UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jernej Južna

Prenos inženirskih aplikacij v oblak s platformo mOSAIC

MAGISTRSKO DELO

Mentor: doc. dr. Matija Marolt
Somentor: doc. dr. Vlado Stankovski

Ljubljana, 2013

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

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

Št.: 147-MAG-RI/2012
Datum: 16. 04. 2012

Jernej Južna, univ. dipl. inž. rač. in inf.

L j u b l j a n a

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **Prenos inženirskih aplikacij v oblak s platformo mOSAIC**

Enabling engineering applications for the cloud on the mOSAIC platform

Tematika naloge:

V zadnjih letih se področje porazdeljenega računalništva osredotoča na računalništvo v oblaku (ang. cloud computing). Razvoj aplikacij, ki oblačno platformo oz. oblak polno izkoriščajo, je zelo zahteven proces, saj zahteva poznavanje širokega spektra različnih tehnologij. Prenos obstoječih aplikacij v oblak je še precej večji problem, saj uporaba oblačnih tehnologij zahteva temeljite posege v obstoječe stanje nemalokrat pa celo ponovni razvoj. Slednje je praktično nemogoče pri inženirskih aplikacijah, ki temeljijo na zahtevnih, matematično nestabilnih algoritmih, strogo vezanih na določeno okolje, ki pa v oblaku ni na voljo. Hkrati pa je uporaba oblaka za te aplikacije zelo privlačna, saj med drugim po sprejemljivih cenah nudi veliko računske in pomnilniške zmogljivosti.
V okviru magistrske naloge preučite možnost prenosa obstoječe, zahtevne inženirske aplikacije razvite v okolju Mathworks Matlab, v oblak z uporabo platforme mOSAIC. Pri tem skušajte najti čim bolj enostaven in splošen način, ki bi bil tudi nadalje uporaben predvsem s stani inženirjev, strokovnjakov na svojem področju, ki bi želeli razviti ali prenesti svoje aplikacije v oblak. Vašo rešitev tudi ustrezno ovrednotite, predvsem s stališča ustreznega izkoriščanja oblačnih tehnologij.

M e n t o r:

doc. dr. Matija Marolt

S o m e n t o r:

doc. dr. Vlado Stankovski

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

D e k a n:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

magistrskega dela

Spodaj podpisani **Jernej Južna**,
z vpisno številko **63990228**,

sem avtor magistrskega dela z naslovom

Prenos inženirskih aplikacij v oblak s platformo mOSAIC.

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod vodstvom mentorja
  doc. dr. Matije Marolta
  in somentorstvom
  doc. dr. Vlada Stankovskega
- so elektronska oblika magistrskega dela, naslova (slov., angl.), povzetka (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela
- in soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, dne _____ Podpis avtorja: _____

Zahvala

Najprej bi se rad zahvalil mentorju doc. dr. Matiji Maroltu in somentorju doc. dr. Vladu Stankovskemu za odzivno, strokovno in nesebično pomoč pri izdelavi magistrskega dela.

Vladu gre dodatna zahvala tudi za vodenje mojega dela pri projektu mOSAIC, nešteto ur razprav, sproščene neformalne pogovore in seveda za priganjanje pri pisanju strokovnih člankov.

Za pomoč pri rokovanju z aplikacijo NoDeK bi se rad zahvalil Petru Češarku, članu Katedre za mehaniko, Fakultete za gradbeništvo in geodezijo.

Zahvala gre tudi ostalim sodelavcem pri projektu mOSAIC. S svojimi nasveti in modrostjo ste se še posebaj izkazali Ciprian, Silviu, Svatopluk, Tamas in Miha.

Za koristne pogovore, vzpodbude, nasvete in sproščeno delovno vzdušje se želim zahvaliti vsem preostalim sodelavcem Laboratorija za računalniško grafiko in multimedije - Saši, Alenki, Marku, Cirilu in Matevžu. Matlab, LaTeX, prevodi, pravopis in splošna razgledanost bi brez vas precej trpeli.

Rad bi se zahvalil tudi vsem prijateljem in družini za moralno podporo.

Za konec bi se rad zahvalil še Maji za vse razumevanje, ker med pisanjem dela večrat nisem bil to, kar bi moral biti.

Še enkrat iskrena hvala vsem.

Kazalo

Seznam uporabljenih kratic in simbolov

AJAX (Asynchronous JavaScript And XML) - Asinhroni JavaScript in XML

AMQP (Advanced Message Queuing Protocol) - Napredni protokol za sporočilne vrste

API (Application Programming Interface) - Programski vmesnik

CA (Cloud Agency) - Agencija za oblake

CPF (Call For Proposal) - Poziv za predlog

GAE (Google App Engine) - PaaS okolje podjetja Google

HTTP (Hyper Text Transfer Protocol) - Protokol za prenos hiperteksta

IaaS (Infrastructure as a Service) - Infrastruktura kot storitev

JAR (Java ARchive) - Arhiv, ki vsebuje javanske razrede

JSON (JavaScript Object Notation) - Objektna notacija v JavaScriptu

KE - Končni element

KV (Key-Value) - Ključ-vrednost

MCJ (Matlab Compiler Build JA) - Prevajalnik Matlab za uporabo v Javi

MCR (Matlab Compiler Runtime) - Izvajalno okolje prevajalnika Matlab

MDCS (Matlab Distributed Computing Server) - Strežnik Matlab za porazdeljeno računanje

mOS (mOSAIC Operating System) - Operacijski sistem mOSAIC

mOSAIC (Open Source API and Platform for Multiple Clouds) - Odprtokodni programski vmesnik in platforma za več oblakov

MPI (Message Passing Interface) - Vmesnik za prenašanje sporočil

MQ (Message Queue) - Sporočilna vrsta

NoSQL (Not only SQL) - Oznaka za nerelacijske podatkovne baze

OCCI (Open Cloud Computing Interface) - Odprti vmesnik za računalnišvo v oblaku

PaaS (Platform as a Service) - Platforma kot storitev

PPMM - Preprost prevajalnik za Matlabove matrike

PTC (Portable Testbed Cluster) - Prenosna testna gruča

REST (REpresentational State Transfer) - Prenos reprezentativnega stanja

SaaS (Software as a Service) - Programska oprema kot storitev

SAN (Storage Area Network) - Diskovno omrežje

SFTP (SSH File Transfer Protocol) - Protokol za prenos datotek prek varnega kanala

SLA (Service Level Agreement) - Dogovor o ravni storitve

SOAP (Simple Object Access Protocol) - Protokol za izmenjavo podatkov med spletnimi storitvami

SQL (Structured Query Language) - Strukturirani povpraševalni jezik

SSH (Secure Shell) - Varna lupina

VM (Virtual Machine) - Navidezna naprava

XML (Extensible Markup Language) - Razširljiv označevalni jezik

Povzetek

V magistrskem delu smo raziskali problem prenosa obstoječih računsko in podatkovno zahtevnih inženirskih aplikacij v oblak. Tehnologije računalništva v oblaku lahko inženirskim aplikacijam ponudijo veliko naprednih možnosti, kot so vseprisotnost, skalabilnost in odpornost na napake, vendar jih sedanje inženirske aplikacije še zdaleč ne izkoriščajo. Za uspešen prenos je namreč treba poznati širok spekter zahtevnih tehnologij in nato vložiti veliko časa in napora za predelavo obstoječih aplikacij. Zato veliko inženirskih aplikacij zanimivih za širšo javnost ostane v uporabi v omejenem krogu razvojnih oddelkov.

V okviru naloge smo razvili učinkovito in splošno namensko programje, ki rešuje zastavljeni problem in je tudi nadalje uporabno, predvsem za inženirje, ker omogoča prenos aplikacij v oblak z minimalnimi posegi v obstoječe aplikacije. Za poskuse smo se omejili na aplikacije razvite v okolju MathWorks Matlab in na odprtokodno platformo kot storitev mOSAIC. Slednja se lahko namesti pri poljubnem ponudniku infrastrukture kot storitev in tako odpravlja problem zaklepa aplikacije na posameznega ponudnika.

Razvito programje smo dopolnili s prevajalnikom, ki končnim uporabnikom omogoča izmenjavo podatkov z nastalo spletno storitvijo v programskem jeziku Matlab. Njegovo delovanje smo preverili z dvema aplikacijama: aplikacijo za oceno hitrosti izvajanja posameznih sklopov Matlaba in zahtevno inženirsko aplikacijo za analizo statičnih obremenitev struktur, ki smo ju namestili pri dveh različnih ponudnikih računalništva v oblaku – Amazon EC2 in Eucalyptus. Rezultati kažejo, da razvito programje omogoča uspešen prenos obeh aplikaciji, pri tem pa je zagotovljena pravilnost delovanja in primerljiva hitrost izvajanja. Razvito programje je uporabno tudi v primeru sočasne uporabe prenesenih aplikacij s strani več uporabnikov, saj obe skalirata skoraj linearno.

Navsezadnje smo obravnavali tudi možnosti za paralelno izvajanje aplikacije za analizo statičnih obremenitev struktur. Pri tem smo želeli izkoristiti možnosti, ki jih ponujajo porazdeljena okolja računalništva v oblaku. S tem namenom smo aplikacijo preuredili in s testiranjem ugotovili, da lahko dosežemo do 2x pohitritev, vendar ob hkratni 4x večji porabi virov. Ugotovili smo, da je nizka stopnja pohitritve izbrane aplikacije rezultat narave uporabljenega algoritma in pa nizke prepustnosti vhodno/izhodnega sistema, ki je posledica velikih režijskih stroškov platforme mOSAIC. Nekatere druge inženirske aplikacije bi vsekakor imele več koristi od paralelizacije.

KLJUČNE BESEDE
računalništvo v oblaku, inženirska aplikacija, Matlab, platforma mOSAIC, vzporedno računalništvo

Abstract

This Master's Thesis investigates the problem of porting existing computationally and data intensive engineering applications to the Cloud. Cloud computing technologies can offer a lot of advanced possibilities, such as ubiquity, scalability and fault tolerance to engineering applications. However, existing applications do not take advantage of Cloud computing technologies. Successful porting necessitates the knowledge of a wide spectrum of technologies as well as development time for adaptation of existing applications. For this reason many engineering applications that would be of interest to the general public, are used by a limited number of people within research and development departments.

In the context of this work, we developed an effective and general purpose programming code that can be used by engineers to port their applications to the Cloud with minimal changes of their existing applications. Our experiments focused on applications developed in MathWorks' Matlab and the open source mOSAIC Platform as a Service (PaaS). The mOSAIC PaaS can be installed on top of an arbitrary Infrastructure as a Service (IaaS) provider and thus eliminates the vendor lock-in problem.

The developed procedure was implemented and extended with a compiler, which facilitates data exchange between the end users and the resulting Web service in the Matlab programming language. The performance of the overall solution was tested against two applications: a benchmarking application for the estimation of the Matlab performance and a civil engineering application for analysis of structures under static loading, for which we used two different Cloud providers - Amazon EC2 and Eucalyptus. The results indicate that the developed programming code facilitates successful porting of both applications to the Cloud. At the same time numerical stability and adequate performance are achieved. The developed programming code is useful also in the case of parallel usage of the Cloud enabled applications by an increasing number of users as both applications exhibit linear scale-up.

Finally, we considered the possibility for parallel implementation of the application for analysis of structures under static loading. The aim was to exploit the possibilities offered by distributed computing environments, such as the Cloud. The application was rearranged and the testing results show that approximately double speed-up can be achieved by a four times increase in the consumption of Cloud resources. The low speed-up may be attributed to the nature of the employed algorithm and to the low throughput of the input/output system, which is caused by the mOSAIC platform computing stack. Some other engineering applications would certainly gain more with parallelization.

KEYWORDS

cloud computing, engineering application, Matlab, mOSAIC platform, parallel computing

1 Uvod

V zadnjih letih se področje porazdeljenega računalništva osredotoča na računalništvo v oblaku (angl. cloud computing) [4]. Glavni razlog za množično selitev na oblačno platformo oz. v oblak so prednosti, ki jih takšno okolje nudi aplikacijam oz. njihovim uporabnikom, razvijalcem in upravljavcem. Okolje namreč omogoča razvoj inovativnih aplikacij, ki imajo lahko mnogo koristnih lastnosti [15], med drugim elastičnost, skalabilnost, odpornost na napake, nadzor in vseprisotnost. Hkrati razvoj ne zahteva investicij v potrebno infrastrukturo, odpade pa tudi skrb za njeno vzdrževanje. S tem se izognemo visokim začetnim stroškom, saj se v večini primerov plača le za porabljene računalniške vire. Ob kasnejšem morebitnem uspehu pa se lahko dodatne vire dokupi in tako prilagodi trenutnim potrebam aplikacije [1].

1.1 Motivacija

A dosegljivost vseh naštetih koristnih lastnosti ni samoumevna, pač pa je potrebno za njihovo uveljavitev znati oblak ustrezno izkoristiti. Namensko razvite aplikacije z zagotavljanjem teh lastnosti nimajo težav, a je njihov razvoj (pre)drag, (pre)počasen in dostopen le ozkim strokovnjakom, saj je za ustrezno izkoriščenost treba poznati širok spekter zahtevnih tehnologij [34].

Prenos obstoječih aplikacij v oblak je še precej večji problem, saj uporaba oblačnih tehnologij zahteva temeljite posege v obstoječe stanje, nemalokrat pa celo ponovni razvoj. Prenos je še posebej težaven pri inženirskih aplikacijah, ki temeljijo na zahtevnih, matematično nestabilnih algoritmih, strogo vezanih na določeno okolje, ki pa v oblaku velikokrat ni na voljo [34]. Prav v izdelavo teh je bilo v zadnjih desetletjih vloženo mnogo napora in sredstev. Čeprav jih je veliko razvitih za visokozmogljive platforme (npr. superračunalnike ali mrežno računalništvo [38]), pa se žal večina takšnih aplikacij uporablja le občasno na manj zmogljivi lokalni infrastrukturi s strani maloštevilčnih ekspertov, običajno članov oddelka, kjer je bila aplikacija razvita.

Za takšne aplikacije bi bila uporaba oblaka zelo privlačna z več vidikov, saj omogoča enostaven dostop do zmogljivih računskih in pomnilniških virov po sprejemljivih cenah [1,10]. V

primeru občasnih potreb po kratkotrajnih, a intenzivnih izračunih je tako bolj smiselno namesto zahtevnega in dragega vzdrževanja lastne infrastrukture najeti potrebne dodatne računske vire le za čas opravljanja izračunov. Podobno se lahko aplikacija brez večjih stroškov odpre širši znanstveni skupnosti brez bojazni, kako bi morebitna povečana obremenitev vplivala na lastno infrastrukturo.

Selitev v oblak je torej cenovno dostopna opcija, a je prav zahtevnost prenosa eden glavnih razlogov, da aplikacije "obtičijo" v razvojnih oddelkih. Posamezni ponudniki računalništva v oblaku se težavnosti prenosa zavedajo in skušajo z lastnimi orodji postopek poenostaviti. A njihova uporaba lahko razvijalce vodi v odvisnost/zaklep na posameznega ponudnika (angl. vendor lock-in) [23], kar lahko kasneje predstavlja še precej večji problem.

Možno rešitev nakazuje platforma mOSAIC [12], ki nudi podporo razvijalcem, a hkrati deluje neodvisno od posameznega ponudnika računalništva v oblaku.

1.2 Cilji in namen dela

Namen tega dela je predstaviti možne pristope pri preoblikovanju inženirske aplikacije v obliko spletne storitve, ki se izvaja v oblaku. Pri tem se bomo osredotočili le na aplikacije, razvite v okolju Mathworks Matlab. Predstavili bomo posamezne modele računalništva v oblaku in se opredelili do njihove primernosti za podporo preoblikovanju. Posebno pozornost bomo namenili platformi mOSAIC [12].

Cilj dela je razvoj programskega ogrodja (angl. software framework) za platformo mOSAIC, ki omogoča, da poljubno aplikacijo v okolju Matlab na enostaven način preoblikujemo v storitev v oblaku za končne uporabnike. Naš pristop bomo ovrednotili na primeru prenosa generične aplikacije, namenjene testiranju zmogljivosti, in obstoječe aplikacije NoDeK [9], razvite na Fakulteti za gradbeništvo in geodezijo Univerze v Ljubljani, ki analizira obnašanje struktur pod statično obremenitvijo.

V zadnjo aplikacijo bomo nadalje še globlje posegli in skušali ugotoviti, kako z vpeljavo vzporednega računanja zagotoviti čim boljšo izrabo razpoložljivih virov za čim večjo pohitritev njenega izvajanja.

Platforma mOSAIC zagotavlja neodvisnost aplikacij od ponudnika računalništva v oblaku. Podpora za vse glavne ponudnike je v platformo že vgrajena, v delu pa bomo opisali tudi potrebne korake za dodajanje podpore novemu ponudniku.

1.3 Struktura dela

V prvem poglavju smo grobo opisali ovire, s katerimi se srečujejo inženirji pri prenosu obstoječih aplikacij v oblak. V drugem poglavju se bomo natančneje seznanili s tematiko računalništva v oblaku, inženirskimi aplikacijami ter okoljem Matlab in nakazali možno reševanje problema prenosa. Sledi poglavje, kjer bomo predstavili in podrobno opisali platformo mOSAIC. V četrtem poglavju je predstavljeno programsko ogrodje, ki izkorišča prednosti platforme in rešuje prenos obstoječih aplikacij v oblak. V petem poglavju našo rešitev ovrednotimo na podlagi opravljenih testiranj. V šestem poglavju je opisan postopek paralelizacije aplikacije NoDeK. Sledi še zadnje poglavje, ki povzema celotno delo in poda iztočnice za nadaljnje delo.

2 Pregled področja

V našem delu želimo obstoječo zahtevno inženirsko aplikacijo, napisano za okolje Matlab, prenesti v oblak. V ta namen moramo podati nekaj definicij, zato v podpoglavju 2.1 najprej predstavimo pojem oblak, njegovo definicijo in najpogostejše lastnosti. V naslednjem podpoglavju (2.2) je narejen pregled uporabe oblaka za izvajanje inženirskih aplikacij. Na koncu sledi še pregled dosedanjih prizadevanj pri uporabi okolja Matlab v oblaku (podpoglavje 2.3).

2.1 Oblak

S podporo virtualizacije, ki omogoča učinkovito konsolidacijo strežniških sistemov in podatkovnih centrov v enotno infrastrukturo, so se stroški znižali do te mere, da je presežke virov takšne infrastrukture mogoče ponuditi na trgu za dovolj dostopno ceno [15]. Odjemalci jih lahko izkoristijo za izvajanje raznovrstnih storitev, ob tem pa podrobnosti o infrastrukturi lahko odmislijo, saj je ni treba vzdrževati, videti je neomejeno zmogljiva in se jo plačuje po modelu "plačaj, kolikor porabi" (angl. pay-as-you-go). V zadnjih nekaj letih je tako računalništvo v oblaku postalo vodilni tehnološki trend na področju informacijske tehnologije in je prodrlo v vse sfere računalništva, promovirajo pa ga praktično vsa največja svetovna podjetja s tega področja. Zaradi razširjenosti obstaja veliko različnih definicij, kaj računalništvo v oblaku sploh je. Ena najpogosteje uporabljenih je naslednja:

Računalništvo v oblaku je model za omogočanje vseprisotnega in udobnega omrežnega dostopa do deljenih in nastavljivih računalniških virov (omrežje, strežniki, shramba, aplikacije in storitve), ki jih lahko pridobimo ali opustimo v vsakem trenutku z minimalno interakcijo s ponudnikom storitve [30].

Glede na definicijo je pet bistvenih značilnosti računalništva v oblaku naslednjih:

Samopostrežnost (On-demand self-service) Uporabnik lahko enostransko pridobi ali opusti vire glede na lastne potrebe avtomatsko brez človeškega posredovanja na strani ponudnika.

Vseprisotnost (Broad network access) Viri so uporabniku prek omrežja z uporabo stan-

dardnih mehanizmov vedno na voljo, ne glede na to, kje se nahaja oz. s kakšno napravo do njih dostopa.

Konsolidacija virov (Resource pooling) Viri ponudnika so logično združeni in se delijo med več hkratnih uporabnikov na način, da se medsebojno ne motijo (izolacija). Ponudnik vire posamezniku dodeljuje dinamično glede na njegove trenutne potrebe. Uporabnik nima nadzora nad tem, kateri konkretni fizični vir se mu bo dodelil.

Elastičnost (Rapid elasticity) Viri se lahko hipno pridobijo ali opustijo glede na trenutne potrebe. Z vidika uporabnika viri delujejo neomejeno zmogljivi in jih lahko v katerem-koli trenutku zakupi poljubno mnogo.

Merljivost (Measured service) Uporaba kateregakoli vira s strani uporabnika je v vsakem trenutku natančno zabeležena. Meritve so osnova za nadziranje, načrtovanje, spremljanje porabe in obračunavanje storitve ter so na voljo tako ponudniku kot uporabniku, kar vnaša transparentnost v medsebojno sodelovanje.

Delitev računalništva v oblaku na storitvene modele omogoča, da se heterogenost in kompleksnost računalništva v oblaku skrije za ustreznim nivojem abstrakcije, a hkrati še vedno zagotavlja njegovo učinkovito izrabo. Ločimo tri osnovne storitvene modele:

Infrastruktura kot storitev (Infrastructure as a Servise, IaaS) Zmožnost pridobivanja procesnih, podatkovnih, omrežnih ter drugih temeljnih računalniških virov. Uporabnik na pridobljenih virih lahko zaganja poljubno programsko opremo, kar vključuje tudi operacijski sistem. Nima pa nadzora nad infrastrukturo oblaka, mogoče le delni nadzor nad omrežno komponento (npr. požarni zid).

Platforma kot storitev (Platform as a Service, PaaS) Zmožnost namestitve in izvajanja lastne aplikacije, razvite z uporabo programskih jezikov in orodij, ki jih podpira ponudnik. Uporabnik nima nadzora nad spodaj ležečo infrastrukturo (omrežje, shramba, računske zmogljivosti, operacijski sistem), le delni nadzor prek konfiguracijskih nastavitev okolja.

Programska oprema kot storitev (Software as a Service, SaaS) Zmožnost uporabe ponudnikove aplikacije, ki se izvaja na oblačni infrastrukturi, prek tankega odjemalca (npr. spletnega brskalnika) ali programskega vmesnika. Uporabnik nima nadzora niti nad spodaj ležečo infrastrukturo niti nad samo aplikacijo. Nadzor ima le nad podatki, ki jih posreduje aplikaciji.

Čeprav je za oblake značilno, da nas njihova fizična narava ne zanima, pa to velja le z aplikacijskega stališča uporabe virov. Za končne uporabnike pa so odgovori na vprašanja, "kje se podatki hranijo, kdo skrbi za infrastrukturo, kdo odgovarja v primeru izpada itd." zelo pomembni, še posebno v luči varovanja osebnih in poslovnih podatkov, skladnosti z zakonodajo, odgovornosti v primeru izpada itd. Tako računalništvo v oblaku ločimo glede na postavitveni model, ki oblake deli glede na to, komu je oblak namenjen in kdo skrbi za infrastrukturo. Štirje postavitveni modeli so naslednji:

Privatni oblak (Private cloud) Infrastruktura je namenjena in dostopna točno določeni organizaciji za lastne potrebe. Le-ta si jo običajno tudi lasti in jo vzdržuje, nad njo pa ima popoln nadzor. Tipični uporabniki takega modela so banke, velika podjetja in institucije z varnostno občutljivimi podatki.

Skupnostni oblak (Community cloud) Porazdeljena infrastruktura se deli med več organizacij, ki imajo skupni interes in potrebo po takšni infrastrukturi. Infrastrukturo si lastijo vse organizacije, njeno vzdrževanje pa je običajno centralizirano. Največkrat gre za velike raziskovalne institucije, ki sodelujejo v globalnih projektih (npr. Cern, ESA).

Javni oblak (Public cloud) Infrastruktura je namenjena in dostopna širši javnosti. Vzdržuje jo komercialno podjetje, ki jo tudi trži. Ponudniki so velika svetovna IT-podjetja, uporabniki pa fizične osebe ali mala/srednja podjetja, ki ne premorejo lastne infrastrukture.

Hibridni oblak (Hybrid cloud) Je kombinacija dveh ali več ostalih postavitvenih modelov, ki imajo sicer lastno identiteto, a so povezani v logično celoto. Npr. ko potreba po virih preseže lastne zmogljivosti privatnega oblaka, se vire začne pridobivati v javnem oblaku.

Okolje, ki ga vzpostavlja računalništvo v oblaku, s svojimi bistvenimi značilnostmi nudi uporabniškim programom, ki se v takem okolju izvajajo (t. i. aplikacije v oblaku), določene koristne lastnosti, med drugim:

Zanesljivost Aplikacija se izvaja na infrastrukturi, ki je visoko redundantna in zaradi razpršenosti virov nima enotne točke odpovedi (angl. single point of failure). Ob delni odpovedi je možno samodejno okrevanje ob ponovni vzpostavitvi normalnega stanja.

Vseprisotnost Aplikacija je prek omrežja vedno dostopna na raznoliki množici naprav.

Vzdrževalnost (angl. maintainability) Aplikacija je nameščena le v oblaku in ne pri končnih uporabnikih, kar olajša njeno posodabljanje, testiranje, spremljanje delovanja in odpravljanje napak na enem mestu.

Skalabilnost Aplikacija ima na voljo neomejeno veliko virov, zato lahko z uporabo porazdeljenega računalništva prenese poljubno veliko breme (npr. število sočasnih uporabnikov, količina podatkov).

Elastičnost Aplikacija samodejno prilagaja porabo virov glede na trenutno breme in se tako lahko po potrebi poljubno širi oz. krči.

A dosegljivost teh lastnosti ni samoumevna, pač pa je potrebno za njihovo pridobitev znati prednosti okolja ustrezno izkoristiti. Namensko razvite aplikacije z zagotavljanjem teh lastnosti nimajo težav, a je njihov razvoj dostopen le ozkim strokovnjakom, saj je za ustrezno izkoriščenost treba poznati širok spekter zahtevnih tehnologij [3]. Dodatno oviro predstavlja heterogenost računalništva v oblaku, saj se ne glede na delitev storitvenih modelov ponudniki medsebojno lahko precej razlikujejo v načinih, kako infrastrukturo nuditi aplikacijam. Za razvoj aplikacij sta primerna le storitvena modela IaaS in PaaS, ki sta v nadaljevanju podrobneje predstavljena.

2.1.1 Razvoj aplikacij v okolju IaaS

IaaS je najosnovnejši storitveni model, ki z nizkim nivojem abstrakcije uporabniku nudi dostop do temeljnih računalniških virov (računsko moč, delovni spomin, diskovni prostor in mrežno infrastrukturo). Viri so na voljo v obliki virtualnega računalnika (angl. Virtual Machine, VM), na katerega se namesti eden izmed množice prednastavljenih operacijskih sistemov. Prek oddaljenega dostopa se VM nato uporablja kot običajni osebni računalnik. Upravljanje zaobjema dinamično povečevanje/zmanjševanje zmogljivosti posameznih virov obstoječemu VM, prižiganje/ugašanje VM, vzpostavitev novega (ali več) VM in nadzor porabe. Uporabnik omenjene akcije vrši ročno prek uporabniškega vmesnika ali pa programsko prek aplikacijskega programskega vmesnika (angl. Application Programming Interface, API).

