TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MILAN MÄKIPÄÄ

PYTHON-OHJELMIEN SUORITUSKYVYN PARANTAMINEN

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen
30. heinäkuuta 2018

TIIVISTELMÄ

Milan Mäkipää: Python-ohjelmien suorituskyvyn parantaminen
Tampereen teknillinen yliopisto
Kandidaatintyö, 16 sivua
Heinäkuu 2018
Tietotekniikan tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: Python, PyPy, Cython, suorituskyky

Python on moderni ohjelmointikieli, jolla ohjelmien kehitys on monia muita kieliä nopeampaa, mutta se ei yksinään sovi useimpiin suorituskykykriittisiin käyttökohteisiin. Tämän suorituskykyongelman ratkaisemiseksi on luotu useita ratkaisuja, jotka eroavat toisistaan lähestymistavassaan. Ratkaisut eroavat myös suuresti siinä kuinka paljon alkuperäistä Python-ohjelmaa tulee muokata suorituskykyongelman ratkaisemiseksi.

Tämän työn tarkoitus on vertailla esimerkkiohjelman avulla kolmea eri tapaa, jolla Python-ohjelmien suorituskykyä voidaan parantaa. Esimerkkiohjelmasta tehtyjen toteutuksien suorituskykyä vertaillaan keskenään. Samalla saavutetaan jonkinasteinen käsitys siitä, kuinka paljon alkuperäistä ohjelmaa tulee muokata, jotta voidaan hyödyntää eriratkaisutapoja.

Kaikilla työssä vertailluilla ratkaisuilla saavutettiin merkittävä parannus testiohjelman suorituskykyyn. Cythonilla ja Shed Skinillä saavutettiin parhaat tulokset. PyPyn tuoma suorituskykyparannus hieman heikompi, mutta sen hyödyntämistä varten testiohjelmaan ei tarvinnut tehdä lainkaan muutoksia. Cythonia varten testiohjelmaa tuli muokata eniten, sillä toteutus hyödynsi C++:n tietorakenteita.

SISÄLLYSLUETTELO

1. JOHDANTO

Python on moderni tulkattu ohjelmointikieli. Tulkkaus tarkoittaa sitä, että ohjelmakoodista käännetään ajettava versio vasta ajonaikaisesti. Tästä syystä Python on paljon hitaampi kuin monet sellaiset kielet, jotka käännetään kokonaan ennen niiden suoritusta. [1, s. 39]

Tämän työn tarkoituksena on vertailla muutamaa eri ratkaisua, joilla vähennetään Pythonin ajonaikaisesta tulkkauksesta johtuvaa hitautta. Toteutetussa vertailussa pyritään erittelemään eri työkalujen lähestymistapoja, sillä ne eroavat toisistaan esimerkiksi siinä, käännetäänkö ohjelmaa kokonaan ennen sen suoritusta, vai onko ratkaisu jokin ajonaikaisen kääntämisen paremmin optimoitu muoto.

Työn alussa selvitetään kääntämisen ja tulkkauksen eroja, sekä niistä seuraavia etuja ja haittoja. Tämän jälkeen kuvaillaan muutamaa erilaista ratkaisutapaa, joilla Pythonilla kirjoitettuja ohjelmia saadaan mahdollisesti nopeutettua. Kuvausten jälkeen työssä toteutetaan vertailu, jossa yksinkertaisesta algoritmista sovitetaan versio jokaiselle tutkitulle ratkaisutavalle. Lopuksi tässä työssä vertaillaan sovitusten suorituskykyeroja sekä toistensa, että pelkällä Pythonilla toteutetun version välillä. Samalla arvioidaan myös sitä, kuinka työlääksi alkuperäisen Python-kielisen ohjelman sovittaminen jokaista ratkaisutapaa noudattavaksi osoittautui.

2. TULKATTAVAT JA KÄÄNNETTÄVÄT KIELET

Korkeamman tason ohjelmointikielillä kirjoitettuja ohjelmia ei voida suoraan ajaa prosessoreilla, sillä ne sisältävät vain tietyn rajallisen ryhmän käskyjä, joita ne voivat käsitellä [2, s. 24]. Ohjelmia ei ole kuitenkaan yleensä järkevää kirjoittaa suoraan prosessorien ymmärtämillä konekielillä. Konekielisten ohjelmien luettavuus ja kirjoitettavuus ovat paljon heikompia, kuin korkeamman tason ohjelmointikielillä kirjoitettujen ohjelmien. Tästä johtuen todennäköisyys sille, että kirjoitettuun ohjelmakoodiin päätyy virheitä kasvaa ja ohjelman ylläpidettävyys heikkenee. [2, s. 16]

