Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Novosel

Vzporedni programski jeziki namesto vzporednih programskih ogrodij

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2019

Avtorske pravice. Rezultati magistrskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov magistrskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

©2019 Rok Novosel

Zahvala

Predvsem bi se rad zahvalil moji punci Tjaši za neomajno podporo tekom izdelave magistrske naloge in tekom celotnega študija.

Poleg tega bi se rad zahvalil celotni družini in vsem prijateljem za vse nasvete in spodbudo med študijem.

Za strokovno podporo in napotke se iskreno zahvaljujem mentorju doc. dr. Boštjanu Slivniku. Za pomoč pri zagotovitvi in uporabi strežniškega sistema na Inštitutu Jožef Štefan se zahvaljujem dr. Matjažu Depolliju.

Rok Novosel, 2019

"Programs must be written for people to read, and only incidentally for machines to execute."

— Harold Abelson in Gerald Sussman

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
HPC	High Performance Computing	Visoko zmogljivo računanje
MPI	Message Passing Interface	Vmesnik za posredovanje sporočil
OpenMP	Open Multi-Processing	Odprtokodno večprocesiranje
OpenCL	Open Computing Language	Odprtokodni računski jezik
CUDA	Compute Unified Device Architecture	Poenotena arhitektura za računanje
GCC	GNU Compiler Collection	Zbirka prevajalnikov GNU
LAPACK	Linear Algebra Package	Paket za linearno algebro
JVM	Java Virtual Machine	Javanski navidezni stroj
CSP	Communicating sequential processes	Komunicirajoči zaporedni procesi
LLVM	Low Level Virtual Machine	Nizkonivojski navidezni stroj
ORB	Orthogonal recursive bisection	Ortogonalna rekurzivna bisekcija
RAM	Random Access Memory	Pomnilnik z naključnim dostopom
DDR	Double Data Rate	Dvojna hitrost prenosa podatkov
SPMD	Single Program Multiple Data	En program mnogo podatkov
KB	Terabyte (2^{10} bytes)	Kilobajt (2^{10} bajtov)
MB	Megabyte (2^{20} bytes)	Megabajt (2^{20} bajtov)
GB	Gigabyte (2^{30} bytes)	Gigabajt (2^{30} bajtov)
TB	Terabyte (2^{40} bytes)	Terabajt (2^{40} bajtov)

Povzetek

Naslov: Vzporedni programski jeziki namesto vzporednih programskih ogrodij

Čeprav vzporedni programski jeziki obstajajo že desetletja, v znanstvenem vzporednem programiranju še vedno prevladujejo Fortran, C in C++ dopolnjeni z vzporednimi programskimi ogrodji MPI, OpenMP, OpenCL in CUDA. V tem delu izvedemo primerjalno študijo novih vzporednih programskih jezikov Chapel in Julia. Glede na vzporedne zmožnosti se oba jezika precej razlikujeta med sabo in v primerjavi s Fortranom in Cjem. Študijo izvedemo na testnih problemih, ki izpostavljajo potrebo po različnih pristopih vzporednega programiranja. Testne probleme implementiramo v Chapelu in Juliji ter Cju, ki ga dopolnimo z MPI-jem in OpenMP-jem. Pokažemo, da tako Chapel kot Julia predstavljata uspešni alternativi Fortranu in C/C++ z vzporednimi programskimi ogrodji. Učinkovitost programerja se znatno izboljša, medtem ko hitrost programov ni bistveno poslabšana.

Ključne besede

programski jeziki, vzporedno programiranje, vzporedni programski jeziki

Abstract

Title: Parallel programming languages instead of parallel programming frameworks

Although parallel programming languages have existed for decades, (scientific) parallel programming is still dominated by Fortran and C/C++ augmented with parallel programming frameworks, e.g., MPI, OpenMP, OpenCL and CUDA. We perform a comparative study of Chapel and Julia, two languages quite different from one another as well as from Fortran and C, in regard to parallel programming on distributed and shared memory computers. The study is carried out using test cases that expose the need for different approaches to parallel programming. Test cases are implemented in Chapel and Julia, and in C augmented with MPI and OpenMP. It is shown that both languages, Chapel and Julia, represent a viable alternative to Fortran and C/C++ augmented with parallel programming frameworks: the programmer's efficiency is considerably improved while the speed of programs is not significantly affected.

Keywords

programming languages, parallel programming, parallel programming languages

Poglavje 1

Uvod

Danes so vsi računalniki vzporedni. Vzporedno računanje je primarni način, s katerim se proizvajalci procesorjev soočajo s fizičnimi omejitvami procesorske tehnologije, ki temelji na tranzistorjih. Več procesorjev ali jeder združijo na enem integriranem vezju, kar prinaša boljšo zmogljivost in energijsko učinkovitost v primerjavi z enim procesorjem [31]. Vsi sodobni računalniški sistemi podpirajo strojno vzporedno izvajanje na vsaj en način. To dosežejo predvsem z vektorskimi ukazi, večjedrnimi procesorji, več procesorji in grafičnimi karticami.

Težava pri vzporednem računanju je, da programska oprema ne bo več avtomatsko deležna pohitritev zaradi napredka v procesorski tehnologiji kot je bila v preteklosti. Moorov zakon [34] nam je 40 let zagotavljal, da se bo procesorska moč vsaki dve leti podvojila. Vendar pa so proizvajalci procesorjev naleteli na omejitve kot so odvajanje toplote, naraščajoča razlika med hitrostjo procesorja in pomnilnika ter omejitve pri količini ukazov, ki jih procesor lahko avtomatsko izvede vzporedno. To je znano tudi kot konec "zastonj kosila" [49]. Nove napredke lahko izkoristimo samo z eksplicitnim vzporednim programiranjem. Avtomatski pristopi, ki poskušajo pretvoriti zaporedne programe v vzporedne, trenutno ne znajo obravnavati temeljnih sprememb v strukturi algoritmov, ki so potrebni za učinkovito pretvorbo.

Izdelovalci procesorskih čipov so zaradi omenjenih omejitev prisiljeni opti-

mizirati učinkovitost procesorja na drugačne načine. To vključuje povečevanje procesorskega predpomnilnika, organiziranje ukazov v cevovod, napoved vej izvajanja (angl. branch prediction) in preurejanje ukazov za izvajanje izven zaporedja (angl. out-of-order execution). Te optimizacije lahko izkoristijo zlonamerni programi, ki jih uporabijo za dostop do podatkov ostalih programov. Primer zlonamernih napadov, ki izkoriščata napoved vej izvajanja in cevovod, sta Spectre [21] in Meltdown [29]. Popravki, ki popravijo ranljivosti, znatno upočasnijo delovanje procesorja. To je še dodaten razlog, zakaj se morajo proizvajalci računalniških sistemov in njihovi programerji v prihodnosti osredotočati na strojno in programsko podporo za vzporedno izvajanje.

Vzporedni programski jeziki morajo biti sposobni izkoristiti vso vzporednost, ki je na voljo v programski in strojni opremi. Algoritmi vsebujejo vzporednost na različnih ravneh. Vzporednost lahko uporabimo pri klicu funkcij, iteracijah zank in tudi pri posameznih izrazih. Sodobna strojna oprema omogoča vzporednost preko več povezanih računalnikov, procesorjev in vektorskih operacij v procesorskih ukazih. Večina vzporednih programskih jezikov implementira samo podmnožico omenjenih vzporednih zmožnosti programske in strojne opreme. Če bi hoteli izkoristiti celotno zmožnost večračunalniškega sistema, bi za vsako raven vzporednosti morali uporabiti različno vzporedno ogrodje. MPI [18] bi uporabili, da porazdelimo programe po več računalnikih in nato OpenMP [15], da izkoristimo jedra procesorjev na vsakem računalniku. Poleg tega bi radi izkoristili tudi grafične kartice, ki so vse bolj prisotne pri visoko zmogljivem računanju. Za to bi zopet uporabili eno izmed ogrodij za pisanje vzporednih programov na grafičnih karticah, kot sta OpenCL [48] ali CUDA [36]. V nasprotju s tem si želimo vzporedni programski jezik, ki bo omogočal izražanje vzporednosti z enotnim naborom jezikovnih konceptov.

V nalogi bomo primerjali dva sodobna programska jezika, Chapel in Julia. Izbrana vzporedna programska jezika bomo ovrednotili kot ustrezni alternativi obstoječim ogrodjem za vzporedno programiranje. Predvsem nas

bosta zanimali programerska izkušnja in učinkovitost izbranih jezikov. Pod programersko izkušnjo štejemo enostavnost pisanja učinkovitih programov, uporabo vzporednih funkcionalnosti jezika in dostop do funkcionalnosti sodobnih programskih jezikov (npr. sklepanje o tipih, sporočanje napak in zunanje knjižnice).

V razdelku 2 bomo naredili pregled primerjav vzporednih programskih jezikov in vzporednih programskih jezikov na splošno. Predvsem nas bodo zanimali predhodniki Chapela, Julije in ostalih sodobnih programskih jezikov, ki vključujejo zmožnosti vzporednega programiranja. V razdelku 3 bomo opisali in naredili primerjavo vzporednih zmožnosti Chapela in Julije. Jezika bomo ovrednotili tako, da implementiramo vzporedne algoritme treh problemov. Za vsak problem bomo opisali zaporedni algoritem, algoritem za skupni pomnilnik in algoritem za porazdeljeni pomnilnik. Za primerjavo bomo vzeli programski jezik C s primernim vzporednim ogrodjem. V razdelku 4 bomo predstavili izbrane probleme in njihove algoritme. V razdelku 5 bomo opisali implementacije, čase izvajanja in programersko izkušnjo. V zadnjem razdelku 6 bomo analizirali rezultate. Pričakujemo, da se bomo približali odgovoru na vprašanje, v katerem jeziku ali s katerim ogrodjem naj v prihodnje pišemo vzporedne programe za različne računsko intenzivne probleme.

Poglavje 2

Pregled področja

2.1 Primerjave vzporednih programskih jezikov

Vzporedno programiranje je velikokrat videno kot pretežko opravilo za navadne programerje. Zato je pomembno raziskati ali novi vzporedni programski jeziki lahko zapolnejo vrzeli med začetniki in strokovnjaki za vzporedno programiranje.

V [17] je bila izvedena primerjalna študija osmih vzporednih programskih jezikov na podlagi rešitev štirih problemov. Izbrani so bili tipični problemi iz področja linearne algebre, kemije in vzporednega programiranja. Vsakemu jeziku je namenjeno poglavje, kjer je opisana njegova filozofija, sintaksa, semantika in rešitve problemov. Poglavja so napisana s strani priznanih strokovnjakov na področju posameznega programskega jezika. Podoben pristop so imeli v [6], kjer na dveh problemih primerjajo pet vzporednih programskih jezikov. V obeh člankih manjka primerjava s programskima jezikoma C in Fortran, ki sta največja tekmeca vzporednim programskim jezikom. Iz skupno 13 opisanih jezikov sta danes v večji uporabi samo še Haskell in Ada.

V [35] so se odločili raziskati in ovrednotiti vrzel med začetniki in strokovnjaki za vzporedno programiranje. Vrzel so opredelili kot razliko med napisanimi vrsticami kode, učinkovitostjo implementacij in stroški popravkov, ki

so potrebni, da se koda začetnika popravi v skladu s standardi strokovnjaka. Za ovrednotenje so izbrali štiri vzporedne programske jezike: Chapel, Go, Cilk in TBB (Threading Building Blocks). V študiji so strokovnjaki za vsak jezik (najpogosteje razvijalci jezika) pregledali šest implementacij vzporednih programov za skupni pomnilnik. Implementirali so jih izkušeni programerji, ki niso imeli predhodnega znanja o vzporednem programiranju. Strokovnjaki so implementacije pregledali in podali svoje komentarje. Programerji so implementacije popravili in jih znova oddali v pregled. Na koncu so imeli 24 originalnih implementacij in 24 implementacij po komentarjih strokovnjakov. Avtorji so ugotovili, da komentarji strokovnjakov le zmerno vplivajo na štiri metrike, ki so si jih izbrali. V povprečju se je število vrstic izvorne kode zmanjšalo za 1.6% (standardni odklon 13.9%), čas izvajanja se je zmanjšal za 18.1% (standardni odklon 38.3%), pohitritev se je povečala za 1.5 (standardni odklon 1.1). Čas potreben za popravilo komentarjev strokovnjakov je bil 29.9% časa originalne implementacije (standardni odklon 24.4%). To nakazuje na dejstvo, da so izbrani vzporedni programski jeziki dobro zasnovani in primerni za novince pri vzporednem programiranju. Pri tem se avtorji zavedajo, da njihovi rezultati ne bodo nujno veljali na večjih vzporednih aplikacijah. V raziskavi je imela najdaljša implementacija 345 vrstic izvorne kode. Poleg tega bi bilo potrebno izvesti raziskavo vzporednih programskih jezikov, ki podpirajo programiranje na porazdeljenem pomnilniku.

2.2 Vzporedni programski jeziki

Poleg primerjav vzporednih programski jezikov nas zanimajo tudi jeziki sami. Pregledali bomo jezike, ki so služili kot osnova za jezike, ki jih bomo obravnavali in jezike, ki se uveljavljajo kot vzporedni programski jeziki prihodnosti.

2.2.1 ZPL

ZPL je vzporedni programski jezik, ki je bil osredotočen na podatkovno vzporednost na osnovi tabel [12]. ZPL ne vsebuje eksplicitnih vzporednih kon-

struktov. Prevajalnik najde vso implicitno vzporednost, izvede komunikacijo med procesorji, doda potrebno sinhronizacijo in izvede vse možne optimizacije nad vzporednimi in skalarnimi operacijami. ZPL je vpeljal koncept regije, ki predstavlja množico indeksov v koordinatnem prostoru poljubnih dimenzij. Regije se uporabljajo za ustvarjanje vzporednih tabel, ki so primarna enota vzporednega računanja. Indeksi znotraj regije so porazdeljeni med procese, ki izvajajo ZPL program. To nadzoruje porazdelitev elementov tabele in porazdelitev operacij nad elementi. Regije so prevzeli tudi v programskem jeziku Chapel in jih preimenovali v domene. Razvoj ZPL-ja se je ustavil leta 2005, danes pa ga je v veliki meri nadomestil Chapel.

2.2.2 Fortress

Fortress je vzporedni programski jezik, ki so ga razvijali pri podjetju Sun Microsystems med leti 2005 in 2012 [1]. Podobno kot pri ZPL-ju, so tudi pri Fortressu uporabili implicitno vzporednost. Prevajalnik za Fortress jo lahko uporabi pri funkcijskih argumentih in elementih terk, ki so po definiciji neodvisni in zato idealni za vzporedno ovrednotenje. Implicitna vzporednost uporablja medsebojno krajo nalog med procesorji (angl. work stealing). Kraja nalog je način razporejanja dela med procesorje v sistemih s skupnim pomnilnikom. Vsak procesor ima zaporedje nalog, ki jih mora opraviti. Tekom izvajanja naloge lahko ta ustvari nove naloge, ki so dodane na konec zaporedja nalog, ki jih trenutni procesor izvaja. Ko procesorju zmanjka nalog, pogleda zaporedja nalog ostalih procesorjev in enemu izmed njih "ukrade" določeno število nalog.

Fortress je poleg vzporednosti ponujal še sklepanje o tipih, avtomatsko upravljanje s pomnilnikom in matematično sintakso. Zasnovan je bil tako, da izgleda čim bolj podobno izvedljivi psevdokodi [46]. Po Fortressu so se zgledovali tudi razvijalci Julije in uporabili vse omenjene zmožnosti Fortressa razen implicitne vzporednosti. Prva implementacija specifikacije Fortressa 1.0 je tekla na JVM [28]. Nadaljna cilja sta bila, da se Fortress programi direktno prevedejo v Java bytecode in da podpirajo sisteme s porazdeljenim

pomnilnikom.

2.2.3 Rust

Rust je sistemski programski jezik, ki ga razvijajo pri podjetju Mozilla. Služi kot osnova za njihov spletni brskalnik Firefox. Sintaktično je podoben jezikoma C in C++, a veliko vplivov črpa iz funkcijskih jezikov iz družine ML (SML in OCaml) in Haskell [26]. Obljublja lažje upravljanje in boljšo varnost pomnilnika, brezskrbno vzporedno programiranje in visoko učinkovitost. Rust ne uporablja avtomatskega upravljanja s pomnilnikom (angl. garbage collection) in ne prelaga te odgovornosti na programerja z ročnim dodeljevanjem in sproščanjem pomnilnika. Namesto tega uvede koncept "lastništva" (angl. ownership) za vrednosti v jeziku [32]. Pravila lastništva so sledeča:

- vsaka vrednost v Rustu ima spremenljivko, ki je njena lastnica,
- vrednost ima lahko samo eno lastnico,
- ko gre lastnica izven področja uporabe (angl. scope) je vrednost avtomatsko sproščena.

Če želimo uporabiti neko vrednost izven njenega področja uporabe, si jo lahko izposodimo (angl. borrowing). Rust statično preveri, da izposoje ne živijo dlje kot vrednosti, ki jih izposojamo. Z lastništvom in izposojanjem vrednosti nam prevajalnik zagotavlja konsistentne dostope do pomnilnika. Za vzporednost to pomeni, da lahko brez skrbi programiramo v več paradigmah (s pošiljanjem vrednosti preko kanalov, z deljenim stanjem ali s ključavnicami), in da nam prevajalnik pomaga, da se izognemo običajnim pastem. Ko pošiljamo vrednost preko kanala, prenesemo tudi njeno lastništvo prejemniku te vrednosti. Zato lahko pošiljamo vrednosti med nitmi brez skrbi, da bi prišlo do tekmovanja za podatke (angl. data race). Ključavnice v Rustu vedo, katere vrednosti ščitijo. Rust zagotavlja, da lahko dostopamo do teh vrednosti samo, če si trenutno lastimo ključavnico. Rust podpira

vzporedno programiranje samo za skupni pomnilnik in nima vgrajene podpore za porazdeljeni pomnilnik.

Vzporedne zmožnosti Rusta so uporabili, da so pri Mozilli izdelali motor brskalnika imenovan Servo [2]. Z uporabo Rusta so v kratkem obdobju in z majhnim številom ljudi uspeli ustvariti motor brskalnika, ki lahko tekmuje z drugimi sodobnimi brskalniki v zmogljivosti in funkcionalnosti.

2.2.4 Go

Go je programski jezik zasnovan pri podjetju Google. Eden izmed njegovih snovalcev je bil Ken Thompson, ki je soustvaril operacijski sistem Unix in programski jezik B - direktni predhodnik programskega jezika C. Zato je tudi Go sintaktično najbolj podoben jeziku C. Go vključuje avtomatsko upravljanje s pomnilnikom in vzporedno programiranje z uporabo kanalov. Kanali so izpeljani iz Hoarejevih komunicirajočih zaporednih procesov (CSP) [20]. Primarna enota vzporednosti v Goju je gorutina (angl. goroutine), ki je tip izjemno lahke niti. Trenutne implementacije Goja multipleksirajo gorutine na manjše število niti operacijskega sistema. To je primer hibridnega ali M:N modela nitenja [44]. Gojeve vzporedne zmožnosti niso direktno namenjene za visoko zmogljivo računanje. Zato so bile izvedene različne študije o učinkovitosti vzporednega programiranja za skupni pomnilnik z Gojem.