Sčasoma se je pokazala potreba tudi po ostalih virih, ki dopolnjujejo samostojno VM z uporabnimi možnostmi. Ena največjih pomanjkljivosti pristopa nudenja virov samo na osnovi VM je namreč izguba podatkov na virtualnem disku ob ugasnitvi VM-ja. Tako je nastala storitev trajnega shranjevanja (angl. persistent storage), ki uporabniku zagotavlja ohranitev podatkov tudi ob ugasnjeni VM in je dostopna v obliki v omrežje priključene pomnilniške naprave (angl. network attached storage). Podobna storitev je tudi izdelava posnetka VM

(angl. snapshot), samodejna izdelava varnostne kopije itd. Naslednja uporabna dopolnitev VM je povezovanje ustvarjenih VM v ustrezno mrežno hierarhijo, npr. več mrežnih kartic znotraj ene VM, povezovanje v ločene notranje mreže itd. Tako danes pod pojmom IaaS poleg računskih virov v obliki VM govorimo tudi o shranjevalnih, mrežnih in ostalih virih (kamor sodijo predvsem različne oblike porazdeljenih nerelacijskih podatkovnih baz). Tudi pri teh upravljanje poteka prek ustreznih uporabniških vmesnikov ali pa programsko prek API-jev.

Razvoj aplikacij za oblak na osnovi IaaS ima naslednje značilnosti. Razvijalcem takšen model nudi največ kontrole nad okoljem, saj lahko določijo operacijski sistem VM in nanj namestijo poljubne aplikacije, podporne knjižnice, programska okolja itd. Tako je IaaS zelo primeren za prenos obstoječih aplikacij v oblak, saj zamenjava tehnologije ni potrebna. Hkrati velja, da je razvijalcu prepuščena celotna implementacija vseh potrebnih funkcionalnosti za dosego lastnosti, ki se jih od aplikacij v oblaku pričakuje. Razvoj teh funkcionalnosti se navezuje na API, prek katerega se upravlja z viri. Različni ponudniki storitve imajo različne API-je, zato takšne aplikacije niso neposredno prenosljive med ponudniki. Na srečo pa so si API-ji različnih ponudnikov medsebojno zelo podobni, kar je posledica nizkega nivoja abstrakcije in omejenega nabora možnih virov [34]. Tako je običajno pri prenosu z enega na drugega ponudnika potrebna le majhna prilagoditev aplikacije, obstajajo pa tudi knjižnice, ki znajo sodelovati z API-ji več ponudnikov, npr. libcloud[1].

Primeri javnih ponudnikov IaaS so Amazon Elastic Compute Cloud (EC2)[2], Google Compute Engine[3], HP Cloud[4], Rackspace Cloud[5] itd. Za vzpostavitev privatnega oblaka IaaS pa so na voljo komercialne (npr. OnApp[6], vCloud[7]) in tudi nekatere odprtokodne rešitve (npr. Eucalyptus[8] in OpenStack[9]).

2.1.2 Razvoj aplikacij v okolju PaaS

PaaS nudi višji nivo abstrakcije kot IaaS, saj uporabniku neposredno nudi le izvajalno okolje za aplikacije [34]. Potrebo po virih pri tem modelu predstavljajo prostorske in časovne zahteve aplikacije med izvajanjem ter opravljene vhodno/izhodne operacije. Viri se aplikaciji dode-

[1] http://libcloud.apache.org/
[2] http://aws.amazon.com/ec2/
[3] https://cloud.google.com/
[4] https://www.hpcloud.com/
[5] http://www.rackspace.com/
[6] http://onapp.com/
[7] http://vcloud.vmware.com/
[8] http://www.eucalyptus.com/
[9] http://www.openstack.org/

ljujejo samodejno, glede na potrebe, uporabnik določi le zgornjo mejo še dopustne uporabe. Ob doseženi meji aplikacija preneha z izvajanjem oz. izgubi dostop do določenih storitev. Pri modelu PaaS se tako kompleksnosti, povezane z dinamičnim dodeljevanjem temeljnih računalniških virov, nameščanjem vmesne programske opreme, ki jo aplikacije potrebujejo za svoje delovanje, in implementacijo lastnosti aplikacije v oblaku, preloži na ponudnika PaaS.

Za razvoj aplikacij v oblaku na osnovi PaaS vse zgoraj navedeno pomeni, da se razvijalci lahko bolj posvetijo samemu razvoju aplikacij in se ne ozirajo na vzpostavitev in upravljanje okolja, v katerem se bo aplikacija izvajala. Po drugi strani izgubijo večino nadzora nad tem okoljem, saj ne morejo vplivati na operacijski sistem ali nameščanje/nastavljanje uporabljenih podpornih knjižnic in programskih okolij. Dodatno se od njih zahteva uporaba točno določenega programskega jezika, orodij in tehnologij, ki jih definira ponudnik. Slednje je potrebno za dostop do storitev, ki jih običajno nudi okolje PaaS (shranjevanje podatkov, uporaba podatkovne baze, izmenjava sporočil, obdelava zahtevkov HTTP itd.). V nasprotju z modelom IaaS se pri PaaS programski vmesniki od ponudnika do ponudnika zelo razlikujejo, hkrati pa je posamezna rešitev velikokrat vezana na strojne zmogljivosti ponudnika (izvajalnega okolja ni mogoče prenesti na drugega ponudnika). Takšna situacija vodi v zaklep na posameznega ponudnika (angl. vendor lock-in), kar pa ni zaželeno niti priporočljivo.

Primeri javnih ponudnikov PaaS so Google AppEngine (GAE)[10], Windows Azure[11], Engine Yard[12], Heroku[13] itd. Na lastni infrastrukturi (ali na IaaS) pa so na voljo AppScale[14], CloudFoundry[15], OpenShift[16] in drugi.

2.2 Inženirske aplikacije v oblaku

Znanstveno računalništvo je veja računalništva, ki se ukvarja z reševanjem in analizo znanstvenih problemov s pomočjo računalnika. Ustreznim aplikacijam pravimo tudi inženirske aplikacije. Gre za napredne aplikacije, ki probleme rešujejo z numerično analizo, simulacijo itd. [39]. Njihov skupni imenovalec je ogromna potreba po računalniških virih, zato se za ustrezno moč zanašajo na visokozmogljivo računalništvo (superračunalniki in mrežno računalništvo). A takšna infrastruktura ni poceni in si jo lastijo le večje organizacije. Dostikrat je dostop omogočen tudi zunanjim raziskovalcem, a je predhodno potrebno dobiti

[10] https://developers.google.com/appengine/
[11] http://www.windowsazure.com/
[12] https://www.engineyard.com/
[13] https://www.heroku.com/
[14] http://www.appscale.com/
[15] http://www.cloudfoundry.com/
[16] https://www.openshift.com/

ustrezno dovoljenje za uporabo in si rezervirati termin, ko bo zahtevana infrastruktura na voljo. Za lažjo in hitrejšo opcijo se lahko izkaže uporaba oblaka, ki je dostopen širši množici, saj ne zahteva posebnih pooblastil (kreditna kartica je običajno dovolj) in je vedno na voljo. A pojavi se problem, kako takšne aplikacije prenesti v oblak. Obstoječo inženirsko aplikacijo v oblaku namreč obravnavamo kot zastarelo aplikacijo (angl. legacy application) [34], saj ne pozna oz. uporablja programskih vmesnikov okolja, nemalokrat pa vanjo tudi ni mogoče enostavno posegati.

2.2.1 Inženirske aplikacije v okolju IaaS

V okolju IaaS običajno lahko obstoječe aplikacije spravimo v delovanje brez večjih naporov, saj imamo nadzor nad celotnim izvajalnim okoljem. Tako lahko vedno izberemo kombinacijo operacijskega sistema, izvajalnega okolja in ostalih potrebnih podpornih komponent, ki ustreza naši aplikaciji brez potrebe po njenem spreminjanju. Tako prenesene aplikacije ne izkoriščajo vseh prednosti oblačnega okolja, saj ne znajo samodejno upravljati z viri. Za odpravo te pomanjkljivosti je potrebno razviti ločene mehanizme, ki uporabljajo API za dodeljevanje in upravljanje virov. Uspešen prenos je odvisen tudi od aplikacije same, predvsem če je skalabilna in če zna izkoriščati porazdeljene vire s pomočjo vzporednega računanja.

V [17, 20, 41] je narejen pregled uporabe in cen modela IaaS za potrebe inženirskih aplikacij in njihove performančne lastnosti. Izkaže se, da je računska moč v oblaku ustrezna, a je največja ovira in hkrati tudi glavna razlika med okoljem IaaS in standardnim visokozmogljivim računalništvom v omejeni zmogljivosti povezav med posameznimi VM. Medtem ko v okolju IaaS za povezavo skrbi klasična mrežna povezava z visoko latenco in precejšnjimi režijskimi stroški, je pri visokozmogljivem računalništvu običajno prisotna specialna mrežna infrastruktura, ki ima nizko latenco in visoko prepustnost (npr. Infiniband). Naslednja težava je prenos podatkov, ki jih aplikacija potrebuje, v oblak - zmogljivost prenosnih poti je omejena z zmogljivostjo internetne povezave tako uporabnika kot ponudnika oblaka. Največ uspeha v oblaku imajo tako trivialno paralelni problemi (angl. embarrassingly parallel), ki pri delovanju ne potrebujejo večje količine podatkov, npr. simulacije, razbijanje kriptografskih problemov, evolucijsko računanje ipd. [13, 19].

2.2.2 Inženirske aplikacije v okolju PaaS

Okolje PaaS predvideva, da bo aplikacija v oblaku komunicirala z okoljem le prek nudenih API-jev, ker le tako lahko avtomatsko zagotavlja določene lastnosti aplikacij v oblaku. Ti so lahko vključeni v programe le, če so napisani v določenih programskih jezikih in če se

pri delovanju zanašajo na točno določena podporna ogrodja. V primeru neuporabe API-jev pa se različna okolja PaaS medsebojno zelo razlikujejo: nekatera takšnih aplikacij sploh ne podpirajo, druga pa je mogoče z večjimi ali manjšimi posegi prepričati v delovanje, a končna rešitev običajno ne vsebuje vseh lastnosti aplikacij v oblaku.

Med okolja, ki ne podpirajo zastarelih aplikacij, sodijo praktično vsi komercialni ponudniki. Tipičen primer je eno izmed najbolj priljubljenih komercialnih okolij PaaS - GAE. Ta od aplikacij izrecno pričakuje uporabo nudenih API-jev (dosegljivi so v delno okrnjenih programskih jezikih Python, Java in Go), sicer aplikacije sploh ni mogoče namestiti in jo zagnati. Odprtokodna okolja delujejo podobno, a imajo to prednost, da jih lahko namestimo na lastni infrastrukturi. AppScale je klon GAE, ki ga razvija skupnost, in nudi večino njegovih funkcionalnosti. CloudFoundry razvija VMWare in vzpostavlja svoje okolje PaaS, ločeno od GAE, ki podpira Javo, Ruby, PHP, Node.js, Scalo in nekatera druga ogrodja, temelječa na javanskem navideznem stroju. Podobno velja tudi za OpenShift, ki ga razvija Redhad in ki podpira programske jezike Node.js, Ruby, Python, PHP, Perl in Java.

Za vsa našteta okolja velja, da so namenjena predvsem izvajanju skalabilnih spletnih aplikacij brez podpore za izvajanje zastarelih aplikacij. Pri tistih okoljih, ki se izvajajo na ponudnikovi infrastrukturi, do katere nimamo dostopa, in pri tistih, ki so zaprto-kodne narave, to pomeni, da niso primerna za izvajanje takšnih aplikacij, saj v njih ni mogoče posegati in vzpostaviti potrebne podpore. Pri odprtokodnih projektih vedno ostaja zmožnost izvajanja zastarelih aplikacij, a jo je potrebno implementirati. To pa zahteva velike posege v okolje, zato trenutno še ni zaslediti razvoja v tej smeri.

Obstaja pa tudi nekaj izjem, ki so prijaznejše do izvajanja obstoječih aplikacij v okolju PaaS.

Stackato[17] je komercialno okolje PaaS, ki ga razvija ActiveState in ki ga je mogoče vzpostaviti na poljubni infrastrukturi. Podpira vse programske jezike, saj ne deluje prek API-jev, pač pa zahteva, da se celotna aplikacija (izvršljiva koda z vsemi potrebnimi knjižnicami) združi v paket t. i. droplet, ki ga poganja računski pogon DEA (Droplet Execution Agent). Le-ta potem poskrbi na nadzorovano izvajanje aplikacije in delno doseganje lastnosti oblačnih aplikacij. Za čim bolj uspešen prenos je vseeno priporočljivo zamenjati nekatere pogoste storitve s predpripravljenimi, saj tako bolje izkoristimo okolje.

Aneka[18] je komercialno okolje PaaS na osnovi ogrodja .NET, ki ga razvija Manjrasoft. Rešitev se osredotoča predvsem na povezovanje najrazličnejših heterogenih arhitektur (od osebnega računalnika do mrežnega računalništva) v logično celoto, ki jo je nato prek API-jev

[17] http://www.activestate.com/stackato
[18] http://www.manjrasoft.com/

mogoče izkoristiti v celoti. Okolje podpira več programskih modelov ter ima vključeno tudi delno podporo za zastarele aplikacije [40].

Posebno okolje predstavlja tudi Windows Azure, ki ga razvija Microsoft. Gre za prilagodljivo platformo za oblačno računalništvo, ki se izvaja na njihovih strežnikih ter zaobjema celo paleto storitev, ki segajo od IaaS prek skalabilnih spletnih aplikacij do namenskih gostovanih storitev. Rešitev PaaS je na voljo prek posebnih programskih razvojnih orodij, ki podpirajo programske jezike Python, Java, Node.js in .NET, vendar spet ne nudijo neposredne podpore za zastarele aplikacije. A ker platforma hkrati vsebuje rešitev IaaS, jo lahko izkoristimo za poganjanje inženirskih aplikacij, medtem ko jih nadzorujemo prek plasti PaaS. Primer uporabe je podan v [25].

Prednosti okolja PaaS so vodile tudi v razvoj specialnih okolij PaaS, ki so namenjena le točno določenim izračunom. Primer takšnega okolja je podan v [5], kjer so predstavili lastno okolje BioDAP za analizo bioloških podatkov. Okolje sicer ne omogoča poganjanja zastarelih aplikacij, a imajo uporabniki na voljo večino orodij, ki se jih na tem področju uporablja, zato sam razvoj ni tako zahteven. Podobno okolje, namenjeno analizi satelitskih posnetkov, je predstavljeno v [24].

2.3 Matlab v oblaku

V delu smo se osredotočili le na prenos inženirskih aplikacij, razvitih v okolju Mathworks Matlab, zato v tem podpoglavju podrobneje navajamo možnosti za izvajanje takšnih aplikacij v oblaku.

Matlab je zmogljivo orodje za preračunavanje in vizualizacijo numeričnih podatkov, ki je namenjeno predvsem inženirjem, akademikom in znanstvenikom, ki ga uporabljajo na najrazličnejših področjih: od razvoja v avtomobilski industriji, preizkušanja letalskih in vesoljskih modelov, prepoznavanja govora in slik do napovedovanja vremena, analize borznih tečajev in drugega, kar zahteva preračunavanje, simuliranje ali vizualizacijo podatkov.

Orodje se uporablja s podajanjem ukazov oziroma stavkov v jeziku Matlab. Jezik je posebej prilagojen za učinkovito in naravno opisovanje matrik in operacij nad njimi, hkrati pa omogoča tudi delo s spremenljivkami, kontrolne stavke, zanke, delo z vhodom/izhodom, definiranje funkcij itd. in tako sodi med t. i. visokonivojske programske jezike četrte generacije. Za razvoj programov, njihovo razhroščevanje, testiranje in izvajanje se običajno uporablja grafično razvojno okolje Matlab. Izvrševanje programa temelji na interpretiranju stavkov, ki ga izvaja računski pogon Matlab (Matlab engine). Poleg interakcije prek grafičnega okolja je mogoče računski pogon uporabiti tudi neinteraktivno prek vmesnika z ukazno vrstico (angl.

command line interface).

Osnovni nabor ukazov jezika se lahko razširi z različnimi dodatki, imenovanimi orodjarna (angl. toolbox), ki vsebujejo množico funkcionalnosti, ki so posebej prilagojene za določena področja, kot npr. statistična obdelava, optimizacijske metode, digitalno procesiranje signalov, simulacije in modeliranje itd. Samo podjetje MathWorks ponuja več kot 30 orodjarn, še precej več, tako plačljivih kot brezplačnih, pa jih je na voljo iz drugih virov.

Ena glavnih pomanjkljivosti Matlaba je njegova cena, saj potrebujemo ustrezno licenco za računski pogon, hkrati pa dodatne licence za vsako orodjarno. Licence so žal vezane na število uporabljenih računalnikov, in zato neprimerne za oblak. V zadnjem času se sicer vzpostavlja nov način licenciranja v povezavi z Amazon EC2[19], vendar je projekt v začetni fazi in je na voljo le izbranim strankam. Obstajajo tudi brezplačne alternative računskega pogona Matlab (npr. GNU Octave[20] in Scilab[21]), ki podpirajo praktično celotno sintakso jezika Matlab, imajo pa zato toliko slabšo podporo, kar zadeva orodjarne. Distribuiranje programa v Matlabu na način, ki ne zahteva ustrezne licence, je zahteven proces, eden izmed možnih načinov pa je opisan v naslednjem podpoglavju.

2.3.1 Matlab v okolju IaaS

Pri IaaS-u lahko uporabljamo obstoječi program brez potrebe po spreminjanju, če nanj namestimo ustrezno okolje Matlab. Še pred srečanjem z že omenjenim problemom licenc je potrebno za uspešno delovanje Matlaba na taki infrastrukturi razrešiti drug problem. Matlab v svojem osnovnem načinu namreč deluje kot enonitni proces in kot tak ne zna polno izkoristiti današnjih standardnih večjedrnih procesorjev v osebnih računalnikih, kaj šele množico VM, povezanih v gručo.

Zmožnost delovanja okolja Matlab na porazdeljeni infrastrukturi razširja plačljiva orodjarna Parallel Computing Toolbox [29], ki omogoča reševanje računsko in podatkovno intenzivnih problemov z večjedrnimi procesorji, zmogljivimi grafičnimi procesorji (GPU) in tudi gručami. Orodjarna jezik Matlab dopolni s potrebnimi višjenivojskimi strukturami, kot so npr. paralelne zanke (angl. parallel for-loops), specifične vrste porazdeljenih matrik ter zbirka paralelno implementiranih numeričnih algoritmov, in tako omogoča razporeditev izvajanja med več računskimi pogoni Matlab. Do dvanajst hkratnih računskih pogonov lahko poganjamo lokalno le z uporabo orodjarne, za uporabo več pogonov oz. za izrabo porazdeljene infrastrukture pa je treba uporabiti produkt Matlab Distributed Computing Server

[19] http://www.mathworks.com/programs/mdcs-cloud.html
[20] http://www.gnu.org/software/octave/
[21] http://www.scilab.org/

(MDCS). Podrobna navodila in analiza delovanja takega pristopa na oblaku Amazon EC2 so podani v [26]. Z malenkostnimi popravki se pristop lahko uporabi tudi pri ostalih ponudnikih IaaS.

Opisan pristop je eden izmed najlažjih načinov uporabe oblaka IaaS za programe v Matlabu, vendar pa uporaba MDCS zahteva ustrezno licenco za vsak uporabljen računski pogon, hkrati pa je konfiguracija posameznega pogona časovno potratna naloga. Če želimo infrastrukturo uporabiti le za kratke izračune ali če želimo določeno breme pognati le enkrat, je precej boljša opcija uporaba že postavljene infrastrukture, ki zahteva minimalno konfiguracijo. Eden takih sistemov je Red Cloud.[22]

Glavni problem pri uporabi MDCS je, da končni uporabnik še vedno potrebuje okolje Matlab, ker je le-to nujno za razporejanje bremen med posameznimi računskimi pogoni in za obdelavo rezultatov. Boljši način, ki bi bil primeren za širšo javnost, bi bila uporaba uporabniškega vmesnika, ki bi uporabniku prikril uporabo Matlaba in od njega ne bi zahteval uporabe plačljivega okolja. Primer uporabe preprostega spletnega vmesnika, ki v ozadju uporablja računski pogon neinteraktivno, je naveden v [36].

Žal tudi ta pristop še vedno zahteva uporabo ustrezne licence na strežniški strani. Eden izmed načinov, da se temu izognemo, je, da uporabimo orodjarno Matlab Compiler [28], ki osnovno okolje razširi s prevajalnikom. Le-ta iz programa Matlab zgradi binarno kodo v obliki samostojne izvršljive datoteke ali dinamične knjižnice, ob uporabi ustreznih dodatkov pa tudi razredov Java ali .NET. Binarna datoteka, ki je izhod iz prevajalnika, ima, ne glede na izhodno obliko, enake numerične lastnosti kot vhodni program, a za izvajanje ne potrebuje več računskega pogona Matlab. Za izvajanje zadošča Matlab Compiler Runtime (MCR), zbirka dinamičnih knjižnic, ki so na voljo brezplačno. Natančnejša navodila, kako povezati program Matlab s spletnim vmesnikom ob pomoči prevajalnika, so podana v [27].

Z uporabo prevajalnika orodjarna Parallel Computing Toolbox postane posredno neuporabna, ker ne more razporejati bremena med računske pogone Matlab, saj le-ti niso na voljo. Naloga razporejanja bremena tako pade na razvijalce, ki morajo vzpostaviti ustrezen sistem komunikacije med posameznimi deli porazdeljenega okolja. Običajno se v takem okolju uporablja vmesnik za prenašanje sporočil (angl. Message Passing Interface, MPI) [37]. Njegova neposredna uporaba v okolju Matlab ni mogoča, a je dosegljiva z različnimi orodjarnami, ena izmed njih je brezplačna Matlab MPI [22]. Posredno pa je s pomočjo prevajalnika mogoče uporabiti tudi katerokoli drugo pogosto implementacijo MPI oziroma tudi kakšen drug protokol za izmenjavo sporočil.

Uporaba Matlaba v porazdeljenem okolju, ki ga nudi model IaaS, zahteva posebne pro-

[22] http://www.cac.cornell.edu/redcloud/

gramske vzorce, paralelne algoritme in skrbno načrtovanje. Celoten pregled uporabe v takem okolju najdemo v [21] in [6].

2.3.2 Matlab v okolju PaaS

Okolje PaaS je za Matlab zelo neprijazno iz enakih razlogov, kot veljajo za splošne inženirske aplikacije. Inovativen pristop je podan v [11]. Podjetje Techila Technologies[23] je razvilo vmesno programsko plast (angl. middleware), ki zna nadzirati različna opravila v porazdeljenem okolju. Upravljalni del programske plasti uporablja del okolja Windows Azure PaaS in skrbi za razporejanje in nadzor opravil ter komunikacijo med njimi in zunanjim okoljem. Opravila pa se pri izvajanju zanašajo na del okolja IaaS. Uporabniki najprej prek posebnega vmesnika namestijo svoja opravila, nato pa jih lahko izvajajo in jim posredujejo podatke. Rešitev omogoča opravila, napisana v veliko programskih jezikih, med drugim tudi opravila, napisana v Matlabu. Da je vsa stvar uporabniku še bolj prijazna, je na voljo tudi poseben API, ki omogoča izvajanje opravil neposredno prek lokalno nameščenega okolja Matlab.

[23] www.techilatechnologies.com

3 mOSAIC

V okviru magistrskega dela smo si kot cilj zastavili prenos obstoječih inženirskih aplikacij v oblak z odprtokodno platformo mOSAIC (Open Source API and Platform for Multiple Clouds, odprtokodni programski vmesnik in platforma za več oblakov)[1] [45].

Gre za platformo, ki zna:

- zgraditi okolje PaaS nad okoljem IaaS, zato so aplikacijam na voljo prednosti obeh okolij,
- zagotoviti popolno neodvisnost aplikacij od ponudnika IaaS,
- zagotoviti podporo aplikacijam, pisanim v poljubnem programskem jeziku, in
- pridobiti potrebne vire od več ponudnikov.

Takšna rešitev obljublja veliko prednosti pred ostalimi rešitvami PaaS. Poglavitna je, da gre za okolje PaaS, ki je neodvisno od uporabljenega programskega jezika ter hkrati tudi od posameznega ponudnika IaaS. S tem je aplikacijam zagotovljena popolna avtonomija, zato se pri njihovem razvoju ni treba obremenjevati s tem, kje se bo izvajala. Prenos aplikacije z enega na drugega ponudnika je mogoče narediti brez poseganja v aplikacijo. Izbira poljubnega ponudnika prinaša prednost tudi v smislu minimizacije stroškov, saj lahko v vsakem trenutku zamenjamo ponudnika IaaS z ugodnejšim. Nadalje omogoča poln nadzor nad osnovno programsko opremo (angl. enabling software), tj. operacijski sistem, podporne knjižnice in programska okolja, in tudi njeno spreminjanje za potrebe izvajanja aplikacij. S tem se lahko pri razvoju uporabi tudi zastarele aplikacije brez bojazni, da bi izgubili prednosti modela PaaS.

Hkrati platforma vsebuje orodja in storitve, ki uporabnikom olajšajo njeno uporabo - načrtovanje, razvoj in izvajanje aplikacije, nadzor nad platformo, pomoč pri pridobivanju virov itd.

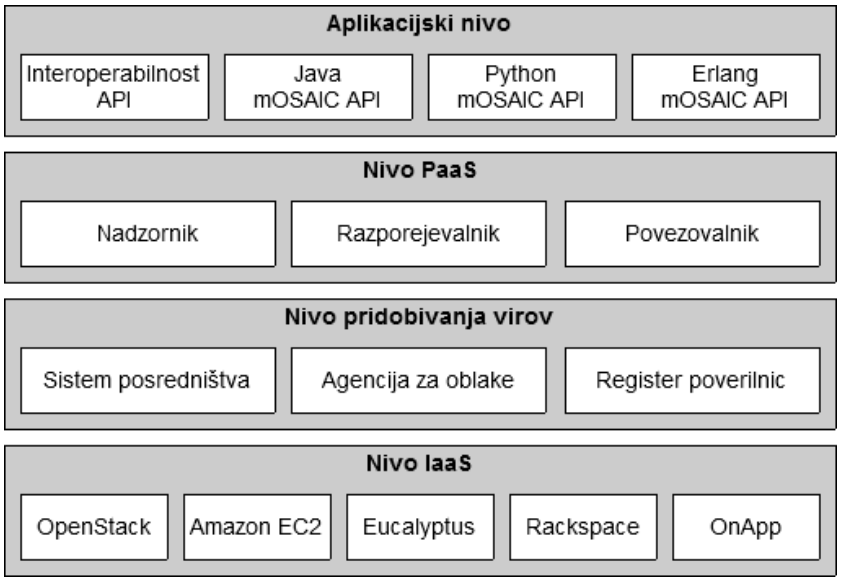
Najprej je v podpoglavju 3.1 predstavljena zgradba platforme mOASIC. Sledi podpoglavje 3.2, kjer je predstavljen programski model, ki ga uveljavlja platforma mOSAIC. Podrobno

[1] Nastala v okviru istoimenskega evropskega projekta znotraj 7. okvirnega programa.

delovanje platforme je opisano v podpoglavju 3.3. V naslednjem podpoglavju (3.4) je opravljen pregled najpomembnejših storitev, ki so v platformo mOSAIC že vgrajene. Na koncu poglavja (podpoglavje 3.5) sledi še seznam orodij, ki smo jih pri svojem delu potrebovali.

3.1 Zgradba platforme

Za doseganje predstavljenih lastnosti, opisanih v uvodu poglavja, je platforma zgrajena v obliki večnivojskega sklada. Shematsko je prikazan na sliki 3.1.



Slika 3.1: Arhitektura platforme mOSAIC.

