zgodila, ko je dinamični element aktiviran.

Novi tipi obrazcev

Obrazci imajo sedaj poleg starih elementov še naslednje tri elemente:

- `<datalist>` definira listo izbir za vnosno polje,
- `<keygen>` omogoča šifrirano prenašanje podatkov,
- `<output>` za prikaz različnih rezultatov, kot na primer vrnjena vrednost funkcije.

Poleg novih elementov pa ima element `<input>` definirane nove vrednosti atributa 'type'. Sedaj lahko poleg standardnih tipov vnašamo še: email, url (spletni naslov), number (številko), range (vrednost v obsegu števil), date (datum, čas, mesec...), search (iskalni niz) in color (barvo) [6].

Element `<canvas>`

Element `<canvas>` omogoča izrisovanje grafike na spletni strani. Za izrisovanje grafičnih gradnikov uporablja JavaScript. Element določa le prostor, kamor se bodo gradniki izrisali. Uporaben je za izrisovanje grafov, grafike iger ali drugih vizualnih slik v realnem času. Element `<canvas>` določi kvadratni prostor, kjer imamo kontrolo nad posameznim pikslom. V elementu `<canvas>` lahko izrisujemo črte, kvadrate, kroge, pisave in slike. Koordinatno izhodišče se nahaja v zgornjem levem kotu. Grafične gradnike izrisujemo tako, da jim podamo koordinate in velikost [6].

Odstranjeni elementi

Poleg dodanih novih elementov je tukaj tudi nekaj takih, ki so bili odstranjeni iz specifikacije HTML5. Veliko takih elementov je že sedaj opuščenih. V glavnem gre za elemente, ki so namenjeni oblikovanju spletne strani. Oblikovanje je v sodobnih spletnih straneh prevzel CSS. Nekateri elementi so zamenjani, kot je na primer element `<applet>`, spet drugi zaradi neuporabnosti odstranjeni [6].

Novosti oblikovanja v CSS3

CSS je postal neločljivi del oblikovanja spletnih strani. Skupaj s prihodom HTML5 je prišel tudi novi kaskadni slog verzije 3. CSS3 je prinesel veliko novosti pri oblikovanju spletnih elementov. Ker je CSS3 še v razvoju, vse novosti oblikovanja še niso popolnoma definirane in posledično tudi niso popolnoma implementirane v spletne brskalnike. Toda glavnina jih je in se tudi že uporabljajo za izdelavo sodobnih spletnih strani. Tudi mi bomo za oblikovanje testne strani uporabili nove možnosti oblikovanja kaskadnega sloga inačice 3 [3].

3.3 JavaScript

JavaScript je objektno usmerjen skriptni programski jezik in je implementacija jezikovnega standarda ECMAScript [11]. Uporablja se kot skriptni jezik na strani odjemalca (v spletnem brskalniku) za izdelavo dinamičnih spletnih strani in izboljšanje uporabniškega vmesnika. Omogoča dinamično spreminjanje gradnikov spletne strani.

JavaScript se uporablja tudi v aplikacijah, ki niso del spletne strani. Taka primera sta dokument PDF in dodatki namizja. Novejša in hitrejša JavaScript VM omogoča izdelavo spletnih aplikacij na strani spletnega strežnika.

Nove funkcionalnosti

HTML5 ni prinesel le novih elementov in naprednega oblikovanja s CSS3, ampak tudi nove zmožnosti skriptnega jezika JavaScript. JavaScript je pridobil veliko novih in naprednih funkcij predvsem zaradi prej omenjenih novosti v HTML in CSS. Največ z namenom opustitve uporabe različnih vtičnikov, kot je na primer Flash. JavaScript prinaša veliko novih zmožnosti pri ustvarjanju dinamične vsebine na spletni strani.

Novi selektorji

JavaScript ima funkcije, ki omogočajo izbiro elementov HTML nad katerimi lahko nato izvajamo različne operacije. Nova inačica JavaScript ima poleg starih funkcij kot so selektor po id ali imenu, tudi možnost izbire po imenu razreda (`getElementsByClassName('string')`). Ime razreda označuje sklic na oblikovalski slog v CSS. Poleg tega sta tu še funkciji (`querySelectAll('string')` in `querySelect('string')`)za izbiro vseh elementov, ki se ujemajo s podano CSS sintakso [6].

Spletno skladiščenje in baza SQL

Spletno skladiščenje omogoča hranjenje podatkov lokalno na strani odjemalca. Podatki se shranjujejo kot ključ in vrednost. Podobno kot piškoti, ti podatki ostanejo shranjeni, čeprav je uporabnik že zapustil stran, zaprl zavihek ali brskalnik. Za razliko od piškotov se ti podatki nikoli ne pošljejo na spletni strežnik [6]. Tako lahko pišemo bolj zapletene programe, ki se bodo izvajali v spletnem brskalniku, saj se njihovi podatki ne bodo izgubili, ko bomo zapustili stran.

Paralelno izvajanje

Ena izmed novih funkcionalnosti JavaScripta je tudi paralelno izvajanje s pomočjo niti imenovanih tudi Web Workers. V naslednjem poglavju si bomo podrobneje pogledali njihove lastnosti, kako delujejo in kako jih ustvarimo.

Poglavje 4

JavaScript in paralelno izvajanje

4.1 Niti v jeziku JavaScript

Ena izmed novih funkcionalnosti JavaScripta so tudi takoimenovani Web Workers, ki prinašajo možnost ustvarjanja niti v JavaScriptu. To nam omogoča poganjanje daljših skript ne da bi osrednja stran postala neodzivna. Predvsem pa je bil glavni namen uvedbe Web Workers, pohitritev izvajanja skript na spletni strani in s tem omogočiti izdelavo kompleksnejših spletnih aplikacij. Web Workers ustvarijo novo nit na nivoju operacijskega sistema, kar pomeni, da lahko tečeta dve niti na ločenem jedru procesorja, če seveda skripte zaženemo na večjedrnem sistemu. Slaba lastnost tega je, da je za ustvarjanje nove niti (Web Worker) potrebno veliko procesorskega časa in veliko pomnilniškega prostora [6].

