Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Janez Pintar

Programski jeziki za vzporedno programiranje

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preglejte obstoječe programske jezike, ki z uporabo konstruktov za porazdelitev podatkovnih struktur in toka programa omogočajo vzporedno programiranje na višjem abstraktnem nivoju, kot je to na voljo v jezikih C in C++ z uporabo okolja OpenMP in knjižnice MPI. Izberite nekaj tipičnih predstavnikov teh jezikov in na primeru nekaj testnih problemov izmerite hitrost programov in ocenite programerjevo izkušnjo pri programiranju v teh jezikih.

Univerzitetni študij ni vedno enostaven in moram se zahvaliti vsem tem, ki so mi pomagali po poti, brez katerih bi najverjetneje študija ne dokončal. Hvala Irena. Hvala Rudi. Grazie Massimo. Grazie Denise.

Kazalo

Povzetek

Naslov: Programski jeziki za vzporedno programiranje

Avtor: Janez Pintar

Medtem ko so se večjedrni računalniki industrijsko že povsem uveljavili, se ni *pomembno* uveljavil še noben programski jezik, ki bi implicitno dovolil izkoriščanje vzporednega računanja. Trenutno je *de facto standard* za vzporedno programiranje programski jezik C++ z okoljem OpenMP za večnitno programiranje ter s knjižnico MPI za porazdeljeno programiranje.
V delu smo spoznali *nekatere* razvijajoče se programske jezike, ki omogočajo implicitno vzporedno računanje in nato primerjali te med seboj in z že uveljavljenimi pristopi. Ugotovili smo, da je vzporedno programiranje v jeziku **Chapel** zaradi opisa vzporednosti na visokem nivoju bolj produktivno kot z jezikom C++ in pri tem so hitrosti izvajanja primerljive. Žal je jezik Chapel zaradi majhnega ekosistema še vedno bol namenjen zgolj *računalničarjem*. Jezik **Julia** je posebno primeren za znanstvene programerje, ki nimajo nujno bogatega računalniškega znanja. Kljub interpretirani naravi je hitrost jezika presenetljiva, čeprav ne more tekmovati s hitrostjo prevedenih jezikov. Končno se jezik **X10** ni posebno izkazal ne v hitrosti izvajanja ne v produktivnosti.

Ključne besede: vzpredno programiranje, Chapel, Julia, X10.

Abstract

Title: Programming languages for parallel programming

Author: Janez Pintar

Multi-core computers have already become fully established, but there is no programming language, supporting simple exploitation of parallel computing. The current *de facto standard* for parallel programming is the C++ programming language, with the OpenMP framework for multi-threaded programming and with the MPI library for distributed programming.
In this work we studied *some* evolving programming languages that allow parallel programming. We compared them with the established methods. We found that thanks to the **Chapel** high-level parallel programming approach, programming in Chapel is more productive than programming in the C++ language, even though the execution speeds are comparable. Unfortunately, because of the small ecosystem, Chapel is still mostly meant for people with knowledge in *computer science* and *parallel computing*. **Julia** is especially suitable for scientific programmers who do not necessarily have *computer science* skills. Despite the interpreted nature, the speed of the language is surprising, although it can not compete with the speed of the compiled languages. Finally, the language **X10** did not performed very well, not in the speed of execution, neither in productivity.

Keywords: parallel programming, Chapel, Julia, X10.

Poglavje 1

Uvod

1.1 Kje smo? Kaj želimo?

Živimo v obdobju, kjer nas hitrost elektrike (približno 300.000.000 m/s [19]) ovira pri razvoju hitrejših procesorjev. Stvar je enostavna: običajni procesorji so digitalni in delujejo z elektriko, ta pa ima končno hitrost. To pomeni, da imajo tudi procesorji neko zgornjo mejo hitrosti. Dan danes smo prav ob tej meji, kar pomeni, da bodoči procesorji, ki slonijo na istem modelu delovanja, ne bojo nič kaj posebno hitrejši od današnjih.
Trenutno se raziskuje alternative običajnim procesorjem, kjer med najbolj znanimi lahko zasledimo kvantne [18] ali organske [23]. Taki procesorji so še v fazi raziskovanja in niso še tako razviti, da bi se začeli uporabljati. Zdaj pa nazaj na običajne procesorje: kot rešitev problema hitrosti enega procesorja sta se uvedla dva nova načina računanja oz. programiranja:

- **večnitno programiranje:** sloni na večjedrnih procesorjih, tj. procesor, ki ima več kot eno enoto za računanje. To pomeni, da lahko taki procesorji *več* stvari istočasno računajo, ne pomeni pa, da stvari *hitrejše* računajo.
- **porazdeljeno programiranje:** več računalnikov istočasno računa, pri tem si preko mreže izmenjujejo podatke, da si pomagajo pri reševanju skupnega problema.

Trenutno nam ni tuje kupiti telefona, računalnika, ali tudi zakaj ne... hladilnika, ki vsebuje večjedrni procesor. Ampak medtem, ko so se večjedrni računalniki industrijsko že povsem uveljavili, se ni *pomembno* uveljavil še noben programski jezik, ki bi implicitno dovolil izkoriščanje vzporednega računanja. Trenutno je *de facto standard* za vzporedno programiranje programski jezik C++ z okoljem OpenMP za večnitno programiranje ter s knjižnico MPI za porazdeljeno programiranje.
Cilj tega dela je spoznati *nekatere* razvijajoče se programske jezike, ki omogočajo implicitno vzporedno računanje in nato primerjati te med seboj in z že uveljavljenimi pristopi. V sklopu dela se bomo posvetili raziskovanju naslednjih programskih jezikov: **Chapel**, **Julia** in **X10** [7].

1.2 Uveljavljeni pristopi

Kot že omenjeno, je *de facto standard* za vzporedno programiranje jezik C++ s knjižnico OpenMP ter protokolom MPI. Ta ekosistem tehnologij je tipa *od spodaj navzgor* (angl. bottom-up), ki programerju nudi velik nabor nizko-nivojskih funkcionalnosti. Dobra stran tega je, da lahko programsko rešitev definiramo do najmanjših možnih detajlov in pri tem optimiziramo performanse. Slabost je pa v nizki produktivnosti: programer se mora med programiranjem soočati z veliko izbirami, za katere bi verjetno rad imel v večini primerih privzeto obnašanje.

1.2.1 Programski jezik C++

Leta 1979 je Bjarne Stroustrup začel razvijati nek programski jezik, ki ga danes poznamo kot C++ [1]. Ta je splošno-namenski, imperativen, objektno-usmerjen, ima podporo za nizko-nivojsko upravljanje s pomnilnikom, ter podpira generično programiranje. Načrtovan je bil za uporabo na področjih, kjer so performanse pomembne: sistemsko programiranje, programiranje vgrajenih sistemov, ... Jezik ima ogromno število funkcionalnosti in je izredno močan. Veliko (tudi znanih) programerjev kritizira jezik C++ in pravi, da je

ta preveč kompleksen. Kljub temu je eden izmed globalno najbolj popularnih jezikov.

1.2.2 Okolje OpenMP

Okolje OpenMP podpira vzporedno programiranje na večjedrnih sistemih s skupnim pomnilnikom [15]. Okolje je nabor *direktiv za prevajalnik*, *funkcij* in *okoljskih spremenljivk*. Za razliko od običajnih C/C++ knjižnic mora okolje OpenMP podpirati tudi prevajalnik. Oglejmo si primer kode, ko izvajanje zanke izvedemo vzporedno:

```
int main(int argc, char** argv) {
    int a[1024];
    #pragma omp parallel for
    for (int i = 0; i < 1024; ++i) {
        a[i] = i;
    }
    return 0;
}
```

1.2.3 Knjižnica MPI

Protokol MPI se uporablja za medprocesno komunikacijo, kjer se lahko procesi porazdeljeno izvajajo na različnih računalnikih [14]. MPI je dominantni model za medprocesno komunikacijo na področju vzporednega programiranja na sistemih s porazdeljenim pomnilnikom. Oglejmo si nek enostaven primer uporabe:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    char buffer[64];
```

```
    int id;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    switch (id) {
        case 0: {
            MPI_Recv(buffer, 64, MPI_CHAR, 1, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("pokemon: %s\n", buffer);
            break;
        }
        case 1: {
            strcpy(buffer, "bulbasaur");
            MPI_Send(buffer, 64, MPI_CHAR, 0, 0,
                MPI_COMM_WORLD);
            break;
        }
        default:
            break;
    }
    MPI_Finalize();
    return 0;
}
```

Zgornji primer je mišljen, da se ga izvede z dvema procesoma. Drugi proces bo prvemu poslal niz "bulbasaur". Nato bo prvi proces ta niz izpisal na standardni izhod. Funkcija `MPI_Comm_rank(...)` bo privzela identifikator procesa. Drugi proces bo prvemu s funkcijo `MPI_Send(...)` poslal sporočilo. Prvi proces bo nato sporočilo sprejel s funkcijo `MPI_Recv(...)`. Ko program prevedemo, ga lahko zaženemo z ukazom: `mpiexec -n 2 ./program`. Ta bo ustvaril dva lokalna procesa prevedenega programa, kjer prvi bo imel identifikator 0, drugi pa 1.

1.3 Raziskovani programski jeziki

1.3.1 Chapel

Chapel je prevajan vzporedni objektno-usmerjen programski jezik, ki ga razvija Cray Inc od leta 2009. Glavni cilj jezika je biti produktiven kot Python, hiter kot Fortran, prenosen kot C in skalabilen kot MPI [2]. Chapel abstrahira podatkovno vzporednost in distribucijo ter vzporednost nalog. S tem podpira vzporedno programiranje na visokem nivoju.

Medtem ko si jezik sposoja večino konceptov iz popularnih programskih jezikih razvitih pred njim (Ada, C++, C#, Java, ...), so koncepti povezani z vzporednostjo v večini prevzeti od jezika HPF (High Performance Fortran), jezika ZPL in razširitvijo MTA jezikom Fortran in C [5].

1.3.2 Julia

Julia je *interpretiran* programski jezik, ki obljublja podobne performanse kot prevedeni programski jeziki, na primer C in Fortran. Razvijajo ga od leta 2009. Na začetku je bil jezik mišljen za programiranje visoko performančne numerične analize, dandanes pa se ga uporablja kot splošno-namenski programski jezik. Jezik Julia lahko uporabljamo kot dinamičen programski jezik (to pomeni, da nam tipov spremenljivk ni potrebno vnaprej definirati). Lahko pa katerokoli spremenljivko eksplicitno deklariramo in ji dodamo tip (to ima pogosto pozitiven vpliv na performanse in razumljivost kode). Seveda podpira konkurenčno, vzporedno in porazdeljeno programiranje. Ima veliko zbirko matematičnih knjižnic, na primer za linearno algebro, generiranje naključnih števil, ... Med jeziki, ki so imeli vpliv nad načrtom Julije, lahko naštejemo C, List, Lua, Mathematica, Matlab, Perl, Python, R, Ruby in Scheme [11].

1.3.3 X10

Programski jezik X10 ga razvija IBM od leta 2004 [27]. Glavni cilj jezika je omogočiti produktivnost pisanja učinkovitih porazdeljenih programskih rešitev. Jezik je mišljen, da se ga primarno izvaja na računalniških gručah, torej na več večjedrnih procesorjih, vsak z lastnim pomnilnikom, ki so med seboj povezani z mrežo. Velik poudarek v načrtu jezika je na produktivnosti programiranja, ponovno uporabnostjo X10 izvorne kode in uporabnostjo že obstoječih knjižnic napisanih v drugih programskih jezikih. Seznam jezikov, ki so imeli vpliv nad načrtom jezika X10, je zelo kratek: C++ in Java. To se hitro opazi pri programiranju s tem [25].

Poglavje 2

Julia

2.1 Splošno o jeziku

Julia je dinamičen interpretiran jezik. Programer bi pričakoval, da je zaradi tega počasen jezik, ampak ni tako. Ob *pravilni* (več o tem v razdelku 2.4) uporabi je hitrost skoraj primerljiva tisti, ki jo dosegajo prevedeni jeziki. Tipiziranje spremenljivk je opcijsko, v splošnem je le priporočeno (več o tem v razdelku 2.4.1).

2.1.1 Matematično usmerjena sintaksa

Zaradi zelo dinamične sintakse jezika je mogoče matematične formule v jeziku napisati tako, da so lahko berljive kateremukoli matematiku, brez potrebe, da ima sploh pojma o programskih jezikih. Primer za današnji čas še vedno eksotične sintakse si lahko ogledamo v sliki 2.1.
Indeksi polj v jeziku Julia se začnejo s številom 1 in ne z običajnim številom 0. To je seveda matematikom všeč, za običajne programerje je lahko ta lastnost zapeljiva.

```
using Base.MathConstants
δ = ℯ
Σ = sum
for x ∈ [φ, π, 2δ]
    y = Σ([√x, 42])
    println(y)
end
```

Slika 2.1: Primer eksotične sintakse v jeziku Julia.

2.2 Podpora za vzporedno programiranje

Podpora za vzporedno programiranje je za jezik Julia še eksperimentalna. To pomeni, da se bo od časa pisanja te diplomske naloge programski vmesnik lahko še spremenil [16].

2.2.1 Večnitno programiranje

Povečanje števila niti

Privzeto je število niti programa enako 1, da lahko izkoristimo vsa jedra našega procesorja pa je potrebno to velikost povečati. To lahko naredimo tako, da pred zagonom interpreterja postavimo *spremenljivko okolja* `JULIA_NUM_THREADS` na število jeder našega procesorja. Na primer z ukazom:

```
$ export JULIA_NUM_THREADS=4
```

Vzporedno izvajanje zank

Vzporedno izvajanje zank lahko dosežemo z anotacijo `Threads.@threads`, ki jo postavimo pred zanko, za katero želimo, da se iteracije izvedejo vzporedno. Na primer program

```
a = zeros(Int64, 10)
Threads.@threads for i = 1:10
```

```
    a[i] = Threads.threadid()
end
println(a)
```

bo ob zagonu izpisal nekaj kot:

```
[1, 1, 1, 2, 2, 2, 3, 3, 4, 4]
```

2.2.2 Porazdeljeno programiranje

Dodajanje procesov

Ko zaženemo interpreter, se na nivoju operacijskega sistema, ustvari en glavni proces, ki začne izvajati kodo. Med izvajanjem lahko ustvarimo poljubno število dodatnih procesov interpreterja, ki preko mreže med seboj komunicirajo. Dodatni procesi se lahko izvajajo na lokalnem stroju ali pa na oddaljenih strojih preko protokola SSH. Dodajanje procesov dosežemo z uporabo funkcije `addprocs(...)`.
Dobra praksa je, da se na nivoju datotek ločuje kodo, ki je namenjena dodajanju procesov, ter kodo, ki je namenjena konkretnemu programu [22]. Sledi primer, kjer glavni proces zaženemo na *računalniku A*, ta bo najprej zagnal kodo, ki je vsebovana v datoteki *init.jl* in se potom protokola SSH povezal na *računalnik B*, končno pa zagnal še glavni program iz datoteke *main.jl*

init.jl

```
using Distributed;
addprocs(["lalg@212.235.189.201:5110"];
    exename = "/home/lalg/bin/julia",
    dir = "/home/lalg")
```

main.jl

```
using Distributed;
for i = procs()
```

```
    info = () -> (gethostname(), Threads.nthreads())
    data = fetch(@spawnat i info())
    println(data)
end
```

Vse to dosežemo, če programa *init.lj* in *main.jl* zaženemo z ukazom

