UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matija Cankar

Povečevanje učinkovitosti izvajanja nalog s sočasnim delnim dodeljevanjem virov v rahlo sklopljenih računalniških strukturah

DOKTORSKA DISERTACIJA

Mentor: prof. dr. Uroš Lotrič

Ljubljana, 2014

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matija Cankar

Povečevanje učinkovitosti izvajanja nalog s sočasnim delnim dodeljevanjem virov v rahlo sklopljenih računalniških strukturah

DOKTORSKA DISERTACIJA

Mentor: prof. dr. Uroš Lotrič

Ljubljana, 2014

IZJAVA O AVTORSTVU

doktorskega dela

Podpisani Matija Cankar,

z vpisno številko 63020016,

sem avtor doktorskega dela z naslovom:

Povečevanje učinkovitosti izvajanja nalog s sočasnim delnim dodeljevanjem virov v rahlo sklopljenih računalniških strukturah

S svojim podpisom zagotavljam, da:

- sem doktorsko delo izdelal samostojno pod mentorstvom
  izr. prof. dr. Uroša Lotriča,
- so elektronska oblika doktorskega dela, naslov (slov., angl.), povzetek (slov., angl.) in ključne besede (slov., angl.) identični s tiskano obliko doktorskega dela,
- soglašam z javno objavo elektronske oblike doktorskega dela v zbirki "Dela FRI".

V Ljubljani, dne 9. 9. 2014 Podpis avtorja/-ice:

Zahvala

Za mentorstvo se zahvaljujem izr. prof. dr. Urošu Lotriču, ki me je usmerjal pri raziskovanju in mi bil v pomoč pri nastajanju doktorskega dela, ter doc. dr. Boštjanu Slivniku, ki je mentorsko delo opravljal, še preden sem se odločil za prehod na doktorski študij.

Zahvaljujem se tudi sodelavcem in raziskovalni skupini iz podjetja XLAB, zlasti dr. Mateju Artaču in dr. Marjanu Šterku za nasvete pri raziskovalnih in predstavitvenih zagatah ter dr. Gregorju Pipanu za vso podporo pri izvajanju programa mladega raziskovalca v gospodarstvu. V XLAB-u sem izkusil tudi kakovostno raziskovalno ozračje v skupini mladih raziskovalcev in njihovih mentorjev. Med nastajanjem tega dela smo imeli veliko sestankov in predstavitev, na katerih smo reševali vsebinske in tudi manj vsebinske težave na svojih poteh do cilja. V tej ekipi so vsaj nekaj časa sodelovali mag. Aleš Černivec, dr. Bojan Blažica, dr. Rok Orel, Manja Gorenc Novak, Jolanda Modic, Robert Dukarić, Špela Jezernik-Širca, Boris Cergol, Saba Resnik in Polona Tomašič ter mentorja dr. Daniel Vladušič in mag. Uroš Jovanovič. Pri urejanju birokratskih zadev sem se lahko vedno zanesel na Darjo Peterlin in Mersiho Zajec.

Za potrpljenje in moralno podporo se zahvaljujem ženi Ani in preostali družini ter prijateljem, ki so mi vedno stali ob strani.

Zahvaljujem se tudi Evropski uniji, in sicer Evropskemu socialnemu skladu, ki je delno sofinanciral nastajanje doktorskega dela.

Družini in prijateljem

Kazalo

Povzetek **3**

Abstract **5**

1 Uvod **7**
1.1 Motivacija in ozadje 7
1.2 Opis ožjega področja in stanja tehnologije 9
1.3 Osnovni hipotezi 10
1.4 Potek disertacije 11

2 Pregled področja **13**
2.1 Vir v računalniškem sistemu 14
2.2 Porazdeljene računske infrastrukture 14
2.2.1 Superračunalniki, gruče in definicija porazdeljenega računalništva 14
2.2.2 Omrežno računalništvo 15
2.2.3 Prostovoljno računalništvo in sistemi enak z enakim 16
2.2.4 Računalništvo v oblaku 17
2.2.5 Virtualizacija infrastrukture 18
2.3 Lastnosti opravil 19
2.3.1 Toga opravila 19
2.3.2 Fleksibilna opravila 20
2.4 Pristopi k razvrščanju in upravljanju z viri 21
2.4.1 Pogosti pristopi k razvrščanju 21
2.4.2 Razvrščanje z vrstami 22
2.4.3 Razvrščanje z rezervacijami 23
2.4.4 Organizacija infrastrukture 24
2.4.5 Dodeljevanje vozlišč 24

2.5 Upravljalniki z viri in razvrščevalniki 25
2.5.1 Paketni razvrščevalniki 25
2.5.2 Upravljalniki oblakov 26
2.5.3 Porazdeljeni upravitelji virov 30
2.5.4 Metarazvrščanje 31
2.6 Simulatorji upravljanja virov 32
2.6.1 Haizea 32
2.6.2 GridSim, CloudSim in EMUSIM 32
2.7 Ocenjevanje razvrščanja 33
2.8 Zaključek 35

3 Sočasno dodeljevanje v omrežnem računalništvu 37
3.1 Motivacija 38
3.2 Definicija vira in sočasno dodeljevanje vozlišč 38
3.3 Algoritem za sočasno dodeljevanje infrastrukture 40
3.3.1 Faza I – iskanje vozlišč 40
3.3.2 Faza II – preverjanje vozlišč 41
3.3.3 Faza III – iskanje množice potencialnih začetnih časov . . . 41
3.3.4 Faza IV – sestavljanje dopustne rešitve 42
3.3.5 Faza V – optimizacija rešitve 45
3.3.6 Faza VI – rezervacija virov na vozliščih 45
3.4 Poskusi in rezultati 46
3.4.1 Implementacija 46
3.4.2 Podatki za testiranje 46
3.4.3 Parametri algoritma 47
3.4.4 Kakovost rešitve 50
3.4.5 Časovna zahtevnost 51
3.4.6 Režijski stroški operacijskega sistema in infrastrukture . . . 52
3.5 Zaključek 53

4 Sobivanje in vertikalno skaliranje 55
4.1 Motivacija 55
4.2 Razvrščanje opravil in izkoristek virov 58
4.3 Združevanje in skaliranje opravil 60
4.3.1 Iskanje komplementarnih opravil 61
4.3.2 Združevanje popolnoma komplementarnih opravil 62
4.3.3 Združevanje delno komplementarnih opravil 64
4.3.4 Skaliranje preostalih opravil 65
4.4 Priprava opravil za razvrščevalnik 65

4.5 Poskusi in rezultati 66
4.5.1 Implementacija 67
4.5.2 Množice testnih opravil 67
4.5.3 Izbira parametrov algoritma 69
4.5.4 Ocena uspešnosti 69
4.5.5 Poraba časa in časovna kompleksnost združevanja 73
4.6 Zaključek 75

5 Zaključek **77**
5.1 Prispevki k znanosti 78
5.2 Smernice in nadaljnje delo 79

Seznam slik **80**
Seznam preglednic **82**

Literatura **84**

Dodatki **97**

A Oznake v poglavjih 3 in 4 **99**
A.1 Oznake v poglavju 3 99
A.2 Oznake v poglavju 4 100

Povzetek

Področje porazdeljenih računalniških infrastruktur v računalništvu ni novost, a v industriji in akademskih krogih še vedno žanje veliko zanimanja. Močnejši računalniki, boljša omreženost in hitrejše povezave ter zahtevna porazdeljena opravila pospešujejo razcvet porazdeljenega računalništva. Veliko število računalnikov, povezanih v eno omrežje, ponuja uporabnikom dodatno računsko moč, kadarkoli jo potrebujejo. Toda tak sistem ni poceni in njegovo vzdrževanje ni preprosto, zato so lahko stroški z njim neupravičeno visoki, če infrastruktura ni učinkovito izkoriščena.

Trenutno najzanimivejši obliki porazdeljenih sistemov sta omrežno računalništvo in računalništvo v oblaku. V tej doktorski disertaciji pod drobnogled vzamemo pogoste načine upravljanja z viri v obeh omenjenih oblikah. Proučimo pristope razvrščanja na omrežjih s porazdeljenim in centralnim upravljanjem infrastrukture ter izpostavimo ključne lastnosti opravil, ki se pogosto izvajajo na porazdeljeni infrastrukturi.

V doktorskem delu predstavimo prednosti razvrščanja opravil, ki lahko prilagodijo svoje delovanje količini dodeljenih virov, in predlagamo dva pristopa razvrščanja. Prvi omogoča sočasno dodeljevanje virov opravilom v porazdeljeni infrastrukturi s porazdeljenim upravljanjem z viri, kar pomeni, da hkrati teče več avtonomnih razvrščevalnikov, ki nimajo globalnega pogleda na stanje vseh virov na vozliščih. Pri tem pristopu se omejimo na razvrščevalnike, ki v danem trenutku v sistem umeščajo zadnje prispelo opravilo. Predlagani pristop podpira kolektivne zahteve, to so zahteve po množici vozlišč, ki morajo kot celota ugoditi tem zahtevam. Pristop smo implementirali za sistem XtreemOS ter preskusili njegovo delovanje v realnem in umetnem okolju. Rezultati potrjujejo, da s pristopom računsko infrastrukturo obremenimo varčneje, hkrati pa se opravila začnejo izvajati ob zgodnejšem času. Cena izboljšav je nekoliko daljše trajanje iskanja primernih vozlišč.

Drugi pristop predvideva odloženo dodeljevanje virov opravilom v porazdeljeni infrastrukturi s centralnim upravljanjem z viri, kar pomeni, da imamo le en razvr-

ščevalnik, ki v danem trenutku umešča več opravil sočasno. Predlagali smo preiskovanje opravil v paketih in hkratno prilagajanje opravil po virih tako, da izboljšamo njihovo sobivanje in posledično učinkovitost izrabe vozlišč. Predlagani pristop smo implementirali tako, da deluje skupaj z razvrščevalnikom Haizea, in preskusili delovanje. Rezultati potrjujejo, da lahko s prilagajanjem manjše množice opravil izboljšamo učinkovitost izrabe celotne računske infrastrukture, saj prihranki pri tem več kot odtehtajo dodatno delo prilagajanja opravil.

Predlagana pristopa razvrščevalnikom omogočata boljši izkoristek infrastrukture in večjo verjetnost, da bo razvrščevalnik našel primerne vire za opravilo. Z opisi pristopov in poskusi, predstavljenimi v tem delu, prispevamo k oblikovanju novih rešitev za razvrščevalnike na področju omrežnega in oblačnega računalništva, v zaključkih pa navajamo tudi nekatere mogoče razširitve.

Ključne besede:

Porazdeljeni sistemi, upravljanje z viri, razvrščanje, omrežno računalništvo, računalništvo v oblaku.

Abstract

The field of distributed computer systems, while not new in computer science, is still the subject of a lot of interest in both industry and academia. More powerful computers, faster and more ubiquitous networks, and complex distributed applications are accelerating the growth of distributed computing. Large numbers of computers interconnected in a single network provide additional computing power to users whenever required. Such systems are, however, expensive and complex to manage, which can lead to unduly high expenses unless the infrastructure is efficiently utilised.

Currently the most attractive forms of distributed systems are grid and cloud computing. In this dissertation we review some of the resource management approaches commonly used in grid and cloud computing. We examine scheduling approaches in systems with distributed and centralised infrastructure management and highlight the key properties of the applications for which distributed infrastructures are typically used.

We present the advantages of scheduling flexible jobs which can scale themselves to the amount of allocated resources, and propose two scheduling approaches. The first approach supports co-allocation of computer resources to jobs on distributed infrastructures with distributed resource management. The latter implies that the system can use multiple autonomous schedulers, which do not have global control over the state of the resources on the nodes. We focus on schedulers that only map a single job to the infrastructure at a time. We propose an approach that supports collective demands, i.e. requests for a set of nodes that must collectively meet the specified demands for resources. We implemented this approach in the XtreemOS operating system and evaluated it in real and simulated environments. The results show that the use of collective demands extends search times, but this is compensated by the fact that the scheduled jobs load the infrastructure more sparingly and allow the jobs to start earlier.

The second approach is applicable to offline resource scheduling in distributed

infrastructures with global control over the resources. In other words, there is a single central scheduler that can schedule a whole set of jobs simultaneously. For such a set-up we propose analysing the jobs in a batch in order to pair and scale them into co-located subsets and thus improve utilisation. We implemented the proposed approach to run with the Haizea scheduler and evaluated its activity. The results show that the adjusting of a small job subset improves the utilisation of the infrastructure and the savings obtained more than outweigh the extra work needed for the adjusting.

The proposed approaches allow the schedulers to better utilise the infrastructure and increase the likelihood of finding the appropriate resources for the job. Through the approaches described and experiments presented, we contribute to the formulation of new solutions for schedulers in the fields of grid and cloud computing. Some possible extensions are given in the conclusions.

Keywords:

Distributed systems, resource management, scheduling, grid computing, cloud computing.

Poglavje 1

Uvod

V doktorski disertaciji smo se lotili problema povečevanja učinkovitosti izvajanja nalog s sočasnim delnim dodeljevanjem virov v rahlo sklopljenih računalniških strukturah. V uvodnem poglavju bomo predstavili motivacijo s splošnimi definicijami učinkovitosti in značilen pogled na računalniške infrastrukture današnjega časa. Natančneje bomo definirali področje raziskovanja, stanje tehnologij in hipotezi. Nazadnje bomo v uvodnem poglavju predstavili strukturo doktorske disertacije.

1.1 Motivacija in ozadje

Učinkovitost je lastnost, ki si je želi vsak posameznik ali gospodarska družba, saj zagotavlja, da bodo cilji doseženi z manj dela ali manjšimi sredstvi. Ljudje opazujemo svet okoli sebe ter z znanjem, izkušnjami, ustvarjalnostjo, raziskovanjem in razvojem izdelujemo nova orodja in postopke, ki izboljšujejo učinkovitost. Pomnjenje in obdelovanje podatkov v naših glavah nista najučinkovitejši, zato smo že kmalu začeli podatke zapisovati na različne nosilce in jih postopoma obdelovati. Pri tem si pomagamo tudi z računalniki. Učinkovitost računalnikov se povečuje s hitrejšim obdelovanjem podatkov ter boljšimi pristopi za njihov zajem, shranjevanje in prikaz. Zaradi vsesplošne uporabe računalniških virov in potreb po njihovi učinkoviti uporabi postavljamo predstavljeno tematiko v središče te disertacije.

Kaj je učinkovitost pri uporabi računskih virov, je stvar perspektive in namena uporabe. Najmočnejši računalniki na svetu, ki so na seznamu TOP500 [1], so učinkoviti zato, ker opravijo svoje delo hitreje kot njihovi počasnejši sodobniki in predhodniki. Hitrost je pomembna, saj nam omogoča, da lahko izvajamo natančnejše

simulacije časovno kritičnih pojavov, denimo napoved vremena. Po drugi strani so računalniki učinkovitejši, če glede na opravljeno delo porabijo manj energije. Seznam takih računalnikov je dostopen na seznamu Green500 [2]. Učinkovitost lahko merimo tudi s količino dela, ki jo računalniki opravijo v določenem času. Ogromno povezanih računalnikov, ki rešujejo isti cilj (npr. projekt SETI@home[3]), lahko opravi veliko dela, ampak niso ne hitri ne energijsko varčni. Poleg omenjenega so računalniki učinkoviti, če so dovolj majhni, da jih lahko skrijemo v kapsulo za endoskopijo, ali dobro povezani, da nam omogočijo komunikacijo po vsem svetu. Nenazadnje lahko učinkovitost določimo s finančno komponento. V poslovnem krogu so učinkoviti tisti računalniki, ki s svojim obratovanjem prinesejo čim več denarja. Delno je tako mišljenje tudi botrovalo razvoju računalništva v oblaku. Ker bomo v tem delu obravnavali učinkovitost rabe računalnikov, povezanih v isto omrežje, bomo učinkovitost definirali bodisi kot večjo količino opravljenega dela ali kot manjše število obremenjenih računalnikov.

Pristop k uravnavanju učinkovitosti rabe je odvisen tudi od tega, kje se nahajajo računski viri in kako jih lahko zaposlimo. Razmestitev računalniških virov in njihova uporaba se skozi čas spreminjata, toda na nek način se trendi tudi v zgodovini računalništva ponavljajo. V sredini prejšnjega stoletja so bili viri centralizirani na posameznih lokacijah, na katerih so bili postavljeni superračunalniki (angl. *supercomputers*) in osrednji računalniki (angl. *mainframe computers*). Že takrat je John McCarthy predvideval uporabo računskih virov, podobno kot porabljamo javne vire (voda in elektrika). V 1960-ih so proizvajalci osrednjih računalnikov pripravili storitve za upravno poslovanje, ki so temeljile na dodeljevanju časa (angl. *time sharing*) in storitvenem računalništvu (angl. *utility computing*). Te storitve so visoko učinkovitost rabe omogočile s ponujanjem računskih virov in podatkovne hrambe iz centrov, ki so bili postavljeni po vsem svetu, za banke in druge velike organizacije.

Pozneje se je začela doba osebnih računalnikov in del računalniških virov se je približal ljudem ter porazdelil po svetu. Tedaj je bil centralni model manj primeren, saj domači računalniki niso bili povezani v omrežje in je bila učinkovitost rabe skrb posameznika. V prvem desetletju 21. stoletja, ko je povezanost računalnikov v medmrežje postala nepogrešljiva, je McCarthyjeva ideja zopet postala priljubljena. Paradigmi omrežno računalništvo (angl. *grid computing* [4, 5, 6]) in računalništvo v oblaku (angl. *cloud computing* [7, 8]) sta vire centralizirali ob pomoči večjih računskih centrov ter začeli ponujati vire na zahtevo uporabnikov. Na pogled se zdi, kot da se trend vrača v centraliziranje, in hkrati je jasno, da se bo število naprav, ki bodo vire potrebovale, povečalo. Delno to nakazujeta novi paradigmi, ki napovedujeta nadaljnjo centralizacijo računalniških virov (računalništvo na nebu, angl. *sky computing*) ter povezovanje več naprav in elementov v koncept računalniške obdelave in povezovanja (internet stvari - IoT, angl. *Internet of Things*). Ne bomo

se čudili, če bodo osebni računalniki kmalu zapustili naše domove. Namesto njih bomo uporabljali drugačne računalnike v obliki pametnih naprav, kot so telefoni, televizorji, ure in tablice, ki jim bodo dodatne računske vire zagotovili veliki računski centri v obliki storitev.

Porast uporabe računalnikov in pestrost opravil, ki jih izvajajo, omogočata dodatno optimizacijo uporabe. Še največ koristi pri optimizaciji rabe omogočajo opravila, ki nimajo strogih omejitev glede časa izvajanja in želenih računskih virov, na katerih se izvajajo. Tako imamo na eni strani pestra opravila, ki potrebujejo vire, in na drugi strani dostop do velike količine računskih virov, povezanih v omrežje. Ker opravila virov na vozliščih ne porabijo vedno v celoti, je naravno nadaljevanje optimizacije upravljanja z računskimi viri delno dodeljevanje infrastrukture, ki v zadnjem času postaja vse bolj ključno in ga podrobneje obravnavamo v nadaljevanju.

1.2 Opis ožjega področja in stanja tehnologije

Ta doktorska disertacija spada v širše področje računalniških znanosti, v področje porazdeljenih sistemov in vzporednega računanja, natančneje v področje upravljanja in uporabe računskih virov, ki jih zagotavlja računalniška infrastruktura. Osredotočamo se na učinkovito izrabo porazdeljenih rahlo sklopljenih računalniških struktur pri posebnih oblikah omrežnega računalništva in računalništva v oblaku ter na opravila, ki jih je na takem sistemu mogoče učinkovito izvajati. Prispevki tega doktorskega dela naslavljajo področja sočasnega in delnega dodeljevanja ter sobivanja in preoblikovanja opravil pred izvajanjem.

Stanje tehnologije

Raziskovalci v industriji in raziskovalnih ustanovah se dnevno srečujejo z zahtevnimi problemi, ki za hitro in učinkovito reševanje potrebujejo veliko računskih virov. Mnoge med njimi je mogoče dobro paralelizirati in s primernimi računalniškimi infrastrukturami pohitriti skladno z Gustafson-Barsisovim in Amdahlovim zakonom [9, 10]. Vzporedne probleme z visoko stopnjo usklajevanja lahko rešujemo s knjižnico MPI (angl. *message passing interface*). Številne probleme, ki med izvajanjem ne potrebujejo neprestanega usklajevanja in jih lahko poljubno razdelimo, pa lahko dobro rešujemo na velikem številu različno zmogljivih virov. Primeri takih so *trivialno vzporedni problemi* (angl. *embarrassingly parallel*, tudi *pleasantly parallel*) [11], ki jih pogosto imenujemo tudi *vreča nalog* ali *vreča opravkov* (*BoT*, angl. *Bag-of-Tasks*), kamor sodijo vzporedne parametrične študije, preiskovanje podat-

kov pri strojnem učenju, široka preiskovanja[1], nekatere simulacije, računanje fraktalov in upodabljanje v računalniški grafiki. Nekatere te probleme je mogoče opisati s trenutno priljubljenim pristopom MapReduce [12].

Vzporedna in nevzporedna opravila za svoje izvajanje potrebujejo vire. Veliko količino virov je mogoče zagotoviti s povezanimi računalniškimi strukturami porazdeljenega računalništva, kot so omrežno računalništvo in računalništvo v oblaku, vendar potrebujemo aplikacije, ki te vire primerno dodelijo opravilom. Aplikacije za dodeljevanje virov se razlikujejo glede na raven porazdeljenosti virov in omejitve dodeljevanja. Za pravilno izvajanje paralelnih opravil potrebujemo sistem za sočasno dodeljevanje virov (angl. *resource co-allocation*), ki ga je mogoče implementirati z različnimi pristopi [13, 14, 15, 16, 17, 18, 19, 20, 21]. Sočasno dodeljevanje je primarno predmet omrežnega računalništva, ki večinoma dodeljuje vozlišča le v celoti. Posledično ti sistemi ne podpirajo delnega dodeljevanja vozlišč, kar bomo predstavili v poglavju 2.

V računalništvu v oblaku, ki ob pomoči virtualizacije omogoča delno dodeljevanje virov, se soočamo s problemom sobivanja opravil (angl. *co-location*) [22, 23]. Toda ker je računska infrastruktura vse pestrejša, je pomemben korak pri dodeljevanju virov upoštevanje fleksibilnosti opravil, ki se lahko izvajajo na različni konfiguraciji virov [24] ali ob različnem času. Prilagodljivost opravil lahko s pridom izkoristimo za prilagajanje dela na razpoložljive vire ter s tem za izboljšanje urnika izvajanja opravil in povečanje učinkovitosti rabe infrastrukture. Podrobneje bomo vse koncepte in tudi povezano terminologijo predstavili v Poglavju 2.

1.3 Osnovni hipotezi

V doktorskem delu smo se lotili izboljšanja učinkovite rabe računskih virov s sočasnim in delnim dodeljevanjem računskih virov opravilom. Glavni motiv tega dela je potrditev naslednjih hipotez:

1. Sočasno delno dodeljevanje virov ob pomoči zahtev, ki poleg omejitev posameznih vozlišč vsebujejo tudi omejitve za vsa vozlišča v celoti, izboljša učinkovitost izrabe infrastrukture in omogoča opravilom, da se začnejo prej.

2. Sobivanje opravil na vozliščih lahko izboljšamo s prilagajanjem količine virov, ki so na voljo opravilom, kar posledično izboljša učinkovitost izrabe infrastrukture.

[1] Sem sodijo potratni problemi, kot je ugibanje ključev in gesel pri testiranju varnosti sistemov ali rudarjenje denarne valute BitCoin.

Obema hipotezama so skupni zagotavljanje virov na infrastrukturi, zmožnost preoblikovanja opravil in omogočanje delnega zasedanja infrastrukture. Ključna razlika je, da se prva osredotoča na opisovanje množice vozlišč v zahtevi in tako išče najboljšo množico vozlišč za opravilo, druga pa se osredotoča na prilagajanje opravil, ki sobivajo na vozlišču.

1.4 Potek disertacije

V poglavju 2 nadaljujemo s predstavitvijo temeljev doktorskega dela in podamo pregled področja, v katerem opišemo organizacije računskih vozlišč, upravljanje z vozlišči, obliko in organizacijo opravil ter načine ocenjevanja učinkovitosti. Poglavje 3 predstavi delo, povezano z vsebino članka, objavljenega v reviji *Journal of Universal Computer Science* [25]. Osredotočimo se na zelo porazdeljene večuporabniške sisteme omrežnega računalništva in upravljanje z viri v takem sistemu. Zanima nas, ali v množici vozlišč lahko najdemo primerno podmnožico za svoje vzporedno opravilo tako, da čim manj razdrobimo infrastrukturo. Poglavje 4 v drugačnem okolju predstavi podoben koncept preoblikovanja opravil, kot ga spoznamo v poglavju 3. Okolje omogoča centralizirano poznavanje sistema v manjših podatkovnih centrih in uporablja virtualizacijo. Zanima nas, kako preoblikovati opravila, da jih bo razvrščevalnik lahko čim bolje postavil v urnik. Disertacijo zaključimo z razpravo in predlogi za nadaljnje delo v poglavju 5.

Poglavje 2

Pregled področja

Kakšna je razlika med omrežnim računalništvom in računalništvom v oblaku? Kako se razlikujejo vzporedna opravila? Je mogoče vsa opravila, namenjena porazdeljenemu računanju, poganjati na vseh oblikah porazdeljenega računalništva? Ta vprašanja se večkrat pojavijo med raziskovalci in odgovori se v nekaterih primerih zelo razlikujejo. Različni odgovori so posledica dejstva, da porazdeljeno računalništvo in njegovi inkarnaciji, omrežno računalništvo in računalništvo v oblaku, niso dokončno izoblikovani koncepti, saj zanje nimamo stroge definicije ali univerzalnih specifikacij oblike in namena. Vse oblike potrebujejo infrastrukturo in opravila, poleg tega pa še orodja za urejanje njihovega delovanja. Vsi trije gradniki so pri vsakem primerku porazdeljenega računalništva lahko zelo specifični. Zaradi razlik med infrastrukturami in opravili ne more prav vsako porazdeljeno opravilo teči na vsaki porazdeljeni infrastrukturi [26].

Naši prispevki k znanosti se osredotočajo na orodja za razvrščanje opravil. Za razvoj teh orodij je pomembno poznavanje vseh gradnikov in pristopov, ki vplivajo na delovanje porazdeljenega sistema. To poglavje bomo namenili pregledu področja, v katerem predstavimo gradnike in odpravimo dileme o tem, kaj v tem delu pomeni posamezen termin. Na začetku predstavimo arhitekture in način upravljanja ter nadaljujemo z lastnostmi in tipologijo opravil. Sledijo razvrščevalni pristopi in razvrščevalniki, poglavje pa zaključimo s simulatorji in metrikami za merjenje učinkovitosti.

2.1 Vir v računalniškem sistemu

Računalniški sistem, ki je lahko posamezen računalnik, gruča ali večji sistem povezanih računalnikov, je sestavljen iz komponent, ki jim rečemo viri. Uporabniki omenjenih virov so gostujoča opravila. Kaj točno je vir, je odvisno od posameznega računalniškega sistema, kot to opiše tudi [27]:

> Računalniški sistem je tako skupek povezanih virov, ki so tako organizirani, da rešujejo probleme na osnovi sodobne obdelave podatkov, informacij, znanj itd. Računalniški sistem je navzven, to je nasproti okolju uporabe, v katerem rešuje probleme, tudi vir[1].

Tako so viri v namiznem računalniku kapaciteta pomnilnika, zmogljivost CPE, pasovna širina mrežne povezave, medtem ko so viri v gruči celotna vozlišča, samostojne računske enote z vsemi komponentami. V tem doktorskem delu bomo z besedo vir označili vire znotraj računalnika, kot je denimo pomnilnik, medtem ko bomo za avtonomne računske enote, ki vsebujejo vse osnovne vire za izvajanje opravila, uporabljali izraz vozlišče ali pa navidezna naprava.

2.2 Porazdeljene računske infrastrukture

Računske in hrambne vire nam zagotavlja infrastruktura, kjer se opravi računsko ali podatkovno zahtevno delo. Ključno vodilo pri načrtovanju in nabavi infrastrukture je gradnja najprimernejšega sistema s čim manj stroški. V tem razdelku predstavimo gradnike in ideje, ki so skozi čas najbolj prispevali k razvoju porazdeljenih računalniških infrastruktur ter oblikovali prostovoljno, omrežno in oblačno računalništvo.

2.2.1 Superračunalniki, gruče in definicija porazdeljenega računalništva

Razvoj porazdeljenega računalništva se je začel s superračunalniki in gručami. Skozi čas se je način njihove gradnje in uporabe spremenil in s tem so se razvile tudi nove metode porazdeljenega računalništva. *Superračunalniki* so računalniki, ki jih v določenem času označimo kot najsodobnejše in najzmogljivejše računske stroje. Seznam najzmogljivejših 500 računalnikov se osvežuje dvakrat letno in je prosto dostopen [1]. Arhitektura superračunalnikov je od svojih začetkov v šestdesetih letih prejšnjega stoletja doživela veliko preobratov. Najznačilnejši je ta, da so prve

[1] Avtor v izvirniku uporablja besedo *resurs* namesto besede *vir*.

arhitekture imele majhno število namenskih procesorjev, povezanih z deljenim pomnilnikom. Problem omenjenega pristopa je, da sta zasnova in razvoj namenskih komponent za superračunalnike velik strošek[2], predvsem pa v določenih primerih traja tako dolgo, da se do konca izdelave popolnoma spremenita načrt in namen uporabe superračunalnika (npr. ILLIAC IV [28]). Tako se od devetdesetih let prejšnjega stoletja dalje superračunalnike sestavlja ob pomoči velikega števila standardnih procesorjev in visoko zmogljivih povezav med procesorji. Takim sistemom pravimo tesno (procesorji, povezani na skupno vodilo) ali rahlo (procesorji, povezani z lokalno mrežo z visoko zmogljivostjo) povezane *gruče* (angl. *tightly and loosely coupled cluster*). Prednosti gruč sta nižja začetna investicija in nižji stroški vzdrževanja, saj se gradniki proizvajajo masovno ter se uporabljajo tudi za strežnike, delovne postaje in osebne računalnike. Lažja je tudi nadgradnja gruče z dodajanjem novih elementov.