Jotta korkean tason ohjelmointikielellä kirjoitettua ohjelmaa voitaisiin ajaa prosessorilla, on se käännettävä prosessorin ymmärtämään muotoon. Kääntäjät ovat kieli- ja alustakohtaisia ja tuottavat korkeamman tason ohjelmakoodin perusteella käytetyn järjestelmän tai alustan ymmärtämän toteutuksen ohjelmasta. [2, s. 24]

Ohjelmointikielten kääntäjillä on useita erilaisia toteutuksia, joilla on kaksi ääripäätä. Toisessa ohjelmat käännetään joukoksi konekielisiä käskyjä, jotka voidaan suorittaa prosessorilla. Tämä käännös tapahtuu ennen ohjelman suoritusta. Toisessa ääripäässä ovat ajonaikaisesti käännettävät ohjelmointikielet, joita käännetään ajonaikaisesti. [2, ss. 24–29]

Pelkän kääntämisen tai tulkkauksen lisäksi joissain ohjelmointikielissä käytetään niiden yhdistelmää. Niissä ohjelmaa ei käännetä heti konekieliseksi, vaan johonkin toiseen matalamman tason kieleen. Tätä toista kieltä käytetään, sillä sen kääntäminen konekieliseksi ajonaikaisesti on nopeampaa kuin käännös suoraan siitä kielestä, jolla ohjelma on alun perin kirjoitettu. Tällaisella ratkaisulla saavutetaan osa sekä käännettävien, että tulkattavien ohjelmointikielien eduista. [2, ss. 24-29]

2.1 Kääntäminen ja tulkkaus

Ohjelman kääntäminen tapahtuu ennen ohjelman suoritusta erikseen jokaiselle järjestelmälle. Jos käännöksen luontiin kuluva aika ei ole tärkeää, kääntäjä voi pyrkiä tekemään ohjelmaan suorituskykyä parantavia optimointeja, jolloin lopputuloksena syntyneen käännetyn ohjelman suorituskyky paranee. Käännetyn ohjelman suorituskykyä rajoittaa ensisijaisesti tietokoneen muistin ja prosessorin välisen yhteyden hitaus. [1, ss. 24–27]

Välillä ennen suoritusta tapahtuva käännös konekieliseksi saattaa ensin tuottaa käännöksen aikana myös muun kuin konekielisen ohjelman. Tämän käännöksen perusteella luodaan lopullinen konekielinen käännös. Toteutus eroaa kuitenkin ajonaikaisesta käännöksestä siten, että ohjelma käännetään kokonaan ennen. [1, ss. 24–27]

Ajonaikaiset kielet eroavat ennen suoritusta käännettävistä kielistä siinä, että niitä ei ajeta suoraan prosessorilla vaan jonkin alustan, kuten virtuaalikoneen, päällä. Ohjelmaa suoritetaan tämän alustan päällä samankaltaisesti kuin konekielistä ohjelmaa prosessorilla, mutta käskyt ovat korkeamman tason käskyjä, joista tulkataan prosessorille annettavat käskyt ajonaikaisesti. [2, s. 29]

2.2 Etuja ja haittoja

Käännettävien kielten suurin etu on se, että niillä kirjoitettujen ohjelmien nopeus on usein paljon tulkattavia parempi. Tämä johtuu siitä, että kääntäjä voi tuottaa ohjelmakoodiin optimointeja, joilla suorituskykyä saadaan parannettua. Tulkattavissa kielissä tämä ei usein ole mahdollista, sillä ohjelmakoodin ajonaikainen kääntäminen on ohjelman suoritukseen eniten vaikuttava tekijä, ja optimoinnit hidastaisivat sitä. [2, s. 27, 29]

Tulkattavat kielet hyötyvät ajonaikaisesta kääntämisestä siten, että niillä tehtyjä ohjelmia voidaan ajaa millä tahansa alustalla, jolle löytyy sopiva versio kääntäjästä. Useissa tapauksissa samaa ohjelmaa voidaan suorittaa eri ympäristöissä ilman minkäänlaisia muutoksia ohjelmakoodiin. Tästä johtuen tulkattavilla kielillä tehtyjen ohjelmien siirrettävyys eri alustojen välillä on helpompaa. Tämä saattaa myös vähentää ohjelman kehitykseen kuluvaa työmäärää, sillä ohjelmaa voidaan testata erilaisessa ympäristössä, kuin se, johon valmis ohjelma päätyy. [2, s. 33]