```
$ julia --load init.jl main.jl
```

Rezultat zagona teh dveh programov je izpis

```
("zora", 4)
("gpufarm.fri.uni-lj.si", 24)
```

Prednost take prakse je, da ni potrebno spreminjati glavnega programa glede na okoliščino izvajanja, naj bo ta prenosni računalnik ali gruča računalnikov.

Anotacije za porazdeljeno izvajanje

Julija ima nabor anotacij, s katerimi določamo porazdeljeno obnašanje kode [9]. Med temi lahko zasledimo:

- `@everywhere`: Dodana pred izrazom, ki se bo izvedel na vseh povezanih procesih.
- `@spawnat <id>`: Dodana pred izrazom, ki se bo izvedel na procesu, ki je identificiran s številko `id`. Vrne kot rezultat objekt tipa `Future`, ki vsebuje rezultat izraza.
- `@spawn`: Je semantično ekvivalenta anotaciji `@spawnat` z razliko, da bo Julija samo določila, kateri proces bo izraz izvedel.
- `@distributed`: Dodana je pred zanko, katere iteracije bodo disjunktno razdeljene po procesih. Vsak proces bo začel ločeno izvajati njemu določene iteracije.

Porazdeljenost podatkov

Vsakič, ko ustvarimo nov proces, bo ta podedoval vse globalne spremenljivke procesa, ki ga je ustvaril. Pomembno se je zavedati, da se ustvari kopija spremenljivk. To pomeni, na primer, da če spremenimo vrednost spremenljivke v nekem procesu, bo ta sprememba vidna le temu. Med možnimi rešitvami jezik nudi dva tipa: `SharedArray` in `DArray`.

Tip `SharedArray`

Tip `SharedArray` uporablja sistemski deljeni pomnilnik (angl. shared memory), da dovoli procesom dostop do istega pomnilnika. Tip je primeren v slučajih, ko več lokalnih procesov potrebuje istočasni dostop do velike količine podatkov. Seveda taka vrsta deljenja ni mogoča, če si procesi ne delijo istega lokalnega pomnilnika, ampak se izvajajo na primer na ločenih računalnikih. Na primer, program

```
using Distributed
N = Sys.CPU_THREADS
addprocs(N)
@everywhere using SharedArrays

a = SharedArray{Int8, 1}(2N)
b = zeros(Int8, 2N)
@sync @distributed for i = 1:2N
    a[i] = myid()
    b[i] = myid()
end
println(myid())
println(a)
println(b)
```

bo ob zagonu izpisal nekaj kot

```
1
```

```
Int8[2, 2, 3, 3, 4, 4, 5, 5]
Int8[0, 0, 0, 0, 0, 0, 0, 0]
```

Opazimo, kako je delo razdeljeno le dodanim procesom (identifikatorji med 2 in 5), medtem ko *glavni* proces (identifikator 1) ostane neobremenjen.

Tip `DArray`

Tip `DArray` je abstrakcija polja, pri katerem velja, da so podatki porazdeljeno shranjeni med različnimi procesi (tudi na ločenih računalnikih) [8]. Vsak proces ima direkten dostop le do podmnožice vseh elementov. Poglejmo si primer uporabe:

```
using Distributed
N = Sys.CPU_THREADS
addprocs(N)
@everywhere using DistributedArrays

a = dzeros(Int8, (N, N))
for id in workers()
    @fetchfrom id begin
        fill!(localpart(a), myid())
    end
end
println(a)
```

Instanco tipa `DArray` smo ustvarili z uporabo funkcije `dzeros(...)`, ki je ekvivalentna klicu `distribute(zeros(...))`. Podobno kot za funkcijo `zeros(...)` obstajajo še druge kot: `dones(...)`, `drand(...)`, `dfill(...)`, itd.. Z funkcijo `localpart(...)` pridobimo dostop do lokalno shranjene podmnožice polja.

Ob zagonu bo program izpisal nekaj kot:

```
Int8[ 2 2 4 4;
```

```
      2 2 4 4;
      3 3 5 5;
      3 3 5 5 ]
```

Iz izpisa lahko opazimo, kako je bilo polje porazdeljeno različnim procesom.

Pozor: Tip `DArray` je vključen v paketu `DistributedArrays`, ta pa ni vključen med paketi, ki se namestijo ob namestitvi jezika Julija. Zaradi tega ga je potrebno ročno namestiti. To lahko naredimo na primer z naslednjima ukaza:

```
$ code="import Pkg; Pkg.add(\"DistributedArrays\")"
$ julia <<< $code
```

2.3 Podpora za programiranje GPE

Implicitno programiranje grafičnih kartic v jeziku Julija je podprto s paketom `CLArrays` [6], ki sloni na uporabi knjižnice `OpenCL`.

2.3.1 Kako naj bi delovalo?

Oglejmo si primer:

```
using CLArrays
x = CLArray(rand(100))
y = CLArray(rand(100))
b = CLArray(rand(100))
f(x) = 1 / (1 + exp(-x))
y .= f.(x .+ b)
```

S konstruktom `CLArray(...)` alociramo spomin na GPE, v našem primeru so to 3 polja 100 naključnih elementov privzetega tipa `Float64`.
Prvotno bo interpreter poskušal izračunati izraz `x .+ b`. V paketu `CLArrays` so vektorske aritmetične operacije za tip `CLArray` implementirane tako, da se

te izvajajo kar na GPE z uporabo trdo-kodiranih (angl. hard-coded) funkcij. Nato nad rezultatom prejšnjega izraza vektorsko apliciramo funkcijo `f(...)`. Ta je do tega trenutka ostala na nekem abstraktnem nivoju v obliki abstraktne sintakse jezika. Interpreter bo s podporo knjižnice *Transpiler.jl* prevedel abstraktno funkcijo v OpenCL funkcijo. To bo namestil na GPE in jo izvedel nad rezultatom prejšnjega izraza [12].

2.3.2 Problemi z namestitvijo paketov

Žal nam med časom pisanja te diplomske naloge ni nikoli uspelo namestiti paketa `CLArrays`. Namestitev bi morala biti zelo enostavna in jo lahko dosežemo z ukazoma:

```
$ code="import Pkg; Pkg.add(\"CLArrays\")"
$ julia <<< $code
```

Paket smo poskušali namestiti z različnimi verzijami Julije. Problem je, da paket *še* ni združljiv z novejšo verzijo jezika Julia 1.0. Pri uporabi starejše verzije 0.7 pride do napak pri prevajanju paketov. Pri tem smo namestitev poskušali opraviti na različnih operacijskih sistemih (openSUSE LEAP 15, Windows 10 in CentOS 6.8), a nikoli uspešno.

2.4 Nasveti za boljše performanse

Kljub temu, da je Julia interpretiran jezik, je zaradi zgradbe interpreterja zelo hitra. *Dobro* napisana koda v Juliji je lahko performančno primerljiva s hitrostjo programov napisanih v prevedenih jezikih. Pomembno pa je razumeti, kaj je v jeziku Julia *dobra koda* [17].

2.4.1 Tipi spremenljivk

Vsakič, ko deklariramo neko spremenljivko, ji lahko določimo tudi tip. Če spremenljivki ne definiramo tipa, bo Julia poskušala inferenčno določit njen

tip. Če tega ne more *varno* storiti, obdrži spremenljivko na nekem abstraktnem nivoju, kar je velika performančna kazen. V interesu programerja je, da Julia *vedno* določi ne-abstrakten tip spremenljivke.

Globalne spremenljivke

Globalnim spremenljivkam se je bolje izogniti. Implementacija interpreterja je taka, da ne more določiti tipa globalne spremenljivke, zaradi tega jo obdrži na nekem abstraktnem ne-optimiziranem nivoju. Vzemimo kot primer dva programa, ki izvedeta neko zanko 10.000.000-krat. Prvi naj kot števec iteracije uporablja globalno spremenljivko,

```
i = 0
function f()
    global i
    while i < 100_000_000
        i += 1
    end
end
println(@elapsed f());
```

drugi pa naj kot števec uporablja lokalno spremenljivko.

```
function f()
    i = 0
    while i < 100_000_000
        i += 1
    end
end
println(@elapsed f());
```

Rezultati hitrosti izvajanja so res presenetljivi. Sledi nekaj časov izvajanja:

- z globalno spremenljivko: 8.552s, 8.447s, 8.437s, 8.477s, ...
- z lokalno spremenljivko: 0.015s, 0.016s, 0.015s, 0.015s, ...

Rezultati kažejo, da je, za dani problem, uporaba globalne spremenljivke časovno približno 500-krat dražja od uporabe lokalne spremenljivke.

Abstraktni tipi

Uporaba abstraktnih tipov je na nek način kot, da eksplicitno onemogočimo jeziku opraviti optimizacije nad spremenljivko. Seveda je možnost poljubne abstrakcije dobra lastnost Julije. Potrebno pa se je zavedati, da je lahko abstrakcija časovno zelo draga. En primer slabe abstrakcije so polja, ki vsebujejo elemente abstraktnega tipa. Vzemimo kot primer naslednji program:

```
function f(a)
    local s = 0
    for i = 1:length(a)
        s += a[i] * π
    end
    return s
end

a0 = rand(Float64, 50_000_000)
a1 = append!(Real[], a0)

println(@elapsed f(a0))
println(@elapsed f(a1))
```

Funkcija `f(a)` opravlja neke (*recimo naključne*) operacije nad vsakim elementom v polju, ki je podano kot parameter. Nato program generira dve polji z enako naključno vsebino. Razlika med polji je, da prvo vsebuje elemente ne abstraktnega tipa **Float64**, medtem ko drugo vsebuje elemente abstraktnega tipa **Real**. Poglejmo si par rezultatov izvajanja:

```
0.170737863
11.297855719
```

```
0.177733364
12.301044122
```

```
0.165784447
10.954320798
```

```
0.165588891
11.204983576
```

Iz rezultatov je razvidno, da je računanje s poljem, ki vsebuje ne abstraktne tipe, približno 68-krat hitrejše.

2.4.2 Performančne anotacije

Neka razburljiva funkcionalnost jezika so *performančne anotacije*. To so anotacije, ki jih lahko pritaknemo delom kode in ob pravilni uporabi bistveno uplivajo nad performansami kode.

Anotacija `@inbounds`

Anotacijo `@inbounds` lahko dodamo pred nek izraz v kodo, v kateri Julija ne bo preverjala, če elementi, do katerih dostopamo, obstajajo ali ne. To pomeni hitrejšo kodo, ampak pri dostopu do napačnega elementa program ne bo sprožil napake in bo naredil dostop do napačnega naslova pomnilnika. Posledice tega so lahko korupcija podatkov ali takojšnja prekinitev izvajanja programa s strani operacijskega sistema. Primer uporabe:

```
function f(x, y)
    z = 0
    for i = eachindex(x)
        @inbounds z += x[i] * y[i]
    end
    return z
```

```
end
```

Ko se bo zgornja funkcija izvajala, ne bo preverjala, če so dostopi do elementov polj `x` in `y` legalni.

Anotacija `@fastmath`

Anotacijo `@fastmath` lahko dodamo pred nek izraz v kodo in s tem dovolimo interpreterju, da lahko opravi optimizacije, ki ne sledijo standardu IEEE za računanje s števili v plavajoči vejici. Posledično bo izvajanje hitrejše. Primer uporabe:

```
x = π - 1.0
y = @fastmath sin(4x) ^ cos(2π)
println(y)
```

Anotacija `@simd`

Anotacijo `@simd` lahko dodamo pred zanko in s tem povemo interpreterju, da so iteracije zanke med seboj neodvisne in se lahko posledično izvajajo v poljubnem vrstnem redu. To pomeni celo vzporedno. Primer uporabe:

```
function f(a::Vector)
    s = zero(eltype(a))
    @simd for i = eachindex(a)
        @inbounds s += a[i] ^ 2
    end
    return s
end
```

Zgornja funkcija, z anotacijo `@simd`, je približno 1,36-krat hitrejša od implementaciji funkcije brez te anotacije.

2.5 Ekosistem jezika

2.5.1 Interpreter

V času pisanja te diplomske naloge smo uporabili interpreter verzije 1.0.2. Interpreter jezika Julia je krasna tehnična umetnina, ki sloni na ogrodju *LLVM* za prevajanje abstraktne sintakse jezika v hitro binarno kodo. Zaradi tega je interpreter performančno presenetljiv. Jezik Julia pa je zato bistveno hitrejši od popularnih interpretiranih jezikov kot so JavaScript, Python, Matlab, R ali Octave, ter primerljivo hiter s statičnimi prevedenimi jeziki kot so C, Fortran ali Rust[10].

Paketi

Interpreter jezika Julia ni le program za izvajanje programov, ampak tudi program, ki upravlja s knjižnicami jezika. Knjižnice pridejo v obliki paketov in jih ob namestitvi interpreter prevede. Glavni namen prevajanja knjižnice je, da se prihrani čas pri dejanski uporabi teh. Če interpreter `julia` izvedemo v interaktivnem načinu, lahko pritisnemo tipko `]` in s tem vstopimo v način upravljanja paketov. Oglejmo si primer, kjer namestimo paket `JSON`:

```
 $ julia
julia> # pritisnimo ']'

(v1.0) pkg> add JSON
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegi...
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [682c06a0] + JSON v0.20.0
  Updating `~/.julia/environments/v1.0/Manifest.toml`
 [no changes]

(v1.0) pkg>
```

2.5.2 Knjižnice

Standardna knjižnica jezika Julia je taka, kot jo programer lahko pričakuje od vsakega splošno-namenskega programskega jezika. Pri tem ima seveda dobro podporo za vzporedno ter matematično-usmerjeno programiranje [11]. Na spletu je objavljenih veliko knjižnic za jezik Julia (približno 7.000 na platformi GitHub). Večina teh je za reševanje problemov na matematično-znanstvenih področjih kot linearna algebra, statistika, strojno učenje, ...

2.5.3 Skupnost

Jezik Julia je 37. svetovno najbolj popularen jezik [21]. Skupnost programerjev jezika Julia je že velika in aktivna. Po spletu se brez problemov najde vsebine kot so članki, diskusije na forumih, tutorske vsebine, ...

Poglavje 3

Chapel

3.1 Osnove jezika

Jezik Chapel nekoliko izstopa od skupine običajnih prevedenih jezikov, saj ima presenetljivo visok nivo abstrakcije, ki je tipičen za interpretirane jezike. Hkrati za optimizacijo programov dovoli dostop do *relativno* nizkonivojskih funkcijonalnosti. Je strogo tipiziran, ampak v praksi lahko veliko večino definiranja tipov izpustimo, ker bo prevajalnik samodejno inferenčno tipiziral spremenljivke.

3.1.1 Primer *hello world* programa

```
proc main() {
    writef("hello world\n");
}
```

3.1.2 Primer *hello world* programa v obliki skripte

Chapel dovoli, kot je običajno v skriptnih jezikih (Python, Bash, Julia, ...), pisanje programa kar izven glavne funkcije *main*.