Najnižji nivo je namenjen podpori API-jev različnih ponudnikov IaaS. Podpora za najbolj priljubljene je v platformo že vključena in obsega Amazon EC2, Eucalyptus, OpenStack, Rackspace, FlexiScale in druge [35]. Podporo posameznih ponudnikov v platformi zagotavlja ustrezen gonilnik ponudnika, imenovan VendorModule. Njegov namen je preslikava ponudnikovega API-ja na skupni imenovalec - upravljanje virov po zgledu odprtokodnega vmesnika za oblačno računalništvo (angl. open cloud computing interface, OCCI [31]). Ta preslikava nam omogoča medsebojno primerljivost med posameznimi ponudniki ter enoten dostop do virov brez poznavanja specifik posameznega API-ja. Množico podprtih ponudnikov IaaS je vedno mogoče razširiti z razvojem novega ustreznega gonilnika. Zaradi podobnosti med različnimi API-ji na nivoju IaaS je tak razvoj izvedljiv brez pretiranega napora. Za namen dela smo tako razvili tudi gonilnik za ponudnika IaaS Hostko, ki temelji na programski opremi OnApp. Razvoj je podrobneje razložen v dodatku A.

Naslednji nivo je namenjen podpori pridobivanja potrebnih virov od enega ali več ponu-

dnikov IaaS. Sistem deluje s pomočjo naprednega sistema posredništva več-agentnega sistema (angl. multi-agent brokering system). Platforma mOSAIC lahko samodejno pridobi ustrezne vire s pomočjo vgrajenega orodja, imenovanega Agencija za oblake (angl. Cloud Agency, CA), ki je natančneje opisana v podpoglavju 3.5. Nivo hrani tudi bazo poverilnic (angl. credential) za avtorizacijo dostopa do posameznih ponudnikov IaaS, od katerih pridobivamo vire.

Tretji nivo zagotavlja glavne funkcionalnosti tipičnega okolja Paas. Na nižjem nivoju povezuje pridobljene vire v celoto - gručo, ki zagotavlja računsko moč, na drugi strani pa nadzoruje izvajanje aplikacij, jim zagotavlja samodejno skalabilnost in elastičnost ter razvršča aplikacije po razpoložljivi infrastrukturi. Prek funkcionalnosti spodnjega nivoja skrbi, da je vedno na voljo dovolj virov. Delovanje je natančneje opisano v podpoglavju 3.3.

Četrti nivo je namenjen programskim vmesnikom. Glavni je programski vmesnik za interoperabilnost (angl. Interoperability API), ki aplikacijam omogoča medsebojno komunikacijo, ne glede na to, kje na gruči se nahajajo in v katerem programskem jeziku so napisane. Sledijo implementacije programskega vmesnika mOSAIC (angl. mOSAIC API) za posamezni programski jezik. Platforma trenutno podpira programske jezike Java, Phyton, Erlang in Node.js, vanjo pa je vedno mogoče vključiti implementacijo za ostale jezike. Pri razvoju aplikacije, ki bo znala izkoristiti vse prednosti platforme, je tako treba uporabljati vmesnik v ustreznem jeziku. Nivo določa tudi uporabljene programske modele in je predstavljen v nadaljevanju.

3.2 Programski model

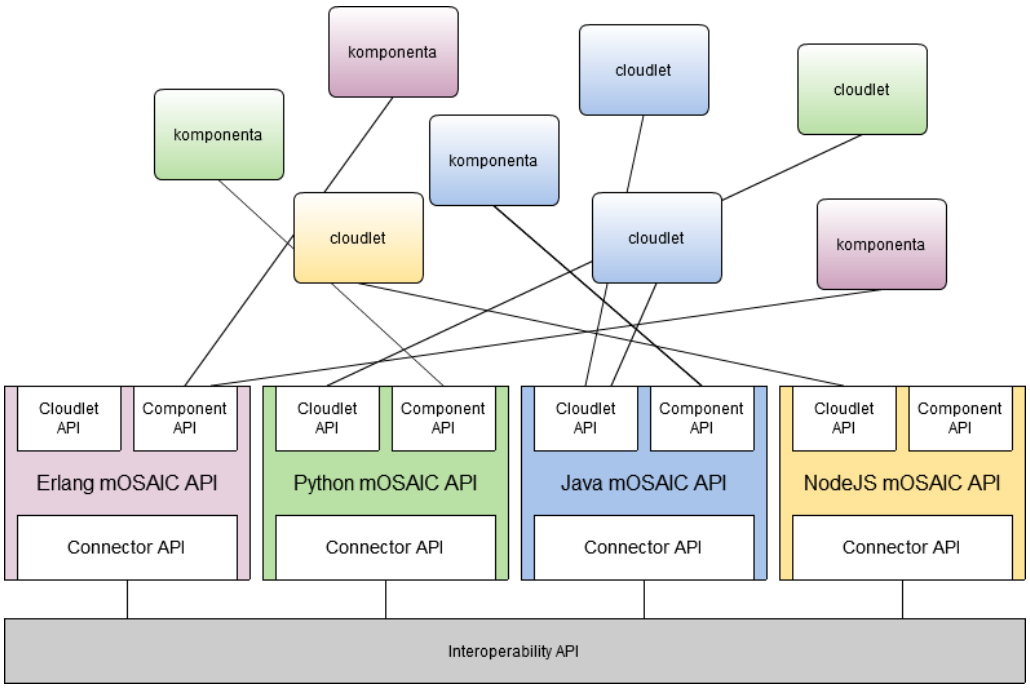
Platforma mOSAIC celotno aplikacijo v oblaku vidi kot skupek ohlapno povezanih sestavnih delov, imenovanih komponente mOSAIC. Komponenta mOSAIC je definirana kot osnovni gradnik, ki ima svojo natančno določeno funkcionalnost, ki jo definira razvijalec in ki implementira točno določen vmesnik - Component API, del mOSAIC API. Instance komponent se izvajajo znotraj nadzorovanega okolja, kjer celoten življenjski cikel instance nadzira platforma. Platforma je tako odgovorna za njen zagon, ustavitev ter komunikacijo z ostalimi komponentami ali zunanjimi storitvami. S popolnim nadzorom nad vsemi instancami vseh komponent na celotni infrastrukturi lahko platforma dinamično prerazporeja vire med posameznimi instancami in tako zagotavlja optimalno delovanje celotne aplikacije ob minimalni porabi virov.

Notranja arhitektura celotnega programskega vmesnika mOSAIC je v skladu z modelom asinhronega vhoda/izhoda (angl. asynchronous input/output) in modelom dogodkovno vo-

denega programiranja (angl. event-driven programming). Oba pristopa omogočata, da ob nizkem številu niti dosežemo neblokirajoč (angl. non-blocking), visoko učinkovit sistem. A sistem lahko dobro deluje le, če je komponenta mOSAIC zgrajena v skladu z istimi principi. Komponenta tako ne sme povzročati blokirajočih vhodno/izhodnih operacij, pač pa mora uporabiti princip izvedbe neblokirajočega klica in hkratne registracije povratnega klica (angl. callback). Prav tako ni priporočljivo, da komponenta ustvarja svoje niti, saj na njih temelji mehanizem razporejanja komponent po infrastrukturi, ki pa ne deluje optimalno ob hitrem spreminjanju le-teh.

Kljub naštetim omejitvam pa razvoj največkrat ni okrnjen, saj so najpogostejši protokoli vhoda/izhoda, ki se pojavljajo pri računalništvu v oblaku, na voljo prek programskega vmesnika za povezovanje (angl. Connector API). Z ustreznimi dopolnitvami pa lahko implementiramo tudi lastne protokole. Že vgrajeni protokoli omogočajo izmenjavo sporočil prek sporočilnih vrst, pretok podatkov s porazdeljeno podatkovno bazo in delo z datotekami v porazdeljenem datotečnem sistemu. Za delno odpravo omejitve števila niti v komponenti je na voljo delovna nit (angl. worker thread), ki lahko prevzame zahtevno, dalj časa trajajoče breme in s tem zagotovi odzivnost komponente na ostale dogodke platforme.

Posebna vrsta komponent mOSAIC so tiste, ki implementirajo še dodatni vmesnik - Cloudlet. Predstavljajo najvišji nivo s platformo skladnih komponent, saj jim platforma



Slika 3.2: Zgradba mOSAIC API.

zagotavlja samodejno razširljivost (povečanje/zmanjšanje števila instanc posameznega Cloudleta glede na breme), odpornost na napake in avtonomijo. Takšne lastnosti zna platforma doseči le tako, da med izvajanjem ne razlikuje med posameznimi instancami istega Cloudleta. To pomeni, da je želena komunikacija s Cloudletom lahko preusmerjena h katerikoli instanci brez možnosti uporabnikovega vpliva na odločitev. Zato je zelo pomembno, da je Cloudlet napisan na način, da je njegovo izvajanje popolnoma neodvisno od trenutnega stanja in števila instanc.

Celotno zgradbo mOSAIC API z vsemi podvmesniki in relacijami z ostalimi nivoji hierarhije mOSAIC prikazuje slika 3.2.

3.3 Delovanje platforme

Za vzpostavitev platforme mOSAIC je potrebno na operacijski sistem GNU/Linux namestiti ustrezno programsko opremo. Uporabi se lahko poljubna distribucija operacijskega sistema (npr. CentOS, Ubuntu), na voljo pa je tudi posebna distribucija, imenovana mOS (mOSAIC operating system), ki temelji na prosto dostopni minidistribuciji Slitaz[2] brez grafičnega okolja. Njena prednost je v tem, da je optimizirana za hitro in tekoče delovanje tudi na slabše zmogljivi strojni opremi, predvsem pa za svoje delovanje ne porabi veliko virov. Celotna programska oprema je zaradi obsežnosti modularno zasnovana. Osnova se namesti neposredno na operacijski sistem (v mOS je osnova že nameščena), posamezni moduli pa so na voljo v obliki paketov Tazpkg[3], privzeti standard za distribucijo paketov v okolju Slitaz. Vsi paketi so prosto dostopni v centralnem skladišču (angl. repository)[4], mogoče pa jih je prenesti na lastno infrastrukturo in ustvariti privatno skladišče. Glavni namen modularne zasnove je, da se namestijo le tisti moduli, ki jih potrebujemo, s čimer prihranimo razpoložljive vire. Občutno se skrajša tudi čas, potreben za namestitev programske opreme, kar je še posebno opazno pri vzpostavitvi platforme na novo ustvarjeni VM.

Več VM z nameščeno programsko opremo se zna ob primerni mrežni infrastrukturi[5] samodejno povezati v gručo (angl. cluster), lahko pa se jo vzpostavi tudi ročno. Gruča deluje po principu enak z enakim (angl. peer to peer), kar pomeni, da ni centralnega strežnika, ki bi nadziral celotno gručo. Tak način združevanja VM v gručo omogoča redundantno arhitekturo, odporno na napake, omogoča pa tudi enostavno prilagajanje njene velikosti glede na obremenjenost z dodajanjem ali odvzemanjem VM.

[2] http://www.slitaz.org/
[3] http://www.slitaz.org/en/packages/
[4] ftp://ftp.info.uvt.ro/mosaic/mos/
[5] Mrežna infrastruktura mora podpirati razpršeno oddajanje (angl. broadcasting).

Celotno gručo v enotno platformo združujeta dva podsistema. Prvi je porazdeljeni razporejevalnik (angl. scheduler), ki zna na podlagi razpoložljivih virov dinamično prerazporejati posamezne komponente med posameznimi VM-ji, drugi pa porazdeljeni nadzornik (angl. controller), ki bdi nad izvajanjem komponent in njihovimi komunikacijami. S stališča komponent je torej celotna gruča videti kot enotna platforma in tako ne vidi razlike med posameznimi VM-ji oz. če gruča vsebuje eno ali več VM-jev.

Upravljanje platforme se izvaja prek spletne storitve, ki temelji na arhitekturi REST (REpresentational State Transfer) in je vzpostavljena na vsaki VM. Z izmenjavo ustreznih sporočil v formatu JSON (JavaScript Object Notation) je prek protokola HTTP mogoče zagnati ali ugasniti posamezne komponente, pridobiti seznam izvajajočih komponent, pregledati njihove dnevnike ali pa jim poslati sistemsko sporočilo (npr. poizvedba o statistiki uporabe). Pri komunikaciji s spletno storitvijo na določeni VM nismo omejeni le na stanje te VM in komponent, ki se na njej izvajajo, pač pa lahko prek nje upravljamo celotno platformo.

3.4 Vgrajene komponente

Nekatere storitve, ki jih pogosto uporabljajo aplikacije v oblaku, morda že zagotavlja ponudnik IaaS kot dodatno ponudbo poleg VM, npr. porazdeljeni datotečni sistem, porazdeljena podatkovna baza, sporočilne vrste itd., vendar se te storitve od ponudnika do ponudnika razlikujejo ali pa sploh niso na voljo. V takih primerih je potrebno želene storitve vzpostaviti z ustrezno programsko opremo, ki se izvaja na zakupljeni infrastrukturi. Platforma mOSAIC nudi širok spekter takih storitev, ki so na voljo v obliki komponent mOSAIC in jih je torej mogoče obravnavati na enak način kot ostale komponente aplikacije, obenem je njihova razpoložljivost neodvisna od njihove podpore s strani posameznega ponudnika IaaS. Vsaka vgrajena storitev ima tudi ustrezen gonilnik, tako da je enostavno dosegljiva ostalim komponentam prek programskega vmesnika mOSAIC.

Vgrajene komponente, ki jih uporabljamo v našem delu, so opisane v nadaljevanju podpoglavja.

3.4.1 Komponenta shranjevanja

Komponenta nudi možnost zapisovanja oziroma branja podatkov v podatkovni bazi po principu ključ-vrednost (angl. Key-Value, KV). Temelji na odprtokodni programski opremi Riak[6], ki implementira visokozmogljivo porazdeljeno podatkovno bazo, temelječo na porazdeljeni razpršilni tabeli. Sodi med t. i. podatkovne baze NoSQL (not-only SQL) in je odporna

[6] http://wiki.basho.com/

na napake ter poljubno skalabilna, a ne zagotavlja takojšnje konsistentnosti. Zapisani podatki mogoče niso takoj na voljo za branje, ampak šele po določenem času. Ena instanca komponente te storitve predstavlja eno vozlišče podatkovne baze in ne omogoča odpornosti na napake, več zagnanih instanc pa se zna samodejno povezati v enotno gručo Riak, ki ima ustrezno redundantnost podatkov (podatki z istim ključem so shranjeni na več vozliščih) ter hkrati višjo zmogljivost pri dostopu do podatkov.

3.4.2 Komponenta sporočilnih vrst

Komponenta nudi možnost pošiljanja in sprejemanja sporočil prek sporočilnih vrst (angl. message queue, MQ). Temelji na odprtokodni programski opremi RabbitMQ[7], ki implementira porazdeljeni sistem posredovanja sporočil (angl. message broker system) na osnovi naprednega protokola za sporočilne vrste (angl. Advanced Message Queuing Protocol, AMQP). Sistem temelji na principu pošiljatelja in prejemnika. Prvi sporočilo generira in ga pošlje sistemu, sistem ga na podlagi notranjih pravil razporedi v pravilno sporočilno vrsto. V vrsti sporočilo čaka toliko časa, dokler ga ne prevzame in obdela prejemnik (le-ta je o sporočilu obveščen). Poglavitna prednost implementacije RabbitMQ je v tem, da sporočilnih vrst in pravil razporejanja ni potrebno navesti ob zagonu sistema, pač pa jih lahko naknadno doda pošiljatelj ali prejemnik. Ena instanca komponente te storitve predstavlja eno vozlišče sistema, več zagnanih instanc skupaj pa tvori visoko zmogljiv sistem, odporen na izpade posameznih vozlišč.

3.4.3 Komponenta prehoda HTTP

Komponenta prehoda HTTP (angl. HTTP gateway) omogoča prenos prometa HTTP med zunanjim svetom (npr. internet) in komponentami mOSAIC. Deluje kot strežnik HTTP, ki na prednastavljenih vratih posluša zahtevke HTTP, ki pridejo od zunaj. Komponenta prejetega zahtevka ne obdela, pač pa ga vključi v sporočilo in ga posreduje naprej prek komponente za sporočanje. Tam je v določeni sporočilni vrsti na voljo komponentam, ki se prijavijo kot prejemniki te vrste. Komponenta, ki prejme sporočila, je zadolžena za obdelavo originalnega zahtevka HTTP, ki je vsebovan v sporočilu. Rezultat obdelave mora biti pretvorjen v odziv HTTP, ki ga mora komponenta vključiti v novo sporočilo in posredovati v novo sporočilno vrsto (določa jo prejeto sporočilo). Na tej vrsti je kot potrošnik prijavljena komponenta prehoda HTTP, ki ob prejemu ustreznega sporočila odziv HTTP posreduje pošiljatelju začetnega zahtevka. Opisani mehanizem obdelave zahtevkov HTTP je skalabilen (kot prejemnik sporočil

[7] http://www.rabbitmq.com/

z zahtevki se lahko prijavi mnogo komponent) in hkrati deluje kot porazdeljevalec bremena (angl. load balancer), saj se zahtevki v MQ porazdeljujejo med prijavljenimi komponentami po principu Round-Robin.

3.4.4 Komponenta strežnika spletnih strani

Komponenta strežnika spletnih strani omogoča enostavno vključitev streženja spletnih strani na podlagi zahtevkov HTTP, ki jih posreduje komponenta prehoda HTTP. Temelji na odprtokodnem aplikacijskem strežniku Jetty[8], ki zna streči tako spletne strani s statično kot dinamično vsebino, slednje na podlagi tehnologije javanskih servletov (angl. Java Servlet). Komponenta tako razvijalcu omogoča, da razvije želeno spletno stran v poznanih, standardnih tehnologijah, brez poznavanja podrobnosti ostalih komponent, ki sodelujejo pri prenosu prometa HTTP med uporabnikom in spletnim strežnikom. Dodatno komponenta omogoča dostop do mOSAIC API-ja. Tako lahko razvijalec spletne strani dostopa do ostalih komponent ali storitev platforme mOSAIC, kar naredi komponento zelo primerno za razvoj spletnih uporabniških vmesnikov.

3.4.5 Komponenta oddaljenega izvajanja

Komponenta oddaljenega izvajanja (angl. remote execution) omogoča prenos vse komunikacije, ki poteka med platformo in komponento, na oddaljeno lokacijo. Z njeno pomočjo je ostalim komponentam omogočena vključenost v platformo, čeprav se izvajajo izven nje, npr. na drugi infrastrukturi. Razvita je bila za potrebe razhroščevanja komponent v fazi razvoja. V tem primeru se le-te izvajajo lokalno pod nadzorom razhroščevalnika, a so hkrati prek komponente oddaljenega izvajanja deležne vse vhodno/izhodne komunikacije s platformo in so tako del nje.

3.5 Orodja za podporo pri razvoju aplikacij v mOSAIC-u

Razvoj aplikacij v oblaku za platformo mOSAIC je zaradi obsežnosti in potrebnega poznavanja tehnologij zelo zahteven proces. V izdatno pomoč pri tem opravilu je tako lahko velika množica orodij, ki so vanjo že vključena. Le-ta prek ustreznih uporabniških vmesnikov tako razvijalcem kot skrbnikom lajšajo vse faze življenjskega cikla aplikacije v oblaku - njeno načrtovanje, kodiranje, testiranje, razhroščevanje, postavitev in nadzor. Najbolj pogosto uporabljena orodja so opisana v nadaljevanju podpoglavja.

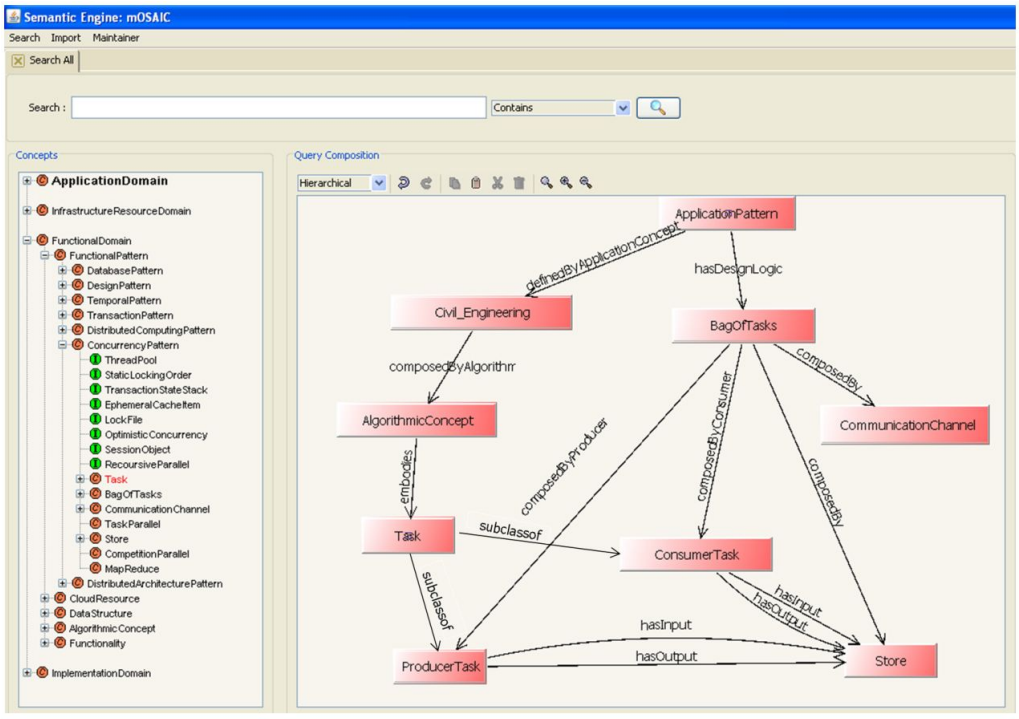
[8] http://jetty.codehaus.org/jetty/

3.5.1 Semantični pogon

Semantični pogon (angl. Semantic Engine) [8] je namenjen razvijalcu aplikacije v fazi načrtovanja. Danes na trgu obstaja velika pestrost različnih pristopov, storitev, orodij in tehnik računanja v oblaku. Za polno izkoriščenost vseh možnosti je potrebno odlično poznavanje specifičnih pristopov pri razvoju aplikacij v oblaku. Vse skupaj pomeni, da je razvoj za platformo za nekoga, ki se s tehnologijo šele spoznava, precej zahtevna naloga s strmo krivuljo učenja.

Orodje je torej eden izmed načinov, da se s platformo lažje spoznamo in jo začnemo hitreje in pravilneje uporabljati. V njem so na enem mestu zbrani in na semantični način zapisani pogledi na platformo mOSAIC, vsebuje pa tudi ostale splošne koncepte računalništva v oblaku. Med drugim ontologija, ki jo uporablja orodje, vsebuje celotno programsko paradigmo mOSAIC-a. V pripadajoči bazi znanja so shranjeni izseki izvorne kode vzorčnih aplikacij, ki demonstrirajo posamezne koncepte, tipični primeri uporabe, terminološki slovar, zbirka navodil, dokumentacija itd.

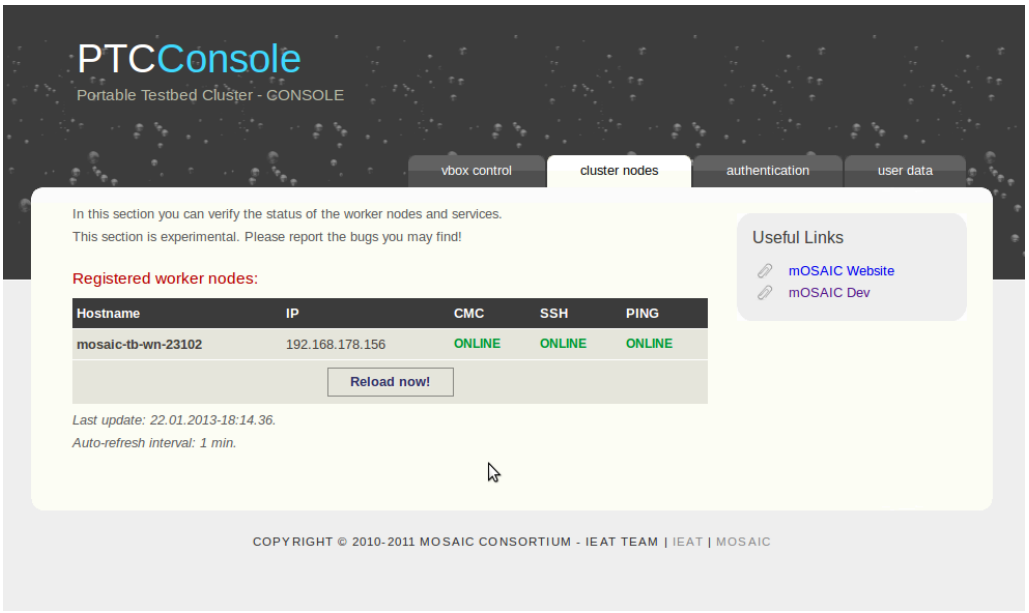
Prek hierarhičnega iskalnika lahko uporabnik preiskuje domeno računalništva v oblaku in se seznani z ustreznimi pojmi. Uporabnik lahko išče podobne implementacije tudi po ključnih besedah s področja svoje aplikacije.



Slika 3.3: Zaslonski posnetek Semantičnega pogona. Povzeto po [8].

Primer uporabe pogona je prikazan na sliki 3.3. Slika prikazuje semantični opis programskega modela vreče opravil (angl. bag-of-tasks). S slike so razvidne komponente in njihovi medsebojni odnosi, ki jih pričakujemo pri aplikaciji, ki temelji na tem programskem modelu.

3.5.2 Prenosna testna gruča



Slika 3.4: Vzpostavitev lokalne platforme mOSAIC z orodjem PTC.

S pomočjo komponente oddaljenega izvajanja je sicer razvoj posameznih komponent mogoče izvesti lokalno, a se za izvajanje celotne platforme še vedno potrebuje ustrezna infrastruktura. Čeprav je ta lahko v fazi razvoja precej okrnjena v smislu uporabljenih virov, lahko kljub temu predstavlja nepotreben strošek, če jo je treba zakupiti pri ponudniku. Prenosna testna gruča (angl. Portable Testbed Cluster, PTC) omogoča vzpostavitev pravega, a zmogljivostno okrnjenega okolja računalništva v oblaku na domačem računalniku. Z brezplačno programsko opremo VirtualBox[9] orodje vzpostavi eno ali več manj zmogljivih VM. Na njih nato namesti in zažene platformo mOSAIC, ki temelji na osnovi mOS, in jih poveže v gručo.

Orodje omogoča tudi vzpostavitev skladišča paketov Tazpkg in izdelavo predlog VM. V osnovi se orodje uporablja prek ukazne vrstice, večino osnovnih operacij pa se lahko opravi tudi prek vgrajenega spletnega vmesnika.

PTC lahko torej uporabimo, da na enostaven način hitro vzpostavimo manjši privatni

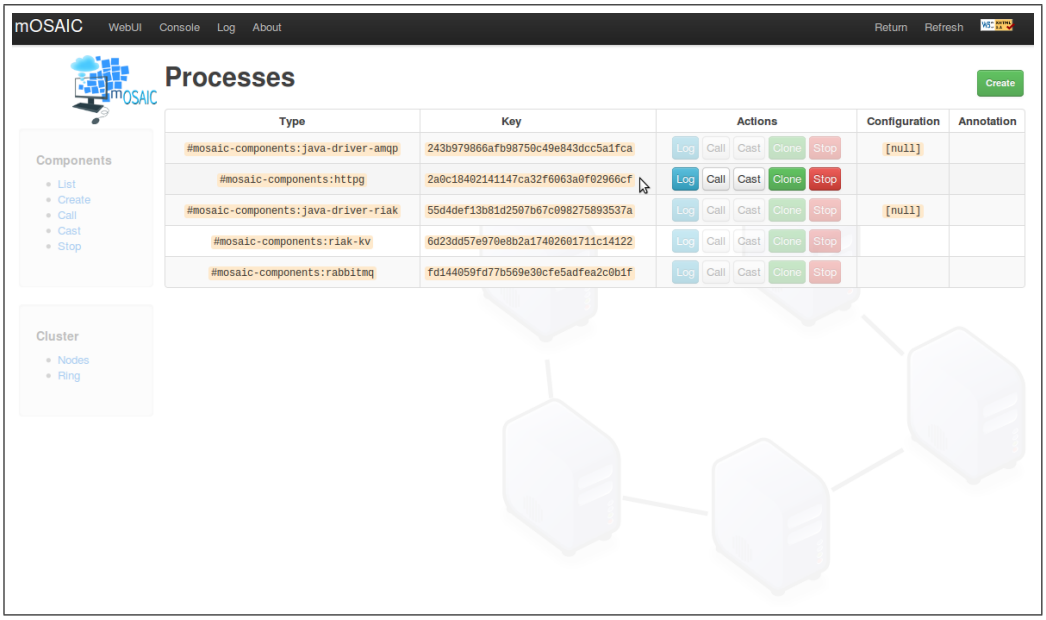
[9] http://www.virtualbox.org/

oblak z nameščeno platformo mOSAIC. Skupaj s komponento oddaljenega izvajanja tako tvori prilagodljivo okolje, ki omogoča razhroščevanje in testiranje aplikacij na lastnem delovnem računalniku, kar je nepogrešljivo za vse razvijalce, ki uporabljajo platformo mOSAIC.