Ajonaikainen kääntäminen mahdollistaa myös ohjelmakoodin yksinkertaistamisen, sillä tällaisista kielistä osa tukee ominaisuuksia, kuten dynaamista tyyppitarkastusta. Se mahdollistaa yksittäisen luokan tai funktion hyödyntämistä useammalle eri tietotyypille, jolloin ohjelman kehitys on nopeampaa. [1, s. 1232]

Ajonaikaisesti käännettävien kielten heikkouksia ovat käskyjen dekoodauksen hitaus ja ohjelmien koko muistissa konekielisiin ohjelmiin verrattuna. Tulkattavan ohjelman suorituskyvyn ensisijainen rajoite ei ole muistin ja prosessorin välinen yhteys, vaan tämän dekoodauksen hitaus. [2, s. 29]

3. PYTHON

Python on yleiskäyttöinen korkean tason tulkattava ohjelmointikieli. Yksi sen suunnitteluperiaatteista on ohjelmien kehitykseen kuluvan ajan minimointi. Tätä kuvastaa muun muassa se, että Python-ohjelmissa ei käytetä tyyppimäärittelyitä, sekä se, että Python sisältää automaattisen muistinhallinnan. Pythonin syntaksi on myös tarkoituksella helposti luettavaa, jotta sillä kirjoitettujen ohjelmien ymmärtäminen ja ylläpito olisi helpompaa. [1, s. 32, 36]

3.1 Alustariippumattomuus

Python on tulkattava kieli, joten sillä kirjoitettuja ohjelmia voidaan ajaa ajonaikaisen tulkin avulla suoraan lähdekooditiedoston avulla. Tästä johtuen Python-kieliset ohjelmat ovat yleensä lähes täysin alustariippumattomia, eli samaa ohjelmaa voidaan ajaa hyvin erilaisilla järjestelmillä. Ainoastaan tulkista täytyy löytyä käytetylle alustalle sopiva versio. [1, s. 33]

Python-ohjelma voi olla riippuvainen käytetystä järjestelmästä, jos siinä hyödynnetään vain tietylle käyttöjärjestelmälle saatavissa olevia ominaisuuksia. Suuri osa Pythonissa käytettävistä apukirjastoista on kuitenkin alustariippumattomia ja monet niistä voidaan asentaa käyttämällä Pythonin omaa paketinhallintajärjestelmää. [3]

3.2 Hitaus

Pythonilla toteutetut ohjelmat ovat, kuten muilla tulkatuilla kielillä toteutetut, hitaampia kuin käännetyillä kielillä toteutetut ohjelmat. Syyt kielen hitauteen ovat samoja kuin kaikilla muillakin tulkattavilla kielillä. Ajonaikainen tulkkaus on siis ensisijainen suorituskykyä rajoittava tekijä. [1, s. 39]

Suorituskykyongelmien ratkaisemiseksi python mahdollistaa esimerkiksi C:llä toteutettujen ohjelman osien hyödyntämisen muun Python-ohjelman laajennuksina. Tällä tavalla voidaan toteuttaa ohjelman suorituskykykriittisimmät osat C:llä ja hallinnoida niitä Pythonin avulla. Tällä tavalla saavutetaan osa Pythonin tuomasta ohjelman kehitysajan vähentämisestä, mutta saavutetaan myös mahdolliset suorituskykyvaatimukset. [1, ss. 39–40]

Esimerkki suorituskykyisistä laajennuksista, joita voidaan hyödyntää Pythonin avulla, on NumPy. Se sisältää etukäteen käännettyjä funktioita, joilla voidaan tehdä erinäisiä matemaattisia operaatioita hyvin nopeasti. NumPyn kaltaisen apukirjaston hyödyntämisellä

ohjelmassa vältetään myös tarve kirjoittaa käytettyjä funktioita itse. Tässä työssä ei kuitenkaan käsitellä NumPyn kaltaisia apukirjastoja, vaan keskitytään työkaluihin, joilla olemassaolevasta Python-ohjelmasta saadaan muunnettua nopeampi versio. [4, s. 1450]

Pythonia voidaan hyödyntää myös prototyyppien toteuttamiseen ohjelmista. Sen mahdollistaman lyhyemmän kehitysajan avulla voidaan valita paras toteutustapa, joka toteutetaan uudelleen jollain muulla suorituskykyisemmällä kielellä. [1, s. 40]