```
writef("hello world\n");
```

3.1.3 Konfiguracijske spremenljivke

Jezika Chapel podpira *konfiguracijske spremenljivke*. To so spremenljivke, za katere velja, da:

- imajo med izvajanjem programa konstantno vrednost,
- morajo imeti ob prevajanju programa neko privzeto vrednost, in
- da lahko privzeto vrednost nadomestimo ob zagonu programa, z uporabo parametrov ukazne vrstice

Program

```
config const n = 100;
config const salt = "Ad3x";
writef("salt: %s, n: %i\n", salt, n);
```

lahko večkrat zaženemo iz ukazne vrstice z različnimi parametri in dobimo rezultate kot:

```
 $ ./program
salt: Ad3x, n: 100
 $ ./program --n=40
salt: Ad3x, n: 40
 $ ./program --n=40 --salt="dex"
salt: dex, n: 40
 $ ./program --salt="dex"
salt: dex, n: 100
 $
```

3.1.4 Abstrakcija domen

Jezik uporablja koncept *domen*. Te so v jeziku množice indeksov. Oglejmo si primer programa:

```
const D = {3..4, 0..2};
for (i, j) in D do
    writeln(i, " ", j);
```

Program bo od zagonu izpisal na standardni izhod nekaj takega kot:

```
3 0
3 1
3 2
4 0
4 1
4 2
```

Z izrazom `{3..4, 0..2}` ustvarimo dvodimenzionalno *pravokotno* domeno, ki vsebuje indekse kartezičnega produkta intervala $[3, 4]$ in $[0, 2]$. Poleg pravokotnih obstajajo še druge vrste domen, med temi omenimo predvsem porazdeljene (angl. sparse) in asociativne domene.

Domene in polja

Ko v jeziku Chapel deklariramo polje, moramo temu določiti domeno, ki bo definirala, kateri indeksi polja so dostopni. Program

```
use Random;
const D = {1..4};
var a: [D] int;
var b: [D] int;
fillRandom(a);
fillRandom(b);
for i in D do
    writeln((a[i] + b[i]) % 8);
```

bo od zagonu izpisal na standardni izhod

```
3
```

```
-5
4
-7
```

Jezik Chapel nudi veliko funkcionalnosti za upravljanje z domenami. Posledično ima programer na razpolago izredno močne operacije za delo s polji.

3.1.5 Iteratorji

Zanimiva funkcionalnost jezika Chapel so iteratorji, ki so v jeziku Chapel neke vrste funkcije, ki generirajo zaporedja. Program

```
iter factorial(n: int) {
    var c: int = 1;
    for i in 1..n {
        c *= i;
        yield c;
    }
}
for i in factorial(5) do
    writeln(i);
```

bo na standardni izhod izpisal

```
1
2
6
24
120
```

Vsakič, ko se izvede konstrukt `yield` postane argument tega nov element zaporedja. Prava moč iteratorjev se pokaže pri vzporednem izvajanju zank, saj lahko takrat dinamično načrtujemo strategijo izvajanja iteracij [4].

3.2 Podpora za vzporedno programiranje

3.2.1 Večnitno programiranje

Konstrukt `begin`

Konstrukt `begin` lahko dodamo pred ukaz. Posledično bo jezik ustvaril novo opravilo (angl. task), ki bo ukaz izvajal vzporedno s trenutnim opravilom. Program

```
use Time;
begin while (true) {
    writef("aenigma\n");
    sleep(1);
}
while (true) {
    writef("bar\n");
    sleep(1);
}
```

bo generiral zaporedje besed: aenigma, bar, aenigma, bar, ...

Konstrukt `sync`

Konstrukt `sync` lahko dodamo pred ukaz. Posledično bo program počakal, da se zaključijo vsa vzporedna opravila, ki so bila tranzitivno generirana v ukazu. Program

```
sync {
    begin writef("muha\n");
    begin writef("veverica\n");
    begin writef("opica\n");
}
writef("medved\n");
```

bo na standardni izhod vzporedno izpisal besede: veverica, muha in opica, a vrstni red teh besed ni natančno določen. Nato bo program počakal, da se vsa generirana opravila zaključijo, in končno izpisal še besedo "medved".

Vzporedne zanke ter konstrukta `forall` in `coforall`

Vzporedne zanke lahko napišemo s kombinacijo konstruktov `begin` in `sync` na naslednji način:

```
sync for i in 1..10 do begin
    writef("%i\n", i);
```

Ta program bo ustvaril eno nit izvajanja za vsako iteracijo zanke in vzporedno izpisal na standardni izhod števila od 1 do 10. Obstajata dva boljša načina vzporednega izvajanja zank, prvi z uporabo konstrukta `forall`, drugi z uporabo konstrukta `coforall`. Zgornji program lahko napišemo še kot

```
forall i in 1..10 do
    writef("%i\n", i);
```

ali kot

```
coforall i in 1..10 do
    writef("%i\n", i);
```

Razlika v uporabi teh dveh konstruktov je v številu niti, ki bodo ustvarjene za izvajanje zanke, saj

- `forall` ustvari toliko niti kolikor je jeder procesorja,
- `coforall` ustvari eno nit za vsako iteracijo zanke.

Torej je pomembno razumeti, kdaj uporabiti kateri konstrukt. Za primere, ko je pomembno, da se vsaka iteracije zanke *zares* izvaja vzporedno z drugimi iteracijami, je primerno uporabiti konstrukt `coforall`. Obratno, če je le pomembno, da se iteracije izvedejo v čim krajšem času (to naj bi se moralo doseči z uporabo vseh jedrih procesorja), je primernejše uporabiti konstrukt

`forall`, saj bo ta ustvaril manj niti in bo posledično manj preklopov med nitmi.

3.2.2 Porazdeljeno programiranje

Porazdeljenost opravil

Porazdeljenost opravil [3] dosežemo s konstruktom `on`, ki sprejme kot parameter lokacijo (objekt tipa `locale`). Konstrukt lahko zapišemo pred poljubnim ukazom. Posledično se bo izvajanje programa ustavilo na trenutni lokaciji in ukaz se bo izvedel na določeni lokaciji. Lokacija je proces, ki se lahko izvaja lokalno ali na kakšnem oddaljenem računalniku. Na primer program

```
coforall loc in Locales {
    on loc {
        writeln(here.name);
    }
}
```

lahko zaženemo z zastavico `-ln`, ki specificira število procesov, ki jih želimo zagnati. Če torej program poženemo z ukazom

```
./program -nl 4
```

se bo na standardni izhod izpisalo nekaj kot

```
zora-0
zora-1
zora-3
zora-2
```

Opozoriti moramo, da je `Locales` polje, ki vsebuje vse lokacije, kjer se trenutno program izvaja. Z besedo `here` označimo trenutno lokacijo za izvajajoči se proces.

Porazdeljenost podatkov

Domenam lahko s konstruktom `dmapped` določimo strategijo porazdelitve podatkov na različnih lokacijah. Seveda če polje definiramo z domeno, ki ima strategijo porazdelitve, bo posledično tudi polje fizično porazdeljeno po lokacijah. Na primer program

```
use CyclicDist;
const D = {1..8} dmapped Cyclic(startIdx=1);
var A: [D] int;

forall a in A do
    a = here.id;

writeln("[", A, "]");
```

bo na standardni izhod, če ga zaženemo na 4 lokacijah, izpisal nekaj takega kot

```
[0 1 2 3 0 1 2 3]
```

Iz primera lahko opazimo, kako so bili indeksi zaradi uporabe strategije `Cyclic` ciklično dodeljeni lokacijam: $\{1,5\},\{2,6\},\{3,7\},\{4,8\}$. Jezik Chapel nudi različne strategije porazdelitve in dovoli definiranje novih strategij.

Implicitna komunikacija med lokacijami

Jezik Chapel podpira medlokacijsko implicitno branje spremenljivk in pisanje v te. Ta lastnost je *zlata vredna* za produktivno porazdeljeno programiranje. Oglejmo si delovanje na primeru:

```
var x = 42;
on Locales[1] {
    var y: int;
    y = x - 2;
    x = y - 30;
```

```
}
writeln(x);
```

Program je mišljen, da se izvaja na dveh lokacijah. Izvajanje se začne na lokaciji 0, kjer bo alocirana spremenljivka `x`. Nato se izvajanje nadaljuje na lokaciji 1, kjer bo alocirana spremenljivka `y`. Sledi njena inicializacija z izrazom `x - 2`. Program se v tem koraku zaveda, da je uporabljena spremenljivka `x` shranjena na različni lokaciji in bo torej prebral implicitno vrednost spremenljivke `x` iz oddaljene lokacije. Po tem sledi ukaz, kjer spremenljivki `x` želimo prirediti vrednost izraza `y - 30`. Program iz lokacije 1 bo izračunal vrednost izraza in vrednost poslal oddaljeni lokaciji 0, da lahko vrednost shrani v spremenljivko `x`. Izvajanje programa se nato nadaljuje iz lokacije 0, kjer bo program končno izpisal vrednost spremenljivke `x`, torej število "10". Te vrste operacij so lahko časovno drage. Na primer, če želimo prebrati spremenljivko iz druge lokacije in je ta v lasti procesa, ki se izvaja na oddaljenem računalniku. Komunikacija bo potekala preko mreže, kar je v primerjavi z dostopom do pomnilnika seveda počasna operacija.

3.3 Podpora za programiranje GPE

Uradno jezik Chapel še nima podpore za programiranje grafičnih procesnih enot. Ta podpora je med 5-letnimi cilji, ki so bili leta 2016 postavljeni [2]. Po spletu se lahko že dobi kakšen dokument, ki govori o tem kakšne oblike bi morala biti ta podpora, to pa je v začetku leta 2019 tudi vse.

3.4 Ekosistem jezika

3.4.1 Prevajalnik

V času pisanja te diplomske naloge smo uporabili prevajalnik jezika Chapel verzije 1.18.0.

Namestitev

V času pisanja diplomske naloge je edini način za namestitev prevajalnika jezika Chapel ta, da prevajalnik prevedemo iz izvirne kode, ki jo lahko prenesemo iz uradne spletne strani jezika. Pred prevajanjem lahko nastavimo različne okoljske spremenljivke, da prevajalniku jezika Chapel spremenimo ali dodamo funkcionalnosti. Če želimo vklopiti funkcionalnost distribuiranega programiranja, moramo nastaviti okoljsko spremenljivko `CHPL_COMM` *na primer* na vrednost `gasnet`. Nato se lahko postavimo v koren izvorne kode in prevajalnik prevedemo z ukazi:

```
$ export CHPL_COMM=gasnet
$ ./configure
$ make
$ make install
```

Performančne zastavice

Privzeto prevajalnik izbere, da ne bo optimiziral kode, da so časi prevajanja krajši. Prevajalnik nudi širši nabor možnih optimizacij, ki jih lahko vklopimo z zastavicami. V splošnem lahko z naborom zastavic `--fast`, `--optimize` in `--specialize` vklopimo vse optimizacije prevajalnika.
Poleg tega prevajalnik privzeto tudi dodaja v končno kodo ukaze, ki preverjajo možne dinamične napake (dostop do neobstoječega elementa polja, deljenje z ničlo, ...). Seveda so ta preverjanja performančno draga in jih nočemo, ko prevajamo končno programsko rešitev. V ta namen lahko uporabimo zastavico `--no-checks`.

Ne najboljša sporočila napak

Sporočila napak niso med najboljšimi. Večkrat se med programiranjem zgodi, da sporočilo napake pri prevajanju ne pove nič več kot samo to, da je napaka v kodi. Na primer za naslednji program:

```
loop writef("konstrukt loop ne obstaja\n");
```

bo pri prevajanju prevajalnik izpisal sporočilo

```
main.chpl:1: syntax error: near 'writef'
```

Napaka v programu je, da konstrukt `loop` ne obstaja. Programer bi verjetno pričakoval, da nam prevajalnik sporoči nekaj kot: "hej, ne vem kaj je `loop`". Medtem ko v našem slučaju prevajalnik le sporoči, da je prišlo do napake in da je napaka *nekje ob* besedi `writef`.

Manjkajoče upravljanje skrajnih napak

Ni tako nemogoče, da se prevajanje kakšne napačne izvorne kode konča z notranjo napako prevajalnika. To se je med pisanjem tega diplomskega dela nekajkrat zgodilo. Notranje napake prevajalnika smo sporočili organizaciji Cray (kot je prikazano v sliki 3.1), ki je sporočilom tudi odgovorila (kot je prikazano v sliki 3.2). Potrebno je izpostaviti, da se je prevajalnik porušil le ob primerih napačne izvorne kode, medtem ko je pravilno izvorno kodo vedno prevedel kot bi moral: prevajalnik ni napačen, je le nepopoln.

3.4.2 Knjižnice

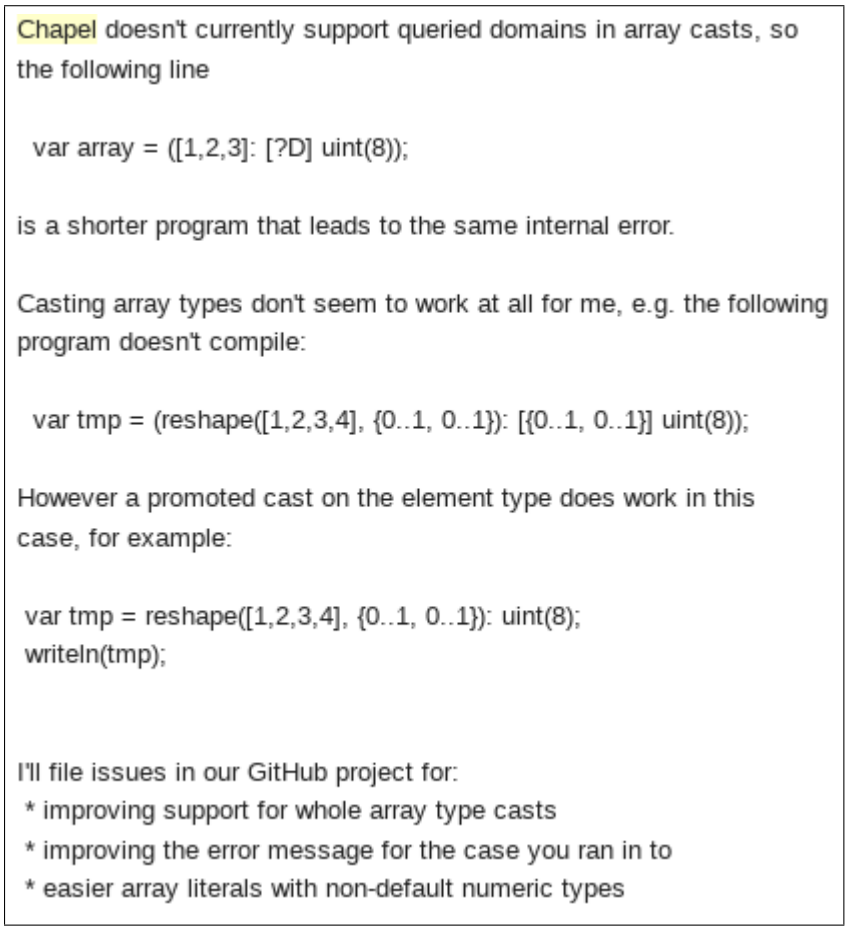
Standardna knjižnica jezika Chapel je pretežno osredotočena na funkcionalnosti, ki se uporabljajo na področju visoko-performančnega računanja.
Res se težko po spletu najde kakšno knjižnico, ki bi jo ustvarila skupnost programerjev jezika Chapel. Primer za občutek: na spletni strani https://github.com je objavljenih le 59 repozitorijev v jeziku Chapel. Za primerjavo, jezik C++ jih ima več kot 750.000.

3.4.3 Skupnost

Sam jezik Chapel ni med prvimi 100 svetovno najbolj popularnimi jeziki [21]. Skupnost programerjev jezika Chapel ni velika, a vendarle obstaja. Po spletu

Slika 3.1: Elektronsko sporočilo, ki smo ga poslali organizaciji Cray.

se lahko najde vsebine kot so članki, diskusije na forumih, tutorske vsebine, ...



Chapel doesn't currently support queried domains in array casts, so the following line