V [50] so ovrednotili jezik Go na vzporedni integraciji in vzporednem dinamičnem programiranju. Prikazali so, da lahko uporabljamo gorutine v računsko intenzivnih problemih in dosežemo skoraj idealno pohitritev. Go so uporabili kot enega izmed programskih jezikov v raziskavi vrzeli med začetniki in strokovnjaki vzporednega programiranja [35].

Zaradi majhne standardne knjižnice je velikost implementacije v jeziku Go velikokrat narasla po komentarjih strokovnjakov. V eni izmed implemetacij je začetnik uporabil funkcijo za urejanje iz standardne knjižnice, ki pa ni imela dobre vzporedne učinkovitosti. Zato je strokovnjak predlagal, da jo zamenja z lastno implementacijo vzporednega urejanja z združevanjem (angl. merge sort). Komentarji strokovnjakov so najbolj vplivali na izvajalni čas.

Začetniki so uporabili deli-in-vladaj vzorec in za vsako nalogo izvedli eno gorutino. Tak pristop ni bil učinkovit zaradi stroškov ustvarjanja pretiranega števila gorutin [35]. Strokovnjaki so predlagali, da ustvarijo eno gorutino za vsako jedro procesorja in jim porazdelijo naloge. Go se trenutno osredotoča na učinkovito programiranje skupnega pomnilnika in ne podpira vzporednega programiranja za sisteme s porazdeljenim pomnilnikom.

Poglavje 3

Primerjava

V nalogi bomo primerjali programska jezika Chapel in Julia. Glavni kriteriji za izbiro so bili aktivnost razvoja, število uporabnikov in razpoložljiva dokumentacija. Oba jezika se aktivno razvijata na spletni strani GitHub in tekom izdelave magistrske naloge sta bili že izdani novi verziji jezikov. Poleg Chapela in Julije smo za primerjavo obravnavali še programski jezik X10. Vendar kot so že ugotovili v [40], jezika aktivno ne razvijajo in je slabo dokumentiran. Zaradi tega smo ga izpustili iz primerjave in se osredotočili na Chapel in Julijo. V nadaljevanju bomo oba podrobno opisali in predstavili njuno sintakso in semantiko vzporednih zmožnosti na dveh kratkih primerih.

3.1 Chapel

Chapel je vzporedni programski jezik narejen za produktivno vzporedno programiranje na področju visoko zmogljivega računanja [7, 11]. Njegov razvoj se je začel leta 2002 in ga vodi podjetje Cray, ki je eno izmed vodilnih podjetij v svetu superračunalništva. Razvijalci Chapela si prizadevajo ustvariti jezik, ki se programira preprosto kot Python, dosega učinkovitost Fortrana in skalabilnost MPI-ja. Podpirati želijo splošno vzporedno programiranje v nasprotju z jeziki kot je ZPL, ki so osredotočeni na eno samo vrsto vzporednosti. Zavedajo se, da morajo podpirati široko paleto vzporednih računskih

zmožnosti, če hočejo, da bo jezik sprejet na raznolikem področju kot je visoko zmogljivo računanje. Zato je bil Chapel zasnovan, da podpira podatkovno vzporednost in vzporednost na podlagi nalog (angl. task parallelism).

Posebno pozornost so namenili novim programerjem, ki so vajeni programiranja v jezikih kot so Python, Java in Matlab. Na področju visoko zmogljivega računanja pa programerji delajo z jeziki C, C++ in Fortran skupaj z ogrodjema OpenMP in MPI. S temi jeziki in ogrodji novi programerji nimajo dovolj izkušenj, da bi lahko bili produktivni. Del Chapelovega cilja je izboljšati produktivnost novih programerjev pri vzporednem programiranju. Zato Chapel vključuje sklepanje o tipih, podporo za objektno usmerjeno programiranje in podporo za popolnoma imperativni stil programiranja za tiste, ki prisegajo na C in Fortran. Glavna karakteristika Chapela je, da omogoča globalni pogled na računanje, s katerim lahko programerji izražajo algoritme in podatkovne strukture kot celoto. Program se izvaja na enem logičnem procesu, kjer programer sam uvaja potrebno vzporednost preko jezikovnih konstruktov. To je v nasprotju s SPMD vzporednim modelom, kjer programerji napišejo svoj program ob predpostavki, da bo več kopij njihovih programov teklo v vzporednih procesih.

Chapel podpira globalni pogled vzporednosti z uporabo domen. Domena je poimenovana prvo-razredna vrednost, ki opisuje množico indeksov. Uporablja se za določanje velikosti in oblike tabel. Chapelove domene so evolucija ZPL-jevih regij. Chapel podpira aritmetične domene, nedoločene domene in anonimne domene. Aritmetične domene predstavljajo klasično množico številskih indeksov. Lahko se dinamično razširjajo in uporabimo jih lahko kot goste (angl. dense) ali redke (angl. sparse). Nedoločene domene se uporabljajo za implementacijo množic ali asociativnih tabel, ker so indeksi lahko poljubnega tipa. Anonimne domene vsebujejo indekse, ki nimajo relacij med sabo in se največkrat uporabijo za implementacijo podatkovnih struktur grafov. Indekse domene lahko porazdelimo med več računskimi mesti (angl. locale). Računsko mesto v Chapelu se nanaša na eno enoto vzporedne arhitekture, ki je sposobno izvajati programe in ima enoten dostop do pomnil-

nika. Najpogosteje je to eno vozlišče v večračunalniških sistemih. Chapel ima vgrajen tip `locale` in vgrajeno tabelo `Locales` z vsemi prisotnimi računskimi mesti v sistemu. Domeno porazdelimo med izbrana računska mesta in nato lahko vsako izmed poddomen vzporedno obdelamo.

3.2 Julia

Julia je bila zasnovana kot visokonivojski dinamični programski jezik za numerično računanje [9]. Originalni jezik za numerično računanje je bil Fortran, ki je izšel leta 1957. Od takrat se je ekosistem znanstvenega računanja drastično spremenil. Dinamični programski jeziki kot so Python, R in Matlab prevladujejo med raziskovalci. V teh jezikih lahko pišejo preprosto visokonivojsko kodo brez skrbi glede tipov spremenljivk. Kljub temu C in Fortran še vedno postavljata standard glede učinkovitosti. Kolikor programerji dinamičnih programskih jezikov izgubijo pri učinkovitosti, programerji jezikov C in Fortran izgubijo pri produktivnosti. Ta problem so dosedaj reševali tako, da se programi še naprej pišejo v dinamični visokonivojski kodi, računsko intenzivne funkcije pa so skrite v C knjižnicah (npr. Python in NumPy [39]).

Julia je ta problem poimenovala "problem dveh jezikov" in si ga prizadeva rešiti. Ključna filozofija Julije je, da mora biti osnovna funkcionalnost hitra. Sem spadajo zanke `for`, rekurzija, operacije s plavajočo vejico in klicanje C funkcij. Julia omogoča fleksibilen sistem tipov z neobveznimi oznakami. Na podlagi tipov lahko implementira pomemben koncept za učinkovitost imenovan večkratno razpošiljanje (angl. multiple dispatch). To je izbira funkcije na podlagi tipov argumentov pri klicu funkcije. Z večkratnim razpošiljanjem lahko napišemo specializirane funkcije za tipe, ki jih uporabljamo v našem programu.

Kar ločuje Julijo od večine visoko zmogljivih programskih jezikov, je dejstvo, da programov ni potrebno predhodno prevajati. Uporablja sprotno prevajanje (angl. just-in-time compilation) s pomočjo prevajalniškega ogrodja LLVM [27], ki vse funkcije neposredno prevede v strojno kodo tik preden se

prvič uporabijo.

Julia podpira vzporednost na več nivojih. Z uporabo makroja `@simd` lahko označimo del kode, za katerega pričakujemo, da bo prevajalnik odkril vzporednost na nivoju posameznih procesorskih ukazov. Za skupni pomnilnik in večnitenje vključuje eksperimentalno podporo z `Threads` modulom in makrojem `@threads`. Pri porazdeljenem pomnilniku so se odločili za implementacijo komunicirajočih zaporednih procesov. V Juliji se imenujejo oddaljeni kanali (angl. remote channels). Vsi kanali vsebujejo tip spremenljivke, ki ga pošiljajo ter kapaciteto kanala. Julia avtomatsko poskrbi za serializacijo podatkov pri pošiljatelju in deserializacijo pri prejemniku. Tako kot Chapel tudi Julia teče v enem logičnem procesu na začetku in programer sam dodaja vzporednost preko jezikovnih konstruktov.

Kot primer uporabe lahko navedemo projekt Celeste [41], pri katerem so katalogirali vidno vesolje iz 55 TB slikovnih podatkov. Uporabili so 1.3 milijona niti na 650.000 Intel Xeon Phi jedrih in dosegli 1.54×10^{15} operacij s plavajočo vejico na sekundo.

3.3 Primeri izvorne kode Chapela in Julije

Za opis sintakse in semantike vzporednih zmožnosti obeh jezikov smo si izbrali dva primera.

3.3.1 Izračun števila π z metodo Monte Carlo

Na primeru izračuna števila π z metodo Monte Carlo bomo prikazali vzporedne zmožnosti jezikov za skupni pomnilnik. Metode Monte Carlo so širok razred algoritmov, ki temeljijo na večkratnih naključnih vzorčenjih za pridobivanje numeričnih rezultatov [42]. Taki algoritmi so idealni za paralelizacijo, ker so posamezna vzorčenja neodvisna med sabo.

Za izračun števila π z metodo Monte Carlo potrebujemo krog s polmerom $r = 0.5$, ki je vrisan v kvadrat s stranico dolžine 1. Površina kroga je $\pi r^2 = \frac{\pi}{4}$ in površina kvadrata je 1. Razmerje med površino kroga in površino kvadrata

je enako $\frac{\pi}{4}$. Da izračunamo π, generiramo veliko število enakomerno porazdeljenih točk v kvadratu in sledimo številu točk, ki padejo znotraj kroga. Razmerje med številom točk znotraj kroga N_{krog} in številom točk znotraj kvadrata $N_{kvadrat}$ vzamemo kot približek razmerja med površinami. Tako dobimo povezavo med številom π in razmerjem med številom točk, ki je izražena s formulo

$$\frac{\pi}{4} \approx \frac{N_{krog}}{N_{kvadrat}}$$

oziroma

$$\pi \approx 4\frac{N_{krog}}{N_{kvadrat}}.$$

Večje kot je število vseh točk, bolj bo natančen približek števila π. Na začetku algoritma določimo število točk in jih nato razdelimo med niti. Vsaka nit izračuna svoj približek in vse približke na koncu združimo v končnega.

Vzporedni izračun števila π z metodo Monte Carlo v jeziku Chapel je v izvorni kodi 3.1. Za vzporedno iteracijo v Chapelu uporabimo zanko `forall`, ki med trenutno razpoložljive niti razdeli iteracije zanke. Spremenljivka `here` je posebna spremenljivka tipa `locale` in opisuje trenutno fizično vozlišče. Nas zanima `here.maxTaskPar`, ki nam pove maksimalno število niti na voljo na trenutnem vozlišču. Zanka `forall` s posebno ključno besedo `with` sprejme seznam navodil za reduciranje spremenljivk v zanki. Vsaka nit dobi kopijo spremenljivke `result`, ki jih po koncu zanke seštejemo in shranimo nazaj v globalno spremenljivko. V OpenMP-ju bi tako zanko zapisali kot: `#pragma omp parallel for reduction(+:result)`.

Vzporedni izračun števila π z metodo Monte Carlo v jeziku Julia je v izvorni kodi 3.2. Pri Juliji uporabimo makro `@threads` za implementacijo na skupnem pomnilniku. Makro vzame telo zanke in statično razdeli iteracije zanke med niti, ki so na voljo. Makro `@threads` nima možnosti podati željene redukcije spremenljivk, zato jo moramo izračunati sami. Rezultat vsake niti shranimo v tabelo, ki jo na koncu seštejemo. Dve pomožni funkciji za delo z nitmi sta `threadid` in `nthreads`. Prva vrne identifikator niti, druga pa vrne skupno število niti.

Izvorna koda 3.1: Vzporedni izračun števila π z metodo Monte Carlo v jeziku Chapel

```
use Random;
proc computePi(n: int) {
    var withinCircle = 0;
    var randStream = new owned RandomStream(real);
    for _i in 0..#n {
        var x = randStream.getNext() * 2 - 1;
        var y = randStream.getNext() * 2 - 1;
        var r2 = x * x + y * y;
        if r2 < 1.0 {
            withinCircle += 1;
        }
    }
    return withinCircle / n:real * 4.0;
}
param N = 100000;
var result = 0.0;
forall _i in 0..#here.maxTaskPar with (+ reduce result) {
    result = computePi(Math.ceil(N / here.maxTaskPar):int);
}
writeln(result / here.maxTaskPar);
```

Izvorna koda 3.2: Vzporedni izračun števila π z metodo Monte Carlo v jeziku Julia

```
using Base.Threads
function compute_pi(n::Int)
    within_circle = 0
    for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x * x + y * y
        if r2 < 1.0
            within_circle += 1
        end
    end
    within_circle / n * 4.0
end
const N = 10000000
results = zeros(nthreads())
@threads for _ in 1:nthreads()
    results[threadid()] = compute_pi(ceil(Int, N / nthreads()))
end
println(sum(results) / nthreads())
```

3.3.2 Prevajanje toplote v eni dimenziji

Za prikaz vzporednih zmožnosti za porazdeljeni pomnilnik bomo simulirali prevajanje toplote v eni dimenziji. Namen analize prevajanja toplote je določiti porazdelitev temperature po notranjosti telesa ob znani porazdelitvi temperature po zunanjosti telesa in njeno časovno spreminjanje. Zvezna enačba za prevajanje toplote je

$$\frac{\partial T}{\partial t} = \lambda \frac{\partial^2 T}{\partial x^2},$$

kjer je T temperatura, t čas, x razdalja in λ toplotna prevodnost.

Preden jo lahko uporabimo za implementacijo simulacije prevajanja toplote, jo moramo diskretizirati. Enačbo diskretiziramo z uporabo metode končnih razlik tako, da jo prepišemo v enačbo

$$T_i^{n+1} = T_i^n + \lambda \frac{\Delta t}{\Delta x^2}(T_{i+1}^n - 2T_i^n + T_{i-1}^n).$$

Osnovna ideja metode končnih razlik je, da aproksimiramo odvode funkcijskih vrednosti, ki nastopajo v diferencialni enačbi in robnih pogojih. S tem dobimo eksplicitni predpis za posodabljanje temperature T v vsaki diskretni točki i ob času n. To je znano tudi kot eksplicitna Eulerjeva metoda. Zaradi lažjega izračuna dodamo prazni robni točki (angl. ghost cells) na začetek in konec domene.

Za simulacijo prevajanja toplote na porazdeljenem pomnilniku domeno enakomerno porazdelimo med P procesov. Vsak proces lahko skoraj neodvisno izračuna porazdelitev temperature na svojem delu domene. Izjemoma mora vsak proces prejeti kopijo robnih vrednosti predhodnega in naslednjega procesa. Vsak proces mora poslati svojo prvo vrednost predhodnemu sosedu in svojo zadnjo vrednost naslednjemu sosedu. Poleg pošiljanja mora vsak proces tudi sprejeti vrednosti. Razlika sta prvi proces, ki ne rabi pošiljati predhodniku in zadnji proces, ki ne rabi pošiljati nasledniku.

Vzporedna simulacija prevajanja toplote v eni dimenziji v jeziku Julia za porazdeljeni pomnilnik je v izvorni kodi 3.3. Da poženemo Julijo v porazdeljenem načinu dodamo ukaz `-p` pri zagonu programa in število procesov, ki jih želimo pognati.

Za uporabo porazdeljenih zmožnosti v Juliji uporabimo modul `Distributed` in `DistributedArrays` za porazdeljene tabele. Na začetku definiramo vse potrebne konstante za delovanje programa. Makro `@everywhere` vzame izraz na desni in ga definira na vseh možnih procesih. Brez uporabe makroja `@everywhere` bi bil izraz definiran samo na glavem procesu. Definiramo tabelo z vrednostmi temperature `T` in jo porazdelimo s funkcijo `distribute`.

Nato definiramo tabelo oddaljenih kanalov. Vsak proces si bo lastil enega izmed kanalov, iz katerega bo bral in na katerega bodo ostali procesi pošiljali vrednosti. Pri definiciji kanala moramo podati tudi njegovo velikost. Z velikostjo kanala določimo koliko vrednosti lahko hrani preden blokira dodajanje novih vrednosti. Ko kanal ne vsebuje vrednosti, blokira branje iz kanala. Velikost kanalov smo nastavili na 1, da ne pride do zastojev pri komunikaciji

Izvorna koda 3.3: Vzporedna simulacija prevajanja toplote v eni dimenziji v jeziku Julia

```
using Distributed
@everywhere using DistributedArrays
@everywhere const N = 10
@everywhere const MAX_TIME = 3
@everywhere const DT = 0.001
@everywhere const DX = 0.1
@everywhere const K = 0.5
@everywhere const CFL = DT * K / DX^2
@everywhere const FloatCh = Channel{Float64}

@everywhere function compute_local_heat(
    T_DIST::DArray,
    neighbors::Array{RemoteChannel{FloatCh}}
)
    rank = myid() - 1
    N = length(T_DIST[:L])
    within = collect(2:N+1)
    LT = zeros(N + 2)
    LT[within] = T_DIST[:L]

    if rank > 1
        put!(neighbors[rank - 1], LT[2])
    end
    if rank < nworkers()
        LT[N + 2] = take!(neighbors[rank])
    end
    if rank < nworkers()
        put!(neighbors[rank + 1], LT[N + 1])
    end
    if rank > 1
        LT[1] = take!(neighbors[rank])
    end
    T_DIST[:L] = LT[within] +
                 CFL * (LT[within .+ 1] - 2 .* LT[within] + LT[within .- 1])
end

T = sin.(1:N)
T_DIST = distribute(T)
neighbors = [RemoteChannel(()->FloatCh(1)) for _ in 1:nworkers()]
for t in 1:MAX_TIME
    @sync for w in workers()
        @spawnat w compute_local_heat(T_DIST, neighbors)
    end
end
println(T_DIST)
```

(angl. deadlock).

Za vsak časovni interval zaženemo vse procese, ki izračunajo porazdelitev temperature na svojem delu tabele. Makro `@sync` poskrbi, da se vsi procesi znotraj zanke `for` končajo preden se nadaljuje izvajanje ostalega programa. Funkcija `workers` vrne identifikatorje procesov, ki so na voljo za vzporedno izvajanje. Identifikatorji se začnejo s številom 2, identifikator 1 pa je rezerviran za glavni proces. Z makrojem `@spawnat` vzporedno zaženemo izvajanje funkcije na željenem procesu. Funkcija `compute_local_heat` sprejme porazdeljeno tabelo in tabelo kanalov za komunikacijo s sosedi. Lokalno tabelo, ki si jo lasti vsak proces dobimo z indeksiranjem v globalno tabelo `T_DIST` in simbolom `:L`. Nato ustvarimo pomožno lokalno tabelo za temperaturo `LH`, ki ji dodamo dve robni točki. V notranje točke vstavimo vrednosti temperature, ki si jih lasti trenutni proces. S funkcijo `put!` pošljemo vrednost po željenem kanalu in s funkcijo `take!` sprejemo vrednost iz kanala. Z uporabo pike pred standardnimi operatorji kot sta plus in minus označimo, da bo operacija izvedena nad vsemi elementi tabele. S tem lahko izjemno na kratko zapišemo celotno enačbo prevajanja toplote.