4. ERI TYÖKALUJA TULKKAUKSEN HITAUDEN VÄLTTÄMISEEN

Pythonin tulkin hitauteen on olemassa useita erilaisia ratkaisuja. Niistä on valittu vertailtavaksi muutama sellainen, jotka edustavat erilaista lähestymistapaa ongelman ratkaisuun. Valittujen ratkaisujen lisäksi olisi ollut monia muitakin vaihtoehtoja, mutta monet niistä eivät ole saaneet päivityksiä pitkään aikaan, ja näin tuki esimerkiksi Pythonin kolmannelle versiolle puuttuu. Esimerkki tällaisesta on Psyco, jonka viimeisin versio on julkastu vuonna 2007. Uusin Psycon kanssa yhteensopiva Pythonin versio on 2.6. Psyco ei myöskään tue lainkaan 64-bittisiä järjestelmiä. [5]

4.1 Cython

Pythonin CPython-tulkki sallii Pythonilla kirjoitetun ohjelmakoodin lisäksi erillisten C-kielisten laajennuksien kutsumisen Python-kielisestä ohjelmasta käsin. Cython hyödyntää tätä ominaisuutta siten, että sen kääntäjä luo Cythonilla kirjoitetuista osista ohjelmasta, kun se on mahdollista, C tai C++ -koodia, jota muut ohjelman osat voivat kutsua kuten muitakin C-laajennuksia. [6]

Cythonin tarkoitus on olla laajennus pelkkään Pythoniin nähden. Tämän seurauksena Cythonilla voidaan kääntää jo olemassa olevia Python-ohjelmia. Merkittävien suorituskykyparannuksien saavuttamiseksi ohjelmakoodia on kuitenkin muokattava hyödyntämään esimerkiksi C-kielen omia muuttujatyyppejä, sillä tällöin Cythonin kääntäjä voi tehdä muokatusta osasta todennäköisemmin käännöksen C-kieliseksi ennen ohjelman suoritusta. [6]

Kaikkia ohjelman osia ei ole pakko muuntaa käyttämään Cythonia, vaan sen sisältämä kääntäjä kääntää staattisesti vain ne ohjelman osat, joille käännös on mahdollista. Loppuosa ohjelmasta ajetaan normaalisti CPython-tulkin avulla. [6]

Ohjelman osittainen staattinen käännös mahdollistaa sen, että ohjelmasta muunnetaan Cythoniksi vain ne osat, joiden nopeuttaminen parantaa suorituskykyä eniten. Tällöin säilytetään osa ohjelman kehityksen vaivattomuuden eduista, jotka Pythonin käytöllä saavutetaan, mutta suorituskyvyn kannalta kriittisimmissä osissa vältetään tulkkauksen haitat. [7, s. 32]

4.2 PyPy

PyPy eroaa Cythonista eniten lähestymistavassaan koko hitausongelman ratkaisuun. Siinä missä Cython pyrkii tuottamaan ohjelmasta käännöksen mahdollisuuksien mukaan

ennen suoritusta PyPy:n tavoite on Pythonin omaan CPython-tulkkiin nähden nopeuttaa tulkkausta Pythonin ja ajettavan konekielisen version välillä.

PyPy on vaihtoehtoinen toteutus Python-kielestä, jonka tulkki korvaa pythonin oman CPython tulkin. PyPy:n tulkki on kirjoitettu C:n sijaan RPythonilla, joka on rajoitettu versio Pythonista. [8] Siitä puuttuu joitain Pythonissa olevia ominaisuuksia, mutta muuten ne ovat yhteensopivia, eli CPython-tulkilla on mahdollista ajaa RPythonilla kirjoitettuja ohjelmia. [9] Se, että tulkki on kirjoitettu RPythonilla C:n sijaan, on mahdollistanut tulkin monimutkaisempien ominaisuuksien toteuttamisen korkeamman tason ohjelmointikiellellä. [8].

PyPy:n tulkki eroaa CPythonista siinä, että sisältää JIT tai "just-in-time"-tyyppisen tulkin. JIT-tulkki tekee käännöksen korkeamman tason kielestä matalamman tason kielelle samalla tavalla kuin normaali tulkki, mutta tulkkauksen tuloksia pidetään tallessa siltä varalta, että samaa komentoa pyritään myöhemmin suorittamaan myöhemmin. JIT-tulkin käyttö lisää tämän takia jonkin verran ohjelman resurssien kulutusta. [10]

JIT-tulkin käytöstä johtuen PyPy:ä hyödyntävät ohjelmat saavat suurimman suorituskykyhyödyn sellaisissa tilanteissa, joissa ohjelma ajaa eniten Pythonilla kirjoitettuja osia ohjelmasta. C-kielisiä laajennuksia ei käännetä ajonaikaisesti, jolloin JIT-tulkilla ei saavuteta hyötyä niiden kanssa. PyPy ei myöskään juurikaan auta hyvin pienten lyhyen aikaa ajettavien ohjelmien suorituskykyä. Tämä johtuu siitä, että JIT-tulkki ei voi hyödyntää tallentamiaan tulkkauksen tuloksia, jos käskyä ajetaan ensimmäistä kertaa. [8]

Tässä työssä vertailuun käytetyksi algoritmiksi on valittu sellainen, että PyPy:n pitäisi tuottaa huomattava suorituskykyparannus. Algoritmi käyttää suuren osan ajastaan Pythonilla kirjoitetuissa silmukoissa, joiden kanssa JIT-tulkista pitäisi olla huomattavasti apua.

