# SISSEJUHATUS

HAJUSARVUTUSE KASUTADES PYTHONIT

Joonas Puura

## SISUKORD

<table>
<thead>
<tr>
<th>Osavaband</th>
<th>Sõltumised</th>
<th>Lüliti on alguses</th>
<th>ülesestates</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sissejuhatus</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ligipääs Rocket klastrile</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vajaliku töökeskkonna sätestamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Failide kättesaamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI4PY paigaldamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tööde jooksutamine klastril</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bash skripti näide</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tere maailm! näide</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ülesanne 1</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teadete edastamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rebane</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ülesanne 2</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsükkel</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teadete märgistamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Märgistamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jagamine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kollektiivsed operatsioonid</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broadcast</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scatter ja Gather</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ülesanne 3</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Muud operatsioonid</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efektiivsem MPI4PY</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmide kiiruse mõõtmine</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lahendused</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lisa ja kasutatud materjal</td>
<td>……………………………………………………………………………………………………………………….</td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NB: Siin kasutame me Python 2.7.3, mis on oma süntaksi poolelt ligilähedane Python 3 versioonide süntaksile, aga näiteks print(x) asemel on print x.

Samuti võiks vigade ennetamiseks mitte kasutada täpitähti, kuid kui programmile lisada algusesse rida

```
# coding: utf-8
```

, annab see võimaluse kasutada täpitähti.

Sissejuhatuse raames kasutame Tartu Ülikooli arvutusklastrit Rocket (http://www.hpc.ut.ee/rocket_cluster).

**LIGIPÄÄS ROCKET KLASTRILE**

Ülikooli klastride kasutamiseks peame kõigepealt sellele ligi pääsema.

Iga kuu on Tartu Ülikooli tudengitel võimalik tasuta kasutada 300 *walltime* tundi arvutusmahtu. Seda aega arvutatakse kasutatud tuuma kaupa. Ehk kui kasutusel on 4 tuuma, siis korrutatakse iga kasutatud sekund neljaga. Nt kui töö, mis kasutab 1 tund aega ja 4 tuuma, siis on see 4 *walltime* tundi.

Kui kasutaja ei asu ülikooli võrgus, siis on vaja kõigepealt ühenduda ülikooli võrguga. Väljastpoolt ülikooli võrku ligipääsemiseks on vaja kasutada VPN-i.


**WINDOWS**

Windowsi puhul saame kasutada näiteks programmi nimena Putty.

Putty saab alla laadida: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (valides putty.exe)

Host Name on rocket.hpc.ut.ee ja Port 22. Connection type: SSH
Vajuta nupule „Open“.

Nüüd peaks avanema terminal, kus küsitakse sinu kasutajainformatsiooni.

Login as: kasutajanimi (kasutajanimi on sinu ülikooli kasutajanimi. Näiteks, millega logid ÕISi)
kasutajanimi@rocket.hpc.ut.ee – sinu parool.

Önnestunud autentimise korral peaksid nägema midagi sarnast:
LINUX

Linuxis on võimalik ühenduda, kasutades terminalis käsku

ssh kasutajanimi@rocket.hpc.ut.ee (kasutajanimi on sama, mis eduroami ühendudes)

Siis küsitakse sinu käest sinu kasutaja parooli

kasutajanimi@rocket.hpc.ut.ee's password:

Pärast parooli sisestamist vajuta enter ja kui kõik on korrektne, siis peaks terminaliskriptuvatama midagi sarnast:

```
Last login: Fri Aug 14 00:06:56 2015 from 172.18.20.66
Rocket cluster.
Price is 0.012 EUR per core walltime hour. 300 walltime hours per month are free for University of Tartu employees and students.
Current balance for period from 2015-08-01T00:00:00 to 2015-08-23T12:30:34
Number of jobs: 117  Duration: 4.9 hours  Cost: .06 EUR
```

VAJALIKU TÖÖKESKKONNA SÄTESTAMINE

FAILIDE KÄTTESAAMINE

Tömbame alla vajalikud failid, kasutades terminalis wget käsku.

```
wget http://kodu.ut.ee/~jopuura/parallelcomputing/materjalid.tar.gz
```

Pakime saadud failid lahti:

```
tar xvzf materjalid.tar.gz
```

Navigeerime kausta „materjalid“, kasutades käsku cd

```
cd materjalid
```

Rohkem informatsiooni terminalis kasutavate käskude kohta saab aine „Operatsioonisüsteemid“ praktikumilehelt:

[https://courses.cs.ut.ee/2014/os/fall/Main/Praktikum3Kasud](https://courses.cs.ut.ee/2014/os/fall/Main/Praktikum3Kasud)

MPI4PY PAIGALDAMINE

Olles navigeerinud alla laetud failide kausta, peame kõigepealt jooksutama ühe skripti.