Vzporedna simulacija prevajanja toplote v eni dimenziji v jeziku Chapel za porazdeljeni pomnilnik je v izvorni kodi 3.4. Podobno kot pri Juliji na začetku definiramo potrebne konstante. V spremenljivki `T_SPACE` definiramo območje nad katerim bomo računali porazdelitev temperature. Za razliko od Julije in MPI-ja tukaj ne potrebujemo eksplicitne komunikacije med procesi. S ključno besedo `dmapped` ustvarimo vzporedno porazdeljeno domeno `T_DOMAIN`. Levi argument je prostor, nad katerim operiramo, desni argument pa je željeni način porazdelitve. Mi smo izbrali `Stencil` porazdelitev ali porazdelitev s šablono. Porazdelitev s šablono sprejme prostor in argument `fluff`, ki nadzoruje koliko robnih točk naj doda v vsaki dimenziji. Z domeno `T_DOMAIN` pa ustvarimo porazdeljeno tabelo `T_DIST` in jo napolnemo z vrednostmi.

Nato za vsak časovni interval zaženemo vzporedne procese, ki izračunajo porazdelitev temperature. Uporabimo zanko `coforall`, ki za vsak proces

Izvorna koda 3.4: Vzporedna simulacija prevajanja toplote v eni dimenziji v jeziku Chapel

```
use StencilDist;

const DT = 0.001;
const DX = 0.1;
const K = 0.5;
const CFL = DT * K / DX**2;
const N = 10;
const MAX_T = 3;

var T_SPACE = {1..N};
var T_DOMAIN = T_SPACE dmapped Stencil(T_SPACE, fluff=(1,));
var T_DIST: [T_DOMAIN] real = sin(T_SPACE);

for t in 1..MAX_T {
    coforall L in Locales do on L {
        var LS = T_DIST.localSubdomain();
        for i in LS {
            T_DIST[i] += CFL *
                         (T_DIST[i + 1] - 2 * T_DIST[i] + T_DIST[i - 1]);
        }
    }

    T_DIST.updateFluff();
}

writeln(T_DIST);
```

zažene posebno nalogo (angl. task). S ključnima besedama `do on` za vsako iteracijo določimo, na katerem računskem mestu naj se naloga izvede. Vsak proces s klicem funkcije `localSubdomain` na globalno porazdeljeni tabeli izve katera poddomena mu je bila dodeljena. Ker porazdelitev s šablono avtomatsko poskrbi za robne točke, nam jih ni potrebno eksplicitno dodati. Za vsako točko v poddomeni nato izračunamo porazdelitev temperature. Ko vsi procesi zaključijo z nalogami, kličemo funkcijo `updateFluff`, ki posodobi vse robne točke z novimi vrednostmi.

Implicitno komunikacijo med več procesi v Chapelu lahko izpišemo z uporabo modula `CommDiagnostics`. Pred začetkom relevantnega dela kode dodamo klic funkcije `startVerboseComm()` in na koncu klic funkcije `stopVerboseComm()`. Vsa komunikacija med klicem funkcij bo izpisana na standardni izhod. Izsek iz izpisa za implementacijo porazdeljenege simulacije prevajanja toplote je v izvorni kodi 3.5. Uporabili smo dva procesa. Format izpisa je sledeč: identifikator procesa, ki je sprožil komunikacijo, vrstica v programu, tip komunikacije, identifikator procesa s katerim je komuniciral in količina podatkov, ki je bila prenesena. V našem primeru komunikacijo sprožimo s klicem funkcije `updateFluff()`. Ta pričakovano prenese 8 bajtov ali eno 64 bitno realno število na vsakem procesu, ker mora vsak proces posodobiti eno robno točko.

Izvorna koda 3.5: Skrajšan izpis implicitne komunikacije v Chapelu z modulom `CommDiagnostics`

```
...
0: heat.chpl:25: remote get, node 1, 8 bytes
...
1: heat.chpl:25: remote get, node 0, 8 bytes
...
```

Poglavje 4

Izbrani testi problemi

Izbrane jezike bomo ovrednotili na treh izbranih testnih problemih. Za vsak problem bomo predstavili njegov zaporedni algoritem, ki bo služil kot osnova ter algoritma za skupni in porazdeljeni pomnilnik. Probleme smo izbirali na podlagi njihove kompleksnosti in vzporednih značilnosti. Predvsem nas zanimajo različni načini porazdelitve podatkov in različni vzorci komunikacije.

4.1 Metoda voditeljev

Metoda voditeljev (angl. k-means, Lloyd's algorithm) je algoritem za iskanje gruč v podatkih [30]. Spada v skupino nenadzorovanih algoritmov strojnega učenja. Gruča se nanaša na množico točk, ki so združene skupaj zaradi določene podobnosti. V primeru metode voditeljev je podobnost definirana kot Evklidska razdalja med točkama. Cilj metode voditeljev je razdeliti n točk v d-dimenzionalnem prostoru v k gruč. Na koncu vsaka točka pripada gruči z najbližjim središčem (voditelj oz. centroid) glede na Evklidsko razdaljo. Voditelj je izračunan kot povprečje vseh točk v gruči. Za metodo voditeljev smo se odločili, ker je dober primer map-reduce vzorca [16]. Lahko se posploši tudi na druge algoritme strojnega učenja, ki jih je mogoče paralelizirati z istim vzorcem [14].

V algoritmu 1 je psevdokoda zaporednega algoritma metode voditeljev.

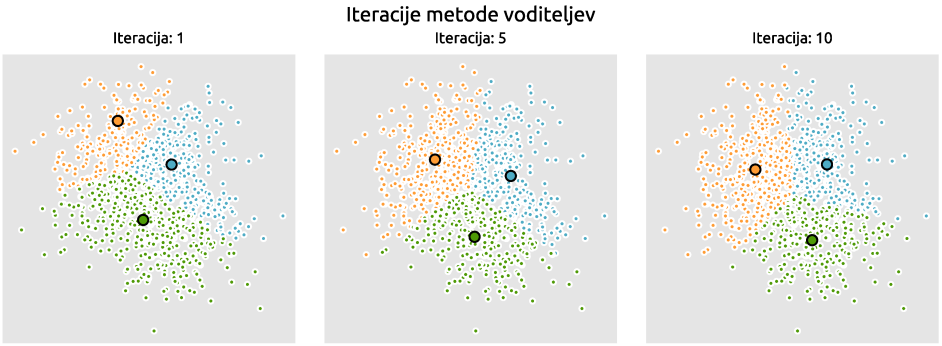
Algoritem 1 Psevdokoda zaporednega algoritma metode voditeljev

```
1:  clusters ← getInitialClusters(points, k)
2:  labels ← emptyLabels(length(points))
3:  for iteration ← 1 to MaxIteration do
4:      newClusters ← emptyClusters(k)
5:      for point in points do
6:          cluster ← findClosestCluster(point, clusters)
7:          labels[point] ← cluster
8:          addPoint(newClusters[cluster], point)
9:      end for
10:     clusters ← computeMeans(newClusters)
11: end for
```

Parametra algoritma sta tabela točk in število gruč. Na začetku inicializiramo k voditeljev iz vhodnih točk. Obstaja več možnih načinov inicializacije začetnih voditeljev. Kljub sodobnim metodam kot je `kmeans++` [4], izberemo naključne točke, ki bodo služile kot začetni voditelji. To nam omogoča večji nadzor nad algoritmom in lažjo kasnejšo primerjavo med implementacijami. Nato ustvarimo tabelo, ki bo hranila oznako gruče za vsako točko. Glavno zanko algoritma ponavljamo za vnaprej določeno število iteracij. Algoritem se lahko prilagodi tako, da se izvajanje zaključi, ko ni zaznanih novih prehodov točk med gručami. Na začetku vsake iteracije inicializiramo k praznih gruč, ki bodo hranile vmesne rezultate. Za vsako točko poiščemo najbližjo gručo, shranimo njeno oznako in jo dodamo v najbližjo gručo. Ko izračunamo oznake za vse točke, lahko iz začasnih gruč izračunamo nove položaje voditeljev. Končni rezultat algoritma so oznake gruč in položaji voditeljev. Slika 4.1 ilustrira iteracije metode voditeljev za $n = 1000$, $d = 2$ in $k = 3$.

Za učinkovito paralelizacijo je pomembno dejstvo, da je število točk tipično veliko večje od števila gruč. To pomeni, da je najbolje paralelizirati zanko kjer računamo oznake točk. S tem bo imel vsak vzporedni proces ali nit dovolj dela, da izničimo stroške upravljanja vzporednega okolja. Pod to spada ustvarjanje vzporednih procesov, komunikacija in ustavitev vzporednih procesov. To lahko predstavlja znaten del celotnega časa izvajanja vzporednega programa.

V algoritmu 2 je psevdokoda vzporednega algoritma metode voditeljev



Slika 4.1: Iteracije metode voditeljev za $n = 1000$, $d = 2$ in $k = 3$.

za skupni pomnilnik. Za vsako nit ustvarimo lastne lokalne gruče. S tem se izognemo stanju, kjer bi več niti posodabljalo isto gručo. Nato vsaka nit pridobi svoje lokalne točke in izračuna njihove oznake ter posodobi lokalne gruče. Na koncu vsaka nit izračuna voditelje svojih lokalnih gruč. Končna naloga glavne niti je, da združi lokalne gruče vseh niti in izračuna globalne voditelje gruč.

Algoritem 2 Psevdokoda vzporednega algoritma metode voditeljev za skupni pomnilnik

```
1:  clusters ← getInitialClusters(points, k)
2:  labels ← emptyLabels(length(points))
3:  for iteration ← 1 to MaxIteration do
4:      clustersPerThread ← emptyClustersPerThread(p)
5:      parallelFor thread ← threads do
6:          localClusters ← emptyClusters(k)
7:          localPoints ← getLocalPoints(points, thread)
8:          for point ← localPoints do
9:              cluster ← findClosestCluster(point, clusters)
10:             labels[point] ← cluster
11:             addPoint(localClusters[cluster], point)
12:         end for
13:         clustersPerThread[thread] ← computeMeans(localClusters)
14:     end parallelFor
15:     combinedClusters ← combineLocalClusters(clustersPerThread)
16:     clusters ← computeMeans(combinedClusters)
17: end for
```

V algoritmu 3 je psevdokoda vzporednega algoritma metode voditeljev za

porazdeljeni pomnilnik. Glavna razlika v primerjavi z algoritmom za skupni pomnilnik je, da mora vsak proces sedaj lokalno shraniti oznake točk, ki so na koncu združene. Nezanemarljiva je tudi komunikacija, ki je potrebna, da vsak proces prenese svoj del točk in globalne gruče. Točke so preprostega podatkovnega tipa (dvodimenzionalna tabela števil), medtem ko so gruče kompleksen podatkovni tip (`struct` v C in Juliji ter `record` v Chapelu) in bodo potrebovale posebno obravnavo.

Algoritem 3 Psevdokoda vzporednega algoritma metode voditeljev za porazdeljeni pomnilnik

```
 1: function computeClustersAndLabels(localPoints, clusters)
 2:     localClusters ← emptyClusters(k)
 3:     localLabels ← emptyLabels(length(localPoints))
 4:     for point ← localPoints do
 5:         cluster ← findClosestCluster(point, clusters)
 6:         localLabels[point] ← cluster
 7:         addPoint(localClusters[cluster], point)
 8:     end for
 9:     return (localClusters, localLabels)
10: end function
11: clusters ← getInitialClusters(points, k)
12: for iteration ← 1 to MaxIteration do
13:     clustersPerProcess ← emptyClustersPerProcess(p)
14:     labelsPerProcess ← emptyLabelsPerProcess(p)
15:     parallelFor process ← processes do
16:         localPoints ← getLocalPoints(points, process)
17:         (localClusters, localLabels) ←
                computeClustersAndLabels(localPoints, clusters)
18:         clustersPerProcess[process] ← localClusters
19:         labelsPerProcess[process] ← localLabels
20:     end parallelFor
21:     labels ← combineLocalLabels(labelsPerProcess)
22:     combinedClusters ← combineLocalClusters(clustersPerProcess)
23:     clusters ← computeMeans(combinedClusters)
24: end for
```

4.2 Urejanje z vzorčenjem

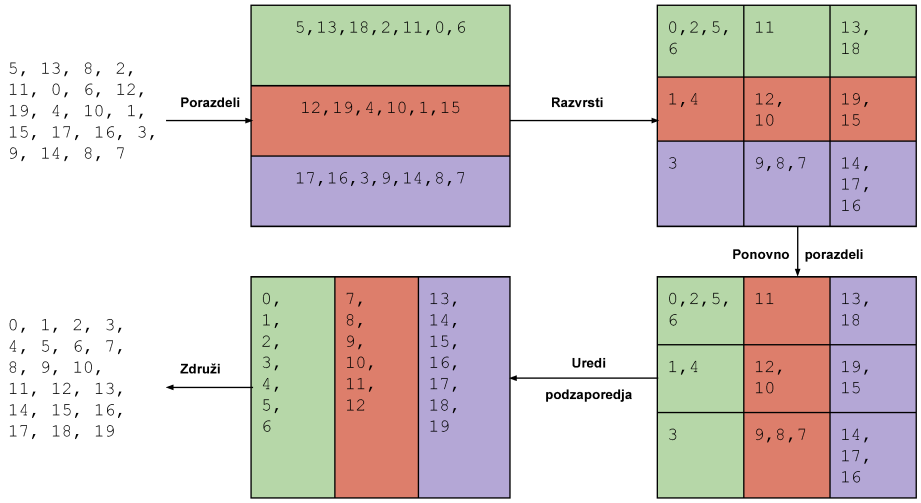
Urejanje je vseprisotna operacija v računalništvu, ki mora v kratkem času obdelati velike količine podatkov. Zato so potrebne učinkovite vzporedne

implementacije. Urejanje z vzorčenjem (angl. Samplesort) [19] zagotavlja dobro merilo za nove vzporedne programske jezike, ali so zmožni obvladati omenjene omejitve. Urejanje z vzorčenjem spada med deli in vladaj algoritme in je posplošena različica algoritma Quicksort. Urejanje z vzorčenjem začne z vzorčenjem ločilnih elementov iz vhodne tabele elementov. Vzorčimo $m-1$ elementov, kjer je m število vzporednih procesov. Ločilni elementi so uporabljeni za ločevanje elementov in morajo biti izbrani tako, da bo ločitev enakomerna. V primeru slabo izbranih ločilnih elementov, vzporedni procesi dobijo neenakomerno količino dela. To pomeni slabšo učinkovitost algoritma in počasnejši izvajalni čas. Zato uvedemo dodatni faktor vzorčenja o in vzorčimo om elementov. Vzorčene elemente uredimo, izberemo $m-1$ enakomerno razporejenih elementov, ki jih uporabimo za ločilne elemente.

Ključna podatkovna struktura v algoritmu je $m \times m$ matrika košev (angl. bins). Vsak vzporedni proces dobi del vhodne tabele in si lasti eno vrstico v matriki. Lokalne elemente nato razdelijo v ustrezne koše v svoji vrstici na podlagi ločilnih elementov. Ko je matrika napolnjena, procesi prevzamejo lastništvo nad stolpci. Vsak stolpec vsebuje podzaporedja deljena glede na ločilne elemente. Stolpce vzporedno uredijo v urejena podzaporedja, ki so na koncu združena v urejeno tabelo. Izbira dejanskega algoritma za urejanje om vzorcev in podzaporedij je poljubna. Mi smo izbrali algoritem Quicksort in ga prav tako uporabili kot zaporedno različico tega problema. Psevdokodo algoritma Quicksort smo izpustili, ker je njegova psevdokoda že splošno znana.

Slika 4.2 prikazuje potek urejanja z vzorčenjem za 20 elementov in 3 vzporedne procese. Vsaka izmed različnih barv prikazuje različen vzporedni proces.

V algoritmu 4 je psevdokoda vzporednega urejanja z vzorčenjem za skupni pomnilnik. Za skupni pomnilnik ustvarimo dvodimenzionalno tabelo, ki jo bomo uporabili kot matriko košev. Vsaka nit pridobi svoje elemente in jih razvrsti v koše v svoji vrstici. Ko so vsi elementi razvrščeni, vsaka nit prevzame stolpec matrike in uredi elemente v stolpcu. Na koncu glavna nit



Slika 4.2: Potek urejanja z vzorčenjem za 20 elementov in 3 vzporedne procese

združi vsa urejena podzaporedja v urejeno tabelo.

V algoritmu 5 je psevdokoda vzporednega urejanja z vzorčenjem za porazdeljeni pomnilniku. Pri porazdeljenem pomnilniku ni mogoče uporabiti dejanske dvodimenzionalne tabele, ampak si vsak proces lokalno inicializira svojo vrstico. Ko svoje lokalne elemente razporedijo v koše, morajo poslati koše, ki jih potrebujejo ostali procesi. S tem bo vsak proces prevzel koše iz svojega stolpca. Vsak proces počaka na potrebne koše iz ostalih procesov in uredi njihove elemente. Procesi vrnejo urejena podzaporedja, ki jih glavni proces na koncu združi v urejeno tabelo.

4.3 Problem n teles

Pri problemu n teles moramo v diskretnih časovnih intervalih izračunati položaje in hitrosti n teles, ki vplivajo en na drugega zaradi gravitacijske sile. Za vsako telo imamo podatek o začetnem položaju in hitrosti.

Problem n teles je uporaben na mnogih znanstvenih področjih. Fiziki

Algoritem 4 Psevdokoda vzporednega urejanja z vzorčenjem za skupni pomnilnik

```
1: samples ← getSamples(m)
2: bins ← initializeBinMatrix(m)
3: parallelFor thread in threads do
4:     localElements ← getLocalElements(elements, thread)
5:     for element ← localElements do
6:         bin ← getBin(element, samples)
7:         AddElement(bins[thread][bin], element)
8:     end for
9: end parallelFor
10: sortedSubsequencesPerThread ← []
11: parallelFor thread ← threads do
12:     columnElements ← getColumnElements(bins, thread)
13:     sortedSubsequencesPerThread[thread] ← sort(columnElements)
14: end parallelFor
15: sortedElements ← combineSubsequences(sortedSubsequencesPerThread)
```

ga uporabljajo za določanje položajev in hitrosti zvezd, medtem ko kemike zanima gibanje atomov. Za nas pa je zanimiv, ker se pogosto uporablja za ovrednotenje vzporednih jezikov in ogrodij [38, 45]. Uvrščen je bil tudi kot eden izmed pomembnih vzporednih razredov (angl. parallel dwarf) [5]. Vzporedni razred je skupina algoritmov in aplikacij, ki si delijo skupni vzorec računanja in komunikacije.