Pri problemih, pri katerih usklajevanje ni tako pomembno, lahko poljubne računalnike in gruče povezujemo na različnih lokacijah. Omenjeni pristop omogoča lažje reševanje infrastrukturnih problemov napajanja in hlajenja sistema ter v nekaterih primerih tudi olajša vzdrževanje. Po drugi strani imamo zaradi takšne *heterogene* infrastrukture dodatno delo z usklajevanjem različnih vrst komponent.

Termin *porazdeljeno računalništvo* (angl. *distributed computing*) v grobem uporabljamo za tesno in rahlo povezane računalnike, ki jih uporabljamo za vzporedno, porazdeljeno in sočasno računanje (angl. *parallel, distributed, concurrent*). Področja omenjenih izrazov se prekrivajo, zato jih ne bomo strogo omejevali. V tem delu bomo termin *porazdeljeno računalništvo* uporabljali za *povezane računalnike, ki služijo skupnemu cilju*.

V nadaljevanju predstavimo za svoje delo zanimiva podpodročja porazdeljenega računalništva; *omrežno računalništvo, prostovoljno računalništvo in računalništvo v oblaku*. Ta ne postavljajo posebnih zahtev glede načina medsebojnega povezovanja računalnikov, zato jih v tem delu naslavljamo kot *rahlo sklopljene strukture*.

2.2.2 Omrežno računalništvo

Področje porazdeljenega računalništva, za katerega je značilno povezovanje računalniških gruč v celoto pod okriljem enega nadzornega sistema, je *omrežno računalništvo* (angl. *grid computing* [4]). Ta uporabnikom omogoča uporabo več gruč (ali pa le delov gruč) hkrati in se večinoma uporablja le v akademskih krogih. Značilnosti, ki jih pripisujemo računskim vozliščem v omrežnem računalništvu, so rahla

[2] *"Manjše start-up podjetje potrebuje 50 milijonov dolarjev, da ustvari in postavi na tržišče nov izdelek na področju superračunalništva."* – povzeto po *The Eleven Rules of Supercomputer Design (Gordon Bell 1989)*.

povezanost, heterogenost, geografska razpršenost in uporaba za doseganje istih ciljev. Zakasnitve v omrežnem računalništvu so višje kot znotraj gruče in programi ne smejo pogosto uporabljati sinhronizacije med vozlišči, ali pa morajo biti bolje prilagojeni za tako okolje, kar od uporabnika zahteva poznavanje arhitekture. Pristop je zato primernejši za probleme, razdeljene v neodvisne dele. Ker je infrastruktura omrežnega računalništva lahko v lasti različnih organizacij, je razvoj močno usmerjen v nadzor nad dostopom, pravicami in načini uporabe infrastrukture. Če so vozlišča pod nadzorom različnih organizacij, posamezna organizacija nikoli nima nadzora nad celotnim omrežjem.

Pri najosnovnejšem načinu uporabe omrežnega računalništva uporabnik odda v sistem zahtevo za opravilo. Zahteva za izvajanje opravila vsebuje izvršljivo kodo opravila, zraven pa tudi opis vozlišč, ki jih potrebuje. Sistem uporabniku pošlje naslov, na katerem lahko spremlja delovanje opravila, in naslov, na katerem bodo po končanem opravilu na voljo rezultati.

Primeri omrežnega računalništva so Slovenski nacionalni grid – Sling [29], ki je tudi del evropske iniciative za omrežno računalništvo – EGI, ter razvojni okolji Grid'5000 [30] (Francija) in FutureGrid (ZDA).

2.2.3 Prostovoljno računalništvo in sistemi enak z enakim

Najbolj porazdeljena oblika računalništva je prostovoljno računalništvo (angl. *volunteer computing*), pri katerem lahko računsko moč svojih računalnikov ponudi vsak posameznik ali organizacija z namestitvijo posebnega programa na svojo infrastrukturo – npr. BOINC [31]. Vsak prostovoljec dobi iz centralnega računalnika opravilo, in ko je opravilo opravljeno, pošlje rezultate nazaj centralnemu računalniku. Uporabnik se ne more zanašati na kontinuiteto vozlišč, saj se lahko v vsakem trenutku vključijo ali prenehajo z delovanjem. Najbolj znana projekta sta SETI@home in Folding@home [3, 32]. Podobne koncepte souporabe računskih virov na ravni gruč se uporablja tudi v omrežnem računalništvu.

Kljub veliki moči, ki jo lahko doseže ta pristop, je za veliko naravnih problemov manj uporaben kot gruče. Težava je v zelo omejeni sinhronizaciji in visoki dinamiki sistema ter dinamiki prisotnosti virov na vozliščih. Napake in izpade vozlišč težje predvidimo, saj imamo zelo slab nadzor nad posameznimi vozlišči. Rešuje se jih s ponovnim računanjem, kar pomeni, da so nekatera vozlišča zelo slabo izkoriščena.

Med porazdeljeno računalništvo uvrščamo tudi sisteme enak z enakim (angl. *peer-to-peer*), ki se označujejo s kratico P2P. Sistem P2P je podoben prostovoljnemu računanju z vidika, da se vsako vozlišče prostovoljno vključi v sistem in ponudi svoje vire. Razlika pa je, da sistemi P2P nimajo skupnega osrednjega vozlišča ali strežnika, ampak vsako vozlišče nastopa v obeh vlogah – kot strežnik in kot

odjemalec. Sistemi P2P so zelo specifični in niso primerni za splošno uporabo, vendar se še vedno uporabljajo za izmenjavo podatkov, ki jo uporabljajo tudi nekateri porazdeljeni pristopi za upravljanje z viri, denimo sistem za upravljanje z viri pri XtreemOS.

Pomembno orodje za razvoj in testiranje komunikacijskih porazdeljenih algoritmov (tudi sistemov P2P) z realnimi zakasnitvami med vozlišči je infrastruktura PlanetLab [33]. V konzorcij uporabnikov PlanetLab je vključenih veliko organizacij in vsaka članica v omrežje doda nekaj vozlišč. Na ta način so dosegli, da ima omrežje vozlišča po vsem svetu ter imajo simulacije na njem prave zakasnitve med mesti in celinami.

2.2.4 Računalništvo v oblaku

Računalništvo v oblaku [34] ni povsem nov koncept, saj je na nek način osnovan na spoznanjih omrežnega računalništva, gruč, storitvenega računalništva in drugih [35]. Računalništvo v oblaku je deležno veliko večje pozornosti kot omrežno računalništvo predvsem zato, ker je ekonomsko zanimivejše in je njegova uporaba bližja povprečnim uporabnikom. Omogoča razmeroma preprosto transparentno zagotavljanje in obračunavanje računskih virov ali opravljenih storitev, ki jih poganjajo računski centri.

Najosnovnejše opravilo v oblaku je naročilo navidezne naprave. Uporabnik se prijavi v oblak, kjer izbere infrastrukturo v obliki navideznih naprav in omrežij. Navidezna naprava je lahko abstrakcija strojne opreme ali pa že vsebuje prednaložen operacijski sistem. Upravljalnik oblaka ustvari navidezne naprave, jih postavi na fizično infrastrukturo in uporabniku sporoči informacije o dostopu do njegove infrastrukture. Ko uporabnik navidezne infrastrukture ne potrebuje več, jo lahko uniči ali shrani.

Prednost računalništva v oblaku je raven abstrakcije, ki skrije heterogenost fizične infrastrukture in uporabnikom ponudi infrastrukturo, platformo ali programsko opremo kot storitev, ki je neodvisna od fizične infrastrukture. Poleg abstrakcije ponuja tudi boljšo izolacijo, ker si uporabniki na isti infrastrukturi ne delijo operacijskega sistema. Arhitektura fizične infrastrukture je še vedno podobna kot pri omrežnem računalništvu, le omenjeni način ponujanja virov s storitvami omogoča način plačevanja virov po porabi, kar daje občutek, da se računanje lahko obravnava kot javna storitev, podobno kot distribucija vode, elektrike, telefonije [34]. Storitve, ki jih ponuja računalništvo v oblaku, so infrastruktura kot storitev – IaaS (angl. *IaaS – infrastructure as a service*), platforma kot storitev – PaaS (angl. *PaaS – platform as a service*) in programska oprema kot storitev – SaaS (angl. *SaaS – software as a service*).

Abstrakcija virov in storitveni pristop omogočata zelo pestro uporabo računalniških virov. Tako je infrastruktura računalništva v oblaku lahko namenjena tudi za doseganje več različnih ciljev hkrati, medtem ko je infrastruktura gruč in omrežnega računalništva primarno povezana z namenom doseganja istega cilja.

Glede na zasebnost in vpliv uporabnika na infrastrukturo obstajajo različne vrste oblakov. Če je infrastruktura izključno namenjena eni organizaciji, ki ima popoln nadzor nad njo, temu pravimo zasebni oblak (angl. *private cloud*). Javni oblak (angl. *public cloud*) je namenjen in dostopen širši javnosti, infrastruktura pa je v lasti zasebnega podjetja. Za skupnostni oblak (angl. *community cloud*) velja, da je infrastruktura v lasti konzorcija organizacij s skupnimi potrebami po računskih virih in lahko tudi skupnimi cilji. V takem primeru je infrastruktura v lasti organizacij in si jo med seboj posojajo. Zasebnemu oblaku, ki se ob konicah razširi v javni oblak in s tem zagotavlja zadostno količino virov za izvajanje opravila, rečemo hibridni oblak (angl. *hybrid cloud*). Omenjenemu procesu razširitve v javni oblak pravimo tudi razširitev oblaka (angl. *cloudbursting*). Hibridni oblak tako združuje prednosti zasebnega oblaka in navidezno neskončno moč javnega oblaka [36].

2.2.5 Virtualizacija infrastrukture

Virtualizacija v računalniškem okolju pomeni, da uporabniki (ali programi) dobijo na voljo le abstrakcijo dejanskih računskih virov, vozlišč, omrežij ipd. Tako poznamo več vrst virtualizacij, kot so strojna ali programska virtualizacija, virtualizacija namizja, pomnilnika, hrambe, podatkov in omrežij. V tem delu se osredotočamo na prednosti strojne virtualizacije, ki večjemu številu procesov omogoča souporabo fizične infrastrukture ob pomoči navideznih naprav. Z drugimi besedami, na fizična vozlišča lahko postavimo več navideznih naprav, kjer se vsaka obnaša kot resnični računalnik s svojim operacijskim sistemom.

Za nas najpomembnejša prednost virtualizacije je delno dodeljevanje fizične infrastrukture ob uporabi navideznih naprav. Virtualizacijo omogočajo nadzorniki navideznih naprav (angl. *hypervisors*), ki skrbijo za dodeljevanje virov fizične infrastrukture navideznim napravam in za izolacijo virov. Trenutno pogosto izbrani nadzorniki navideznih naprav so Xen [37], Virtuozzo[38], VMWare vSphere [39]. Navidezne naprave se med seboj lahko motijo, če imajo opravila znotraj njih neusklajene potrebe po predpomnilniku ali po zaseganju V/I-enot. Probleme s predpomnilnikom je mogoče zmanjšati s CPE z večjo in boljšo pomnilniško arhitekturo [40], pripenjanjem jeder (angl. *core pinning*) in naprednim sobivanjem opravil z različnim pomnilniškim odtisom (angl. *cache footprint*) [41]. Prizadevanja za kakovostno izolacijo opravil znotraj navideznih naprav so pomembna tudi zaradi varnosti, ki je osnova za obstoj javnih oblakov.

Poleg delnega dodeljevanja in višje ravni izolacije, ima virtualizacija še druge prednosti pri izrabi infrastrukture. Omogoča izvajanje različnih operacijskih sistemov na isti infrastrukturi, predpripravo slik operacijskih sistemov in prožno izvajanje opravil. Prožnost omogoča premikanje navideznih naprav med fizičnimi vozlišči, dodajanje ali odvzemanje virov ter prekinjanje in nadaljevanje opravila pozneje.

Pri virtualizaciji velja, da navidezna naprava ne more biti močnejša od gostitelja. Kljub temu je mogoče na gostiteljih poganjati skupino navideznih naprav, katerih skupna zmogljivost je močnejša. Takemu dodeljevanju navideznih naprav rečemo prekomerno zagotavljanje virov (angl. *overprovisioning*). To pomeni, da nekatere navidezne naprave v določenem času ne dobijo toliko virov, kolikor bi jih potrebovale. Ker navidezne naprave niso vedno polno obremenjene, se lahko zgodi, da opravila znotraj njih niti ne opazijo, da nimajo vedno na voljo vseh zagotovljenih virov.

2.3 Lastnosti opravil

Vsaka zahteva za izvajanje opravila vsebuje opis želenih virov in omejitev s strani uporabnika. Različne lastnosti virov, po katerih je mogoče opravilo kategorizirati, lahko zelo vplivajo na izrabo vozlišč. Razvrstitev opravil v kategorije je ključnega pomena, saj je to podlaga za izbiro razvrščevalnega postopka. Določanje kategorij je lahko različno glede na cilje in posebnosti računskega centra, saj je mogoče opravila deliti po tem, kakšne probleme rešujejo in katero ureditev infrastrukture potrebujejo. Mi bomo predstavili zelo splošno kategorizacijo na podlagi posameznih lastnosti opravila, ki je neodvisna od infrastrukture.

2.3.1 Toga opravila

Značilnost togih opravil je, da jim uporabnik v njihovem opisu fiksno določi lastnosti in zahtevane količine virov. Večina razvrščevalnikov na svojem vhodu predvideva taka opravila. Toga opravila imajo navadno določene naslednje omejitve:

Skrajni čas zaključka opravil. Skrajni čas zaključka prepreči stradanje opravil (tj. prekomerno odlašanje z začetkom izvajanja), a po drugi plati omeji možnosti pri razvrščanju.

Vzporednost opravil. Vzporedna opravila so tista, ki sočasno potrebujejo več vozlišč in navadno tudi redno komunikacijo med njimi. Opravila, ki so vzporedna, so bolj toga, saj je težje zagotoviti prostor na več vozliščih hkrati.

Soodvisnost opravil. Soodvisna opravila so mešanica med zaporednimi in vzporednimi opravili, ki so povezana s (pred)pogoji. Predpogoji opravil določajo, katera opravila se morajo končati pred začetkom izvajanja drugih opravil. Na ta način dobimo strogo zaporedna in nekaj vzporednih opravil. Množici tako povezanih opravil rečemo delovni tok (angl. *workflow*) [42]. Ker relacije med opravili otežijo njihovo razvrščanje, obstaja veliko načinov razvrščanja delovnih tokov [43, 44, 45].

Prioriteta opravil. Opravila so bolj toga, če imajo različne prioritete, ki določajo, katera opravila imajo pri izvajanju prednost pred drugimi. Na nek način je to poseben primer soodvisnih opravil.

Homogenost in heterogenost opravil. Homogena vzporedna opravila zahtevajo več enako zmogljivih vozlišč, oziroma so omejena z zmogljivostjo najpočasnejšega dodeljenega vozlišča. Najpogosteje homogena opravila na različnih vozliščih opravljajo enako delo z drugačnimi vhodnimi podatki. Heterogena opravila pa na vozliščih opravljajo popolnoma različno delo in želijo različno zmogljive skupine vozlišč (npr. delovna in podatkovna vozlišča pri MapReduce).

2.3.2 Fleksibilna opravila

Razvrščanje opravil je učinkovitejše, če omejitve niso stroge in jih lahko razvrščevalnik prilagodi glede na stanje virov na infrastrukturi. Najpogostejše lastnosti, ki naredijo opravila fleksibilnejša, so:

Časovna fleksibilnost. Najpogostejša fleksibilnost je zamik začetka izvajanja opravila. Odstranitev časovnih omejitev pomaga razvrščevalniku, vendar ni primerna za vsa opravila. Nekatera opravila potrebujejo interakcijo z drugimi entitetami zunaj opravila. Strežniki storitev so denimo opravila, ki jih navadno ne moremo izvajati ob poljubnem času, saj morajo biti na voljo, kadarkoli nek proces potrebuje njihovo strežbo.

Možnost prekinjanja. Nekatera opravila lahko prekinejo (angl. *suspend*) svoje izvajanje v korist drugih, pomembnejših opravil. Poznamo dve možnosti obravnave s prekinjenimi opravili. Prva možnost je, da se morajo prekinjena opravila začeti izvajati ponovno od začetka. Toda v nekaterih primerih se lahko stanje prekinjenega opravila shrani in je mogoče opravilo nadaljevati tam, kjer smo ga prekinili (angl. *suspend-resume*) [46].

Prilagodljivost. Prilagodljiva opravila so tista, ki jim lahko zahteve po virih spremenimo pred ali celo med izvajanjem in s tem vplivamo na trajanje izvajanja opravila. To velja za spremembo vzporednosti opravil [47] ali spremembo fizične infrastrukture [48]. Spremembo vzporednosti, oziroma števila uporabljenih vozlišč, imenujemo *horizontalno skaliranje* (angl. *horizontal scaling*), medtem ko spremembo zmogljivosti opravilu dodeljenih vozlišč imenujemo *vertikalno skaliranje* (angl. *vertical scaling*).

Opravila, ki jih lahko preoblikujemo pred izvajanjem, imenujemo kalupljena, tista, ki jih lahko preoblikujemo med izvajanjem, pa kovna (angl. *moldable and malleable jobs*) [24]. Jasno je, da niso vsa opravila primerna za to, prav tako tudi ne vsi operacijski sistemi, na katerih opravila tečejo.

Prilagodljiva opravila močno zapletejo razvrščevalnik, saj omogočajo veliko več rešitev za razvrščanje. Za vsako prilagodljivo opravilo velja, da je v zahtevi navedenih več konfiguracij infrastrukture, ki jih opravilo lahko učinkovito izrabi, v nekaterih primerih pa je podana funkcija, ki povezuje čas izvajanja opravila glede na konfiguracijo virov. Problem razvrščanja tako postane kompleksnejši, saj že konfiguracije na podlagi različnega števila vozlišč močno razširijo prostor reševanja. Kljub temu podatki o izvajanju opravila na vsaki konfiguraciji vzporednih virov pogosto niso dovolj, saj na opravilo vplivajo tudi drugi viri zunaj vozlišč (npr. omrežje), veliko opravil pa je tudi občutljivih na lokalnost (angl. *locality-sensitive*) [49].

2.4 Pristopi k razvrščanju in upravljanju z viri

Porazdeljeno računalništvo potrebuje aplikacije, ki omogočajo, da se zahteve pravilno obdelajo, opravila dobijo primerne vire (ali vozlišča) in hkrati ohranjajo visoko učinkovitost sistema. Poleg aplikacij za razvrščanje sistem potrebuje tudi druge, ki skrbijo za nadzor (angl. *monitoring*) in iskanje virov (angl. *resource discovery*). Ker morajo vse aplikacije delovati v simbiozi, moramo vedeti, kako vplivajo druga na drugo. Zato je to poglavje namenjeno najpogostejšim načinom razvrščanja, razvrščanju z vrstami in rezervacijami ter ureditvi in načinu dodeljevanja infrastrukture.

2.4.1 Pogosti pristopi k razvrščanju

Razvrščanje, ki ga opravlja razvrščevalnik, predstavlja časovno odvisno preslikavo, ki preslika opravila na vozlišča in ustreza vsem omejitvam. Del omejitev predstavljajo zahteve uporabnika in zmožnosti infrastrukture, drugi del pa politike razvr-

ščanja, ki celostno urejajo nefunkcionalne zahteve razvrščanja – skrbijo za prioriteto, pravičnost razvrščanja in podobno. Nekaj znanih pristopov k razvrščanju:

Sprotno razvrščanje. Sprotno razvrščanje (angl. *online scheduling*) se proži dogodkovno; denimo ko pride zahteva v sistem ali ko se je na sistemu sprostilo dovolj virov. Različni pristopi sprotnega razvrščanja (če omenimo le nekatere) uravnavajo izvajanje opravil glede na pogostost zahtev [50], intenzivnost dela v posameznem opravilu [51] ter kompromis med točno in netočno rešitvijo posameznih opravkov v delovnem toku [52].

Odloženo razvrščanje. Odloženo razvrščanje (angl. *offline scheduling*) se največkrat proži časovno; ciklično po preteku določenega časa denimo preverimo, ali so v sistem prišla nova opravila. Druga možnost je, da se proži dogodkovno, denimo da razvrščanje odložimo na trenutek, ko v sistem pride zadostno število opravil. Najmanjša časovna oznaka se šteje kot čas med prožen jem razvrščanja, ki mu pravimo časovni korak (angl. *slot*).

Zasipanje. Zasipanje (angl. *backfilling*) [53] je način razvrščanja, pri katerem z manj nujnimi opravili zapolnimo proste vrzeli v urniku, ko je urnik že sestavljen za vsa nujna opravila. Manj nujno opravilo lahko denimo postavimo v prvo primerno vrzel, najprimernejšo vrzel ipd. (angl. *first fit, best fit*).

Ponovno razvrščanje. Ponovno razvrščanje (angl. *rescheduling*) [54] se izvaja naknadno, ko so opravila že razvrščena. Lahko bi rekli, da je to poseben primer odloženega razvrščanja, ki popravlja že sestavljeni urnik. Ponovno razvrščanje se uporablja za odpravljanje poslabšanja učinkovitosti, ki je posledica predčasno zaključenih opravil, izpadov vozlišč ipd. Prednost ponovnega razvrščanja je aktualnejši pogled na stanje infrastrukture in opravil. Pristop se najpogosteje uporablja v sistemih omrežnega računalništva [48, 55], pogoj za ponovno razvrščanje pa so fleksibilna opravila.

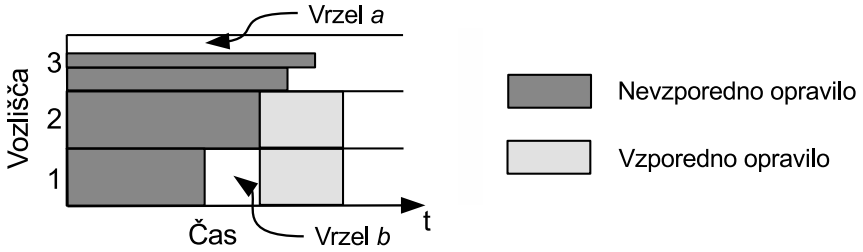
Omenjene pristope k razvrščanju lahko uporabimo na različne načine. V nadaljevanju bomo predstavili razvrščanje z vrstami in rezervacijami.

2.4.2 Razvrščanje z vrstami

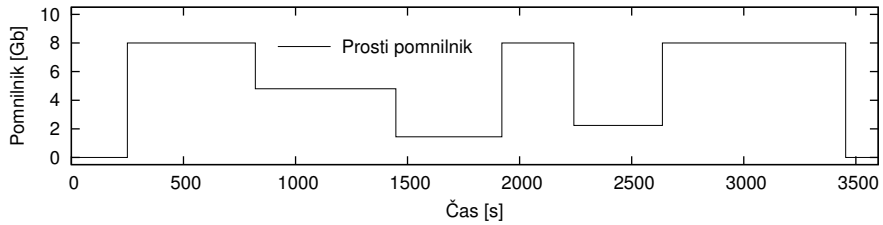
Preprosto razvrščanje lahko naredimo tako, da opravila postavimo v vrsto in potem vsakemu dodelimo vire na infrastrukturi. Z urejanjem opravil v vrsti lahko vplivamo na prednost opravil pri razvrščanju. Pogosto sprotno razvrščanje uporablja vrste tako, da nove sproščene vire v infrastrukturi dodeli naslednjemu opravilu v vrsti. Ker opravila do začetka izvajanja čakajo in še nimajo dodeljenega časa za začetek izvajanja, je mogoče razvrščanje opravil sproti prilagajati spremembam v sistemu.

2.4.3 Razvrščanje z rezervacijami

Rezervacije so zahteve za opravila, ki imajo poleg kapacitet virov določen začetni čas in trajanje. Informacija o vseh sprejetih opravilih in rezerviranih virih na vozliščih tvori urnik. Uspešna rezervacija natančno določi čas izvajanja opravila in ga postavi v urnik. Če razvrščevalnik v urniku ni našel prostora za rezervacijo, se zahteva zavrne in se o tem obvesti uporabnika. Nezasedeni prostor v urniku je vrzel, ki nastane, če opravila ne porabijo vseh virov na vozlišču (vrzel *a* na sliki 2.1a) ali če zaradi omejitev opravil na katerega od vozlišč ne moremo uvrstiti nobenega opravila (vrzel *b* na sliki 2.1a prikazuje urnik nevzporednih opravil, v katerega dodamo vzporedno opravilo). Količino vira, ki je na voljo v vrzeli, lahko dobimo iz urnika razpoložljivosti posameznega vira na vozlišču (slika 2.1b).



(a) Urnik opravil



(b) Urnik razpoložljivosti vira na vozlišču

Slika 2.1: Urniki pri rezervacijah.

Pristop z rezervacijami ne dovoljuje spreminjanja urnika za obstoječe rezervacije, ob prihodu novih opravil ali spremembah opravil v izvajanju. Rezervacije so ključnega pomena, kadar želimo čas izvajanja opravila povezati z zunanjimi entitetami, kot so merilni inštrumenti ali drugi računski viri, pri sočasnem dostopu do vozlišč znotraj različnih računskih centrov – sočasno dodeljevanje vozlišč [21].

Z rezervacijami preprosteje določimo čakalno dobo opravila, ki je pri drugih pristopih ni vedno mogoče napovedati [56]. Zato so rezervacije pomemben del zagotavljanja kakovosti storitve. Slabost rezervacij je ta, da zasedanje točno takrat, ko

si uporabniki želijo, za infrastrukturo ni najučinkovitejše, saj pelje k povečevanju števila vrzeli v urniku. Zato je smiselno, da so opravila fleksibilna, dokler rezervacija ni potrjena. Tak način uporablja pristop s pogajanjem in pripravo protiponudb [57], ki lahko omeji število vrzeli v urniku, vendar smo s tem omejeni na umeščanje le zadnje rezervacije naenkrat.

Dodatno neizkoriščenost pri uporabi rezervacij povzročijo predčasno končane rezervacije, ki tudi povzročijo vrzeli v urniku. Te vrzeli lahko zapolnjujemo z opravili z nizko prednostjo.

2.4.4 Organizacija infrastrukture

Kakovostno dodeljevanje infrastrukture potrebuje podatke o vozliščih, ki jih dobimo s sistemi za nadzor in iskanje virov, ter podatke o opravilih. Sistem je lahko centraliziran, kar pomeni, da ima razvrščevalnik globalen pogled na stanje infrastrukture in tudi odločitev o razvrstitvi opravila se izvrši na enem mestu. Druga oblika je porazdeljen sistem, kjer se podatki o stanju infrastrukture hranijo in osvežujejo na različnih lokacijah ob uporabi skupine manjših procesov [58, 59]. Pri taki ureditvi o razvrstitvah odločajo manjši procesi na različnih lokacijah samostojno ali v sodelovanju z drugimi deli sistema.

Razlogi, kateri način upravljanja izbrati, so različni od vsakega posameznega primerka infrastrukture. V osnovi je odvisno od zmožnosti obvladovanja količine podatkov, ki so potrebni za razvrščanje. Ti so odvisni tudi od načina razvrščanja. Pri uporabi rezervacij denimo ni dovolj le trenutno stanje, ampak moramo poznati še urnik za vsako vozlišče, kar zelo poveča količino podatkov. Osveževanje podatkov pri centraliziranem pristopu je lahko neučinkovito pri velikih porazdeljenih sistemih, prav tako pa pogosto ni potrebno, da bi na eni točki imeli vse podatke o vseh vozliščih. V teh primerih je bolje uporabiti porazdeljeno upravljanje infrastrukture. Na ta način je lažje obdelati večjo količino zahtev, saj manjše lokalne zahteve po infrastrukturi obravnavajo posamezna vozlišča samostojno, za večja pa si izmenjajo podatke s sosedi.

2.4.5 Dodeljevanje vozlišč

Način razvrščanja je odvisen od politike razvrščanja in različnih načinov dodeljevanja virov. Na začetku so se superračunalniki in gruče dodeljevali v celoti, kot se dodeljujejo še danes za reševanje velikih problemov na namenskih infrastrukturah. Ker danes ni veliko opravil, ki bi potrebovala take zmogljivosti, so se razvili upravljalniki virov, ki omogočajo dodeljevanje posameznih vozlišč znotraj gruč ali mrežnega računalništva. Pred uveljavitvijo virtualizacije je bila osnovna enota

za dodeljevanje posamezno vozlišče, le redke izjeme so omogočale več opravil na enem vozlišču [25, 60]. Pri uporabi delnega dodeljevanja infrastrukture si opravila delijo isto vozlišče in njegove vire, kar je lahko problematično. Lahko se zgodi, da opravila zaradi medsebojnih vplivov, kot je zaseganje virov na vozlišču, ne dobijo toliko virov, kolikor bi jih potrebovala.