Prva specifikacija, kjer so bili opisani Web Workers, je bila javnosti predstavljena aprila leta 2009. Specifikacija Web Workers še vedno ni končana in se iz meseca v mesec nadgrajuje in popravlja. –Vendar pa lahko v bližnji prihodnosti pričakujemo končno verzijo.

4.1.1 Omejitve

Web Workers delujejo neodvisno od niti uporabniškega vmesnika, zato nimajo dostopa do vseh funkcij JavaScripta. Glavna omejitev je nezmožnost dostopa do DOM. Web Workers ne morejo brati in spreminjati dokumenta HTML, dostopati do globalnih spremenljivk ali funkcij. Omejen je dostop do nekaterih objektov, kot primer so nastavitve `windows.location` mogoče le za branje [6].

4.1.2 Prednosti

Ne glede na vse omejitve pa lahko Web Workers uporabljajo standardne podatkovne tipe JavaScript, upravljajo s klici `XMLHttpRequest` (Ajax), uporabljajo časovnike in lahko uvozijo druge Web Workerse. Web Workers so idealni za računanje časovno potratnih zaporedij operacij, kot so analiziranje velikega števila podatkov, umetna inteligenca, sledenje žarkom in drugi [6].

4.1.3 Podpora brskalnikov

Trenutno so Web Workers podprti le v najnovejših inačicah spletnih brskalnikov Firefox, Chrome, Safari in Opera, vendar ne v celoti. Web Workers niso podprti v trenutni inačici Internet Explorerja, bili pa naj bi v naslednji inačici. Posledično izdelava spletne strani z Web Workers ni najboljša izbira, saj še vedno velik odstotek uporabnikov uporablja brskalnik Internet Explorer [6].

4.2 Programiranje z Web Workers

4.2.1 Ustvarjanje

Nov Web Worker ustvarimo tako, da kličemo konstruktor `Worker()`. V konstruktor podamo naslov skripte, ki se bo izvajala v novi niti Web Workerja [6]. Web Worker kreiramo na naslednji način:

```
var worker = new Worker('worker.js');
```

4.2.2 Komunikacija

Ker Web Worker ne more dostopati do DOM ali drugih funkcij na spletni strani, vsa komunikacija poteka preko dogotkovnega vmesnika. Skripta spletne strani pošlje podatke Web Workerju preko argumenta metode `postMessage()`. Ko Web Worker pošlje podatke nazaj, jih skripta spletne strani prestreže kot dogodek in sproži izvajanje določene funkcije. Primer:

Skripta spletne strani:

```
var worker=new Worker('worker.js');

//funkcija, ki se izvede ob prispelih
//podatkih iz Web Workerja
worker.onmessage = function(e){
        alert(e.data);
};

//poslani podatki Web Workerju
worker.postMessage('Alert!');
```

Web Worker sprejema in pošilja podatke na enak način kot skripta spletne strani.

Poslani podatki so lahko tipa string, number, boolean, array, object, null ali undefined. Podatki se med procesom komunikacije kopirajo. Web Worker in glavna skripta si ne delijo podatkov, ampak imajo vsak svojo primerek le-teh [6].

4.2.3 Lovljenje napak

Kadar se v Web Workerju pojavi napaka, le-ta pošlje dogodek skripti spletne strani [6]. Dogodek z napako ima naslednja tri polja:

- `filename`: ime skripte, katera je povzročila napako,
- `lineno`: številka vrstice, kjer se je zgodila napaka,
- `message`: opis napake.

Primer lovljenja napake v skripti spletne strani:

```
worker.onerror = function(e) {
        alert('Error in file: '+e.filename+' Line:
'+e.lineno +' Description: '+e.message);
};
```

4.2.4 Uvoz skript in knjižnic

Niti Web Workerjev imajo dostop do globalne funkcije `importScripts()`, ki omogoča uvoz skript in knjižnic. Funkcija kot argument sprejme nič ali več naslovov virov. Brskalnik nato naloži in požene uvožene skripte. Vsi globalni objekti iz posamezne skripte so dostopni Web Workerjem [6].

4.2.5 Končanje izvajanja

Web Worker lahko iz skripte spletne strani zaključimo s klicem metode `terminate()`. Nit Web Workerja se nemudoma ustavi brez možnosti dokončanja ukaza. Web Worker se lahko zaključi tudi sam s klicem svoje lastne metode `close()` [6].

Poglavje 5

Testiranje paralelnega izvajanja programov v spletnem brskalniku

Naš namen je bil izdelati testno stran, na kateri bi lahko preverili zmogljivost paralelnega izvajanja spletnih aplikacij. Na spletu obstaja več različnih testov paralelnega izvajanja. Takšna primera sta spletna stran MoonBat JavaScript Benchmark [18], ki ponuja teste hitrosti paralelnega izvajanja različnih algoritmov in JavaScript Web Workers test [19], ki omogoča izvajanje testa algoritma na različnih nastavitvah. Na naši testni strani smo nanizali več različnih testov, ki se lahko izvajajo neodvisno eden od drugega. Testna stran omogoča izvajanje testov le na tistih brskalnikih, ki podpirajo ustvarjanje Web Workerjev.

Na testni strani preverjamo hitrost paralelnega izvajanja pri različnem številu niti. Izdelali smo skripto, ki požene teste pri različnem številu niti. Testno stran smo nadgradili tako, da po končanem testiranju rezultate izriše na grafu. Tako lahko lažje primerjamo rezultate. Kot dodatno primerjavo pa smo dodali rezultate iz našega testiranja v testnem okolju. Rezultati iz testnega okolja bodo predvsem uporabni v prihodnosti za primerjavo z rezultati iz novejših brskalnikov in sodobnejše strojne opreme.