4.3 Shed Skin

Shed Skin on kääntäjä, joka tuottaa Pythonkoodista C++ -koodia. Käännöksen tulosta käytetään kuten tavallista C++ -ohjelmaa eli se käännetään konekieliseksi ennen ohjelman suoritusta. Käännös C++:aan on mahdollista, sillä Shed Skin korvaa pythonin tietotyypit omilla toteutuksillaan. Näistä tietotyypeistä saadaan muunnettua C++:n vastaavat käännöksen aikana. [11] Shed Skinin dokumentaation mukaan Python-ohjelman suorituskyky saattaa kääntäjää käyttämällä parantua 2-200 –kertaiseksi CPythonilla tulkattuun versioon verrattuna [12].

Shed Skiniä varten kirjoitettu koodi on yhteensopivaa CPython-tulkin kanssa, mutta siihen kohdistuu joitain rajoitteita. Esimerkiksi tietotyypit ovat C++:n kaltaisia. Tämä tarkoittaa sitä, että ohjelmassa käytettyihin muuttujiin ei voida ohjelman suorituksen aikana

sijoittaa erityyppisiä arvoja. Shed Skinin kanssa ei voi myöskään käyttää kaikkia Pythoniin sisältyviä kirjastoja, vaan vain osa niistä on kääntäjän kanssa yhteensopivia. [11]

Uusin Pythonin versio, jota Shed Skin tukee on 2.7 [11]. Tämä tarkoittaa sitä, että Python3 –versioilla kirjoitetut ohjelmat tulee muuntaa yhteensopiviksi 2.7 –version kanssa. Tässä työssä vertailuun käytetyn ohjelman muuttaminen on yksinkertaista, sillä ohjelma on lyhyt ja yksinkertainen, mutta laajojen ohjelmien muuttaminen voi olla aikaavievää. Pidemmällä aikavälillä Shed Skinin ongelmaksi voi nousta se, että Python2:n tuki on taattu vain vuoteen 2020 asti [13]. Tämän takia sen käyttö ei ole välttämättä järkevää silloin, jos Shed Skiniä hyödyntävää ohjelmaa käytetään pitkään.

5. TYÖKALUJEN VERTAILU

Tämä luku sisältää kuvauksen eri työkalujen vertailuun käytettävästä algoritmista. Luvussa kuvataan myös testialusta ja esitellään vertailun tulokset. Lopuksi saatuja tuloksia verrataan muussa kirjallisuudessa esiintyviin tuloksiin työssä käsiteltyjen työkalujen osalta.

5.1 Vertailuun käytetyn ohjelman ja alustan kuvaus

Työssä käsiteltävien ratkaisutapojen vertailuun toteutettiin mergesort-järjestysalogritmi, joka järjestää lukuarvoja suuruusjärjestykseen. Tämä algoritmi valittiin testiohjelmaa varten siksi, että tarpeeksi suurella käsiteltävien alkioiden määrällä eri ratkaisuiden suorituskykyerot tulevat hyvin näkyviin.

Ohjelma 1 sisältää ainoastaan Pythonia hyödyntävän verrokkiohjelman, johon muita ratkaisuja verrataan. Ohjelma luo aluksi miljoonan alkion listan kokonaislukuja, jotka järjestetään sen jälkeen nousevaan suuruusjärjestykseen. Ohjelman suorituksen seurauksena tulosteena saadaan aika, joka on käytetty alkioiden järjestämiseen. Alkuperäisen järjestämättömän listan luontia ei ole huomioitu mitatussa ajankäytössä.