Skripti jooksutamiseks kasutame käsku bash:

```
bash mpi4pyInstall.sh
```

See skript installeerib meile vajaliku MPI4PY liidese.

Navigeerime nüüd tagasi kausta „materjalid“
cd ~/materjalid

Et kontrollida, kas vajalik keskkond on korrektelt sätestatud, siis jooksuta järgnev skript, kasutades käsku sbatch, mis käivitab helloworld näite.

**sbatch kontroll.sh**

Natukese aja pärast peaks kausta ilmuma fail kontroll.out. Kausta sisu saab kuvada, kasutades käsku ls.

Vaatame nüüd, mis sisu on kontroll.out’is kasutades käsku cat

**cat kontroll.out**

Kui sisu on midagi sarnast:

```
Tere, Maailm! Protsessi nr: 0 kokk u protsesse: 4 nimega stage35.
Tere, Maailm! Protsessi nr: 1 kokku protsesse: 4 nimega stage35.
Tere, Maailm! Protsessi nr: 3 kokk u protsesse: 4 nimega stage36.
Tere, Maailm! Protsessi nr: 2 kokku protsesse: 4 nimega stage36.
```

siis võiks eeldada, et keskkond on õigesti sätestatud.

## TÖÖDE JOOKSUTAMINE KLASTRIL


Kõik arvutustööd, mida klastril tehakse, võiks käia läbi SLURMi. Selleks, et oma töid jooksutada, siis oleks tarvilik luua oma programmi jaoks bashi skript, millega vastavat tööd jooksutatakse. Lihtsuse huvides on siin materjalides olevate programmide jaoks bashi skripti ‘iptid olemas, seega selle pärast ei pea muretsema.

Selleks, et bashi skripti kasutades tööd jooksutada, on olemas käsk sbatch skriptiNimi.sh.

### BASH SKRİPTI NÄIDE

Vaatame skripti, millega jooksutasime hello world, et kontrollida kas MPI4PY töötab (kontroll.sh).

```
#!/bin/bash

# Töö nimi on HelloWorld
#SBATCH -J HelloWorld
# See töö nõuab kahte arvutussõlmme
#SBATCH -N 2
# Mitu tööülesanet sõlme kohta
#SBATCH --ntasks-per-node=2
# Väljundfail
#SBATCH --output=kontroll.out
```
# Vajalike moodulite laadimine.
module purge
module load openmpi-1.7.3
module load python-2.7.3

# Kausta nimi
export MPI4PYDIR=parallelarvutused

# Pythoni wrapperi asukoht
export PYTHONPATH=$HOME/$MPI4PYDIR/install/lib/python

# Jooksutame kasutades mpi'd.
mpirun python helloWorld/helloworld.py

Praegusel juhul huvitavad meid rohkem read:

1. #SBATCH --J HelloWorld
2. #SBATCH --N=2
3. #SBATCH --ntasks-per-node=2
4. #SBATCH --output=kontroll.out
5. mpirun python helloWorld/helloworld.py

1. Määrab ära programmi nime. Klastri peal jooksvaid programme saab näha kasutades käsku squeue
3. Määrab ära, mitu ülesannet ühele arvutussõlmele antakse
4. Väljundfaili nimi – programm ei tagasta väljundit kohe ekraanile, vaid see tagastatakse faili.
5. Programm, mida jooksutatakse. Antud juhul jooksutab mpirun programm python, mis omakorda jooksutab kaustas helloWorld paiknevad programmi helloworld.py

Vaadates nüüd uuesti failis kontroll.out olevat väljundit, näeme, et kokku on 4 tööd – 2 füüsilist sõlme x 2 tööd sõlme kohta = 4 tööd ja kahe erineva nimega sõlme: stage35 ja stage36.