Rešitev problema n teles najdemo s simulacijo obnašanja teles. Vsako telo se obnaša kot točkasto in deluje na ostala telesa z gravitacijsko silo. Pri točkastih telesih lahko zanemarimo njihove razsežnosti. Simulacija poteka v diskretnih časovnih intervalih. Z uporabo Newtonovega gravitacijskega zakona izračunamo gravitacijsko silo, ki deluje na telesa. Če ima telo q položaj $\vec{s}_q(t)$ ob času t in telo k položaj $\vec{s}_k(t)$, potem je sila s katero deluje telo k na telo q določena z enačbo

$$\vec{F}_{qk}(t) = -\frac{Gm_q m_k}{|\vec{s}_q(t) - \vec{s}_k(t)|^3}[\vec{s}_q(t) - \vec{s}_k(t)].$$

V prejšnji enačbi je G gravitacijska konstanta z vrednostjo

$$6.673 \times 10^{-11} m^3 kg^{-1} s^{-2},$$

kjer sta m_q in m_k masi teles q in k. Z uporabo te enačbe izračunamo skupno

Algoritem 5 Psevdokoda vzporednega urejanja z vzorčenjem za porazdeljeni pomnilnik

```
 1: function getSortedSubsequence(process, localElements, samples)
 2:     row ← initializeBinRow(m)
 3:     for element in localElements do
 4:         bin ← getBin(element, samples)
 5:         AddElement(bins[bin], element)
 6:     end for
 7:     for (bin, otherProcess) ← zip(row, processes) do
 8:         if process != otherProcess then
 9:             sendToProcess(otherProcess, bin)
10:         end if
11:     end for
12:     column ← []
13:     for otherProcess ← processes do
14:         if process != otherProcess then
15:             column[otherProcess] ← receiveFromProcess(otherProcess)
16:         end if
17:     end for
18:     subsequence ← getColumnElements(column)
19:     return sort(subsequence)
20: end function
21: samples ← getSamples(m)
22: sortedSubsequencesPerProcess ← []
23: parallelFor process ← processes do
24:     localElements ← getLocalElements(elements, process)
25:     sortedSubsequence ←
            getSortedSubsequence(process, localElements, samples)
26:     sortedSubsequencesPerProcess[process] ← sortedSubsequence
27: end parallelFor
28: sortedElements ← combineSubsequences(sortedSubsequencesPerProcess)
```

silo na določeno telo tako, da seštejemo vplive sil vseh ostalih teles. Izračun skupne sile je podan z enačbo

$$\vec{F}_q(t) = \sum_{k=0 \wedge k \neq q}^{n-1} \vec{F}_{qk}.$$

Spremembo položaja in hitrosti teles izračunamo z uporabo drugega Newtonovega zakona. Ta pravi, da je sila na telo enaka produktu njegove mase in njegovega pospeška $\vec{F}_q(t) = m_q \vec{a}_q(t)$, kjer je $\vec{a}_q(t)$ pospešek telesa q ob času t. Iz tega sledi

$$\vec{a}_q(t) = \vec{s}_q{}''(t) = \frac{\vec{F}_q(t)}{m_q}.$$

S tem dobimo sistem diferencialnih enačb, kjer moramo za vsak čas t najti položaj $\vec{s}_q(t)$ in $\vec{v}_q(t) = \vec{s}_q{}'(t)$. Sistem rešimo z uporabo Eulerjeve metode, ki je numerična metoda za integriranje diferencialnih enačb. Če poznamo vrednost funkcije $g(t_0)$ in njen odvod $g'(t_0)$ ob času t_0, potem lahko aproksimiramo njeno vrednost ob času $t_0+\Delta t$. Če vemo, da je točka $(t_0, g(t_0))$ na premici, in vemo naklon premice $g'(t_0)$, potem je enačba premice

$$y = g(t_0) + g'(t_0)(t - t_0).$$

S tem izračunamo vrednost ob času $t = t_0 + \Delta t$ in dobimo aproksimacijo

$$g(t + \Delta t) \approx g(t_0) + g'(t_0)(t_0 + \Delta t - t_0) = g(t_0) + \Delta t g'(t_0).$$

Ker poznamo vrednosti $\vec{s}_q(\Delta t - 1)$ in $\vec{s}_q{}'(\Delta t - 1)$, lahko uporabimo Eulerjevo metodo in formulo za pospešek, da izračunamo $\vec{s}_q(\Delta t)$ in $\vec{v}_q(\Delta t)$. Izračun položaja telesa je določen z enačbo

$$\vec{s}_q(\Delta t) \approx \vec{s}_q(\Delta t - 1) + \Delta t \vec{s}_q{}'(\Delta t - 1) = \vec{s}_q(\Delta t - 1) + \Delta t \vec{v}_q(\Delta t - 1),$$

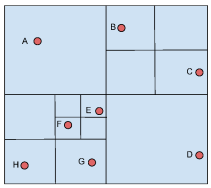
izračun hitrosti telesa pa z enačbo

$$\vec{v}_q(\Delta t) \approx \vec{v}_q(\Delta t - 1) + \Delta t \vec{v}_q{}'(\Delta t - 1) = \vec{v}_q(\Delta t - 1) + \Delta t \vec{a}_q(\Delta t - 1)$$

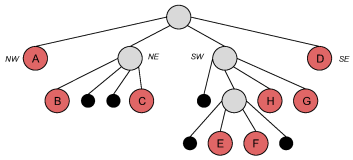
$$= \vec{v}_q(\Delta t - 1) + \Delta t \frac{\vec{F}_q(\Delta t - 1)}{m_q}.$$

Najpreprostejši algoritem simulacije je tak, ki izračuna vpliv vseh teles na določeno telo. Rezultat je $O(n^2)$ algoritem, ki je časovno preveč kompleksen za veliko količino teles. Zaradi tega so se razvile hierarhične drevesne metode, ki zmanjšajo časovno kompleksnost na $O(nlogn)$. To dosežejo z združevanjem bližnjih teles, ki jih obravnavajo kot eno telo. Če je skupina teles dovolj oddaljena, lahko aproksimiramo njihov vpliv z uporabo masnega središča.

Algoritem Barnes-Hut [8] je osnovan na podatkovni strukturi octree [33]. Barnes-Hut razdeli tridimenzionalni prostor teles v osem enako velikih celic. V dveh dimenzijah uporabimo poenostavljeno različico octreeja, imenovano quadtree, ki razdeli prostor teles v štiri enake kvadrante. Koren drevesa predstavlja celico, ki vsebuje vsa telesa. Drevo gradimo tako, da dodajamo telesa v prazen koren drevesa. Ko celica vsebuje več kot eno telo, jo razdelimo na osem podcelic. Rezultat je drevo, ki ima celice za vozlišča in posamezna telesa za liste. Na sliki 4.3 smo uporabili dvodimenzionalno predstavitev za lažjo razlago. Dvodimenzionalni prostor teles na sliki 4.3a je razdeljen v kvadrante, taki delitvi pa ustreza drevo (quadtree) na sliki 4.3b, ki ga zgradi algoritem Barnes-Hut.



(a) Prostor teles razdeljen v kvadrante



(b) Barnes-Hut drevo

Slika 4.3: Dvodimenzionalna predstavitev prostora teles in ustrezno Barnes-Hut drevo

Sile na telesa izračunamo tako, da za vsako telo naredimo obhod Barnes-Hut drevesa. Začnemo v korenu drevesa in rekurzivno nadaljujemo za vsako podcelico. Za vsako celico preverimo, če je dovolj oddaljeno od telesa, da jo lahko aproksimiramo z njenim masnim središčem. Celica je dovolj oddaljena, če velja pogoj

$$\frac{l}{d} < \theta,$$

kjer je l dolžina stranice celice, d razdalja med telesom in masnim središčem celice in θ poljubno nastavljen parameter, ki nadzira natančnost algoritma.

Za $\theta = 0$ se algoritem izrodi v $O(n^2)$ različico, kjer izračunamo vpliv sil med vsemi telesi. Za $\theta = 1.2$ več celic smatramo za dovolj oddaljene in lahko hitreje izračunamo sile, a dobimo manj natančne rezultate. Zato je parameter θ najpogosteje nastavljen na 0.5 za ravnotežje natančnosti in hitrosti [43]. Če celica ni dovolj oddaljena, jo "odpremo" in obiščemo njene podcelice. Ko dosežemo liste drevesa, izračunamo neposreden vpliv sile telesa v listu na trenutno telo. Vse implementacije si bodo delile funkcije za vstavljanje telesa v drevo in za izračun gravitacijske sile. Njihovo psevdokodo smo zapisali v algoritmu 6. Zaporedna različica algoritma Barnes-Hut je v algoritmu 7.

Psevodokoda algoritma Barnes-Hut za skupni pomnilnik je v algoritmu 8. Pri algoritmu za skupni pomnilnik smo se odločili za statično paralelizacijo. Pri statični paralelizaciji med niti razdelimo vozlišča na določenem nivoju drevesa. Mi smo si izbrali drugi nivo, kjer ima drevo Barnes-Hut 64 vozlišč, kar je dovolj, da uporabimo do 64 niti za paralelizacijo.

Na začetku zgradimo prazno drevo do drugega nivoja in si shranimo liste praznega drevesa, ki jih bomo razdelili med niti. Vsaka nit za vsako vozlišče pregleda telesa in vstavi tista, ki spadajo v prostor, ki ga predstavlja vozlišče. Ko niti končajo z vstavljanjem teles, je potrebno posodobiti maso in masno središče praznih vozlišč od drugega nivoja navzgor. S pravilno posodobljenim drevesom lahko vzporedno izračunamo sile in nato še nove hitrosti in položaje.

Prednost statične paralelizacije je lažja implementacija, ki se dobro prilega sistemom s skupnim pomnilnikom. Vse niti vzporedno delajo na isti podatkovni strukturi in ni potrebnega dodatnega dela, da sestavimo končno drevo. Slabost statične paralelizacije je, da lahko uporabimo le 64 niti. To lahko popravimo tako, da razdelimo vozlišča na poljubnem nižjem nivoju. Statična paralelizacija je odvisna od porazdelitve teles po prostoru. Če so telesa neenakomerno porazdeljena, bodo ene izmed niti dobile več teles in posledično upočasnile celotno izvajanje algoritma.

Algoritem 6 Psevdokoda funkcij za vstavljanje telesa v Barnes-Hut drevo in izračun gravitacijske sile

```
1:  function addForce(body, position, mass)
2:      Δposition ← position − body.position
3:      distance ← distance(body.position, position)
4:      force ← (−G · body.mass · mass) / distance^3
5:      body.force ← body.force + force · Δposition
6:  end function
7:  function computeForce(cell, body)
8:      if isLeaf(cell) then
9:          addForce(body, cell.body.position, cell.body.mass)
10:     else
11:         l ← getCellLength(cell)
12:         d ← distance(cell, body)
13:         if l / d < θ then
14:             addForce(body, cell.centerOfMass, cell.mass)
15:         else
16:             for child in children do
17:                 computeForce(child, body)
18:             end for
19:         end if
20:     end if
21: end function
22: function insertBody(cell, body)
23:     if !cell.body and isLeaf(cell) then
24:         cell.body ← body
25:         cell.centerOfMass ← body.position
26:         cell.mass ← body.mass
27:     else
28:         if isLeaf(cell) then
29:             openCell(cell)
30:         end if
31:         for child ← cell.children do
32:             if cellContainsPosition(child, body.position) then
33:                 insertBody(child, body)
34:                 break
35:             end if
36:         end for
37:         cell.centerOfMass ← (cell.mass · cell.centerOfMass +
              body.mass · body.position) / (cell.mass + body.mass)
38:         cell.mass ← cell.mass + body.mass
39:     end if
40: end function
```

Algoritem 7 Psevdokoda zaporednega algoritma Barnes-Hut

```
1:  for iteration ← 1 to MaxIteration do
2:      root← initializeRootCell( )
3:      for body ← bodies do
4:          insertBody(root, body)
5:      end for
6:      for body ← bodies do
7:          computeForce(root, body)
8:      end for
9:      for body ← bodies do
10:         body.position ← body.position + Δt · body.velocity
11:         body.velocity ← body.velocity + Δt · body.force / body.mass
12:     end for
13: end for
```

Algoritem 8 Psevdokoda algoritma Barnes-Hut za skupni pomnilnik

```
1:  maxLevel ← 2
2:  for iteration ← 1 to MaxIteration do
3:      root← initializeRootCell( )
4:      constructEmptyTree(root, maxLevel)
5:      leafCells ← getLeafCells(root)
6:      parallelFor leafCell ← leafCells do
7:          for body ← bodies do
8:              if cellContainsPosition(leafCell, body.position) then
9:                  insertBody(leafCell, body)
10:             end if
11:         end for
12:     end parallelFor
13:     updateEmptyCells(root, maxLevel)
14:     parallelFor body ← bodies do
15:         computeForce(root, body)
16:     end parallelFor
17:     parallelFor body ← bodies do
18:         body.position ← body.position + Δt · body.velocity
19:         body.velocity ← body.velocity + Δt · body.force / body.mass
20:     end parallelFor
21: end for
```

Za porazdeljeni pomnilnik smo izbrali dinamično paralelizacijo algoritma opisano v [43]. Pri dinamični paralelizaciji uporabimo ortogonalno rekurzivno bisekcijo (ORB), ki enakomerno razdeli telesa med procese. Psevdokoda algoritma ORB je v algoritmu 9. ORB izvede $log_2(p)$ iteracij, kjer je p število procesov. Zato mora biti število procesov enako potenci števila 2. Vsak proces na začetku pripada skupini vseh procesov. V vsaki iteraciji izbere koordinato, po kateri bo razdelil prostor (X, Y ali Z). Za vsako delitev se odloči ali bo "pod" ali "nad" delitvijo. Sledi bisekcija, ki najde tako delitev teles po izbrani koordinati, da imata oba dela enako količino dela. Delo za vsako telo dobimo tako, da izmerimo čas, ki je potreben, da izračunamo silo na telo. Na začetku ima vsako telo delo enako 1, zato procesi dobijo enako število teles. Vsak proces nato razdeli svoja lokalna telesa glede na stran delitve. Vsakemu procesu pripada partnerski proces, ki se nahaja na drugi strani delitve. Partnerskemu procesu pošljejo svoja lokalna telesa, ki se nahajajo na drugi strani delitve. Prav tako sprejmejo telesa od partnerskega procesa in jih dodajo svojim lokalnim telesom. V vsaki naslednji iteraciji procesi posodobijo skupino procesov tako, da vsebuje samo tiste, ki se nahajajo na isti strani delitve. Rezultat algoritma za vsak proces so telesa, zaporedje delitev in pripadajoči partnerski procesi pri vsaki delitvi. Ko enakomerno porazdelimo telesa, lahko nadaljujemo z algoritmom Barnes-Hut za porazdeljeni pomnilnik. Njegova psevdokoda je v algoritmu 10. Vsak proces zgradi lokalno Barnes-Hut drevo iz lokalnih teles. Samo lokalno drevo ni dovolj, da izračunamo sile na telesa. Sestaviti moramo "lokalno bistveno" drevo, ki bo vsebovalo zadostno količino informacij. Lokalno bistveno drevo zgradimo tako, da vsak proces pošlje drugemu procesu del svojega drevesa, za katerega ve, da ga potrebuje. Vsak proces sprejme ostale dele dreves in jih vstavi v svoje lokalno drevo. Ko ima vsak proces lokalno bistveno telo, lahko nadaljuje z izračunom sil, novih položajev in hitrosti.

Algoritem 9 Psevdokoda algoritma za ortogonalno rekurzivno bisekcijo

```
1:  function ORB(process, bodies, bounds)
2:      procBounds ← bounds
3:      procGroup ← 0
4:      aboveSplit ← false
5:      nSplits ← LOG2(length(processes))
6:      splitBounds, partners ← []
7:      for i ← 0 to nSplits do
8:          procGroup ← GETGROUP(aboveSplit)
9:          groupPartners ← GETGROUPPARTNERS(procGroup)
10:         coordinate ← GETCOORDINATE(i)
11:         aboveSplit ← ISABOVESPLIT(process)
12:         split ← BISECTION(procBounds, coordinate, groupPartners, bodies)
13:         (procBounds, otherBounds) ← GETUPDATEDBOUNDS(procBounds, coordinate, split)
14:         ADDBOUNDS(splitBounds, (procBounds, otherBounds))
15:         procBodies, otherBodies ← []
16:         for body ← bodies do
17:             if (body.position[coordinate] > split) == aboveSplit then
18:                 ADDBODY(procBodies, body)
19:             else
20:                 ADDBODY(otherBodies, body)
21:             end if
22:         end for
23:         partner ← GETPARTNERPROCESS(process)
24:         ADDPARTNER(partners, partner)
25:         SENDBODIES(partner, otherBodies)
26:         newBodies ← RECEIVEBODIES(process)
27:         bodies ← MERGE(newBodies, procBodies)
28:     end for
29:     return (bodies, splitBounds, partners)
30: end function
```

Algoritem 10 Psevdokoda algoritma Barnes-Hut za porazdeljeni pomnilnik

```
 1: for iteration ← 1 to MaxIteration do
 2:     parallelFor process ← processes do
 3:         root← initializeRootCell( )
 4:         bounds ← getBounds(bodies)
            (bodies, splitBounds, partners) ← orb(process, bodies, bounds)
 5:         for body ← bodies do
 6:             insertBody(root, body)
 7:         end for
 8:         for (split, partner) ← zip(splitBounds, partners) do
 9:             cellsToSend ← getCellsToSend(root, split)
10:             sendCells(partner, cellsToSend)
11:             cellsToInsert ← receiveCells(process)
12:             insertCells(root, cellsToInsert)
13:         end for
14:         for body ← bodies do
15:             startTime ← currentTime( )
16:             computeForce(root, body)
17:             body.work ← currentTime( ) − startTime
18:         end for
19:         for body ← bodies do
20:             body.position ← body.position + Δt · body.velocity
21:             body.velocity ← body.velocity + Δt · body.force / body.mass
22:         end for
23:     end parallelFor
24: end for
```

Poglavje 5

Rezultati

Programske jezike smo ovrednotili s primerjavo izvajalnih časov in težavnostjo implementacij.

Za zaporedne implementacije in implementacije za skupni pomnilnik smo uporabljali procesor AMD Ryzen 2700X in 16 GB DDR4 RAM pomnilnika. Procesor je imel 8 jeder in 16 niti skupaj z 768 KB L1 predpomnilnika, 4 MB L2 predpomnilnika in 16 MB L3 predpomnilnika.

Za porazdeljene implementacije smo imeli na voljo 32 HP DL160 G6 strežnikov s procesorjem Intel Xeon 5520 in 6 GB DDR3 RAM pomnilnika. Procesor je imel 8 jeder in 8 navideznih jeder (angl. hyperthreading) skupaj s 64 KB L1 predpomnilnika, 256 KB L2 predpomnilnika in 8 MB L3 predpomnilnika.