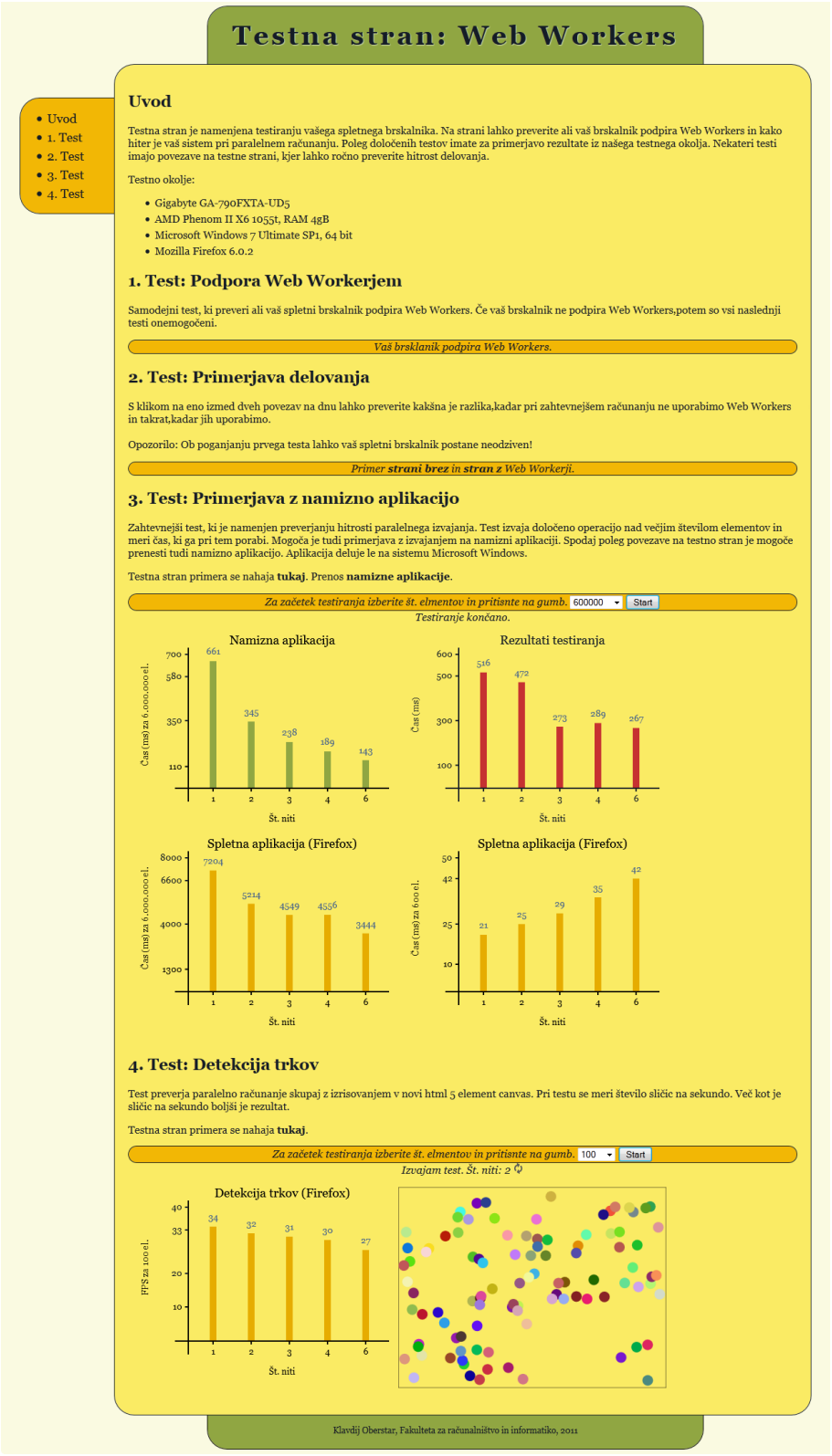
Na testni strani smo predvsem želeli dodati teste na različnih nivojih. Od tistih najpreprostejših, ki preverijo podporo Web Workerjem, do tistih malo bolj zahtevnejših. Tako testna stran omogoča celotno testiranje paralelnega procesiranja na spletnih straneh.

5.1 Izdelava testne strani

Testna stran je napisana z označevalnim jezikom HTML5. Tako hkrati preverimo tudi kako spletni brskalnik podpira nove elemente HTML5 [20]. Predvsem smo potrebovali nove elemente HTML5 za prikaz rezultatov. Pri izdelavi smo poleg novih elementov HTML5 uporabili tudi nove sloge oblikovanja CSS3. Sloge oblikovanja smo uporabili za oblikovanje testne strani in vsebine. Na testni strani se nahajajo različni testi podpore paralelnemu izvajanju. Uporabnik lahko posamezen test požene in preveri zmogljivost svojega računalnika, predvsem pa spletnega brskalnika. Ko se test zaključi, se rezultati izpišejo v grafu. Vsak test ima poleg možnosti testiranja tudi predstavljene rezultate našega testiranja. Rezultate smo predstavili z namenom primerjave z drugimi sistemi, spletnimi brskalniki in morebitnim razvojem na področju podpore paralelizaciji spletnih aplikacij. Slika 5.1 prikazuje spletno stran s testi.

5.2 Testi na testni strani

Na strani so nanizani štirje testi, ki preverijo zmogljivost brskalnika in strojne opreme. Testi preverijo podporo brskalnika Web Workerjem, razliko med računanjem v skripti glavne strani ter v Web Workerjem, hitrost paralelnega računanja in hitrost izrisa animacije v elementu canvas s paralelnim računanjem koordinat.



Slika 5.1: Spletna stran v HTML5 za testiranje zmožnosti paralelnega procesiranju v JavaScriptu.

5.2.1 Podpora Web Workerjem

Prvi test na strani preveri ali spletni brskalnik, s katerim pregledujemo stran, podpira Web Workerje. Če spletni brskalnik podpira Web Workerje, potem je mogoče izvajanje nadaljnjih testov na strani. V nasprotnem primeru so testi onemogočeni. V ozadju test s funkcijo v JavaScriptu preveri, ali je mogoče ustvariti nov Web Worker. Ob potrditvi ne stori ničesar in testi ostanejo omogočeni, če pa ustvarjanje ni mogoče, se gumbi za začetek izvajanja testov onemogočijo.

5.2.2 Primerjava delovanja

Da bi, kar se da najbolje, predstavili primerjavo med izvajanjem algoritma z in brez Web Workerjev, smo izdelali dve testni podstrani. Na prvi podstrani se algoritem, ki napolni veliko tabelo z naključnimi števili, izvaja v glavni niti, ne da bi se ustvaril dodatni Web Worker. V drugi podstrani pa se ustvari nov Web Worker, na katerem se nato izvede algoritem. Namen tega testa je bil predstaviti neodzivnost ali celo zamrznitev spletnega brskalnika ob neuporabi Web Workerjev pri algoritmih z daljšim časom izvajanja.