Vzpostavitev lokalne testne gruče, ki temelji na eni VM, z uporabo orodja PTC preko spletnega vmesnika je prikazana na sliki 3.4.

3.5.3 Spletni upravljavec platforme mOSAIC

Upravljanje platforme mOSAIC prek spletne storitve zahteva poznavanje ukazne sintakse in več tehnologij (REST, JSON), torej je ni enostavno izvajati ročno. Platforma zato vsebuje spletni vmesnik, ki omogoča izvedbo najpogostejših opravil s preprostim klikom v brskalniku in tako omogoča preprostejše upravljanje s platformo. Podpira pregled komponent in njihovih konfiguracij, zagon in zaustavitev komponent, pregled njihovih dnevnikov ter pošiljanje sporočil vsem ali le posameznim komponentam. Omogoča tudi nadzor celotne gruče, npr. kateri VM so vključeni, dodajanje/odstranjevanje novih VM itd. Sledimo lahko tudi dnevniku posamezne VM ali pa celotne gruče. Zaslonski posnetek spletnega vmesnika je prikazan na sliki 3.5 in kaže pet zagnanih komponent na celotni gruči ter akcije, ki jih lahko izvedemo nad njimi.



Slika 3.5: Spletni vmesnik upravljavca platforme mOSAIC.

3.5.4 Urejevalnik opisnika aplikacije

Celotna aplikacija v oblaku za platformo mOSAIC je sestavljena iz množice komponent. Za njeno pravilno izvajanje morajo biti zagnane vse komponente, zaradi medsebojne odvisnosti pa je pomemben tudi vrstni red zagona posamezne komponente. Npr. komponenta prehoda HTTP mora biti zagnana za komponento sporočilnih vrst, saj prva za delovanje potrebuje storitev, ki jo nudi druga. Prek vgrajene spletne storitve ali prek spletnega upravljavca za nadzor platforme sicer lahko zaganjamo vsako komponento posebej v pravilnem vrstnem redu, a prek spletne storitve lahko posredujemo tudi t. i. opisnik aplikacije (angl. Application Descriptor), ki poskrbi, da se v pravilnem vrstnem redu zaženejo vse komponente aplikacije. Opisnik aplikacije je tekstovna datoteka v formatu JSON, ki vsebuje seznam vseh komponent, njihove konfiguracije, vrstni red zagona ter morebitno zakasnitev med zagonom posameznih komponent, da imajo komponente dovolj časa za inicializacijo. Primer opisnika je prikazan na sliki 3.6.

Ročno pisanje datoteke je zaradi enostavnosti formata mogoče, a podvrženo napakam.

```
{
    "rabbitmq": {
        "type": "#mosaic-components:rabbitmq",
        "configuration": null,
        "annotation": null,
        "count": 1,
        "order": 1,
        "delay": 3000
    },
    "rabbitmq driver": {
        "type": "#mosaic-components:java-driver-amqp",
        "configuration": [
            null
        ],
        "annotation": null,
        "count": 1,
        "order": 2,
        "delay": 5000
    },
    "httpg": {
        "type": "#mosaic-components:httpg",
        "configuration": null,
        "annotation": null,
        "count": 1,
        "order": 3,
        "delay": 4000
    }
}
```

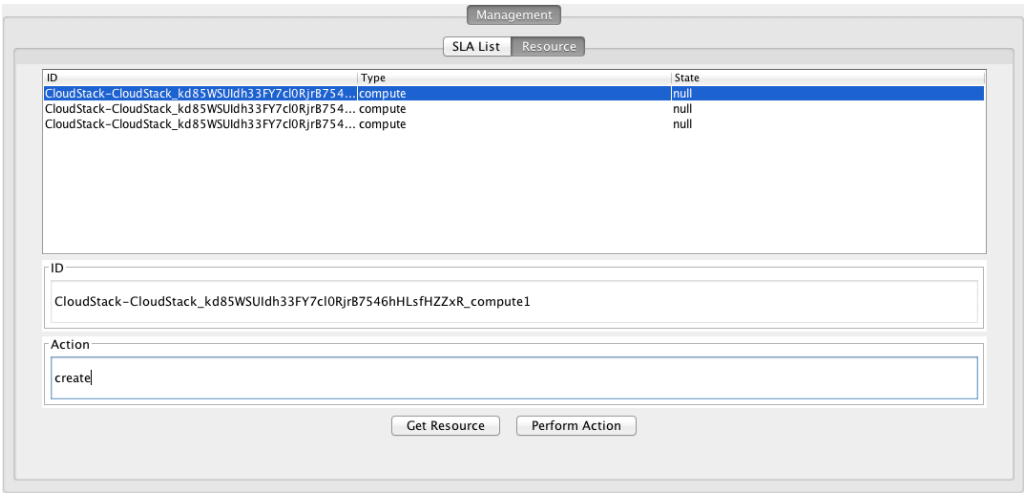
Slika 3.6: Primer opisnika aplikacije, ki zažene tri komponente: komponento sporočilnih vrst, komponento gonilnika za sporočilne vrste in komponento prehoda HTTP.

Zato je bil za okolje Eclipse[10] izdelan grafični urejevalnik opisnika aplikacije, ki omogoča udobno izdelavo pravilnega opisnika.

3.5.5 Agencija za oblake

Agencija za oblake je orodje, ki je namenjeno pridobivanju virov za potrebe delovanja platforme mOSAIC. Uporablja se lahko samostojno, hkrati pa je vgrajeno v platformo, kjer je na voljo aplikacijam. Deluje tako, da najprej uporabnik sestavi t. i. poziv za sestavo predlogov (angl. Call-For-Proposal, CPF), ki opisuje želene vire. Agencija pridobljeni CFP prek gonilnikov na nivoju IaaS (slika 3.1) pošlje ponudnikom, za katere ima ustrezne poverilnice, tj. uporabniško ime, geslo itd. Agencija nato na podlagi prispelih ponudb izbere najugodnejšo in z izbranim ponudnikom sklene dogovor o nivoju storitev (angl. service-level-agreement, SLA). Pridobljena sredstva tako postanejo na voljo uporabniku oz. aplikaciji, prek agencije pa jih lahko nadzoruje (npr. sprosti, ko jih ne potrebuje več).

Grafični vmesnik Agencije za oblake v samostojni obliki na sliki 3.7 prikazuje pridobitev treh VM pri ponudniku CloudStack.



Slika 3.7: Grafični vmesnik Agencije za oblake in prikaz pridobljenih virov.

[10] http://www.eclipse.org/

4 Programsko ogrodje za prenos v oblak

Glavni cilj našega dela je izdelava programskega ogrodja, s katerim lahko uspešno prenesemo zahtevne inženirske aplikacije Matlab v oblak, temelječ na platformi mOSAIC. Poudarka pri razvoju ogrodja sta bila enostavnost in splošnost, tako da bi bilo ogrodje lahko tudi nadalje uporabno, predvsem za inženirje, ki bi želeli prenesti svoje aplikacije v oblak z minimalnim trudom.

V nadaljevanju so v podpoglavjih predstavljene zahteve, ki jim mora zadostiti ogrodje (4.1) in njegova arhitektura (4.2), opisan je postopek implementacije Cloudleta, ki je potreben za uporabo ogrodja (4.3). Sledi opis formata datotek, ki jih končni uporabnik uporablja za komunikacijo s preneseno aplikacijo (4.4), naštete so vse omejitve, ki jih moramo odstraniti iz obstoječe aplikacije, če jo želimo prenesti v oblak (4.5). Na koncu pa je povzet celoten postopek uporabe programskega ogrodja (4.6).

4.1 Načrtovanje ogrodja

Pri načrtovanju ogrodja in tudi pri sami izdelavi smo si najprej postavili splošne omejitve in zahteve, ki jih od ogrodja pričakujemo. Izpeljane so iz osnovne želje - iz splošne obstoječe aplikacije Matlab s čim manj napora (v smislu kodiranja in spreminjanja obstoječe izvorne kode) zgraditi spletno storitev v oblaku, ki širši javnosti nudi funkcionalnost osnovne aplikacije. Bolj formalno lahko omejitve in zahteve strnemo v naslednjem seznamu:

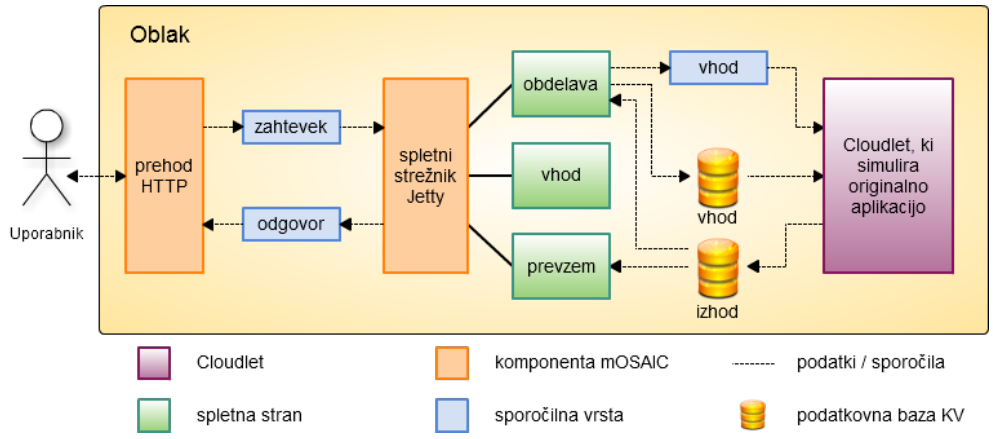
- aplikacija mora biti dostopna prek interneta v obliki programske opreme kot storitve (SaaS),
- ob istem vhodu mora biti izhod prenesene aplikacije enak izhodu začetne aplikacije,
- izvajanje aplikacije mora biti mogoče brez uporabe katerekoli plačljive licence Matlab,
- aplikacija mora biti skalabilna - v primeru velikega števila sočasnih uporabnikov mora znati izkoristiti dane vire,
- aplikacija mora biti elastična - uporaba razpoložljivih virov se dinamično prilagaja (viri se pridobijo ali sprostijo) potrebam aplikacije,

- uporaba ogrodja mora biti preprosta in vloženo delo minimalno (v smislu posegov v obstoječo aplikacijo) in
- ogrodje mora biti primerno za splošne aplikacije Matlab (z manjšimi omejitvami).

Dodatne koristi, ki jih lahko pričakujemo v oblačnem okolju z veliko razpoložljivimi viri, npr. hitrejše računanje s pomočjo vzporednega izvajanja, optimizacija glede na zmogljivost posameznih virov itd., niso bile upoštevane pri načrtovanju ogrodja. Omenjene koristi je sicer pri nekaterih aplikacijah mogoče doseči, vendar postopek zahteva poglobljeno poznavanje delovanja aplikacije in običajno obsega velike posege v izvorno kodo, npr. uporaba vzporednih algoritmov, porazdeljeno računanje, vzpostavitev komunikacije med posameznimi deli aplikacije v oblaku itd. Način, na katerega dosežemo te koristi na konkretni aplikaciji, je opisan v poglavju 6.

4.2 Arhitektura ogrodja

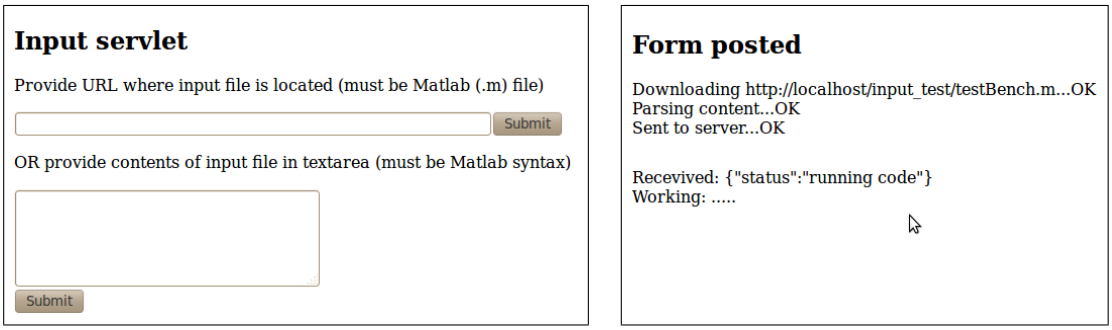
Iz podanih splošnih omejitev in zahtev smo zgradili modularno programsko ogrodje za platformo mOSAIC, katerega arhitektura je prikazana na sliki 4.1. Ogrodje je sestavljeno iz standardnih komponent, ki so v platformo že vgrajene, spletnih strani in pa posebnega Cloudleta, ki simulira delovanje osnovne aplikacije Matlab. Čeprav je prikazana arhitektura že polno delujoča aplikacija, lahko govorimo o programskem ogrodju, saj je omenjeni Cloudlet edina komponenta, ki se spremeni v primeru spremembe osnovne aplikacije.



Slika 4.1: Arhitektura programskega ogrodja za prenos aplikacij.

Iz arhitekture je razvidno, da končni uporabnik do aplikacije dostopa prek enostavnih spletnih strani, saj tak način zahteva le uporabo spletnega brskalnika. Osnovna naloga spletnih strani je posredovanje podatkov med uporabnikom in aplikacijo. Ker morajo biti spletne

strani splošne oz. primerne za vse aplikacije, smo se odločili, da uporabnik vse vhodne podatke posreduje v obliki ene datoteke. Enak princip velja za izhod - rezultat aplikacije je ena datoteka, ki vsebuje definicije vseh izhodnih podatkov. Uporabniški vmesnik tako sestavljajo tri spletne strani: prva je namenjena oddaji vhodne datoteke, druga je namenjena prikazu poteka izračuna, tretja pa prevzemu izhodne datoteke po končanem izračunu. Zaslonski posnetek spletne strani `vhod` in `obdelava` je prikazan na sliki 4.2.



(a) (b)

Slika 4.2: Zaslonski posnetek spletnih strani programskega ogrodja: (a) “vhod” in (b) “obdelava”.

Natančneje lahko delovanje z ogrodjem prenesene aplikacije opišemo na naslednji način. Uporabnik aplikacije najprej obišče URL vhodne spletne strani. Zahtevek HTTP prevzame vgrajena komponenta prehoda HTTP (HTTPg), ki ga posreduje naprej komponenti strežnika spletnih strani (Jetty). Ta zgradi vsebino vhodne spletne strani (`vhod`), ki omogoča oddajo datoteke ali pa posredovanje vsebine le-te in jo prek komponente HTTPg posreduje nazaj do uporabnika. Ko uporabnik odda primerno datoteko, se prek istega mehanizma sproži naslednja spletna stran (`obdelava`). Ta datoteko pregleda, preveri njeno veljavnost, jo zapiše v podatkovno bazo `vhod` (za storitev skrbi vgrajena komponenta shranjevanja ključ-vrednost) pod naključno generiranim ključem in pošlje sporočilo, ki ta ključ vsebuje, v ustrezno sporočilno vrsto (za storitev skrbi vgrajena komponenta sporočilnih vrst). Tam sporočilo prevzame Cloudlet, ki iz podatkovne baze `vhod`, na podlagi ključa v sporočilu, prevzame vhodne podatke in z njimi simulira osnovno aplikacijo. Ob uspešni izvedbi se izhodni podatki zapišejo v podatkovno bazo `izhod`. Spletna stran Obdelava na podlagi mehanizma AJAX (angl. Asynchronous JavaScript and XML) zapis detektira in uporabniku prikaže zmožnost prevzema datoteke. Če se uporabnik za to možnost odloči, se zažene zadnja spletna stran (`prevzem`), ki podatke najprej prevzame iz podatkovne baze `izhod` ter jih nato v obliki izhodne datoteke posreduje uporabniku.

Predstavljena arhitektura reši precej zahtev, ki smo jih našteli na začetku. Z ogrodjem prenesena aplikacija je dostopna prek interneta v obliki SaaS in je hkrati elastična. Slednja

lastnost izhaja iz dejstva, da se glavnina izvajanja zgodi v Cloudletu, saj ostale komponente sodelujejo le pri prenosu podatkov med uporabnikom in Cloudletom. Elastičnost Cloudleta pa samodejno zagotavlja platforma mOSAIC. Aplikacija je tudi skalabilna v primeru velikega števila sočasnih uporabnikov, saj je zaradi neodvisnosti posameznih izračunov (vsak uporabnik posreduje svoj vhod) in uporabe sporočilnih vrst za razporejanje obdelav mogoče zagnati več primerkov Cloudleta, če so razpoložljivi viri na voljo (pri oblačnem računalništvu v splošnem so). Na ta način je mogoče streči veliki množici sočasnih uporabnikov, in je torej celotna aplikacija skalabilna, saj se Cloudleti izvajajo neodvisno drug od drugega, stične točke (sporočilne vrste in podatkovna baza) pa ne predstavljajo velikih stroškov režije (angl. overhead), saj so implementirane na skalabilen način [2].

4.3 Gradnja Cloudleta, primernega za ogrodje

Uporaba ogrodja in zadostitev preostalim zahtevam je torej omejena na zmožnost implementacije ustreznega Cloudleta, ki simulira delovanje osnovne aplikacije. V nadaljevanju je predstavljen način, ki zagotovi ohranitev pravilnosti delovanja aplikacije ter se hkrati drži tudi ostalih zahtev.

Največjo težavo pri implementaciji predstavlja natančno posnemanje izvajanja Matlaba. Prepis v programski jezik Cloudleta (izbrali smo jezik Java) je nepraktičen in zelo zahteven proces, saj Matlab pri določenih funkcijah notranje uporablja zahtevne, numerično nestabilne algoritme, ki navsezadnje niso javno objavljeni. Boljši način bi bila uporaba katere od odprtokodnih rešitev za numerično računanje, ki razume sintakso programskega jezika Matlab in ga zna vključiti tudi v druge jezike (Scilab, Octave). Žal tudi s temi rešitvami ne moremo zagotoviti popolne pravilnosti delovanja, saj se tukaj pojavi enak problem glede numerične stabilnosti. Izpolnitev zahteve po pravilnosti delovanja za splošne aplikacije je tako mogoča le z uporabo pogona Matlab. Edini brezplačni pogon, primeren za končnega uporabnika, je na voljo z uporabo prevajalnika Matlab. Le-ta zagotavlja, da se prevedene aplikacije izvajajo popolnoma enako kot v osnovnem okolju Matlab. Pri uporabi prevajalnika je na voljo več izhodnih oblik za različna programska okolja. Za naše potrebe smo uporabili dodatek, imenovan Builder JA (skupno ime MCJ), kjer izhod prevajanja dobimo v obliki arhiva JAR, in je torej izhod lahko neposredno uporaben v Cloudletu, ki je implementiran v programskem jeziku Java.

Glede na to, da pridobljeni JAR popolnoma opravi delo osnovne aplikacije, mora delovanje preostalega dela Cloudleta poskrbeti le še za ustrezno komunikacijo z okoljem (sprejemanje in obdelava sporočil, prevzem in zapis podatkov v podatkovno bazo) in posredovanje izračuna

v obdelavo pridobljenemu arhivu JAR. Razvidno je, da je ta del delovanja neodvisen od osnovne aplikacije in celotni delovni tok (angl. workflow) Cloudleta je tako vnaprej natančno znan. Na tej podlagi smo lahko pripravili predlogo (angl. template)[1], ki vsebuje vso potrebno izvorno kodo za enostavno izdelavo Cloudleta, ki posnema delovanje osnovne aplikacije.

4.4 Format izmenjave podatkov

V podpoglavju 4.2 smo omenili, da uporabnik vse vhodne podatke posreduje v obliki ene datoteke, enako pa velja tudi v nasprotni smeri - za pridobitev izhodnih podatkov. Osnovna struktura, s katero operira Matlab, je matrika in seznam matrik se uporablja kot vhod ali izhod v program. Matrike so lahko eno-, dvo- in tudi večdimenzionalne ter lahko vsebujejo različne tipe podatkov (cela, realna ali kompleksna števila, črke itd.). Obstaja več načinov zapisov takih matrik v datoteke, v našem delu pa smo za format zapisa izbrali kar programski jezik Matlab. S tem smo dobili tudi možnost, da pri preneseni aplikaciji v oblaku uporabljamo enake vhode (npr. testne primere) kot pri osnovni aplikaciji. Enako velja tudi za izhod - brez sprememb ga je mogoče uporabiti v morebitnih že obstoječih postopkih naknadne obdelave (angl. post-processing), kot npr. pri vizualizaciji podatkov.

Uporaba sintakse Matlab za prenos podatkov je zahteven problem. Ob klicu ustrezne metode, pridobljene z MCJ, mora namreč Cloudlet vhodne podatke posredovati v točno določeni obliki - kot seznam primerkov razreda MWArray. Razred MWArray je osnovni javanski razred za hranjenje matrike (za različne tipe vrednosti matrik se uporablja različne izpeljane razrede, kot so npr. MWNumericArray za številčni tip, MWCharArray za nize itd.). Najpreprosteje bi preslikavo iz ene v drugo obliko naredili tako, da bi vsebino celotne vhodne datoteke predstavili kot en primerek razreda MWCharArray, ki bi se nato notranje ovrednotila s pomočjo funkcije `eval`. Žal ta rešitev odpade, saj funkcija `eval` ni podprta v prevajalniku MCJ (z njo bi bilo enostavno zgraditi okolje, ki bi izvajalo poljubno kodo brez plačljivih oblik Matlaba). Posledično smo zato implementirali preprost prevajalnik za programski jezik Matlab, imenovan PPMM (Preprost prevajalnik za Matlabove matrike), ki omogoča preslikavo vhodne datoteke v ustrezen seznam primerkov razreda MWArray. Gre za prevajalnik, ki razume le majhno podmnožico celotnega jezika Matlab, saj je potrebno le razumevanje definiranja matrik in nekaterih osnovnih operacij nad njimi. Več o prevajalniku PPMM je na voljo v dodatku B.

Nasprotni, a precej lažji postopek je uporabljen za zapis izhodnega seznama primerkov razreda MWArray v izhodno datoteko.

[1]Predloga je prosto dostopna na https://bitbucket.org/mosaic/mosaic-applications-matlab.

4.5 Predelava osnovne aplikacije

Na tem mestu so podane potrebne predelave osnovne aplikacije, ki izhajajo iz uporabe prevajalnika MCJ, uporabe v Cloudletu in izvajanja v oblaku.

Za začetek mora aplikacija vsebovati le tiste vgrajene funkcije Matlaba, ki jih prevajalnik MCJ podpira (seznam je na voljo v [28]). Na srečo je seznam precej kratek, a večji problem predstavlja uporaba orodjarn - podprte so le nekatere, pa še te praviloma ne v celoti. Prvi korak prilagoditve je torej odstranitev nepodprtih funkcij.

Nadalje je potrebno aplikacijo prilagoditi tako, da bo zadostila ostalim omejitvam, ki izhajajo iz arhitekture ogrodja in zahtev platforme mOSAIC. Glede na delovanje ogrodja, ki predvideva način "en zahtevek, eno izvajanje", je naslednja prilagoditev ta, da moramo celotno aplikacijo preoblikovati v eno samo funkcijo. S tem pridobimo eno samo vhodno točko (angl. entry point), kjer imamo podane vse vhodne in izhodne parametre. Hkrati se delovanje te funkcije ne sme zanašati na globalne spremenljivke (vse morajo biti podane kot vhod funkcije). Iz dejstva, da se aplikacija izvaja v oblaku, izhaja tudi, da njeno delovanje ne sme biti odvisno od lokalnega okolja, pač pa se morajo vsi podatki, ki jih aplikacija potrebuje, nahajati ali biti dosegljivi v oblaku. Npr. imena datoteke ne smemo podati kot pot do lokalne datoteke, pač pa kvečjemu kot javno dostopen internetni naslov URL (angl. uniform resource locator).

Naslednje prilagoditve izhajajo iz dejstva, da bo prevedena funkcija vključena v Cloudlet in se mora tako držati pravil, ki veljajo za njih. To pomeni, da se v primeru več zagnanih primerkov prevedene funkcije ne smejo medsebojno motiti oz. morajo biti odvisne druga od druge. Obenem velja tudi prepoved uporabe standardnega vhoda in izhoda, saj ju platforma mOSAIC uporablja za pravilno delovanje in nadzor nad Cloudleti. Enako velja tudi za uporabo grafičnega okolja (npr. prikaz slik), saj ta v oblačnem okolju ni na voljo. Glede na to, da funkcija ne sme uporabljati standardnega in grafičnega izhoda ter da ne sme zapisovati lokalnih datotek, mora vse podatke računanja vrniti kot izhodne podatke funkcije.

Zadnje prilagoditve je potrebno izvesti zaradi omejitev prevajalnika PPMM. Funkcija tako lahko vhodne podatke sprejema le v obliki 2D-matrik (vektorjev, skalarjev) realnih števil ali nizov, saj so to edini podprti tipi. Če torej aplikacija na vhodu pričakuje kakšno od nepodprtih oblik, jo je potrebno ustrezno prepisati (npr. namesto vhodne matrike kompleksnih števil lahko funkcijo prilagodimo, da na vhodu pričakuje dve matriki realnih števil - prva vsebuje realne, druga pa imaginarne dele kompleksnih števil).

Če povzamemo, mora aplikacija, da bi bila primerna za uporabo v ogrodju, upoštevati naslednje omejitve:

- uporablja le funkcije in orodjarne, ki jih podpira prevajalnik MCJ,
- preoblikovana je v eno samo funkcijo (lahko vsebuje podfunkcije),
- vse podatke za delovanje pridobi kot vhod v funkcijo, npr. ne uporablja globalnih spremenljivk, lokalnega datotečnega sistema itd.,
- vse podatke vrne kot izhod, npr. ne zapisuje v lokalni datotečni sistem, ne nastavlja globalnih spremenljivk itd.,
- ne uporablja standardnega vhoda in izhoda,
- ne uporablja grafičnega okolja, npr. prikaz slik,
- na vhodu pričakuje 2D-matrike (vektorje, skalarje) realnih števil ali nizov.

4.6 Uporaba ogrodja

Na tem mestu je po korakih opisana izdelava lastne oblačne aplikacije na osnovi obstoječe aplikacije Matlab z uporabo predstavljenega ogrodja.

Večine delov ogrodja ni potrebno v ničemer spreminjati, saj ga sestavljajo v platformo vgrajene komponente (prehod HTTP, strežnik spletnih strani, sporočilne vrste, podatkovna baza). Spletne strani, ki jih ogrodje uporablja za uporabniški vmesnik, so splošne in jih za pravilno delovanje ni potrebno spreminjati. Priporočljivo pa je, da se popravijo v smislu prikazane vsebine in dokumentacije, kako aplikacija deluje in kakšne vhodne podatke pričakuje. Večjo pozornost mora uporabnik ogrodja nameniti le implementaciji ustreznega Cloudleta. S predstavljeno predlogo lahko postopek izvedemo z naslednjimi koraki:

1. Prilagoditev osnovne aplikacije.
2. Izdelava ustreznega arhiva JAR s prevajalnikom MCJ.
3. Vključitev arhiva v predlogo.
4. Prevajanje in testiranje Cloudleta.