PyPy:ä ja Shed Skiniä hyödynnettäessä ei tarvinnut tehdä muutoksia alkuperäiseen Python-ohjelmaan. Se, että Shed Skiniä käytettäessä ohjelma oli yhteensopiva Python2:n kanssa johtuu siitä, että siinä hyödynnettiin vain sellaisia apukirjastoja, jotka ovat sopivia sekä Pythonin toisen, että kolmannen version kanssa.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import random
import datetime
import sys
sys.setrecursionlimit(1500)

def generateIntegerList(size):
	result = []
	for i in range (0, size):
		result.append(random.randint(0, 10000000))
	return result

def merge(numbers, left, right, mid):
	backUp = []
	for i in range (left, right+1):
		backUp.append(numbers[i])
	index = left
	leftIndex = left
	leftIterations = 0
	rightIndex = mid+1
	rightIterations = rightIndex - leftIndex
	while (leftIndex <= mid and rightIndex <= right):
		if (backUp[leftIterations] <= backUp[rightIterations]):
			numbers[index] = backUp[leftIterations]
			leftIndex +=1
			leftIterations += 1
		else:
			numbers[index] = backUp[rightIterations]
			rightIndex +=1
			rightIterations += 1
		index += 1
	while (leftIndex <= mid):
		numbers[index] = backUp[leftIterations]
		leftIndex += 1
		leftIterations += 1
		index += 1
	while (rightIndex <= right):
		numbers[index] = backUp[rightIterations]
		rightIndex += 1
		rightIterations += 1
		index += 1

def mergeSort(numbers, left, right):
	if (left < right):
		mid = (left+right) // 2;
		mergeSort(numbers, left, mid)
		mergeSort(numbers, mid+1, right)
		merge(numbers, left, right, mid);
	return

def main(args):
	lista = generateIntegerList(1000000)
	start = datetime.datetime.now()
	mergeSort(lista, 0, len(lista)-1)
	end = datetime.datetime.now()
	diff = end-start
	print(diff)

if __name__ == '__main__':
	sys.exit(main(sys.argv))
```

Ohjelma 1. Pelkällä Pythonilla toteutettu mergesort-algoritmi.

```
import random
import datetime
import sys
from mergesort import *
from libcpp.vector cimport vector
sys.setrecursionlimit(1500)

cdef vector[int] generateIntegerVector(size):
        cdef vector[int] result
        for i in range (0, size):
                result.push_back(random.randint(0, 10000000))
        return result

cdef void merge(vector[int]& numbers, int& left, int& right, int& mid):
        cdef vector[int] backUp
        for i in range (left, right+1):
                backUp.push_back(numbers[i])
        cdef int index = left
        cdef int leftIndex = left
        cdef int leftIterations = 0
        cdef int rightIndex = mid+1
        cdef int rightIterations = rightIndex - leftIndex
        while (leftIndex <= mid and rightIndex <= right):
                if (backUp[leftIterations] <= backUp[rightIterations]):
                        numbers[index] = backUp[leftIterations]
                        leftIndex +=1
                        leftIterations += 1
                else:
                        numbers[index] = backUp[rightIterations]
                        rightIndex +=1
                        rightIterations += 1
                index += 1

        while (leftIndex <= mid):
                numbers[index] = backUp[leftIterations]
                leftIndex += 1
                leftIterations += 1
                index += 1
        while (rightIndex <= right):
                numbers[index] = backUp[rightIterations]
                rightIndex += 1
                rightIterations += 1
                index += 1

cdef void mergeSort(vector[int]& numbers, int left, int right):
        if (left < right):
                mid = (left+right) // 2;
                mergeSort(numbers, left, mid)
                mergeSort(numbers, mid+1, right)
                merge(numbers, left, right, mid);
        return

def sort():
        cdef vector[int] vec = generateIntegerVector(1000000)
        start = datetime.datetime.now()
        mergeSort(vec, 0, vec.size()-1)
        end = datetime.datetime.now()
        diff = end-start
        print(diff)