5.2.3 Primerjava s programom v jeziku C++

Za test smo uporabili algoritem, ki ga je zelo preprosto paralelizirati. Algoritem sprejme tabelo števil. S številom iz tabele pomnoži v naprej pripravljeno matriko in nato sešteje vsa števila iz pomnožene tabele. Dobljeno vrednost nato deli s številom elementov v matriki in vrednost shrani nazaj v tabelo. Algoritem smo paralelizirali tako, da se tabela razdeli na manjše dele in se ti nato obdelajo v ločenih nitih. Ustvarjanje novih niti je mogoče le iz glavne skripte spletne strani, ni pa mogoče iz že obstoječe niti.

Test prikaže uporabniku potreben čas za dokončanje algoritma za posamezno število niti. Rezultati se izpišejo v grafu. Uporabnik lahko test izvede nad različno veliko tabelo. Polega samega testa je na strani tudi povezava na podstran in prenos namizne aplikacije. Na podstrani je mogoč opraviti

posamezen test z različnim številom niti in velikostjo tabele. Enako deluje tudi program v C++.

Namen testa je prikaz hitrosti delovanja algoritma v spletnem brskalniku, v primerjavi z izvajanjem algoritma v navadni C++ aplikaciji. Lahko pa se uporabi tudi kot splošno testiranje hitrosti izvajanja v določenem sistemu.

5.2.4 Detekcija trkov

Test izrisuje animacijo odbijanja žogic od sten in pri tem meri število sličic na sekundo. Algoritem najprej napolni tabelo objektov (žogice), ki imajo določene atribute kot so pozicija, hitrost, velikost in barva. Ko je tabela napolnjena, se začne simulacija premikanja. Algoritem v vsakem obhodu zanke spremeni položaj posamezne žogice in preveri ali se je ta zadela v steno. Ob primeru trka se žogica odbije od stene in nadaljuje premikanje v drugi smeri. Žogice in njihova animacija je prikazana s pomočjo novega elementa canvas.

Test požene algoritem na različnem številu niti in izpiše v grafu doseženo število sličic na sekundo. Več kot je sličic na sekundo boljši je rezultat. Prav tako kot v prejšnjem testu, je tu povezava na podstran, kjer je možno ročno pognati teste.

S testom je predstavljen praktični primer uporabe Web Workerjev skupaj z novim elementom HTML5 canvas. Namen testa je prikaz pohitritev oziroma upočasnitev animacije v primeru, da bolj zapletene izračune opravimo v ločenih nitih. Test naj bi bil zelo preprost primer, ki simulira obdelavo grafike pri igrah.

5.3 Primerjalni testi Web Workerjev

Da bi bolje raziskali podporo paralelizaciji na spletni strani, smo v testnem okolju pognali algoritme, ki smo jih uporabili na testni strani. Dobljene rezultate smo uporabili kot primerjalne primere pri posameznem testu na testni strani.

5.3.1 Testno okolje

Za testno okolje smo uporabili namizni osebni računalnik s procesorjem AMD Phenom II X6 1055t, matično ploščo Gigabyte GA-790FXTA-UD5 in 4gB rama tipa DDR3. Na računalniku je nameščena 64-bitna različica operacijskega sistema Microsoft Windows 7 Ultimate SP1.

Vse primerjalne teste na spletni strani smo pognali v brskalnikih Mozilla Firefox 6.0.2, Google Chrome 14, Internet Explorer 9 in Opera 11.51. Za brskalnike smo se odločili zaradi priljubljenosti med uporabniki. Za primerjavo smo napisali program v jeziku C++ in uporabili API operacijskega sistema Windows za ustvarjanje niti.

5.3.2 Testiranje preprostega algoritma

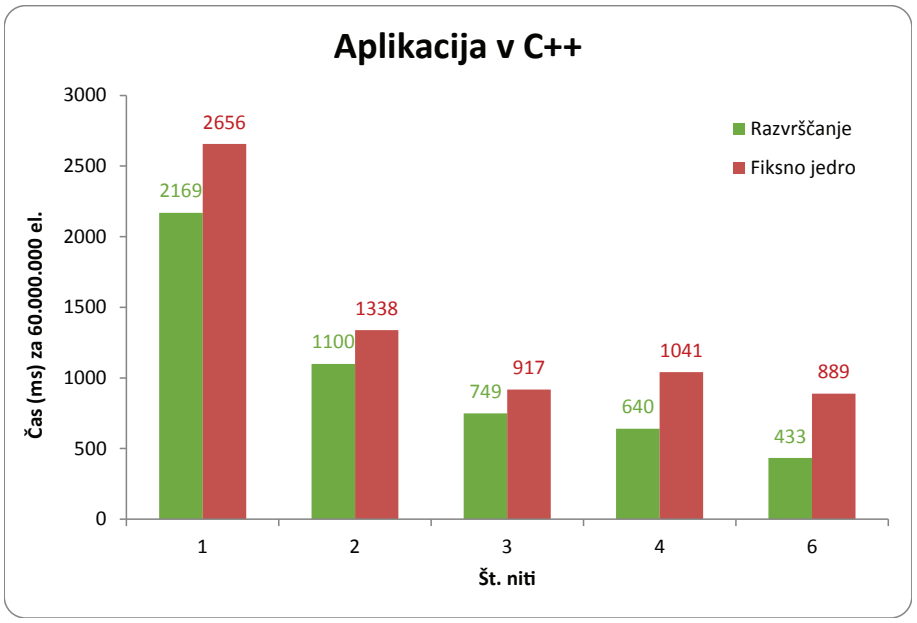
Za testiranje preprostega algoritma smo najprej pripravili spletno stran za testiranje. Preko spletne strani je mogoče vnesti število elementov v tabeli in izbrati število ustvarjenih niti. Ob pritisku na gumb za začetek se ustvari tabela naključnih števil. Tabela se nato glede na število niti razdeli in ustvarijo se Web Workerji. Skripta nato pošlje Web Workerjem ukaz za začetek računanja. Čas izvajanja smo merili od začetka ustvarjanja Web Workerjev do takrat, ko so vsi Web Workerji končali z računanjem.