Pri jeziku C smo uporabili ogrodje OpenMP za skupni pomnilnik in ogrodje MPI za porazdeljeni pomnilnik. Uporabili smo naslednje različice jezikov in njihovih prevajalnikov:

- Chapel 1.18
- Julia 1.1
- GCC 7.2
- MPICH 3.2, ki uporablja GCC različico 5.4

- OpenMP 4.5

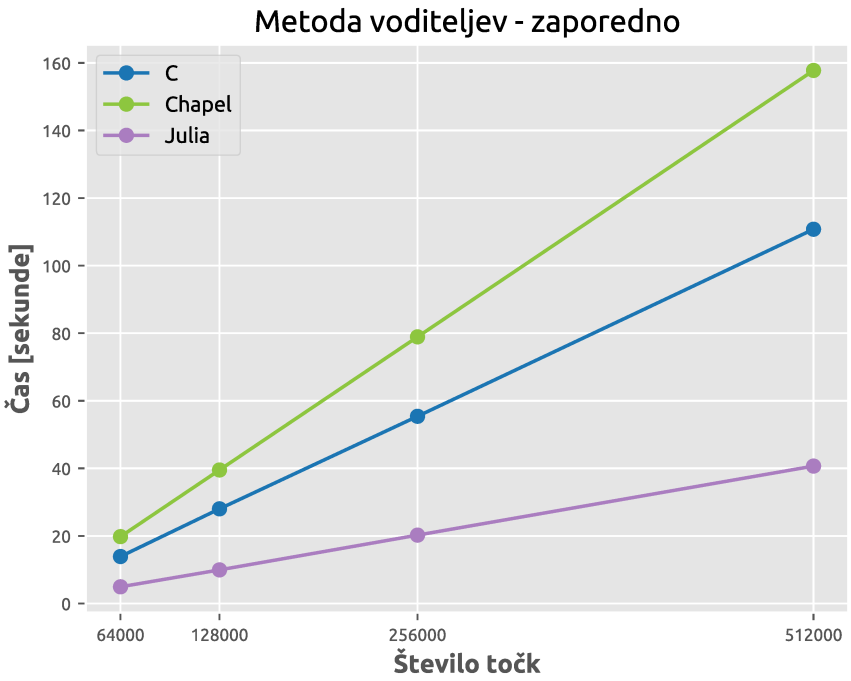
Pri prevajanju vseh C programov smo uporabili standardni ukaz za optimizacijo `-O3`. Pri prevajanju Chapel programov smo uporabili ukaz `--fast`, ki je alias za ukaze `--no-checks`, `--specialize` in `-O`. Ukaz `--no-checks` izklopi preverjanje mej pri dostopanju do elementov tabel, ukaz `--specialize` iz okoljske spremenljivke `CHPL_TARGET_CPU` prebere model procesorja in zanj zgenerira specializirano kodo in ukaz `-O`, ki vklopi optimizacijo pri prevajanju generirane C kode. Julia programe smo poganjali z ukazi `-O3`, `--check-bounds=no` in `--math-mode=fast`. Ukaz `-O3` vklopi najvišjo stopnjo optimizacij, `--check-bounds=no` izklopi preverjanje mej pri dostopanju do elementov tabel in `--math-mode=fast` omogoči uporabo optimiziranih operacij s plavajočo vejico.

5.1 Metoda voditeljev

Implementacije algoritma metode voditeljev smo ovrednotili na štirih umetno generiranih podatkovnih zbirkah. Vsaka podatkovna zbirka je vsebovala 256 gruč, točke pa so imele 128 dimenzij. Vse implementacije so se izvajale 10 iteracij. Izvajalni časi zaporednih implementacij metode voditeljev so na sliki 5.1.

Ozko grlo vseh implementacij je izračun Evklidske razdalje. V Cju in Chapelu uporabljamo lastno implementacijo Evklidske razdalje. Julia ima vgrajen modul za linearno algebro (`LinearAlgebra`) osnovan na visoko optimizirani knjižnici LAPACK [3]. Iz modula uporabimo funkcijo `norm`, ki izračuna normo vektorja in jo lahko uporabimo za izračun Evklidske razdalje.

Na začetku smo pri Juliji prav tako uporabili lastno implementacijo Evklidske razdalje, a je bila zaradi velike porabe pomnilnika počasnejša od C in Chapel implementacij. Izvajalni časi zaporednih implementacij metode voditeljev z lastno in `norm` implementacijo Evklidske razdalje so na sliki 5.2. Ker povečana poraba pomnilnika ni bila pričakovana, smo dodatno raziskali



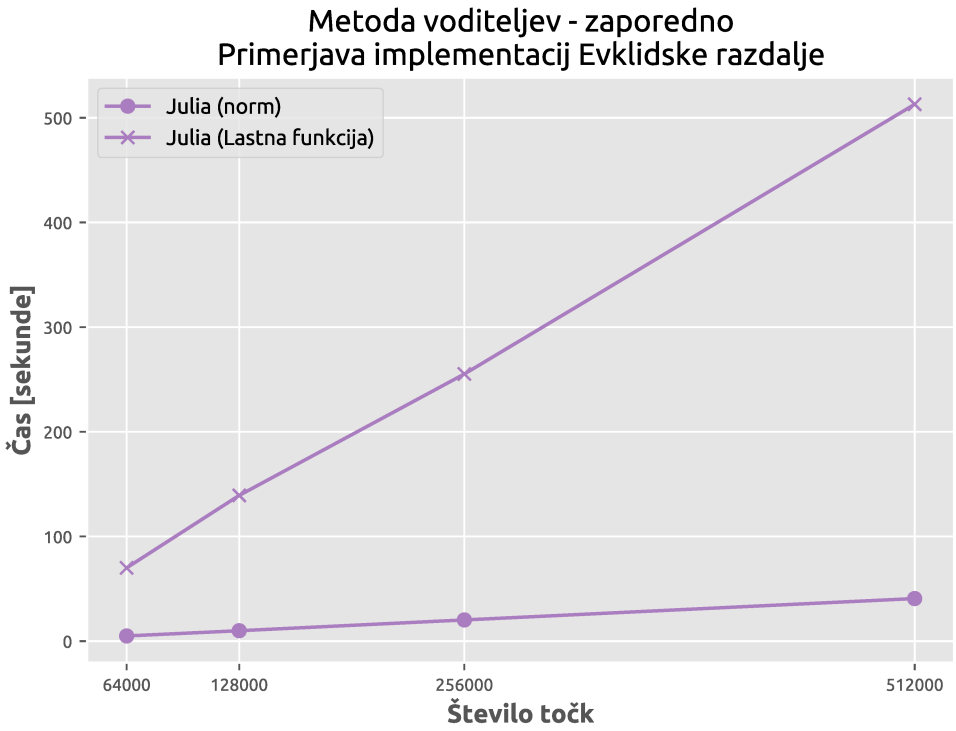
Slika 5.1: Izvajalni časi zaporednih implementacij metode voditeljev

njen vzrok. Pri zagonu Julije v ukazni vrstici smo uporabili dodaten ukaz `--track-allocation=user`. S tem ukazom Julia meri in shrani vse alokacije, ki jih zahteva uporabnikova koda. Rezultat so datoteke s končnico `.mem`, ki vsebujejo originalne vrstice izvorne kode označene s številom bajtov, ki so jim bili dodeljeni.

Poraba pomnilnika za izračun Evklidske razdalje z lastno implementacijo je v izvorni kodi 5.1, z `norm` implementacijo pa v izvorni kodi 5.2. Lastna implementacija je dodelila 2621440000 bajtov ali 2.6 GB, medtem ko `norm` ni zahtevala nobene dodelitve pomnilnika.

Izvorna koda 5.1: Poraba pomnilnika za izračun Evklidske razdalje z lastno implementacijo (izsek iz `.mem` datoteke)

```
        - function distance(cluster::Cluster, point::Point)
2621440000      sqrt(sum((cluster.mean .- point) .^ 2))
        - end
```



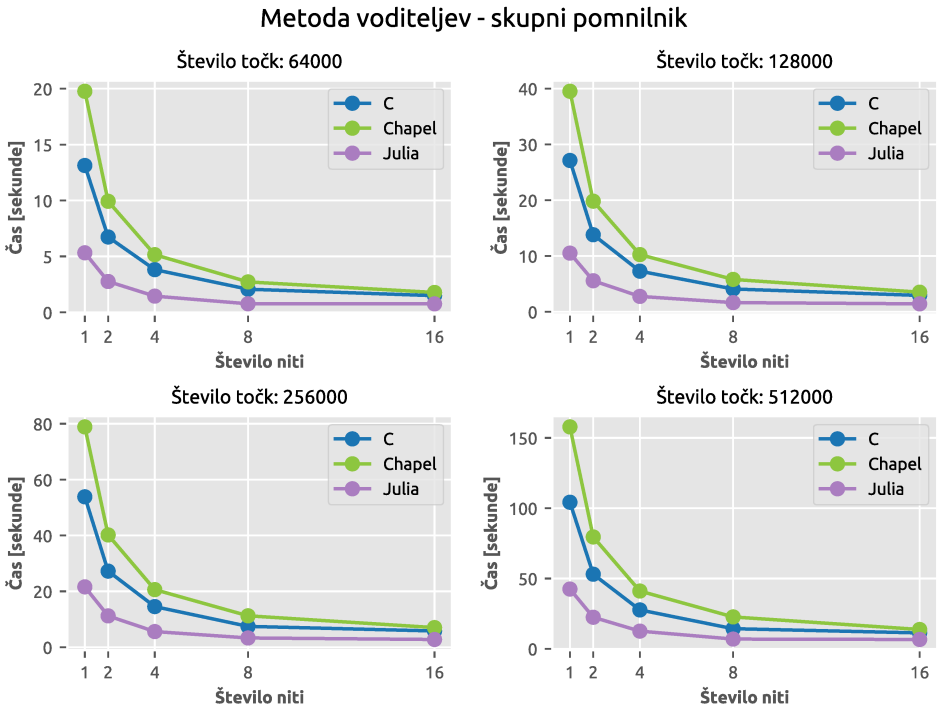
Slika 5.2: Izvajalni časi zaporednih implementacij metode voditeljev z lastno in `norm` implementacijo Evklidske razdalje

Izvorna koda 5.2: Poraba pomnilnika za izračun Evklidske razdalje z `norm` implementacijo (izsek iz `.mem` datoteke)

```
- function distance(cluster::Cluster, point::Point)
0     norm(cluster.mean .- point)
- end
```

Taka poraba pomnilnika izhaja iz dejstva, da uporabljamo nespremenljive (angl. immutable) tabele za predstavitev vektorjev. Nespremenljive tabele omogočajo Juliji, da bolje optimizira kodo, ker ve, da se tekom izvajanja programa ne bodo spreminjale. Slabost je, da vsaka operacija nad nespremenljivimi tabelami ustvari novo tabelo, kar poveča porabo pomnilnika.

Izvajalni časi implementacij metode voditeljev za skupni pomnilnik so na sliki 5.3. Julia je zopet najhitrejša zaradi uporabe optimiziranega izračuna Evklidske razdalje.



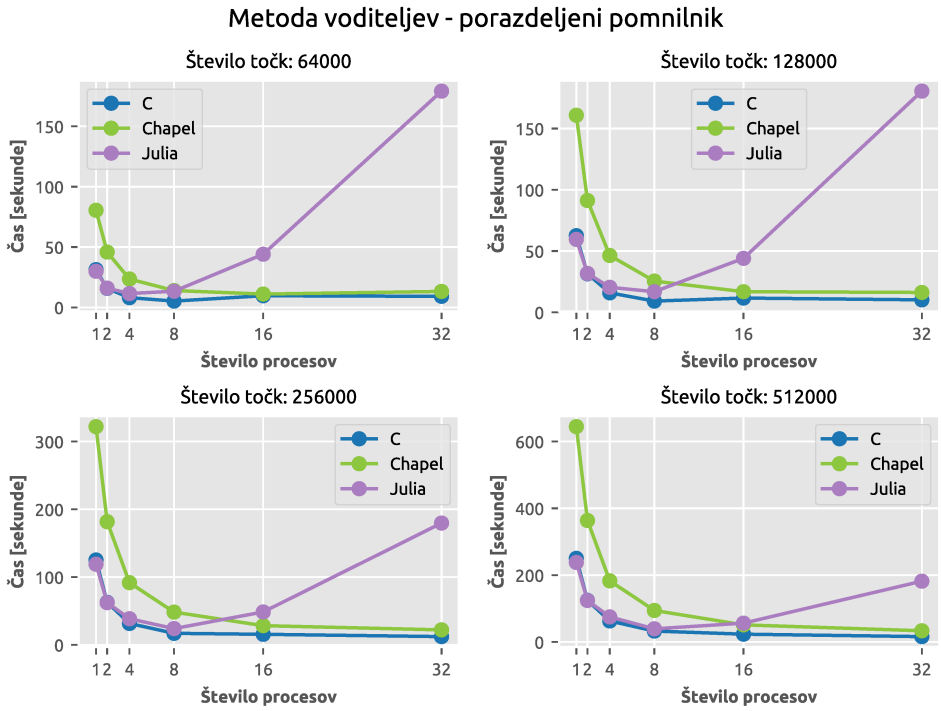
Slika 5.3: Izvajalni časi implementacij metode voditeljev za skupni pomnilnik

Izvajalni časi implementacij metode voditeljev za porazdeljeni pomnilnik so na sliki 5.4. Pri Juliji opazimo, da se poslabša izvajalni čas pri majhnem številu točk in velikem številu vzporednih procesov. To nakazuje na visoke stroške komunikacije pri pošiljanju lokalnih gruč ostalim procesom in sprejem gruč od ostalih procesov.

Gruče smo v naših implementacijah predstavili kot kompleksen podatkovni tip, ki vsebuje število točk, njihovo vsoto in povprečje. Chapel in Julia avtomatsko serializirata kompleksne podatkovne tipe pri prenosu med vzporednimi procesi. C in MPI pričakujeta, da implementiramo lastno serializacijo kompleksnih podatkovnih tipov. V izvorni kodi 5.3 je koda potrebna za pretvorbo gruče v MPI podatkovni tip.

Izvorna koda 5.3: Funkcija, ki pretvori C podatkovni tip v MPI podatkovni tip za prenos med vzporednimi procesi

```
typedef struct _cluster_t {
    point_t sum;
    point_t mean;
    int count;
} cluster_t;
void create_cluster_mpi_datatype(MPI_Datatype* cluster_datatype) {
    cluster_t tmp_cluster;
    MPI_Datatype tmp_type;
    int blocklen[3] = { POINT_SIZE, POINT_SIZE, 1 };
    MPI_Datatype type[3] = { MPI_DOUBLE, MPI_DOUBLE, MPI_INT };
    MPI_Aint disp[3];
    disp[0] = offsetof(cluster_t, sum);
    disp[1] = offsetof(cluster_t, mean);
    disp[2] = offsetof(cluster_t, count);
    MPI_Type_create_struct(3, blocklen, disp, type, &tmp_type);
    MPI_Aint lb, extent;
    MPI_Type_get_extent(tmp_type, &lb, &extent);
    MPI_Type_create_resized(tmp_type, lb, extent, cluster_datatype);
    MPI_Type_commit(cluster_datatype);
}
```



Slika 5.4: Izvajalni časi implementacij metode voditeljev za porazdeljeni pomnilnik

5.2 Urejanje z vzorčenjem

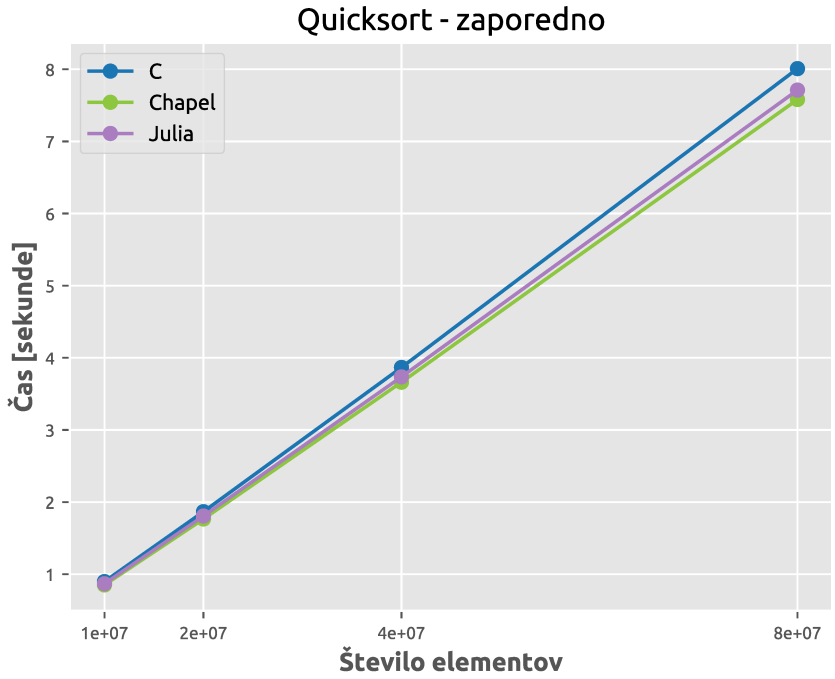
Učinkovitost urejanja je odvisna od porazdelitve elementov v tabeli, ki jo želimo urediti. Zaradi tega nismo merili učinkovitosti urejanja na eni sami vhodni tabeli za različne velikosti tabele. Za vsako velikost generiramo 100 naključnih tabel in izmerimo čas izvajanja na vsaki. Končni rezultat je povprečje vseh izvajalnih časov. Za naključni generator smo izbrali linearni kongruenčni generator določen z enačbo

$$X_{n+1} = (X_n * 1103515245 + 12345) \bmod \text{RANDMAX},$$

kjer je RANDMAX maksimalna vrednost naključnih števil.

Za zaporedno implementacijo urejanja z vzorčenjem smo izbrali Quicksort. Izvajalni časi implementacij Quicksorta so na sliki 5.5. Vse implemen-

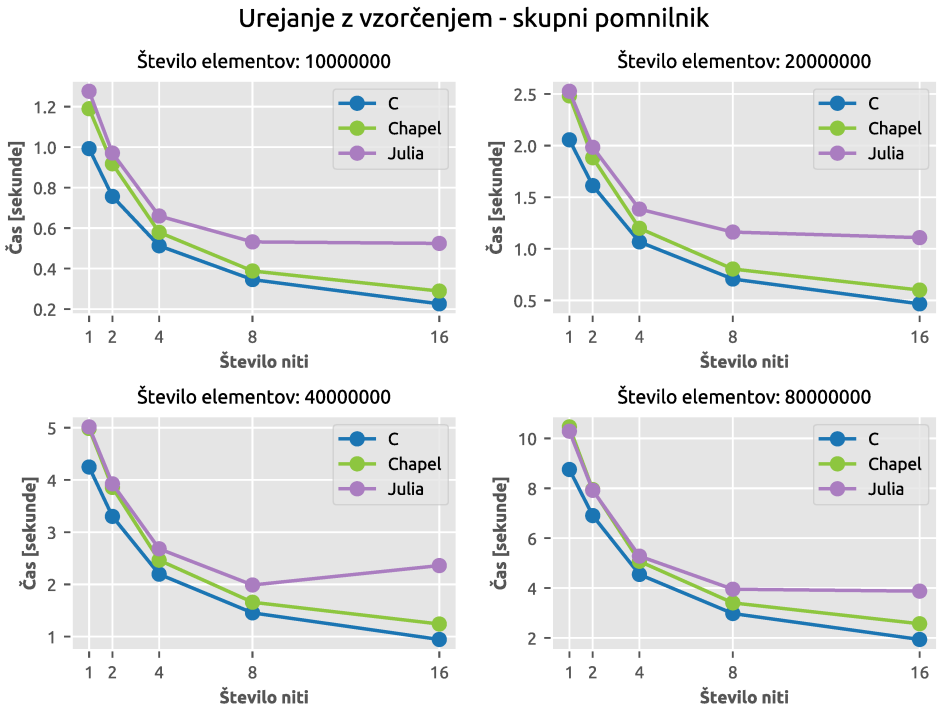
tacije dosegajo približno enake rezultate, a za večje velikosti tabel Julia in Chapel prehitita C.