```
var array = ([1,2,3]: [?D] uint(8));
```

is a shorter program that leads to the same internal error.

Casting array types don't seem to work at all for me, e.g. the following program doesn't compile:

```
var tmp = (reshape([1,2,3,4], {0..1, 0..1}): [{0..1, 0..1}] uint(8));
```

However a promoted cast on the element type does work in this case, for example:

```
var tmp = reshape([1,2,3,4], {0..1, 0..1}): uint(8);
writeln(tmp);
```

I'll file issues in our GitHub project for:
* improving support for whole array type casts
* improving the error message for the case you ran in to
* easier array literals with non-default numeric types

Slika 3.2: Del odgovora organizacije Cray.

Poglavje 4

X10

4.1 Splošno o jeziku

Jezik X10 je zelo podoben Javi, torej objektno-usmerjen, strogo tipiziran, osnovan na razredih, z avtomatskim čiščenjem pomnilnika, ob tem pa vzporeden [26].

4.1.1 Primer *hello world* programa

```
import x10.io.Console;
public class HelloWorld {
    public static def main(args: Rail[String]) {
        Console.OUT.printf("Hello world!\n");
    }
}
```

Že ta enostaven primer programa spominja na jezik Java.

4.1.2 Tipizirana števila

V jeziku so števila tipizirana. Na primer, če v izvorno kodo napišemo "8", bo ta privzetega tipa *Long* (64-bitno celo število). Če bi želeli določiti drugačen tip, bi morali številu dodati pripono. Na primer:

- `8y` 8-bitno naravno število (Byte)
- `8uy` 8-bitno celo število (UByte)
- `8s` 16-bitno naravno število (Short)
- `8us` 16-bitno celo število (UShort)
- `8n` 32-bitno naravno število (Int)
- ...
- `8l` 64-bitno naravno število (Long)

Večkrat je ta lastnost jezika kar zoprna. Za primer si lahko pogledamo izraz `x != 0`, kjer je spremenljivka `x` tipa Byte. Prevajalnik X10 bo pri tem izrazu sprožil napako, saj je privzeto število "0" tipa Long in je prevajalnik ne bo implicitno prevedel v tip Byte. Zato je potrebno dodati pripono "y": `x != 0y`.

4.2 Podpora za vzporedno programiranje

Jezik X10 podpira vzporedno programiranje na osnovi modela APGAS, ki definira 4 konstrukte, za katere model trdi, da so dovolj splošni, da lahko izpolnijo katerokoli zahtevo programerja [20].

4.2.1 Konstrukt `async`

Konstrukt `async` lahko dodamo pred ukaz. Posledično bo jezik ustvaril novo opravilo (angl. task), ki bo ukaz izvajalo vzporedno s trenutnim opravilom. Na primer del programa

```
async while (true) {
    Console.OUT.printf("0\n");
    System.sleep(1000);
}
```

```
while (true) {
    Console.OUT.printf("1\n");
    System.sleep(333);
}
```

bo prvo zanko izvajal vzporedno z drugo in na standardi izhod generiral zaporedje: 0, 1, 1, 1, 0, 1, 1, 1, 0, ...

4.2.2 Konstrukt `finish`

Konstrukt `finish` lahko dodamo pred ukaz. Posledično bo program počakal, da se zaključijo vsa vzporedna opravila, ki so bila tranzitivno generirana v ukazu. Na primer del programa

```
finish {
    finish async System.sleep(1000);
    async {
        System.sleep(1000);
        System.sleep(1000);
    }
    System.sleep(1000);
}
```

se bo izvajal približno 3s.

Vzporedne zanke

S kombinacijo konstruktov `async` in `finish` lahko vzporedno izvajamo vsebino zank. Del programa:

```
finish for (i in 1..10) async {
    Console.OUT.printf("%i\n", i);
}
```

bo na standardni izhod vzporedno izpisal neko (recimo naključno) permutacijo števil od 1 do 10, recimo: 1, 10, 3, 9, 4, 8, 5, 7, 6, 2.

4.2.3 Konstrukt `when` in sintaksni sladkorček `atomic`

Oglejmo si naslednji primer kode:

```
var value: Long = 0;
finish {
    async for (i in 1..1000000)
        ++value;
    async for (i in 1..1000000)
        ++value;
}
Console.OUT.printf("%ld\n", value);
```

Programer bi lahko pričakoval, da bo končno izpisano število 2.000.000. V praksi to ne drži, saj so bila ob večkratni izvedbi programa izpisana končna števila: 1.169.624, 919.526, 886.754, 1.239.742, 688.169, ... To se zgodi, ker pride pri vzporednem izvajanju zank pride do dogodka *data race*, tj. ko različna fizična jedra procesorja izvajajo operacije nad istim pomnilniškim naslovom in lahko postane vsebina celice nedoločena. Rešitev za to so atomske operacije. Jezik X10 to vrsto operacij omogoča z dvema konstruktoma:

- `when (<pogoj>)`; ki ga dodamo pred ukaz, povzroči pa atomsko izvedbo ukaza, če je `<pogoj>` resničen,
- `atomic`; ki ni nič drugega kot nadomestek za `when (true)`.

Če bi na primer prejšnjo kodo želeli popraviti, da se bo *pravilno* izvajala, bi to lahko naredili tako:

```
val condition = true;
var value: Long = 0;
finish {
    async for (i in 1..1000000)
        when (condition) ++value;
    async for (i in 1..1000000)
        atomic ++value;
```

```
}
Console.OUT.printf("%ld\n", value);
```

4.2.4 Konstrukt `at`

Konstrukt `at <place>` lahko dodamo pred ukaz. Posledično se bo izvajanje programa ustavilo na trenutni lokaciji in ukaz izvedla na določeni lokaciji. Lokacija je lahko nek drugi računalniški sistem, gruča računalnikov, grafična enota, ... Del programa

```
at (here) Console.OUT.printf("kobilica\n");
```

bo na standardni izhod izpisal "kobilica", saj `here` označuje trenutno izvajajočo se lokacijo.

Vzporedno izvajanje na različnih lokacijah

Če želimo program vzporedno izvajati na več lokacijah, lahko pred konstruktom `at` dodamo še konstrukt `async`. Na primer:

```
finish {
    for (place in Place.places())
        async at (place)
            Console.OUT.printf("tiger\n");
}
```

Ta del kode bo na vsaki konfigurirani lokaciji vzporedno izpisal "tiger"na standardni izhod. Dodatne lokacije lahko sporočimo programu preko konfiguracijske datoteke.

4.3 Podpora za programiranje GPE

Jezik X10 ima podporo za programiranje GPE. Žal je podpora omejena samo na tehnologijo CUDA, ki se izvaja ekskluzivno na grafičnih karticah znamke NVIDIA.

Oglejmo si najprej, kako izgleda izvorna koda, ki se izvaja na grafičnih karticah:

```
async at (gpu) @CUDA {
    val blocks = CUDAUtilities.autoBlocks();
    val threads = CUDAUtilities.autoThreads();
    finish for (block in 0n..(blocks-1n)) async {
        clocked finish for (thread in
            0n..(threads-1n)) clocked async {
            // "kernel" koda tukaj...
        }
    }
}
```

S konstruktom `at` določimo, na kateri grafični kartici se bo koda izvajala. Anotacija @CUDA pove prevajalniku X10, da se bo koda izvajala na GPE. Za kodo v tem obsegu veljajo posebna pravila, kot na primer to, da vsebuje dve zanki zgornje oblike. Interval prve zanke določa število blokov, na katero delimo naš problem. Interval druge zanke določa število niti, ki se bo izvajalo v vsakem bloku. V telo ugnezdene zanke zapišemo kodo, ki se bo dejansko izvajala na grafični procesni enoti.
Jezik nudi tudi različne funkcije ali razrede, ki podpirajo razne operacije na GPE, recimo dodeljevanje pomnilnika, kopiranje med polji, sinhronizacija na nivoju izvajanja bloka, ...

4.4 Ekosistem jezika

4.4.1 Prevajalnik

V času pisanja te diplomske naloge smo uporabili prevajalnik verzije 2.6.1. Obstajata dva načina prevajanja in izvajanja X10 kode:

- X10 izvorno kodo se prevede s programom `x10c`. Kot izhod prevajanja se ustvarijo datoteke javanskih razredov (razširitev `.class`), te lahko

izvedemo z interpreterjem `x10`. Zanimiva lastnost je, da lahko s tem načinom prevajanja pri programiranju uporabimo javanske razrede, ki so bili napisani tudi v drugih jezikih (Java, Kotlin, Scala, ...).

- X10 izvorno kodo se prevede s programom `x10c++`. Kot izhod prevajanja se generira C++ izvorna koda, ki se jo nato prevede v binarno izvedljiv program.

Dolgi časi prevajanja

Ena izmet slabosti prevajalnika jezika X10 je, da so časi prevajanja programov relativno dolgi. Na primer iz časov prevajanja

```
 $ time gcc hello_world.c
real    0m0.335s
user    0m0.058s
sys     0m0.058s

 $ time x10c++ HelloWorld.x10
real    0m15.448s
user    0m35.752s
sys     0m1.793s
```

je razvidno, da se *hello world* program (iz razdelka 4.1.1) prevede v približno 15s, medtem ko se ekvivalentni klasični program v jeziku C prevede v manj kot sekundi.

Performančne zastavice

Privzeto prevajalnik izbere, da ne bo optimiziral kode, da so časi prevajanja krajši. Prevajalnik pozna 5 nivojev optimizacije, ki jih lahko vklopimo s klasično zastavico `-O`.
Poleg tega prevajalnik privzeto tudi dodaja v končno kodo ukaze, ki ob vsakem dostopu do polja preverjajo, če je dostop veljaven. Seveda so ta prever-

janja performančno draga in jih nočemo, ko prevajamo končno programsko rešitev. V ta namen lahko uporabimo zastavico `-NO_CHECKS`.
Torej, ko želimo prevajati končne rešitve, dodamo ti dve zastavici. Na primer tako:

```
$ x10c++ -O5 -NO_CHECKS Program.x10
```

4.4.2 Knjižnice

Standardna knjižnica jezika X10 je pretežno osredotočena nad funkcionalnosti, ki se uporablja na področju visoko-performančnega računanja.
Potencialno lahko jezik X10 uporabi katerokoli knjižnico, ki se izvaja na navideznem stroju jezika Java. Seveda, če za prevajanje uporabimo program `x10c`.
Poleg tega je na uradni spletni strani objavljenih nekaj knjižnic v jeziku X10, ki podpirajo reševanje nekaterih problemov na področju visoko-performančnega programiranja, kot so knjižnica za porazdeljeno linearno algebro, knjižnica za analizo porazdeljenih grafov, ...
Res se zelo težko po spletu najde kakšno knjižnico, ki bi jo ustvarila skupnost programerjev jezika X10. Primer za občutek: na spletni strani https://github.com je objavljenih le 19 repozitorijev v jeziku X10. Medtem, ko jih ima jezik C++ 750.000+.

4.4.3 Skupnost

Če skupnost jezika X10 obstaja, pomeni, da se dobro skriva. Jezik X10 ni med prvimi 100 svetovno najbolj popularnimi jeziki [21]. Po spletu se težko najde katerokoli vsebino glede jezika in kar se najde, je običajno objavljeno na uradni spletni strani jezika http://x10-lang.org.
Večkrat je potrebno tudi za relativno enostavne funkcionalnosti jezika pogledati v dokument specifikacije, saj se težko najde tutorske vsebine (angl. tutorial) ali objave na forumih (https://stackoverflow.com, ...).

Še samo primer za občutek: X10 je edini izmed omenjenimi jeziki, ki nima razširitev za barvanje sintakse v razvojnem okolju *Visual Studio Code*.

Poglavje 5

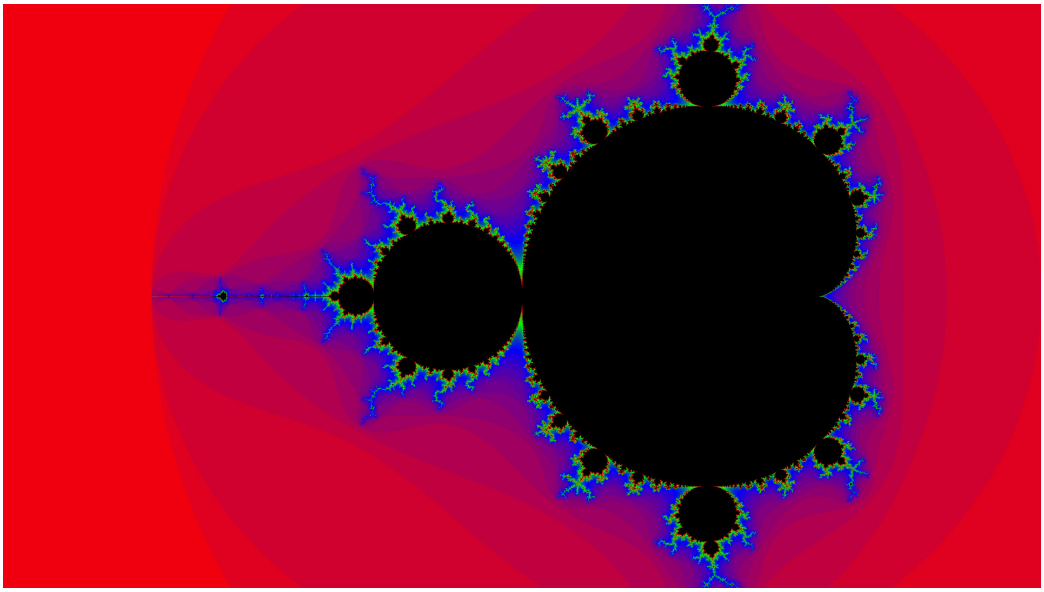
Reševanje konkretnih problemov

V izbranih programskih jezikih smo implementirali programske rešitve nekaj problemov iz sveta programiranja, da lahko pokažemo, kako se obnašajo v praksi. Programske rešitve smo izvajali na dveh računalnikih: prenosniku Zora ter strežniku GpuFarm (več v razdelku A). Za boljše razumevanje in primerjavo rezultatov smo rešitve problemov implementirali tudi za izvajanje na grafičnih karticah (v jeziku C++ in s podporo knjižnice OpenCL). Celotna izvorna koda programov, ki so opisani v tem poglavju (več kot 4.000 vrstic kode), presega velikost, ki bi dopuščala vključitev v diplomsko delo. Izvorna koda je dostopna na spletnem naslovu https://github.com/pintarj/diplomska-naloga.

5.1 Generiranje slik Mandelbrotove množice

5.1.1 Mandelbrotova množica

Mandelbrotova množica [13] je množica točk c na kompleksni ravnini, za katere velja, da vrednost funkcije $f(Z) = Z^2 + c$ ne divergira, če jo iteriramo z začetno vrednostjo $Z = 0$. Primer Mandelbrotove množice je prikazan na sliki 5.1.



Slika 5.1: Začetna slika povečav Mandelbrotove množice z zveznim pobarvanim okoljem (generirana z lastnim programom v jeziku C++).

5.1.2 Algoritem za izris množice