Pri prvem koraku je potrebno osnovno aplikacijo prilagoditi na način, da ustreza omejitvam, podanim v 4.5. Ta korak je najtežji in je v veliki meri odvisen od uporabljene aplikacije.

Drugi korak zahteva uporabo prevajalnika MCJ. Postopek je enostaven in izvedljiv prek vgrajenega grafičnega vmesnika, saj je potrebno le določiti, katero funkcijo želimo prevesti in kako jo poimenovati.

V tretjem koraku je potrebno pri vključevanju nastalega arhiva JAR v pripravljeno predlogo dopolniti tri stvari:

1. popraviti ime metode in javanskega razreda znotraj arhiva JAR, ki opravi izračun,
2. določiti pravilen vrstni red vhodnih matrik v seznamu, ki je podan kot vhod klica metode, in
3. definirati imena izhodnih matrik, ki jih metoda vrne.

Slednja popravka sta posledica tega, da Cloudlet ob klicu ustrezne metode v arhivu JAR vhodne/izhodne podatke posreduje/pridobi v obliki seznama. V seznamih je pomemben vrstni red matrik, medtem ko je v programskem jeziku Matlab pomembno ime matrike, zato je potrebno izvesti preslikavo.

V zadnjem koraku je najprej treba prevesti spremenjeno predlogo Cloudleta v arhiv JAR, kar je v okolju Eclipse enostavno, saj predloga vsebuje tudi potrebne napotke za enostavno prevajanje. Na koncu je seveda potrebno celotno aplikacijo namestiti v ustrezno okolje mOSAIC, jo zagnati in preveriti pravilnost njenega delovanja. Dodatno oviro predstavlja dejstvo, da tako pridobljena aplikacija pri izvajanju potrebuje izvajalno okolje MCR. To pomeni, da je treba na vsako VM, kjer se bo aplikacija izvajala, namestiti omenjeno okolje. Za lažji postopek smo pripravili paket Tazpkg, primeren za uporabo znotraj platforme mOSAIC, ki zna MCR za prevajalnik MCJ različice 2012a namestiti samodejno.

5 Poskusi in rezultati

V prejšnjem poglavju smo podrobneje opisali zgradbo in implementacijo programskega ogrodja za prenos aplikacij na platformo mOSAIC, v tem poglavju pa želimo oceniti primernost ogrodja na podlagi opravljenih testiranj.

Najprej smo vzeli dve različni obstoječi aplikaciji in ju z uporabo ogrodja prenesli na platformo mOSAIC. Opis obeh aplikacij in postopek njunega prenosa je opisan v podpoglavju 5.1. Uspešnost prenosa smo merili glede na pravilnost in hitrost izvajanja prenesene aplikacije na več različnih konfiguracijah, ki so predstavljene v podpoglavju 5.2. Opravili smo štiri sklope testiranj. S prvim (5.3.1) smo merili razliko v hitrosti izvajanja izračunov med obstoječo in preneseno aplikacijo, z drugim (5.3.2) ocenili vpliv režije ogrodja na celoten čas izvajanja aplikacije, v tretjem sklopu (5.3.3) smo testirali še skalabilnost prenesene aplikacije, v zadnjem (5.3.4) pa smo si ogledali, ali se ohranijo numerične lastnosti prenesene aplikacije. Na koncu poglavja (5.4) je predstavljen še povzetek vseh testiranj.

5.1 Aplikaciji

5.1.1 Generična aplikacija

Prva aplikacija, s katero smo testirali ogrodje, je generična aplikacija, ki izvede sintetične teste, namenjene preverjanju zmogljivosti različnih sklopov Matlaba. Aplikacija je povzeta po [7] in izvede 15 testov, ki jih lahko razdelimo v tri sklope po pet testov: prvi sklop obsega delo z matrikami, drugi izvaja funkcije nad matrikami, tretji pa testira hitrost izvajanja programske kode. Testi so podrobneje predstavljeni v tabeli 5.1. Vhod v aplikacijo je samo en parameter in definira število ponovitev posameznega testa. Izhod aplikacije je izpis povprečnih časov izvajanja posameznega testa na standardni izhod in pa tudi matrika, ki vsebuje enake podatke. Za prilagoditev te aplikacije smo se odločili, ker zaobjame večino funkcionalnosti Matlaba, a hkrati za delovanje ne potrebuje grafičnega izhoda (to je glavni razlog, da sintetičnih testov nismo opravili z vgrajeno funkcijo `bench`).

Celotno aplikacijo smo nato preoblikovali v eno funkcijo, ki ustreza pogojem opisanim v 4.5. Pri podani aplikaciji je postopek kratek in enostaven. Glede na to, da je izvajanje odvi-

Test	Operacija
1	gradnja/preoblikovanje matrik
2	potenciranje
3	sortiranje
4	vektorski produkt
5	linearna regresija
6	hitra Fourierjeva transformacija
7	lastni vektorji
8	determinanta
9	razcep Choleskega
10	obratna matrika
11	Fibonaccijeva števila
12	Hilbertova matrika
13	največji skupni delitelj
14	Toeplitzova matrika
15	RV koeficient

Tabela 5.1: Seznam testov generične aplikacije.

sno le od enega samega vhodnega parametra (skalarna vrednost), podobno pa velja tudi za rezultat (vektor časov), dela s prilagoditvijo vhodno/izhodnih parametrov ni bilo. Aplikacija hkrati ne vsebuje nobene funkcije, ki je prevajalnik MCJ ne bi podpiral. Največ dela pri samem prilagajanju je bilo z odstranjevanjem sprotnih izpisov, saj funkcija ne sme pisati na standardni izhod.

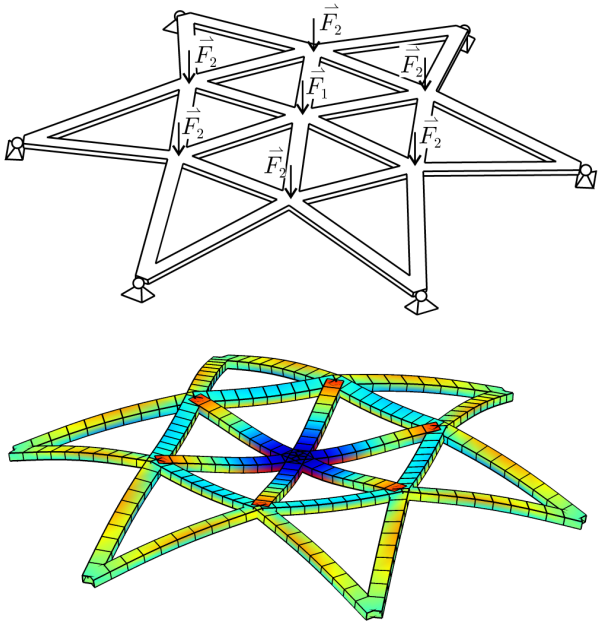
Pri izdelavi ustreznega Cloudleta, primernega za naše ogrodje, smo uporabili postopek, opisan v 4.6. Izdelano funkcijo smo s pomočjo MCJ prevedli v ustrezen arhiv JAR. Nato smo izdelano predlogo za Cloudlet dopolnili z ustreznimi podatki: imeni vhodnih in izhodnih parametrov. Poskrbeli smo tudi za ustrezno vključenost zgrajenega arhiva JAR v gradnjo ciljnega razreda ter pravilne klice ustreznih metod prevedene funkcije.

V vseh testih smo za aplikacijo uporabili enak vhodni podatek - število ponovitev posameznega testa za ustrezen izračun povprečja smo nastavili na pet.

5.1.2 Aplikacija NoDeK

Naslednja aplikacija, ki smo jo uporabili za testiranje ogrodja, je zahtevna inženirska aplikacija z imenom NoDeK [9]. Razvili so jo na katedri za mehaniko na Fakulteti za gradbeništvo in geodezijo Univerze v Ljubljani in je namenjena analizi statičnih obremenitev. Aplikacija temelji na metodi končnih elementov (angl. finite element method) [44], kjer se učinkovanje sil na celotno strukturo numerično modelira kot učinkovanje teh sil na posamezne dele strukture

- majhne prostorske nosilce, imenovane končni elementi (KE). Prednost aplikacije NoDeK je uporaba geometrijsko točne teorije prostorskih nosilcev s konstantnimi deformacijami (angl. geometrically exact spatial beam with constant strains), ki omogočajo analitični izračun njihovega odziva na sile in se s tem izognejo numeričnim napakam, prisotnim pri uporabi običajnih prostorskih nosilcev, ki deformacije računajo z numeričnimi približki. Dodatno, z dovolj gosto mrežo KE, metoda omogoča izračun ekstremnih deformacij v smislu zvijanja in premikov, kar omogoča natančno analizo kompleksnih struktur (npr. vzmeti). Primer vhodnega problema in rezultata izračuna aplikacije je prikazan na sliki 5.1.



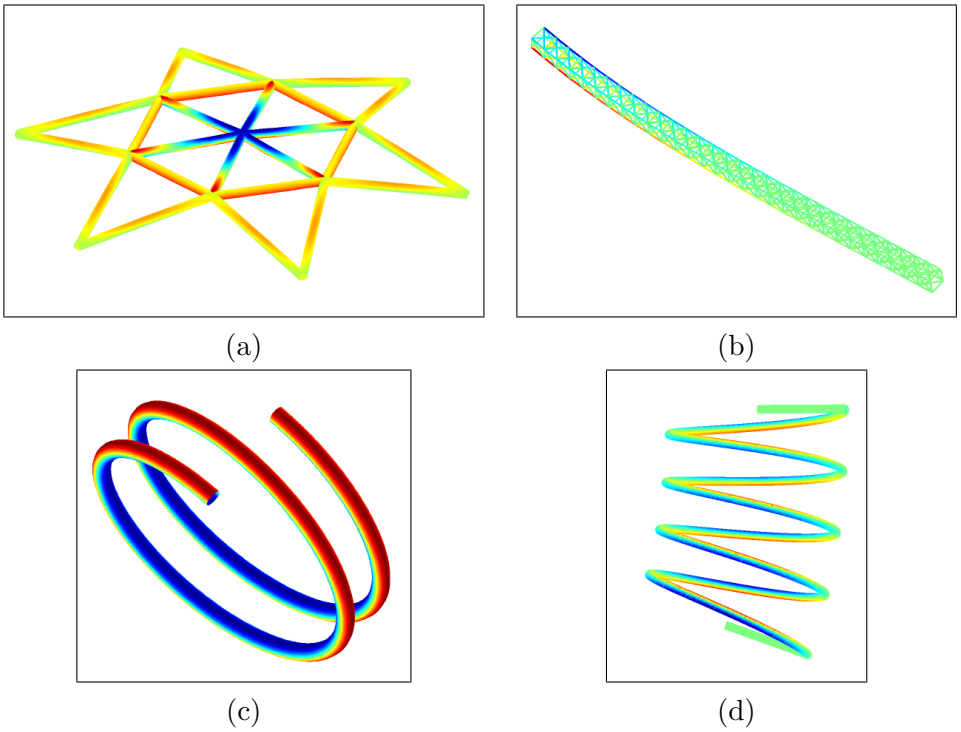
Slika 5.1: Prikaz strukture pred obremenitvijo in načrtovanimi silami (zgoraj) in deformacije po obremenitvi (spodaj), kot jo izračuna aplikacija NoDeK.

Aplikacija je bila razvita v okolju Matlab kot prototip za potrditev koncepta (angl. proof-of-concept) uporabe omenjenih naprednih prostorskih nosilcev. Je brez grafičnega vmesnika in se uporablja le za interne potrebe oddelka. Spodbudni rezultati, v smislu izboljšane analize struktur, sestavljenih iz do nekaj tisoč KE, v primerjavi z običajnimi pristopi, so pripeljali do ideje, da se aplikacija lahko ponudi širši javnosti kot spletna storitev. Ena izmed možnosti za dosego tega cilja je prenos aplikacije v oblak s pomočjo predstavljenega ogrodja.

Pri pretvorbi celotne aplikacije v eno funkcijo je bilo tokrat precej več dela, saj je bilo glavno vodilo pri razvoju pravilnost delovanja in natančnost izračunov, ne pa hitrost ali strukturiranost izvorne kode. Tako je bilo v okviru postopka preoblikovanja v eno funkcijo potrebno odpraviti odvisnost od globalnih spremenljivk, izbrisati testne izpise in onemogočiti

gradnjo dnevniške datoteke. Največ dela je bilo z združevanjem različnih vhodov in izhodov na eno mesto. K sreči aplikacija ne uporablja funkcij, ki jih prevajalnik Matlab ne podpira, in za delovanje ne potrebuje grafičnega izhoda (rezultate aplikacije je priporočljivo vizualizirati, a se za to uporablja ločena aplikacija). Končna funkcija pri delovanju potrebuje precej vhodnih podatkov (podatki o strukturi, geometriji, materialu, silah, obtežbah, sprostitvah itd.), ki so zelo obsežni in zahtevajo ekspertno poznavanje domene analize struktur. Podobno velja tudi za rezultat funkcije (obsega podatke o premikih, zasukih itd. za vsak KE). Vsi vhodni in izhodni podatki so standardne eno- ali dvodimenzionalne matrike, tako da smo lahko brez problema uporabili razviti prevajalnik PPMM.

Izdelave ustreznega Cloudleta je potekala enako kot pri generični aplikaciji. Izdelano funkcijo smo s pomočjo MCJ prevedli v ustrezen arhiv JAR, izdelano predlogo dopolnili z imeni vhodnih in izhodnih parametrov in na koncu poskrbeli za ustrezno vključenost arhiva JAR v gradnjo ciljnega razreda ter pravilne klice ustreznih metod.



Slika 5.2: Vizualizacija testnih primerov za aplikacijo NoDeK.
(a) Dome (b) Arm (c) Bent (d) Helix

Za potrebe testiranj aplikacije smo pripravili štiri testne vhode, ki simulirajo realne scenarije in naraščajo po zahtevnosti izračuna. Najosnovnejši je "Dome", ki analizira vpliv točkovnih sil na kupolo (slika 5.1). Naslednji je "Arm", ki izračunava vpliv prečne sile na vesoljsko robotsko roko. Tretji vhod "Bent" simulira vpliv potisne sile na upogljiv nosilec,

pritrjen na eni strani, kar povzroči njegovo zvijanje. Zadnji primer "Helix" izračunava deformacijo obešene vzmeti pod vplivom stranske sile. Osnovni podatki o vhodih so podani v tabeli 5.2, slike vhodnih podatkov in njihovih izračunov pa na sliki 5.2.

Vhod	Število KE	Št. iteracij	Št. izračunov
Dome	240	45	10800
Arm	968	40	38720
Bent	1000	79	79000
Helix	4012	40	160480

Tabela 5.2: Podatki o vhodih za testiranje aplikacije NoDeK. V drugem stolpcu je število KE, ki sestavljajo strukturo, tretji stolpec prikazuje skupno število iteracij v vseh korakih, zadnji pa je zmnožek prejšnjih dveh in vsebuje celotno število izračunov nad KE.

5.2 Konfiguracije za izvajanje aplikacije

Vsa testiranja z obema aplikacijama smo opravili na več konfiguracijah. Osnovna konfiguracija je razvojno okolje Matlab, ki se izvaja na osebnem računalniku in je služila kot osnova za primerjavo časov in pravilnosti delovanja ogrodja. Čase pri tej konfiguraciji smo merili z vgrajeno funkcijo `profile`.

SISTEM	CPE	RAM	OS
Matlab IDE	2x 2640M@2.80GHz	4 GB	Ubuntu 10.04
Lokalno+PTC	2x 2640M@2.80GHz	4 GB	Ubuntu 10.04
PTC	1x 2640M@2.80GHz	1.5 GB	mOS
Eucalyptus	2x E5504@2.00Ghz	2 GB	mOS
Amazon EC2	2x 2.5 ECU[1]	1.7 GB	Ubuntu 10.04

Tabela 5.3: Podatki o konfiguracijah za testiranje. Vsi sistemi so 32-bitni in uporabljajo Matlab 2012a.

Za izvajanja platforme mOSAIC oz. ogrodja smo uporabili štiri različne konfiguracije. Prva je obsegala zagon Cloudleta v okolju Eclipse na lokalnem računalniku, prek komponente oddaljenega izvajanja pa se je povezala z ostalimi deli ogrodja, ki so se izvajali na PTC. Namen te konfiguracije je zmožnost direktne primerjave med hitrostjo izvajanja Cloudleta in hitrostjo osnovne konfiguracije, saj se oba izvajata na isti infrastrukturi. Druga konfiguracija je zagon celotnega ogrodja na PTC. Pri tej konfiguraciji se ugotavlja primernost PTC za potrebe razvoja in testiranja. Zadnji dve konfiguraciji sta v oblaku. Enkrat se ogrodje izvaja

[1]ECU je Amazonova mera za zmogljivost CPE, 1 ECU ustreza 1.2 GHz Intel Xeon letnik 2007.

na infrastrukturi Eucalyptus (privatni ponudnik West University Of Timisoara) in drugič na infrastrukturi Amazon EC2. Namen je neposredna primerljivost med ponudnikoma in njuna primernost za izvajanje ogrodja ter prednosti in slabosti glede na osnovno okolje. Vse konfiguracije so podrobneje opisane v tabeli 5.3.

5.3 Testiranje in rezultati

5.3.1 Hitrost izvajanja Matlaba

Pri tem testu nas je zanimala morebitna razlika med hitrostjo izvajanja aplikacije v okolju Matlab in hitrostjo izvajanja aplikacije v obliki Cloudleta. Meritev smo opravili tako, da smo najprej aplikacijo, v obliki preoblikovane funkcije, izvedli na osnovni konfiguraciji, nato pa v obliki Cloudleta še na preostalih štirih konfiguracijah, kjer smo merili le čas izvajanja prevedene preoblikovane funkcije. Rezultati za generično aplikacijo so zbrani v tabeli 5.4. Glede na to, da so njeni rezultati tudi povprečni izvajalni časi posameznih testov, smo tudi te vključili v tabelo. V tabeli 5.5 so zbrani podatki za aplikacijo NoDeK.

Test	Matlab IDE	Lokalno+PTC	PTC	Eucalyptus	Amazon EC2
1	0.101	0.114	2.923	0.287	0.261
2	0.972	0.927	2.631	1.051	0.624
3	0.983	0.796	3.910	1.097	0.751
4	1.122	1.201	5.403	1.820	1.073
5	0.646	0.768	3.109	1.118	0.721
6	0.155	0.158	3.828	0.526	0.380
7	3.057	2.935	5.395	6.095	4.729
8	0.700	0.610	3.562	1.180	0.692
9	0.907	0.819	4.059	1.389	0.818
10	0.586	0.532	2.201	0.866	0.599
11	0.595	0.594	1.495	0.636	0.349
12	0.547	0.526	18.28	2.168	1.275
13	0.599	0.516	1.105	0.888	1.006
14	0.008	0.008	0.013	0.012	0.014
15	0.726	0.602	1.354	0.741	1.021
Skupaj	58.53	55.53	296.3	99.37	71.58

Tabela 5.4: Izvajalni časi posameznih testov in skupni čas izvajanja (v s) generične aplikacije na posameznih testnih konfiguracijah.

Iz primerjanja rezultatov izvajanja preoblikovane funkcije na osnovni konfiguraciji z rezultati izvajanja preoblikovane funkcije znotraj prenesene aplikacije na konfiguraciji Lokalno+PTC je pri obeh razvidno, da prenos v oblak ne vpliva na hitrost izvajanja. V veliki

Vhod	Matlab IDE	Lokalno+PTC	PTC	Eucalyptus	Amazon EC2
Dome	18.44	18.11	31.16	29.76	24.44
Arm	70.83	70.51	139.9	106.0	90.05
Bent	140.3	135.4	282.5	215.9	183.0
Helix	290.6	284.2	604.7	432.7	365.2

Tabela 5.5: Izvajalni časi testnih primerov (v s) na posameznih testnih konfiguracijah.

večini primerov se preoblikovane funkcije na platformi izvajajo celo nekaj odstotkov hitreje.

Rezultati na konfiguraciji PTC pri generični aplikaciji kažejo, da je ta konfiguracija primerna le za razvoj in poganjanje manjših bremen, saj so nekateri testi počasnejši tudi za faktor 20 in več, celotno testiranje pa za faktor 5. Največji vpliv na počasno izvedbo ima virtualizacija na osebnem računalniku, ki ne uporablja strojno pospešenih ukazov. Najbolj je to opazno pri testih, ki pogosto zasegajo in sproščajo veliko pomnilnika (test 1 in test 12). Podoben vpliv je zaznati tudi pri aplikaciji NoDeK, kjer je izvajanje počasnejše sicer za manjši faktor, a še vedno dvakrat počasnejše.

Rezultati izvajanja generične aplikacije na konfiguracijah Eucalyptus in Amazon EC2 kažejo na primernost uporabe ogrodja na oblaku. Tukaj sicer prihaja tudi do večjih razlik, vendar je to predvsem posledica različnih sistemov. Tudi tukaj velja, da se najslabše odrežejo testi, ki zasegajo in sproščajo veliko pomnilnika. Verjetno je to tudi posledica dejstva, da je v obeh konfiguracijah na voljo manj glavnega pomnilnika. V nekaterih testih se ogrodje na oblaku izkaže tudi za hitrejšega od okolja Matlab. Podobne ugotovitve potrjujejo tudi rezultati aplikacije NoDeK.

5.3.2 Hitrost izvajanja ogrodja

S tem testom smo merili vpliv celotnega ogrodja na hitrost izvajanja. In sicer tako, da smo merili čas od trenutka, ko uporabnik poda vhod, do trenutka, ko prejme izhod. Bolj natančno - merili smo čas od prejetja zahtevka prek spletne strani `Obdelava` do uspešnega zapisa izhoda v podatkovno bazo. Čas tako vključuje: prevajanje vhodne datoteke, prenos podatkov v Cloudlet, inicializacijo, izvedbo računanja in gradnjo ter zapis izhoda. Rezultati meritev za vse štiri konfiguracije so zbrani v tabelah 5.6 in 5.7.

Pri obeh aplikacijah je razvidno, da prevajanje vhodne datoteke s prevajalnikom PPMM poteka zelo hitro, edina izjema je pri aplikaciji NoDeK pri vhodu “Helix”. Razlog je v tem, da je ta vhod zelo obsežen in zavzema ∼10.000 vrstic (377 KB), medtem ko drugi največji vhod zavzema le ∼400 vrstic (13 KB). Prevajanje vhoda k celotnemu izvajalnemu času ogrodja prispeva zanemarljiv delež.

Sklop	Lokalno+PTC	PTC	Eucalyptus	Amazon EC2
Prevajanje	0.004	0.021	0.003	0.004
Inicializacija	2.534	6.345	1.956	2.171
Cloudlet	55.53	296.3	99.37	71.58
Ostalo	4.177	8.191	3.386	3.963
Skupaj	62.26	310.8	104.7	77.72
Izkoristek	0.892	0.953	0.949	0.921

Tabela 5.6: Izvajalni časi (v s) posameznih sklopov ogrodja pri generični aplikaciji. Izkoristek predstavlja razmerje med izvajalnim časom Cloudleta in celotnim ogrodjem.

Precej bolj opazen delež k času izvajanja prispeva inicializacija. Inicializacija je neodvisna od velikosti problema in tudi na zmogljivih konfiguracijah zahteva ∼1.5 s. V teoriji je potrebno inicializacijo opraviti le ob prvem klicu Cloudleta, pri vseh nadaljnjih pa ni več potrebna. Žal pa smo ob podrobnejšem testiranju ugotovili, da aplikacija v primeru zaporednih klicev ne zna pravilno sproščati pomnilnika. Prevedena aplikacija v ozadju (izven nadzora razvijalca) prek MCR zasega pomnilnik za svoje potrebe, vendar ga ob koncu izvajanja včasih ne sprosti. Edini način za sprostitev tega pomnilnika je uničenje primerka prevedenega razreda in nato ponovna inicializacija, kar vodi v daljši zagonski čas ob naslednjem klicu.

Preostali delež režije ogrodja odpade na prenos vhoda med spletno stranjo in Cloudletom ter na gradnjo izhoda in njegov prenos v podatkovno bazo. Pri prenosu velikost vhoda in izhoda skoraj ne vpliva na hitrost prenosa, večji del predstavljajo režijski stroški kot npr. vzpostavitve povezave, čakanje v čakalni vrsti itd. Gradnja izhoda (zapis seznama matrik v tekstovno datoteko) je seveda odvisna od velikosti izhoda, pri aplikaciji NoDeK je njegova velikost lahko precejšnja (npr. izhod pri vhodu “Helix” zavzema ∼25 MB).

Izkoristek ogrodja (delež časa, ki se porabi za izvajanje Cloudleta, proti izvajalnemu času celotnega ogrodja) je kljub vsem naštetim porabnikom precej visok. Pri “lažjih” vhodih je izkoristek lahko le ∼70 %, vendar se odstotek z zahtevnostjo problema veča in lahko doseže tudi ∼95 %.

5.3.3 Skalabilnost prenesene aplikacije

Poleg sposobnosti enostavnega izvajanja nas je zanimalo tudi delovanje aplikacije pri več hkratnih uporabnikih oz. ali je aplikacija skalabilna. Testirali smo tako, da smo na vhodu zvrstili osem enakih zahtev, nato pa smo merili čas izvajanja vseh obdelav. Aplikacijo smo testirali večkrat, vsakič z drugim številom primerkov Cloudleta (naprej en primerek, nato dva, štiri in na koncu osem). Za izvajanje teh poskusov smo uporabili le zadnji dve konfiguraciji,

Sklop	Lokal+PTC	PTC	Eucalyptus	Amazon EC2
Dome				
Prevajanje	0.181	0.162	0.041	0.057
Inicializacija	2.144	3.698	1.777	1.414
Cloudlet	18.11	31.16	29.76	24.44
Ostalo	4.738	6.541	2.067	2.281
Skupaj	25.17	41.56	33.65	28.19
Izkoristek	0.719	0.750	0.885	0.867
Arm				
Prevajanje	0.125	0.332	0.207	0.235
Inicializacija	1.895	3.412	1.391	1.343
Cloudlet	70.51	139.9	106.0	90.05
Ostalo	7.042	9.380	4.486	5.025
Skupaj	79.57	153.0	112.1	96.66
Izkoristek	0.886	0.914	0.946	0.932
Bent				
Prevajanje	0.060	0.126	0.043	0.034
Inicializacija	1.841	3.570	1.384	1.428
Cloudlet	135.4	282.5	215.9	183.0
Ostalo	6.963	9.621	4.601	5.196
Skupaj	144.3	295.8	222.0	189.6
Izkoristek	0.939	0.955	0.973	0.965
Helix				
Prevajanje	0.474	2.172	0.584	0.574
Inicializacija	1.703	6.377	1.498	1.509
Cloudlet	284.2	604.7	432.7	365.2
Ostalo	13.56	21.71	14.49	16.16
Skupaj	300.0	635.0	449.3	383.4
Izkoristek	0.948	0.952	0.963	0.952

Tabela 5.7: Izvajalni časi (v s) posameznih sklopov ogrodja pri aplikaciji NoDeK. Izkoristek predstavlja razmerje med izvajalnim časom Cloudleta in celotnim ogrodjem.

saj so se prve tri (Matlab, lokalno+PTC in PTC) izkazale za premalo zmogljive za poganjanje več primerkov Cloudleta. Preostali dve konfiguraciji pa smo spremenili tako, da smo povečali število VM v gruči. Na prvi VM so se izvajali vsi deli ogrodja, razen Cloudletov. Na preostalih VM pa so se izvajali le Cloudleti, pri čemer je imel vsak za izvajanje na voljo svoje jedro. Rezultati so podani v tabelah 5.8 in 5.9, za aplikacijo NoDeK smo rezultate tudi grafično prikazali na sliki 5.3.