NB! Kui ei ole soovi ise määratleda, et mitu tööd mingil sõlmel on, siis on võimalik kasutada lihtsalt parameetrit #SBATCH --n=4 (väärt, mis ütleb, et vaja on nelja tööd ning SLURM võtab nende jagamise enda hoole alla. Mugavuse huvides on selle kasutamine parem.

Rohkem teavet jooksutamisskriptide kohta saab http://www.hpc.ut.ee/user_guides/SLURM#runningjobs

TERE MAAILM! NÄIDE

Asume nüüd reaalse programmide juurde, et tutvustada paralleelprogrammeerimist kasutades teadete edastamise meetodit. Vaatame programmi, mida me jooksutasime ka eelnevalt, et kontrollida, kas keskkond
töötab. Programm paikneb kaustas materjalid/helloWorld nimega helloworld.py. Lühidalt küsib iga protsess, et milline arvuti teda töötleb ja tema järjekorranumbri ning kirjutab saadud informatsiooni väljundfaili. NB: Tavaliselt pole hea tava kutsuda I/O operatsioone (ehk print lauseid) teistelt protsessidelt kui 0, siin on aga see pedagoogilistel põhjustel.

```python
#!/usr/bin/env python
# coding: utf-8
# helloworld.py

# Impordime vajaliku mpi4py teegi.
from mpi4py import MPI

# Kommunikaaorite küsimine
comm = MPI.COMM_WORLD

# Küsimme muutujasse size kogu protsesside arvu.
size = comm.Get_size()

# Küsimme muutujasse nimega rank, et mitmes protsess on. Loendamine algab 0st
rank = comm.Get_rank()

# Küsimme nime.
name = MPI.Get_processor_name()

# Trükime väljundi.
print "Tere, Maailm! Protsessi nr: %d kokku protsesse: %d nimega %s.\n" % (rank, size, name)
```

Selleks, et programmi jooksutada, navegeerime esmalt kausta:

cd ~/materjalid/helloWorld

siis jooksutame käsuga sbatch helloworld.sh vastava programmi. Terminali peaks käsu tulemusena ilmuma sarnane tekst, kus number võib olla erinev:

Submitted batch job 354978

Kui töö on lõpetatud, siis ilmub kaustal fail slurm-354978.out, mille sisu on meil võimalik vaadata käsuga cat slurm-354978.out, kus peaks olema 4 rida teksti, kus protsessid arvudega 0st kuni 3ni trükivad Tere, Maailm!

Lahkame nüüd etteantud helloworld.py programmi.

Kõigepealt imporditakeme vajalik mpi4py teek – „mpi4py import MPI“, mis võimaldab meil kasutada MPI’d ehk teadete edastamise liidese võimalusi.

```python
import mpi4py
from mpi4py import MPI

# Kommunikaaorite küsimine
comm = MPI.COMM_WORLD

# Küsimme muutujasse size kogu protsesside arvu.
size = comm.Get_size()

# Küsimme muutujasse nimega rank, et mitmes protsess on. Loendamine algab 0st
rank = comm.Get_rank()

# Küsimme nime.
name = MPI.Get_processor_name()

# Trükime väljundi.
print "Tere, Maailm! Protsessi nr: %d kokku protsesse: %d nimega %s.\n" % (rank, size, name)
```

Küigepeal imporditakeme vajalik mpi4py teek – „mpi4py import MPI“, mis võimaldab meil kasutada MPI’d ehk teadete edastamise liidese võimalusi.


name = MPI.Get_processor_name() – MPI.Get_processor_name() tagastab sõlme nime.

ÜLESANNE 1

1) Muuda skripti helloworld.sh nii, et kasutatakse kolme sõlme ja igal sõlmel on 4 ülesannet.

Et terminalis tekstifaili muuta, saame kasutada programmi nimega *nano failinimi* ehk praegusel juhul *nano helloworld.sh*. Muudatuste lõpetamisel vajuta `^x` (^

Jooksuta nüüd programmi uuesti: `sbatch helloworld.sh`

2) Mitu rida on nüüd failis? Mitu protsessi jooksis kokku?
3) Miks ei pruugi väljundfailis olev väljund olla protsside suhtes kasvavas järjekorras?
4) Kirjuta programm, milles iga paarivarvine protsess teretab ja iga paarivarvine protsess ütleb head aega.