V programu C++ smo za ustvarjanje niti uporabili API operacijskega sistema Windows, ki omogoča kreiranje niti na jedrnem nivoju. Nit operacijskega sistema Windows ustvarimo tako, da kličemo funkcijo `CreateThread`, ki je del knjižnice `windows.h`. Po klicu funkcije se ustvari nova nit na jedrnem nivoju in kliče se funkcija, v kateri je naša koda ter se začne izvajati.

Za primerjalni test med spletno in običajno aplikacijo smo uporabili tabelo naključnih števil velikosti 6 000 000. Teste smo izvajali pri različnem številu ustvarjenih niti. Začeli smo pri eni niti, ter nadaljevali z dvema, tremi, štirimi, šestimi, osmimi, dvanajstimi in šestnajstimi nitmi. Web Worker nima možnosti nastavitve prioritete, kot jo imajo ostale niti operacijskega sistema Windows, zato smo se morali zanašati na to da bo operacijski sistem

sam razvrstil niti tako, da bo vsaka tekla na svojem jedru.

Razvrščevalnik Windows razporeja niti in ne procese [8]. Razporejanje niti temelji na prioritetah in razvrščanju s krožnim dodeljevanjem. Windows pregleda stanja jeder in določi nit tistemu, ki je v mirovanju. V nasprotnem primeru pa določi tistemu, ki je najmanj obremenjen. Prav tako razvrščevalnik poskuša določiti nit tistemu jedru, na katerem je predhodno že tekla. V registrih jedra so lahko shranjeni podatki, ki jih nit potrebuje. Na tak način pohitri izvajanje, saj ni potrebe po prenašanju podatkov iz glavnega pomnilnika. Ko niti določimo, na katerem jedru naj se izvaja, s tem onemogočimo možnost zamenjave jedra ob obremenitvi. Izvajanje tako poteka dalj časa. Pri številu niti, ki je manjše ali enako od števila jeder, lahko pričakujemo, da bo vsaka nit tekla na svojem jedru.



Slika 5.2: Graf hitrosti izvajanja aplikacije v C++ pri razvrščanju in določenem fiksnemu jedru niti.

Da bi preverili ali razvrščevalnik optimalno razporeja niti, smo aplikacijo v C++ prilagodili tako, da smo vsaki posamezni niti določili, na katerem jedru naj se izvaja. Teste smo izvajali nad tabelo s 60 milijoni naključnih

števil in različnem številu niti. Iz rezultatov (prikazani na sliki 5.2) smo opazili, da je bil potreben čas za dokončanje ukaza med 300 in 400 ms daljši pri aplikaciji z nitmi določenimi na posameznem jedru kot v primerjavi z aplikacijo, pri kateri je operacijski sistem Windows sam razvrščal niti med jedri.

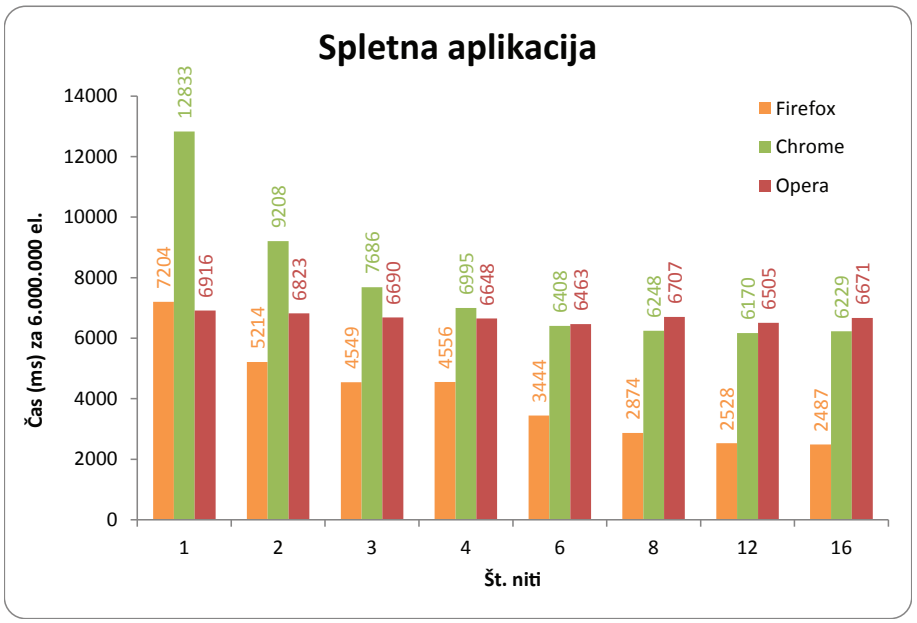
Teste smo izvajali tako na aplikaciji v C++ kot tudi na spletni aplikaciji pri enakih pogojih. Spletno aplikacijo smo pognali na različnih spletnih brskalnikih. Posamezen test smo opravili večkrat, končno hitrost izvajanja pa smo izračunali iz povprečne vrednosti vseh testov. Da bi zagotovili čimbolj natančne rezultate smo med testi prekinili vse nepotrebne procese in pustili le tiste najnujnejše.

Kot dodatni test smo tabelo napolnili le s 600 številkami ter test izvajali pod enakimi pogoji kot prejšnjega, toda le v spletni aplikaciji. Aplikacija v C++ se je izvajala prehitro, zato ni bilo mogoče izmeriti točnega časa. Namen tega testa je bil, ugotoviti koliko časa porabi spletna aplikacija za ustvarjanje niti.

5.3.3 Rezultati testiranja

Teste smo pognali na različnih brskalnikih. Internet Explorer nima podpore Web Workerjev, zato smo ga iz testov izpustili. Preverili smo le ali testna stran pravilno deluje in onemogoči nadaljnje teste. Iz slike 5.3 lahko vidimo, da smo dobili zelo različne rezultate. Najbolje se je odrezal brskalnik Mozilla Firefox, saj je bil najhitrejši. Opera je bila sicer v testu z eno nitjo sicer hitrejša od Mozille Firefox, toda je že pri dveh nitih ostala v ozadju. Pri Operi je bilo presenetljivo tudi to, da ni bilo opaznejše pohitritve ob povečevanju niti. Google Chrome sicer ima pohitritve pri povečevanju niti, toda še vedno je veliko počasnejši od Mozille Firefox. Pri številu niti večjem od števila jeder se pohitritev ustali oziroma se začne zmanjševati.