Delovanje aplikacije na obeh sistemih pri osmih hkratnih vhodih pri enem zagnanem

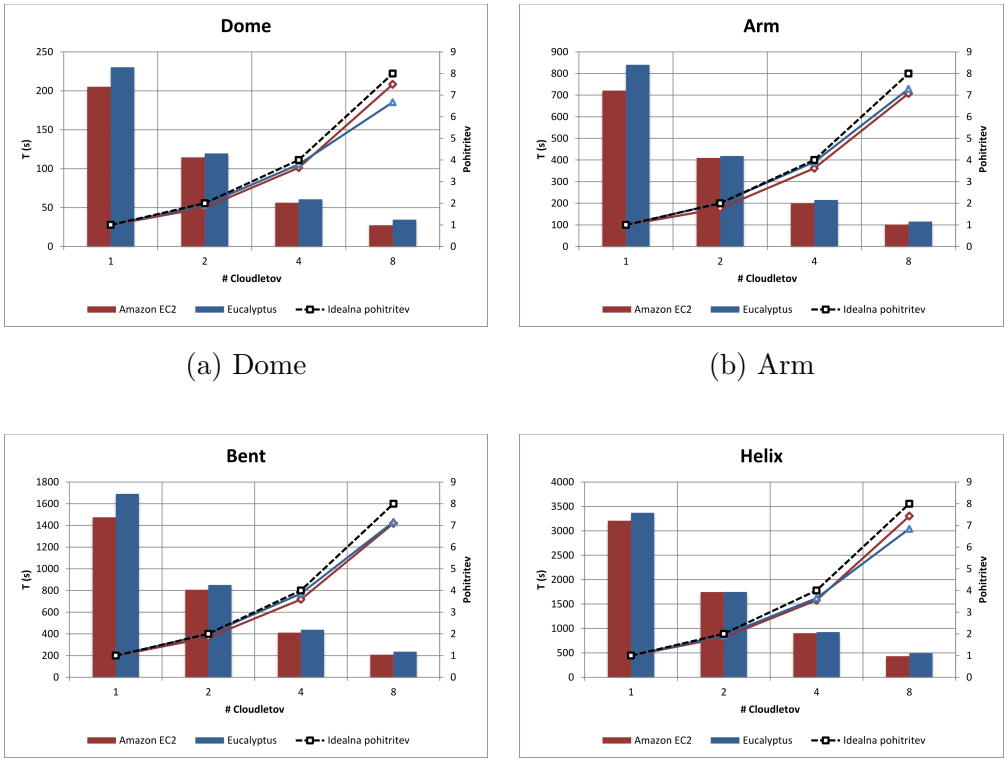
Št. primerkov	Eucalyptus	Amazon EC2
1 (2 VM)	838.8	644.6
2 (2 VM)	419.3	323.5
4 (3 VM)	210.5	160.5
8 (5 VM)	106.3	79.51

Tabela 5.8: Izvajalni časi (v s) ogrodja pri generični aplikaciji za osem hkratnih izračunov glede na število zagnanih primerkov Cloudleta. Poleg števila primerkov je v oklepaju podano število uporabljenih VM.

Št. primerkov	Eucalyptus	Amazon EC2
Dome		
1 (2 VM)	230.2	205.1
2 (2 VM)	119.6	114.5
4 (3 VM)	60.75	56.30
8 (5 VM)	34.51	27.35
Arm		
1 (2 VM)	840.2	720.8
2 (2 VM)	418.5	409.7
4 (3 VM)	215.1	199.4
8 (5 VM)	115.3	101.9
Bent		
1 (2 VM)	1692	1475
2 (2 VM)	850.9	806.5
4 (3 VM)	438.5	411.0
8 (5 VM)	236.2	207.8
Helix		
1 (2 VM)	3371	3207
2 (2 VM)	1749	1745
4 (3 VM)	924.6	904.4
8 (5 VM)	492.7	431.8

Tabela 5.9: Izvajalni časi (v s) ogrodja pri aplikaciji NoDeK za osem hkratnih izračunov glede na število zagnanih primerkov Cloudleta. Poleg števila primerkov je v oklepaju podano število uporabljenih VM.

primerku Cloudleta je pričakovano. Ogrodje porabi skoraj natančno osemkrat več časa za izvedbo vseh izračunov kot pri obdelavi le enega vhoda. Z večanjem števila zagnanih primerkov se zmanjšuje tudi celotni čas izvajanja vseh zahtevkov. Glede na rezultate gre za skoraj popolnoma linearno zmanjšanje, iz česar lahko sklepamo, da je zadeva skalabilna v primeru več sočasnih uporabnikov. Pri uporabi osmih primerkov Cloudleta se vse zahteve izvajajo sočasno, a je skupni čas izvajanja vseeno daljši kot v primeru enega primerka in enega vhoda.



(a) Dome (b) Arm

(c) Bent (d) Helix

Slika 5.3: Graf odvisnosti celotnega izvajalnega časa za osem sočasnih bremen v odvisnosti od števila zagnanih primerkov Cloudleta pri aplikaciji NoDeK. Z barvno črto je prikazana tudi dosežena pohitritev izvajanja pri posameznem ponudniku, za primerjavo pa je dodana črna črta, ki predstavlja idealno pohitritev.

Razlog je v tem, da so v tem primeru hkrati obremenjeni vsi ostali deli ogrodja - na začetku je treba vsem Cloudletom posredovati podatke, na koncu izračuna pa želijo rezultate vsi hkrati zapisati nazaj v podatkovno bazo. V primerih, ko se vse zahteve ne izvajajo sočasno, se obremenitev ogrodja pri naslednjem izračunu časovno zamakne, zato je vpliv na celotni čas izvajanja manj izrazit.

5.3.4 Ohranitev numeričnih lastnosti

Vsakič, ko imamo opravka z numeričnim reševanjem, se moramo zavedati, da so pri tem postopku vedno prisotne numerične napake [16]. Čeprav je to skrb predvsem pri razvoju aplikacije npr. z uporabo števil v plavajoči vejici z dvojno natančnostjo po standardu IEEE754, uporaba stabilnih algoritmov itd., pa je potrebno biti na to dejstvo pozoren tudi ob preoblikovanju aplikacije ali njenem prenosu na drug sistem, npr. prenos iz 32- na 64-bitno arhitekturo.

Ogrodje obstoječo aplikacijo Matlab v oblak prenese s pomočjo prevajalnika Matlab, ki zagotavlja, da se prevedeni programi izvajajo enako kot v osnovnem okolju Matlab [28]. Trditev smo preverili s pomočjo aplikacije NoDeK. Razlog za opustitev generične aplikacije pri tem testu je, da aplikacija nima determinističnega izhoda, ker pri računanju uporablja generator naključnih števil. Po drugi strani je aplikacija NoDeK popolnoma deterministična, dodatno pa še precej občutljiva na matematično stabilnost algoritmov, saj nad vhodnimi podatki izvaja zahtevne matematične operacije. Te operacije se prav tako izvajajo iterativno, tako da bi najmanjša napaka pri katerikoli operaciji vodila v opazno drugačen rezultat.

S primerjanjem pridobljenih izhodov izvajanja vseh testnih primerov na vseh konfiguracijah se je pokazalo, da med njimi ni nobenih razlik, torej da ogrodje popolnoma ohranja numerične lastnosti osnovne aplikacije. Vendar, ko smo ogrodje preizkusili na variaciji konfiguracije Amazon EC2, ki je temeljila na 64-bitnem operacijskem sistemu, je prišlo do minimalnega odstopanja v primerjavi z rezultati na 32-bitnih sistemih - rezultati so se razlikovali na šesti decimalki. Pri uporabi ogrodja je torej potrebno za pravilno delovanje zagotoviti enako arhitekturo VM, kot jo uporablja obstoječa aplikacija.

5.4 Povzetek testiranja

V prejšnjem poglavju predstavljeno ogrodje smo v tem poglavju testirali s prenosom dveh obstoječih aplikacij Matlab na štiri različne konfiguracije. S predstavljenimi testi smo ugotovili, da prenesena aplikacija ohranja svoje numerične lastnosti in da njena hitrost izvajanja ostane primerljiva z osnovno aplikacijo. Ogrodje k izvajalnemu času doda določene režijske stroške, vendar so ti v veliki meri neodvisni od prenesene aplikacije (čas inicializacije je vedno konstanten, enako velja tudi za čas vzpostavitve povezave med posameznimi deli ogrodja) in pri aplikacijah z daljšim časom izvajanja in majhnim vhodom/izhodom predstavljajo le ∼5 % časa. V zameno zna ogrodje poljubno aplikacijo Matlab pretvoriti v skalabilno aplikacijo, ki se izvaja v poljubnem oblaku in je v obliki spletne storitve na voljo množici sočasnih uporabnikov.

6 Možnosti paralelizacije aplikacije NoDeK

S predstavljenim ogrodjem (4) smo aplikacijo NoDeK (5.1.2) preoblikovali v skalabilno spletno storitev. Ena izmed omejitev takšnega pristopa je, da je aplikacija zmožna razpoložljive vire (npr. več VM) izkoristiti le v primeru več sočasnih uporabnikov. Če je sočasnih zahtev manj kot virov, določeni viri mirujejo, namesto da bi jih aplikacija izkoristila za hitrejši izračun posamezne zahteve. V izogib tej omejitvi v tem poglavju predstavljamo način, kako aplikacijo NoDeK preoblikovati v paralelno aplikacijo, ki se izvaja v oblaku in zna za hitrejše računanje izkoristiti vse razpoložljive vire.

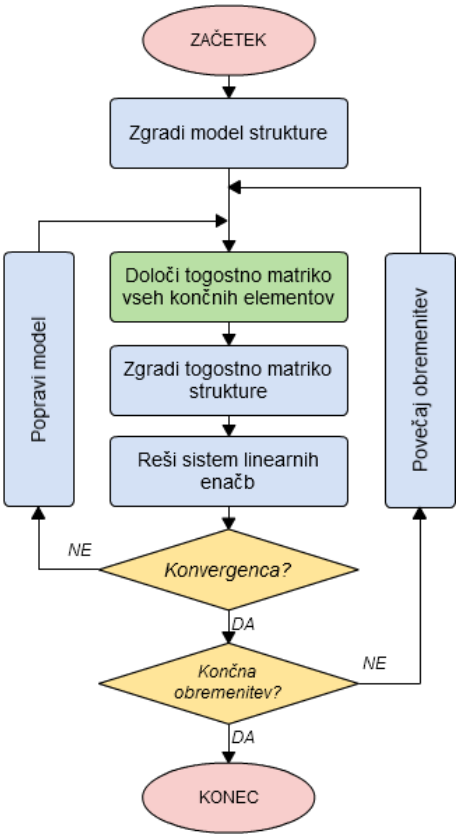
Vpeljava vzporednega računanja za potrebe hitrejšega računanja posamezne zahteve s pomočjo več porazdeljenih virov je običajno zahteven postopek [14] in navadno zahteva zamenjavo uporabljenih algoritmov, reorganizacijo podatkov itd., kar vodi v globoke posege v izvorno kodo. Postopek tudi ni splošen, saj se razlikuje od aplikacije do aplikacije, enako pa velja za njegovo uspešnost. Nekatere aplikacije lahko s paralelizacijo ogromno pridobijo, drugih pa zaradi njihove strukture in potrebe po vsaj delnem zaporednem izvajanju z uporabo vzporednega računanja niti ni možno izvajati dosti hitreje(Amdahlov zakon). Iz tega sledi, da ne govorimo več o ogrodju, ki bi bilo primerno za splošne aplikacije, pač pa je potrebno postopek prenosa v oblak prilagoditi vsaki aplikaciji posebej.

V podpoglavju 6.1 je predstavljena notranja zgradba aplikacije NoDeK, katere poznavanje je nujno za vpeljavo vzporednega računanja. Pristopi k paralelizaciji in prenos v oblak so predstavljeni v podpoglavju 6.2. Sledi podpoglavje 6.3 z opisom testiranj prenesene paralelne aplikacije in njihovimi rezultati. V zadnjem (6.4) je predstavljen še povzetek uspešnosti postopka.

6.1 Zgradba aplikacije NoDeK

Za uspešno paralelizacijo moramo razumeti notranjo zgradbo aplikacije in identificirati mesta, kjer lahko vpeljemo vzporedno računanje. Diagram poteka (angl. flowchart) aplikacije NoDeK na sliki 6.1 prikazuje potek možnih poti podatkov od vhoda proti izhodu in njihove medsebojne odnose. Iz diagrama je razvidno, da postopek izračuna poteka iterativno. V pr-

vem koraku se iz vhodnih podatkov zgradi model celotne strukture, sestavljene iz KE. Nato se celotno obremenitev razdeli glede na vnaprej določeno število korakov. V zanki se potem v vsakem koraku obremenitev na strukturo enakomerno povečuje.



Slika 6.1: Diagram poteka aplikacije NoDeK.

Odziv strukture na delno obremenitev se računa tako, da se najprej izračuna odziv vsakega KE, kot da so neodvisni eden od drugega, nato pa se upošteva še njihov medsebojni vpliv. Le-ta se izračuna prek reševanja sistema linearnih enačb, ki jih definira togostna matrika strukture. Ker se končni elementi računajo neodvisno, kasneje pa se zaradi medsebojnega vpliva rezultati popravijo, je potrebno postopek iterativno ponavljati z upoštevanjem popravkov. Izračun se ponavlja, dokler ne dosežemo konvergence (popravek medsebojnega vpliva je manjši od predpisane natančnosti).

Lastnost, ki jo lahko uporabimo za vpeljavo vzporednega računanja, je, da lahko za izračun odziva celotne strukture na podano breme odziv vsakega KE izračunamo neodvisno od ostalih. To pomeni, da lahko izračune KE razporedimo po vseh razpoložljivih virih. Žal narava algoritma zahteva, da moramo pred naslednjo iteracijo vse izračune KE zbrati v togostno matriko strukture in rešiti sistem linearnih enačb, da se oceni njihov kumulativni

vpliv.

Teoretično lahko zgornjo mejo pohitritve izračunamo z merjenjem časa izvajanja zaporednega (izračun togostne matrike in korak iteracije) in vzporednega dela (izračun KE) celotne aplikacije. Meritve za posamezne vhode (tabela 5.2) smo opravili in zbrali v tabeli 6.1. Na podlagi teh meritev smo z Amdahlovim zakonom izračunali, da lahko teoretično dosežemo le ∼4x pohitritev delovanja, ob nerealnih predpostavkah, da lahko vse izračune KE opravimo v neskončno kratkem času in da komunikacija med VM poteka hipno brez zakasnitev.

Vhod	T(zaporedno)	T(vzporedno)	T(skupaj)	Možna pohitritev
Dome	4.779	13.66	18.44	3.859
Arm	18.18	52.65	70.83	3.895
Bent	35.51	104.8	140.3	3.951
Helix	72.68	217.92	290.6	3.998

Tabela 6.1: Izvajalni časi vzporednega in zaporednega dela (v s) aplikacije NoDeK pri različnih vhodih. Teoretična pohitritev je izračunana kot T(skupaj)/T(zaporedno).

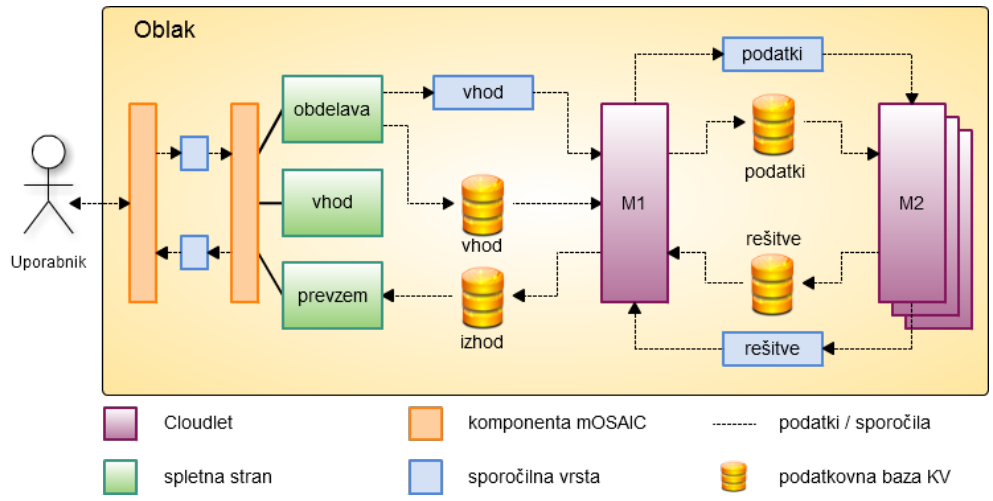
6.2 Paralelizacija aplikacije

Celotno osnovno aplikacijo je potrebno najprej razbiti na dva dela, na del, ki opravi zaporedni del algoritma, in del, ki vsebuje vzporedni del algoritma. Tokrat imamo namesto enega samega Cloudleta, ki opravi ves izračun, dva različna Cloudleta: Cloudlet M1 izvaja zaporedne korake algoritma: zgradi celoten model strukture, sestavi togostno matriko ter reši sistem linearnih enačb, Cloudlet M2 pa izvaja vzporedni del algoritma - izračun KE.

Nadalje je potrebno med njima vzpostaviti izmenjavo podatkov. Po opravljeni inicializaciji mora Cloudlet M1 podatke za izračun vseh KE razbiti na več delov in jih posredovati Cloudletu M2. Ta mora vse prejete KE preračunati in jih poslali nazaj Cloudletu M1 v ponovno obravnavo. Tak cikel nato poteka iterativno, dokler ne dosežemo končnega stanja. To vodi v ogromno količino izmenjanih podatkov in omejuje pohitritev, ki bi jo sicer lahko dosegli.

Ker v poglavju 4 predstavljeno ogrodje ne omogoča prenosa podatkov med posameznimi Cloudleti in ker hkrati predpostavlja popolno neodvisnost med njimi, je potrebno njegovo arhitekturo ustrezno spremeniti. Prenovljena arhitektura aplikacije je prikazana na sliki 6.2.

Levi del arhitekture je enak osnovnemu ogrodju in skrbi za prenos zahtevka od uporabnika k Cloudletu M1 z vmesnim prevajanjem ter za pridobitev rezultatov. Glavna razlika je na desni strani arhitekture, kjer je dodana možnost izmenjave podatkov med Cloudletoma. Pri

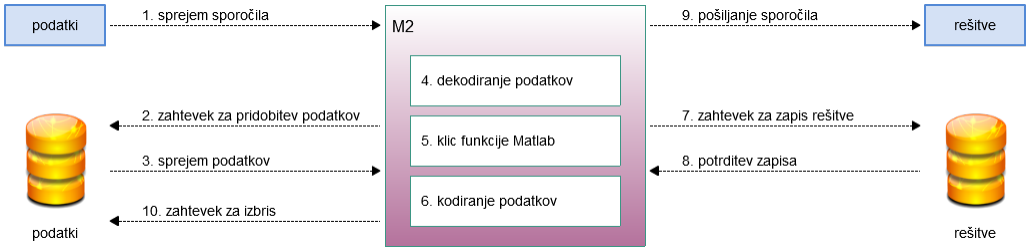


Slika 6.2: Arhitektura prenesene aplikacije NoDeK za vzporedno računanje KE.

tej arhitekturi je lahko zagnanih več primerkov Cloudleta M1, vendar je bolj smiselno imeti zagnanih več primerkov Cloudleta M2, da ti izvajajo vzporedni del algoritma in tako bolje izkoristijo razpoložljive vire.

Zaradi ogromne količine izmenjanih podatkov mora biti posebna pozornost pri uvajanju take arhitekture namenjena optimizaciji izmenjave podatkov. Ker imamo opravka le z dvema Cloudletoma, je dovolj, če se osredotočimo na komunikacijo, ki jo izvaja Cloudlet M2. V prvem koraku Cloudlet prejme sporočilo o tem, da so podatki na voljo. Nato sproži asinhroni klic podatkovni bazi `podatki` za pridobitev podatkov. Ko so podatki na voljo, jih prevzame in pretvori v ustrezen javanski objekt (angl. deserialization), nato z njim kliče prevedeno funkcijo, ki opravi izračun. Rezultate funkcije mora nato spet pretvoriti v ustrezno binarno obliko (angl. serialization) ter jih z asinhronim klicem zapisati v podatkovno bazo `rešitve`. Ko prejme potrditev, da je bil zapis uspešen, lahko pošlje sporočilo v sporočilno vrsto `rešitve`, da so rezultati na voljo, hkrati pa lahko izbriše vhodne podatke. Celotno zaporedje korakov, ki sodelujejo pri izmenjavi na strani Cloudleta M2, je prikazano na sliki 6.3. Izmenjava na strani Cloudleta M1 je podobna, a vključuje še dodatna koraka: razbijanje vhodnih podatkov na posamezne neodvisne dele in sestavljanje rezultatov posameznih neodvisnih delov nazaj v celoto.

Iz zaporedja korakov je razvidno, da komunikacija vedno poteka dvostopenjsko - najprej je Cloudlet obveščen o tem, da so podatki na voljo in kje se nahajajo, nato sledi še njihov prevzem. Komunikacija bi potekala precej hitreje, če bi sporočilo že v prvem koraku vsebovalo tudi potrebne podatke in bi bil tako drugi korak nepotreben, vendar pa je žal največja velikost sporočila precej nizka, zato taka rešitev ni mogoča.



Slika 6.3: Zaporedje korakov pri izmenjavi podatkov znotraj Cloudleta M2.

Celotna predstavljena arhitektura je splošna in jo je mogoče uporabiti za vse aplikacije, ki delujejo po principu izmenjave podatkov med zaporedno/vzporednimi deli algoritma. Vendar pa je sedaj uporaba precej bolj zahtevna. Uporabnik mora pred prenosom v oblak aplikacijo preoblikovati v tri funkcije Matlab: prva je namenjena obdelavi uporabnikovega vhoda, druga izvaja vzporedni del algoritma, tretja pa je namenjena obdelavi rezultatov vzporednega računanja. Prva in tretja funkcija morata biti vključeni v Cloudlet M1, druga pa v Cloudlet M2. Pri tem mora veljati tudi, da mora biti izhod prve in tretje funkcije enak, hkrati pa je to vhod v drugo funkcijo. Dodatno moramo v Cloudletu M1 definirati tudi način razbijanja izhoda prve in tretje funkcije na posamezne dele vhodov za drugo funkcijo, da se pri dani aplikaciji zagotovi ustrezno razmerje med režijskimi stroški (angl. overhead) izmenjave na eni strani in pohitritvijo zaradi vzporednega računanja na drugi strani. Žal mora biti ta delitev konstantna, ne glede na število zagnanih primerkov Cloudleta M2, saj je ena izmed zahtev platforme mOSAIC, da mora biti izvajanje Clouldetov neodvisno od števila zagnanih primerkov. V nasprotnem primeru bi bila optimalna delitev na toliko enakih delov, kolikor je zagnanih primerkov Cloudleta M2.

Celotno aplikacijo NoDeK smo prenesli tako, da smo osnovno aplikacijo najprej preoblikovali v tri omenjene funkcije in nato s prevajalnikom MCJ zgradili zahtevana Cloudleta.

6.3 Testiranje

6.3.1 Vpliv drugačne organizacije algoritma

Najprej nas je zanimalo, koliko izgubimo samo z razbitjem osnovne aplikacije z ene na tri funkcije, brez izmenjave podatkov med Cloudletoma. V ta namen smo zgradili nov Cloudlet, ki je združeval vse tri funkcije, ga vključili v osnovno ogrodje in ponovili merjenja, predstavljena v podpoglavju 5.3.1 (merili smo torej le izvajalni čas prevedene funkcije, ne celotne aplikacije), na enakih vhodih (podpoglavje 5.1.2) in na enakih konfiguracijah (podpoglavje 5.2). Rezultati meritev so zbrani v tabeli 6.2.

Vhod	Matlab IDE	Lokalno+PTC	PTC	Eucalyptus	Amazon EC2
Dome	18.62	24.15	59.61	36.45	32.17
Arm	71.77	92.19	195.5	117.0	104.6
Bent	140.5	188.5	358.8	238.1	217.5
Helix	291.0	344.8	787.6	476.5	430.4

Tabela 6.2: Izvajalni časi (v s) posameznih testnih primerov na posameznih testnih konfiguracijah z novim Cloudletom sestavljenim iz treh funkcij.

Iz primerjave meritev uporabe ene funkcije znotraj Cloudleta (tabela 5.5) z meritvami uporabe treh funkcij znotraj Cloudleta je razvidno, da se izvajalni časi podaljšajo. Pri konfiguraciji Matlab so razlike minimalne, pri konfiguraciji PTC pa se izvajanje lahko podaljša tudi za ∼40 %, podobno je tudi pri konfiguraciji Lokalno+PTC. Pri obeh ponudnikih je izvajanje počasnejše, nekje v obsegu 10-15 %. Pri vseh konfiguracijah, ki vključujejo Cloudlet, je izvajanje torej opazno počasnejše. Razlog je v tem, da si morajo funkcije večkrat izmenjevati podatke, kar vodi v veliko kontekstnih preklopov (angl. context switch) med Javo in prevedeno funkcijo, ki v ozadju še vedno temelji na pogonu Matlab. Pri osnovnem ogrodju se preklop zgodi le enkrat, ob klicu prevedene funkcije. Celotni izračun se nato izvede v tej funkciji, zato ne pride do upočasnitve v primerjavi z osnovno konfiguracijo.

6.3.2 Vpliv vzporednega računanja

Naslednje testiranje je namenjeno merjenju izvajalnega časa prenesene aplikacije ob več zagnanih primerkih Cloudleta M2 oziroma merjenju vpliva vzporednega računanja. Pri tem smo merili izvajalni čas celotne aplikacije (čas od prejetja zahtevka prek spletne strani `Obdelava` do uspešnega zapisa izhoda v podatkovno bazo) za posamezni vhod. Celotna množica KE se je za potrebe vzporednega računanja delila na osem enakih delov. Zaradi omejenosti razpoložljivih virov smo teste opravili le na dveh konfiguracijah (Eucalyptus in Amazon EC2). Konfiguraciji smo nastavili tako, da je Cloudlet M1 imel na voljo lastno VM, enako je veljalo za preostale dele aplikacije (prehod HTTP, strežnik spletnih strani, sporočilne vrste, podatkovna baza KV so se izvajali na lastni VM). Ostali viri so bili namenjeni primerkom Cloudleta M2, pri čemer je imel vsak primerek na voljo lastno procesorsko jedro. Opravljene meritve so zbrane v tabeli 6.3.