Slika 5.5: Izvajalni časi implementacij Quicksorta

Izvajalni časi implementacij urejanja z vzorčenjem za skupni pomnilnik so na sliki 5.6. Pri Juliji smo uporabili `@threads` makro za implementacijo na skupnem pomnilniku. Uporaba spremenljivk ujetih v makro ovojnico še vedno velja za počasno operacijo v Juliji [24]. Priporočljivo je premakniti telo zanke v funkcijo in vse potrebne spremenljivke podati preko funkcijskih parametrov. S tem se izognemo dolgotrajnemu branju iz makro ovojnice. Kljub temu še vedno izgubimo določeno mero učinkovitosti, ker dodamo klic funkcije in prenos parametrov funkcije. V našem primeru moramo v funkcijo prenesti veliko tabelo števil. Zato pride do slabših rezultatov Julije v primerjavi s Cjem in Chapelom.

Izvajalni časi implementacij urejanja z vzorčenjem za porazdeljeni po-

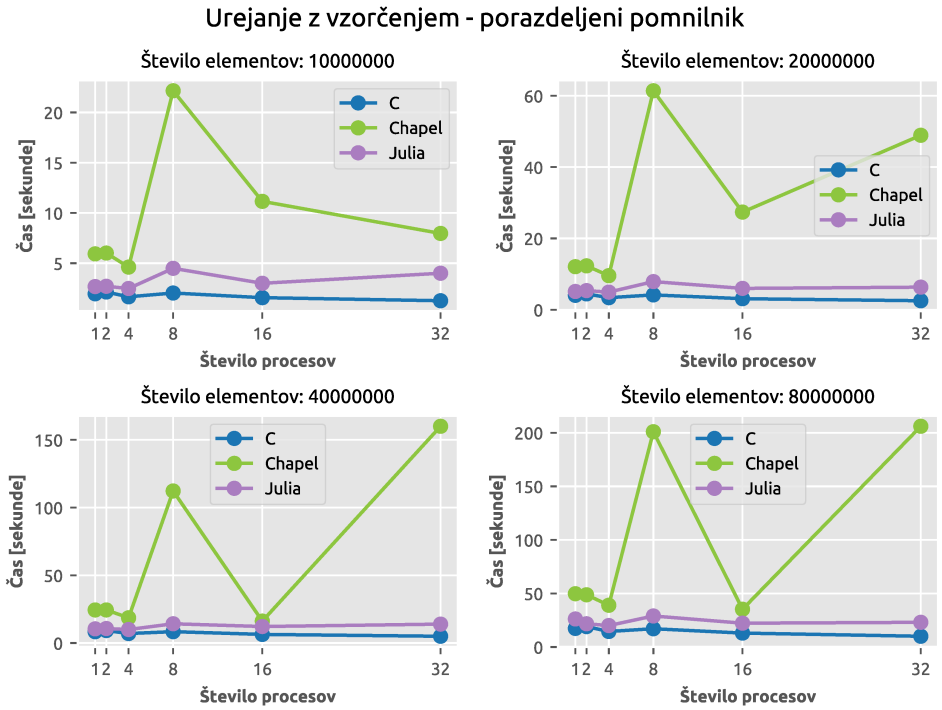


Slika 5.6: Izvajalni časi implementacij urejanja z vzorčenjem za skupni pomnilnik

mnilnik so na sliki 5.7. V Chapelu smo lahko matriko košev enostavno predstavili z vgrajeno `Block` porazdelitvijo. Ukazi, ki ustvarijo matriko košev so v izvorni kodi 5.4. V vrsticah 1 in 2 ustvarimo $m \times m$ matriko, ki vsebuje elemente tipa `locale`. V vrsticah 3 in 4 ustvarimo domeno in ji določimo prostor, ki ga zavzema ter točno določimo strojne vire, ki naj jih uporabi za določene elemente v domeni. V vrstici 5 sledi inicializacija matrike košev s prej ustvarjeno domeno.

Preprosta predstavitev problema v Chapelu se ni prenesla v hiter izvajalni čas. Kot elemente matrike ne uporabimo tabel, ker Chapel ne podpira gnezdenih dinamičnih tabel. Zato moramo dinamične tabele oviti v lasten `BinArray` objekt, ki je dodeljen na kopici. Prenos teh objektov in njihovih tabel je najverjetneje vzrok za anomalije pri Chapelovih izvajalnih časih za

8 in 32 procesov. Standardni način diagnosticiranja težav pri porazdeljenih programih v Chapelu je uporaba `CommDiagnostics` modula. V tem primeru ni izpisal nič, kar bi nas usmerilo bližje rešitvi problema. Za rešitev bi morali pogledati globlje v Chapel prevajalnik in GASNET [10] knjižnico, ki jo Chapel uporablja za komunikacijo.



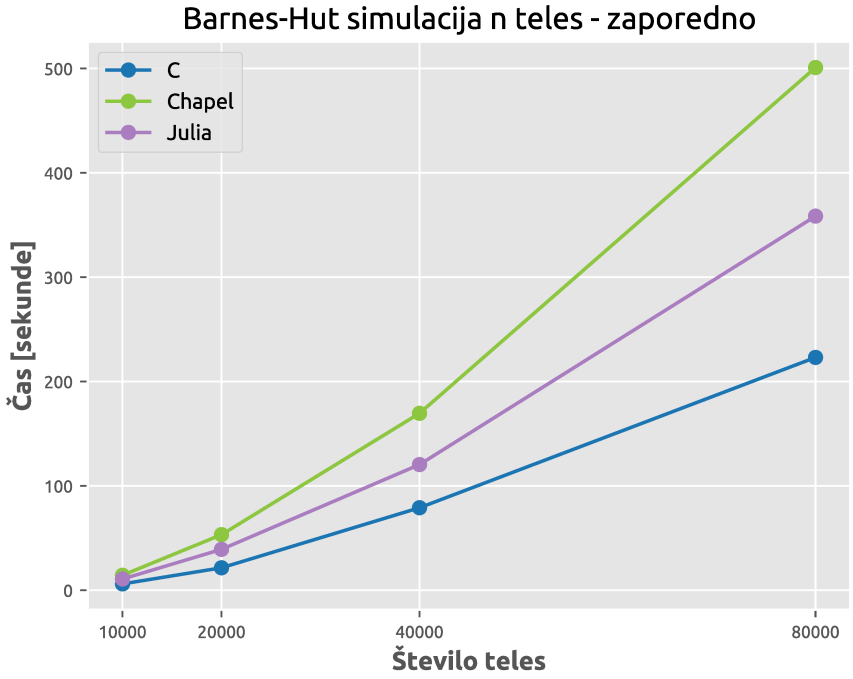
Slika 5.7: Izvajalni časi implementacij urejanja z vzorčenjem za porazdeljeni pomnilnik

Izvorna koda 5.4: Inicializacija matrike košev v Chapelu za porazdeljeni pomnilnik

```
1 var binsLocaleView = {0..0, 0..#m};
2 var binsLocales: [binsLocaleView] locale = reshape(Locales, binsLocaleView);
3 var binsSpace = {0..#m, 0..#m};
4 var binsDomain = binsSpace dmapped Block(boundingBox=binsSpace,
      targetLocales=binsLocales);
5 var bins: [binsDomain] unmanaged BinArray;
```

5.3 Problem n teles

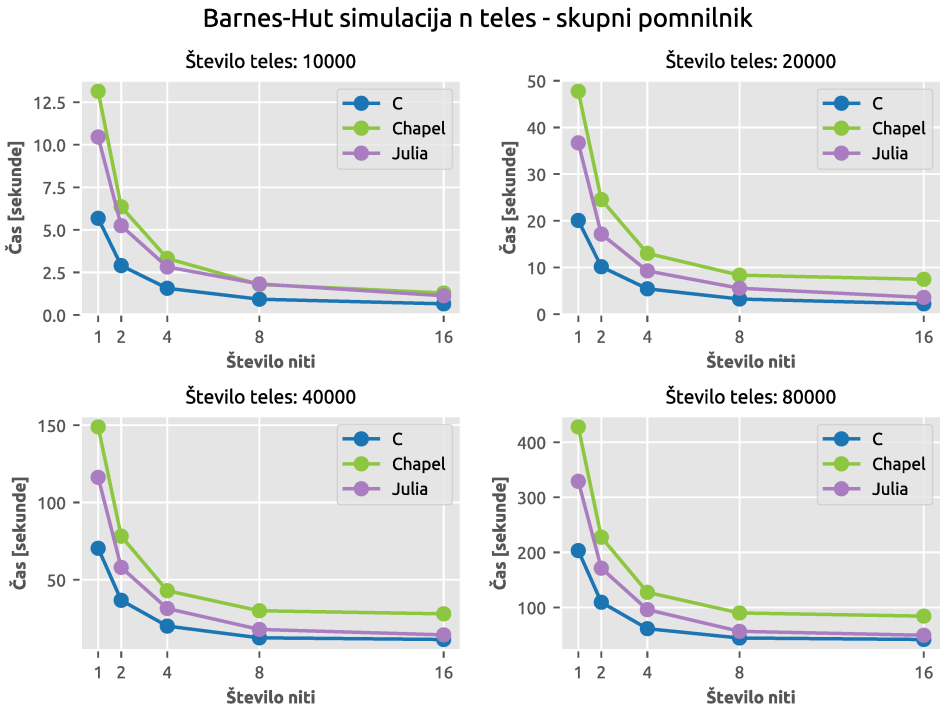
Simulacije problema n teles smo ovrednotili na štirih naključnih zbirkah teles. Telesa so bila enakomerno generirana v vnaprej definiranem tridimenzionalnem prostoru. Izvajalni časi zaporednih implementacij problema n teles so na sliki 5.8.



Slika 5.8: Izvajalni časi zaporednih implementacij problema n teles

Izvajalni časi implementacij problema n teles za skupni pomnilnik so na sliki 5.9. V tem primeru nismo naleteli na težave in algoritem je bil relativno enostavna nadgradnja zaporednega algoritma. Pri 16-ih nitih se je Julia že popolnoma približala C implementaciji. Zanimivo bi bilo preveriti, če bi Julia lahko prehitela C in OpenMP, če bi jima dali še več niti.

Algoritem, ki smo ga izbrali za implementacijo na porazdeljenem pomnilniku je osnovan na SPMD vzporednem modelu. Ko je bil algoritem zasnovan, je bil idealna tarča za implementacijo z MPI-jem. Julia in Chapel nimata



Slika 5.9: Izvajalni časi implementacij problema n teles za skupni pomnilnik

direktne podpore za SPMD model. Zaradi tega smo bili prisiljeni sami implementirati približek SPMD modela z vgrajenimi vzporednimi konstrukti. V obeh primerih na začetku programa zaženemo vse procese z isto funkcijo. Največji problem je predstavljala implementacija `MPI_Allreduce` funkcije, ki jo potrebujemo znotraj `bisection` funkcije v ORB algoritmu. Uporabimo jo za izračun količine dela pod in nad točko bisekcije v vseh sodelujočih procesih.

Pri Juliji smo uporabili oddaljene kanale za komunikacijo med procesi. Chapel ne podpira direktne komunikacije med procesi, zato jo implicitno dosežemo z uporabo globalno porazdeljenih tabel. Uporabili smo sync spremenljivke, da smo preprečili, da bi več procesov pisalo v iste elemente v tabeli. Sync spremenljivke so mešanica navadnih spremenljivk in ključavnic (angl. mutex locks). Vsak osnovni tip v Chapelu lahko postane sync spremenljivka tako, da ji dodamo ključno besedo `sync` ob deklaraciji. Sync spremenljivke

poleg svoje vrednosti vzdržujejo še povezano stanje. Sync spremenljivke so lahko polne ali prazne. Iz sync spremenljivke lahko beremo samo, če je polna (predhodno smo vanjo zapisali vrednost). Proces, ki bere iz prazne spremenljivke mora počakati, da se napolni. Prav tako mora proces, ki piše v polno spremenljivko, počakati da se spremenljivka izprazni. Spremenljivko izpraznimo tako, da proces prebere njeno vrednost.

Chapel ne podpira sync spremenljivk s kompleksnim tipom (objekti, tabele, ipd.). Zato smo morali za kompleksne tipe uvesti dve globalno porazdeljeni tabeli. V prvi tabeli smo hranili dejanske vrednosti, v drugi pa smo imeli preproste sync spremenljivke, ki smo jih uporabljali kot ključavnice za elemente v prvi tabeli. Podroben opis komunikacije s porazdeljenimi globalnimi tabelami in sync spremenljivkami v Chapelu je v dodatku A. Izvajalni časi implementacij problema n teles za porazdeljeni pomnilnik so na sliki 5.10. Za manjše število teles in večje število procesov pričakovano narastejo stroški komunikacije pri Chapelu in Juliji. To je posledica naše implementacije komunikacije med vsemi procesi, ki ni dovolj optimizirana za manjšo količino dela kot pri MPI-ju.

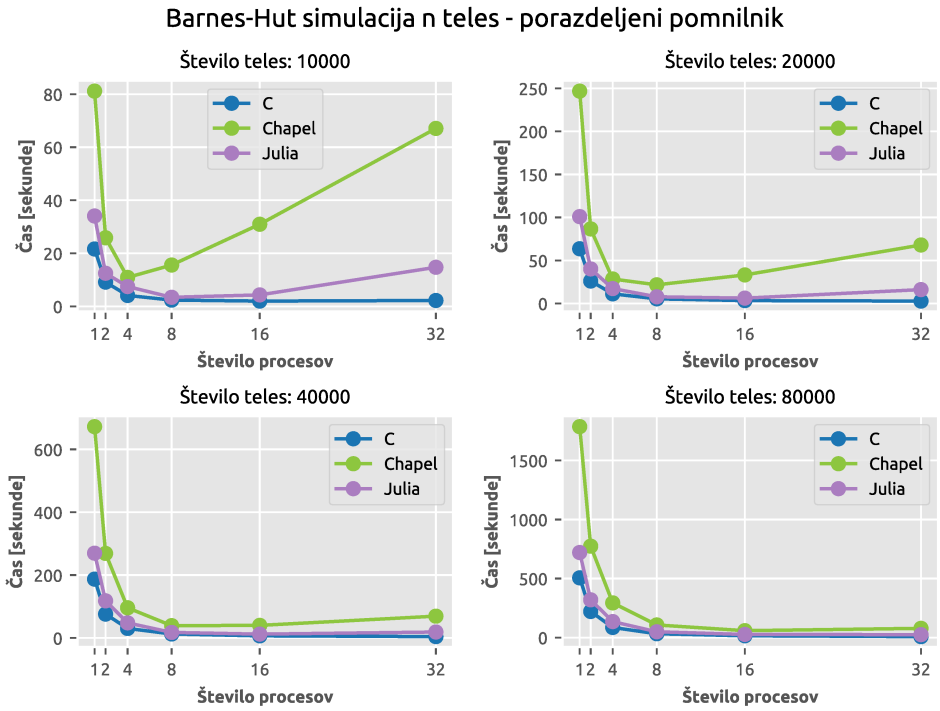
5.4 Programerska izkušnja

Julia ponuja dobro programersko izkušnjo. S sklepanjem o tipih in vgrajenimi standardnimi moduli omogoča hitro prototipiranje. V večini primerov smo z relativno malo kode uspeli sprogramirati velik del funkcionalnosti.

Kljub temu, da je večprocesorska podpora (`@threads` makro in `Threads` modul) označena kot eksperimentalna, smo dobili dobre rezultate. Seveda je potrebno upoštevati omejitve, da moramo oviti notranjost `@threads` makroja v funkcijo in ne smemo uporabljati vhodno-izhodnih operacij znotraj niti.

Oddaljeni kanali ponujajo preprost a močan primitiv za vzporedno programiranje. Še posebno je dobrodošla avtomatska serializacija tipov, ki jih pošiljamo preko kanalov. Pri MPI-ju pa jih moramo serializirati sami.

Problemi nastanejo, ko želimo, da se naša hitro prototipirana koda tudi hi-



Slika 5.10: Izvajalni časi implementacij problema n teles za porazdeljeni pomnilnik

tro izvaja. Takrat je potrebno pregledati Julijino stran z nasveti za učinkovitost [25] in se jih dosledno držati. Najpomembnejša nasveta za nas sta bila pisanje funkcij s stabilnimi tipi in uporaba orodij za merjenje dodeljevanja pomnilnika. Uporabo orodij za merjenje alokacij pomnilnika smo predstavili v podpoglavju 5.1.

Za nestabilne tipe označujejo tiste, ki so presplošni in niso dovolj določeni. Za pisanje funkcij s stabilnimi tipi nam pomaga makro `@code_warntype`. Makro vzame klic funkcije kot argument in preveri tipe v telesu in tip vrednosti, ki jo vrača. Primer uporabe makroja `@code_warntype` je na sliki 5.11. Definirali smo preprosto funkcijo `f` z dvoumnim tipom vrednosti, ki jo vrne. Klic makroja `@code_warntype` izpiše funkcijo v vmesni predstavitvi in vse tipe, ki jih sklepa. Z rdečo pobarva tipe, ki so nestabilni. Nestabilni tipi so proble-

matični zato, ker jih Julia ne more prevesti v optimizirano kodo. V našem primeru Julia sklepa, da funkcija vrača vrednost s tipom `Union{Float64, Int64}` (unija 64 bitne plavajoče vejice in celega števila). Če hoče operirati nad tem tipom, mora vsakič preveriti kateri izmed navedenih tipov je prisoten in izvesti primerno kodo. Zato se moramo izogibati takih funkcij, ker znatno upočasnijo intenzivne dele naše kode. Vrednosti z nestabilnimi tipi popravimo tako, da jim eksplicitno dodamo tipe ob deklaraciji.

```
function f(x)
   if x > 1
        1.0
    else
        0
    end
end
```

```
f (generic function with 1 method)
```

```
@code_warntype f(10)
```

```
Body::Union{Float64, Int64}
│╷╷ >2 1 — %1 = (Base.slt_int)(1, x)::Bool
│    └──      goto #3 if not %1
│  3 2 —      return 1.0
│  5 3 —      return 0
```

Slika 5.11: Primer uporabe makroja `@code_warntype`

V izvorni kodi 5.5 smo pripravili primerjavo funkcij s stabilnimi in nestabilnimi tipi v Juliji. Funkcijo iz slike 5.11 smo preimenovali v `f_unstable` in jo uporabili znotraj nove funkcije `g_unstable`. Ker je funkcija `f_unstable` nestabilna, bo funkcija `g_unstable` morala zgenerirati ukaze za prištevanje celih števil in števil s plavajočo vejico.