Če vzamemo za primer Mozillo Fireox kot najhitrejši brskalnik in primerjamo izvajanje algoritma na spletni in aplikaciji v C++ lahko opazimo, da je spletna aplikacija pri eni niti več kot tridesetkrat počasnejša od aplikacije

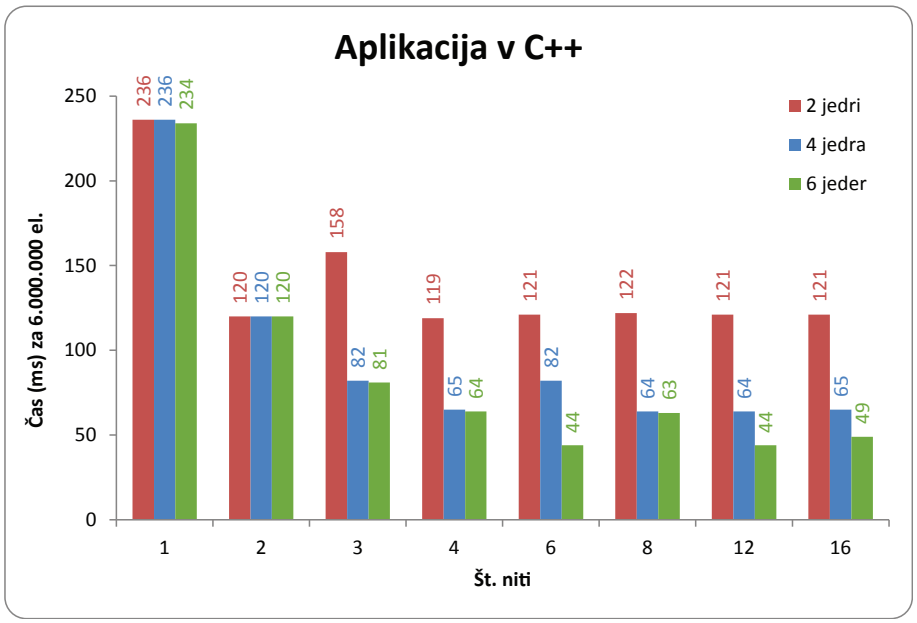


Slika 5.3: Graf hitrosti izvajanja spletne aplikacije na različnih brskalnikih. Hitrost se povečuje s številom niti.

v C++. S povečevanjem števila niti se ta razlika še povečuje. Če lahko pri aplikaciji v C++ rečemo, da se hitrost skoraj konstantno povečuje, tega ne moremo trditi za spletno aplikacijo. Pri spletni aplikaciji šele pri šestih nitih dosežemo dvakratno pohitritev. Toda pri aplikaciji v C++ se pohitritve bolj ali manj ustalijo pri šestih nitih za razliko od spletne, kjer so pohitritve opazne tudi pri večjem številu niti.

Seveda pa so pohitritve odvisne od števila jeder. Če se pri šestjedrnem procesorju pohitritev ustali pri šestih nitih, potem se pri štirijedrnem ustali že pri štirih nitih. Iz grafa na sliki 5.4 lahko vidimo, kakšne so pohitritve pri različnem številu jeder. Pri dvojedrnem procesorju ukaz porabi približno 50% manj časa, pri štirijedrnem 70% in pri šestjedrnem 80%.

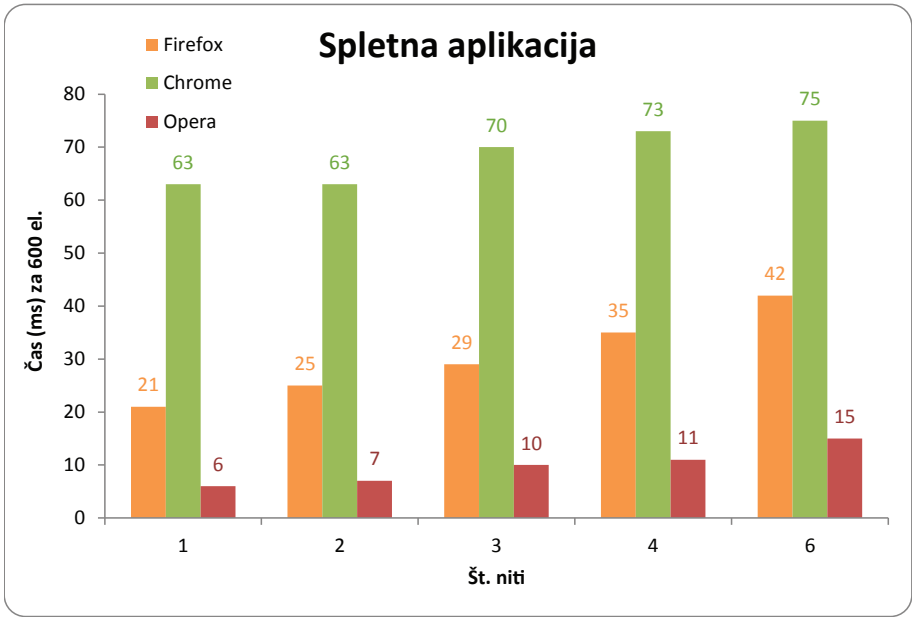
Razlike v hitrosti so predvsem posledica programskega jezika in načina ustvarjanja novih niti. JavaScript je skriptni jezik, ki se sproti interpretira in izvaja, za razliko od programskega jezika C++, ki se mora pred izvajanjem prevesti v izvršljivo kodo. Vsi skriptni jeziki so zaradi prej omenjenega



Slika 5.4: Graf hitrosti izvajanja namizne aplikacije. Več kot je jeder, večja je pohitritev.

načina izvajanja počasnejši. Aplikacija v C++ za ustvarjanje niti pošlje ukaz operacijskemu sistemu, kar je zelo hitro. Spletna aplikacija pa mora najprej ukaz poslati spletnemu brskalniku, nato pa spletni brskalnik ustvari nove niti.