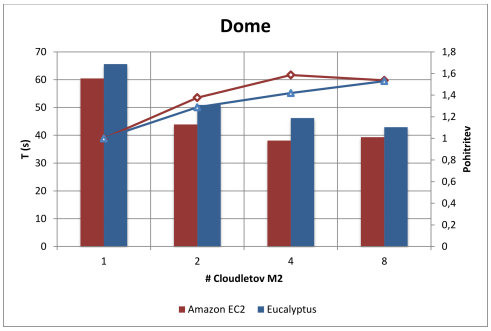
Iz meritev je razvidno, da s paralelnim izvajanjem ne dosežemo želene pohitritve izvajanja. Zaradi že v osnovi počasnejšega algoritma in zaradi latence pri prenosu podatkov je računanje pri enem primerku Cloudleta M2 precej počasnejše od osnovnega algoritma (tabela 5.5), za konfiguracijo Amazon EC2 ∼80 % pri vhodu Dome in ∼15 % pri vhodu Helix (za Eucalyptus ∼130 % za Dome in ∼40 % za Helix). Z večanjem števila primerkov Cloudleta M2 se sicer

Št. primerkov M2	Eucalyptus	Amazon EC2
Dome		
1 (3 VM)	65.62	60.43
2 (3 VM)	50.95	43.89
4 (4 VM)	46.23	38.09
8 (6 VM)	42.90	39.33
Arm		
1 (3 VM)	142.4	134.4
2 (3 VM)	101.6	97.18
4 (4 VM)	98.43	82.89
8 (6 VM)	93.10	87.39
Bent		
1 (3 VM)	296.8	262.9
2 (3 VM)	210.6	194.6
4 (4 VM)	182.8	166.9
8 (6 VM)	197.5	172.1
Helix		
1 (3 VM)	537.9	513.5
2 (3 VM)	373.0	363.6
4 (4 VM)	300.7	306.8
8 (6 VM)	364.9	331.3

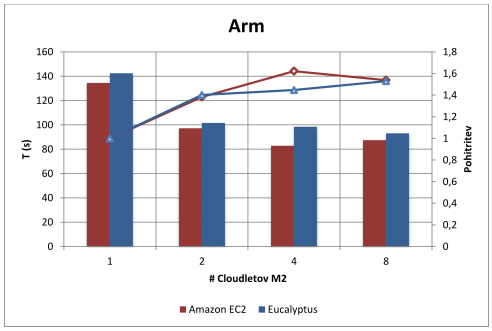
Tabela 6.3: Izvajalni časi (v s) aplikacije NoDek z vzporednim računanjem glede na število zagnanih primerkov Cloudleta M2. Poleg števila primerkov Cloudleta M2 je v oklepaju podano število uporabljenih VM.

čas izvajanja krajša in pri težjih problemih z velikim številom KE (Bent, Helix) se celoten izračun lahko izvede hitreje kot pri osnovnem prenosu z ogrodjem, vendar pa je za dosego te pohitritve potrebnih štirikrat več sredstev.

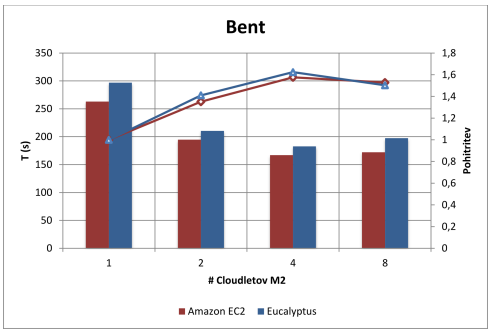
Iz meritev je opazna tudi določena anomalija. V večini primerov z osmimi primerki Cloudleta M2 je pohitritev v primerjavi z uporabo samo štirih primerkov negativna oziroma se čas izvajanja podaljša. Vzrok je v zasičenosti zmogljivosti komunikacijskih povezav med posameznimi VM zaradi velike količine izmenjanih podatkov. Skupni čas prenosa podatkov v tem primeru je daljši kot prihranek, ki ga prinese vzporedno računanje na več primerkih Cloudleta M2.



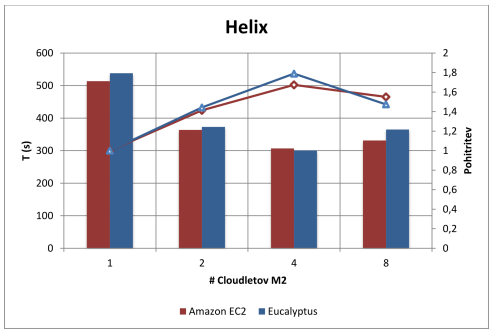
(a) Dome



(b) Arm



(c) Bent



(d) Helix

Slika 6.4: Grafična primerjava izvajalnih časov (v s) prenesene aplikacije pri posameznih vhodih glede na število primerkov Cloudleta M2 pri aplikaciji NoDeK. S črto je prikazana tudi dosežena pohitritev izvajanja.

6.3.3 Paralelizacija v primeru več sočasnih uporabnikov

Zadnji test bi bilo smiselno pogledati tudi v luči več sočasnih uporabnikov, skladno s testom, izvedenim v podpoglavju 5.3.3. Vendar pa nam testa zaradi hrošča v platformi mOSAIC ni uspelo izvesti. Za testiranje več sočasnih uporabnikov bi potrebovali več primerkov Cloudleta M1, odkriti hrošč pa ne dopušča, da bi Cloudletu M2 med izvajanjem določili, kateremu primerku Cloudleta M1 mora vrniti rezultat. Če je torej v sistemu prisotnih več primerkov, se rezultati med njimi razporedijo naključno. Ker brez vseh zbranih rezultatov Cloudlet M1 ne more nadaljevati izvajanja (ne zmore rešiti sistema enačb), je torej edina rešitev samo en zagnan primerek Cloudleta M1. Glede na doseženo zasičenost celotnega sistema pri uporabi več kot štirih primerkov Cloudleta M2, menimo, da odprava omejitve na en primerek Cloudleta M1 ne bi prinesla opaznih pohitritev.

6.4 Povzetek paralelizacije

V tem poglavju predstavljen postopek paralelizacije aplikacije NoDeK temelji na dejstvu, da lahko izračunamo odziv posameznega KE neodvisno od ostalih. V ta namen smo začetni algoritem razdelili na dva dela. Prvi je vzporedni del, ki računa odziv KE, drugi pa zaporedni del, ki razreši sistem enačb, ki jih definirajo posamezni KE. Oba dela se ciklično izmenjujeta, dokler z iteracijo ne dosežemo stabilnosti sistema. Glede na začetne meritve originalne aplikacije bi torej s takim pristopom teoretično dosegli do ∼4x pohitritev izračunov. Po realnih meritvah se izkaže, da s paralelnim izvajanjem ne dosežemo želene pohitritve izvajanja. Eden izmed razlogov je počasnejše delovanje samega algoritma zaradi kontekstnih preklopov, precej večji vpliv pa ima ogromna količina izmenjanih podatkov med obema deloma algoritma. Ta povzroči zasičenost omrežja med posameznimi VM, kar vodi v dvig latence. Prenos podatkov v tem primeru lahko vzame celo več časa, kot je prihranek, ki ga prinese vzporedno računanje. Le v primeru zahtevnejših problemov z velikim številom KE se celoten izračun lahko izvede hitreje kot pri osnovnem prenosu z ogrodjem, vendar pa je za dosego te pohitritve potrebnih štirikrat več sredstev.

7 Zaključek

Računalništvo v oblaku je v zadnjih letih postalo zelo priljubljeno, predvsem zaradi privlačnih lastnosti, kot so dostopnost, nizka cena, zmogljivost itd. Ker je oblačne zmogljivosti mogoče najeti pri ustreznem ponudniku samo za določeno obdobje, je takšno okolje zaželeno tudi za izvajanje zahtevnih inženirskih aplikacij, ki dostikrat potrebujejo veliko zmogljivosti, a le za kratek čas. Vendar pa je prenos aplikacij v to okolje težaven proces, ki zahteva specifična znanja o okolju, da ga lahko polno izkoristi.

V delu smo si ogledali možne pristope k preoblikovanju inženirske aplikacije v obliko spletne storitve, ki se izvaja v oblaku. Pri tem smo se osredotočili le na aplikacije, razvite v okolju Mathworks Matlab. Najprej smo opravili pregled računalništva v oblaku in si ogledali ustreznost posameznih modelov za naš namen - predvsem nas je zanimala podpora okolju Matlab.

V nadaljevanju smo se osredotočili na odprtokodno platformo mOSAIC, katere namen je vzpostaviti okolje PaaS nad okoljem IaaS. V splošnem okolje PaaS razvijalcem ne omogoča nadzora nad spodaj ležečo infrastrukturo, a v primeru platforme mOSAIC je ta kontrola možna in tako primerna za potrebe našega dela. Velika prednost uporabe platforme mOSAIC je tudi v tem, da odpravi odvisnost aplikacije v oblaku od ponudnika IaaS, saj njena uporaba ni omejena na določenega ponudnika. Že v osnovi podpira vse večje ponudnike IaaS, ima pa tudi možnost dodajanja podpore svojega ponudnika (podporo za enega izmed njih smo predstavili tudi v delu).

V okviru magistrske naloge smo izdelali ogrodje, ki omogoča prenos obstoječih inženirskih aplikacij v oblak, temelječ na platformi mOSAIC. Pri tem smo se omejili na podporo splošnim aplikacijam, napisanim v programskem jeziku Matlab. Obstoječo, poljubno kompleksno aplikacijo je mogoče prek uporabe MCJ prenesti v model SaaS, ki se izvaja v oblaku, z minimalnimi posegi v izvorno kodo. Postopek je primeren tudi za razvijalce, ki niso seznanjeni z vsemi podrobnostmi oblačnega okolja, ker za polno izkoriščenost oblaka samodejno poskrbi ogrodje. Dodatno udobje je zagotovljeno s podporo sprejemanja vhoda v obliki matrik, definiranih v jeziku Matlab. Enako velja tudi za izhod.

Za potrebe testiranja ogrodja smo uporabili inženirsko aplikacijo NoDeK, ki se uporablja

za analizo struktur pod statičnimi obremenitvami s pomočjo metode KE. Rezultati testiranj kažejo, da z ogrodjem prenesena aplikacija deluje popolnoma enako kot izvorna aplikacija v smislu ponovljivosti izračunov in numerične stabilnosti. Tudi performančne lastnosti s prenosom ostanejo skoraj nespremenjene, saj je vpliv režije, ki ga doda ogrodje, minimalen. Z dodatnimi testiranji smo potrdili tudi, da s prenosom na oblak dostop do aplikacije lahko omogočimo večjemu številu uporabnikov, saj rezultati kažejo, da je ogrodje zmožno izkoristiti vse razpoložljive zmogljivosti za več hkratnih uporabnikov in da je torej skalabilno.

Z ogrodjem prenesena aplikacija zna izkoristiti zmogljivosti okolja le za sočasno računanje v primeru več uporabnikov, ne pa tudi za hitrejše računanje problemov ali pa računanje zahtevnejših problemov. Omejitev smo reševali s paralelizacijo osnovnega algoritma in drugačno arhitekturo prenosa v oblak. Rezultati testiranj kažejo, da naš pristop ni najbolj primeren, saj se zaradi režije pri prenosu podatkov med posameznimi enotami aplikacije v oblaku v veliki meri izgubi prednosti vzporednega računanja. Predstavljen pristop takega prenosa tudi ni splošen in ga je potrebno prilagoditi vsaki posamezni aplikaciji.

Izboljšave pri nadaljnjem delu gre iskati predvsem pri večji enostavnosti in boljši podpori za lažji prenos obstoječih aplikacij. Precej je mogoče razširiti tudi podporo trenutnemu formatu vhoda in izhoda, da bo še lažje uporabiti že obstoječe podatke v povezavi s prenesenimi aplikacijami. Največ rezerv se skriva pri boljšem izkoriščanju vseh zmogljivosti za vzporedno računanje. S tem bi aplikacije v oblaku lahko hitreje reševale velike probleme. Čeprav je stopnja paralelizacije odvisna od narave posamezne aplikacije, pa trenutno največjo oviro pri prenosu v oblak s platformo mOSAIC predstavlja slaba performančna zmogljivost komunikacije med posameznimi deli aplikacije v oblaku. Razloga sta naslednja: manj zmogljive povezave z visoko latenco med posameznimi VM in pa ogromni režijski stroški pri prenosu podatkov prek sporočilnih vrst in podatkovne baze. Režijske stroške je mogoče v veliki meri odpraviti z vpeljavo protokola MPI, ki je standard v visokozmogljivih računskih okoljih. Raziskave [13, 43] kažejo, da ima MPI v oblaku sicer določene omejitve, vendar pa bi vgraditev v platformo mOSAIC verjetno prinesla drastične izboljšave. Za še večjo izrabo razpoložljivih zmogljivosti pa bi moral ponudnik IaaS vpeljati visokozmogljivo povezavo z nizko latenco (npr. Infiniband) med posameznimi VM. Platforma, ki bi jo lahko namestili na poljubno oblačno infrastrukturo in bi nudila podporo MPI, bi skupaj z ogrodjem, ki bi zmoglo v oblak enostavno prenesti obstoječe aplikacije, predstavljala idealno okolje za vzporedno računanje v oblaku.

Dodatek A

Razvoj gonilnika za podporo ponudnika IaaS

Ena poglavitnih prednosti platforme mOSAIC je v tem, da je popolnoma prenosljiva na poljubnega ponudnika IaaS. Glavni problem pri prenosljivosti je v tem, da so med ponudniki razlike. Večina med njimi poleg standardnih virov (VM, mreža, prostor) nudi tudi dodatne storitve (npr. požarni zid, izravnalnik bremena, avtomatsko izdelavo varnostne kopije itd.), enako pa velja tudi za dostop do teh storitev - nekatere so dostopne prek spletne storitve, druge prek namenskega API-ja, uporablja se različne protokole (npr. REST, SOAP) itd. Problem prenosljivosti platforma razreši tako, da se pri svojem delovanju opira zgolj na abstraktni model IaaS in ne na posameznega ponudnika IaaS. Abstraktni model temelji na standardu OCCI [31] in omogoča opis vseh potrebnih operacij nad standardnimi viri modela IaaS za nemoteno delovanje platforme. Preslikavo med abstraktnim in konkretnim ponudnikom opravlja gonilnik, imenovan VendorModule. Platforma ima že sedaj vključene gonilnike za vse največje ponudnike IaaS (podpoglavje 3.1), kljub temu pa lahko naletimo na takega, ki ni podprt. Resnično neodvisnost platforme od posameznega ponudnika lahko tako zagotovimo le, če je nepodprtega ponudnika IaaS mogoče enostavno dodati v platformo, saj le s tem lahko svoji aplikaciji zagotovimo najboljšo storitev (npr. glede na ceno, dosegljivost, lokacijo itd.).

V tem dodatku je predstavljen postopek implementacije gonilnika za tiste ponudnike, ki za dostop do svojih storitev uporabljajo programsko opremo OnApp. V nadaljevanju je najprej predstavljena vzpostavitev testnega okolja (A.1), sledi pregled OnApp API (A.2) in nato podrobnosti implementacije (A.3). Na koncu (A.4) je predstavljen tudi primer uporabe razvitega gonilnika na slovenskem ponudniku IaaS - Hostko[1], ki uporablja programsko opremo OnApp.

[1] http://www.hostko.si/

A.1 Vzpostavitev testne infrastrukture

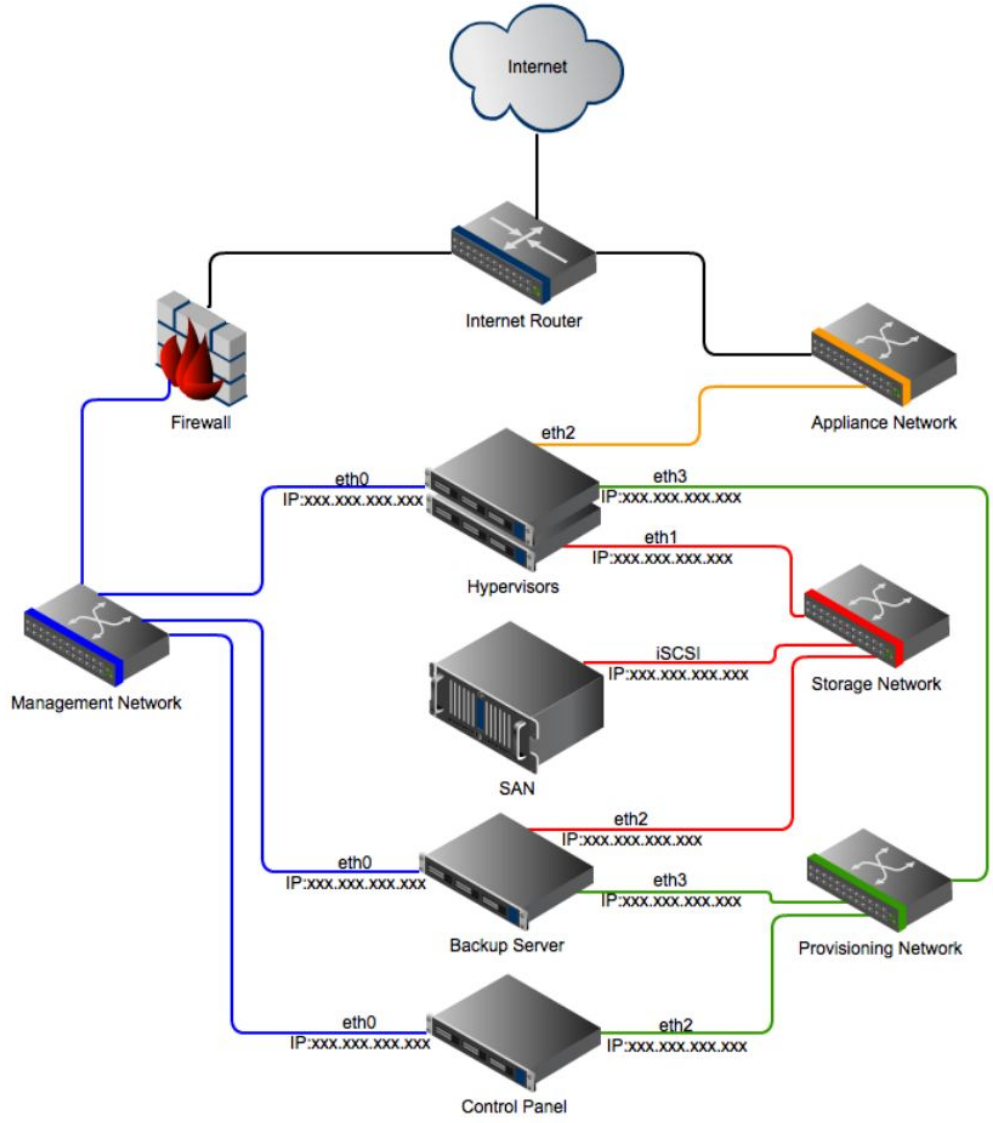
V fazi razvoja gonilnika potrebujemo dostop do ponudnika storitev, da lahko preverimo, ali naš gonilnik deluje pravilno ali ne. Pri tem moramo vsaj za kratek čas najeti izbrane vire, kar je lahko pri nekaterih ponudnikih povezano z veliki stroški. Ti se namreč medsebojno razlikujejo glede na način obračunavanja - lahko uporabljajo model "plačaj, kolikor porabiš", kjer se plačuje le dejansko porabljene vire, lahko pa uporabljajo model "zakupljenih virov", kjer so viri določen čas na voljo ne glede na to, ali se jih nato polno izkoristi ali ne. Za razvoj gonilnika je veliko bolj primeren prvi model plačevanja, saj se izvaja le osnovno testiranje, ki ne zahteva veliko virov. Dodatno se lahko pri manjših ponudnikih pojavi problem zasičenosti razpoložljivih virov. Čeprav je eno izmed pravil računalništva v oblaku izoliranost med uporabniki, torej da dejavnosti enega uporabnika nimajo nobenega vpliva na ostale uporabnike, pa to velja le pri velikih ponudnikih. Pri manjših lahko z napačno (namerno ali nenamerno) dejavnostjo hitro presežemo zmogljivost virov, ki se delijo med uporabnike (npr. pasovno širino povezave z internetom).

V primeru ponudnika Hostko gre za majhnega ponudnika z omejenimi viri, dodatno pa uporablja model "zakupljenih virov" z minimalno časovno enoto enega meseca, kar s stališča testiranja ni najbolj primerno. Zato smo se odločili vzpostaviti lastno testno infrastrukturo.

Celotno potrebno infrastrukturo za namestitev programske opreme OnApp prikazuje slika A.1 in je sestavljena iz treh strežnikov (nadzorni, rezervni in delovni), diskovnega omrežja (Storage Area Network, SAN) in štirih ločenih omrežij (upravljavsko, shranjevalno, rezervno in dostopno).

Zaradi pomanjkanja fizičnih virov smo potrebno infrastrukturo vzpostavili s pomočjo virtualizacije v okolju VMWare ESXi[2] na enem fizičnem strežniku. Rezervni strežnik in rezervno omrežje sta opcijska in ju za potrebe razvoja nismo vzpostavili, za preostala dva strežnika (delovni in nadzorni) pa smo uporabili 2 VM-ja z nameščenim operacijskim sistemom CentOS in ustrezno programsko opremo OnApp. Uporabili smo brezplačno različico programske opreme, ki omogoča enako funkcionalnost kot polna različica, vendar ima postavljeno omejitev na največ 16 jeder za delovne strežnike. Tudi diskovno omrežje SAN smo simulirali, uporabili smo VM z nameščenim operacijskim sistemom OpenFiler[3], ki lahko deluje tudi kot SAN prek protokola iSCSI. Vse komponente smo v ustrezna omrežja povezali prek virtualnih omrežij, ki jih zna okolje VMWare vzpostaviti brez potreb po fizični strojni opremi.

[2] http://www.vmware.com/
[3] http://www.openfiler.com/



Slika A.1: Infrastruktura programske opreme OnApp. Povzeto po [32].

A.2 OnApp API

Nadzorni strežnik OnApp skrbi za vse funkcije celotnega sistema: nadzira delovne strežnike, ustvarja VM-je ter jih zaganja/ugaša na prostih delovnih strežnikih, vzdržuje podatkovno bazo uporabnikov, spremlja njihovo porabo itd. Vse naštete akcije je mogoče izvesti prek OnApp API-ja. Končnim uporabnikom tako omogoča tipične akcije ponudnika IaaS (ustvarjanje VM, zagon VM, pregled porabe itd). Hkrati API uporabnikom z administrativnimi pravicami omogoča tudi dodatne tipične akcije upravljavca ponudnika IaaS (pregled prostih virov, namestitev novih slik operacijskih sistemov za VM, upravljanje podatkovne baze

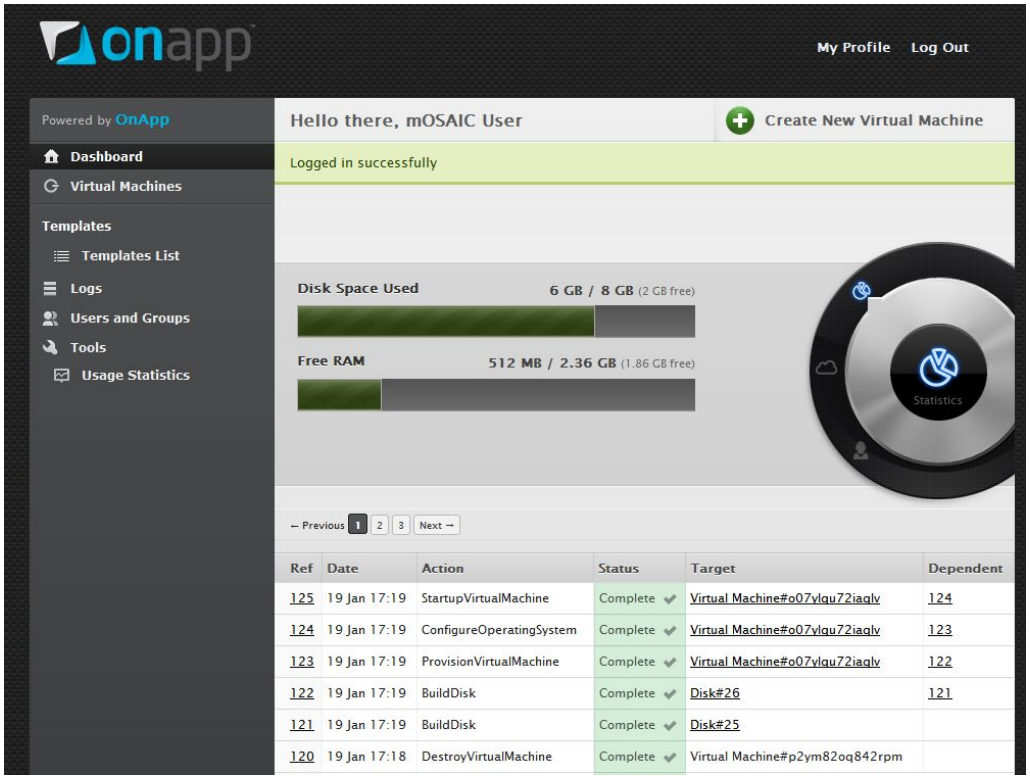
uporabnikov itd.), vendar teh pri razvoju gonilnika nismo potrebovali. API je v obliki spletne storitve dosegljiv prek protokola HTTP na nadzornem strežniku. Temelji na arhitekturi REST in sprejema sporočila v formatu JSON ali formatu XML. Klic ustrezne metode OnApp API-ja poleg imena metode in njenih parametrov zahteva še varnostni ključ (angl. API key), ki preprečuje neavtorizirano rabo API-ja. (Ključ uporabnik pridobi ob registraciji na spletnem vmesniku.) Celoten pregled in dokumentacija OnApp API-ja je na voljo v [33], primer izvedbe klica metode pridobitve seznama razpoložljivih VM in odziva je prikazan na sliki A.2.

```
curl -X GET -u mosaic:?????  -H 'Accept: application/json'
-H 'Content-type: application/json'
http://212.235.189.230/virtual_machines.json
```

```
[{
  "virtual_machine": {
    "admin_note": null,
    "total_disk_size": 6,
    "built": true,
    "template_id": 6,
    "memory": 512,
    "min_disk_size": 5,
    "ip_addresses": [{
      "ip_address": {
         "updated_at": "2013-01-17T23:43:57+01:00",
         "network_address": "10.0.1.0",
         "gateway": "10.0.1.3",
         "address": "10.0.1.102",
         "netmask": "255.255.255.0",
       }
    }],
    "cpus": 1,
    "template_label": "Ubuntu 12.04 x64",
    "user_id": 2,
    "booted": true,
    "identifier": "o07ylqu72iaqlv",
  }
}]
```

Slika A.2: Izvedba klica OnApp API za prikaz seznama razpoložljivih VM in pripadajoči odziv.

Vse funkcije OnApp API-ja so na voljo tudi prek spletnega uporabniškega vmesnika (slika A.3), ki je namenjen ročnemu upravljanju celotnega sistema.



Slika A.3: Spletni uporabniški vmesnik OnApp.

A.3 Implementacija gonilnika VendorModule za OnApp API

Naloga gonilnika je, da zmožnosti konkretnega ponudnika platformi predstavi prek znanega vmesnika, enotnega za vse ponudnike. Vmesnik je v splošnem osnovan na standardu OCCI z nekaterimi razširitvami. Gonilnik mora, da ga zna platforma uporabiti, nuditi naslednjo minimalno funkcionalnost. Vsebovati mora podporo za pridobitev vsaj računskih virov (OCCI tip Compute). Prepoznati mora vsaj štiri lastnosti VM, ki jih predvideva standard (arhitekturo procesorja, število jeder, velikost pomnilnika in hitrost procesorja) ter dodatne tri, ki so lastne platformi (velikost diska, tip operacijskega sistema ter njegova verzija). Nad viri mora znati izvesti vsaj dve akciji, ki ju predvideva standard (zagon, zaustavitev), in dve, ki ju zahteva platforma (prenos datoteke, zagon programa).

Gonilnik se v platformo mOSAIC vključi kot vtičnik (angl. plug-in) in mora biti zato realiziran v programskem jeziku Java in obsegati vsaj dva razreda. Prvi je namenjen opisu virov in mora hraniti vse potrebne podatke o posamezni VM ter razširjati razred ComputeResourceInfo. Implementirali smo ga z javanskim razredom OnAppComputeResourceInfo, ki hrani naslednje podatke o VM: identifikacijska številka, IP-naslov, geslo za oddaljen dostop za uporabnika `root` in številko vrat strežnika SSH.