sort()
```

Ohjelma 2. Ohjelmakoodi, josta muodostetaan Pythonilla hyödynnettävä Cythonmoduuli

Ohjelma 2 sisältää ohjelmasta 1 muunnetun Cythonkoodin. Ohjelma on hyvin samankaltainen kuin pelkällä Pythonilla toteutettu versio, mutta siinä on hyödynnetty C++:n tietorakenteita ja funktiotyyppejä sellaisissa kohdissa, kun se on ollut mahdollista. Ohjelmasta 2 muodostetaan pythonlaajennus, joka suoritetaan kutsumalla sitä tavallisesta pythonohjelmasta käsin.

Työssä testialustana käytettiin tietokonetta, jossa oli prosessorina Intelin Xeon e5640, 12 gigatavua keskusmuistia ja käyttöjärjestelmänä Debian 9. Käytetyn Pythontulkin versio oli 3.5.3, PyPy:n 6.0.0, Shed Skinin 0.9.4 ja Cythonin 0.26.1. Shed Skinin ja Cythonin käytetyt versiot olivat ne, jotka löytyivät Debianin käytetyn version pakettivarastosta. PyPystä käytettiin uusinta sen kotisivuilta saatavilla olevaa versiota. Shed Skin ei ole yhteensopiva Pythonin 3. version kanssa, joten sitä hyödynnettäessä käytettiin Pythonin versiota 2.7.13

PyPyä käytettäessä ohjelmaa ajettiin aivan kuten Python-ohjelmaa muutenkin, ohjelmapolku annettiin parametrina CPython tulkin sijaan PyPyn tulkille. Shed Skiniä käytettiin antamalla Python-ohjelma parametrina sitä kutsuttaessa. Shed Skin muodosti tarvittavat C++ otsikko- ja lähdekooditiedostot sekä Makefilen. Sen avulla saatiin käännettyä binääritiedosto, josta ohjelma ajettiin. Cythonia käytettäessä ajettiin erillinen määritystiedosto, jonka avulla Cython muodosti ohjelmasta C++ version, jota voitiin hyödyntää yhdessä tavallisen Python-ohjelman kanssa.

5.2 Tulokset

Vertailtava ohjelma suoritettiin jokaista ratkaisua käyttäen kymmenen kertaa ja suoritusajoista laskettiin keskiarvo. Tulokset on esitetty taulukossa 1. Taulukossa Suoritusajan parantuminen on laskettu laskemalla suoritusajan erotus CPython-toteutuksen ja tutkittavan ratkaisun välillä ja jakamalla saatu tulos CPython-toteutuksen suoritusajalla. Tämä sarake on laskettu, sillä sen avulla voidaan vertailla tuloksia muussa kirjallisuudessa saatuihin tuloksiin.

Taulukko 1. Testiohjelmien keskimääräiset suoritusajat ja suorituskykyparannukset eri ratkaisuilla.

Käytetty ratkaisu	**Suoritusaikojen keskiarvo (s)**	**Suoritusajan parantuminen CPython-toteutukseen nähden (%)**
CPython	21,75536	0,0
PyPy	1,279462	94,1
Cython	0,726605	96,7
Shed Skin	0.713426	96,7

Tuloksista nähdään, että kaikilla ratkaisuilla saavutettiin suuri parannus ohjelman suorituskykyyn. Cythonin ja Shed Skinin testien tulokset ovat lähestulkoon identtisiä. Tämä vaikuttaa järkevältä, sillä niillä molemmilla on samankaltainen lähestymistapa suorituskykyongelman ratkaisuun. Merkittävin ero on se, että Cythonia hyödynnettäessä testialgoritmia kutsuttiin Python-ohjelmasta käsin, mutta Shed Skinillä ohjelmasta muodostettiin ainoastaan binääritiedosto.

Taulukko 2. Keskimääräinen suorituskykyparannus eri ratkaisuilla tässä työssä saatujen tulosten kanssa vertailua varten. Muunnettu lähteestä 12.

Käytetty ratkaisu	**Suoritusaikojen keskiarvo (s)**
PyPy	17,7
Cython	12,3
Shed Skin	59,5

Talulukossa 2 on esitetty muussa kirjallisuudessa esiintyviä tuloksia, kun on testattu tässä työssä tarkasteltujen työkalujen lisäksi muitakin suorituskykyä parantavia ratkaisuja. Verrokkitulokset eroavat huomattavasti tässä työssä saaduista arvoista. Eroa selittää todennäköisesti eniten se, että tässä työssä on käytetty eri algoritmia ratkaisujen testaamiseen. Verrokkituloksissa käytettiin seitsemää eri algoritmia, joiden perusteella laskettiin keskiarvo suorituskykyparannukselle. Esimerkiksi PyPyä käytettäessä verrokkituloksissa testiohjelma oli 164,5 % hitaampi kuin CPythonia hyödyntävä toteutus. [14, s. 39]

Eri algoritmien lisäksi työssä käytettyjen työkalujen versiot eroavat niistä, joilla vertailutulokset on saavutettu. Niissä on käytetty Pythonin versiota 2.6.1 kaikissa testeissä. Tässä työssä käytetyistä työkaluista esimerkiksi PyPystä julkaistiin Python3 yhteensopiva versio vuoden 2017 lopussa, joten vertailutuloksissa Pythonin kolmannen version hyödyntäminen ei olisi PyPyä tarkasteltaessa ollut mahdollista. [15]