Iz dodatnega testa (slika 5.5) pri majhnem številu elementov v tabeli je razvidno, da je ustvarjenje novih Web Workerjev zelo potratna operacija. Pri nekaterih brskalnikih bolj pri drugih manj. Ugotovimo pa lahko, da ustvarjanje novih Web Workerjev ni ključnega pomena pri hitrosti izvajanja celotnega algoritma. Mozilla Firefox potrebuje zelo veliko časa, da ustvari nov Web Worker, toda še vedno potrebuje manj časa za dokončanje izvajanja kot Google Chrome. Opera je v tem testu nesporni zmagovalec, saj je od obeh veliko hitrejša, tako po kreiranju Web Workerjev kot celotnem izvajanju.



Slika 5.5: Graf hitrosti izvajanja spletne aplikacije na različnih brskalnikih pri majhni tabeli (manjše je boljše).

5.3.4 Testiranje detekcije trkov

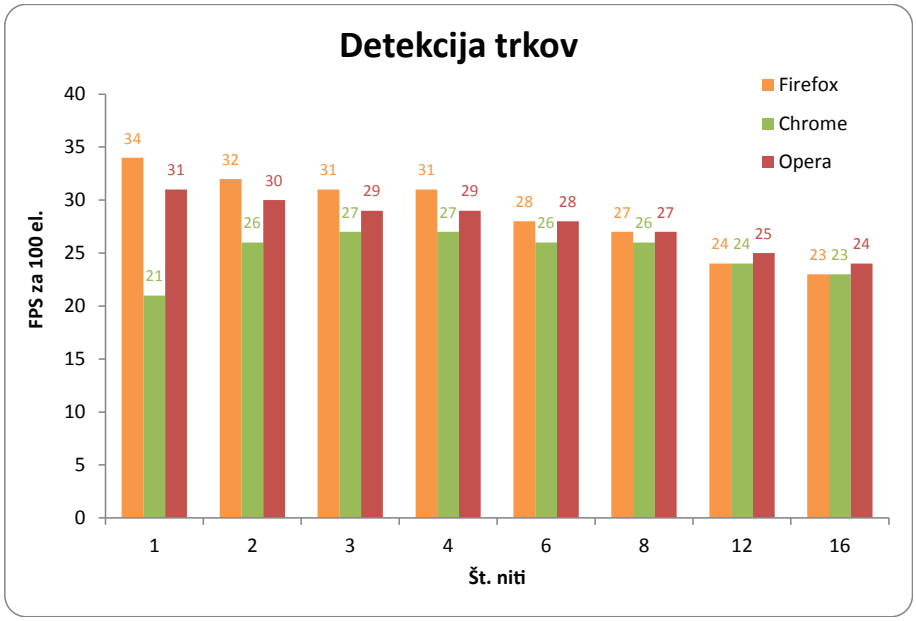
Za detekcijo trkov smo na spletno stran dodali vnosno polje za vnos števila elementov v tabelo, izbiro za število niti in polje canvas za izris animacije. Pred začetkom izvajanja testiranja se tabela napolni z objekti (žogice), ki hranijo vrednosti kot so položaj, velikost, hitrost itd. Ker smo pri testu merili število sličic na sekundo, se pravi število obhodov čez zanko, smo Web Workerje ustvarili na začetku in pri vsakem obhodu le poslali nove podatke Web Workerju za izračun novih vrednosti.

Algoritem je ob vsakem obhodu skozi zanko preveril, ali se je zgodil trk, preračunal položaj objektov in nato vse objekte izrisal. Zaznavanje trkov se je preverjalo le med stenami risalne površine canvas, saj drugače ne bi bilo mogoče algoritma paralelizirati na preprost način. Algoritem je najprej tabelo razdelil na toliko delov, kolikor je bilo niti. Vsaka nit je nato preverila svoj del objektov, če se je pri katerem zgodil trk in mu preračunala nove

koordinate. Ko so niti zaključile s preračunavanjem, so se vsi podatki ponovno združili v skupno tabelo. Po združitvi je skripta spletne strani izrisala objekte na risalni površini.

Teste smo večkrat ponovili pri različnem številu elementov in niti. Pri testu smo merili število sličic na sekundo. Glavni namen testa je bil uporaba niti v primeru, kjer se morajo ukazi izvajati v realnem času.

5.3.5 Rezultati testiranja detekcije trkov



Slika 5.6: Graf sličic na sekundo pri detekciji trkov (več je boljše). Hitrost pada zaradi komunikacije med nitmi.

Iz grafa na sliki 5.6 je razvidno, da pri večjem številu niti pada število sličic na sekundo. Pri testih z večjim številom objektov ni večje razlike in je izvajanje počasnejše. Pri zelo velikem številu objektov se število sličic na sekundo ustali in ni odvisno od števila niti. Brskalniki so si po hitrosti izvajanja podobni. Izstopa le Google Chrome, ki je pri eni niti veliko počasnejši od ostalih dveh. Toda rezultati se pri večjem številu niti izenačijo.

Kot smo že v prejšnjih poglavjih omenili, Web Workerji nimajo dostopa do DOM zaradi izrisovanja objektov iz glavne skripte strani. Web Workerji zgolj preračunajo nove pozicije objektov. Kot dodaten test smo pognali detekcijo trkov brez animacije in dobili smo trikratne pohitritve. Iz testov je razvidno, da je izrisovanje objektov v elementu canvas veliko bolj potratna operacija, kot pa preverjanje trkov in izračuni novega položaja objekta.

Podatki se pred prenašanjem iz glavne skripte strani na Web Worker prekopirajo, kar zahteva svoj čas. Pri velikih tabelah podatkov kopiranje in prenašanje podatkov porabi toliko časa, da se izniči pohitritev, ki jo pridobimo s paralelnim izvajanjem.

5.4 Ugotovitve

Testi nam pokažejo, da je ustvarjanje Web Workerja zelo potratna operacija, prav tako pošiljanje podatkov, zato je smiselno paralelizirati algoritem takrat, ko gre za preračunavanje velikih količin podatkov. Uporaba Web Workerjev je smiselna tudi takrat, ko želimo ohraniti odzivnost uporabniškega vmesnika pri daljših operacijah, ki se izvajajo v ozadju.