Drugi razred pa mora razširjati abstraktni razred AbstractVendorModule, ki zahteva implementacijo naslednjih metod:

submitCFP - metoda, ki na podlagi vhodnega CFP, ki vsebuje želene lastnosti VM, izoblikuje ustrezen SLA, ki ga ponudnik podpira.

ensureCredentials - metoda, ki od platforme pridobi poverilnico (angl. credential), ki jo uporabi za delo s ponudnikom.

doCreateResource - metoda, ki ustvari vire na podlagi vhodne SLA.

doReleaseResource - metoda, ki sprosti vire.

doPerformAction - metoda, ki nad določenim virom izvrši želeno akcijo.

getFileUploader - metoda, ki vrne razred, ki omogoča prenos datotek na določen vir tipa Compute.

getProgramRunner - metoda, ki vrne razred, ki omogoča zagon programa na določenem viru tipa Compute.

Razred smo implementirali z razredom OnAppVendorModule, ki vsebuje vseh sedem potrebnih metod.

Prva metoda (submitCFP) lahko vrne vse podprte SLA ponudnika, saj se izbor najbolj ustreznega pridobljenega SLA od vseh gonilnikov opravlja na drugem nivoju (znotraj Agencije za oblake, CA). V primeru OnApp je mogoče prek API-ja definirati skoraj vse želene lastnosti, manjka le podpora izbire poljubne arhitekture (saj lahko ponudnik nudi le eno, ki pa je konstantna) in tipa/verzije operacijskega sistema (izbrati je potrebno enega iz množice možnih). Implementacija te metode v našem gonilniku tako vrne SLA, ki večinoma ustreza CFP, brez komunikacije s ponudnikom.

Druga metoda (ensureCredentials) pri platformi preveri, ali je na voljo poverilnica (sestavlja jo uporabniško ime in ustrezen varnostni ključ) za izbranega ponudnika OnApp. Preveri se le prisotnost obeh delov poverilnice in njuna sintaktična pravilnost (uporabniško ime mora biti v obliki e-naslova, varnostni ključ pa mora obsegati 32 heksadecimalnih znakov). Pravilnosti v smislu avtentikacije se v metodi ne preverja, zato komunikacija s ponudnikom ni potrebna.

Tretja metoda (doCreateResource) pozna le vire tipa Compute, vse ostale tipe zavrne z napako. Ustrezen klic OnApp API-ja za generiranje nove VM je zahtevek HTTP prek metode POST na url `<nadzorni_streznik>/virtual_machines.json` z ustreznimi parametri,

kodiranimi v načinu JSON. Za večino to ni problem, saj obstaja direktna ali zelo enostavna preslikava med parametri, podanimi v SLA (velikost pomnilnika, velikost diska, število jeder, hitrost procesorja, tip/verzija operacijskega sistema), ostale pa je potrebno generirati (ime gostitelja, oznaka VM, velikost izmenjalnega diska). Ob uspešnem klicu kot rezultat dobimo identifikacijsko številko VM (`VM_ID`), njen IP-naslov ter geslo za oddaljen dostop za uporabnika `root`. Iz podatkov ustvarimo primerek razreda OnAppComputeResourceInfo, ki je osnova za vse nadaljnje akcije v povezavi z dodeljeno VM.

Četrta metoda (doReleaseResource) mora sprostiti vire, tako da se storitev neha zaračunavati. V primeru vira tipa Compute je dovolj, da VM ugasnemo, ni pa je potrebno izbrisati. Metoda je realizirana s klicem metode doPerformAction z akcijo zaustavitev.

Peta metoda (doPerformAction) mora poznati akciji zagon in zaustavitev. Ustrezen odziv sprožita sledeča klica OnApp API-ja v obliki zahtevka HTTP prek metode POST:

- zagon
  `<nadzorni_streznik>/virtual_machines/<VM_ID>/startup.json`,
- zaustavitev
  `<nadzorni_streznik>/virtual_machines/<VM_ID>/shutdown.json`.

Oba klica zahtevata le parameter `VM_ID`, ki je na voljo v primerku razreda OnAppComputeResourceInfo, ki je bil generiran v metodi doCreateResource.

Šesta metoda (getFileUploader) v primeru OnApp vrne primerek razreda SecureFileUploaderWithPassword, ki je del platforme. Ta razred zna datoteke prenašati prek protokola SFTP (SSH File Transport Protocol), za delovanje pa potrebuje le parametre: uporabniško ime, geslo, številko IP ter vrata, na katerih strežnik sprejema zahtevke SSH. Vse potrebne podatke imamo na voljo v primerku razreda OnAppComputeResourceInfo.

Sedma metoda (getProgramRunner) je realizirana podobno kot prejšnja. V tem primeru vrne primerek razreda SecureProgramRunnerWithPassword, ki potrebuje enake parametre. Edina razlika je, da se znotraj razreda uporablja protokol SSH, vendar pa tako zahtevke SSH kot SFTP VM sprejema na istih vratih, tako da se tudi ta parameter ne razlikuje.

Vse klice REST smo realizirali z uporabo javanske knjižnice Jersey [4], ki nudi enostavno uporabo in natančno obravnavo morebitnih napak pri komunikaciji.

[4] http://jersey.java.net/

A.4 Uporaba gonilnika

Zgrajeni gonilnik smo med razvijanjem testirali na lokalno postavljeni infrastrukturi OnApp s testnimi orodji, ki so del platforme. Po končanem razvoju smo gonilnik vključili v platformo mOSAIC in jo tudi uspešno zagnali na lokalni infrastrukturi. Na koncu smo izvedli še test s ponudnikom Hostko, kjer smo platformo uspešno namestili na 2 VM s predpisanimi lastnostmi. Postavljeno platformo smo uspešno uporabili v okviru projekta mOSAIC za potrebe testiranja zmogljivosti posameznega ponudnika. Žal pa je ponudnik Hostko predrag in premalo fleksibilen za izvedbo obsežnejših testiranj, tako da smo ga izvzeli kot testno konfiguracijo za izvedbo testov, opisanih v poglavjih 5 in 6.

Dodatek B

Prevajalnik PPMM

Za lažjo uporabo predstavljenega programskega ogrodja za prenos aplikacij Matlab v oblak, smo implementirali preprost prevajalnik za programski jezik Matlab. Zgrajeni prevajalnik razume le majhen del celotne sintakse Matlaba in obsega le sintakso za definiranje matrik in nekatere osnovne operacije nad njimi. Od tu izhaja tudi njegovo ime - Preprost prevajalnik za Matlabove matrike - PPMM.

V nadaljevanju je v podpoglavju B.1 najprej podano splošno ozadje prevajalnika PPMM. V naslednjih podpoglavjih so predstavljeni leksikalni (B.2), sintaksni (B.3) in semantični (B.4) analizator.

B.1 Obseg prevajalnika PPMM

V splošnem je prevajalnik program, ki na vhodu dobi zapis nekega programa v določenem programskem jeziku, njegov izhod pa je prevod istega programa v strojni jezik kakega računalnika [42]. Za naše potrebe je dovolj, da vzamemo le podmnožico celotnega programskega jezika Matlab in ga "razumemo" le v smislu, da znamo iz vhodnega programa razbrati, kakšne matrike definira (velikost matrike in vrednost vseh elementov). Na vhodu našega prevajalnika tako pričakujemo le izvorno kodo, ki definira matrike, za izhod pa dobimo seznam teh matrik.

V programskem jeziku Matlab je mogoče matrike (ali le dele matrike) določiti z določanjem vrednosti posameznih elementov matrike ali s kombiniranjem obstoječih matrik (sestavljanje, filtriranje, izbira podmnožice) ali pa kot rezultat funkcije ali matematične operacije. Pri implementaciji prevajalnika smo se omejili na določanje celih matrik realnih števil z določanjem posameznih elementov in kombiniranjem matrik na osnovi sestavljanja matrik ter podporo operacijama - transponiranje matrike (`A'`) in vektorizacija matrike (`A(:)`). Odločitev za podporo določanja le celih matrik sloni na dejstvu, da se tako precej zmanjša obseg stavkov programskega jezika, ki jih mora poznati naš prevajalnik. S tem izgubimo le na priročnosti,

ne pa na splošnosti, saj posamezne dele matrike še vedno lahko najprej definiramo kot samostojne matrike, ki jih s sestavljanjem združimo v končno matriko. Omejitev na realna števila precej zmanjša kompleksnost prevajalnika, a hkrati zmanjša tudi možen obseg matrik, ki jih lahko opišemo. Omejitev vseeno ni prehuda, saj lahko v večini primerov želeno nepodprto matriko zapišemo kot množico več podprtih matrik (npr. matriko kompleksnih števil razbijemo na dve matriki - prva predstavlja realne dele, druga pa imaginarne dele kompleksnih števil). Preostali izbor je posledica želje, da z našim prevajalnikom ne vplivamo na matematično stabilnost programa, saj vse podprte operacije omogočajo poln prenos informacije iz vhodne datoteke v matriko, ki jo vrnemo kot izhod iz prevajalnika - vse vrednosti se skozi celoten postopek prevajanja obravnavajo kot niz in jih tako ni potrebno nikoli pretvoriti v celo število ali število s plavajočo vejico.

Tipični prevajalnik je običajno sestavljen iz šestih razmeroma samostojnih enot [42], a za naše potrebe so dovolj le prve tri: leksikalni analizator, sintaksni analizator in semantični analizator. Vse tri enote so podrobneje opisane v nadaljevanju.

B.2 Leksikalni analizator

Leksikalni analizator obravnava vhodno datoteko kot niz znakov, ki jih nato združuje v večje pomenske enote, kot so npr. imena, ključne besede, številčne konstante in posebni simboli. Tem enotam pravimo osnovni simboli, v kontekstno neodvisni gramatiki programskega jezika, na podlagi katere deluje sintaksni analizator, pa nastopajo kot končni simboli gramatike [42].

Poglavitna težava pri realizaciji leksikalnega analizatorja za našo izbrano podmnožico programskega jezika Matlab je dejstvo, da se za matematično operacijo transponiranja uporablja enak znak kot za začetek oz. konec niza - znak `'`. Za pravilno prepoznavo moramo tako poleg osnovnega stanja definirati tudi dodatno stanje, ki analizatorju pove, kdaj se lahko pojavi znak v eni ali drugi obliki. Znak `'` lahko predstavlja operacijo transponiranja le neposredno (brez presledkov) po koncu oklepajskega izraza (npr. `([10 20])'`), koncu definicije matrike (npr. `[10 20]'`), imena (npr. `b'`) ali pa predhodnega transponiranja (npr. `[10 20]''`). V vseh ostalih primerih znak `'` obravnavamo kot začetek niza.

Naslednja težava je dejstvo, da v Matlabu znotraj definiranja matrike (med znakom `[` in `]`) presledki niso nepomembni, v različnih situacijah pa lahko pomenijo več različnih stvari. Presledek lahko pomeni razmak med elementi (npr. `a = [10 20]` $\rightarrow$ `[10,20]`), lahko je brez pomena (npr. `a = [ 5 + 2]` $\rightarrow$ `a = [5+2]`) ali pa je njegov namen zavajajoč (npr. `a= [5 +2]` $\rightarrow$ `a = [5,+2]`). Da se izognemo tem situacijam in olajšamo delovanje sintaksnega analizatorja, je najlažje vpeljati pravilo, da vsakič, ko presledek pomeni razmak med

elementi, sintaksnemu analizatorju vrnemo osnovni simbol vejice `,`, ki ima enak pomen. Presledek (ali poljubna kombinacija presledkov in tabulatorjev) pomeni razmak med elementi v primeru, da sledi neposredno številu, imenu, koncu oklepajskega izraza, nizu ali operatorju za transponiranje in se kasneje nadaljuje s številom, imenom, začetkom oklepajskega izraza ali nizom (npr. `[a b]` → `[a,b]` , `[10 (3)]` → `[10,(3)]`, `[5' 'pet']` → `[5','pet']`).

Seznam osnovnih simbolov, ki jih prepozna naš leksikalni analizator, je zbran v tabeli B.1. Regularni izrazi, ki definirajo posamezen osnovni simbol, temeljijo na [18]. Za implementacijo leksikalnega analizatorja smo uporabili orodje JFlex[1]. To na podlagi podanega seznama osnovnih simbolov, regularnih izrazov in stanj ter pravil prehodov med njimi zgradi javanski razred, ki implementira ustrezen leksikalni analizator.

Osnovni simbol	**Regularni izraz**
HSPACE*	`[ \t]`
HSPACES*	`{HSPACE}+`
NEWLINE*	`\n|\r|\f`
LINE	`{NEWLINE}+`
COMMENT*	`%[^\n\r\f]*{NEWLINE}?`
IDENTIFIER	`[a-zA-Z][_a-zA-Z0-9]`
DIGIT*	`[0-9]`
INTEGER	`[-]?{DIGIT}+`
EXPONENT*	`[DdEe][+-]?{DIGIT}+`
MANTISSA*	`({DIGIT}+\.)|({DIGIT}*\.{DIGIT}+)`
FLOATINGPOINT*	`[-]?{MANTISSA}{EXPONENT}?`
DOUBLE	`({INTEGER}{EXPONENT})|{FLOATINGPOINT}`
TEXT	`'[^'\n\f\r]*'`
COMMA	`,`
SEMICOLON	`;`
EQUAL	`=`
LPARENTHESIS	`(`
RPARENTHESIS	`)`
COLON	`:`
TRANSPOSE	`'`
LBRACKET	`[`
RBRACKET	`]`
ERROR*	`vse ostalo`

Tabela B.1: Tabela osnovnih simbolov leksikalnega analizatorja s pripadajočimi regularnimi izrazi. *pomožni regularni izrazi, ki vplivajo na prepoznavo, a jih leksikalni analizator ne vrača sintaksnemu analizatorju

[1] http://jflex.de/

B.3 Sintaksni analizator

Sintakso programskih jezikov opisujemo s kontekstno neodvisnimi gramatikami (KNG) z dodatkom manjšega števila kontekstno odvisnih elementov [42]. Glavni namen sintaksnega analizatorja je določitev drevesa izpeljav (zaporedje produkcij), ki ustreza vhodnemu izvornemu programu. Obstaja več vrst KNG, a za prevajalnike je najprimernejša t. i. družina LALR(1). Njena prednost je v tem, da lahko na podlagi branja vhoda z leve proti desni po le enem osnovnem simbolu naenkrat nedvoumno določimo produkcije, ki so del drevesa izpeljav. Sintaksni analizator na podlagi gramatike LALR(1) za vhod tako uporablja zaporedno pridobljene osnovne simbole leksikalnega analizatorja in zgradi ustrezno drevo izpeljav.

Celotna gramatika LALR(1), ki jo prepozna prevajalnik PPMM, je podana v tabeli B.2 in temelji na [18]. Začetni nekončni simbol je `matlab_input`.

Iz gramatike je razvidno (produkciji 1, 2), da prevajalnik neposredno podpira le matrike števil, prek vmesnih spremenljivk pa tudi matrike nizov. Podpira le prireditvene stavke (angl. assigment), pri čemer je na levi strani podprta le gola spremenljivka (produkciji 1, 2). Tak način ne omogoča definiranja večdimenzijskih matrik (npr. `A(:,:,1)=...`) ali delno prirejanje matrike (npr. `A(1:5) =...`). Način odprave delnega prirejanja je že opisan v uvodu poglavja, večdimenzionalne matrike pa se lahko zapiše kot množico dvodimenzionalnih matrik, ki so za nadaljnjo uporabo na voljo na izhodu prevajalnika.

Za implementacijo sinktaksnega analizatorja smo uporabili orodje CUP[2]. To na podlagi podane gramatike LALR(1) zgradi javanski razred, ki implementira ustrezen sintaksni analizator.

B.4 Semantična analiza

Naslednja stopnja pri prevajanju izvornega programa je semantična analiza, ki sestavi notranjo predstavitev vseh objektov, ki so prisotni v izvornem programu (spremenljivke, tipi itd.). Na podlagi takšne predstavitve opravi tudi preverjanja, ali stavki izvornega programa pravilno uporabljajo operatorje in se tudi sicer podrejajo semantičnim pravilom izvornega jezika [42]. V našem primeru je to tudi končna stopnja prevajalnika, saj za potrebe tega dela potrebujemo le notranjo predstavitev matrik, ki so podane v vhodnem programu.

Naša implementacija semantične analize je napisana v jeziku Java in na podlagi drevesa izpeljav ter izhoda sintaksnega analizatorja vrne seznam vseh matrik. Podatke o posamezni matriki hranimo s pomočjo razreda Matrix, ki lahko hrani vse potrebne podatke - njeno

[2] http://www2.cs.tum.edu/projects/cup/

matlab_input	→	opt_delimiter	
		| opt_delimiter statement_list	
opt_delimiter	→	delimiter | ϵ	
delimiter	→	null_lines | empty_lines	
		| null_lines empty_lines	
null_lines	→	null_line | null_lines null_line	
null_line	→	COMMA | SEMICOLON | empty_lines COMMA	
		| empty_lines SEMICOLON	
empty_lines	→	LINE | empty_lines LINE	
statement_list	→	assignment opt_delimiter	
		| assignment delimiter statement_list	
assignment	→	IDENTIFIER EQUAL expr	(1)
		| IDENTIFIER EQUAL TEXT	(2)
expr	→	INTEGER	(3)
		| DOUBLE	(4)
		| LPARENTHESIS expr RPARENTHESIS	(5)
		| expr TRANSPOSE	(6)
		| IDENTIFIER	(7)
		| IDENTIFIER LPARENTHESIS COLON RPARENTHESIS	(8)
		| matrix	(9)
matrix	→	LBRACKET rows RBRACKET	
rows	→	row	(10)
		| rows SEMICOLON	(11)
		| rows LINE	(12)
		| rows SEMICOLON row	(13)
		| rows LINE row	(14)
		| ϵ	
row	→	expr	(15)
		| row_with_commas	(16)
		| row_with_commas expr	(17)
row_with_commas	→	expr COMMA | row_with_commas expr COMMA	

Tabela B.2: Gramatika sintaksnega analizatorja.

velikost, tip (števila ali znaki) ter vse vrednosti matrike v obliki niza (String) ne glede na tip (breizgubna pretvorba izhoda). Seznam matrik hranimo v slovarju, kjer je ključ ime matrike, vrednost pa primerek razreda Matrix. Končni seznam je tudi rezultat semantične analize oziroma celotnega prevajalnika.

V drevesu izpeljav je vsaka nova matrika določena z eno izmed možnih produkcij nekončnega simbola `assignment` (produkcija 1, 2 v tabeli B.2). V primeru produkcije 2 imamo na voljo že vse podatke, ki definirajo matriko z enim elementom (v Matlabu se skalarne vrednosti obravnavajo kot matrike velikosti 1x1), kar pomeni, da jo lahko zapišemo v slovar. V

primeru produkcije 1 pa moramo najprej ovrednotiti vrednost izraza (nekončni simbol `expr`) ter nato shraniti ustrezne podatke v slovar. Ovrednotenje izraza je v primeru produkcij 3 in 4 enostavno, saj so dani vsi podatki in lahko zgradimo ustrezne matrike velikosti 1x1 z ustrezno vrednostjo. V primeru produkcij 5 in 6 opravimo rekurzivno ovrednotenje s tem, da na koncu produkcije 6 izvedemo še operacijo transponiranja nad rezultatom. V primeru produkcij 7 in 8 vrednosti pridobimo iz že delno zgrajenega slovarja matrik, in če matrika še ni definirana, sporočimo napako. Na koncu produkcije 8 nad pridobljeno matriko izvedemo operacijo vektorizacije (vse elemente zapišemo v matriko oblike 1 x n*m po stolpcih, kjer je n x m velikost osnovne matrike). V primeru produkcije 9 pa moramo matriko zgraditi po elementih. Pri takšni gradnji moramo biti pozorni na to, kdaj dodamo novo vrstico (produkcije 10-14), da ima matrika dovolj elementov ob zaključku vrstice (produkcije 15-17) in da se tip matrike pravilno spreminja glede na dodajanje elementov (v primeru, da je tip matrike ali dodanega elementa znak, potem je tip matrike znak; v vseh drugih primerih je tipa realno število). Posamezni elementi matrike se spet ovrednotijo kot izraz (produkcije 3-9). Semantična analiza omogoča tudi, da so posamezni elementi matrike večji od 1x1, pri tem pa je pomembno le, da ima vsaka vrstica dovolj elementov.

Seznam slik

Seznam tabel

Literatura

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica in ostali. A view of cloud computing. *Communications of the ACM*, zv. 53, št. 4, str. 50–58, 2010.

[2] G. Aversano, M. Rak, U. Villano. The mOSAIC Benchmarking Framework: Development and Execution of Custom Cloud Benchmarks. *Scalable Computing: Practice and Experience*, zv. 14, št. 1, , 2013.

[3] R. Buyya, K. Sukumar. Platforms for building and deploying applications for cloud computing. *Dostopno na arXiv - 1104.4379*, 2011.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, zv. 25, št. 6, str. 599–616, 2009.

[5] H. Choi, S.-H. Lee, D.-I. Park. Biologic Data Analysis Platform Based on the Cloud. *International Journal of Bio-Science and Bio-Technology*, zv. 5, št. 3, , 2013.

[6] R. Choy, A. Edelman. Parallel Matlab: Doing it right. *Proceedings of the IEEE*, zv. 93, št. 2, str. 331–341, 2005.

[7] P. Coffey. *Benchmarking the Amazon Elastic Compute Cloud (EC2).* Doktorska disertacija, Worcester Polytechnic Institute, 2011.

[8] G. Cretella, B. Di Martino, V. Stankovski. Using the mOSAIC's semantic engine to design and develop civil engineering cloud applications. V zborniku *International Conference on Information Integration and Web-based Applications & Services*, str. 378–386. ACM, 2012.

[9] P. Češarek, M. Saje, D. Zupan. Kinematically exact curved and twisted strain-based beam. *International Journal of Solids and Structures*, zv. 49, str. 1802–1817, 2012.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, J. Good. The cost of doing science on the cloud: The Montage example. V zborniku *International Conference for High Performance Computing, Networking, Storage and Analysis*, str. 1–12, 2008.

[11] K. Dejaeger, P. Louis, S. Vanden Broucke, T. Eerola, L. Goedhuys, B. Baesens. Beyond the hype: cloud computing in analytics. *Dostopno na SSRN - 2165720*, 2012.

[12] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, M. Loichate. Building a mosaic of clouds. V zborniku *Euro-Par 2010 Parallel Processing Workshops*, str. 571–578. Springer, 2011.

[13] C. Evangelinos, C. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. *ratio*, zv. 2, št. 2.40, str. 2–34, 2008.

[14] I. Foster. *Designing and building parallel programs*, zv. 95. Addison-Wesley Reading, MA, 1995.

[15] I. Foster, Y. Zhao, I. Raicu, S. Lu. Cloud computing and grid computing 360-degree compared. V zborniku *Grid Computing Environments Workshop*, str. 1–10. IEEE, 2008.

[16] N. J. Higham. *Accuracy and Stability of Numerical Analysis*. Št. 48. Siam, 1996.

[17] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, D. H. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, zv. 22, št. 6, str. 931–945, 2011.

[18] P. G. Joisha, A. Kanhere, P. Banerjee, U. N. Shenoy, A. Choudhary. *The Design and Implementation of a Parser and Scanner for the Matlab Language in the MATCH Compiler*. Tehnično poročilo, Center for Parallel and Distributed Computing, Northwestern University, 1999.

[19] K. Jorissen, F. D. Vila, J. J. Rehr. A high performance scientific cloud computing environment for materials simulations. *Computer Physics Communications*, zv. 183, št. 9, str. 1911–1919, 2012.

[20] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, P. Maechling. An evaluation of the cost and performance of scientific workflows on amazon ec2. *Journal of Grid Computing*, zv. 10, št. 1, str. 5–21, 2012.

[21] J. Kepner. *Parallel Matlab for multicore and multinode computers*. Society for Industrial Mathematics, 2009.

[22] J. Kepner, S. Ahalt. MatlabMPI. *Journal of Parallel and Distributed Computing*, zv. 64, št. 8, str. 997–1005, 2004.

[23] N. Leavitt. Is cloud computing really ready for prime time. *Growth*, zv. 27, št. 5, str. 15–20, 2009.

[24] J. Li, M. Humphrey, D. Agarwal, K. Jackson, C. Ingen, Y. Ryu. eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. V zborniku *IEEE International Symposium on Parallel & Distributed Processing*, str. 1–10. IEEE, 2010.

[25] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, N. Araujo. Performing large science experiments on azure: Pitfalls and solutions. V zborniku *IEEE International Conference on Cloud Computing Technology and Science*, str. 209–217. IEEE, 2010.

[26] MathWorks, Inc., Natick, MA, ZDA. *Parallel Computing with Matlab on Amazon Elastic Compute Cloud*, 2008.

[27] MathWorks, Inc., Natick, MA, ZDA. *Matlab Application Deployment Web Example Guide*, izdaja 2012b, 2012.

[28] MathWorks, Inc., Natick, MA, ZDA. *Matlab Compiler User's Guide*, izdaja 2012b, 2012.

[29] MathWorks, Inc., Natick, MA, ZDA. *Parallel Computing Toolbox User's Guide*, izdaja 2012b, 2012.

[30] P. Mell, T. Grance. The NIST definition of cloud computing (draft). *NIST special publication*, zv. 800, št. 145, str. 7, 2011.

[31] T. Metsch, A. Edmonds in ostali. Open Cloud Computing Interface–Infrastructure. V zborniku *The Open Grid Forum Document Series, Open Cloud Computing Interface Working Group, Muncie*, 2010.

[32] OnApp Limited, Gibraltar. *OnApp Cloud Admin Guide*, izdaja 2.3, 2012.

[33] OnApp Limited, Gibraltar. *OnApp Cloud API Guide*, izdaja 2.3.2, 2012.

[34] S. Panica, M. Neagul, C. Craciun, D. Petcu. Serving Legacy Distributed Applications by a Self-configuring Cloud Processing Platform. V zborniku *International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, zv. 1, str. 139–144, 2011.

[35] D. Petcu, B. Di Martino, S. Venticinque, M. Rak, T. Máhr, G. E. Lopez, F. Brito, R. Cossu, M. Stopar, V. Stankovski in ostali. Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, zv. 2, št. 1, str. 12, 2013.

[36] N. Sharma, M. K. Gobbert. *Performance studies for multithreading in Matlab with usage instructions on hpc.* Tehnično poročilo, University of Maryland, 2009.

[37] M. Snir. *Mpi the Complete Reference: The Mpi Core*, zv. 1. MIT press, 1998.

[38] V. Stankovski, M. Swain, V. Kravtsov, T. Niessen, D. Wegener, J. Kindermann, W. Dubitzky. Grid-enabling data mining applications with DataMiningGrid: An architectural perspective. *Future Generation Computer Systems*, zv. 24, št. 4, str. 259–279, 2008.

[39] G. Strang. *Computational science and engineering.* Wellesley-Cambridge Press Wellesley, 2007.

[40] C. Vecchiola, X. Chu, R. Buyya. Aneka: a software platform for .NET-based cloud computing. *High Speed and Large Scale Scientific Computing*, str. 267–295, 2009.

[41] C. Vecchiola, S. Pandey, R. Buyya. High-performance cloud computing: A view of scientific applications. V zborniku *International Symposium on Pervasive Systems, Algorithms, and Networks*, str. 4–16. IEEE, 2009.

[42] B. Vilfan. *Prevajanje programskih jezikov.* Fakulteta za računalništvo in informatiko, 2004.

[43] Y. Zhai, M. Liu, J. Zhai, X. Ma, W. Chen. Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. V zborniku *State of the Practice Reports*, str. 11. ACM, 2011.

[44] O. C. Zienkiewicz, R. L. Taylor, J. Z. Zhu. *The finite element method: its basis and fundamentals*, zv. 1. Butterworth-Heinemann, 2005.

[45] (2013) mOSAIC: Open-Source API and Platform for Multiple Clouds. Dostopno na: http://www.mosaic-cloud.eu/.