```
For each pixel (Px, Py) on the screen, do: {
    x0 = scaled x coordinate of pixel
    y0 = scaled y coordinate of pixel
    x = 0.0
    y = 0.0
    iteration = 0
    max_iteration = 1000
    while (x*x + y*y <= 2*2
            AND  iteration < max_iteration) {
        xtemp = x*x - y*y + x0
        y = 2*x*y + y0
        x = xtemp
        iteration = iteration + 1
    }
```

```
    color = palette[iteration]
    plot(Px, Py, color)
}
```

5.1.3 Specifični problem

Meritve smo opravili na naslednjem konkretnem problemu: generiranje slike Mandelbrotove množice z resolucijo 3840 × 2160 pik na dvodimenzionalni kompleksni ravnini $\{x + iy \mid x \in [-2.5, 1.0], y \in [-1.0, 1.0]\}$. Izvedeno je bilo zaporednih 100 iteracij jedra programa.

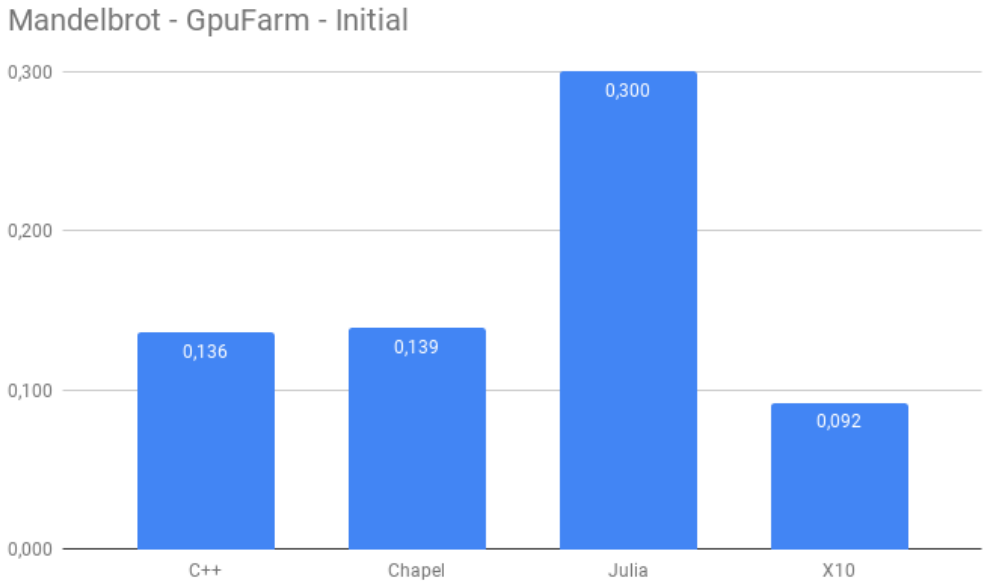
5.1.4 Potek razvoja in meritve

Julia in *katastrofalne* globalne spremenljivke

Prvič, ko smo za vsak jezik zagnali programsko rešitev nad specifičnem problemom, je bilo izvajanje rešitve v jeziku Julia tako dolgo, da nismo dočakali konca izvajanja oziroma smo izvajanje prej ročno prekinili. Tam, kjer so C++, X10 in Chapel uporabili kakšno desetino sekunde, je Julia potrebovala *več* kot pol ure. To je čudno, saj programski jezik Julia obljublja podobno hitrost, kot jo dosegajo prevedeni jeziki. Rešitev za ta problem smo našli na uradni spletni strani jezika oziroma v razdelku za *performančne namige*. Tam je bilo opisano, kaj pozitivno oziroma negativno vpliva na hitrost izvajanja. Problem v našem primeru je bila uporaba *globalnih spremenljivk*, ki so v primerjavi z lokalnimi spremenljivkami približno 500-krat počasnejše (več v 2.4.1).

Izhodiščno stanje časov izvajanja

Programsko rešitev v jeziku Julia smo popravili. Nato smo izmerili čase izvajanja, ki si jih lahko ogledamo na sliki 5.2.



Slika 5.2: Čas izvajanja (v sekundah) začetnih verzij programskih rešitev za problem Mandelbrot z uporabo 24 niti na računalniku GpuFarm (1).

Dinamična porazdeljenost dela niti

Iz meritev je razvidno, da je rešitev v jeziku X10 izredno hitra, če jo primerjamo z drugimi. Verjetno še *preveč* hitra. Razlago te hitrosti smo dobili, ko smo si prebrali *performančne namige* za jezik X10 [24], ki so objavljeni na uradni spletni strani.

Za jezik X10 velja, da so vse iteracije zanke postavljene v vrsto. Nato posamezne niti zaporedno izvajajo vse elemente iz vrste. V nasprotju s tem načinom velja za jezike C++, Chapel in Julia velja, da je pred izvedbo vzporedne zanke vsaki niti dodeljen zvezni disjunktni pod-interval glavnega intervala, ki ga bo nit izvajala.

Lastnost problema računanja Mandelbrotove množice je, da se čas za računanje barve vsake pike bistveno razlikuje na podlagi vhodnih parametrov ter položaja pike. Vzemimo kot primer sliko 5.1. Rdeča območja zahtevajo malo računanja, medtem ko je računanje črnih območij časovno zahtevno.

Programske implementacije so take, da vzporedno računajo po vrsticah. Za jezik C++, Chapel in Julia velja, da se bo izvajanje za vsako nit porazdelilo po vrsticah na primer kot v sliki 5.3. Rumeno in zeleno območje je časovno

Slika 5.3: Privzeta porazdelitev vzporednega izvajanja za jezik C++, Chapel in Julia.

zahtevnejše za računati kot rdeče in modro območje. To vodi v neuravnoteženo časovno zahtevnost izvajanja in je torej čas izvajanja daljši. Jezik X10 ne bo začetno porazdelil delo vsaki niti, ampak bo to dinamično opravljal med izvajanjem. Na ta način bo delo za vsako nit ostalo uravnoteženo.
Jezik Julia ne nudi *elegantnih* rešitev za ta problem. Jezik C++ in Chapel pa nudita naslednje:

- **C++:** Že obstoječi znački `#pragma omp` lahko dodamo konstrukt `schedule(...)`, ki specificira kako se interval porazdeli med niti. V našem primeru želimo doseči obnašanje, ki ga ima jezik X10 in to dosežemo tako, da del programa spremenimo iz

```
#pragma omp parallel for
for (int py = 0; py < height; ++py)
```

```
    // ...
```

v

```
#pragma omp parallel for schedule(dynamic, 1)
for (int py = 0; py < height; ++py)
    // ...
```

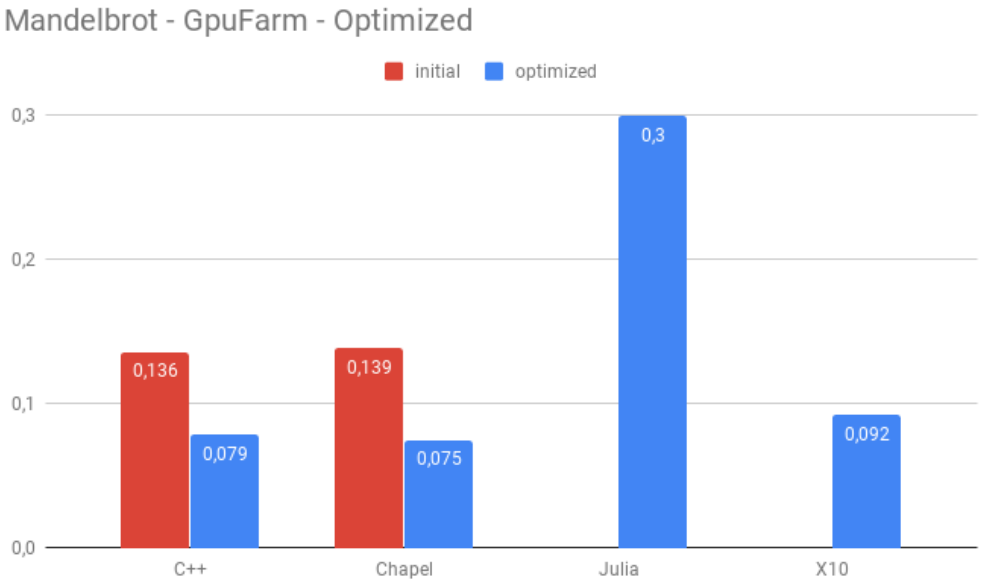
- **Chapel:** V vgrajenem modulu `DynamicIters` najdemo različne iteratorje (več v razdelku 3.1.5), ki omogočijo dinamično porazdelitev dela med niti. V našem primeru želimo doseči obnašanje, ki ga ima jezik X10, in to dosežemo tako, da del programa spremenimo iz

```
forall (py, px) in Domain do
    // ...
```

v

```
use DynamicIters;
// ...
forall (py, px) in dynamic(Domain) do
    // ...
```

Meritve časov izvajanja po optimizaciji si lahko ogledamo na sliki 5.4.



Slika 5.4: Meritve časa izvajanja (v sekundah) *čisto* optimiziranih verzij programskih rešitev (1).

Umažimo si roke pri Juliji

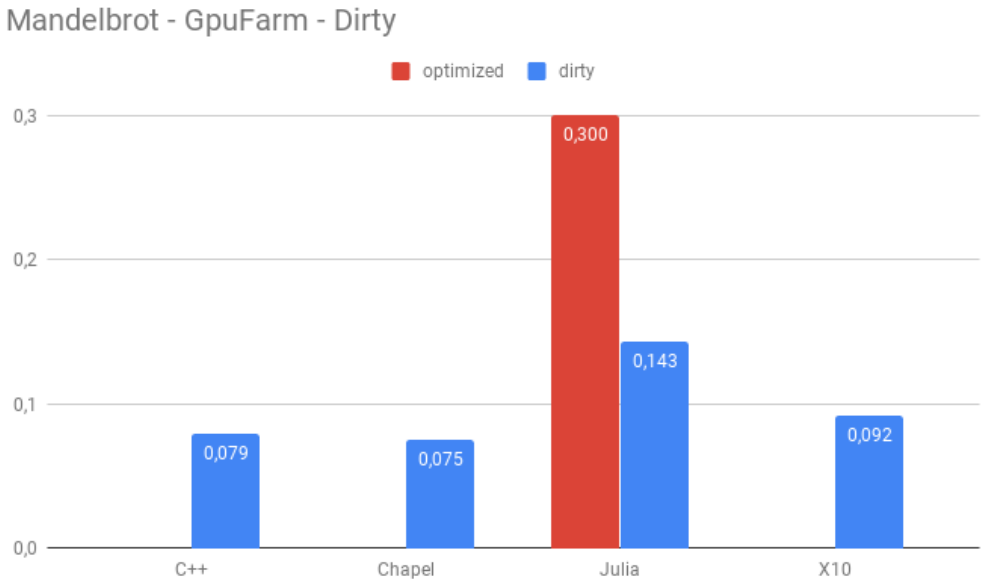
To, da jezik Julia ne nudi *čiste* rešitve, ne pomeni, da ne nudi rešitve. Ker jezik ne nudi zaželene porazdelitve dela med niti, smo to sami programirali oziroma smo del programa spremenili iz