Tehnologija paralelnega procesiranja je še v razvoju, kar pomeni da ni še pripravljena na masovno uporabo. Velikokrat se pojavljajo anomalije pri času izvajanja, saj je čas izvajanja veliko daljši od pričakovanega. Sprva smo mislili, da operacijski sistem slabo razporeja niti med jedra procesorja, toda takšne anomalije se niso pojavile pri aplikaciji v C++. Po vsej verjetnosti gre za slabo optimizacijo spletnega brskalnika.

Ob primerjavi brskalnikov lahko vidimo velika odstopanja pri rezultatih testiranja. Še najbolje se je odrezal Mozillin brskalnik, pri ostalih dveh pa se pojavljajo napake oziroma počasno delovanje. Razvijalci spletnih brskalnikov se trudijo implementirati čim več novih funkcionalnosti, pri čemer pa še ne dajejo veliko poudarka na kakovost. Veliko stvari je še potrebno izpiliti in nadgraditi, predvsem pa dobro optimizirati za čim hitrejše delovanje.

Po drugi strani pa je lahko problem tudi pri specifikaciji novih funkci-

onalnosti, saj so še v razvoju, zato se zdi čakanje z implementacijo novih funkcionalnosti v spletnem brskalniku Internet Explorer logična. Microsoft vgradi nove funkcionalnosti takrat, ko so specifikacije končane, vendar pa mu bo morebitna porast spletnih strani, ki uporabljajo nove tehnologije delala težave.

Poglavje 6

Zaključek

Diplomska naloga opisuje paralelno izvajanje programov v spletnih brskalnikih in se osredotoči na izdelavo testne strani za preverjanje podpore paralelizaciji. Paralelno izvajanje algoritmov lahko dosežemo z uporabo Web Workerjev, ki nam omogočajo večnitno izvajanje skript v JavaScriptu. S testno stranjo je mogoče preveriti, kako dobro spletni brskalnik podpira paralelno izvajanje. Testna stran pa omogoča tudi nadaljnje spremljanje sprememb v razvoju na tem področju.

Da smo lahko izdelali takšno testno stran, smo se morali najprej dodobra seznaniti s paralelnim procesiranjem in najnovejšimi spletnimi tehnologijami. Znanja iz paralelnega procesiranja so nam prišla prav pri razumevanju paralelizacije algoritmov. Nove spletne tehnologije kot na primer HTML5 pa so nam omogočile implementacijo algoritmov na spletni strani.

Testno stran bi lahko izboljšali tako, da bi dodali teste funkcionalnosti, ki jih bodo prinesli novi spletni brskalniki. Prav tako bi lahko nadgradili trenutne teste tako, da bi dodali možnost shranjevanja in primerjanja rezultatov iz različnih spletnih brskalnikov.

Naš glavni namen je bil postavitev delujoče testne strani in opraviti različne teste, s katerimi smo ugotovili, kakšno je trenutno stanje na področju paralelnega izvajanja v spletnih aplikacijah. Podpora paralelnemu izvajanju s pomočjo Web Workerjev je prisotna v večini najbolj razširjenih

spletnih brskalnikov, vendar pa se po učinkovitosti implementacije precej razlikujejo. Samo ustvarjenje Web Workerjev časovno precej zahtevno, zato moramo dobro premisliti, kdaj jih uporabimo. V primerjavi s programi v C++ so večnitni programi v JavaScriptu precej počasnejši, vendar imamo tukaj primerjavo med programi v C++, ki so prevedeni in programi v JavaScriptu, ki se interpretirajo, zato je razlika pričakovana.

Slike

Literatura

[1] (2011) B. Blaise. Introduction to Parallel Computing. Dostopno na: https://computing.llnl.gov/tutorials/parallel_comp

[2] R. Buyya. *Parallel Computing at a Glance*. Dostopno na: http://www.buyya.com/microkernel/chap1.pdf, 2000.

[3] J. Cranford Teague. *Visual QuickStart Guide CSS3*. Peachpit Press, 2011.

[4] J. L. Hennessy, D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 3. izdaja. 2002, str. 43.

[5] G. Jakus, M. Jekovec, S. Tomažič, J. Sodnik. "New technologies for web development", *Elektrotehniški vestnik 77(5)*, 2010, str. 273–280.

[6] P. Lubbers, B. Albers, F. Salim. *Pro HTML5 Programming*. Apress, 2010.

[7] J. M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 1996, str. 235.

[8] D. B. Probert. *Thread Scheduling*. Dostopno na: http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new–documents/Lectures/03–ThreadScheduling/ThreadScheduling.pdf, 2011.

[9] W. Stallings. *Operating Systems: Internals and Design Principles*, 6. izdaja. Prentice Hall, 2008, str. 160–175.

[10] M. Vaidehi, T. R. Gopalakrishnan Nair. *Multicore Applications in Real Time Systems*. Dostopno na: http://arxiv.org/PS_cache/arxiv/pdf/1001/1001.3539v1.pdf, 2010, str. 30–35.

[11] N. C. Zakas. *Professional JavaScript for Web Developers*. Wiley Publishing Inc., 2005, str. 1–9.

[12] (2011) Parallel computing. Dostopno na: http://en.wikipedia.org/wiki/Parallel_computing.

[13] (2011) Distributed shared memory. Dostopno na: http://en.wikipedia.org/wiki/Distributed_shared_memory.

[14] (2011) Amdahl's law. Dostopno na: http://en.wikipedia.org/wiki/Amdahl"s_law.

[15] (2011) Gustafson's law. Dostopno na: http://en.wikipedia.org/wiki/Gustafson"s_Law.

[16] (2011) Thread. Dostopno na: http://en.wikipedia.org/wiki/Thread_(computer_science).

[17] (2011) What makes parallel programming hard?. Dostopno na: http://www.futurechips.org/tips–for–power–coders/parallel–programming.html.

[18] (2011) MoonBat JavaScript Benchmark – Unofficial Web Worker Sample. Dostopno na: http://www.yafla.com/dforbes/resources/moonbat/moonbat–driver.html.

[19] (2011) Javascript Web Workers test. Dostopno na: http://pmav.eu/stuff/javascript–webworkers/.

[20] (2011) HTML5 test. Dostopno na: http://html5test.com/.