Za testiranje smo dodali funkcijo `get_random_array`, ki vrne tabelo, v kateri je približno polovica celih števil in polovica števil s plavajočo vejico. Stabilizirani različici funkcij `f_unstable` in `g_unstable` sta `f_stable` in `g_stable`. Pri obeh naštejemo tipe argumentov in tipe vrednosti, ki jih vračata. Za funkcijo `g_stable` definiramo dve implementaciji za oba tipa

vrednosti v tabeli. S tem izkoristimo večkratno razpošiljanje v Juliji.

Učinkovitost preverimo tako, da na vsakem elementu testne tabele pokličemo funkciji `g_unstable` in `g_stable`. V ta namen definiramo funkcijo za testiranje nestabilnih funkcij `test_unstable` in funkcijo za testiranje stabilnih funkcij `test_stable`. S pomočjo makroja `@benchmark` makroja iz modula `BenchmarkTools` 100-krat izmerimo čas izvajanja funkcij in povprečimo rezultate. Povprečni čas izvajanja nestabilnih funkcij je 1100 ms in čas izvajanja stabilnih funkcij je 630 ms. Z dodajanjem tipov funkcijam in spremenljivkam izgubimo nekaj programerskega časa, ampak se nam povrne v obliki krajših izvajalnih časov. Zaradi tipov moramo tudi bolje razmisliti o problemu, ki ga želimo rešiti in s tem dosežemo globlje razumevanje problema.

Chapel ponuja širok nabor vzporednih zmožnosti za večino vzporednih aplikacij. Razvijalci Chapela se trudijo dodati čim več vzporednih zmožnosti in se trenutno ne ukvarjajo z njihovo učinkovitostjo. Osnova vseh porazdeljenih implementacij v Chapelu je domena. Domene so dobra abstrakcija za mnogo vzporednih problemov, ki so osnovani na porazdeljenih tabelah. Prav tako nam Chapel omogoča, da se ne ukvarjamo s komunikacijo podatkov in se lahko osredotočimo na ključne dele naših implementacij. Ugotovili smo, da to drži za manjše programe in za manjšo količino podatkov. Primer uporabe domen na regularnih vzporednih problemih smo prikazali v razdelku 3.3.2.

Pri večjih programih so se začeli pojavljati dolgi časi prevajanja. Primerjava prevajalnih časov za porazdeljene implementacije med C in MPI ter Chapel so v tabeli 5.1. Dolgi prevajalni časi pri Chapelu onemogočajo hitre iteracije razvijanja, iskanje napak in preverjanje rezultatov.

Velike težave je predstavljalo lokalno testiranje porazdeljenih implementacij. Chapel ima le osnovne zmožnosti za lokalno testiranje in ne zagotavlja učinkovitosti za lokalne porazdeljene programe. Zato smo lokalno preverili samo pravilnost programa in šele nato na oddaljenem večračunalniškem sistemu preverili učinkovitost in hitrost izvajanja. Če smo ugotovili, da imamo težave z učinkovitostjo, smo morali raziskati problem na oddaljenem sistemu

Izvorna koda 5.5: Primerjava stabilnih in nestabilnih funkcij v Juliji

```
using Random; Random.seed!(0)

function get_random_array()
    [rand() < 0.5 ? rand() * 2 - 1 : trunc(Int64, rand() * 2 - 1)
     for _ in 1:10000000]
end
function f_unstable(x)
    if x > 1
        1.0
    else
        0
    end
end
function g_unstable(y)
    f_unstable(y) + 10
end
function test_unstable()
    g_unstable.(get_random_array())
end

function f_stable(x::Float64)::Float64
    if x > 1
        1.0
    else
        0
    end
end
function g_stable(y::Float64)::Float64
    f_stable(y) + 10
end
function g_stable(y::Int64)::Float64
    f_stable(convert(Float64, y)) + 10
end
function test_stable()
    g_stable.(get_random_array())
end
```

Porazdeljena implementacija	Čas prevajanja Chapel (sekunde)	Čas prevajanja C/MPI (sekunde)
Metoda voditeljev	12.75	0.18
Urejanje z vzorčenjem	12.30	0.22
Problem n teles	18.42	0.68

Tabela 5.1: Primerjava prevajalnih časov za porazdeljene implementacije med C in MPI ter Chapel

ali pa lokalno ugibati kje je problem. Med razvijanjem naših implementacij smo našli tudi dva problema v samem prevajalniku [23, 22]. Čeprav naj bi bil Chapel stabilen, nas to ni navdalo z zaupanjem in smo morali ob napakah dvakrat preveriti ali je težava v prevajalniku ali v naši implementaciji.

Poglavje 6

Zaključek

V magistrski nalogi smo predstavili vzporedna programska jezika Chapel in Julijo. Ovrednotili smo ju kot ustrezen nadomestek vzporednim ogrodjem, predvsem za OpenMP in MPI. Na začetku smo opisali oba jezika in predstavili njunine vzporedne zmožnosti. Izbrali smo tri probleme, na katerih smo ovrednotili jezike. Za vsak problem smo predstavili njegovo teoretično ozadje, zaporedni algoritem, algoritem za skupni pomnilnik in algoritem za porazdeljeni pomnilnik. Vse algoritme smo implementirali v jezikih Chapel, Julia in C z vzporednima ogrodjema OpenMP in MPI. Sledila je analiza izvajalnih časov in programerske izkušnje. Analizo in zaključke smo v strnjeni obliki predstavili na konferenci SLATE 2019 (8. simpozij o jezikih, aplikacijah in tehnologijah v Coimbri na Portugalskem) v članku [37].

V splošnem smo imeli malo težav z našimi implementacijami v programskem jeziku C. Ko smo odpravili vse napake, so te implementacije potrebovale le majhne popravke, da so tekle učinkovito. V nasprotju s tem je bilo na začetku implementacije Chapela in Julija lažje napisati, vendar so zahtevale veliko izboljšav in naknadnih prilagoditev, da smo dosegli učinkovitost Cja. Julia ponuja majhen, a zmogljiv nabor vzporednih zmožnosti. Podpora za več niti je še vedno označena za eksperimentalno, venadr smo z njo uspeli dobiti dobre rezultate. Razvijalci Julije že načrtujejo zamenjavo trenutnega modula za nitenje z modulom osnovanim na globinskem vzporednem

razporejanju nalog [13]. Glavna problema z Julijo sta bila nepričakovano dodeljevanje pomnilnika in nestabilni tipi.

Chapel nudi širok spekter vzporednih zmožnosti. Lahko služi kot odličen uvodni jezik za tečaje vzporednega in porazdeljenega programiranja. Vsebuje dovolj visokonivojskih zmožnosti iz ostalih jezikov, da novi programerji nimajo težav s spoznavanjem jezika. Vzporedne koncepte lahko demonstriramo na preprost način z vgrajenimi konstrukti kot so domene, sync spremenljivke, vzporedni iteratorji, itd. Chapel je še vedno v aktivnem razvoju, saj razvijalci skušajo uvesti čim več vzporednih zmožnosti. Zato izvajalni časi Chapel programov trenutno niso tako pomembni za razvijalce. To je glavni razlog zakaj zaostaja za Cjem in Julijo pri naših meritvah.

Zavedamo se, da bi dokončna primerjava jezikov za vzporedno programiranje zahtevalo naključno kontrolirano preskušanje kot je navedeno v [47].

Izmed ovrednotenih jezikov bi v prihodnosti za splošno vzporedno programiranje uporabili Julijo. Pokazali smo, da lahko z nekaj truda tekmuje s C implementacijami in ima izmed ovrednotenih jezikov najboljšo programersko izkušnjo. Chapel bi uporabili za regularne vzporedne probleme, kot je prevajanja toplote v eni dimenziji v razdelku 3.3.2, kjer je uporaba domen najbolj optimalna.

Dodatek A

Vzorec komunikacije vsak z vsakim v Chapelu

Pri porazdeljeni implementaciji problema n teles smo v večih primerih imeli množico procesov, kjer bi vsak proces rad poslal svojo vrednost vsem ostalim. Ker Chapel ne omogoča direktne komunikacije med procesi, smo morali ustvariti lastno implementacijo za tak vzorec komunikacije.

Primer uporabe vzorca je v izvorni kodi A.1. V vrsticah 1 in 2 ustvarimo globalno porazdeljeno domeno s toliko elementi, kolikor je vzporednih procesov. V vrstici 3 ustvarimo globalno porazdeljeno tabelo, v kateri shranjujemo vrednosti, ki jih želimo poslati. V vrstici 4 ustvarimo globalno porazdeljeno tabelo s sync elementi, v kateri bomo shranjevali identifikatorje prejemnikov. Navada v Chapelu je, da sync spremenljivkam dodamo `$` na konec imena. Identifikator trenutnega procesa shranimo v vrstici 5. Nato ustvarimo zanko, v kateri bomo poslali vrednost trenutnega procesa vsem ostalim procesom. V vrsticah 8 in 9 izračunamo identifikatorje prejemnika in pošiljatelja (proces, ki bo trenutnemu procesu poslal svojo vrednost). V vrsticah od 10 do 12 počakamo, da je prejšnja vrednost prebrana. V vrstici 13 nastavimo vrednost, ki bi jo radi poslali in v vrstici 14 zapišemo željenega prejemnika. V vrstici 15 mora trenutni proces počakati, dokler mu njegov pošiljatelj ne pošlje vrednosti. Funkcija `readFF` prebere vrednost sync spremenljivke brez

da bi jo izpraznila. Ko pošiljatelj zapiše vrednost namenjeno trenutnemu procesu, jo lahko prebere. Ko uporabi vrednost, prebere pošiljateljevo sync spremenljivko in mu s tem naznani, da je končal z njegovo vrednostjo. Kot zadnji korak mora trenutni proces še ponastaviti željenega pošiljatelja.

Izvorna koda A.1: Vzorec komunikacije vsak z vsakim v Chapelu

```
1  var localeSpace = {0..#numLocales};
2  var localeDomain = localeSpace dmapped Block(boundingBox=localeSpace);
3  var arr: [localeDomain] SomeType;
4  var arr$: [localeDomain] sync int;
5  var rank = here.id;
6
7  for i in 0..#numLocales-1 {
8      var receiver = (rank + i + 1) % numLocales;
9      var sender = ((rank - (i + 1)) + numLocales) % numLocales;
10     if (arr$[rank].isFull) {
11         arr$[rank];
12     }
13     arr[rank] = valueToSend;
14     arr$[rank] = receiver;
15     while (arr$[sender].readFF() != rank) {}
16     var senderValue = arr[sender];
17     // ... uporabi vrednost
18     arr$[sender];
19     arr$[rank] = -1;
20 }
```

Literatura

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund *et al.*, "The Fortress language specification," http://web.cs.ucla.edu/~palsberg/course/cs239/papers/fortress1.0beta.pdf, dostopano: 2019-05-12.

[2] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, in S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, str. 81–89.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, in D. Sorensen, LAPACK *Users' Guide*, 3. izdaja. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[4] D. Arthur in S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, str. 1027–1035.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Tech. Rep., 2006.

[6] H. E. Bal, "A comparative study of five parallel programming languages," *Future Generation Computer Systems*, zv. 8, št. 1-3, str. 121–135, 1992.

[7] P. Balaji, *Programming Models for Parallel Computing*. The MIT Press, 2015, poglavje 6, str. 129–159.

[8] J. Barnes in P. Hut, "A hierarchical O($n \log n$) force-calculation algorithm," *Nature*, zv. 324, št. 6096, str. 446, 1986.

[9] J. Bezanson, A. Edelman, S. Karpinski, in V. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, zv. 59, št. 1, str. 65–98, 2017.

[10] D. Bonachea in P. Hargrove, "GASNet specification, v1.8.1," https://gasnet.lbl.gov/dist/docs/gasnet.html, 2017.

[11] B. Chamberlain, D. Callahan, in H. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, zv. 21, št. 3, str. 291–312, 2007.

[12] B. L. Chamberlain, *The design and implementation of a region-based parallel programming language*. University of Washington, 2001.

[13] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry *et al.*, "Scheduling threads for constructive cache sharing on cmps," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2007, str. 105–115.

[14] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, in A. Y. Ng, "Map-reduce for machine learning on multicore," in *Advances in neural information processing systems*, 2007, str. 281–288.

[15] L. Dagum in R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, zv. 5, št. 1, str. 46–55, 1998.

[16] J. Dean in S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, zv. 51, št. 1, str. 107–113, 2008.

[17] J. T. Feo, *Comparative Study of Parallel Programming Languages: The Salishan Problems*. New York, NY, USA: Elsevier Science Inc., 1992.

[18] M. P. Forum, "MPI: A message-passing interface standard," University of Tennessee, Tech. Rep., 1994.

[19] W. D. Frazer in A. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM)*, zv. 17, št. 3, str. 496–507, 1970.

[20] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, 1985.

[21] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, in Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[22] C. P. Language, "Chapel issue: Using with clause in coforall loop for distributed k-means," https://github.com/chapel-lang/chapel/issues/12006, dostopano: 2019-04-24.

[23] ——, "Chapel issue: Performance issues when running n-body simulation," https://github.com/chapel-lang/chapel/issues/11333, dostopano: 2019-04-24.

[24] J. P. Language, "Julia issue: performance of captured variables in closures," https://github.com/JuliaLang/julia/issues/15276, dostopano: 2019-04-24.

[25] ——, "Julia performance tips," https://docs.julialang.org/en/v1/manual/performance-tips/index.html, dostopano: 2019-04-22.

[26] R. P. Language, "Rust influences," https://doc.rust-lang.org/reference/influences.html, dostopano: 2019-07-05.

[27] C. Lattner in V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, str. 75.

[28] T. Lindholm, F. Yellin, G. Bracha, in A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, in M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[30] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, zv. 28, št. 2, str. 129–137, 1982.

[31] D. Loghin in Y. M. Teo, "The energy efficiency of modern multicore systems," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ser. ICPP '18. New York, NY, USA: ACM, 2018, str. 28:1–28:10.

[32] N. D. Matsakis in F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, zv. 34. ACM, 2014, str. 103–104.

[33] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, zv. 19, št. 2, str. 129–147, 1982.

[34] G. E. Moore, "Cramming more components onto integrated circuits," *IEEE Solid-State Circuits Society Newsletter*, zv. 11, št. 3, str. 33–35, Sep. 2006.

[35] S. Nanz, S. West, in K. S. Da Silveira, "Examining the expert gap in parallel programming," in *European Conference on Parallel Processing*. Springer, 2013, str. 434–445.

[36] J. Nickolls, I. Buck, M. Garland, in K. Skadron, "Scalable parallel programming with cuda," *Queue*, zv. 6, št. 2, str. 40–53, Mar. 2008.

[37] R. Novosel in B. Slivnik, "Beyond classical parallel programming frameworks: Chapel vs julia," in *8th Symposium on Languages, Applications and Technologies (SLATE 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[38] L. Nylons, M. Harris, in J. Prins, "Fast n-body simulation with CUDA," https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf.

[39] T. E. Oliphant, *Guide to NumPy*, 2. izdaja. USA: CreateSpace Independent Publishing Platform, 2015.

[40] J. Pintar, "Programski jeziki za vzporedno programiranje," 2019, Univerza v Ljubljani.

[41] J. Regier, K. Pamnany, K. Fischer, A. Noack, M. Lam, J. Revels, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. Thomas, in Prabhat, "Cataloging the visible universe through bayesian inference at petascale," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, str. 44–53.

[42] C. Robert in G. Casella, *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[43] J. K. Salmon, "Parallel hierarchical n-body methods," Ph.D. dissertation, California Institute of Technology, 1991.

[44] A. Silberschatz, G. Gagne, in P. B. Galvin, *Operating system concepts*. Wiley, 2018.

[45] J. P. Singh, W.-D. Weber, in A. Gupta, "Splash: Stanford parallel applications for shared-memory," *ACM SIGARCH Computer Architecture News*, zv. 20, št. 1, str. 5–44, 1992.

[46] G. Steele, "Fortress features and lessons learned (2016)," https://www.youtube.com/watch?v=EZD3Scuv02g, dostopano: 2019-04-22.

[47] A. Stefik in S. Hanenberg, "Methodological Irregularities in Programming-Language Research," *Computer*, zv. 50, št. 8, str. 60–63, 2017.

[48] J. E. Stone, D. Gohara, in G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, zv. 12, št. 3, str. 66, 2010.

[49] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, str. 202–210, 2005.

[50] P. Tang, "Multi-core parallel programming in Go," in *Proceedings of the First International Conference on Advanced Computing and Communications*, 2010, str. 64–69.

Beyond Classical Parallel Programming Frameworks: Chapel vs Julia

Rok Novosel
Faculty of Computer and Information Science, University of Ljubljana
Večna pot 113, 1000 Ljubljana, Slovenia
novosel.rok@gmail.com

Boštjan Slivnik[1]
Faculty of Computer and Information Science, University of Ljubljana
Večna pot 113, 1000 Ljubljana, Slovenia
bostjan.slivnik@fri.uni-lj.si

Abstract

Although parallel programming languages have existed for decades, (scientific) parallel programming is still dominated by Fortran and C/C++ augmented with parallel programming frameworks, e.g., MPI, OpenMP, OpenCL and CUDA. This paper contains a comparative study of Chapel and Julia, two languages quite different from one another as well as from Fortran and C, in regard to parallel programming on distributed and shared memory computers. The study is carried out using test cases that expose the need for different approaches to parallel programming. Test cases are implemented in Chapel and Julia, and in C augmented with MPI and OpenMP. It is shown that both languages, Chapel and Julia, represent a viable alternative to Fortran and C/C++ augmented with parallel programming frameworks: the programmer's efficiency is considerably improved while the speed of programs is not significantly affected.

2012 ACM Subject Classification Computing methodologies → Parallel programming languages

Keywords and phrases parallel programming languages, Chapel, Julia

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.12

Supplement Material The sources are at `https://github.com/novoselrok/parallel-algorithms`.

Acknowledgements The authors want to thank Janez Pintar for valuable discussions at the beginning of this work.

1 Introduction

To implement a parallel algorithm or to write a parallel application, most programmers would use Fortran or C/C++ and a parallel programming framework that best suits the target parallel computer's architecture. Hence, MPI and OpenMP would be used for programs designed to run on distributed and shared memory computers, respectively, or OpenCL/CUDA if the computation must run on a GPU. In 2019 this remains de facto approach to parallel programming even though parallel programming languages have been around for decades.

Among parallel programming languages, Fortran stands out as in its 2018 version it includes a wide range of constructs supporting data parallelism and concurrency. Otherwise, many languages died away, e.g., SISAL, ZPL and Fortress, or faded into obscurity, e.g., X10. Nevertheless, it has always been claimed that languages supporting parallel programming will one day boost parallel application development [8], and were therefore studied and analyzed [6, 16].

[1] The corresponing author.


© Rok Novosel and Boštjan Slivnik;
licensed under Creative Commons License CC-BY

In this paper writing parallel programs in two programming languages, namely Chapel and Julia, is considered. Chapel has been developed at Cray, a traditional supercomputer manufacturer, and has been intended for increasing supercomputer productivity and parallel programming from the start. With its explicit support for declaring a topology of processor unit the program is to be run on it is a border case of a domain-specific language. Julia is rather different. Designed for programming high-performance computational science, it is definitely a general programming language. It is a dynamic language with garbage collection and uses just-in-time compilation. Hence, a list of characteristics usually not associated with the fastest possible execution.