Sočasno dodeljevanje infrastrukture (angl. *co-allocation*) [16], ki zagotovi proste vire za isto časovno obdobje na več vozliščih, poveča kompleksnost iskanja primerne vrzeli v urniku. Algoritmi za sočasno dodeljevanje računalniške infrastrukture so razviti predvsem za omrežno računalništvo [21]. S kombinacijo delnega in sočasnega dodeljevanja postane upravljanje z viri kompleksnejše zaradi zahtevnejšega nadzorovanja prostih kapacitet in iskanja rešitev.

2.5 Upravljalniki z viri in razvrščevalniki

Obstaja veliko upravljalnikov z viri in razvrščevalnikov, med katerimi bomo omenili najpomembnejše, ki predstavljajo smernice in ideje, povezane s tem doktorskim delom. Predstavili jih bomo po kategorijah, ki se določijo glede na infrastrukturo. Podrobneje bomo opisali razvrščevalnik Haizea in upravljanje z viri v operacijskem sistemu XtreemOS, ki sta osnovi naših prispevkov. Podrobnejše preglede lahko bralec najde v delih avtorjev Klein-Halmagi et al. [61] in Netto et al. [21].

Večina razvrščevalnikov je zasnovanih modularno. Zato lahko s spremembo konfiguracije modulov spremenimo tudi kategorijo razvrščevalnika. Hkrati lahko razširimo podporo za drugačno infrastrukturo ali vrsto opravil.

2.5.1 Paketni razvrščevalniki

Paketni razvrščevalniki (angl. *batch schedulers*) so dobili ime po paketnih datotekah (angl. *batch file*), ki se uporabljajo za opisovanje izvajanja opravil na sistemih za paketno obdelavo (angl. *batch processing*). Skrbijo za avtomatizirano izvajanje opravil in so najpogostejša oblika upravljanja z viri na večjih računskih infrastrukturah. Njihov vmesnik je največkrat zelo splošen zato, da ustreza večini uporabnikov. Večina paketnih razvrščevalnikov podpira razvrstitev togih opravil v vrste ali pripravo rezervacij, posamezni pa imajo tudi podporo za prilagodljiva opravila, opravila z nizko prednostjo in dinamična opravila. Primeri paketnih razvrščevalnikov so HTCondor [62], SLURM [63], OAR [64], TORQUE [65] in Univa Grid Engine [66].

HTCondor [62] je orodje, ki je bilo pred letom 2012 poznano pod imenom Condor [67] in je specializirano za upravljanje računsko intenzivnih opravil. HTCondor

vsebuje mehanizme za upravljanje vrst, razvrščanje, nadzor nad vozlišči in upravljanje z viri. Za nas je najzanimivejši del funkcionalnosti pristop k preslikavi opravil na vozlišča. Ta se naredi z jezikom ClassAd (okrajšava za *class advertising*), ki nudi zelo prilagodljivo in izčrpno orodje za iskanje primerne množice virov, navedenih v zahtevi. ClassAd določa opise lastnosti vozlišč, njihovih virov in zahteve za opravilo. Opisi spominjajo na dvostransko dražbo, saj so vozlišča in zahteve za opravila opisani v obliki oglasov, poseben proces ujemanja (angl. *matchmaking*) pa potem poskrbi za razvrščanje opravil na vozlišča. Iz množice vozlišč, ki so primerna za opravilo, se izberejo tista, ki imajo boljšo oceno (angl. *rank*). Način izračunavanja ocene določi uporabnik.

Orodje SLURM (the Simple Linux Utility for Resource Management)[63], poznano tudi kot SLURM Workload Manager, je odprtokodni sistem za upravljanje gruč in razvrščanje opravil. SLURM podpira tudi prilagodljiva opravila, tako da je mogoče pri zahtevi za opravilo določiti različen nabor količin glede na različno število vozlišč. S tem opravilu omogoča boljšo učinkovitost in hitrejše izvajanje z dodatnimi viri, če so na voljo. Vendar specifikacija ne omogoča opisa različnih časov trajanja opravila glede na dodatne vire, kar pomeni, da bi z zasipanjem dobre preslikave v urniku pogosto izpustili. Posledično bi se opravilo začelo pozneje, kot ob boljši specifikaciji. Dodatna posebnost razvrščevalnika SLURM je podpora dinamičnim opravilom z razširjanjem in krčenjem, kar upravlja uporabnik.

Razvrščevalnik OAR [30, 64] se primarno uporablja za upravljanje z viri na raziskovalni platformi Grid'5000. Poleg paketnega razvrščanja podpira rezervacije in interaktivna opravila, omogoča postavitev lastnega okolja, zasipanje, izbor točno določenih vozlišč in tako dalje. OAR omogoča uporabniku, da v zahtevi navede več različnih konfiguracij virov, potrebnih za opravilo. Te vsebujejo število vozlišč in maksimalen čas izvajanja. Tako lahko razvrščevalnik pred uvrščanjem izbere tisto, ki bo minimizirala čas izvajanja opravila glede na razpoložljive vire.

Odprtokodni upravljalnik z viri TORQUE [65] je osnovan na projektu PBS (OpenPBS), ki ga je na začetku razvijala organizacija NASA Ames Research Center. Podobno kot OAR tudi TORQUE omogoča uporabniku, da navede več različnih konfiguracij virov za izvajanje opravila in s tem podpira delo s prilagodljivimi opravili. Pri razvrstitvi opravila izbere konfiguracijo virov, ki bo najučinkoviteje izrabila vire na vozliščih.

2.5.2 Upravljalniki oblakov

Upravljalniki oblakov (angl. *cloud managers*) dodeljujejo navidezno infrastrukturo, kot so navidezne naprave, navidezne hrambe in navidezna omrežja, primerni fizični opremi. Upravljanje poteka prek nadzornikov navideznih naprav (angl. *hypervi-*

sors), ki skrbijo za virtualizacijo virov. Virtualizacija omogoča, da se opravila v oblaku premikajo po fizični infrastrukturi, krčijo in širijo. Zato upravljalniki oblakov primarno rešujejo trenutno zasedenost v oblaku ter se ne ukvarjajo z vrstami in rezervacijami. Podrobneje bomo predstavili upravljalnike, kot so OpenNebula [68], razširitev za OpenNebulo, poimenovano Haizea [69, 70], Amazon EC2/EC3 [71] in OpenStack [72]. Preostali pogosto uporabljeni upravljalniki so VMWare vCloud [73], Eucalyptus [74] in Nimbus [75].

OpenNebula in primer uporabe

OpenNebula [68] je eno prvih odprtokodnih orodij na področju upravljalnikov oblakov. Projekt se je začel v letu 2005, prvo programsko opremo pa so izdali v letu 2008. Večino inovacij in funkcionalnosti je pridobila s potrebami raziskovalnih projektov na področju računalništva v oblaku, kot so RESERVOIR, StratusLab, BonFIRE in 4CaaSt. Večina komponent orodja OpenNebula je razvita v jezikih Ruby in C++.

Medtem ko je cilj javnih ponudnikov prodaja računskih virov na način, da izpostavijo vmesnik oblaka zunanjim uporabnikom, želi OpenNebula vire ponuditi lokalnim uporabnikom znotraj administracijske domene. Tako OpenNebula z vmesniki za posamezne komponente upravljanja oblaka ponuja uporabnikom in administratorjem fleksibilno in agilno infrastrukturo za izvajanje navideznih storitev. Ker je bilo orodje OpenNebula med prvimi, je bilo pogosto izbrano za postavljanje zasebnih in hibridnih oblakov, ki se lahko razširijo v oblak drugega ponudnika (kot npr. Amazon EC2).

Orodje OpenNebula upravljamo z ukazi v konzoli ali pa prek grafičnega vmesnika Sunstone [76], kjer imamo nadzor nad navideznimi napravami, njihovimi predpripravljenimi slikami, navideznimi omrežji in uporabniki. Zahtevek za novo navidezno napravo vsebuje izbiro slike, ki jo želimo zagnati, opis navidezne naprave glede na vire (CPE, pomnilnik, trdi disk, navidezno omrežje itd.) in prilagoditve. Prilagoditve vsebujejo nabor skriptov, ki se zaženejo ob zagonu navidezne naprave ter poskrbijo za zagon in namestitev opravil.

Če zahtevek ni veljaven ali ni ustrezen glede na infrastrukturo, ga OpenNebula zavrne. V nasprotnem primeru se navidezna naprava uvrsti na seznam in dobi oznako *pending* (čakanje). V tem času poteka razvrščanje z razvrščevalnikom. Razvrščevalnik na podlagi ujemanj (angl. *match-making*)[3] najprej odstrani vsa tista vozlišča in navidezne naprave, med katerimi ni mogoče preslikave. Nato glede na politiko razvrščanja uredi navidezne naprave in vozlišča ter vsaki napravi določi fizično vozlišče. Povečanje učinkovitosti rabe infrastrukture nam omogočata bremensko

[3] http://archives.opennebula.org/documentation:rel4.4:schg

ozaveščen (angl. *load-aware*) in strpan (angl. *packing*) način izbire vozlišča, na izbiro pa so tudi drugi načini, kot denimo porazdeljeni (angl. *stripping*), fiksni (angl. *fixed*) in prikrojeni (angl. *custom*).

Ko opravilo dobi zahtevane vire, se prenesejo potrebni podatki na vozlišče in se začne zaganjati navidezna naprava (stanje *boot*). Ko se opravilo izvaja, preide v stanje *running* in ob uspešnem zaključku v stanje *done*. Upravljalniki oblakov omogočajo tudi vrsto drugih operacij nad navideznimi napravami, kot so prekinitev in nadaljevanje (angl. *suspend and resume*) ter premikanje med vozlišči.

Haizea

Razvrščevalnik Haizea [69, 70] je odprtokodni upravljalnik oblaka, ki s povezavo na OpenNebulo nadomesti njen notranji razvrščevalnik in vpelje v upravljanje oblaka pristope, ki smo jih predstavili pri paketnih razvrščevalnikih – to so vrste in rezervacije. Razvrščevalnik Haizea abstrahira dodeljevanje in zagotavljanje virov z uvedbo *najemov* (angl. *lease*), ki jih v tem doktorskem delu imenujemo zahteve. Če je zahteva sprejeta, služi kot pogodba o zagotavljanju množice vozlišč med uporabnikom in ponudnikom, v nasprotnem primeru pa je zahteva zavrnjena, neveljavna. Tako uporabnik opiše potrebe opravila v zahtevi, razvrščevalnik Haizea sprejeto zahtevo uvrsti v urnik, pred izvajanjem opravila pa se zahteva pretvori v primerno število virtualiziranih virov (navidezne naprave, hrambe ipd.).

Zahteva mora pri razvrščevalniku Haizea vsebovati kapaciteto virov in trajanje uporabe, izbirno pa so lahko dodani čas izvajanja (začetek, najpoznejši čas zaključka), slika navidezne naprave in podobno. Razvrščevalnik Haizea pozna štiri vrste zahtev, ki se ločijo po fleksibilnosti časovnih omejitev: rezervacija (angl. *advance reservation*), takojšna zahteva (angl. *immediate*), najboljši izkoristek (angl. *best-effort*) in najboljši izkoristek z rokom za zaključek (angl. *best-effort with deadline*). Pri rezervaciji uporabnik določi najzgodnejši čas začetka in najpoznejši zaključek opravila. Takojšnja zahteva se mora začeti ob prihodu v sistem. Dve vrsti zahtev, ki vključujeta najboljši izkoristek, ločimo po tem, da se ena lahko izvaja kadarkoli, saj nima časovnih omejitev, druga pa se mora zaključiti pred iztekom roka za zaključek.

Prednost razvrščevalnika Haizea je predvsem podpora upravljanju rezervacij in načrtovanju razvrščanja s prekinitvami ter nadaljevanji (angl. *suspend and resume*) navideznih naprav. Preostale funkcionalnosti omogočajo še možnost delnega zasedanja virov, dodajanje lastnih politik za sprejemanje opravil v sistem, dodajanje lastnih politik za ovrednotenje vozlišč ter uporabo rezervacij in algoritmov zasipanja (angl. *backfilling*) hkrati. Prednost razvrščevalnika Haizea je tudi ta, da se lahko uporablja kot simulator, kar bomo podrobneje predstavili v poglavju 2.6.

Razvrščevalnik Haizea pripravimo za delovanje tako, da po namestitvi spremenimo konfiguracijske datoteke, v katerih določimo, kje teče OpenNebula. Uporaba razvrščevalnika Haizea je preprosta, če poznamo uporabo orodja OpenNebula. Pri opisih slik le dodamo vrstice, namenjene razvrščevalniku Haizea, s katerimi določimo, kdaj se bo opravilo izvajalo in kako (slika 2.2).

```
NAME= vm-1
CPU = 0.5
MEMORY = 128
  OS = [
  root = "hda" ]
DISK = [
  source = "/nebula/images/vm1.qcow2",
  target = "hda",
  readonly = "no" ]
NIC = [ NETWORK = "private-network" ]
GRAPHICS = [
 type="vnc",
 listen="localhost" ]

HAIZEA = [
 start = "+00:01:00",
 duration = "00:02:00",
 preemptible = "no"
]
```

Slika 2.2: Opis navidezne naprave za OpenNebulo in dodatek s ključno besedo HAIZEA, ki je namenjen razvrščevalniku Haizea.

Drugi upravljalniki oblakov

Amazon [71] je uporabnikom prvi ponudil vire računanja v oblaku z najemanjem navideznih naprav. Uporabnik se odloči, kdaj se navidezna naprava ustvari, kako dolgo bo tekla in na kakšen način želi, da se porabljeni viri obračunavajo (npr. na uro). Hiter prodor Amazona na trg javnih oblakov je poleg dobre prepoznavnosti vplival tudi na standardizacijo, saj večina drugih upravljalnikov uporablja tudi vmesnike za Amazon EC2. Posebnost upravljalnika pri Amazonu je posebna vrsta navideznih naprav, namenjena zapolnitvi vrzeli (angl. *spot instances*) v času nizke učinkovitosti.

OpenStack [72] ima za seboj močno skupnost, ki se trudi narediti odprtokodno vseobsegajočo platformo za računalništvo v oblaku. Značilnost za OpenStack je njegova stroga modularna zgradba, v kateri je vsak modul zadolžen za določeno komponento oblaka (npr. modul za računsko vozlišče (Nova), modul za shranjevanje objektov (Swift), modul za shranjevanje blokov (Cinder), modul za mreženje

(Neutron) ipd). Vsaka komponenta se razvija neodvisno, kar omogoča hitrejše razvojne cikle. Slaba stran omenjenega razvoja je, da so pogosto komponente neke različice združljive le s točno določenimi različicami drugih komponent.

2.5.3 Porazdeljeni upravitelji virov

Paketni razvrščevalniki in upravljalniki oblakov ponujajo zelo urejen in centraliziran pristop do upravljanja z viri. Za velika in zelo agilna okolja je lahko centraliziran pristop neuporaben, zlasti pri urejanju urnikov, pri katerem moramo za vsako vozlišče voditi evidenco zasedenosti. Jasno je, da za umeščanje nekega manjšega opravila ni treba poznati zasedenosti vseh vozlišč. Pogosto je dovolj, da poznamo stanje za manjšo podmnožico vozlišč.

Porazdeljeno upravljanje z viri temelji na manjših procesih, ki tečejo porazdeljeno v omrežju in skupno tvorijo sistem za upravljanje z viri. Omenjeni način upravljanja uporablja XtreemOS.

XtreemOS

XtreemOS [60, 77] je operacijski sistem za omrežno računalništvo, ki omogoča nadzor in upravljanje z viri, ki so pod okriljem več različnih institucij. Cilj projekta XtreemOS je ponuditi uporabnikom podobno funkcionalnost za omrežno računalništvo, kot jo predstavlja tradicionalni operacijski sistem za namizni računalnik. Računalniki z nameščenim operacijskim sistemom XtreemOS pripadajo najmanj eni virtualni organizaciji [78, 79]. Računalniki lahko v omrežju sodelujejo kot delovno vozlišče, kot vstopna točka ali oboje hkrati, če uporabnik in organizacija to dovoljujeta. Operacijski sistem XtreemOS je osnovan na distribuciji Linux Mandriva in ga je mogoče uporabljati tudi kot navaden namizni računalnik z operacijskim sistemom Linux. Omenjeni način hibridnega okolja omogoča vključevanje velikega števila računalnikov v sistem, kar tvori heterogeno in agilno infrastrukturo.

Delovanje sistema je osnovano na pomembnih razvojnih idejah in gradnikih, kot so porazdeljeni datotečni sistem *XtreemFS* [80, 81], ogrodje za razvoj porazdeljenih opravil *Dixi*, transakcijska, skalabilna in porazdeljena shramba *Scalaris* [82] ter podpora mobilnim napravam [83].

Osrednji gradnik, ki upravlja z opravili in viri, je upravitelj izvajanja opravil Application Execution Manager (AEM) [84]. Opravila so opisana z zahtevami po virih z jezikom Job Submission Description Language (JSDL) [85], ki med drugim omogoča tudi ohlapnejši opis z mejami ali razponi. Zahtevo za vir lahko opišemo z razponom tako, da zapišemo minimalno in maksimalno količino, ki jo opravilo želi dobiti na vozliščih. Slednje je potrebno za iskanje primernih vozlišč v heterogenih

sistemih. Sistem za upravljanje z viri podpira takojšnje izvajanje in tudi rezervacije. V obeh primerih dovoljuje dodeljevanje virov na ekskluziven način, pri katerem si opravilo vozlišč noče deliti z drugimi opravili, ali neekskluziven način.

Vire poišče komponenta za iskanje virov in storitev SRDS (angl. *service-resource discovery service*). Sistem za iskanje vozlišč SRDS deluje po sistemu manjših procesov, ki se sporazumevajo po načinu enak z enakim (P2P), in za shranjevanje uporablja porazdeljene zgoščevalne tabele (DHT). Komponenti AEM in SRDS sta implementirani tako, da lahko sočasno tečeta na več namenskih vozliščih, katerih število variira glede na velikost sistema. Ta pristop omogoča, da se sistem za upravljanje z viri bolje prilagaja spremembam na omrežju ter reši probleme skaliranja in uravnavanja obremenitve [86].

Uporaba sistema XtreemOS

Uporabnik odda v sistem XtreemOS zahtevek za opravilo. Opravilo prevzame proces AEM, ki preveri dovoljenja uporabnika in pretvori zahtevo v obliko za iskanje primernih vozlišč, ki jo pošlje sistemu za iskanje virov in storitev SRDS. Sistem poišče primerne vire v okolici in zoži izbor tako, da zavrne vsa tista vozlišča, ki očitno ne ustrezajo zahtevi – postopek je podoben kot filtriranje pri ujemalnem razvrščanju. SRDS nato vrne seznam vozlišč procesu AEM, ki poišče primerno množico vozlišč za izvajanje opravila. Če AEM ne najde primerne množice vozlišč, zaprosi SRDS za nova vozlišča, v nasprotnem primeru pa dodeli vire opravilu. Uporabnik lahko spremlja stanje opravila, poleg tega mu sistem dodeli lokacijo za rezultate v porazdeljenem datotečnem sistemu XtreemFS. To lokacijo lahko uporabnik priključi na svoj sistem in z nje prebere rezultate, ko se opravilo konča.

2.5.4 Metarazvrščanje

Metarazvrščevalniki so orodja, ki urejajo razvrščanje opravil na ravni gruč v povezavi z lokalnimi razvrščevalniki gruče. V tem primeru metarazvrščevalnik izbira le gručo, kjer se bo opravilo izvajalo, lokalni razvrščevalnik pa poskrbi za dodelitev virov znotraj vozlišča. Pri tem velja, da zahteve za opravila lahko pridejo na metarazvrščevalnik in tudi neposredno na lokalni razvrščevalnik gruče. Primer metarazvrščevalnika je KOALA [87, 88], ki omogoča tudi podatkovno in procesorsko sočasno dodeljevanje vozlišč v različnih centrih, osnovano na rezervacijah.

2.6 Simulatorji upravljanja virov

Porazdeljena računalniška oprema zahteva veliko prostora in denarja, prav tako zaradi nenehne uporabe s strani uporabnikov ne omogoča ponovljivosti poskusov. Zato se večina poskusnega in razvojnega dela na področju razvrščevalnih algoritmov opravi s simulatorji. Uporabiti je mogoče splošne simulatorje, vendar za simuliranje omrežnega računalništva in računalništva v oblaku obstaja široka paleta orodij, kot so simulator/razvrščevalnik Haizea, ki ga uporabljamo v tem delu, ter priljubljena simulatorja GridSim [89] in CloudSim [90]. Več informacij o preostalih lahko bralec najde v pregledu [91] ali delu EMUSIM [92], ki je nadgradnja simulatorjev GridSim in CloudSim.

2.6.1 Haizea

Razvrščevalnik Haizea [69] omogoča različne pristope izdelave urnikov za rezervacije ter je pogosta izbira raziskovalcev pri raziskavah in načrtovanju novih razvrščevalnih pristopov [93, 46, 57, 94]. Prednost razvrščevalnika Haizea na raziskovalnem področju je ta, da se lahko uporablja kot simulator ali pa kot razvrščevalnik na realni infrastrukturi s povezavo na upravljalnik oblakov OpenNebula [68]. Delovanje razvrščevalnika smo že predstavili v poglavju 2.5.2. Orodje je razvito v programskem jeziku Python, je odprtokodno in prosto dostopno [70] ter omogoča preprosto modularno spreminjanje ključnih delov programa.

Podatki za simulacijo zajemajo datoteko za opis vozlišč, datoteko za opis podatkov o opravilih (primer zapisa opravila prikazuje slika 2.3) in konfiguracijo razvrščanja. Poleg spremljanja trenutnega delovanja, ki ga Haizea izpisuje na standardni izhod, nam simulacijski del omogoča spremljanje simulacije s tako imenovanimi števci. To so spremenljivke, ki jih osvežujejo upravljalniki dogodkov, ti pa se lahko prožijo na začetku ali koncu vsake simulacije ali vsake rezervacije. Uporabniki lahko števce definirajo poljubno in tako spremljajo vsako spremembo v simulaciji.

2.6.2 GridSim, CloudSim in EMUSIM

Orodjarna GridSim [89] omogoča spreminjanje in simuliranje različnih entitet vzporednega in porazdeljenega računanja, kot so uporabniki, opravila, vozlišča in upravljalniki virov – razvrščevalniki. CloudSim [90, 95] je naslednik simulatorja GridSim in je prilagojen na simulacije delovanja računalništva v oblaku. Omogoča modeliranje računskih centrov računalništva v oblaku, energetsko učinkovitih računskih virov, federacijo oblakov, omrežnih topologij in uporabniško določenih politik sprejemanja opravil, dodeljevanja navideznih naprav fizičnim vozliščem in dode-

```
<lease-request arrival="00:00:00">
<lease preemptible="true">
  <nodes>
    <node-set numnodes="1">
      <res type="CPU" amount="100"/>
      <res type="Memory" amount="1024"/>
    </node-set>
  </nodes>
  <start></start>
  <duration time="01:00:00"/>
  <software>
    <disk-image id="test.img" size="1024"/>
  </software>
</lease>
</lease-request>
```

Slika 2.3: Primer opisa zahteve za simulacijo.

ljevanja fizičnih vozlišč navideznim napravam. Simulacijsko okolje je odprtokodni projekt, ki je razvit v programskem jeziku Java.

Največja razlika v primerjavi s Haizeo je, da ni specializiran za rezervacijo virov vnaprej in zato tudi ne uporablja tako širokega jezika za opis opravil. Njegova glavna usmeritev je uporaba sprotnega razvrščanja ter spremljanje federacij gruč in oblakov.

Nadgradnja obeh sistemov je EMUSIM [92, 96], ki združuje prednosti paketa za emulacijo (AEF) in simulacijo (CloudSim). Ključna pridobitev je, da lahko simuliramo konkreten problem tako, da del pravega opravila poženemo na pravi infrastrukturi in s to emulacijo dobimo podatke o obnašanju, ki jih potrebujemo v simulaciji.

2.7 Ocenjevanje razvrščanja

Pri načrtovanju razvrščevalnikov potrebujemo metrike, ki nam povedo čim več o vedenju in učinkovitosti razvrstitve. Z vrednotenjem istih metrik na različnih opazovanih razvrščanjih omogočimo primerjavo, ki nam pove, kateri razvrščevalnik je v danem okolju opazovanja najboljši, najvarčnejši, najučinkovitejši ipd.

Ocenjevanje razvrščanja je odvisno od tega, kateri cilji so pomembni strankam, vpletenim v rezultate razvrščanja. Uporabniki si želijo hitre odzivnosti, varnosti, zanesljivosti, upoštevanja dogovorov na ravni storitve in pravičnosti pri zagotavljanju virov; ponudniki si želijo visoke energetske učinkovitosti, visoke učinkovitosti infrastrukture in visok finančni izkoristek. Obstaja mnogo metrik, ki se uporabljajo za merjenje učinkovitosti [97, 98, 99], med njimi želimo izpostaviti le nekatere:

Učinkovitost rabe. Rabo je mogoče meriti s povprečno obremenjenostjo virov (npr. obremenjenost CPE skozi čas), ki predstavlja opravljeno delo v določenem času. Metrika je dobra, če je vse delo, ki ga opravljamo, zares koristno. Vendar so lahko v sistemu nekoristna opravila (npr. napaka v programu z neskončno zanko), ki porabijo vozlišča v celoti. V tem primeru metrika kaže visoko učinkovitost, čeprav je dejanska učinkovitost nižja.

Število opravljenih opravil. Število opravljenih opravil je najpreprostejša metrika. Njena glavna pomanjkljivost je ta, da ne razlikuje dobro, če sistem opravlja različno velika opravila. Razvrščevalnik, ki daje prednost manjšim opravilom, jih lahko naredi veliko več kot razvrščevalnik, ki bi dal prednost večjim opravilom.

Čas čakanja. Čas čakanja je skupen čas čakanja vseh zahtev. Čakanje zahteve je čas, ki preteče od takrat, ko je bila zahteva poslana v sistem, do takrat, ko se je opravilo začelo izvajati, ali do takrat, ko se je opravilo končalo. Način, ki meri čas do končanja opravila, je smiseln, kadar sistem omogoča prekinjanje in nadaljevanje opravila. Minimizacija skupnega čakanja poskuša izboljšati uporabniško izkušnjo in je pogosto uporabljena metrika.

Energetska učinkovitost. Energetska učinkovitost [99] je bolj zapletena in jo je mogoče ocenjevati na več načinov. Najosnovnejši način je razmerje med opravljenim delom in porabljeno energijo. Za doseganje dobrih rezultatov mora razvrščevalnik dobro poznati porabo računalnikov pri različnih obremenitvah in pretehtati, kako najbolje obremeniti vozlišča.

Obstaja tudi alternativen predlog [100], ki predlaga ocenjevanje učinkovitosti z razmerjem med energijo, ki je dejansko uporabljena za računanje, in celotno energijo, ki je bila dovedena računskemu centru. Omenjena metrika nam pove, koliko energije izgubimo na preostalih ključnih enotah računskega centra, kot so hlajenje, osvetljevanje prostorov, uravnavanje vlažnosti, omrežje in stikala, brezprekinitveno napajanje ipd.

V nekaterih primerih se energetska učinkovitost ocenjuje brez pravega merjenja energije. Primer take ocene je merjenje časa, ko so računalniki prižgani, v pripravljenosti in ugasnjeni. Seveda ta pristop ni tako natančen kot dejansko merjenje energije, se pa kljub temu pojavi pri raziskavah.

Ekonomska učinkovitost. Pri komercialnih ponudnikih in tudi znotraj podjetij je ključnega pomena ekonomska učinkovitost. Ocenjevanje dobička ponudnikov se oceni s prihodki in stroški, medtem ko uporabnike zanimata strošek na opravljeno koristno delo ali na opravljeno storitev in razvrščanje z omejenimi

sredstvi [101]. Podobne metrike so prisotne pri načrtovanju nabave novih računskih kapacitet, kjer podjetja uporabljajo oceno o donosnosti naložbe, ROI (angl. *return of investment*).

2.8 Zaključek

V tem poglavju smo naredili pregled področij, ki jih potrebujemo za razumevanje prispevkov, predstavljenih v nadaljevanju. Prispevek iz poglavja 3 črpa iz področij omrežnega računalništva, prostovoljnega računalništva in porazdeljenih upraviteljev virov. Prispevek iz poglavja 4 črpa iz področij zasebnih oblakov, virtualizacije, upravljalnikov oblakov in simulatorjev upravljanja virov. Obema prispevkoma je skupna uporaba rezervacij, fleksibilnih opravil, ocenjevanja razvrščanja na podlagi učinkovitosti rabe in delnega dodeljevanja virov.

Poglavje 3

Sočasno delno dodeljevanje infrastrukture v omrežnem računalništvu

Prvi cilj, ki smo si ga zastavili pri izdelavi doktorskega dela, je načrtovati postopek, ki omogoča sočasno dodeljevanje virov, za upravljalnik s porazdeljenim upravljanjem virov. Tak upravljalnik omogoča gradnjo infrastrukture omrežnega računalništva z vozlišči, ki jih ponudijo različne organizacije, in računalniki vsakega uporabnika, če to dovoli. Tako pestro porazdeljeno omrežje zajema lastnosti omrežnega in tudi prostovoljnega računalništva. Zaradi velike količine ponudnikov v sistemu najdemo zelo heterogena vozlišča, na katera nimamo vedno globalnega pogleda. Porazdeljen nadzor nad viri daje sistemu dodatno prožnost pri skaliranju infrastrukture in porastu uporabnikov. Naprednejši upravljalniki omogočajo delno dodeljevanje posameznih vozlišč. S tem omogočajo učinkovitejšo izrabo virov in hkrati predstavljajo večji izziv pri iskanju dobrih razvrstitev opravil na vozlišča.