```
Threads.@threads for j = 1:width
    # ...
```

v

```
local size::Int32 = div(width, Threads.nthreads())
local f = x -> mod(x - 1, size) *
    Threads.nthreads() + div(x - 1, size) + 1
Threads.@threads for j = map(f, 1:width)
    # ...
```

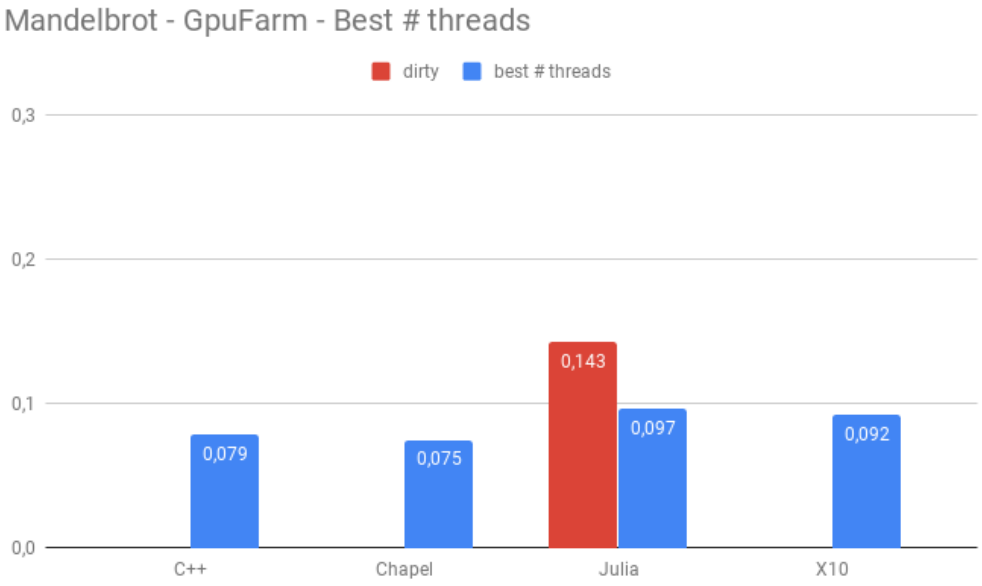
Nova rešitev je bistveno grša in kompleksnejša, ampak kot je prikazano v novih meritvah (slika 5.5), je metoda porazdelitve zelo uspešna.



Slika 5.5: Meritve časa izvajanja (v sekundah) *umazano* optimiziranih verzij programskih rešitev (1).

Anomalija interpreterja jezika Julia

Kot je opisano kasneje pri reševanju problema *Seam carving* (v razdelku 5.2.3), smo pri jeziku Julia opazili anomalijo časa izvajanja, če je število niti 20 ali več. Meritve, kjer interpreter jezika Julia zaženemo z 19 namesto 24 nitmi, so prikazana na sliki. 5.6.

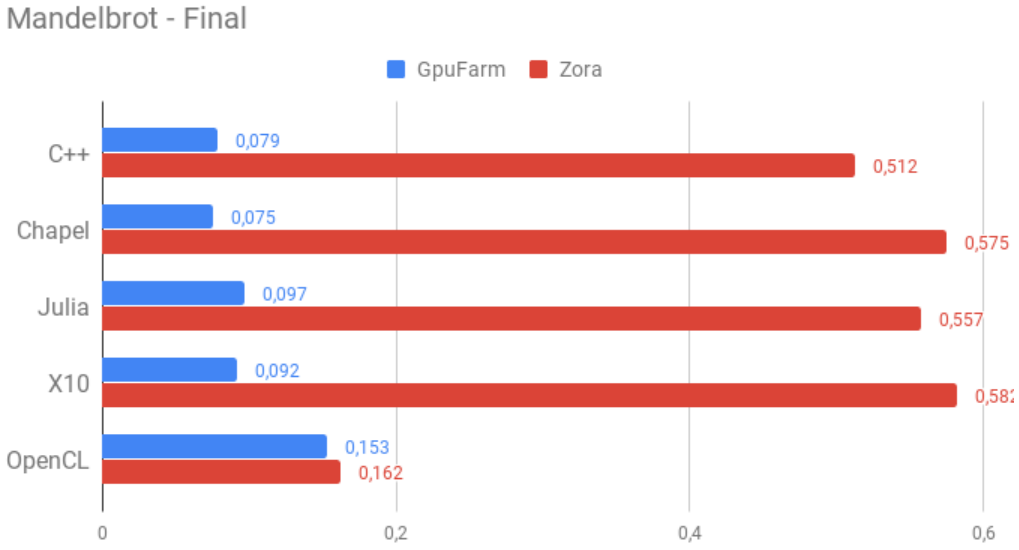


Slika 5.6: Meritve izvajanja *umazano* optimiziranih verzij programskih rešitev, kjer interpreter Julia zaženemo z 19 nitmi namesto 24.

Končne meritve

Na sliki 5.7 si lahko ogledamo končne meritve izvajanja na računalnikih GpuFarm in Zora. Iz rezultatov se vidi, da med jeziki ni absolutnega zmagovalca. Implementacija v jeziku C++ je stabilno hitra. Opazimo, da se z dodatnimi jedri na razpolago jezik Chapel bistveno pohitri. Jezik X10 se ne izkaže posebno. Čeprav se na računalniku GpuFarm hitrejše izvaja od jezika Julia, je potrebno poudariti, da program X10 zaganjamo s 24 niti, medtem ko program Julia s samo 19 nitmi.

Kar lahko koga preseneti, je dejstvo, da računanje z GPE ni absolutno najhitrejše. Čeprav je izvajanje na GPE verjetno še vedno hitrejše, je latenca komunikacije z grafično kartico precejšna. Upoštevati moramo, da je slika, ki jo generira grafična kartica (približno 8MB podatkov), v pomnilniku grafične kartice in jo moramo ob koncu računanja prekopirati v pomnilnik CPE. Taka vrsta komunikacije je časovno draga.



Slika 5.7: Meritve izvajanja končnih verzij programskih rešitev, kjer na računalniku Zora uporabimo 4 niti, na računalniku GpuFarm uporabimo 24 niti, izjemno pri jeziku Julia uporabimo 19 niti.

5.2 Seam carving

5.2.1 Opis

Seam carving [28] je tehnika krčenja digitalnih slik. Slike vsakič krčimo za eno točko (angl. pixel), dokler ne dosežemo zahtevane dimenzije. Vsakič algoritem izbere nabor takih pik, za katere velja, da če jih odstranimo, bo slika *po določenih kriterijih* izgubila najmanj informacije.
Sledi seznam potrebnih operacij, da širino slike skrčimo za eno piko:

- Vsaki piki v sliki določimo količino informacije, ki jo ta pika vsebuje. Kriteriji so lahko različni: kontrast, barva, svetlost, ...
- Od dna slike do vrha izberemo tako pot pik, da velja, da je vsota količine informacije vsebovanih pik v poti čim manjša možna za dano sliko. Pri tem lahko izberemo le eno piko za vsako vrstico slike. Po-

leg tega morajo biti izbrane pike na dveh sosednih vrsticah sosedne. Pri reševanju uporabimo dinamično programiranje. Primer izbire poti lahko vidimo na sliki 5.8.

- Iz slike odstranimo izbrano pot, ter sliko združimo.

Slika 5.8: Primer izbire poti tehnike Seam carving (generirana z lastnim programom v jeziku C++).

5.2.2 Specifični problem

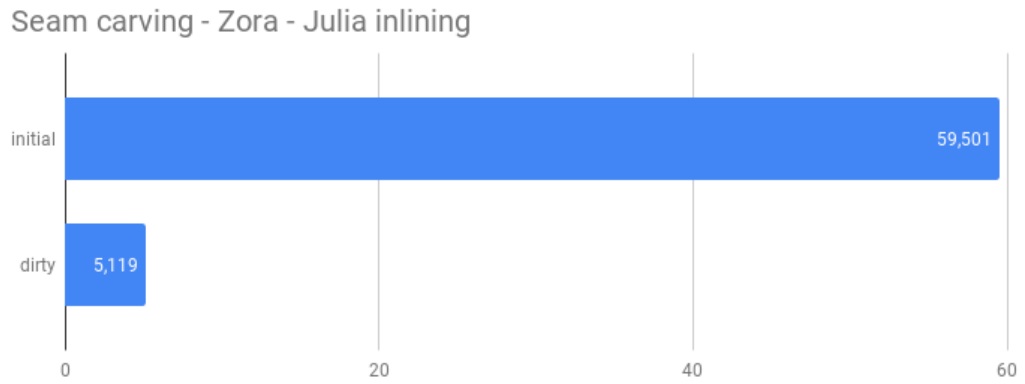
Meritve smo opravili na naslednjem konkretnem problemu: vodoravno krčenje slike, za 200 pik z uporabo tehnike Seam carving. Slika ima naključno vsebino in je resolucije 4096×4096 pik. Sliko smo generirali z ukazoma:

```
$ printf "P5\n4096 4096\n255\n" > 4k-problem.pgm
$ dd if=/dev/random bs=4194304 iflag=fullblock \
      count=4 >> 4k-problem.pgm
```

5.2.3 Potek razvoja in meritve

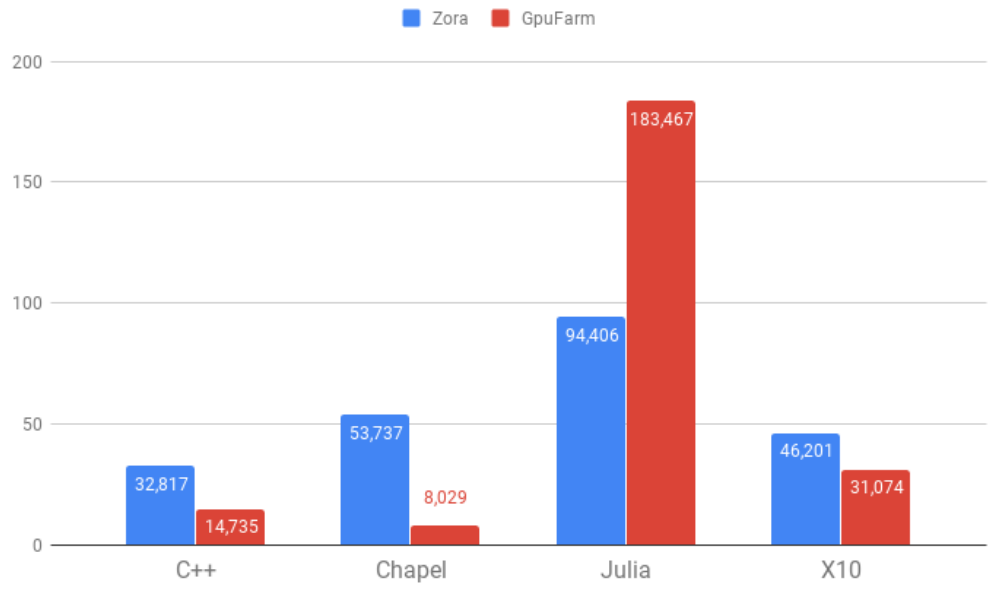
Umažimo si roke pri Juliji (že spet)

Že spet se je interpreter jezika Julia ob prvotnih meritvah časov izvajanja programskih rešitev na konkretnem problemu izvajal *v nedogled* in smo izvajanje interpreterja ročno ukinili, ker je bil potekel čas neprimerljiv z rešitvami v jezikih C++, Chapel in X10. Tokrat je bilo iskanje problema bistveno težje, ker smo se pri programiranju držali vseh *namigov programiranja* objavljenih na uradni spletni strani jezika Julia. Ugotovili smo, da je izvajanje pretirano počasno v delu algoritma, kjer s tehniko dinamičnega programiranja računamo tabelo kumulativne informacije. V tem delu programa se je zaporedje 5 vrstic kode ponavljalo v 4 različnih nezdružljivih `for` zankah. Da bi povečali berljivost kode, smo ponavljajoče zaporedje vrstic prestavili v funkcijo in iz prej omenjenih 4 zank klicali to funkcijo. Pri tem smo si predstavljali, da bo interpreter znal klice do *lokalno* definirane funkcije optimizirati do nekega nivoja, da bi bili ti časovno brezplačni (angl. inline function). *Nepričakovano*: take optimizacije interpreter ne opravi in to je razlog za počasnost izvajanja. Vsebino funkcije ročno prepišemo nazaj na omenjene 4 lokacije: izboljšavo v izvajanju (na reduciranem problemu) si lahko ogledamo v sliki 5.9.



Slika 5.9: Razlika v času izvajanja rešitve v jeziku Julia po ročni optimizaciji.

Koda je redundantna in grša, ampak zaradi te optimizacije se program skoraj 12-krat hitreje izvaja. Meritve vseh programskih rešitev po tej optimizaciji lahko najdemo na sliki 5.10.

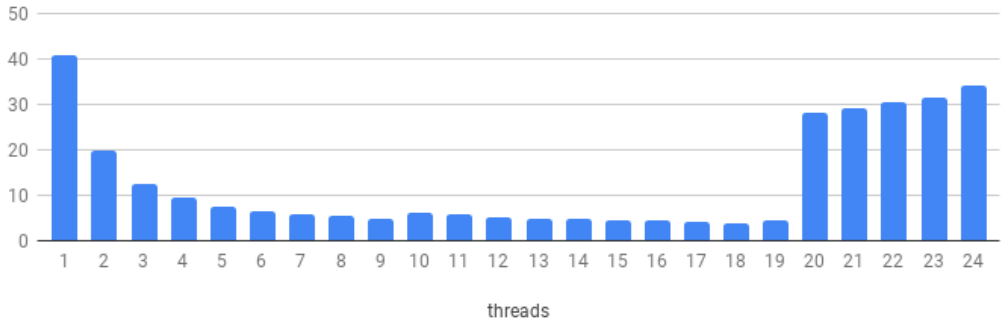


Slika 5.10: Začetne meritve časov (v sekundah) izvajanja programskih rešitev problema Seam carving za 4 niti (Zora) in 24 niti (GpuFarm).

Anomalija interpreterja jezika Julia

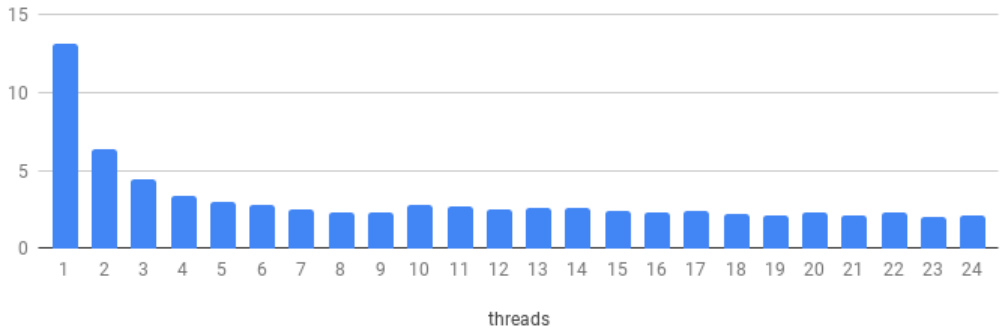
Iz meritev na sliki 5.10 precej izstopa dejstvo, da se programska rešitev v jeziku Julia počasneje izvaja na računalniku *GpuFarm* kot na računalniku *Zora*. Vprašanje, ki smo si ga postavili, je: kako je lahko izvajanje na 24 nitih **toliko** počasnejše od izvajanja na samo 4 nitih? Seveda govorimo o dveh različnih modelih CPE, ampak je razlika v hitrosti tako velika, da je stvar sumljiva. Odločili smo se, da bomo (na reduciranem problemu) interpreter jezika Julia zagnali z različnim številom niti. Meritve izvajanja lahko dobimo vidimo na 5.11.

Rezultati so presenetljivi. Medtem ko se čas izvajanja zmanjšuje s povečanjem



Slika 5.11: Meritve časov izvajanja programske rešitve jezika Julia za problem Seam carving pri različnem številu niti.

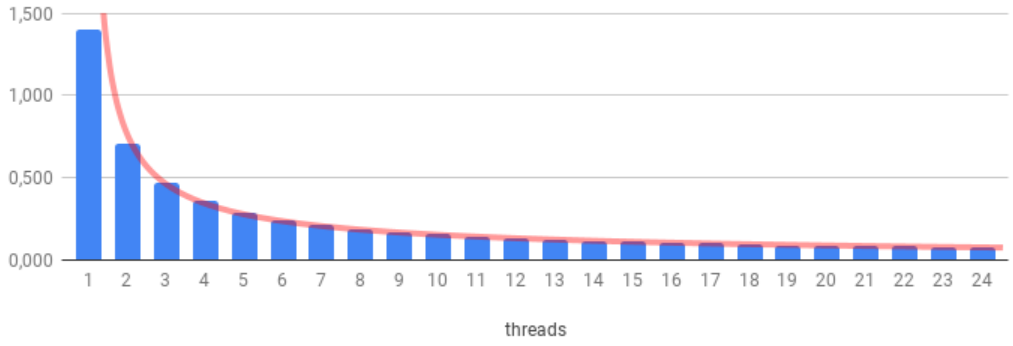
števila niti, se pri uporabi 20 ali več niti čas izvajanja nekonsistentno poveča. Nato smo preverili, če se slučajno kaj podobnega dogaja še v implementacijah drugih jezikov. Kot je razvidno iz slike 5.12, se anomalija pri jeziku C++ ne pojavi. Preverili smo tudi za Chapel in X10, kjer se *tudi* ne pojavi.



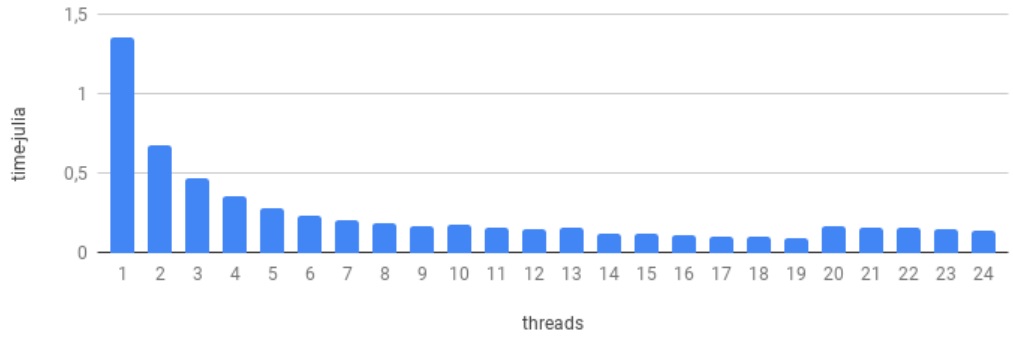
Slika 5.12: Meritve časov izvajanja programske rešitve jezika C++ za problem Seam carving pri različnem številu niti.

Ko se kaj takega programerju zgodi, je visoka verjetnost, da se gre za napako v lastni kodi. Ampak pred razhroščevanjem smo se odločili, da bomo preverili, če se anomalija *v kakšni obliki* pojavlja tudi v problemu *generiranja Mandelbrotove množice* ali se to ekskluzivno dogaja pri problemu *Seam carving*.

Zagnali smo torej še programske rešitve *generiranja Mandelbrotove množice* z različnim številom niti za jezik C++ (meritve na sliki 5.13) ter za jezik Julia (meritve na sliki 5.14). Kot je iz meritev razvidno, se čas izvajanja za rešitev



Slika 5.13: Meritve časov izvajanja programske rešitve jezika C++ za problem Mandelbrot pri različnem številu niti.

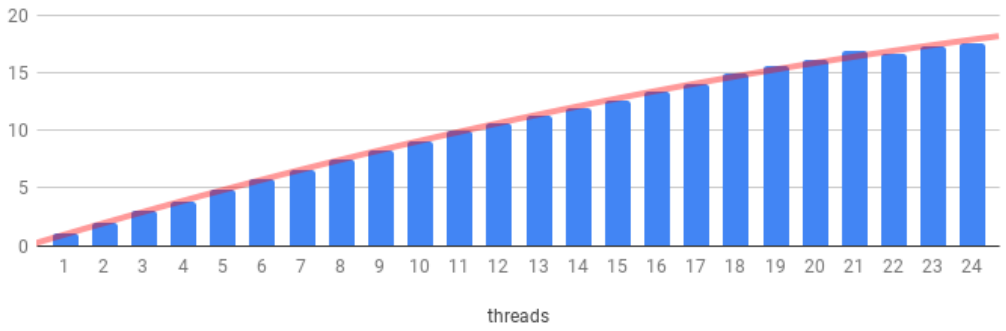


Slika 5.14: Meritve časov izvajanja programske rešitve jezika Julia za problem Mandelbrot pri različnem številu niti.

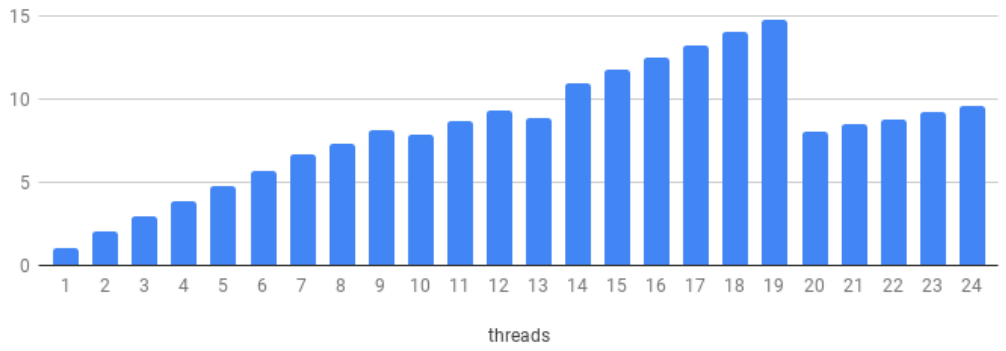
v jeziku C++ konsistentno zmanjšuje z večanjem števila niti. Medtem ko se v Jeziku Julia anomalija znova pokaže pri 20 ali več nitih, čeprav nekoliko manj očitno.

Ta anomalija je še najbolj razvidna, če namesto časov izvajanja primerjamo pohitritev izvajanja programske rešitve pri različnem številu niti. Na sliki

5.15 najdemo meritve za jezik C++, na sliki 5.16 pa meritve za jezik Julia. Po teh ugotovitvah je jasno, da slabe performanse pri programski



Slika 5.15: Pohitritev izvajanja programske rešitve jezika C++ za problem Mandelbrot pri različnem številu niti.



Slika 5.16: Pohitritev izvajanja programske rešitve jezika Julia za problem Mandelbrot pri različnem številu niti.

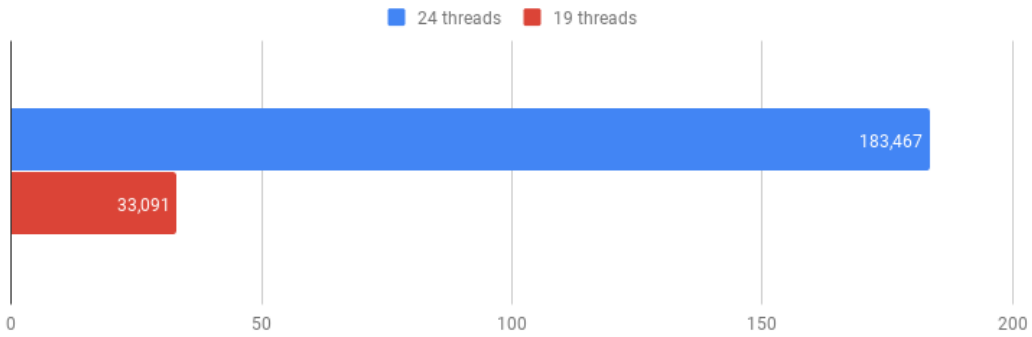
rešitvi problema *Seam carving* v jeziku Julia niso odvisne od naše programske implementacije, saj se anomalija pojavi tudi pri implementaciji problema *generiranja Mandelbrotove množice*.
Razlog za anomalijo, ki se pojavi pri 20 in več nitih verjetno tiči v te, da interpreter *Julia*, vzporedno z našo programsko rešitvijo, izvaja tudi druge notranja opravila, kot so na primer čiščenje pomnilnika, prevajanje/optimiziranje kode med izvajanjem, in zato potrebuje nekaj jeder procesorja. Hipo-

tezo smo preverili tako, da smo z različnim številom niti zagnali interpreter in pri tem opazovali uporabo CPE (s programom `htop`). Če bi bila hipoteza pravilna, bi moral interpreter uporabiti več niti, kot je uporabniško določeno. Posledično bi pričakovali, da je uporaba CPE večja kot maksimalna uporaba, ki jo lahko zahteva izvajanje z določenim številom uporabniških niti. Na primer, maksimalna uporaba 4 niti na 8 jedrih je $4 \times 100\%$ CPE, oz. 400% CPE skupno. Če bi bila hipoteza pravilna, bi za 4 niti morala biti uporaba CPE večja od 400% CPE skupno, ker bi se pokazala tudi poraba za interpreterske niti.

Po prvih meritvah smo hipotezo zavrnili, saj je v splošnem veljalo, da uporaba CPE sledi formuli $n \times 100\%$, kje je n število niti. Pri tem smo ponovno opazili, da se anomalija v meritvah pojavi pri uporabi 20 (ali več) niti. Saj se pri tem številu niti, kot je razvidno iz meritev na sliki 5.18, uporaba CPE nekonsistentno obnaša.

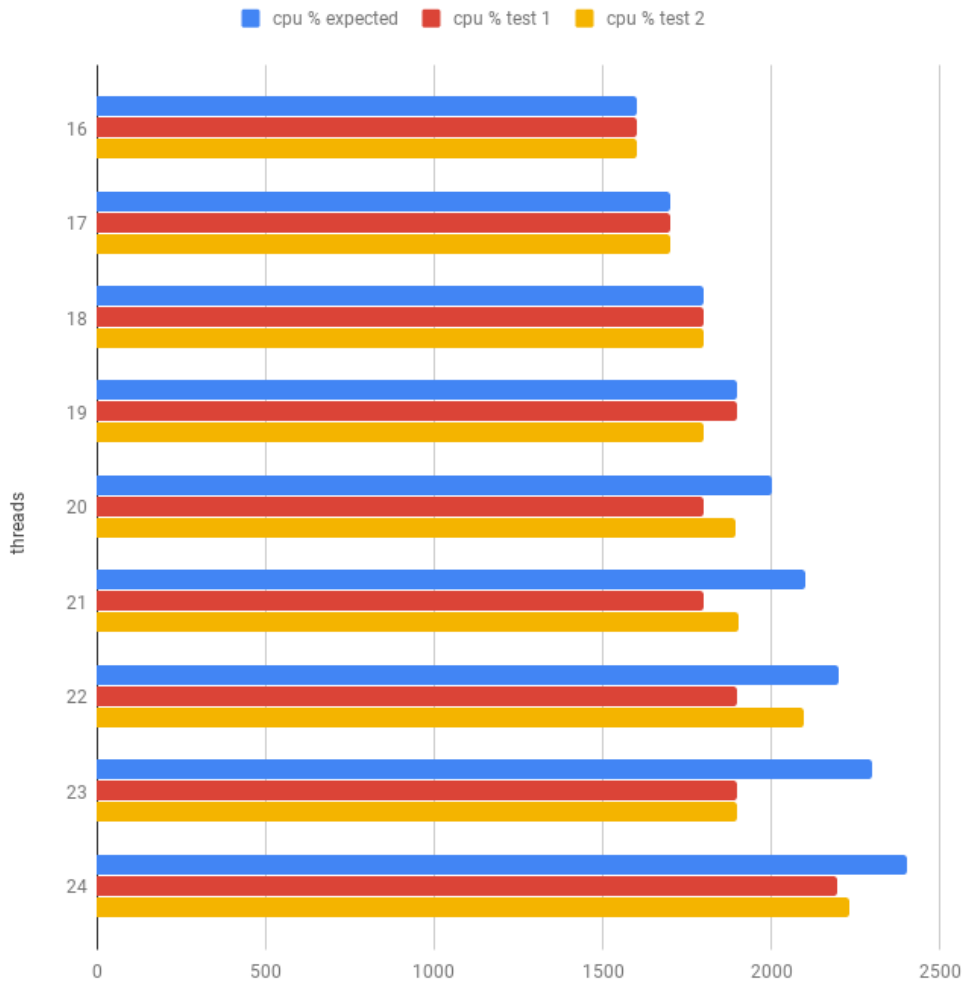
To pomeni, da zaradi *nam neznanega razloga* interpreter jezika Julia ne zmore izkoristiti vse moči, ki mu je dana na razpolago.

Po vseh teh ugotovitvah sklepamo, da je problem v implementaciji interpreterja jezika Julia in ne napačna uporaba jezika z naše strani. Zaradi tega smo se odločili, da bomo interpreter izvajali s takim številom niti, da so pohitritve izvajanja čim večje možne. To se zgodi pri uporabi 19 niti. Meritve izboljšave izvajanja programske rešitve problema *Seam carving* za jezik Julia so prikazane na sliki 5.17.



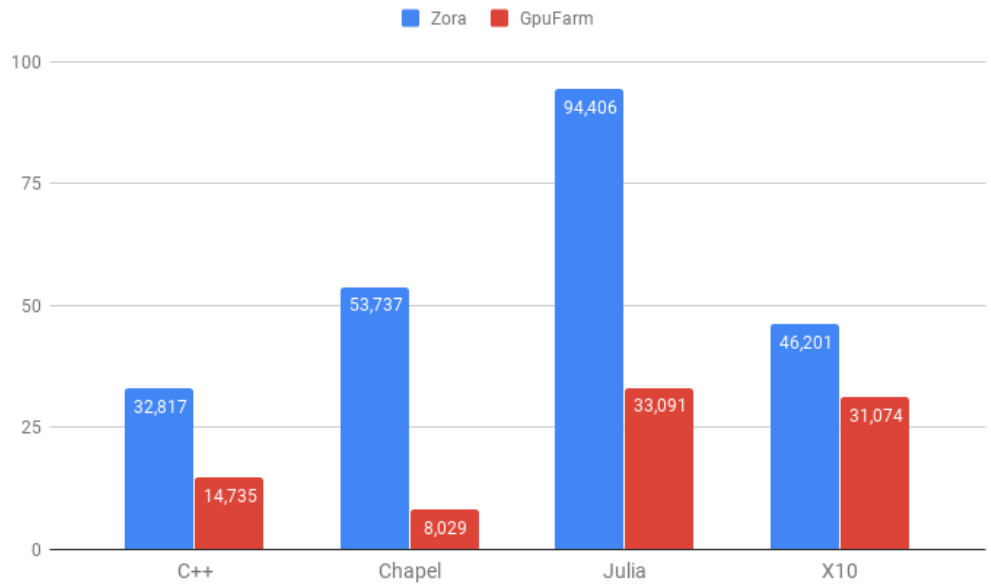
Slika 5.17: Meritve časov izvajanja programske rešitve jezika Julia za problem Seam carving z uporabo 24 ter 19 niti.

Meritve časov izvajanja vseh programskih rešitev za problem Seam carving, kjer interpreter jezika Julia izvajamo z 19 niti je prikazana na 5.19.

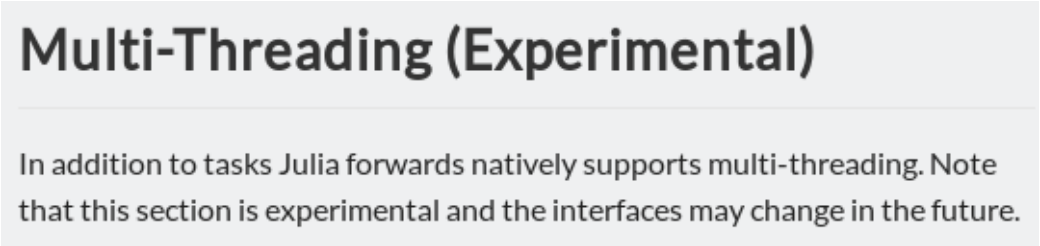


Slika 5.18: Uporaba CPE programske rešitve jezika Julia za problem Mandelbrot pri različnih velikosti bazena niti.

Uradna spletna stran jezika Julia opozarja, da je podpora za večnitno programiranje še vedno eksperimentalna. To lahko vidimo na sliki 5.20. To dejstvo nas še dodatno prepriča, da je eksperimentalna implementacija interpreterja kriva za anomalije v hitrosti izvajanja.



Slika 5.19: Meritve časov izvajanja programskih rešitev za problem Seam carving (Julia na 19 niti).



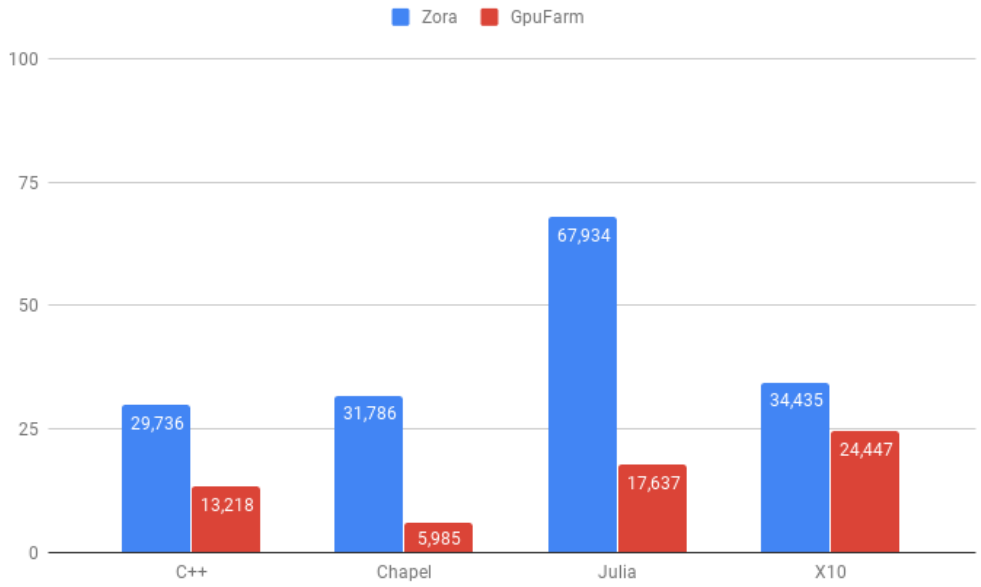
Slika 5.20: Predel uradne spletne strani jezika Julia, kjer piše, da je podpora za večnitno programiranje še eksperimentalna.

Optimizacija ponovne uporabe alociranih tabel

Med izvajanjem je programska rešitev v jeziku X10 na standardni izhod izpisovala opozorilo:

```
GC Warning: Repeated allocation of very large block..
May lead to memory leak and poor performance.
```

Programske rešitve v vseh jezikih so bile take, da je pri vsaki iteraciji krčenja



Slika 5.21: Meritve časov izvajanja programskih rešitev za problem Seam carving pri ponovni uporabi alociranih tabel (pri jeziku Julia uporabimo le 19 niti na računalniku *GpuFarm*).

slike za eno piko prišlo do alokacije dveh *relativno velikih* tabel (približno 70MB za vsako). Torej je bilo med celotnim izvajanjem, pri krčenju slike za 200 pik, skupno alociranih približno 28GB. Seveda lahko taka praksa moti upravljanje pomnilnika s strani jezika in vodi v počasnost.
Rešitev za to je preprosta: enkrat alociramo tabeli in te ponovno uporabimo v vseh naslednjih iteracijah. To optimizacijo smo dodali vsem programskim rešitvam. Meritve izvajanja so prikazane na sliki 5.21.

Pri tem opazimo, da se čas izvajanja izboljša pri vseh programskih rešitvah.

5.3 Komentarji

5.3.1 O performansah

Jezika C++ in Chapel sta se performančno najbolj izkazala. Pri tem je veljalo, da so programske rešitve v jeziku C++ performančno boljše na prenosnem računalniku *Zora*, medtem ko so bile rešitve v jeziku Chapel boljše na strežniku *GpuFarm*. To ugotovitev upravičimo s trditvijo, da je generiranje *strojne kode* s strani prevajalnika jezika Chapel usmerjeno v to, da optimizira performanse na zmogljivejših računalnikih. Medtem ko *privzeto* prevajalnik jezika C++ cilja, da bo izvajanje prevedenega programa performančno čim bolj konsistentno, ne glede na končni računalnik, kjer se bo programska rešitev izvajala.

Za interpretiran jezik je Julia ob *pravilni* uporabi presenetljivo hitra. Kljub temu pa ne more tekmovati s prevedenimi jeziki. Pri tem je pomembno izpostaviti, da performančna krhkost ni nujno posledica dinamične narave jezika. Posledica bi bila lahko *le* ne še optimalna implementacija interpreterja, ali ne optimalno programiranje rešitev z naše strani (saj ni vedno enostavno razumeti, kaj negativno vpliva na performanse jezika Julia). Večkrat se je izkazalo, da smo za doseganje večje hitrosti morali opraviti grde ročne optimizacije kode, ki vodijo v slabe prakse programiranja.

Jezik X10 se performančno ni kaj posebej izkazal, ne v pozitivnem, ne v negativnem.

5.3.2 Zahtevnost programiranja

Večkrat je poleg hitrosti izvajanja pomemben tudi trud oziroma znanje, ki ga nek jezik zahteva od programerja.

Abstraktni nivo programiranja

Za občutek nivoja abstrakcije programiranja sledi za vsak omenjen jezik del kode, ki alocira *dvodimenzionalno* tabelo ter vsaki lokaciji v tabeli priredi neko vrednost.

Primer dela programa v jeziku C++:

```
uint8_t* matrix = new uint8_t[width * height];

#pragma omp parallel for
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width; ++j) {
        matrix[i * width + j] = // ...
    }
}

delete[] matrix;
```

Primer dela programa v jeziku Chapel:

```
const Domain = {1..height, 1..width};
var matrix: [Domain] uint(8);

forall (i, j) in Domain {
    matrix[i, j] = // ...
}
```

Primer dela programa v jeziku Julia:

```
local matrix = Matrix{UInt8}(undef, height, width)
Threads.@threads for j = 1:width
    @simd for i = 1:height
        matrix[i, j] = # ...
    end
end
```

Primer dela programa v jeziku X10:

```
var matrix: Array_2[UByte] =
    new Array_2[UByte](height, width);

finish for (i in 0..(height - 1)) async {
    for (j in 0..(width - 1)) {
        result(i, j) = // ...
    }
}
```

Že iz teh kratkih primerov dobimo občutek nivoja abstrakcije za vsak jezik. Verjetno se glede visoke abstrakcije najbolj izkaže jezik Chapel: enostavno definiramo dvodimenzionalno domeno ter nato za vsak indeks v domeni izračunamo vrednost in jo shranimo v tabelo. Pri tem uporabimo le eno `forall` zanko.
Na nižjem nivoju abstrakcije dobimo rešitvi jezika Julia in X10: v teh primerih uporabimo dve ugnezdeni zanki, vsaka bo iterirala skozi nek specificiran *interval* vrednosti. Jezika (še vedno) nudita abstraktno strukturo dinamično alocirane dvodimenzionalne tabele.
Na zadnje imamo jezik C++: na nivoju sintakse ne podpira dinamičnega alociranja dvodimenzionalnih tabel in zato moramo to napisati sami in uporabiti *računanja z indeksi* (`i * width + j`). Alocirano tabelo moramo ob koncu uporabe ročno sprostiti. Poleg tega jezik ne podpira koncepta *intervalov*.

Zahtevano *računalniško znanje*

Večkrat je za programiranje poleg poznavanja samega jezika potrebno tudi neko minimalno *računalniško znanje*. Med omenjenimi jeziki je C++ definitivno tisti, ki s strani programerja zahteva največ *računalniškega znanja*. Na primer: če želi programer v jeziku C++ karkoli pametnega napisati, bo moral prej ali slej uporabiti *kazalce*. Žal pa globoko razumevanje smisla in

pravilne uporabe kazalcev ni nekaj, kar se lahko nekdo nauči čez noč (razen če si študent FRI-ja in imaš naslednje jutro kolokvij Programiranja 2).
Za druge omenjene jezike velja, da ne zahtevajo posebnega računalniškega znanja s strani programerja. Le s strani jezika X10 se zahteva razumevanje primitivnih tipov. Čeprav računalniško znanje ni zahtevano, lahko seveda taka vrsta znanja bistveno pomaga za doseganje boljših performans.

Dinamično preverjanje napak med izvajanjem

Veliko prednost ki jo imajo jeziki Chapel, Julia in X10 pred jezikom C++, je vgrajeno *dinamično preverjanje napak med izvajanjem*. To jamči višjo kvaliteto programskih rešitev in omogoči hitrejše razhroščevanje. Vzemimo kot primer naslednji scenarij: v programu alociramo tabelo 100 elementov in na lokacijo 111.elementa zapišemo neko vrednost. Problem je seveda ta, da omenjeni element tabele ne obstaja.
Jeziki z dinamičnim preverjanjem se bodo med izvajanjem programa tega zavedali in napako sporočili programerju. Taka vrsta obnašanja je programerju všeč, saj se tako zaveda, da programska rešitev vsebuje napake.
Prava zabava se zgodi tam, ko se omenjeni scenarij pripeti v neki programski rešitvi jezika C++. V *najboljšem* primeru se bo program porušil in s tem na ne konformističen način sporočil programerju, da programska rešitev vsebuje napake. V najslabšem primeru: se bo vrednost zapisala na nek pomnilniški naslov do katerega ima program legitimen dostop, verjetno bo prišlo do prepisa kakšnega podatka, nato se bo izvajanje programa mirno nadaljevalo ne, da bi se programer zavedal napake.
Seveda so taka dinamična preverjanja časovno draga. Zaradi tega vsi omenjeni jeziki dovolijo prevajanje oziroma izvajanje programov z izklopljenim dinamičnim preverjanjem.

Poglavje 6

Sklepne ugotovitve

6.1 Chapel

Verjetno je jezik Chapel tisti izmed omenjenimi, ki ima največ zanimivih ekskluzivnih funkcionalnosti. Performančno se jezik Chapel zelo dobro izkaže, še posebej, ko programske rešitve izvajamo na zmogljivejših računalnikih. Zaradi visoko-nivojske sintakse lahko programer hitro sestavi neko programsko rešitev, ki ima že dobre performanse. Jezik nudi tudi dostop do nižjenivojskih funkcionalnosti, ki jih lahko ob potrebi *računalničar* uporabi, da optimizira izvajanje programskih rešitev. Čeprav je sam jezik konceptualno primeren programerjem vsakega nivoja, je zaradi še nerazvitega ekosistema (na primer: prevajalnik ni prav nič uporabniško prijazen) trenutna uporaba jezika še vedno bolj namenjena predvsem *računalničarjem*. Jezik Chapel bi rad bil alternativa ekosistemu jezika C++ na področju visokoperformančnega programiranja. Rad bi bil in *je* alternativa. Na nivoju sintakse jezika ima dobro podporo za vzporedno programiranje. Veliko prednost ima v produktivnosti: programiranje v jeziku Chapel je lažje in hitrejše kot v jeziku C++. Slabost je seveda bistveno manjše število programerjev v primerjavi z jezikom C++ in v vsem tem kar temu sledi: nizko število zunanjih knjižnic, slabša podpora v razvojnih okoljih, ... Kljub temu ima jezik Chapel v primerjavi z X10 številno in aktivno skupnost programerjev.

6.2 Julia

Nedvoumno je Julia jezik z velikim potencialom. Izrazito se skuša približati zahtevam znanstvenikom, ki niso nujno izkušnjeni programerji. Ima že neko konkretno skupnost in nabor zunanjih knjižnic. Smo mnenja, da je že dobra alternativa matematično usmerjenim jezikom kot MATLAB ali R ter verjetno celo drugim dinamičnim jezikom kot Python. Programska implementacija interpreterja jezika Julia je krasna tehnična umetnina. Ob *določeni uporabi* jezika je hitrost izvajanja res primerljiva prevedenim jezikom. Žal ta *določena uporaba* večkrat zahteva grde ročne optimizacije kode, ki vodijo v slabe prakse programiranja. Čeprav jezik nudi podporo za vzporedno programiranje, se je izkazalo, da ne nudi dostopa, do nižje-nivojskih funkcionalnosti, ki bi si jih *računalničar* želel pri programiranju. Poleg tega smo se pri večnitnem programiranju seznanili z hudimi performančnimi anomalijami s strani interpreterja. Zaradi teh dejstev trdimo, da jezik Julia ni primeren na področjih uporabe, kjer so performanse res pomembne.
Tipičen uporabnik jezika Julia bi lahko bil tak, da rešuje nek problem iz znanstvene domene, nima posebnega računalniškega znanja (saj bi mu to pri Juliji ne pomagalo *veliko*), želi programsko rešitev sestaviti v najkrajšem možnem času, verjetno ima dostop do gruče računalnikov in hoče to moč izkoristit na enostaven način ali pa pri tem ga ne zanima, da so performanse najboljše možne.

6.3 X10

Jezik X10 ni posebno performančno dober. Razen podpore na nivoju sintakse za vzporedno programiranje ne nudi nobene posebne funkcionalnosti. Občasno zna biti programiranje zoprno zaradi različnih razlogov: sama sintaksa jezika, počasen prevajalnik, neobstoječa podpora jeziku strani razvojnih okolji, ... Če skupnost jezika X10 obstaja pomeni, da se ta dobro skriva. Težko si predstavljamo področje uporabe kjer bi lahko bil jezik X10 veljavna alternativa že obstoječim rešitvam.

Dodatek A

Računalniki

A.1 računalnik A: *Zora*

- **Tip:** navaden prenosni računalnik
- **CPE:** AMD FX-7500 (4 fizičnih jeder, 4 niti)
- **RAM:** 8GB
- **GPE:** AMD Radeon R7 M260DX
- **OS:** OpenSUSE Leap 15

A.2 računalnik B: *GpuFarm*

- **Tip:** strežnik z dvema fizičnima procesorjema z skupnim pomnilnikom
- **CPE:** 2× Intel Xeon E5-2620 (6 fizičnih jeder, 12 niti) (skupno 24 niti)
- **RAM:** 64GB
- **GPE:** 2× NVIDIA Tesla K20m
- **OS:** CentOS 6

Literatura

[1] C++ (programming language). https://en.wikipedia.org/wiki/C%2B%2B. [dostopano 2019-01-24].

[2] Chapel ATPESC 2016. https://chapel-lang.org/presentations/ChapelForATPESC2016-presented.pdf. [dostopano 2018-08-24].

[3] Chapel: Multi-Locale execution. https://chapel-lang.org/tutorials/Discovery2015/D2015-5-MULTILOC.pdf. [dostopano 2019-01-18].

[4] Chapel Parallel Iterators: Giving Programmers Productivity with Control. https://www.cray.com/blog/chapel-parallel-iterators-giving-programmers-productivity-with-control/. [dostopano 2019-01-19].

[5] Chapel (programming language). https://en.wikipedia.org/wiki/Chapel_(programming_language). [dostopano 2018-11-21].

[6] Generic GPU Kernels in Julia. https://mikeinnes.github.io/2017/08/24/cudanative.html. [dostopano 2019-01-26].

[7] Implicit parallelism. https://en.wikipedia.org/wiki/Implicit_parallelism. [dostopano 2018-08-23].

[8] Julia - DistributedArray.jl. https://github.com/JuliaParallel/DistributedArrays.jl/blob/master/docs/src/index.md. [dostopano 2019-01-18].

[9] Julia - Standard Library - Distributed Computing. https://docs.julialang.org/en/v1/stdlib/Distributed/index.html. [dostopano 2019-01-18].

[10] Julia Micro-Benchmarks. https://julialang.org/benchmarks/. [dostopano 2019-01-11].

[11] Julia (programming language). https://en.wikipedia.org/wiki/Julia_(programming_language). [dostopano 2018-12-03].

[12] JuliaGPU/CLArrays - OpenCL-backed GPU Arrays. https://github.com/JuliaGPU/CLArrays.jl. [dostopano 2019-01-26].

[13] Mandelbrot. https://en.wikipedia.org/wiki/Mandelbrot_set. [dostopano 2018-12-29].

[14] Message Passing Interface. https://en.wikipedia.org/wiki/Message_Passing_Interface. [dostopano 2019-01-24].

[15] OpenMP. https://en.wikipedia.org/wiki/OpenMP. [dostopano 2019-01-24].

[16] Parallel computing for Julia. https://docs.julialang.org/en/v1/manual/parallel-computing/. [dostopano 2019-01-22].

[17] Performance Tips for Julia. https://docs.julialang.org/en/v1/manual/performance-tips/index.html. [dostopano 2019-01-25].

[18] Quantum computing. https://en.wikipedia.org/wiki/Quantum_computing. [dostopano 2018-11-16].

[19] Speed of electricity. https://en.wikipedia.org/wiki/Speed_of_electricity. [dostopano 2018-11-16].

[20] The APGAS model (constructs). http://x10.sourceforge.net/documentation/intro/latest/html/node4.html. [dostopano 2018-12-07].

[21] TIOBE index. https://www.tiobe.com/tiobe-index/. [dostopano 2019-01-24].

[22] Using julia -L startupfile.jl, rather than machinefiles for starting workers. https://white.ucc.asn.au/2017/08/17/starting-workers.html. [dostopano 2019-01-10].

[23] Wetware computer. https://en.wikipedia.org/wiki/Wetware_computer. [dostopano 2018-11-16].

[24] X10 - Performance Tuning. http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html#PerformanceTuninganX10Application-BestPracticesforPerformanceTuning. [dostopano 2019-02-07].

[25] X10 History. http://x10-lang.org/articles/6.html. [dostopano 2018-12-03].

[26] X10 Introduction. http://x10-lang.org/articles/79.html. [dostopano 2019-01-02].

[27] X10 (programming language). https://en.wikipedia.org/wiki/X10_(programming_language). [dostopano 2018-12-03].

[28] Simon Isaković. *Rezanje šivov*. PhD thesis, Univerza v Ljubljani, 2009.