Cythoniin liittyvien nopeustestien tuloksiin saattaa vaikuttaa se, kuinka kattavasti käsiteltävää ohjelmaa on muunnettu hyödyntämään C:n tai C++:n tietotyyppejä. Tässä työssä käytetyssä ohjelmassa pyrittiin niitä hyödyntämään kaikissa kohdissa, joissa se oli mahdollista.

6. YHTEENVETO

Kaikilla työkaluilla saavutettiin tässä työssä merkittävä suorituskykyparannus testiohjelmaan. Cythonilla saavutettiin käytännössä identtinen suorituskykyparannus, kuin Shed Skinillä, mutta Cythonin käyttöä varten joutui tekemään koodimuutoksia ja hyödyntämään C++:n tietorakenteita. Shed Skiniiä hyödynnettäessä ohjelman muuntamista varten tehty työ oli siis paljon vähäisempää, mutta sitä käytettäessä menetettiin Python3 tuki. PyPy vaikuttaa työn tulosten perusteella olevan hyvä vaihtoehto silloin, kun ohjelmaan ei haluta tehdä muutoksia, mutta tavoitteena on saavuttaa suorituskykyparannuksia.

Tässä työssä saatuja tuloksia on vaikeaa vertailla muiden vastaavien testien kanssa, sillä niissä on käytetty erilaisia algoritmeja ja työkalujen versioita. Se, että työssä saatuja tuloksia käytettäisiin perusteena sille, mitä vertailluista työkaluista tulisi käyttää tiettyyn tarkoitukseen vaatisi myös paljon enemmän eri testialgoritmeja, joilla työkaluja voitaisiin vertailla.

Käytännössä NumPyn tai muiden sen kaltaisten apukirjastojen käyttö voi usein olla järkevämpää kuin koko ohjelman kirjoittaminen Pythonilla, sillä tällöin saavutetaan tehokkuusparannuksia sekä voidaan hyödyntää valmiita apukirjastoja ohjelmassa lyhentäen sen kehitysaikaa. Python toimiikin hyvin skriptauskielenä, jolla hallitaan tehokkaammilla kielillä toteutettuja osia ohjelmasta.

LÄHTEET

[1] M. Lutz, Programming Python, 2nd ed. O'Reilly & Associates, Sebastopol (CA), 2001, pp.22–1232.

[2] R.W. Sebesta, Concepts of programming languages, 10th, international ed. Pearson/Education, Boston, 2012, pp. 16–35.

[3] Python Software Foundation, Installing Packages, verkkosivu 2018, Saatavissa (viitattu 19.6.2018) https://packaging.python.org/tutorials/installing-packages

[4] A. Marowka, Python accelerators for high-performance computing, The Journal of Supercomputing, Vol. 74, Iss. 4, 2018, pp. 1450.

[5] Psyco, Verkkosivu 2010, Saatavissa (viitattu 15.6.2018) http://psyco.sourceforge.net/

[6] Cython - an overview, verkkosivu 2018, Saatavissa (viitattu 15.5.2018) http://docs.cython.org/en/latest/src/quickstart/overview.html

[7] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, K. Smith, Cython: The Best of Both Worlds, Computing in Science & Engineering, Vol. 13, Iss. 2, 2011, p. 32.

[8] The PyPy Project, What is PyPy, verkkosivu 2018, Saatavissa (viitattu 16.5.2018) http://doc.pypy.org/en/latest/introduction.html

[9] The PyPy Project, Goals and Architecture Overview, verkkosivu 2018, Saatavissa (viittattu 16.5.2018) https://pypy.readthedocs.io/en/latest/architecture.html

[10] JIT compiler, in: A Dictionary of Computer Science, 7th ed., Oxford University Press, 2016.

[11] Mark Dufour & the Shed Skin contributors, Shed Skin documentation, verkkosivu 2015, Saatavissa (viitattu 21.5.2018) https://shedskin.readthedocs.io/en/latest/documentation.html

[12] Mark Dufour & the Shed Skin contributors, Shed Skin, verkkosivu 2015, Saatavissa (viitattu 21.5.2018) https://shedskin.readthedocs.io/en/latest/index.html

[13] Python Software Foundation, What's New in Python 2.7, verkkosivu 2018, Saatavissa (viittattu 21.5.2018) https://docs.python.org/dev/whatsnew/2.7.html

[14] J. Swacha, Comparative evaluation of performance-boosting tools for Python, Annales UMCS, Informatica, Vol. 11, Iss. 1, 2011, p. 39.

[15] The PyPy Project, Call for donations - PyPy to support Python3!, verkkosivu 2017, Saatavissa (viitattu 15.6.2018) https://pypy.org/py3donate.html