Sočasno dodeljevanje infrastrukture je ključnega pomena za podporo vzporednim opravilom v sistemih s porazdeljenim delnim dodeljevanjem infrastrukture, vendar je treba pri načrtovanju iskanja in dodeljevanja virov upoštevati, da je množica vozlišč heterogena in delno zasedena. Posledično mora jezik, v katerem se zahteve za opravilo oblikujejo, vsebovati gradnike, s katerimi je mogoče dobro opisati karakteristike vozlišč. Rešitev v tem poglavju uporablja sprotno razvrščanje in optimizira dodeljevanje virov z iskanjem najboljše množice vozlišč za trenutno opravilo. Osnovna ideja postopka je predstavljena v članku [18], podrobnejša razlaga delovanja in analiza skupaj s testi na realni infrastrukturi pa v članku [25].

3.1 Motivacija

V homogenem omrežnem računalništvu, v katerem imajo vozlišča enolično konfiguracijo, se pogosto obravnavajo zahteve, kot je denimo: "Za dvourno opravilo poišči 10 vozlišč, ki imajo po 4 GB pomnilnika." Taka zahteva je mogoče prestroga za heterogena omrežja, ki so sestavljena iz različnih vozlišč. Zelo je verjetno, da bo uporabnik zadovoljen z rešitvijo na zahtevo: "Za dvourno opravilo poišči 10 vozlišč, ki imajo skupaj 40 GB pomnilnika." Ta zahteva je fleksibilnejša in omogoča več rešitev, kar pomeni, da se bo lahko opravilo začelo prej ali pa ob istem času z boljšim izkoristkom sistema.

Glede na predstavljena primera ločujemo med *preprosto zahtevo*, ki vsebuje pogoje le za vsako posamezno vozlišče, medtem ko *kolektivna zahteva* vsebuje vsaj en pogoj, ki določa skupno količino vira brez določitve njegove natančne distribucije po vozliščih.

Preproste zahteve zmore obravnavati večina dobro poznanih sistemov za sočasno dodeljevanje vozlišč, denimo tisti, ki smo jih predstavili v poglavju 2, kot sta OAR [102] in KOALA [87], ter tudi drugi, kot so denimo Gara [103], JSS [104], HARC [105] in GridARS [106]. Pristop opisovanja zahtev z ohlapnejšimi omejitvami ClassAd, ki ga uporablja Condor [107], je dovolj splošen za obravnavanje kolektivnih zahtev. Vendar se njihov primerjalni sistem, ki ga rešujejo kot problem zadovoljevanja omejitev, ne osredotoča na izkoristek sistema. Še en učinkovit sistem za sočasno dodeljevanje vozlišč, ki lahko rešuje le preproste zahteve, je osnovan na intervalnem iskanju za določanje sočasno prostih vozlišč [14]. Če so lahko vozlišča zasedena le delno, je zagotavljanje virov še prožnejše. PlanetLab [33] omogoča deljenje vozlišč, ampak nima takšnega modula za samodejno sočasno dodeljevanje, ki smo ga vajeni v omrežnem računalništvu. OAR [102] na nek način ponuja deljenje vozlišč med uporabniki, ampak le znotraj ene rezervacije. Naprednejšo delno rezervacijo omogoča XtreemOS [77, 84].

Omenjeni sistemi za sočasno dodeljevanje vozlišč imajo pomanjkljivosti, za katere menimo, da jih lahko rešimo, če sistemu za sočasno dodeljevanje vozlišč omogočimo delno rezervacijo virov na vozliščih in uporabimo lokalno optimizacijo, s katero določimo vozlišča, na katera postavimo opravila. Tak pristop zagotavlja manj zavrnjenih opravil in tista opravila, ki so sprejeta, se lahko začnejo prej.

3.2 Definicija vira in sočasno dodeljevanje vozlišč

Kot smo omenili že v uvodu te disertacije, si je mogoče pod terminom *vir* predstavljati različne komponente, ki sistemu omogočajo storitve. V tem poglavju bomo

predpostavljali, da je *vozlišče* najmanjša enota, ki lahko obratuje neodvisno in je opisana s seznamom *virov*. Viri so izraženi v standardnih enotah in kvalitativne lastnosti vozlišč opišemo z vnaprej določenimi oznakami. V značilnem omrežnem sistemu lahko vozlišče opišemo z viri, kot sta količina pomnilnika in frekvenca računalnika, ter kvalitativnimi lastnostmi vozlišč, kot sta arhitektura CPE in različica operacijskega sistema. Dodatno lahko seznam virov in lastnosti vsebuje ceno za izračunavanje, zakasnitve ali širino mrežnih povezav med vozliščem in podatkovnim centrom. Vsako računsko omrežje ima seznam vozlišč in lahko tudi svoj seznam vnaprej določenih virov. Množica vseh virov je označena z $\mathcal{R}$ in količina vira R_i na vozlišču V_j je določena z r_{ij}. Količina vira R_i, rezervirana v času t na vozlišču V_j, je podana z *zasedenostjo* $Z_{ij}(t)$. Problem sočasnega dodeljevanja virov je izbira take množice vozlišč, ki zadosti pogojem izbrane zahteve.

Opravilo je sestavljeno iz aplikacije, vhodnih podatkov in zahteve, ki opisuje potrebno množico vozlišč. V našem primeru *kolektivna zahteva* vsebuje:

- trajanje opravila t_{res}, za katerega potrebujemo rezervacijo;
- število vozlišč N, ki bodo zasedena;
- najzgodnejši in najpoznejši dovoljeni čas začetka, t_{earliest} in t_{latest};
- množico vrednosti d_i, kjer d_i določa minimalno količino vira R_i, prostega na vsakem vozlišču;
- množico vrednosti e_i, kjer e_i določa minimalno skupno količino vira R_i, porazdeljenega čez vseh N vozlišč;
- množico kvalitativnih zahtev.

Obvezno je treba določiti t_{res} in N. Če t_{earliest} in t_{latest} nista določena, je t_{earliest} enak času prihoda zahteve v sistem in $t_{\text{latest}} = t_{\text{earliest}} + t_C$, kjer je t_C vnaprej določena časovna konstanta. Če d_i ni določen, uporabimo majhno pozitivno količino vira. Če e_i ni določen, zahteva postane *preprosta zahteva*, kjer je $e_i = Nd_i$.

Dopustna rešitev problema sočasnega dodeljevanja virov je sestavljena iz začetnega časa t_s in ustrezne množice natanko N vozlišč, ki zadoščajo zahtevi in so na voljo v časovnem bloku, določenem z intervalom $[t_s, t_s + t_{\text{res}}]$.

Učinkovitost vsakega vira R_i v času t na množici vozlišč $\mathcal{V}$ je določena z razmerjem med rezervirano količino vira in vso količino vira:

$$u_i(\mathcal{V}, t) = \frac{\sum_{V_j \in \mathcal{V}} Z_{ij}(t)}{\sum_{V_j \in \mathcal{V}} r_{ij}} \quad . \tag{3.1}$$

Idealno učinkovitost dosežemo, ko je $u_i(\mathcal{V}, t)$ enak 1, medtem ko vrednosti, manjše od 1, izkazujejo odvečno količino vira R_i. Vrednosti, večje od 1, označujejo nedovoljene situacije z izrabo večje količine virov, kot jih je na voljo. Dobro izrabo vozlišč zagotovimo z najvišjimi vrednostmi iz intervala $(0,\ 1]$. Celostno učinkovitost čez vsa vozlišča $\mathcal{V}$ izrazimo s *faktorjem učinkovitosti*:

$$U(\mathcal{V}, t) = \prod_{R_i \in \mathcal{R}} u_i(\mathcal{V}, t) \quad . \tag{3.2}$$

Množenje posameznih učinkovitosti smo izbrali zato, ker daje prednost dopustnim rešitvam z visoko učinkovitostjo $u_i(\mathcal{V}, t)$ močneje kot vsota posameznih učinkovitosti. Faktor učinkovitosti tako spodbuja postavitev novih opravil na vozlišča z visoko učinkovitostjo. Ta strategija lahko vodi do manjšega števila vozlišč v teku in posledično do prihrankov pri energiji.

Dopustna rešitev je tudi optimalna, če ima najzgodnejši začetni čas in nobena druga dopustna rešitev z istim časom nima višjega faktorja učinkovitosti. Čeprav je faktor učinkovitosti pridobljen z množenjem učinkovitosti, je problem optimalnega zadoščanja učinkovitosti še vedno NP-poln problem. Dokaz smo predstavili v prilogi dela [25].

3.3 Algoritem za sočasno dodeljevanje infrastrukture

Na podlagi zahteve nam algoritem pridobi seznam izbranih vozlišč za opravilo in pridobi čas rezervacije, pri vozliščih pa osveži urnike zasedenosti. Če algoritem ni našel rešitve, je seznam vozlišč prazen in čas rezervacije je nedefiniran.

Predlagani algoritem je sestavljen iz šestih faz: v fazi I algoritem od sistema za iskanje virov zahteva začetno množico vozlišč; v fazi II algoritem preveri, ali ima začetna množica vozlišč zadostno količino virov; v fazi III algoritem poišče množico potencialnih začetnih časov; v fazi IV razišče množico začetnih časov in poskuša poiskati najzgodnejšo dopustno rešitev; v fazi V izboljša dopustno rešitev ter končno v fazi VI rezervira vire na vozliščih in oblikuje izhodne podatke.

3.3.1 Faza I – iskanje vozlišč

Algoritem izbere začetno množico $\mathcal{V}_{\text{job}}$ vozlišč, ki ustrezajo vsem omejitvam d_i ne glede na urnike, in jo pošlje fazi II. Za višjo verjetnost uspeha algoritem od sistema za iskanje virov zahteva $M \cdot N$ vozlišč, kjer je $M \in [M_{\text{init}}, M_{\text{max}}]$ za $M_{\text{init}} \geq 1$. Na začetku je M nastavljen na M_{init}, ampak se lahko povečuje na vsakem povratku

iz neuspešnih nadaljnjih faz. Večje vrednosti M povečujejo iskalni prostor in možnosti, da bomo našli rešitev, sočasno pa povečujejo zahtevnost iskanja rešitve in odzivni čas algoritma.

Če je bilo izbranih vsaj N vozlišč, potem algoritem nadaljuje s fazo II. Drugače rešitev ne more biti sestavljena, zahteva se zavrne in algoritem se konča. Algoritem se lahko pozneje iz nadaljnjih faz vrne v fazo I. V tem primeru je opravilo prav tako zavrnjeno, če velja $M > M_{\max}$ ali če algoritem ponovno izbere natanko isto množico vozlišč $\mathcal{V}_{\text{job}}$.

3.3.2 Faza II – preverjanje vozlišč

Ta faza preveri, ali začetna množica vozlišč zagotavlja zadostno količino virov ne glede na njihovo porazdelitev.

Kot prikazuje psevdokoda faza II, najprej uredimo vozlišča po količini vsakega vira R_i v padajočem vrstnem redu in preverimo, ali ima prvih N vozlišč skupno zadostno količino virov glede na zahtevo e_i. Če količina virov ni zadostna za noben vir, se algoritem vrne v fazo I s podvojenim M. V nasprotnem primeru algoritem napreduje na fazo III.

Faza II: Preverjanje vozlišč

```
1 foreach R_i ∈ R do
2     naj bodo V_ω(1), V_ω(2), ..., V_ω(M·N) vozlišča iz V_job,
              urejena po R_i v padajočem redu
3     if Σ_{l=1}^{N} r_iω(l) < e_i then
4         M = 2M
5         pojdi na fazo I
6     end
7 end
```

3.3.3 Faza III – iskanje množice potencialnih začetnih časov

V tej fazi pridobimo množico potencialnih začetnih časov iz urnikov zasedenosti. Kot prikazuje psevdokoda faza III, algoritem najprej izračuna funkcije $a_{ij}(t)$, $A_j(t)$ in $A_j^s(t)$ za vsako vozlišče V_j. Funkcija $a_{ij}(t)$ izraža količino vira $R_i \in \mathcal{R}$ na vozlišču $V_j \in \mathcal{V}_{\text{job}}$ v času t. Funkcija $A_j(t)$ pove, kdaj v času t so količine vira na vozlišču V_j zadostne glede na zahtevo. Funkcija $A_j^s(t)$ ima vrednost 1 natanko takrat, ko je vozlišče V_j prosto v časovnem intervalu $[t, t + t_{\text{res}}]$ brez prekinitev, in 0 v nasprotnem primeru.

Kot prikazuje slika 3.1, s funkcijami $a_{ij}(t)$, $A_j(t)$ in $A_j^s(t)$ pripravimo množico $\mathcal{T}$, ki vsebuje vse mogoče *kandidate začetnih časov*. Opravilo se lahko začne kadarkoli znotraj sivega intervala, ampak moramo preveriti le tiste časovne značke, v katerih količina nekega vira na kateremkoli vozlišču naraste. Upoštevati je treba, da katerokoli povečanje vira na kateremkoli vozlišču lahko vodi do zadostitve pogojev e_i.

Faza III: Iskanje množice potencialnih začetnih časov

1 **foreach** $V_j \in \mathcal{V}_{\text{job}}$ **do**

2 $\quad \forall R_i \in \mathcal{R}\colon a_{ij}(t) = \begin{cases} r_{ij} - Z_{ij}(t) & t \in [t_{\text{earliest}}, t_{\text{latest}} + t_{\text{res}}] \\ 0 & \text{sicer} \end{cases}$

$\quad A_j(t) = \begin{cases} 1 & \forall i\colon d_i \leq a_{ij}(t) \\ 0 & \text{sicer} \end{cases}$

3 $\quad A_j^s(t) = \min_{t' \in [t, t+t_{\text{res}}]} A_j(t')$

4 **end**

5 $\mathcal{T} = \{\, t \,;\, \exists V_j \in \mathcal{V}_{\text{job}}, R_i \in \mathcal{R}\colon A_j^s(t) = 1 \wedge a_{ij}(t) \text{ naraste}\}$

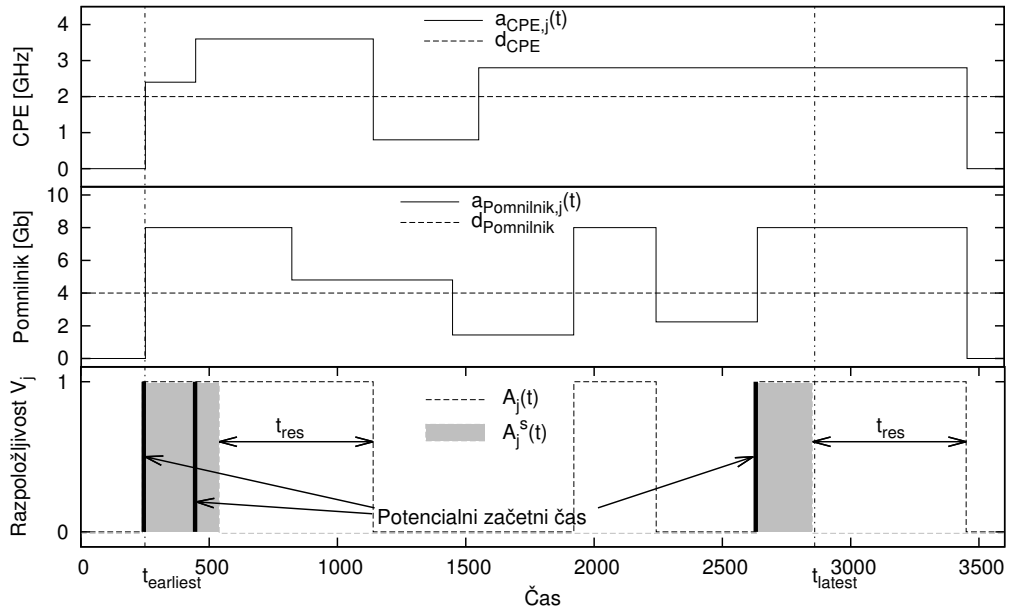
3.3.4 Faza IV – sestavljanje dopustne rešitve

V fazi IV algoritem sestavi dopustno rešitev, osnovano na kar najzgodnejšem začetnem času.

Za vsak čas $t \in \mathcal{T}$, od najzgodnejšega in do najpoznejšega, sestavi množico vozlišč $\mathcal{V}_a$, ki so na voljo v intervalu $[t, t+t_{\text{res}}]$, in izračuna minimalne količine $a_{ij}^t = r_{ij} - Z_{ij}^t$ vira, ki je na voljo na vozlišču skozi celoten interval rezervacije $[t, t+t_{\text{res}}]$, kjer je $Z_{ij}^t = \max_{t' \in [t, t+t_{\text{res}}]} Z_{ij}(t')$ (psevdokoda faza IV, vrstici 2 in 3). Potem poskuša poiskati množico $\mathcal{V}_{\text{sol}} \subseteq \mathcal{V}_a$ iz N vozlišč, ki oblikujejo dopustno rešitev. Če je na voljo več kot N vozlišč, množico $\mathcal{V}_{\text{sol}}$ sestavi funkcija `SelectAndSwap`. Če je množica $\mathcal{V}_{\text{sol}}$ dopustna, algoritem nadaljuje do faze V, sicer pa se vrne v fazo I s podvojeno vrednostjo M.

Funkcija `SelectAndSwap` vrne neprazno množico vozlišč $\mathcal{V}_{\text{sol}}$ samo, kadar je veljaven pogoj $u_i(\mathcal{V}_{\text{sol}}, t') \leq 1$, $t' \in [t, t + t_{\text{res}}]$ za vse vire R_i po tem, ko je opravilo že postavljeno. Da poenostavimo računanje pogoja, preverimo le najslabšo učinkovitost vira

$$\widetilde{u}_i(\mathcal{V}, t) = \frac{e_i + \sum_{V_j \in \mathcal{V}} Z_{ij}^t}{\sum_{V_j \in \mathcal{V}} r_{ij}}. \tag{3.3}$$



Slika 3.1: Izbira potencialnih začetnih časov v fazi III – filtriranje začetnih časov.

Funkcija `SelectAndSwap` najprej poskusi oblikovati dopustno rešitev $\mathcal{V}_{\mathrm{sol}}$ z izbiro N vozlišč, ki imajo visoko količino prostih virov. Ciklično pregleduje vire in v vsakem ciklu izbere vozlišče iz $\mathcal{V}_a$ z najvišjo količino trenutnega vira. Dokler $\mathcal{V}_{\mathrm{sol}}$ ni dopustna, jo algoritem poskuša izboljšati z zamenjavami vozlišč iz $\mathcal{V}_{\mathrm{sol}}$ s tistimi iz $\mathcal{V}_a$ (vrstice 9 do 22). V vsaki iteraciji določi tisti vir R_{ne}, za katerega velja, da ima najbolj nedopustno vrednost učinkovitosti na množici $\mathcal{V}_{\mathrm{sol}}$. Funkcija daje prednost vozliščem $V_m \in \mathcal{V}_a$ z visoko količino vira R_{ne}, medtem ko je izbira vozlišča iz $\mathcal{V}_{\mathrm{sol}}$ naključna. Do zamenjave vozlišč pride le, kadar se nobena količina izmed nezadoščenih virov ne poslabša in noben zadoščeni vir ne postane nezadoščen. Funkcija preneha iskati dopustno rešitev, ko so vsa vozlišča iz $\mathcal{V}_a$ preverjena za isti vir R_{ne}, ali po $L_1 \cdot N$ poskusih, kjer je L_1 parameter algoritma, ki ga nastavlja administrator.

Faza IV: Sestavljanje dopustne rešitve

for $t \in \mathcal{T}$, *urejeni v naraščajočem redu* **do**
  $\mathcal{V}_a = \{V_j \in \mathcal{V}_{\text{job}}; A_j^s(t) = 1\}$
  $\forall V_j \in \mathcal{V}_a, \forall R_i \in \mathcal{R}$: $a_{ij}^t = r_{ij} - Z_{ij}^t$
  switch $|\mathcal{V}_a|$ **do**
    case $< N$: $\mathcal{V}_{\text{sol}} = \emptyset$;
    case $= N$: $\mathcal{V}_{\text{sol}} = \mathcal{V}_a$;
    case $> N$: $\mathcal{V}_{\text{sol}} =$ `SelectAndSwap(`$\mathcal{V}_a$`)`;
  endsw
  if $\forall \mathcal{R}_i \in \mathcal{R} : \sum_{V_j \in \mathcal{V}_{\text{sol}}} a_{ij}^t \geq e_i$ **then**
    pojdi na fazo V
  end
end
$M = 2M$; pojdi na fazo I

---

function $\mathcal{V}_{\text{sol}}$ = `SelectAndSwap(`$\mathcal{V}_a$`)`
$\mathcal{V}_{\text{sol}} = \emptyset$;
for $k = 1$ **to** N **do**
  $k' = (k-1) \bmod |\mathcal{R}| + 1$
  izberi (naključno) V_k iz $\{V_j \in \mathcal{V}_a; \forall V_{j'} \in \mathcal{V}_a : a_{k'j}^t \geq a_{k'j'}^t\}$
  $\mathcal{V}_{\text{sol}} = \mathcal{V}_{\text{sol}} \cup \{V_k\}$; $\mathcal{V}_a = \mathcal{V}_a \setminus \{V_k\}$
end
$c = 1$; $m = 0$
while $(\exists R_i \in \mathcal{R} : \widetilde{u}_i(\mathcal{V}_{\text{sol}}, t) > 1) \wedge (c \leq NL_1) \wedge (m \leq |\mathcal{V}_a|)$ **do**
  $m = m + 1$; $k = 1$
  naj bo R_{ne} naključna lastnost vira iz $\{R_i \in \mathcal{R}; \forall R_{i'} \in \mathcal{R} : \widetilde{u}_i(\mathcal{V}_{\text{sol}}, t) \geq \widetilde{u}_{i'}(\mathcal{V}_{\text{sol}}, t)\}$
  naj bo $V_m \in \mathcal{V}_a$ m-to vozlišče v $V_{\omega(1)}, V_{\omega(2)}, \ldots, V_{\omega(|\mathcal{V}_a|)}$, urejena po R_v v padajočem redu
  naj bodo $V_{\pi(1)}, V_{\pi(2)}, \ldots, V_{\pi(N)}$ vozlišča iz $\mathcal{V}_{\text{sol}}$ v naključnem vrstnem redu
  repeat
    $\mathcal{V}'_{\text{sol}} = \mathcal{V}_{\text{sol}} \cup \{V_m\} \setminus \{V_{\pi(k)}\}$
    $canswap = \bigwedge_{i=1}^{|\mathcal{R}|} \left(\sum_{V_j \in \mathcal{V}'_{\text{sol}}} a_{ij}^t \geq \min(\sum_{V_j \in \mathcal{V}_{\text{sol}}} a_{ij}^t, e_i) \right)$
    if $canswap$ **then**
      $\mathcal{V}_{\text{sol}} = \mathcal{V}'_{\text{sol}}$; $\mathcal{V}_a = \mathcal{V}_a \cup \{V_{\pi(k)}\} \setminus \{V_m\}$; $m = 0$
    end
    $k = k + 1$; $c = c + 1$
  until $(k \leq N) \wedge (c \leq NL_1) \wedge (\neg canswap)$;
end

Naključna menjava omogoča širši pregled čez količine virov na vozliščih, medtem ko ohranjamo algoritem preprost. Naključnost preprečuje, da bi algoritem obstal. Pri determinističnem postopku se namreč lahko zgodi, da se dve vozlišči v dveh zaporednih ponovitvah ponavljajoče izmenjujeta in tako reševanje obstane med dvema nedopustnima rešitvama.

3.3.5 Faza V – optimizacija rešitve

V fazi V algoritem poskusi izboljšati dopustno rešitev z višanjem faktorja učinkovitosti, medtem ko ohranja isti začetni čas (psevdokoda faza V). Ker nas zanima učinkovitost po tem, ko bo opravilo že postavljeno v omrežni sistem, za kriterij uporabimo oceno faktorja učinkovitosti $\widetilde{U}(\mathcal{V}, t) = \prod_{R_i \in \mathcal{R}} \widetilde{u}_i(\mathcal{V}, t)$.

Algoritem naredi $N \cdot L_2$ poskusov izboljšanja dopustne rešitve, kjer je L_2 parameter algoritma, ki ga nastavi administrator sistema. V vsakem poskusu algoritem naključno izbere eno vozlišče iz množice dopustnih rešitev in eno vozlišče iz množice preostalih razpoložljivih vozlišč. Vozlišči se zamenjata le, če nova množica vozlišč sestavlja dopustno rešitev z višjim faktorjem učinkovitosti.

Faza V: Optimizacija rešitve

if $|\mathcal{V}_a| = 0$ **then** pojdi na fazo VI;
for $c = 1$ **to** NL_2 **do**
    naj bo V_{sol} naključno vozlišče iz $\mathcal{V}_{\text{sol}}$
    naj bo V_a naključno vozlišče iz $\mathcal{V}_a$
    $\mathcal{V}'_{\text{sol}} = \mathcal{V}_{\text{sol}} \cup \{V_a\} \setminus \{V_{\text{sol}}\}$
    if $\left(\forall R_i \in \mathcal{R} : \widetilde{u}_i(\mathcal{V}'_{\text{sol}}, t) \leq 1\right) \wedge \left(\widetilde{U}(\mathcal{V}'_{\text{sol}}, t) \geq \widetilde{U}(\mathcal{V}_{\text{sol}}, t)\right)$ **then**
        $\mathcal{V}_{\text{sol}} = \mathcal{V}'_{\text{sol}}$
        $\mathcal{V}_a = \mathcal{V}_a \cup \{V_{\text{sol}}\} \setminus \{V_a\}$
    end
end

3.3.6 Faza VI – rezervacija virov na vozliščih

Na koncu algoritem rezervira vire na vozliščih, ki sestavljajo rešitev, osveži urnike zasedenosti ter vrne začetni čas in seznam vozlišč.

Rezervacijo virov R_i na vsakem vozlišču iz $\mathcal{V}_{\text{sol}}$ naredi algoritem tako, da osveži urnike zasedenosti $Z_{ij}(t') \leftarrow Z_{ij}(t') + d_i + v_i \cdot (e_i - Nd_i)$, kjer je $t' \in [t, t + t_{\text{res}}]$ časovni interval rezervacije in $v_i = (a^t_{ij} - d_i) / \sum_{j=1}^{N} (a^t_{ij} - d_i)$.

Rezervacija je lahko neuspešna, ker so bili med izvajanjem algoritma nekateri viri na vozliščih že zasedeni. V tem primeru se algoritem vrne v fazo I brez spreminjanja parametra M.

3.4 Poskusi in rezultati

Algoritem smo implementirali v operacijski sistem XtreemOS ter ga ovrednotili na infrastrukturi Grid'5000 glede na kakovost rešitve, časovno kompleksnost in režijske stroške operacijskega sistema in infrastrukture.

3.4.1 Implementacija

XtreemOS [60, 77, 84] smo uporabili zaradi njegove modularne arhitekture, ki omogoča preprosto dodajanje predstavljenega algoritma za sočasno dodeljevanje vozlišč v sistem za upravljanje virov. Prav tako je njegova prednost delno dodeljevanje infrastrukture.

Na kratko smo primer uporabe sistema XtreemOS predstavili že v poglavju 2.5.3, v katerem smo opisali glavni komponenti sistema za upravljanje z viri, ki sta upravitelj izvajanja opravil AEM (angl. *Application Execution Manager*) ter sistem za iskanje storitev in vozlišč SRDS (angl. *Service/Resource Discovery Service*). Storitev za sočasno dodeljevanje virov je dodana v upravitelja izvajanja opravil AEM, ki je odgovoren za upravljanje in izbiro vozlišč, ki bodo primerna za poganjanje opravila. Primer uporabe se tako ne spremeni, le sistem AEM z našo razširitvijo zna obravnavati kolektivne zahteve in poiskati učinkovite množice vozlišč med tistimi, ki jih najde komponenta SRDS.

XtreemOS smo namestili na Grid'5000. Da lahko simuliramo uporabo sistema z že delno zasedenimi vozlišči, pripravimo rezervacije vozlišč s podatki za testiranje, ki jih predstavimo v poglavju 3.4.2.

3.4.2 Podatki za testiranje

Odločili smo se, da uporabimo podatke sistema PlanetLab, ki vrednosti obremenitve vsakega vozlišča beleži vsakih 5 minut. Platforma PlanetLab [33] je raziskovalno okolje, sestavljeno iz različnih računskih vozlišč, v katerem lahko vsako vozlišče uporablja več uporabnikov hkrati. Za poskuse smo izbrali 606 vozlišč okolja PlanetLab, za katere so bili na voljo podatki, kot so denimo obremenitev CPE, število jeder CPE in količina pomnilnika. Poskusi so bili osnovani na podatkih o uporabi platforme PlanetLab za delovni dan 18. avgusta 2010, od polnoči do polnoči.

Vir (R_i)	Povprečje ($\bar{r}_i$)	Standardni odklon (σ_i)	Min.	Maks.
Število CPE na vozlišču	2,90	1,51	1	8
Frekvenca CPE [GHz]	2,68	0,36	1,40	3,60
Celotna količina pomnilnika [GB]	2,95	0,84	0,73	7,71

Preglednica 3.1: Statistika virov za 606 vozlišč okolja PlanetLab.