Chapel and Julia are compared one against the other and against the combination of C and a selected parallel programming framework. The comparison is performed by implementing the algorithms for using three selected problems (Section 2) and evaluating the programming process and the final result (Section 3). Although this approach is far from new [12, 4], it has been called for more studies like this as no better methodology for comparing programming languages is available today [26].

2 Test Cases

2.1 *k*-Means Clustering

The k-means algorithm [21], also known as Lloyd's algorithm, is a method of finding clusters in a dataset. Its aim is to organize n points in d-dimensional space into k clusters: in the end, each point should belong to the cluster with the nearest mean (centroid) according to the Euclidean distance. The algorithm starts by creating initial clusters and then iteratively updating them. There are multiple ways of initializing the clusters: despite sophisticated methods like kmeans++ [2], we choose random points from the dataset to serve as initial clusters. Each update performs two steps:

1. Compute the centroid of a cluster.
2. Reassign points to be in the cluster with the nearest centroid.

The iterative procedure ends when no points exchange clusters, or it reaches a maximum iteration limit. Fig. 1 illustrates the iterations of the k-means algorithms where $n = 1000$ and $k = 3$.

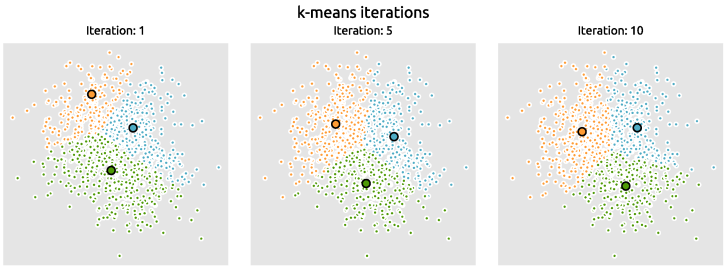


Figure 1 Iterations of the k-means clustering where $n = 1000$ and $k = 3$.

The algorithm is parallelized by distributing the points among the parallel workers and then independently calculating the nearest cluster. We compute global centroids by combining the local centroids of each worker. We decided to parallelize the k-means algorithm since it serves as an example of the map-reduce pattern [11]. It can be generalized to other machine learning algorithms that can be parallelized using the same pattern [10].

2.2 Samplesort

Sorting is a ubiquitous operation in computer science that has to handle large quantities of data in a short amount of time. That is why efficient parallel implementations are necessary. Samplesort [14] provides a good benchmark for testing whether new parallel languages are able to handle the mentioned constraints.

Samplesort is a divide-and-conquer sorting algorithm. It addresses the limitations of the Quicksort algorithm by allowing more than one partitioning element. Partitioning elements are determined by sampling the elements of the input array. Samplesort partitions the elements into m bins by constructing a $m \times m$ matrix where m is the number of parallel workers. Each column in the matrix corresponds to a bin. The entire algorithm consists of the following steps:

1. Sample $m - 1$ elements from the input array.
2. Distribute elements among workers.
3. Each worker partitions the local array into m bins according to the selected samples.
4. The bins matrix is re-distributed, and each worker takes ownership of a column.
5. Each worker sorts the elements in its column.
6. Finally, the columns are merged to get the sorted array.

The choice of the actual sorting algorithm in the fifth step is arbitrary. We used the Quicksort algorithm. Fig. 2 demonstrates sorting 20 input elements with three parallel workers. Each of the differently colored rows or columns indicate that a worker can execute them in parallel.

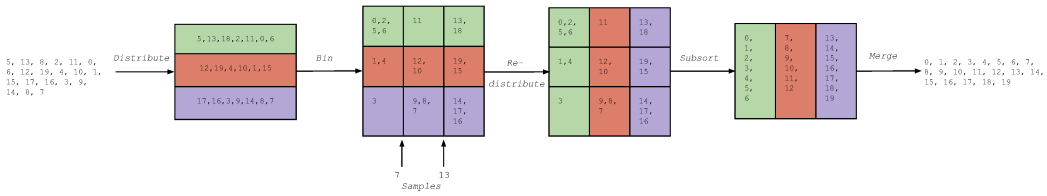


Figure 2 Samplesort with 20 elements and 3 parallel workers.

2.3 *n*-Body Simulation

The n-body problem requires that positions and velocities of n interacting particles is computed, usually in discrete time intervals. It is regularly used when evaluating parallel languages and frameworks [23, 25] as it was classified as a parallel "dwarf" [3]. The solution to the n-body problem is found by simulating the behavior of the particles.

The simplest way of simulating particle behavior is by calculating the effect of all other particles on a specific particle. This results in a $O(n^2)$ algorithm, which is computationally expensive for a large number of particles. The key to lowering the computational cost is to group nearby bodies and treat them as a single body. If it is far enough, we can approximate the gravitational effect of the group by using the center of its mass. The Barnes-Hut algorithm [5] achieves this by using the octree data structure [22] to split the particle space into cubic cells. Once the octree is constructed, the algorithm traverses the tree structure for each body and calculates the gravitational effect of each node. If the bodies contained in the node sub-tree are sufficiently far away, the effect is approximated by their center of mass. Otherwise, the algorithm reaches the leaf nodes which contain the original bodies. In

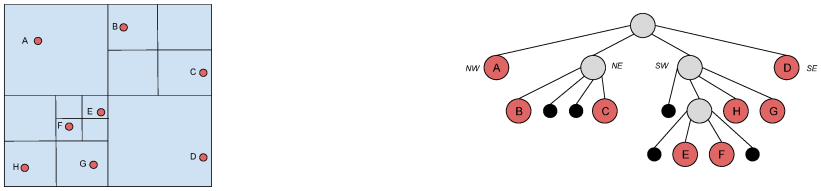


Figure 3 A 2-dimensional representation of the particle space and the corresponding Barnes-Hut quadtree: particle space divided into quadrants (left) and Barnes-Hut quadtree (right).

Fig. 3(a), we used 2-dimensional particle space to simplify the explanation. A 2-dimensional particle space is divided into quadrants and the Barnes-Hut algorithm constructs a quadtree as illustrated in Fig. 3(b).

We used different parallelization strategies for shared and distributed memory implementations. For shared memory, we statically divided the 3rd level of the octree (containing 64 nodes) among the threads. Each thread computes the necessary subtrees and the main thread combines the subtrees into the Barnes-Hut octree. For the distributed implementation we followed the algorithm outlined in [24]. Its main advantage is that it can balance the amount of work for each worker using orthogonal recursive bisection [13]. The algorithm works as follows:

1. Decompose the particle space using orthogonal recursive bisection so that each cell contains equal amount of work.
2. Each worker owns one cell and the particles within it.
3. Each worker builds a local octree from the particles (locally essential tree).
4. Next, it sends its local octree nodes to any other worker that might need them.
5. It receives the necessary octree nodes from other workers and inserts them into his octree.
6. Each worker can now compute the positions and velocities independently.

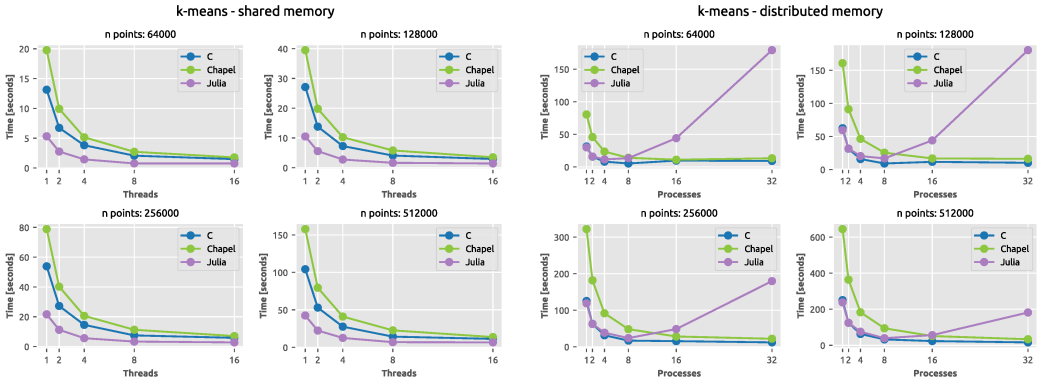
3 Evaluation

We evaluated the parallel capabilities of Chapel and Julia by implementing the test cases from the previous section. For each test case we implemented an algorithm targeted for a shared and a distributed memory computer. We used the programming language C with OpenMP for shared memory and MPI for distributed memory as the baseline for comparison. Since Julia is using JIT compilation, we also discarded the first two execution times to properly "warm-up" the JIT compiler.

We used Chapel version 1.18 and Julia version 1.1. For shared memory, we used Ubuntu 18.04, AMD Ryzen 7 2700X (8 cores, 16 threads, and 3.7GHz) processor and 16GB of RAM. For distributed memory, we used 32 HP DL160 G6 servers, each running Ubuntu Server 11.04 with Intel Xeon 5520 processor and 6GB of RAM.

3.1 *k*-Means Clustering

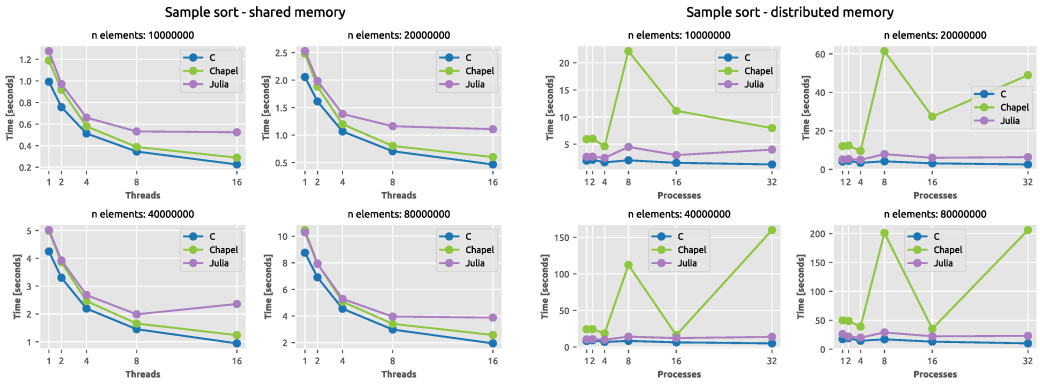
We generated 4 datasets for evaluation purposes. Each dataset contains 256 clusters with 128-dimensional points. The results in Fig. 4 indicate that the performance bottleneck in the k-means clustering is the computation of the Euclidean distance. Julia has a highly optimized linear algebra module based on the LAPACK library [1]. This is the main reason why we were able to outperform the C shared-memory implementation. The Julia distributed



(a) k-means clustering benchmark for the shared memory implementation.

(b) k-means clustering benchmark for the distributed memory implementation.

Figure 4 k-means clustering benchmarks.



(a) Samplesort benchmark for the shared memory implementation.

(b) Samplesort benchmark for the distributed memory implementation.

Figure 5 Samplesort parallel benchmarks.

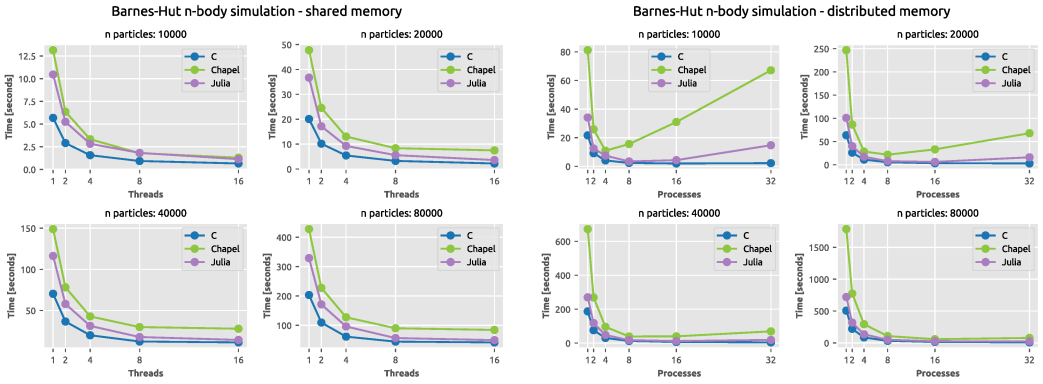
memory implementation suffered from a significant overhead when spawning remote processes and communicating through remote channels. Based on Hoare's CSP [15], a channel is a FIFO data structure enabling remote processes to send and receive data.

Increasing the number of processes only exacerbated the problem. To eliminate the overhead we would have to benchmark a much larger dataset.

3.2 Samplesort

Sorting efficiency depends highly on the input elements themselves. Therefore, we could not pre-generate the input arrays. For each input size, we generated 100 random arrays and measured the execution time for each array. The final result was the average of all the execution times. The results are in Fig. 5.

In Julia, we add multi-threading by prefixing a for loop with the `@threads` macro. The macro wraps the body of the for loop in a closure and splits the iterations between the available threads. Using variables captured in a macro closure is a performance concern and is still an open issue in Julia [19]. The recommended solution, for now, is to extract the body



(a) Barnes-Hut n-body simulation benchmark for the shared memory implementation.

(b) Barnes-Hut n-body simulation benchmark for the distributed memory implementation.

Figure 6 Barnes-Hut n-body simulation benchmarks.

of the for loop in a separate function and thus bypass the macro closure. It is not a perfect solution and that is why we still see a deviation from the C and Chapel results in shared memory implementations.

The Chapel distributed memory implementation caused the most issues. The standard way of diagnosing distributed implementations is by using the `CommDiagnostics` module, which outputs all the implicit Chapel communication. It did not output any unnecessary or unintended communication. Our reasoning is that the Chapel compiler and the underlying GASNET [7] communication library are not able to optimize the transfer of large arrays during sorting. In spite of the mentioned issues, we found that using Chapel domains we were able to elegantly solve the samplesort problem.

3.3 *n*-Body Simulation

For the n-body simulation, we generated equally distributed particles in a pre-defined cube. The results are in Fig. 6. In the shared memory implementations we observed consistent results for all test cases. The execution times of C with OpenMP and Julia both running with 16 threads are almost equal. It would be interesting to see if Julia could outperform C and OpenMP if both are given more threads.

The algorithm we chose for the distributed memory implementations was designed for the SPMD (single program, multiple data) parallel model. As such, MPI was the ideal target when the algorithm was designed. With Julia and Chapel, we were forced to replicate the SPMD behavior. In Julia, it was relatively easy to replicate the SPMD model using channels. On the contrary, Chapel does not have a way for processes to directly communicate with each other. All communication has to be done through distributed global arrays. We used sync variables to prevent multiple processes from altering the same element in the array. Sync variables are roughly analogous to mutex locks in other languages with the exception that a sync variable is itself a lock. Any basic type (integer, float, etc.) can become a sync type by prefixing it with the `sync` keyword. Chapel does not support sync variables on complex types like arrays or objects. For complex types, we had to create two arrays one containing values and one containing sync variables. Considering the orchestration behind the Chapel implementation it is not surprising that for a small amount of data and a large number of processes it does not perform well. Increasing the amount of data hides the overhead from sync variable contention.

It is also important to note that both Julia and Chapel do not support fixed-sized arrays out-of-the-box. Using regular (dynamic) arrays was a huge performance bottleneck in the n-body simulation and the k-means clustering implementations. We used the `StaticArrays` package in Julia but resorted to using tuples in Chapel. Further research is necessary to design n-body simulation algorithms that would better fit Julia and Chapel.

4 Conclusion

In general, we had little to no issues with our C implementations. Once we eliminated all segmentation faults C implementations needed only minor improvements to run efficiently. In contrast, Chapel and Julia implementations were easier to write initially but required a lot of efficiency improvements and fine-tuning afterwards.

Julia provides a small but powerful set of parallel features. The multi-threading module is still marked as experimental, but we were still able to get good results with it. The Julia development team is planning to replace the current multi-threading module with an implementation based on parallel depth-first scheduling [9]. The main two issues with Julia were type instability and unexpected memory allocations. If Julia is not able to infer the type of variable it resorts to runtime type checking which adds significant overhead. The solution was to always annotate all variables with types to avoid runtime type checking. The Julia standard library contains multiple macros for code inspection and benchmarking. Examples include `@code_warntype` which outputs variables with ambiguous types and `@time` which outputs execution time and total memory allocated. Julia provides a web page [20] containing performance tips that have to be followed rigorously if performance is an issue.

Chapel provides a broad spectrum of parallel functionality. It can serve as an excellent introductory language for parallel and distributed programming courses. It includes enough the high-level features from other languages so that it immediately feels familiar to novices. Parallel concepts can be easily demonstrated using built-in constructs like domains, sync variables, parallel iterators, etc. Chapel itself is still under heavy development. Developers are currently trying to implement as many parallel features as possible. Consequently, the performance of Chapel programs is not as important right now. That causes it to lag behind C and Julia in our benchmark tests. During the development of this paper, we also found multiple bugs and performance issues in the compiler [17, 18]. We also experienced that the Chapel compiler is fairly slow when compiling large applications. As an example, our distributed n-body simulation takes roughly 1 minute to compile. During the implementation process, this prevented us from quickly iterating and examining new solutions.

The authors realize that the definitive comparison of languages for parallel computing would require a randomized controlled trial as advocated in [26]. Nevertheless, the results presented here can be understood as a justification for carrying out such experiment.

References

1 E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK *Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

2 David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

3 Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, 2006.

4 Henri E. Bal. A comparative study of five parallel programming languages. *Future Generation Computer Systems*, 8(1–3):121–135, 1992.

5 Josh Barnes and Piet Hut. A hierarchical O($N \log N$) force-calculation algorithm. *Nature*, 324(6096):446, 1986.

6 A. P. W. Böhm and R. R. Oldehoeft. Two Issues in Parallel Language Design. *ACM Transactions on Programming Languages and Systems*, 16(6):1675–1683, 1994.

7 Dan Bonachea and P Hargrove. GASNet Specification, v1.8.1. `https://gasnet.lbl.gov/dist/docs/gasnet.html`, 2017.

8 Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, pages 66–75, 2004.

9 Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM, 2007.

10 Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

11 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

12 John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. North Holland, New York, NY, USA, 1992.

13 Geoffrey C Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 37–61. Springer, 1988.

14 W Donald Frazer and AC McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

15 Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

16 Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 7–1–7–22, 2007.

17 Chapel Programming Language. Chapel Issue: Performance issues when running n-body simulation. `https://github.com/chapel-lang/chapel/issues/11333`. Accessed: 2019-04-24.

18 Chapel Programming Language. Chapel Issue: Using with clause in coforall loop for distributed k-means. `https://github.com/chapel-lang/chapel/issues/12006`. Accessed: 2019-04-24.

19 Julia Programming Language. Julia Issue: performance of captured variables in closures. `https://github.com/JuliaLang/julia/issues/15276`. Accessed: 2019-04-24.

20 Julia Programming Language. Julia Performance Tips. `https://docs.julialang.org/en/v1/manual/performance-tips/index.html`. Accessed: 2019-04-22.

21 Stuart Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

22 Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

23 Lars Nylons, Mark Harris, and Jan Prins. Fast n-body simulation with CUDA. `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf`.

24 John K Salmon. *Parallel hierarchical N-body methods*. PhD thesis, California Institute of Technology, 1991.

25 Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

26 Andreas Stefik and Stefan Hanenberg. Methodological Irregularities in Programming-Language Research. *Computer*, 50(8):60–63, 2017. `doi:10.1109/MC.2017.3001257`.