Osnovna statistika, ki opisuje stanje vozlišč, je predstavljena v preglednici 3.1. Iz opisanih podatkov smo sestavili testne primere.

Vsak testni primer je sestavljen iz množice vozlišč PlanetLab, ki so že zasedena glede na podatke o uporabi sistema, in nove zahteve, ki mora biti zadoščena z vozlišči iz te množice.

Trajanje nove zahteve t_{res} smo izbrali iz množice $\{10\,\mathrm{min},\ 30\,\mathrm{min},\ 1\,\mathrm{h},\ 2\,\mathrm{h},\ 3\,\mathrm{h},\ 5\,\mathrm{h}\}$ in število zahtevanih vozlišč N iz množice $\{2,\ 5,\ 10,\ 15,\ 30,\ 60\}$. Najzgodnejši čas začetka t_{earliest} in najpoznejši čas zaključka $t_{\mathrm{latest}} + t_{\mathrm{res}}$ smo izbrali naključno iz intervalov $[0\,\mathrm{h},\ \ 5\,\mathrm{h}\,50\,\mathrm{min}]$ in $[18\,\mathrm{h}\,10\,\mathrm{min},\ \ 24\,\mathrm{h}]$. Vrednost e_i za vsak vir R_i v novi zahtevi smo izbrali naključno iz intervala

$$[N(\bar{r}_i - 2\sigma_i),\ N(\bar{r}_i + 0{,}5\sigma_i)]\,, \tag{3.4}$$

kjer sta $\bar{r}_i$ in σ_i povprečje in standardni odklon virov r_{ij} čez vsa vozlišča. S postavitvijo zgornje meje intervalov na vrednost, višjo od povprečja, smo dovolili, da so količine nekaterih virov težko zadoščene. S kvalitativnimi pogoji smo zahtevali vozlišča AMD Opteron, ki imajo nameščen operacijski sistem XtreemOS.

Za vsako število zahtevanih vozlišč smo pripravili 90 testnih primerov s preprostimi zahtevami in 90 testnih primerov s kolektivnimi zahtevami. Obe različici primerov imata enake zahteve, razen pogoja d_i. Pri preprostih zahtevah smo določili enakomerno porazdelitev vira po vozliščih, $d_i = e_i/N$, medtem ko smo pri kolektivnih zahtevah nastavili $d_i = e_i/2N$, da omejimo preveč neenakomerno porazdelitev vira po vozliščih.

3.4.3 Parametri algoritma

Administrator sistema lahko ureja kakovost rešitev in odzivnost sistema s parametri algoritma M_{init}, M_{max}, L_1 in L_2, ki smo jih opisali v poglavju 3.3. Parametra M_{init} in M_{max} omejujeta začetno in končno velikost prostora preiskovalne množice. Vpliv parametra L_1 se izraža samo v kritičnih primerih, kadar je težko poiskati primeren seznam vozlišč, zaradi visoke zasedenosti omrežnega računskega sistema ali

kompleksne vhodne zahteve. Parameter L_2 uravnava optimizacijo – večje vrednosti omogočajo algoritmu, da porabi več časa za optimiziranje dopustne rešitve, pridobljene v fazi IV, in podaljša čas iskanja.

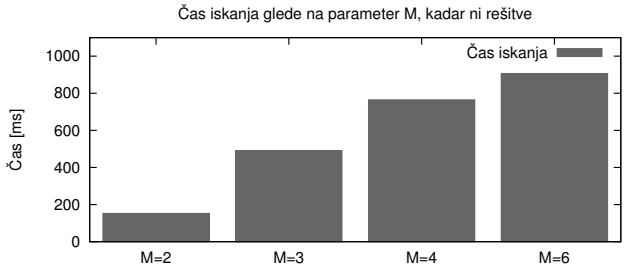
Parametre smo določili glede na statistične podatke iz računskega mesta Nancy, ki je del Grid'5000 [108]. Povprečna zahteva je potrebovala okoli 15 vozlišč in v najbolj obremenjenem tednu je bilo več kot 25.000 rezervacij. V povprečju to pomeni ena uspešna rezervacija na 24 sekund, vendar je med najbolj obremenjenim delom dneva novo zahtevo mogoče pričakovati vsako sekundo.

Za določanje parametra M_{init} smo uporabili množico 15 vozlišč. Da bi zmanjšali učinek interakcij z operacijskim sistemom, smo v testih naredili le en prehod algoritma. V več prehodih se algoritem vrača v fazo I, kar pomeni, da bi v iskalni čas šteli tudi čas komunikacije po mreži med komponentama AEM in SRDS. Parameter M_{init} smo ovrednotili za vrednosti 2, 3, 4 in 6. Rezultati prikazujejo, da $M_{\text{init}} = 3$ daje dober kompromis med kakovostjo rešitve in hitrostjo, saj dobi 50 % več rešitev kot $M_{\text{init}} = 2$ in le 7 % manj kot $M_{\text{init}} = 4$, medtem ko $M_{\text{init}} = 4$ vzame 60 % več časa v primerih brez rešitve (slika 3.2a). Pri vrednostih $M_{\text{init}} > 4$ je izboljšanje premajhno, da bi upravičilo povečanje porabe časa.

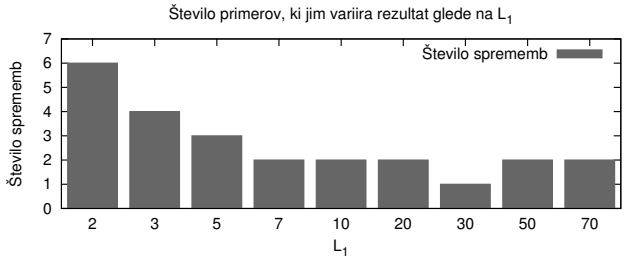
Algoritem je sestavljen iz šestih faz, od katerih prva in zadnja komunicirata s preostalimi moduli operacijskega sistema. Da bi v poskusih omejili vpliv komunikacije in režijske stroške operacijskega sistema, smo upoštevali samo en prehod od faze II do V, torej velja $M_{\max} = M_{\text{init}}$.

Parametra L_1 in L_2 linearno povečujeta časovno porabo faz IV in V (glej fazi IV in V) ter tudi povečujeta verjetnost, da bomo našli najzgodnejšo rešitev z višjo učinkovitostjo. Testirali smo parametra L_1 in L_2 v razponu od 2 do 70 ter ugotovili, da se število najdenih rešitev ali njihova kakovost ne izboljša bistveno pri vrednostih, višjih od $L_1 = 7$ in $L_2 = 10$. Parameter L_1 smo ocenili glede na število primerov, ki jim variirata začetni čas rešitve (slika 3.2b) in porast časa iskanja (slika 3.2c). Stolpci na sliki 3.2b predstavljajo število primerov, ki se jim ob različnih poganjanjih algoritma spreminja začetni čas. Vrednosti, večje od ena, narekujejo, da zaradi premajhnega L_1 ne najdemo rešitev ob najzgodnejšem mogočem času. Slika 3.2c prikazuje čase trajanja za različen L_1. Oba grafa prikazujeta, da je izbira $L_1 = 7$ najboljši kompromis med porabo časa in kakovostjo rešitve. Parameter L_2 smo določili glede na povprečni in maksimalni standardni odklon učinkovitosti (slika 3.2d). Če poganjamo algoritem pri vrednosti $L = 7$, v večini primerov že dobimo vrednosti, ki so blizu najboljših, ki jih algoritem doseže. Ker optimizacija pri zahtevnih primerih ne vzame veliko časa, smo se odločili, da bomo vzeli $L = 10$, saj tako algoritmu omogočimo še nekaj več časa za izboljšanje, kar pride prav pri zahtevah, večjih od 15 vozlišč.

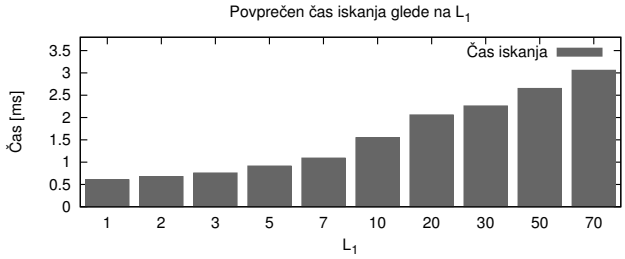
Pri parametrih $M_{\text{init}} = 3$, $L_1 = 7$ in $L_2 = 10$ je iskalni čas algoritma na 2,2 GHz



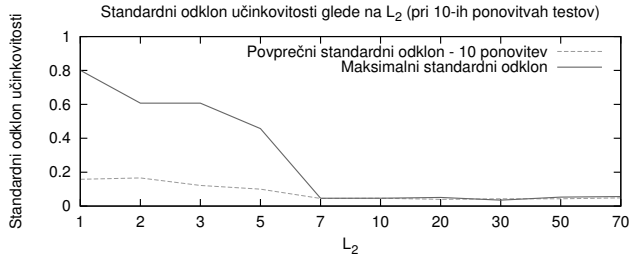
(a) Statistike parametra M



(b) L_1, število sprememb



(c) L_1, iskanje



(d) L_2, odklon učinkovitosti

Slika 3.2: Statistike parametrov M, L_1, L_2.

CPE AMD Opteron 275 za zahtevo s 15 vozlišči v povprečju manj kot 0,1 s in za zahtevo s 60 vozlišči manj kot 1 s. Ti parametri in časi nam zagotavljajo dober odziv sistema tudi v konicah in visoko verjetnost, da se bo algoritem z rezervacijo v fazi 6 uspešno zaključil. Izbrane vrednosti parametrov smo uporabili tudi v poskusih.

3.4.4 Kakovost rešitve

Za ocenjevanje učinkovitosti opisanega algoritma smo njegove izhode primerjali z rešitvami, pridobljenimi s preprostimi zahtevami. Zaradi stohastične narave algoritma smo iskanje vsakega primera ponovili 1000 krat.

Rezultati v preglednici 3.2 prikazujejo, da je preprosti pristop našel rešitev v 27 % testnih primerov, medtem ko je kolektivni pristop uspešnejši, saj je našel rešitev v 60 % vseh testnih primerov. Kolektivni pristop je začel opravila prej v 18 % testnih primerov, v katerih sta oba pristopa dobila rešitev. Natančneje so rezultati pokazali, da so se opravila začela v poprečju 3 ure prej. V testnih primerih, v katerih sta oba pristopa dobila rešitev z istim začetnim časom, kolektivni pristop najde rešitev z višjo učinkovitostjo kot preprosti pristop v več kot trikrat toliko primerih. Podrobna analiza je pokazala, da je relativno izboljšanje faktorja učinkovitosti pri kolektivnem pristopu v povprečju boljše za 8,5 %. Pomembna je tudi opazka, da rezultati ne kažejo korelacije med številom zahtevanih vozlišč in verjetnostjo, da bomo našli rešitev.

Primerjali smo tudi rezultate preprostega in kolektivnega pristopa z optimalno rešitvijo. Zaradi NP-polnosti problema je pristop iskanja optimalne rešitve z grobo silo časovno zelo potraten. Da smo lahko proces izvedli, smo omejili analizo na testne primere, ki zahtevajo le 5 vozlišč.

Pristop z grobo silo najde rešitev v 59 testnih primerih. Preglednica 3.3 prika-

Preglednica 3.2: Primerjava med kolektivnim in preprostim pristopom iskanja v vseh 540 primerih.

	Število zahtevanih vozlišč						
Rešitve najdene s/z:	2	5	10	15	30	60	Σ
Preprostim pristopom	28	27	24	19	27	21	146
Kolektivnim pristopom	49	59	56	55	54	49	322
Zgodnejšim začetkom pri kolektivnem pristopu	4	7	1	7	5	2	26
Enakim začetnim časom	24	20	23	12	22	19	120
Boljšo učinkovitostjo pri preprostem pristopu	0	2	4	5	6	4	21
Boljšo učinkovitostjo pri kolektivnem pristopu	7	13	15	7	16	15	73

Preglednica 3.3: Primerjava sočasnega in preprostega pristopa ter optimalnih rešitev za 5 vozlišč.

	Kolektivni	Preprosti
Najdena rešitev (število)	59	27
Neoptimalen (poznejši) začetni čas (število primerov) / povprečna zakasnitev [h]	3/1,5	7/2,5
Faktor učinkovitosti pri istih začetnih časih (delež od optimalnega faktorja učinkovitosti)	0,92	0,84
Testni primeri, pri katerih je faktor učinkovitosti višji od 99 % optimalnega	22	4

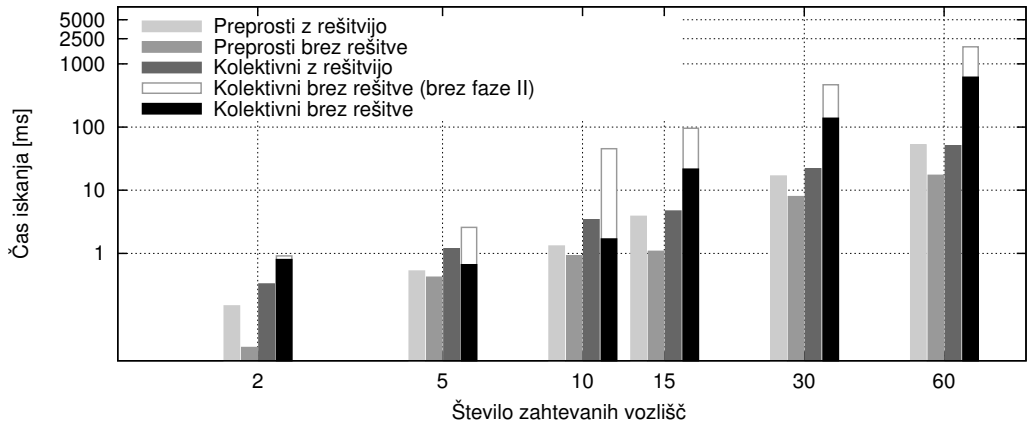
zuje, da je kolektivni pristop uspešen v vseh teh testnih primerih. Samo tri rešitve ne dosežejo optimalnega začetnega časa, medtem ko pri preostalih znaša povprečni faktor učinkovitosti 92 % optimalnega. Podrobna analiza je pokazala, da kolektivni pristop najde 22 testnih primerov s faktorjem učinkovitosti, višjim od 99 % optimalnega. Preprosti pristop je uspešen le v 27 testnih primerih, od katerih jih sedem ne doseže optimalnega časa. Povprečni faktor učinkovitosti znaša 84 % in le v štirih testnih primerih rešitev doseže faktor učinkovitosti, višji od 99 % optimalnega. V splošnem so rezultati, pridobljeni s kolektivnim pristopom, boljši kot tisti, ki so bili pridobljeni s preprostim pristopom.

3.4.5 Časovna zahtevnost

Iskalni čas je odvisen od zasedenosti omrežja, števila zahtevanih vozlišč in zapletenosti zahteve. Povprečni iskalni časi, predstavljeni na sliki 3.3, naraščajo s številom zahtevanih vozlišč. Preprosti pristop je hitrejši, predvsem v testnih primerih brez rešitve. V teh primerih se iskanje preneha v zgodnjih fazah algoritma, ker visoke vrednosti individualnih zahtev algoritmu preprečujejo, da bi našel dovolj vozlišč za oblikovanje rešitve.

Kolektivni pristop zahteva več časa kot preprosti pristop. Večino iskalnega časa kolektivnega pristopa pri testnih primerih brez rešitve porabijo dodatni pregledi za morebitne začetne čase v fazi IV. Začetni časi se povečujejo z večanjem števila zahtevanih vozlišč v zahtevi. Faza II nima vpliva na uspešnost algoritma ali kakovost rešitve, ampak znatno zmanjša povprečen iskalni čas testnih primerov brez rešitve (slika 3.3), zato je dobro, da jo algoritem uporablja. V testnih primerih, ki zahtevajo 5 ali 10 vozlišč, je faza II zavrnila približno polovico testnih primerov brez rešitve.

Razrez iskalnega časa pokaže, da faza II porabi manj kot 1 % in faza III okoli 5 % celotnega iskalnega časa. Preostali čas porabita fazi IV in V. Če je algoritem



Slika 3.3: Povprečni časi iskanja algoritma kot funkcija števila zahtevanih vozlišč.

Preglednica 3.4: Statistika iskalnih časov za 90 testov, ki so zahtevali 5 vozlišč.

	Faze I–VI	Faze II–V
Najmanjša [ms]	126,4	0,008
Največja [ms]	324,9	9,167
Povprečje [ms]	169,8	1,004
Standardni odklon [ms]	29,4	1,664

dosegel fazo IV in je težko poiskati dopustno rešitev, potem faza IV porabi več časa kot faza V. V nasprotnem primeru večino iskalnega časa porabi faza V. Povprečna izboljšava faktorja učinkovitosti v fazi V je okoli 10 %.

3.4.6 Režijski stroški operacijskega sistema in infrastrukture

Da bi lahko predstavili celotno časovno porabo algoritma (vključeni tudi fazi I in VI), smo postavili operacijski sistem XtreemOS na osem vozlišč 2,6 GHz AMD Opteron 2218 in sedem vozlišč 2,2 GHz AMD Opteron 275 omrežja Grid'5000. Upravitelj z viri AEM z našim modulom za sočasno dodeljevanje virov je tekel na enem od vozlišč 2,2 GHz AMD Opteron 2218. V poskusih smo nastavili vse vire in predhodne rezervacije vseh vozlišč z operacijskim sistemom XtreemOS tako, da so odražala enako stanje kot vozlišča, opisana v testnih primerih. Poskus je bil omejen na testne primere, ki so zahtevali 5 vozlišč. Potem ko je algoritem opravil sočasno dodeljevanje, smo urnike zasedenosti vrnili na prvotno stanje, da smo lahko naredili ponoven test. Naredili smo 100 ponovitev vsakega testa.

Časovna poraba operacijskega sistema XtreemOS je predstavljena v preglednici

3.4. Dodaten čas, potreben za fazi I in VI, ki v povprečju znaša skoraj 170 ms, se porabi za mrežno komunikacijo, obdelovanje zahteve in delovanje preostalih modulov, kot so preverjanje uporabnika, nadzorovanje sistema in beleženje akcij.

3.5 Zaključek

V tem poglavju smo predstavili pomanjkljivosti in togost trenutno aktualnih sistemov za sočasno dodeljevanje virov. Nato smo predstavili nov pristop, ki odpravi nekaj pomanjkljivosti in omogoča višjo učinkovitost rabe infrastrukture.

Glavna ideja predlaganega algoritma je razširitev preprostih zahtev za sočasno dodeljevanje virov s kolektivnimi zahtevami, podporo za delno rezervacijo vozlišč in optimizacijo postavitve opravila glede na učinkovitost omrežnega računanja. Algoritem uporablja hevrističen pristop in je razdeljen v šest faz: najprej omeji iskanje na množico vozlišč primerne velikosti, preveri urnike virov na vozliščih iz množice, izbere mogoče začetne čase, išče dopustne rešitve in poskuša poiskati najprimernejšo množico vozlišč. Algoritem ne more zagotavljati optimalne rešitve, ker njeno iskanje lahko traja veliko dlje, kot je uporabnik pripravljen čakati. Da dobimo smiseln kompromis med kakovostjo rešitve in iskalnim časom, se iskanje konča po določenem številu iteracij.

Predlagani algoritem smo implementirali v operacijski sistem XtreemOS. Algoritem je mogoče vključiti v modul za sočasno dodeljevanje infrastrukture kateregakoli omrežnega sistema, s čimer pridobimo podporo kolektivnih zahtev. Delno rezervacijo vozlišč je mogoče uporabiti le, če sistem za upravljanje z viri podpira takšno funkcionalnost.

Poskusi na infrastrukturi Grid'5000 prikazujejo, da algoritem kljub svoji preprostosti v večini primerov najde najboljšo rešitev, če ta obstaja. Algoritem najde dvakrat več rešitev, če namesto preprostih uporabljamo kolektivne zahteve. Tako prispeva k boljši učinkovitosti sistema. Iskalni čas algoritma je najbolj odvisen od števila zahtevanih vozlišč in trenutne zasedenosti računalnikov v omrežju.

Z vsakim klicem algoritem naredi optimizacijo učinkovitosti omrežja na podmnožici vseh vozlišč. Ker se ta množica spreminja vsakič, ko se algoritem požene, lokalna optimizacija z visoko verjetnostjo vodi do globalne optimizacije izrabe omrežja.

V prihodnje želimo raziskati tendence uporabe omrežnega računalništva in na podlagi trenutnega stanja v omrežju ugotoviti verjetnost obstoja rešitve za dano zahtevo sočasnega dodeljevanja virov. Z boljšim ocenjevanjem omrežja želimo izboljšati zmogljivost iskanja. Poleg omenjenega bi lahko z vpeljavo možnosti ponovnega razvrščanja rezervacij dosegli še boljšo izrabo in učinkovitost ter tudi omogočanje

reševanja napak in izpadov vozlišč.

Poglavje 4

Učinkovita izraba infrastrukture s sobivanjem in vertikalnim skaliranjem

Drugi prispevek doktorske disertacije je postopek, ki iz množice opravil poišče tista opravila, ki so najprimernejša za sočasno izvajanje na vozlišču, in jih potem preoblikuje tako, da vozlišče kar najbolje izkoristijo. Pristop prav tako uporablja prilagodljiva opravila, le da ima sedaj globalen pogled na stanje vozlišč in lahko preoblikuje več opravil hkrati. Postopek na ta način izboljša učinkovitost izrabe in je primarno namenjen za upravljanje infrastrukture zasebnih oblakov. S predlagano rešitvijo smo razširili delovanje orodja Haizea in s simulacijami izmerili količino izboljšanja.

4.1 Motivacija

Oblačno računalništvo zmanjšuje stroške delovanja tudi s centralizacijo računalniške infrastrukture v velike podatkovne centre, v katerih je infrastrukturo mogoče učinkoviteje izrabiti in upravljati. Javni oblaki ponujajo uporabnikom navidezno neskončno količino virov brez vnaprejšnjih investicij in s predvidljivimi stroški uporabe. Zasebni oblaki so bolj prilagodljivi in se pogosto uporabljajo, kadar sta zaradi občutljivosti podatkov potrebna popoln nadzor in neodvisnost od zunanjih ponudnikov.

Lastniki zasebnih oblakov želijo s kar najmanjšimi vložki v infrastrukturo opraviti kar največ dela. Ker so zasebni oblaki bolj omejeni z infrastrukturo, je precej

verjetneje, da bo ob konicah prišlo do pomanjkanja virov. Naše delo na zagotavljanju virov v zasebnem oblaku se osredotoča na opravila, ki so fleksibilna glede na čas izvajanja in jih lahko prestavljamo v določenih časovnih mejah ter tudi glede potrebe po virih. Opravila, ki so fleksibilna glede na količino virov, lahko prilagajajo trajanje z vertikalnim ali horizontalnim skaliranjem ter jim rečemo prilagodljiva (poglavje 2.3.2).

Občasno pomanjkanje virov lahko rešimo z razširjanjem oblaka (angl. *cloud bursting*), ki začasno najame dodatne vire iz javnega oblaka, ali pa problem rešimo s časovnim odlogom opravil prek rezervacij in vrst. Razširitev oblaka povzroči dodatne stroške ter zmanjša prednosti na področju varnosti in zasebnosti, ki jo omogoča zasebni oblak. Časovni odlog je dovoljen samo za nekatere vrste opravil; delovanje spletnega strežnika, ki mora biti ves čas na voljo, denimo ne more biti odloženo. Mi se v tem poglavju osredotočamo na sisteme, ki so osnovani na urnikih in za vsako opravilo potrebujejo vnaprej poznano trajanje. V zameno lahko zagotovijo točen začetek in zaključek opravila, česar nekateri sistemi z vrsto ne morejo zagotoviti [109]. Kadar trajanje opravil ni znano, je primernejša uporaba sistema vrst; te imajo tudi manj težav z učinkovitostjo, ki je posledica drobljenja vrzeli v urniku [110].

Opravila, ki v celoti izkoristijo posamezno vozlišče ali skupino vozlišč v oblaku, so redka. Pogosta mešanica opravil, ki teče na splošno-namenski infrastrukturi, vsebuje nekaj opravil, omejenih z zmogljivostjo CPE, nekaj opravil, omejenih z zmogljivostjo V/I-enote, in nekaj opravil, ki porabijo le delčke razpoložljivih virov. Visoko učinkovitost virov in posledično ekonomično uporabo zasebnega oblaka lahko dosežemo le z deljenjem posameznih vozlišč med več opravil. Za visoko učinkovitost morajo na vozliščih sobivati opravila s komplementarnimi potrebami [22, 23, 111]. Dve opravili imata komplementarne potrebe takrat, ko eno opravilo na vozlišču zahteva visok delež prvega vira in nizek delež drugega, medtem ko drugo opravilo uporablja nizek delež prvega vira in visok delež drugega vira. Prvi doprinos v tem poglavju je izbira opravil, primernih za sobivanje, glede na njihove potrebe in razpoložljive vire.

Pomemben korak pri optimizaciji učinkovitosti je upoštevanje prilagodljivosti opravil, kar pomeni, da se lahko razvrščevalnik odloči, kakšne kapacitete virov bo zagotovil za opravilo. Ko se izvajanje opravila dejansko začne, se namreč količina virov ne more več spreminjati. Večina opravil, ki so omejena s CPE, in tudi precej drugih je vertikalno skalabilnih glede na svoje potrebe po virih in trajanju. Vertikalna skalabilnost pomeni, da lahko opravilu dodajamo vire, kot sta moč CPE in količina pomnilnika na istem vozlišču, zato da se bo opravilo končalo prej. To velja le do neke meje in prav tako velja, da se bo opravilo izvajalo počasneje z manj viri ali pa bo celo spodletelo zaradi premajhne količine virov. Viri, na katere se osredo-

točamo v tem poglavju, so zmogljivost CPE, količina pomnilnika in pasovna širina V/I-povezav. Seveda je mogoče pristop razširiti tudi na druge.

Naš pristop izkoristi možnost vertikalnega skaliranja opravil za izboljšanje sobivanja. Drugi prispevek je tako usmerjanje skaliranja opravil. Bistvo pristopa lahko ponazorimo na primeru dveh komplementarnih opravil, katerih skupna zahteva po virih presega razpoložljive vire. Z vertikalnim skaliranjem teh opravil lahko njuni skupni zahtevi zmanjšamo na razpoložljivo količino, pri čemer pa podaljšamo njuno trajanje. Obratno, če opravili skupaj ne izrabita vozlišča v celoti, ju skaliramo tako, da porabita več virov. Na koncu lahko tudi opravila, ki niso primerna za sobivanje, skaliramo tako, da kar najbolje ustrezajo prostim virom na infrastrukturi.

Skaliranje opravil na način, ki smo ga opisali zgoraj, je mogoče le, če imamo vnaprej določene meje, v katerih se opravilo lahko skalira, in funkcijo, ki opisuje odvisnost trajanja od virov, ki so dodeljeni opravilu. V splošnem naš pristop deluje za katerokoli monotono funkcijo, vendar se bomo v tem poglavju omejili in predpostavljali, da obstaja linearna zveza med danimi mejami skaliranja. To pomeni, če opravilo omogoča skaliranje za faktor najmanj dva, potem mu lahko dodelimo dvakrat tolikšno zmogljivost CPE, V/I-pasovno širino in količino pomnilnika, ocena trajanja opravila pa se bo razpolovila. Modeliranje natančne funkcije za oceno trajanja je težavno, ker skaliranje lahko vpliva na pogostost zadetkov v predpomnilniku, čakanje sinhronizacije V/I-naprave ipd. Tovrstni učinki se pripišejo režijskim stroškom delovanja virtualizacije in se pri določanju trajanja opravila ocenijo skupaj. Pristopi, ki zmanjšajo vpliv nekaterih učinkov, so predstavljeni v poglavju 2.2.5. Odločili smo se, da je upoštevanje omenjenih izzivov pri modeliranju funkcije trajanja problem, ki presega meje našega dela, saj večina sistemov, ki temeljijo na urnikih in dovoljujejo sobivanje opravil, trpi zaradi omenjenih težav ne glede na to, ali so opravila vertikalno skalabilna ali ne.

Izboljšanje učinkovitosti s skaliranjem se je že uporabljalo v preteklosti pri uporabi vrst in rezervacij. Konkreten primer je izdelava protiponudb, ki znotraj razpoložljivosti virov v urniku izbrane infrastrukture poskušajo ustreči uporabnikovim zahtevam [57]. Omenjeni pristop s pogajanjem lahko izboljša učinkovitost le s skaliranjem zadnjega opravila ob določenem času. Naslednji pristop k izboljšanju učinkovitosti je skaliranje opravila pred izvajanjem, čemur rečemo kalupljenje opravil (angl. *molding*) [24], ki se uporablja predvsem za horizontalno skaliranje opravil glede na količino razpoložljivih virov [47]. Skaliranje pred izvajanjem se deli na dva pristopa. Prvi je skaliranje ob času prihoda opravila v sistem, ki je manj učinkovit, saj se s prihodom novih opravil stanje spreminja, ter drugi, učinkovitejši, je skaliranje opravila v sklopu razvrščanja [47]. Skaliranje opravil je lahko tudi namerna ali nenamerna posledica sobivanja opravil, ki tekmujejo za isti nabor virov. Če je količina virov nezadostna, se bo trajanje podaljšalo. To lahko rešimo

z občasno migracijo opravil med vozlišči in iskanjem najboljšega sobivanja, ki se izvaja ciklično po vsakem časovnem intervalu [94, 23]. Slaba stran je, da migracija porablja dodatne vire. Poleg tega se pri tem dodeljena infrastruktura med trajanjem opravila spreminja, kar je lahko prednost in tudi slabost.

Naša predlagana rešitev skalira opravila pred razvrščanjem in jo je tako mogoče implementirati kot predprocesor za obstoječe razvrščevalnike. Postopek uporabi hevristiko za izbiro in skaliranje opravil tako, da lahko učinkovito tečejo drugo ob drugem in maksimirajo učinkovitost izrabe med izvajanjem. Po skaliranju se količina dodeljenih virov ne spreminja. Ta pristop k sobivanju izboljša učinkovitost izrabe manjše skupine opravil, ki pripomore k izboljšani izrabi celotnega urnika.

Pristop je primeren za vsak sistem, na katerem tečejo skalabilna opravila, ki sestavlja urnik izvajanja z rezervacijami in podpira delno dodeljevanje virov. Taki so denimo zasebni oblaki, pa tudi nekateri sistemi omrežnega računalništva, kot denimo XtreemOS [25]. Osnovna različica predlaganega predprocesorja razvrščevalniku dostavi množico skaliranih opravil, saj večina razvrščevalnikov pričakuje vhod v taki obliki. Splošni razvrščevalniki niso zmožni interpretirati informacije o tem, katera opravila naj sobivajo. Predprocesor zato opravila, ki naj sobivajo, združi v sestavljeno opravilo.

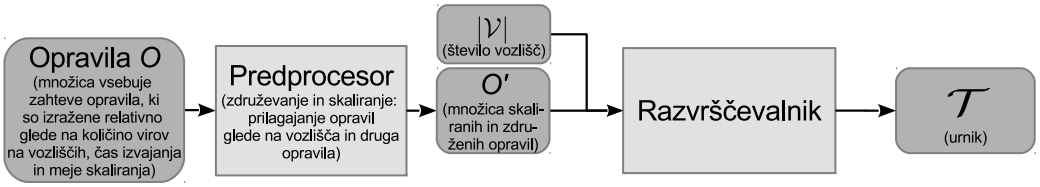
Toda kot smo predstavili v razdelku 2.5.1, je delovanje razvrščevalnikov mogoče prikrojiti. Za njih je lahko primernejša dopolnjena različica predprocesorja, ki razvrščevalniku opravila poda v obliki razbitja. S tem razvrščevalniku predlaga, katera opravila sodijo skupaj za optimalno razvrščanje.

V nadaljevanju bomo najprej predstavili formalno definicijo problema za pristop z množico. Poglavje 4.3 razloži delovanje predlaganega postopka in njegove omejitve za pristop z množico. Poglavje 4.4 predstavi pripravo opravil za razvrščevalnik in primerja pristop z množico s pristopom z razbitjem. Poglavje 4.5 predstavi poskuse in rezultate, opis prispevka pa zaključimo s poglavjem 4.6.

4.2 Razvrščanje opravil in izkoristek virov

Razvrščevalniki preslikajo vhodna opravila na vozlišča, a večina jih zna narediti urnik le s togimi opravili. Zato predlagamo predprocesor, ki izkoristi zmožnost fleksibilnih opravil. Tako opravila in informacijo o infrastrukturi najprej pregleda predprocesor, ki pripravi opravila in jih pošlje razvrščevalniku (slika 4.1). Nato razvrščevalnik sestavi urnik glede na omejitve infrastrukture. Ker mora predprocesor delovati za širok nabor razvrščevalnikov, ne more uporabljati povratne zanke. V splošnem velja, da je predprocesor učinkovitejši, če lahko prilagaja večjo količino opravil sočasno, zato opravila obdeluje v paketih. Delovanje predprocesorja

vodi hevristika. Naš cilj je narediti predprocesor, ki omogoča, da razvrščevalnik ustvari učinkovitejši urnik in posledično izboljša učinkovitost rabe infrastrukture. Kakovost predprocesorja merimo posredno z merjenjem učinkovitosti urnika.



Slika 4.1: Integracija predprocesorja.

Definirajmo računalniško infrastrukturo kot $|\mathcal{V}|$ enakovrednih vozlišč z viri $r \in \mathcal{R}$, kjer je $\mathcal{R} = \{\mathrm{CPE}, \mathrm{RAM}, \mathrm{V/I}\}$. Opravilo $o = \langle D_o, t_o, p_o \rangle$ je definirano z zahtevami po virih ter omejitvami $D_o = \{(d_{o,r}, d_{o,r}^{\mathrm{MIN}}, d_{o,r}^{\mathrm{MAX}}) | \forall r \in \mathcal{R}\}$, trajanjem t_o in številom vzporednih opravkov p_o, ki sestavljajo opravilo. D_o določa zahteve $d_{o,r}$ po vsakem viru r ter najnižjo $d_{o,r}^{\mathrm{MIN}}$ in najvišjo $d_{o,r}^{\mathrm{MAX}}$ količino vira r, ki jo opravilo lahko doseže s skaliranjem. Torej velja $d_{o,r}^{\mathrm{MIN}} \leq d_{o,r} \leq d_{o,r}^{\mathrm{MAX}}$. Zahteve $d_{o,r}$ in meje prilagodljivosti $d_{o,r}^{\mathrm{MIN}}$, $d_{o,r}^{\mathrm{MAX}}$ so izražene relativno na količino vira r na vsakem vozlišču na infrastrukturi, tako da velja $d_{o,r}, d_{o,r}^{\mathrm{MIN}}, d_{o,r}^{\mathrm{MAX}} \in (0, 1]$. V primeru $d_{o,r}^{\mathrm{MIN}} < d_{o,r}^{\mathrm{MAX}}$ je opravilo *prilagodljivo opravilo*. Vzporedno opravilo je tisto, ki ima število vzporednih opravkov $p_o > 1$ in je ekvivalentno p_o enakim opravilom z omejitvijo, da se morajo vsa opravila začeti ob istem času in vsako od teh trajati natanko t_o. Ne nazadnje, t_o definira trajanje opravila, če vsak od p_o opravkov opravila o dobi natanko $d_{o,r}$ vsakega vira $r \in R$.

Razvrščevalnik preslika množico opravil O in število vozlišč $|\mathcal{V}|$ v urnik $\mathcal{T}$. Predpostavljali bomo, da razvrščevalnik ne more skalirati opravil ter zato ne upošteva vrednosti $d_{o,r}^{\mathrm{MAX}}$ in $d_{o,r}^{\mathrm{MIN}}$, ki sta namenjeni izključno za predprocesor. Urnik $\mathcal{T}$ je preslikava, ki vsakemu opravilu $o \in O$ dodeli množico p_o vozlišč v časovnem razponu od začetnega časa $t_{o,s} \geq 0$ do končnega časa $t_{o,s} + t_o$. Urnik je veljaven, če ob kateremkoli času velja, da je vsota zahtev vseh opravil, ki sočasno tečejo na kateremkoli vozlišču, manjša ali enaka 1.

Predprocesor je funkcija, ki vzame množico opravil O in pripravi množico opravil O', kjer je $|O'| \leq |O|$. Ta funkcija preveri opravila in jih skalira s prilagajanjem zahtev $d_{o,r}$ znotraj mej skaliranja $d_{o,r}^{\mathrm{MIN}}$ in $d_{o,r}^{\mathrm{MAX}}$. Skaliranje prilagaja opravila posamezno ali v skupinah tako, da se kar najbolje prilagajajo virom, ki jih ponuja infrastruktura. Skaliranje in prilagajanje opravil v skupinah bomo naslavljali z izrazom združevanje. Skupine opravil, za katere predprocesor meni, da bi bilo dobro, da sobivajo, predprocesor nadomesti z enim skupnim združenim opravilom, zato

velja $|O'| \leq |O|$.

Učinkovitost urnika $\mathcal{T}$ merimo z vsoto zaključnih časov na vsakem vozlišču

$$t_{\text{RUN}}(\mathcal{T}) = \sum_{k=1}^{N} t_{k,f} \ , \tag{4.1}$$

kjer je $t_{k,f}$ čas zadnjega zaključka kateregakoli opravila, razvrščenega na vozlišču k v urniku $\mathcal{T}$. Učinkovitost vozlišč je višja, če je enaka količina opravil opravljena v krajšem času.

Naša naloga je torej definirati predprocesor, ki pretvori množico opravil O v množico opravil O' tako, da bo razvrščevalnik sestavil učinkovitejši urnik.

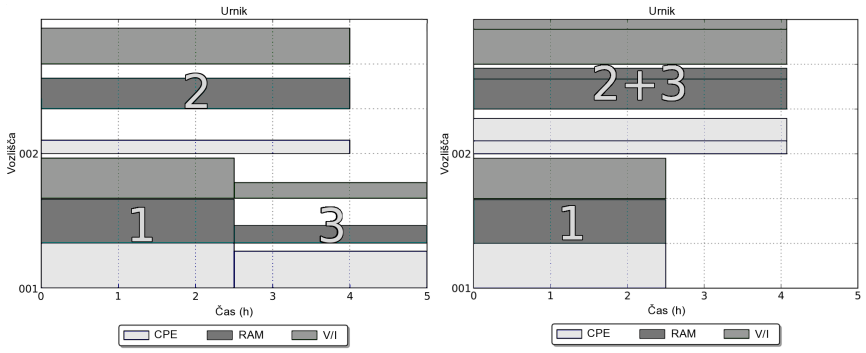
4.3 Združevanje in skaliranje opravil

Iskanje optimalnega združevanja opravil je časovno zahtevno, saj bi morali preveriti vse mogoče združitve opravil in razvrščanja v urnik. Zato predlagani algoritem vsebuje hevristiko, ki obravnava združevanje le tistih opravil, ki z višjo verjetnostjo vodijo do koristnega sobivanja. Uporaba hevristike znatno zmanjša število poskusov združevanja in tako zmanjša časovno zahtevnost algoritma, medtem ko je odločitev o sobivanju nezdruženih opravil prepuščena razvrščevalniku.

Hevristika sledi dvema smernicama skaliranja in združevanja. Prvič, usmerja algoritem, da združuje le tista opravila s komplementarnimi potrebami in podobno dolžino trajanja, saj bodo ta lahko učinkovito izrabila vse tri vire na vozliščih. Drugič, usmerja algoritem, da skalira opravila tako, da popolnoma izkoristijo vsaj en vir r na vozliščih in se zato končajo čim hitreje.

Algoritem združuje le tista opravila, pri katerih je velika verjetnost, da bo njihovo sobivanje koristno. Tak primer prikazujeta opravili 2 in 3 na sliki 4.2, kjer je očitno, da se lahko opravili brez skaliranja izvajata le zaporedno ali na dveh vozliščih (levo), medtem ko se po skaliranju in združitvi lahko izvajata sočasno na enem vozlišču (desno). Opravil, ki nimajo komplementarnih zahtev, algoritem ne združuje, saj ne more vedeti, ali bo sobivanje koristno.

Skica algoritma je prikazana na sliki 4.3. Algoritem začne s klasifikacijo opravil glede na njihove potrebe po virih (korak 1). Potem izbere, združi in skalira popolnoma komplementarna opravila (korak 2). Preostala nezdružena opravila doda delno komplementarnim opravilom, ki jih združi in skalira v naslednjem koraku (korak 3). Rezultat vsake uspešne združitve je množica opravil, primerna za sobivanje na istem vozlišču. Na koncu skalira vsako nezdruženo opravilo tako, da porabi celo vozlišče, če je to mogoče (korak 4). Rezultat procesa združevanja je



Slika 4.2: Razvrščanje treh opravil na dve vozlišči brez (levo) in s skaliranjem (desno). Po skaliranju je učinkovitost izboljšana, ker je skupno trajanje opravil na obeh vozliščih krajše.

razbitje $O'' = \{\{o'_1\}, \{o'_2, o'_3\}, \ldots\}$, ki vsebuje podmnožice opravil za sobivanje in skaliranih opravil.

Slika 4.3: Skica algoritma za združevanje in skaliranje.

4.3.1 Iskanje komplementarnih opravil

Algoritem najprej razdeli opravila na devet razredov, ki ločujejo opravila po razmerju zahtevanih virov $d_{o,r}$ glede na zahtevo po viru CPE (slika 4.3, korak 1). To omogoči preprostejše iskanje komplementarnih opravil. Za klasifikacijo izračuna kvalitativna razmerja zahtev porabe pomnilnika in V/I-enote glede na zahtevo po

CPE

$$c_r(o) = \begin{cases} \text{lt}\,; & & d_{o,r}/d_{o,\text{CPE}} < 1-\alpha \\ \text{eq}\,; & 1-\alpha \leq & d_{o,r}/d_{o,\text{CPE}} \leq 1+\alpha \\ \text{gt}\,; & 1+\alpha < & d_{o,r}/d_{o,\text{CPE}} \end{cases}, \tag{4.2}$$

kjer je $0 \leq \alpha < 1$ parameter algoritma, ki določa mejo klasifikacije. Glede na vrednosti kvalitativnih razmerij so oblikovani razredi

$$C_{q_1,q_2} = \{o \in O \mid c_{\text{RAM}}(o) = q_1 \wedge c_{\text{V/I}}(o) = q_2\}. \tag{4.3}$$

Vsaka oznaka razreda C_{q_1,q_2} predstavlja nizko (lt), podobno (eq) in visoko (gt) zahtevo RAM ali V/I glede na zahtevo vira CPE. Npr., $C_{\text{gt,gt}}$ vsebuje opravila z visoko zahtevo po virih RAM in V/I glede na zahtevo po CPE in $C_{\text{eq,eq}}$ vsebuje le opravila s podobno zahtevo po vseh virih.

4.3.2 Združevanje popolnoma komplementarnih opravil

Opravila iz razreda $C_{\text{gt,gt}}$ so popolnoma komplementarna opravilom iz razreda $C_{\text{lt,lt}}$ in v enaki relaciji sta tudi razreda $C_{\text{lt,gt}}$ in $C_{\text{gt,lt}}$. Algoritem v svojem koraku 2 iz vsakega od teh razredov izbere po eno opravilo, obe dobljeni opravili pa poskusi združiti. Za lažjo razlago bomo razlikovali par komplementarnih razredov tako, da ju naslavljamo kot razred A in razred B.

Pred združevanjem dveh razredov algoritem uredi opravila, najprej po številu vzporednih opravkov p_o (padajoče) in nato še po $d_{o,r_{\text{MAX}}}$ (razred A padajoče in razred B naraščajoče), kjer je r_{MAX} vir, ki ga na podlagi zahtev iz množice O na infrastrukturi najbolj primanjkuje. Torej, r_{MAX} je katerikoli r', za katerega velja

$$\forall r \in R : \sum_{o \in O} (d_{o,r'} \cdot t_o) \geq \sum_{o \in O} (d_{o,r} \cdot t_o).$$

Nato algoritem vzame prvo opravilo o_{A} iz razreda A in prvo opravilo o_{B} iz razreda B, ki ima enako število vzporednih opravkov $p_{o_{\text{A}}} = p_{o_{\text{B}}}$. Vsak tak komplementaren par poskusi združiti in skalirati. Nato nadaljuje z naslednjim opravilom iz razreda A. Če ni nobenega opravila o_{B}, ki bi imelo enako število vzporednih opravil, poskusi opravilo o_{A} združiti z več opravili iz razreda B, kot opišemo v nadaljevanju. Velja tudi obratno, posamezno vzporedno opravilo iz razreda B lahko združi s skupino opravil iz razreda A, če ni primernega opravila o_{A}.

Združevanje nevzporednih opravil

Najprej opišemo združevanje za preprostejši primer nevzporednih opravil $p_{o_\mathrm{A}} = p_{o_\mathrm{B}} = 1$. Algoritem najprej preveri, ali sta si obe opravili podobni po velikosti w_o, ki predstavlja celotno porabo vira, ki najbolj omejuje opravilo, med celotnim trajanjem opravila:

$$w_o = t_o \cdot \max_{r \in \mathcal{R}}(d_{o,r}) \ . \tag{4.4}$$

Opravili združujemo le, če sta w_{o_A} in w_{o_B} dovolj podobna, torej če velja

$$\frac{\max(w_{o_\mathrm{A}}, w_{o_\mathrm{B}})}{\min(w_{o_\mathrm{A}}, w_{o_\mathrm{B}})} < W \ , \tag{4.5}$$

kjer je podobnost $W > 1$ parameter algoritma.

Če velikosti ustrezata, potem algoritem skalira opravili tako, da je njuno trajanje najkrajše in lahko tečeta sočasno na enem vozlišču. Če je trajanje najdaljšega od tako dobljenih opravil daljše, kot bi bilo trajanje obeh zaporedno zloženih opravil, algoritem opravil ne združi.

Skaliranje posameznega opravila

Opravilo o' je skalirana oblika opravila o. Skaliranje opravila vpliva na trajanje opravila in vse njegove zahteve po virih, ki so omejene z mejami:

$$\forall r \in \mathcal{R} \quad : \qquad d_{o,r}^{\mathrm{MIN}} \leq d_{o',r} \leq d_{o,r}^{\mathrm{MAX}} \quad . \tag{4.6}$$

Algoritem vedno skalira opravila linearno po vseh virih:

$$\forall r_1, r_2 \in \mathcal{R} \quad : \qquad d_{o',r_1}/d_{o',r_2} = d_{o,r_1}/d_{o,r_2} \quad . \tag{4.7}$$

Tak pristop je potreben za nekatere vrste opravil, kot je MapReduce, ki samodejno zažene toliko opravkov, kot je potrebno za popolno izrabo virov na prostih jedrih CPE [112], zato povečanje zmogljivosti CPE (in števila dodeljenih jeder) zahteva tudi povečanje RAM. Druga opravila lahko delujejo bolje že samo z izboljšanjem enega ali dveh virov. Fizikalne simulacije se denimo izvajajo hitreje že, če jim povečamo količino CPE in V/I. V našem pristopu se opravila skalirajo z linearno spremembo vseh virov, četudi jih nekatera opravila ne izrabijo v celoti. To naredi postopek preprostejši in dovoljuje pravičnejšo primerjavo z razvrščevalniki, ki ne znajo skalirati opravil, saj linearno skaliranje ohranja količino produkta $d_{o,r} \cdot t_o$.

Trajanje se skalira obratno sorazmerno z vsakim virom, kar lahko izrazimo kar z uporabo $d_{o',\mathrm{CPE}}$:

$$t_{o'} = t_o \cdot d_{o,\mathrm{CPE}}/d_{o',\mathrm{CPE}}. \tag{4.8}$$

Skaliranje in združevanje para opravil

Skaliranje dveh opravil o_{A} in o_{B} tako, da se lahko izvajata sočasno na istem vozlišču, mora zadostovati pogojem skaliranih opravil o'_{A} in o'_{B}:

$$\forall r \in \mathcal{R} \quad : \qquad d_{o'_{\mathrm{A}},r} + d_{o'_{\mathrm{B}},r} \leq 1. \tag{4.9}$$

Optimalna rešitev je par $o'_{\mathrm{A}}, o'_{\mathrm{B}}$, ki minimizira skupno trajanje t_{pair}, ki je enako trajanju daljšega skaliranega opravila:

$$\begin{aligned} t_{\mathrm{pair}}(d_{o'_{\mathrm{A}},\mathrm{CPE}}, d_{o'_{\mathrm{B}},\mathrm{CPE}}) &= \max(t_{o'_{\mathrm{A}}}, t_{o'_{\mathrm{B}}}) \\ &= \max\left(t_{o_{\mathrm{A}}} \cdot \frac{d_{o_{\mathrm{A}},\mathrm{CPE}}}{d_{o'_{\mathrm{A}},\mathrm{CPE}}}, t_{o_{\mathrm{B}}} \cdot \frac{d_{o_{\mathrm{B}},\mathrm{CPE}}}{d_{o'_{\mathrm{B}},\mathrm{CPE}}}\right). \end{aligned} \tag{4.10}$$

V idealnem primeru bosta skalirani opravili porabili vse vire na vozlišču (za vsak r imamo enakost v enačbi 4.9) in velja $t_{o'_{\mathrm{A}}} = t_{o'_{\mathrm{B}}}$. Kot že rečeno, združevanje se ne zgodi, če je trajanje para skaliranih opravil daljše, kot če bi izvajali vsako posebej zaporedno.

Skaliranje in združevanje vzporednih opravil

Združevanje dveh opravil z enakim številom vzporednih opravkov je popolnoma enako kot združevanje dveh nevzporednih opravil.

Algoritem lahko združi vzporedno nezdruženo opravilo o_{A} z drugo množico nezdruženih opravil $\{o_{\mathrm{B},1}, \ldots, o_{\mathrm{B},i}\}$, če velja $\sum_{l=1}^{i} p_{o_{\mathrm{B},l}} = p_{o_{\mathrm{A}}}$. Naj bo $o_{\mathrm{B},1}$ opravilo z najvišjim p_o iz razreda B, torej velja $\forall l \in \{2, \ldots, i\} : o_{\mathrm{B},1} \geq o_{\mathrm{B},l}$. Algoritem skalira opravili o_{A} in $o_{\mathrm{B},1}$ po enakem postopku kot za združevanje nevzporednih opravil. Če je združevanje uspešno, ostane nekaj vozlišč zapolnjenih le z opravilom o_{A}. Na ta vozlišča poskusi algoritem dodati še skalirana opravila $\{o_{\mathrm{B},2}, \ldots, o_{\mathrm{B},i}\}$ tako, da ustrezajo preostalim virom na vozliščih. Pri tem velja, da njihovo trajanje po skaliranju ne sme presegati trajanja skaliranih opravil o_{A} in $o_{\mathrm{B},1}$.

Če vzporednost opravila in množice opravil ni enaka, $\sum_{l=1}^{i} p_{o_{\mathrm{B},l}} \neq p_{o_{\mathrm{A}}}$, nikoli ne združujemo, saj bi v mnogih primerih pokvarili urnik. Naj opozorimo, da lahko razvrščevalnik nezdružena opravila še vedno postavi skupaj.

4.3.3 Združevanje delno komplementarnih opravil

Ko dosežemo korak 3, smo že izrabili vse možnosti združevanja ustreznih popolnoma komplementarnih opravil. Zato ponovno klasificiramo nezdružena opravila

$C^*_{q_1,q_2} \subseteq C_{q_1,q_2}$ iz koraka 2 tako, da se bodo lahko združevala z delno komplementarnimi opravili:

$$\begin{aligned} C'_{\mathrm{lt,eq}} &= C_{\mathrm{lt,eq}} \cup (C^*_{\mathrm{lt,lt}} \cup C^*_{\mathrm{lt,gt}}), \\ C'_{\mathrm{eq,lt}} &= C_{\mathrm{eq,lt}} \cup (C^*_{\mathrm{lt,lt}} \cup C^*_{\mathrm{gt,lt}}), \\ C'_{\mathrm{eq,gt}} &= C_{\mathrm{eq,gt}} \cup (C^*_{\mathrm{lt,gt}} \cup C^*_{\mathrm{gt,gt}}), \\ C'_{\mathrm{gt,eq}} &= C_{\mathrm{gt,eq}} \cup (C^*_{\mathrm{gt,lt}} \cup C^*_{\mathrm{gt,gt}}). \end{aligned} \tag{4.11}$$

Potem združujemo opravila iz razreda $C'_{\mathrm{eq,gt}}$ z opravili iz razreda $C'_{\mathrm{eq,lt}}$ ter opravila iz razreda $C'_{\mathrm{lt,eq}}$ z opravili iz $C'_{\mathrm{gt,eq}}$. Proces združevanja v tem koraku je enak kot pri združevanju popolnoma komplementarnih opravil.

4.3.4 Skaliranje preostalih opravil

V končnem koraku 4 obravnavamo vsa nezdružena opravila. Ta vsebujejo tudi celoten razred uravnovešenih opravil $C_{\mathrm{eq,eq}}$. Če je mogoče, opravila skaliramo tako, da porabijo celotno vozlišče, kar pomeni, da bo po skaliranju maksimalen $d_{o',r}$ imel vrednost 1. Opravila, pri katerih to ni mogoče, ohranimo nespremenjena.

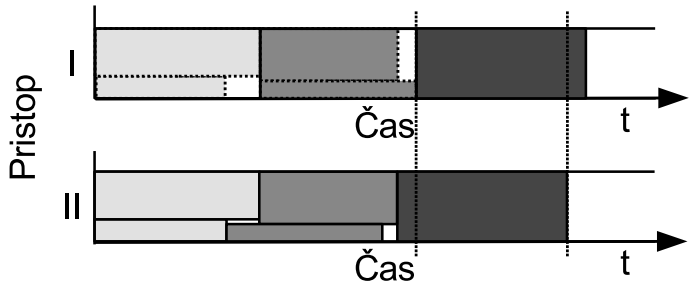
4.4 Priprava opravil za razvrščevalnik

Rezultat predprocesiranja je razbitje O'', torej množica množic takšnih opravil, ki naj bi za optimalno delovanje delovali sočasno. Splošni razvrščevalniki pa na svojem vhodu sprejemajo množice opravil. Zato moramo opis zahtev prilagoditi.

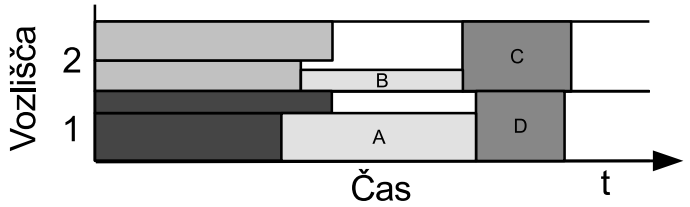
Predprocesor nadomesti vsako podmnožico v razbitju z enim nadomestnim opravilom in s tem zagotovi, da bodo opravila, za katere meni, da morajo sobivati, skupaj. Tako omogoča uspešno delovanje z večjo množico različnih razvrščevalnikov. Podmnožica opravil z enako vzporednostjo dobi nadomestno opravilo, ki ima trajanje enako najdaljšemu opravilu, zahteve pa so enake vsoti zahtev posameznih opravil. Podmnožica opravil z različno vzporednostjo dobi nadomestno opravilo $o'_{\mathrm{A,B}}$, ki ima trajanje $t_{o'_{\mathrm{A,B}}} = t_{\mathrm{pair}}$, potrebe po virih $d_{o'_{\mathrm{A,B}},r} = \max_{l \in \{1,\ldots,i\}}(d_{o'_{\mathrm{A}},r} + d_{o'_{\mathrm{B},l},r})$ in $p_{o'_{\mathrm{A,B}}} = \max(p_{o_{\mathrm{A}}}, p_{o_{\mathrm{B}}})$.

Če lahko vplivamo na notranje delovanje razvrščevalnika, lahko razvrščevalniku pustimo več prostosti pri postavitvi opravil na vozlišča. V tem primeru opravila pošljemo z razbitjem $O'' = \{\{o'_1\}, \{o'_2, o'_3\}, \ldots\}$, kjer vsaka podmnožica vsebuje opravila, ki naj bi sobivala. Na ta način razvrščevalnik ureja opravila skupaj tako, kot je predprocesor ocenil za optimalno, a ima kljub temu možnost, da jih v urnik postavi ločeno. S tem odstranimo omejitev, da se morajo združena opravila začeti sočasno

in na istih vozliščih, ter lahko razvrščevalnik zapolni nekatere vrzeli v urniku (slika 4.4a) ali pa naredi nove (slika 4.4b). Nove vrzeli se naredijo, kadar razvrščevalnik postavi opravila na vozlišča tako, da druga opravila ne morejo dobro izkoristiti nastalih vrzeli. Kako dobro lahko razvrščevalnik izkoristi prostost pri postavljanju opravil, je odvisno od politik razvrščanja in izbire vozlišča. Zato razvrščevalnik poleg razbitja na vhodu potrebuje še definicije politik, ki usmerjajo njegovo delovanje tako, da pravilno obravnavajo opravila v razbitju. Slika 4.5 prikazuje razširjeno shemo integracije predprocesorja.



(a) Pristop z množico (I) in razbitjem (II). Primer razvrščanja, ko so združena opravila predstavljena z nadomestnim opravilom (I) ali brez (II).

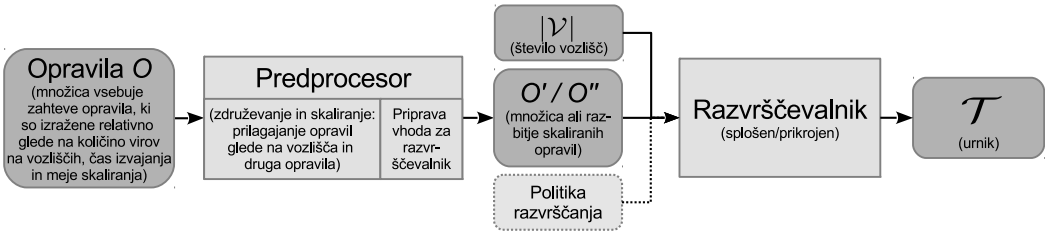


(b) Pri pristopu z razbitjem lahko razvrščevalnik opravili A in B uvrsti na različni vozlišči ter se zato opravili C in D začneta pozneje, kot če bi bili opravili A in B skupaj na vozlišču 1.

Slika 4.4: Primer razvrščanja, kadar so združena opravila predstavljena z nadomestnim opravilom ali brez.

4.5 Poskusi in rezultati

Predlagani pristop smo ovrednotili s poskusi. Izboljšanje učinkovitosti izrabe (enačba 4.1) smo ovrednotili s primerjavo učinkovitosti urnikov, ki jih dobimo iz množice O' ali razbitja O'' združenih in skaliranih opravil ter osnovne množice O.



Slika 4.5: Razširjena shema integracije predprocesorja.

Poskusi so osnovani na dveh referenčnih razvrščevalnih strategijah. Prva je daljše opravilo najprej (LJF, angl. *longest job first*), ki da prednost opravilom z daljšim trajanjem in teži k maksimizaciji izrabe sistema [113]. Druga je najvišja zahteva najprej (HDF, angl. *highest demand first*), ki daje prednost opravilom z višjimi vrednostmi $d_{o,r}$. Ko razvrščevalnik izbere opravilo o, obe razvrščevalni strategiji postopata enako. Opravilo postavita v urnik z metodo zasipanja z izbiro vozlišča z najboljšim ujemanjem. To pomeni, da razvrščevalnik najde najzgodnejšo vrzel, v katero je mogoče vstaviti opravilo. Če je na voljo več kot p_o prostih vozlišč, razvrščevalnik izbere tisto kombinacijo, ki bo po umestitvi opravila med trajanjem opravila imela na umeščenih vozliščih najvišjo učinkovitost.

Razvrščanje v razbitjih deluje tako, da se vsaka podmnožica razbitja obravnava kot združeno opravilo, nato pa se uredi še opravila znotraj podmnožice. Tako velja, da je po urejanju l-to združeno opravilo iz množice O' sestavljeno iz opravil, ki jih po urejanju vsebuje l-ta podmnožica razbitja O''. Opravila znotraj vsake podmnožice v obeh primerih razvrščanja uredimo po načinu višja zahteva najprej.

4.5.1 Implementacija

Za testiranje kot primer uporabnosti smo predlagani pristop implementirali v programskem okolju Python kot dodatek za razvrščevalnik Haizea [69], ki smo ga že predstavili v poglavju 2.5.2. Za svoje potrebe smo morali le razširiti jezik za opis opravil, ki smo mu dodali minimalne in maksimalne vrednosti zahteve po virih. Za minimizacijo funkcije t_{pair} (enačba 4.10) smo uporabili programski paket OpenOpt [114].

4.5.2 Množice testnih opravil

Za preverjanje različnih scenarijev uporabe zasebnega oblaka in testiranje stabilnosti tega pristopa smo uporabili vrsto različnih generiranih podatkov, saj prosto dostopne zbirke podatkov [115, 116] vsebujejo le število zahtevnih vozlišč, brez

podatkov o deležu vira ali njegovi skalabilnosti. Pripravljene množice opravil so vsebovale 100 %, 66 %, 33 % in 0 % popolnoma uravnovešenih opravil. Popolnoma uravnovešeno opravilo je tisto, ki potrebuje enak delež vsakega vira, torej za množico popolnoma uravnovešenih opravil velja $\forall o \in O, \forall r, r' \in \mathcal{R} : d_{o,r} = d_{o,r'}$. Pri tem je treba omeniti, da predlagana hevristika popolnoma uravnovešena opravila zgolj skalira na cela vozlišča, kadar je to mogoče, nikoli pa jih ne skuša združevati.

Količino posameznega vira pri vsakem opravilu smo izbrali naključno iz množice vrednosti $d_{o,r} \in \{0{,}2,\ 0{,}4,\ 0{,}6,\ 0{,}8,\ 1{,}0\}$. Diskreten nabor vrednosti omogoča pravičnejše vrednotenje kot izbira vrednosti iz zveznega območja. Ta bi povzročila drobljenje virov na manjše kose, kar bi povzročilo velike težave razvrščevalnikom, ki opravil ne morejo skalirati vertikalno. Pri vseh neuravnovešenih opravilih so bile vrednosti za zmogljivost CPE, pomnilnik in širino V/I-povezav izbrane neodvisno.

Območje skalabilnosti vsakega posameznega vira pri opravilu $d_{o,r}^{\text{MAX}}/d_{o,r}^{\text{MIN}}$ obsega med 1,25 do 5 s povprečjem pri 3,5, kjer je $d_{o,r}^{\text{MIN}} \in [0{,}15,\ 0{,}8]$ in $d_{o,r}^{\text{MAX}} \in [0{,}2,\ 1]$. Polovica testnih množic vsebuje samo nevzporedna opravila. V drugi polovici opravil je število vzporednih opravkov enakomerno porazdeljeno znotraj meja $p_o \in [1,\ 0{,}3|\mathcal{V}|]$, kjer je $|\mathcal{V}|$ število vozlišč. Za $|\mathcal{V}| = 5$ smo uporabili mejo $p_o \in [1,\ 2]$, saj drugače ne bi imeli vzporednih opravil.

Trajanja opravil so bila enakomerno porazdeljena v razponu $4{,}5 \pm 2{,}5$ h. Skupna količina zahtev vseh opravil je bila sorazmerna kapaciteti virov, ki jo oblak lahko ponudi v 24 urah. Tako je število opravil v vsaki množici odvisno od števila vozlišč in ali množica vsebuje vzporedna opravila ali ne (glej preglednico 4.1). Pri množicah z vzporednimi opravili število opravil ne raste s številom vozlišč.

Preglednica 4.1: Statistika posameznih vhodnih množic opravil.

Število vozlišč	Število opravil v množicah brez vzporednih opravil [interval]	Število opravil v množicah z vzporednimi opravili [interval]	Število vzporednih opravkov [interval]
5	[36, 42]	[25, 32]	[1, 2]
15	[112, 125]	[37, 46]	[1, 4]
30	[215, 240]	[38, 49]	[1, 9]
100	[689, 752]	[44, 52]	[1, 30]

4.5.3 Izbira parametrov algoritma

S prvimi cikli poskusov smo poiskali primerne parametre α in W, ki se uporabljajo v enačbah 4.2 in 4.5. Pri tem smo uporabili učne množice, pripravljene po istem postopku kot za teste.

Pri izbiri α velja, da bo večja vrednost usmerila algoritem, da bo klasificiral več opravil v razred uravnovešenih, in tako jih ne bomo združevali. Po drugi strani večji W omogoča, da bomo parili skupaj opravila tudi, če se njihova velikost bolj razlikuje. Vpliv parametrov smo preverili na učnih množicah s 5, 15 in 30 vozlišči, ampak brez popolnoma uravnovešenih opravil, kjer izbira parametra nima vpliva. Poskusi z vrednostmi parametrov $\alpha \in [0,\ 0{,}5]$ in $W \in [1{,}1,\ 20]$ so pokazali, da učinkovitost znatno pade pri vrednostih $\alpha > 0{,}3$ in $W > 3$. Za bolj fino nastavljanje parametrov smo pognali vsak učni primer z različnimi kombinacijami $\alpha = \{0,\ 0{,}1,\ 0{,}2,\ 0{,}3\}$ in $W = \{1{,}1,\ 1{,}25,\ 1{,}5,\ 1{,}75,\ 2,\ 2{,}5,\ 3\}$. Za vsako kombinacijo smo preverili povprečno učinkovitost čez vse učne primere in tudi učinkovitost pri posameznih primerih. Rezultati so pokazali, da vrednosti $\alpha = 0{,}1$ in $W = 1{,}75$ dajeta optimalne rezultate pri večini primerov pri pristopu z množico in tudi pri pristopu z razbitjem.

Dodatno smo želeli potrditi, da je algoritem stabilen ne glede na manjše spremembe parametrov. Tako smo pognali algoritem za vsak posamezni primer posebej pri vsaki kombinaciji parametrov za fino nastavljanje. Stabilnost na posamičnem primeru smo ovrednotili z relativnim standardnim odklonom učinkovitosti. Pri nobenem učnem primeru ni bil maksimalni relativni standardni odklon višji od $1{,}6\,\%$, povprečje čez vse učne primere pa je znašalo $1{,}1\,\%$ pri pristopu z množico, pri pristopu z razbitjem vrednosti narasteta na $2{,}4\,\%$ in $1{,}2\,\%$.

4.5.4 Ocena uspešnosti

Uspešnost algoritma smo ovrednotili na 128 testnih množicah opravil, izbrali smo po dve iz vsakega od 64 različnih testnih scenarijev.

Pristop z množico

Naš pristop z množico je v primerjavi z referenčnim razvrščevalnikom učinkovitejši v 115 od 128 primerov in enako učinkovit v treh testnih primerih (preglednica 4.2 in slika 4.6). Pri preostalih desetih primerih velja, da vsi vsebujejo vzporedna opravila in šest od teh je iz primerov s 100 vozlišči, kjer predprocesor težje najde primerna opravila za združevanje. Razloga za to sta nizko število opravil glede na število vozlišč in visoka raznolikost v številu vzporednih opravkov pri opravilih (preglednica 4.1). Kadar je pri teh primerih združevanje uspešno, predprocesor pogosto sestavi

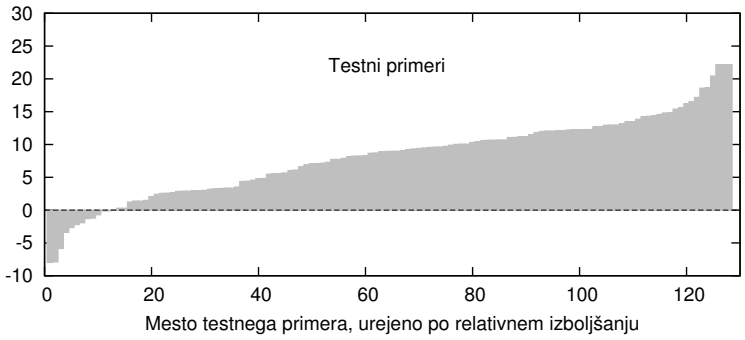
združena opravila z različnimi vzporednostmi vhodnih opravil. Tako ima končna množica O' nižje število vzporednih opravil kot O, vendar pa je njihovo povprečje $p_{o'}$ višje, kar pomeni, da jih je težje razvrščati učinkovito.

Izboljšanje učinkovitosti predlaganega pristopa z množico glede na referenčne razvrščevalnike je v večini med 7 % in 13 % pri množicah brez vzporednih opravil ter med 1 % in 11 % pri primerih, ki vsebujejo tudi vzporedna opravila (slika 4.7), s povprečjem pri 8 % (preglednica 4.2). Rezultati na sliki 4.7 so prikazani kot funkcija števila vozlišč in so združeni po strategiji razvrščanja in vsebnosti vzporednih opravil v testnih množicah. Najpoznejši čas zaključka med vsemi opravili je v povprečju za 1 h 45 min zgodnejši pri predlaganem pristopu (glej preglednico 4.2). Zgodnejše čase zaključka dosežemo pri 103 testnih primerih.

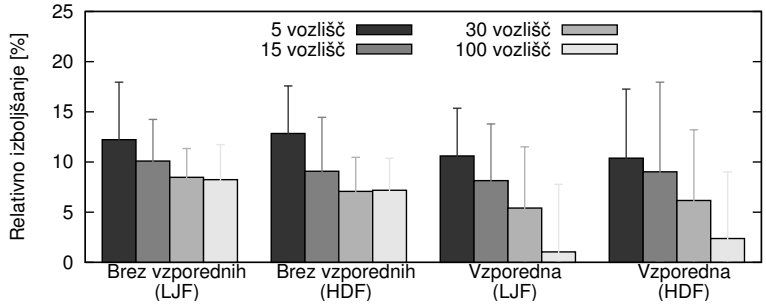
Preglednica 4.2: Povzetek rezultatov pri pristopu z množicami.

Število primerov brez izboljšanja (nižja in enaka učinkovitost)	13 (10 + 3)
Izboljšanje učinkovitosti, povprečje (odklon)	8,0 % (6,0 %)
Izboljšanje zaključka zadnjega opravila, povprečje (odklon)	1 h 45 min (2 h 20 min)
Število primerov brez izboljšanja zaključka zadnjega opravila	25

Izboljšanja učinkovitosti so manjša, a stabilnejša pri primerih, ki vsebujejo več opravil in imajo na voljo več vozlišč. Morebitni razlog za to je, da je pri večji infrastrukturi več primernih kombinacij za razvrščanje opravil in tako lahko tudi referenčni razvrščevalnik preprosteje zgradi razmeroma dober urnik. Rezultati prikazujejo podobno izboljšanje med tehnikama razvrščanja LJF in HDF (slika 4.7).



Slika 4.6: Rezultati, urejeni po izboljšanjih pri pristopu z množicami.



Slika 4.7: Izboljšanje učinkovitosti kot funkcija števila vozlišč pri pristopu z množicami. Vsak stolpec prikazuje povprečje in standardni odklon osmih testnih primerov.

Po drugi strani rezultati pri množicah z vzporednimi opravili prikazujejo višje odklone in nizke izboljšave na večjih infrastrukturah (slika 4.7). Razlog za visoke odklone je v razvrščanju opravil z visokim deležem vzporednih opravkov. V teh primerih lahko ena manjša sprememba razvrščanja znatno vpliva na učinkovitost, saj pri nepravilni razvrstitvi vzporednih opravil v urniku ostanejo vrzeli neizrabljenih vozlišč. Te nastanejo, kadar sočasno za opravilo ni na voljo dovolj vozlišč. Tako lahko zelo podobni množici opravil z visokimi vrednostmi p_o razvrstimo v urnik z manj vrzelmi neizrabljenih vozlišč, kar nam da zelo učinkovit urnik, ali v urnik z veliko vrzelmi neizrabljenih vozlišč, kar naredi urnik neučinkovit. Omeniti je treba, da imajo množice vzporednih opravil razmeroma visoko število vzporednih opravil. Z zmanjšanjem ravni vzporednosti p_o bi rezultati množic z vzporednimi opravili postali stabilnejši in izboljšave bi bile verjetno podobne kot pri množicah brez vzporednih opravil.

Pristop z razbitjem

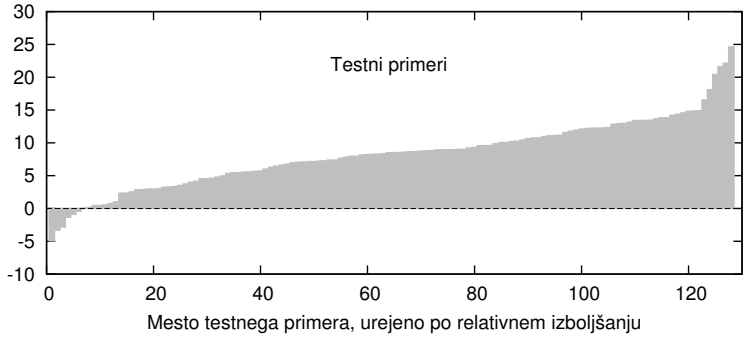
Poskus uspešnosti algoritma z razbitjem pri prenosu podatkov v razvrščevalnik smo naredili pod enakimi pogoji kot pri uporabi množic. Rezultati prikazujejo, da uporaba manj omejujočega pristopa izboljša učinkovitost v 120 testnih primerih in doseže enako učinkovitost v dveh testnih primerih (preglednica 4.3 in slika 4.8). Število izboljšanj je višje kot pri pristopu, ki združena opravila zamenja z nadomestnim opravilom (preglednica 4.2 in slika 4.6).

Izboljšanje učinkovitosti pristopa z razbitjem glede na referenčne razvrščevalnike je v večini med 4 % in 12 % pri vseh testnih množicah (slika 4.9) s povprečjem pri 8,2 % (preglednica 4.3). Rezultat je boljši kot pri pristopu brez uporabe nadomestnih opravil, prav tako so rezultati stabilnejši, kar prikazujejo nižji standardni

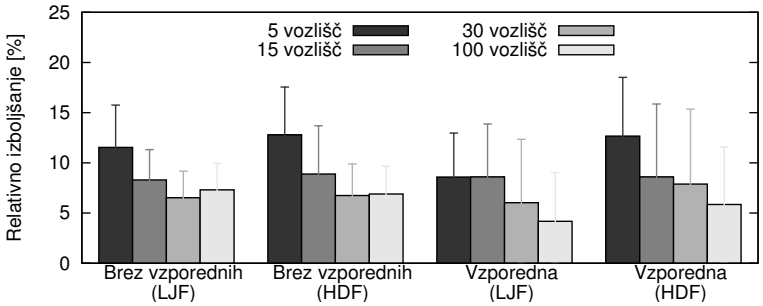
Preglednica 4.3: Povzetek rezultatov pri pristopu z razbitjem.

Število primerov brez izboljšanja (nižja in enaka učinkovitost)	8 (6 + 2)
Izboljšanje učinkovitosti, povprečje (odklon)	8,2 % (5,3 %)
Izboljšanje zaključka zadnjega opravila, povprečje (odklon)	2 h 14 min (1 h 43 min)
Število primerov brez izboljšanja zaključka zadnjega opravila	12

odkloni (preglednica 4.3). Najpoznejši čas zaključka med vsemi opravili v povprečju znaša 2 h 14 min manj kot pri referenčnem pristopu brez predprocesorja, kar je slabe pol ure boljši rezultat kot pri pristopu z množico. Zgodnejše čase zaključka dosežemo pri 116 testnih primerih (preglednica 4.3). V splošnem je večina rezultatov boljših kot pri pristopu, ki uporablja nadomestna opravila.



Slika 4.8: Rezultati, urejeni po izboljšanjih pri pristopu z razbitjem.



Slika 4.9: Izboljšanje učinkovitosti kot funkcija števila vozlišč pri pristopu z razbitjem. Vsak stolpec prikazuje povprečje in standardni odklon osmih testnih primerov.

Še vedno velja, da so izboljšanja manjša in stabilnejša pri testnih primerih z večjim številom vozlišč (slika 4.9). Opazimo lahko, da tehnika razvrščanja HDF daje boljše rezultate kot LJF. Razlika je očitnejša pri vzporednih primerih. Če primerjamo z rezultati, ki smo jih dobili pri pristopu z množico, se v povprečju učinkovitost pri tehniki razvrščanja LJF zmanjša, pri HDF pa poveča pri vzporednih primerih. Razlog je v tem, da je pri tehniki razvrščanja LJF veliko večja možnost, da se pri razvrščanju pojavi primer napačnega obravnavanja opravila (slika 4.4b). Pri tehniki razvrščanja HDF in primerih brez vzporednih opravil pa dobimo enake rezultate kot s pristopom z množico. Ob pregledu urnikov smo ugotovili, da se prednost, predstavljena na sliki 4.4a, ne more zgoditi pogosto. Veljati mora, da sta dve skupini združenih opravil eno za drugim na istem vozlišču, opravila znotraj skupine morajo imeti različna trajanja ter opravila se morajo pravilno (komplementarno) ujemati v porabi vseh treh virov tudi med prvo in drugo skupino opravil. Pomagalo bi, če bi bil razvrščevalnik zmožen prepoznati taka združena opravila in jih razvrstiti zaporedno na ista vozlišča.

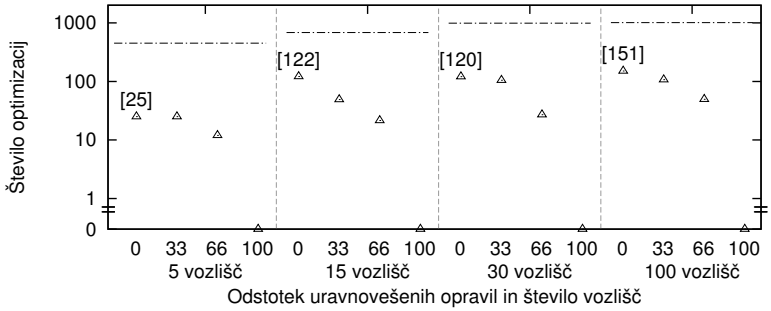
Pristop z razbitjem pri množicah z vzporednimi opravili dobi višja izboljšanja učinkovitosti v primerjavi s pristopom z množico. Eden od razlogov je, da ta pristop ne spremeni povprečne vrednosti p_o v množici z vzporednimi opravili, saj se opravila znotraj množice še vedno obravnavajo posamezno. Opravila so za razvrščevalnik manjša in jih je lažje razvrščati.

4.5.5 Poraba časa in časovna kompleksnost združevanja

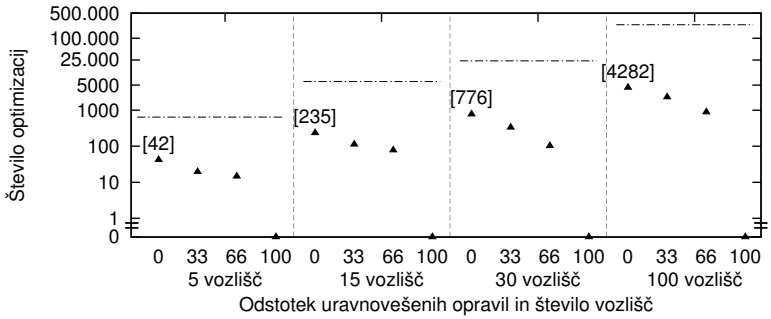
Čas, potreben za predprocesiranje in razvrščanje, smo izmerili na procesni enoti Intel Xeon E5-2620 2,00 GHz. Najbolj časovno potraten del združevanja in skaliranja je minimizacija t_{pair} v enačbi (4.10), kar pomeni, da število optimizacij narekuje časovno porabo predprocesiranja. Kompleksnost posamezne optimizacije je konstantno.

V najzahtevnejšem primeru so vsa opravila klasificirana enakomerno v dva komplementarna razreda ter algoritem neuspešno poskuša poiskati pare v korakih 2 in 3 (poglavji 4.3.2 in 4.3.3). V vsakem razredu je torej $|O|/2$ opravil, in če bi v obeh korakih poskusili združevati vse z vsemi, bi izvedli $2 \cdot |O|^2/4$ optimizacij. V najpreprostejšem primeru so vsa opravila uravnovešena in potem ni združevanja. Pri najpreprostejšem primeru z neuravnovešenimi opravili, pri katerih bi imeli le uspešne združitve, imamo največ $|O|/2$ optimizacij. Naš pristop omeji, katera opravila se poskusi združevati, in s tem naredi znatno manjše število optimizacij v primerjavi s teoretično maksimalno vrednostjo. Število optimizacij za vzporedna in nevzporedna opravila je predstavljeno grafično na slikah 4.10a in 4.10b, kjer vodoravne prekinjene črte prikazujejo teoretični maksimum. Kot je pričakovano, predlagani

pristop opravi več optimizacij na množicah z večjim številom opravil in manj optimizacij na množicah z večjim odstotkom uravnovešenih opravil. Znatna razlika v številu optimizacij pri procesiranju množic z vzporednimi in nevzporednimi opravili je posledica različnega števila opravil v množicah.



(a) Vzporedna opravila



(b) Nevzporedna opravila

Slika 4.10: Število optimizacij združevanja, vodoravna prekinjena črta prikazuje teoretično maksimalno število optimizacij.

Združevanje in skaliranje porabi okrog minute za 100 optimizacij, osem minut za 700 optimizacij in 50 minut za 4200 optimizacij. V naših testnih primerih je združevanje potrebovalo do 55 minut, kar je še vedno veliko manj kot prihranek pri času t_{RUN}, ki ga dobimo z izboljšanimi urniki.

Zahtevnost zasipanja narašča s številom vrzeli, v katere skuša razvrščevalnik uvrstiti opravila, in seveda s številom opravil. Razvrščevalnik Haizea tako potrebuje približno sekundo za procesiranje 40 opravil, deset sekund za 100 opravil, okrog minute za 200 opravil in do 50 minut za 700 opravil, ne glede na izbrano tehniko razvrščanja. Združevanje opravil zmanjša njihovo število, skaliranje pa število vrzeli v urniku, ki jih mora razvrščevalnik preveriti. Čas, porabljen za razvrščanje, se tako zmanjša za 70 % pri nevzporednih opravilih brez uravnovešenih opravil, pri

katerih opravimo veliko združevanj, in za 5 % v primerih z le uravnovešenimi opravili, pri katerih se opravila le skalirajo. Pogosto ti prihranki pri razvrščanju odtehtajo čas, porabljen za združevanje in skaliranje. Manj časa se prihrani pri pristopu z razbitjem. V teh primerih so prihranki enakomerneje razporejeni po vseh primerih, saj se število opravil v množicah s predprocesiranjem ne spreminja. Povprečni prihranek časa znaša 18 %.

4.6 Zaključek

S predlaganim algoritmom izrabo virov v zasebnem oblaku izboljšamo s preoblikovanjem, osnovanem na skaliranju opravil. Algoritem združuje in skalira opravila s komplementarnimi potrebami tako, da lahko učinkovito sočasno uporabljajo vire na posameznem vozlišču. Opravila, ki jih nismo določili za združevanje, pa skuša skalirati tako, da bodo kar se da dobro zapolnila vire.

Algoritem je namenjen kot predprocesor, ki s skaliranjem opravil olajša delo razvrščevalniku in tako omogoči sestavo urnika, ki bolje izrabi vire. Boljša izraba mora opraviti več opravil na isti infrastrukturi ali zmanjšati infrastrukturo ter s tem izboljšati stroškovno in energetsko učinkovitost. Razviti pristop deluje dobro za manjše infrastrukture, ki se jih pretežno uporablja za poganjanje opravil, ki jih je mogoče skalirati in preložiti. Predprocesor lahko dodamo obstoječim in tudi novim razvrščevalnikom, pomembna je le odločitev, na kakšen način bomo združena opravila poslali razvrščevalniku. Prednost pristopa z množico je, da bo njegov izhod pravilno uporabila večina razvrščevalnikov, saj opravila za sobivanje zamenjamo z nadomestnim opravilom. Pri pristopu z razbitjem lahko dobimo boljše rezultate v primerjavi s pristopom z množico, vendar moramo pri tem imeti vpliv na delovanje razvrščevalnika.

Iskanje optimalne združitve opravil vodi hevristika, ki pod drobnogled vzame le tista opravila, ki so komplementarna glede na potrebe po virih, dovolj podobna po velikosti in se ujemajo v številu vzporednih opravkov. To močno zmanjša število poskusov združevanja in posledično tudi njegovo časovno zahtevnost.

S tem prispevkom želimo izpostaviti prožnost razvrščanja, ki jo je mogoče izkoristiti v zasebnih oblakih. Ponudniki večjih javnih oblakov običajno ponujajo le nekaj vnaprej določenih velikosti navideznih naprav (npr. Amazon [71]). Ta pristop sicer zniža učinkovitost izrabe virov, ampak zelo olajša načrtovanje in upravljanje z viri. Ceno nižje učinkovitosti plača uporabnik ali pa jo ponudnik poskuša kompenzirati s prekomernim zasedanjem infrastrukture, kar pomeni, da upa, da vsa opravila ne bodo sočasno v celoti izkoriščala vseh virov. Ta predpostavka je mogoča le v dovolj velikih oblakih, ne pa v manjših, za katere je primernejši naš predlagani pristop.

Rezultati jasno pokažejo, da predlagani pristop izboljša učinkovitost pri večini primerov. Kljub temu da izboljšave padajo z večanjem infrastrukture, je v računskih centrih, ki imajo do 100 vozlišč, mogoče utemeljeno pričakovati izboljšave učinkovitosti rabe med 2 % in 16 %. Ta spodbudna izboljšanja učinkovitosti in preprostost dodajanja predstavljenega predprocesorja obstoječim razvrščevalnikom pomenijo, da je predlagani pristop primeren za uporabo v vseh zasebnih oblakih s prilagodljivimi opravili in razvrščanjem na podlagi urnika.

V prihodnje si bomo prizadevali za izboljšati skaliranje opravil, ki jih algoritem ni združil. Nekaterih zaradi omejitev ne moremo skalirati tako, da porabijo celotno vozlišče, zato bi jih lahko prilagodili v fiksne, vnaprej določene deleže vozlišč. Prednost tega pristopa v primerjavi z javnimi ponudniki oblakov je, da končne velikosti opravil izbira upravljalnik virov in ne uporabnik. Prava izbira deležev vozlišč je problem, ki je ločen od združevanja in je povezan tudi s preudarnim skaliranjem opravil. Potrebna je hevristika, ki po eni strani opravila skalira v primerne deleže in po drugi skrbi, da bo na voljo primerno število različnih skaliranih opravil, ki jih bo razvrščevalnik lahko uporabil pri zapolnjevanju vrzeli. Napačna hevristika lahko poveča število opravil, ki jih razvrščevalnik ne more postaviti na ista vozlišča, kar poslabša učinkovitost urnika. Izbiro pravih vnaprej določenih deležev vozlišč se lahko uporabi tudi pri združevanju. Opravila se tako prilagodi za sobivanje na določenem deležu vozlišča, če nimajo dovolj fleksibilnosti, da bi se prilagodila na celo vozlišče.

Poglavje 5

Zaključek

V doktorskem delu smo naredili pregled nekaterih rešitev pri razvrščanju na področju porazdeljenih in rahlo sklopljenih sistemov ter postavili ločnice med različnim upravljanjem porazdeljene infrastrukture. Na tem področju smo predlagali rešitve za razvrstitev opravil na skupino računalniških vozlišč in primerno pripravo opravil. V delu smo predstavili koncepte, ki se trenutno uporabljajo, vključno z arhitekturo, značilnimi opravili in načini razvrščanja. Proučili smo pristope in izboljšave, povezane z razvrščanjem, ki pozitivno vplivajo na učinkovitost, ter za rešitev predlagali nove koncepte, ki se v danih pogojih dobro izkažejo. Razvili smo dva pristopa za izboljšanje rabe infrastrukture pri sistemih za upravljanje z viri, prvega na področju omrežnega računalništva s porazdeljenim upravljanjem z viri in drugega na področju oblaka.

Na področju omrežnega računalništva se redko uporablja delno dodeljevanje virov na vozliščih in skoraj vedno se informacije o stanju omrežja hranijo na enem mestu. To zelo omeji učinkovito izrabo infrastrukture. Še zlasti pri zelo porazdeljenih omrežjih in veliko uporabnikih je smiselno vpeljati delno dodeljevanje virov na vozliščih in porazdeljeno upravljanje z viri. Težava postane še večja, ko je infrastruktura sestavljena iz vozlišč z različno zmogljivostjo. Da bo iskanje vozlišč v takem sistemu uspešno, mora biti opis zahteve za opravilo dovolj splošen. Zato smo najprej razširili jezik opisovanja zahteve tako, da lahko vsebuje tudi omejitve za vsa vozlišča v celoti in ne samo za posamezno vozlišče. Poleg tega smo izpopolnili modul, ki omogoča sočasno delno dodeljevanje virov tako, da se trudi poiskati najšibkejšo sprejemljivo množico virov za izbrano opravilo. Ta pristop teži k odpravljanju razdrobljenosti izrabe virov, kar omogoča boljšo razvrstitev opravil, ki bodo prišla v sistem pozneje. Posledično skrbimo za učinkovitejšo izrabo celotnega sistema. Rezultati poskusov so pokazali, da se opravila, razvrščena s tem pristopom,

lahko začnejo izvajati prej in pogosto bolje izkoriščajo vire. Ti rezultati potrjujejo prvo hipotezo, navedeno v uvodnem poglavju.

Na področju računalništva v oblaku obstajajo pristopi za učinkovito delno dodeljevanje virov na vozliščih z virtualizacijo. Ker se predpostavlja, da je v računskem oblaku na voljo navidezno neskončna količina virov, večina pristopov dodeljuje vire sprotno (angl. *online*). Pri zasebnih oblakih, ki so bolj omejeni s količino virov, je še vedno uporaben sproten način dodeljevanja virov, vendar ta ne zagotavlja visoke učinkovitosti izrabe sistema. Za izboljšanje učinkovitosti rabe smo razvili pristop, ki temelji na iskanju dobrega ujemanja opravil. Razvili smo predprocesor za upravljalnik oblaka in ga povezali z razvrščevalnikom Haizea. Predprocesor identificira primerne skupine opravil, ki lahko sobivajo na istih vozliščih, in jih vertikalno skalira tako, da čim bolje izkoristijo posamezno vozlišče. Uporaba predprocesorja v večini primerov omogoča, da se izvajanje opravil na vozliščih konča prej, in posledično se izboljša učinkovitost izrabe infrastrukture. Ti rezultati potrjujejo drugo hipotezo, navedeno v uvodnem poglavju.

Pri razvoju in preverjanju delovanja obeh pristopov smo si pomagali s simulacijami. Pri vsakem pristopu smo zastavljeni problem opisali z gradniki, ki sestavljajo njegovo okolje, in v tem okolju osnovali potek delovanja. Delovanje smo preverili na realnih in simuliranih infrastrukturah različnih velikosti in zmogljivosti. S tem smo pokazali, da prvi pristop uspešno najde pravilne rešitve za zahteve do velikosti 60 vozlišč, medtem ko drugi pristop izboljša učinkovitost rabe virov na infrastrukturah do velikosti 100 vozlišč. Pristopa smo ovrednotili tako, da smo primerjali rezultate upravljalnikov, ki uporabljajo naše pristope, s tistimi, ki naših pristopov ne uporabljajo. Izmerili smo tudi hitrost delovanja in izpostavili tiste dele algoritmov, ki porabijo največ časa. Pri obeh pristopih se je izkazalo, da s sprejemljivim časovnim vložkom lahko dobimo boljše rezultate, ki manj obremenijo infrastrukturo.

5.1 Prispevki k znanosti

Nekateri rezultati, opisani v tem doktorskem delu, so že bili objavljeni v reviji [25] in predstavljeni na konferencah [117, 18]. Doktorsko delo vsebuje naslednja prispevka k znanosti:

Sočasno delno dodeljevanje virov v omrežnem računalništvu. Razvili smo algoritem za sočasno dodeljevanje virov [25], ki je primeren za sisteme z visoko porazdeljenimi omrežji, kjer se ne uporablja globalno poznavanje prostih kapacitet virov na vozliščih. Algoritem z delnim dodeljevanjem virov učinkovito razmešča opravila na heterogena vozlišča. Prva prednost omenjenega

prispevka je, da v zahtevi za opravilo lahko vsebuje tudi omejitve za vsa vozlišča v celoti in ne samo za posamezno vozlišče. Tako lahko algoritem lažje najde rešitve v heterogenih infrastrukturah. Druga prednost je izbira najšibkejše množice vozlišč, ki še zadostuje zahtevam opravila. Prispevek je opisan poglavju 3.

Sobivanje s preoblikovanjem opravil. Predlagali smo postopek za preoblikovanje opravil pred izvajanjem, ki upošteva sobivanje opravil na vozliščih, in s tem izboljšali izrabo virov. Postopek deluje kot predprocesor, ki ga dodamo razvrščevalniku, in je primeren za izboljšanje učinkovitosti rabe v manjših oblakih. Predprocesor v možici opravil poišče tista opravila, ki so primerna za sobivanje, in prilagodi njihovo porabo tako, da lahko dobro izkoriščajo vire na vozliščih. Prispevek je opisan v poglavju 4.

5.2 Smernice in nadaljnje delo

Opravila v predlaganih postopkih so fleksibilna do trenutka, ko jih razvrstimo na vire v infrastrukturi. Opravilom zagotovimo vire glede na njihovo začetno oceno porabe virov, kar je učinkovito samo pri opravilih, ki imajo dovolj kratek čas trajanja. Za dolgoživa opravila, kot je denimo spletni strežnik, pa je značilno, da se njihova zahteva po virih skozi čas spreminja. Tako bi bilo smiselno razviti metode, ki skupaj obravnavajo tista dolgoživa opravila, ki se jim zahteva po virih s časom spreminja v medsebojno obratnem sorazmerju. To pomeni, da imajo konice in nizko porabo ob različnih časih. Lahko pa opravilo s spremenljivo porabo kombiniramo z več fleksibilnimi opravili, ki jih lahko prilagajamo v preostali prostor na vozliščih, na katerih teče dolgoživo opravilo.

Dobrodošle bi bile tudi nadaljnje raziskave na zahtevah za opravila v velikih sistemih. Z boljšim poznavanjem želja uporabnikov je mogoče narediti boljše razvrščevalne rešitve. Podatki o zahtevah iz omrežnega računalništva [115] in superračunalnikov [116] trenutno obstajajo v omejenem obsegu, ki predlaganim raziskavam ne zadostujejo. Zato se bomo v prihodnje posvetili obširnejšemu zajemu podatkov o rabi računskih centrov.

Na koncu bi lahko v oba prispevka vpeljali tudi prilagajanje opravil s spreminjanjem vzporednosti, ki se imenuje horizontalno skaliranje. Ta sprememba bi prinesla dodatno prožnost, a tudi večjo zahtevnost. Ponovno bi morali definirati cenilne funkcije, ki bi ustrezno razlikovale in vrednotile rešitve z različnim številom vozlišč. Pot, ki smo jo izbrali in vsebuje uvajanje le ene vrste skaliranja naenkrat, je smiselna tudi zato, da lahko bolje ocenimo, katera vrsta skaliranja prinese večja izboljšanja učinkovitosti. V nadaljevanju bi bilo smiselno preizkusiti, kako se obnese

kombinirano horizontalno in vertikalno skaliranje.

Seznam slik

2.1 Urniki pri rezervacijah 23
2.2 Haizea dodatek k opisu navidezne naprave 29
2.3 Primer opisa zahteve za simulacijo 33

3.1 Izbira potencialnih začetnih časov v fazi III 43
3.2 Statistike parametrov M, L_1, L_2 49
3.3 Povprečni časi iskanja algoritma 52

4.1 Integracija predprocesorja 59
4.2 Razvrščanje treh opravil na dve vozlišči 61
4.3 Skica algoritma za združevanje in skaliranje 61
4.4 Primer razvrščanja, kadar so združena opravila predstavljena z nadomestnim opravilom (množica) ali brez (razbitje) 66
4.5 Razširjena shema integracije predprocesorja 67
4.6 Rezultati, urejeni po izboljšanjih pri pristopu z množicami 70
4.7 Izboljšanje učinkovitosti kot funkcija števila vozlišč pri pristopu z množicami 71
4.8 Rezultati, urejeni po izboljšanjih pri pristopu z razbitjem 72
4.9 Izboljšanje učinkovitosti kot funkcija števila vozlišč pri pristopu z razbitjem 72
4.10 Število optimizacij združevanja 74

Seznam preglednic

3.1 Statistika virov za 606 vozlišč okolja PlanetLab 47
3.2 Primerjava med kolektivnim in preprostim pristopom iskanja v vseh 540 primerih 50
3.3 Primerjava sočasnega in preprostega pristopa ter optimalnih rešitev za 5 vozlišč 51
3.4 Statistika iskalnih časov za 90 testov, ki so zahtevali 5 vozlišč 52

4.1 Statistika posameznih vhodnih množic opravil 68
4.2 Povzetek rezultatov pri pristopu z množicami 70
4.3 Povzetek rezultatov pri pristopu z razbitjem 72

Literatura

[1] Top500.org. http://www.top500.org/, 2014.

[2] Green500.org. http://www.green500.org/, 2014.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[4] I Foster and C Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[5] Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi. *Introduction to grid computing*. Citeseer, 2005.

[6] Viktors Berstis. Fundamentals of grid computing. *IBM Redbooks paper*, pages 1–28, 2002.

[7] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds. *ACM SIGCOMM Computer Communication Review*, 39(1):50, December 2008.

[8] Ilango Sriram and Ali Khajeh-hosseini. Research Agenda in Cloud Technologies. *Methodology*, 2008.

[9] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[10] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[11] Ian Foster. Concepts and Tools for Parallel Software Engineering. In *Designing and Building Parallel Programs*, chapter 1.4.4. Addison-Wesley Longman Publishing Co., Inc., 1995.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, January 2008.

[13] T. Somasundaram, B. Amarnath, Balakrishnan Ponnuram, Kumar Rangasamy, Rajendar Kandan, Rajiv Rajaian, R. Gnanapragasam, M. Ellappan, and M. Bairappan. Achieving co-allocation through virtualization in grid environment. *Advances in Grid and Pervasive Computing*, pages 235–243, 2009.

[14] C. Castillo, G.N. Rouskas, and K. Harfoush. Resource co-allocation for large-scale distributed environments. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 131–140. ACM, 2009.

[15] Hui-xian Li, Chun-tian Cheng, and K.W. Chau. Parallel resource co-allocation for the computational grid. *Computer Languages, Systems & Structures*, 33(1):1–10, April 2007.

[16] Karl Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*, pages 219–228. IEEE Comput. Soc, 1999.

[17] Ali Haydar Ozer and Can Ozturan. An auction based mathematical model and heuristics for resource co-allocation problem in grids and clouds. In *2009 Fifth International Conference on Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control*, pages 1–4. IEEE, September 2009.

[18] Matija Cankar, Primož Hadalin, and Matej Artač. Co-allocation of computational resources in XtreemOS grids. In *Proceedings of Nineteenth International Electrotechnical and Computer Science Conference - ERK 2010*, pages 10–13. Slovenia Section IEEE, 2010.

[19] Chao-Tung Yang, Fang-Yie Leu, and Sung-Yi Chen. Resource brokering using a multi-site resource allocation strategy for computational grids. *Concurrency and Computation: Practice and Experience*, 23(6):573–594, April 2011.

[20] Sid Ahmed Makhlouf and Belabbas Yagoubi. Distributed resources co-allocation in grid computing. In *2010 International Conference on Machine and Web Intelligence*, pages 244–249. IEEE, October 2010.

[21] Marco A. S. Netto and Rajkumar Buyya. Co-Resource Co-allocation in Grid Computing Environments. In *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, chapter 20, pages 476–494 pp. Information Science Publishing - Imprint of: IGI Publishing, Hershey, PA, 1 edition, 2010.

[22] Mark Stillwell, Frederic Vivien, and Henri Casanova. Virtual Machine Resource Allocation for Service Hosting on Heterogeneous Distributed Platforms. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 786–797. IEEE, May 2012.

[23] Jianhua Gu, Jinhua Hu, Tianhai Zhao, and Guofei Sun. A New Resource Scheduling Strategy Based on Genetic Algorithm in Cloud Computing Environment. *Journal of Computers*, 7(1):42–52, January 2012.

[24] Angela Sodan. Adaptive scheduling for qos virtual-machines under different resource availability–first experiences. In *14th Workshop on Job Scheduling Strategies for Parallel Processing*, 2009.

[25] Matija Cankar, Matej Artač, Marjan Šterk, Uroš Lotrič, and Boštjan Slivnik. Co-Allocation with Collective Requests in Grid Systems. *Journal of Universal Computer Science*, 19(3):282–300, 2013.

[26] Shantenu Jha, DS Katz, and Andre Luckow. Understanding scientific applications for cloud environments. In Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski, editors, *Cloud Computing: Principles and Paradigms*, pages 345–372. 2011.

[27] Jernej Virant. *Določanje performans računalniških sistemov*. Didakta, Radovljica, 1994.

[28] ILLIAC IV. http://en.wikipedia.org/wiki/ILLIAC_IV, 2014.

[29] SLING. http://www.sling.si/, 2014.

[30] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 99–106. IEEE, 2005.

[31] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE, 2004.

[32] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, May 2009.

[33] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Petterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM*, 33(3):3–12, 2003.

[34] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, June 2009.

[35] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Piotr Rygielski, Jason Ding, Walfredo Cirne, and Florian Rosenberg. Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios. Technical report, SPEC Research Group, 2013.

[36] Xueli Huang and Xiaojiang Du. Efficiently secure data privacy on hybrid cloud. *2013 IEEE International Conference on Communications (ICC)*, pages 1936–1940, June 2013.

[37] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, volume 37, page 164, New York, New York, USA, October 2003. ACM Press.

[38] Parallels Virtuozzo Containers. http://www.parallels.com/products/pvc/, 2014.

[39] VMware vSphere Hypervisor. http://www.vmware.com/products/vsphere-hypervisor/, 2014.

[40] Hyeonsang Eom and Heon Y. Yeom. Virtual machine scheduling for multicores considering effects of shared on-chip last level cache interference. In *2012 International Green Computing Conference (IGCC)*, pages 1–6. IEEE, June 2012.

[41] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, pages 1–14, New York, New York, USA, 2011. ACM Press.

[42] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. pages 1–33, 1995.

[43] Emmanuel Medernach. *Workload Analysis of a Cluster in a Grid Environment*, pages 36–61. 2005.

[44] M. Silberstein, D. Geiger, A. Schuster, and M. Livny. Scheduling Mixed Workloads in Multi-grids: The Grid Execution Hierarchy. In *15th IEEE International Conference on High Performance Distributed Computing*, pages 291–302. IEEE, 2006.

[45] G Avellino, S Beco, B Cantalupo, A Maraschini, F Pacini, M Sottilaro, A Terracina, D Colling, F Giacomini, E Ronchieri, and Others. The DataGrid workload management system: Challenges and results. *Journal of Grid Computing*, 2(4):353–367, 2004.

[46] Mohsen Amini Salehi, Bahman Javadi, and Rajkumar Buyya. Resource Provisioning based on Preempting Virtual Machines in Resource Sharing Environments. *Concurrency and Computation: Practice & Experience*, pages 1–21, 2013.

[47] Srividya Srinivasan, Vijay Subramani, Rajkumar Kettimuthu, Praveen Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. *LNCS in High Performance Computing 2002*, 2552:174–183, 2002.

[48] Marco A.S. Netto and Rajkumar Buyya. Coordinated rescheduling of Bag-of-Tasks for executions on multiple resource providers. *Concurrency and Computation: Practice and Experience*, 24(12):1362–1376, August 2012.

[49] Javier Navaridas, Jose a. Pascual, and Jose Miguel-Alonso. Effects of Job and Task Placement on Parallel Scientific Applications Performance. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 55–61. IEEE, 2009.

[50] Rui Li, Yinfeng Zhang, Zhengquan Xu, and Huayi Wu. A Load-balancing method for network GISs in a heterogeneous cluster-based system using access density. *Future Generation Computer Systems*, 29(2):528–535, February 2013.

[51] Katia Leal. Self-adjusting resource sharing policies in Federated Grids. *Future Generation Computer Systems*, 29(2):488–496, February 2013.

[52] Georgios L. Stavrinides and Helen D. Karatza. Scheduling real-time DAGs in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes. *Future Generation Computer Systems*, 28(7):977–988, July 2012.

[53] Angela C. Sodan and Wei Jin. Backfilling with Fairness and Slack for Parallel Job Scheduling. *Journal of Physics: Conference Series*, 256:012023, November 2010.

[54] Marco A. S. Netto and Rajkumar Buyya. Rescheduling co-allocation requests based on flexible advance reservations and processor remapping. *2008 9th IEEE/ACM International Conference on Grid Computing*, pages 144–151, September 2008.

[55] Luis Tomás, Agustín C. Caminero, Carmen Carrión, and Blanca Caminero. Network-aware meta-scheduling in advance with autonomous self-tuning system. *Future Generation Computer Systems*, 27(5):486–497, May 2011.

[56] Joerg Schneider and Barry Linnert. Efficiently Managing Advance Reservations Using Lists of Free Blocks. In *23rd International Symposium on Computer Architecture and High Performance Computing*, pages 183–190. IEEE, October 2011.

[57] Janki Akhani, Sanjay Chuadhary, and Gaurav Somani. Negotiation for resource allocation in IaaS cloud. In *Proceedings of the Fourth Annual ACM Bangalore Conference on - COMPUTE '11*, pages 1–7, New York, New York, USA, 2011. ACM Press.

[58] A. Galstyan, K. Czajkowski, and K. Lerman. Resource Allocation in the Grid with Learning Agents. *Journal of Grid Computing*, 3(1):91–100, 2005.

[59] A Galstyan, K Czajkowski, and K Lerman. Resource allocation in the grid using reinforcement learning. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS 2004*, pages 1314–1315, 2004.

[60] Christine Morin. XtreemOS: A Grid Operating System Making your Computer Ready for Participating in Virtual Organizations. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 393–402. IEEE, May 2007.

[61] Cristian Klein-Halmaghi. *Cooperative resource managment for parallel and distributed systems*. PhD thesis, 2012.

[62] HTCondor. http://research.cs.wisc.edu/htcondor/, 2014.

[63] SLRUM - Simple Linux Utility for Resource Management. http://slurm.schedmd.com/slurm.html, 2014.

[64] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, pages 776–783 Vol. 2. IEEE, 2005.

[65] TORQUE Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/, 2014.

[66] Univa Grid Engine. http://www.univa.com/products/grid-engine, 2014.

[67] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: A Distributed Job Scheduler. In *Beowulf Cluster Computing with Linux*, pages 307–350. MIT Press, Cambridge, MA, USA, 2002.

[68] OpenNebula. http://opennebula.org/start, 2013.

[69] Borja Sotomayor, Kate Keahey, and Ian Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th international symposium on High performance distributed computing - HPDC '08*, page 87, New York, New York, USA, 2008. ACM Press.

[70] Haizea - An Open Source VM-based Lease Manager. http://haizea.cs.uchicago.edu/, 2014.

[71] Amazon Instance Types. http://aws.amazon.com/amazon-linux-ami/instance-type-matrix/, 2013.

[72] OpenStack. http://www.openstack.org/, 2014.

[73] VCloud Suite. http://www.vmware.com/products/vcloud-suite/, 2014.

[74] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE, 2009.

[75] Nimbus. http://www.nimbusproject.org/, 2014.

[76] OpenNebula. http://archives.opennebula.org/documentation:rel4.4:sunstone, 2014.

[77] Toni Cortes, Carsten Franke, Yvon Jégou, Thilo Kielmann, and Domenico Laforenza. XtreemOS: a vision for a Grid operating system. *White paper*, 2008.

[78] Erica Y. Yang, Brian Matthews, Amit Lakhani, Yvon Jégou, Carsten Sánchez, David Oscar Franke, Philip Robinson, Adolf Hohl, Bernd Scheuermann, Daniel Vladušič, Haiyan Yu, An Qin, Rubao Lee, Erich Focht, and Massimo Coppola. Virtual Organization Management in XtreemOS: An Overview. In *Towards Next Generation Grids, Proceedings of the CoreGRID Symposium*, pages 73–82. 2007.

[79] An Qin, Haiyan Yu, Chengchun Shu, Xiaoqian Yu, Yvon Jegou, and Christine Morin. Operating System-Level Virtual Organization Support in XtreemOS. In *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 17–24. IEEE, December 2008.

[80] Felix Hupfeld, Toni Cortes, Björn Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. XtreemFS - a case for object-based storage in Grid data management. In *33th International Conference on Very Large Data Bases (VLDB) Workshops*, 2007.

[81] Felix Hupfeld, Toni Cortes, Björn Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. The XtreemFS architecture-a case for object-based file systems in Grids. *Concurrency and Computation: Practice and Experience.*, 20(8):1–12, December 2008.

[82] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG - ERLANG '08*, pages 41–48, New York, New York, USA, 2008. ACM Press.

[83] Alvaro Martínez, Santiago Prieto, Noé Gallego, Ramon Nou, Jacobo Giralt, and Toni Cortes. XtreemOS-MD: Grid computing from mobile devices. In *Mobile Wireless Middleware, Operating Systems, and Applications*, volume 48, pages 45–58. 2010.

[84] Ramon Nou, Jacobo Giralt, Julita Corbalan, Enric Tejedor, J.O. Fitó, J.M. Perez, and T. Cortes. Xtreemos application execution management: A scalable approach. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, volume 2, pages 49–56. IEEE, 2010.

[85] Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher, and Andreas Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, Global Grid Forum, http://www.gridforum.org/documents/GFD.56.pdf.

[86] Ian Foster, Nicolas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 8–15, 2004.

[87] Hashim Mohamed and Dick Epema. KOALA: a co-allocating grid scheduler. *Concurrency and Computation: Practice and Experience*, 20(16):1851–1876, November 2008.

[88] H.H. Mohamed and D.H.J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, pages 784–791.

[89] Rajkumar Buyya and Manzur Murshed. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, November 2002.

[90] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A F De Rose, Rajkumar Buyya, and César a. F. De Rose. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, January 2011.

[91] J Gustedt, E Jeannot, and M Quinson. Experimental methodologies for large-scale systems: a survey. *Parallel Processing Letters*, 19(03):399–418, September 2009.

[92] Rodrigo N. Calheiros, Marco A.S. Netto, César A.F. De Rose, and Rajkumar Buyya. EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of Cloud computing applications. *Software: Practice and Experience*, 43(5):595–612, May 2013.

[93] Amit Nathani, Sanjay Chaudhary, and Gaurav Somani. Policy based resource allocation in IaaS cloud. *Future Generation Computer Systems*, 28(1):94–103, June 2011.

[94] Xin Sun, Sen Su, Peng Xu, Shuang Chi, and Yan Luo. Multi-dimensional Resource Integrated Scheduling in a Shared Data Center. In *31st International Conference on Distributed Computing Systems Workshops*, pages 7–13. IEEE, June 2011.

[95] CloudSim. http://www.cloudbus.org/cloudsim/, 2014.

[96] EMUSIM: Integrated Emulation and Simulation for Evaluation of Cloud Computing Applications. http://www.cloudbus.org/cloudsim/emusim/, 2014.

[97] DG Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. *Job Scheduling Strategies for Parallel Processing*, 1459:1–24, 1998.

[98] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. *Future Generation Computer Systems*, 29(8):2009–2025, October 2013.

[99] S Harizopoulos, M.A. Shah, J Meza, and P Ranganathan. Energy efficiency: The new holy grail of data management systems research. In *Fourth biennial Conference on Innovative Data Systems Research (CIDR'09)*, pages 4–7, 2009.

[100] N. Rasmussen. Electrical Efficiency Modeling of Data Centers, 2006.

[101] Ana-maria Oprescu and Thilo Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 351–359. IEEE, November 2010.

[102] Capit Nicolas, Emeras Joseph, and Saint Martin. OAR Documentation - User Guide, http://oar.imag.fr/documentation/. Technical report, 2013.

[103] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS'99.*

1999 Seventh International Workshop on, number 1, pages 27–36. IEEE, 1999.

[104] Erik Elmroth and Johan Tordsson. A standards-based Grid resource brokering service supporting advance reservations, coallocation, and cross-Grid interoperability. *Concurrency and Computation: Practice and Experience*, 21(18):2298–2335, December 2009.

[105] J. MacLaren. HARC: the highly-available resource co-allocator. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS-Volume Part II*, pages 1385–1402. Springer-Verlag, 2007.

[106] A. Takefusa, H. Nakada, T. Kudoh, Y. Tanaka, and S. Sekiguchi. GridARS: An advance reservation-based grid co-allocation framework for distributed computing and network resources. In *Proceedings of the 13th international conference on Job scheduling strategies for parallel processing*, pages 152–168. Springer-Verlag, 2007.

[107] Chuang Liu and I. Foster. A constraint language approach to grid resource selection. In *Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, number 2, 2003.

[108] Anne-Cécile Orgerie and Laurent Lefèvre. A year in the life of a large scale experimental distributed system: the Grid'5000 platform in 2008. Technical Report March, 2010.

[109] Anthony Sulistio and Rajkumar Buyya. A grid simulation infrastructure supporting advance reservation. In *16th International Conference on Parallel and Distributed Computing and Systems*, 2004.

[110] C. Castillo, G.N. Rouskas, and K. Harfoush. Online algorithms for advance resource reservations. *Journal of Parallel and Distributed Computing*, 71(7):963–973, July 2011.

[111] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, December 2013.

[112] Kamal Kc and Kemafor Anyanwu. Scheduling Hadoop Jobs to Meet Deadlines. In *IEEE Second International Conference on Cloud Computing Technology and Science*, pages 388–392. IEEE, November 2010.

[113] S Iqbal, R Gupta, and YC Fang. Planning considerations for job scheduling in HPC clusters. *High-Performance Computing, reprinted from Dell Power Solutions*, (February):133–136, 2005.

[114] OpenOpt. http://openopt.org/, 2013.

[115] The Grid Workloads Archive. http://gwa.ewi.tudelft.nl/pmwiki/pmwiki.php, 2013.

[116] Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, 2013.

[117] Matija Cankar and Roman Trobec. Experimental evaluation of Grid5000 performance in the solution of PDE. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 203–207. IEEE, 2010.

Dodatki

Dodatek A

Oznake v poglavjih 3 in 4

A.1 Oznake v poglavju 3

i	oznaka vira
j	oznaka vozlišča
$\mathcal{R}$	množica kvantitativnih virov
R_i	vir
V_j	vozlišče
R_{ne}	vir z najbolj nedopustno zahtevano količino
$\mathcal{V}$	množica vozlišč
t	čas
t_{earliest}	najzgodnejši čas začetka
t_{latest}	najpoznejši čas začetka
t_{res}	trajanje rezervacije
t_s	začetni čas
t_C	časovna konstanta
$r_{i,j}$	količina i-tega vira na j-tem vozlišču
d_i	individualna zahteva po i-tem viru
e_i	kolektivna zahteva po i-tem viru
N	število zahtevanih vozlišč
u_i	faktor učinkovitosti
$\widetilde{u}_i$	poenostavljena, najslabša učinkovitost
ω	funkcija razvrstitve
$M_{\mathrm{init}} < M < M_{\mathrm{max}}$	faktor povečanja množice vozlišč
$a_{i,j}(t)$	funkcija zasedenosti vira i na vozlišču j v času t

$A_j(t)$	funkcija prostosti vseh virov (so prosti glede na omejitve)
$A_j^s(t)$	funkcija začetkov območij (vsi viri prosti znotraj meja med trajanjem rezervacije)
$Z_{i,j}(t)$	zasedenost oziroma rezervacije i-tega vira j-tega vozlišča
Z_{ij}^t	največja zasedenost v intervalu $[t, t + t_{\mathrm{res}}]$
$\mathcal{V}_{\mathrm{job}}$	vmesna množica vozlišč
$\mathcal{V}_{\mathrm{sol}}$	množica vozlišč, ki predstavljajo potencialno rešitev
$\mathcal{V}_a$	množica prostih vozlišč
V_m	vozlišče z visoko vrednostjo R_{ne}
L_1	konstanta, število iteracij iskanja dopustne rešitve
L_2	konstanta, število iteracij optimizacije
π	funkcija naključne razvrstitve
$\bar{r}_i$	povprečna vrednost vira čez vsa vozlišča
σ_i	standardni odklon vira i (r_i) čez vsa vozlišča

A.2 Oznake v poglavju 4

r	vir
R	množica virov
o	opravilo
D_o	množica, ki vsebuje zahteve po viru ter minimalno in maksimalno količino vira
$d_{or}^{\mathrm{MIN}} \leq d_{or} \leq d_{or}^{\mathrm{MAX}}$	zahteva opravila o po viru r in omejitvi
t_o	trajanje opravila o
O	množica opravil
$\lvert\mathcal{V}\rvert$	število vozlišč
$\mathcal{T}$	urnik
p_o	vzporednost opravila o
$t_{o,s}$	začetni čas
r_{MAX}	prevladujoči vir v množici O
w_o	velikost opravila
W	prag podobnosti
$C_{\mathrm{eq,eq}}$	razred uravnovešenih opravil
α	klasifikacijski prag
A	oznaka opravila iz razreda A
B	oznaka opravila iz razreda B
o'	skalirano opravilo
t_{pair}	najkrajše skupno trajanje dveh opravil