TEADETE EDASTAMINE


Kommunikaatoril (eelnevas näites `comm = MPI.COMM_WORLD`) on olemas meetodid `comm.send(data, dest=sihktokaNumber)` ja `comm.recv(asukohaNumber)`, mida saab vastavalt kasutada sõnumite saatmiseks ja vastuvõtmiseks.

**Hoiatus:** `comm.recv()` ja `comm.send()` on blokeeruvad operatsioonid, s.t., et programm jääb `recv()` juures ootama, kuni protsessilt, millelt ta send'i ootab, tuleb send().

Näiteks kui me tahame protssessilt numbriga 0 saata protsessile numbrile 1 muutujas „rebane“ sisalduvaid andmeid, siis on meil vaja järgnevad:

1) Protsess 0 puhul – `send(rebane, dest=1)`
2) Protsess 1 puhul – `data (või mõni muu nimi) = recv(source=0)`

Programm oleks seejuures selline:

**REBANE**

```
# coding: utf-8
# rebane.py

from mpi4py import MPI
```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Kui protsessi number on 0, siis saadame muutuja sisuga „Mida rebane ütleb“ protsessile 1.
if rank == 0:
    rebane = "Mida rebane ütleb?"
    print rebane, " ", rank
    comm.send(rebane, 1)
# Kui protsessi number on 1, siis ootame, kuni protsess 0 saadab meile andmeid.
elif rank == 1:
    data = comm.recv(source=0)
    print data, " ", rank

Kui selline programm käivitada kahe protsessiga, siis saame vastuseks:

    Mida rebane ütleb? 0
    Mida rebane ütleb? 1

ÜLESANNE 2

1) Muuta rebane.py programmi nii, et kui see käivitada kolme protsessiga (#SBATCH -n=3) siis saadab teine protsess numbriga 1 veel teksti edasi protsessile numbriga 2.
2) Kui kustutada etteantud programmist rida comm.send(rebane, 1), mis juhtub? (vihje: uurida käsuga squeue -u kasutajanimi. Et oma töö tühistada, saab kasutada käsku scancel töönumber nt scancel 354988).

TSÜKKEL

Toome veel ühe lihtsama näite kasutades send() ja recv(). Selles programmis saadab iga protsess järgmisele protsessile teate, tekitades tsükli. Esimene protsess saadab teisele protsessi teate, teine protsess saadab kolmandale, ..., viimane protsess saadab tagasi esimesele.

    # coding: utf-8
    # tsykkkel.py

    from mpi4py import MPI

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    Fsize = comm.Get_size()

    number = 5
if rank == 0:
    print number, rank
    comm.send(number + 1, dest=rank+1)
    number = comm.recv(source=size-1)
    print number, rank
else:
    number = comm.recv(source=rank-1)
    print number, rank
    comm.send(number + 1, dest=(rank+1)%size)

Vaheküsimus 1: Mida see programm väljundiks annab?

TEADETE MÄRGISTAMINE

Et olla kindel, et õige teate saatmisoperatsioon jõuab õige vastuvõtmisoperatsioonini (õiges järjekorras) saame me märgistada oma teated numbriga. Et seda teha, saame me send() ja recv() operatsioonil lisada parameetri tag=nr. Nt comm.send(data, dest=1, tag=1) ja comm.recv(source=1, tag=1).

MÄRGISTAMINE

# coding: utf-8
# tagging.py

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    lause1 = "Lause #1"
    lause2 = "Lause #2"
    comm.send(lause1, dest=1, tag=1)
    comm.send(lause2, dest=1, tag=2)
else:
    lause1 = comm.recv(source=0, tag=1)
    lause2 = comm.recv(source=0, tag=2)

    print lause1
    print lause2

Selline konstruktsioon tagab, et print lause1 trükib „Lause #1“ ja print lause2 trükib „Lause #2“. Ilma märgistamist kasutamata vöib lause1 saada „Lause #2“ ja vastupidi. Teadete märgistamine tuleb kasuks näiteks siis, kui igale protsessile antakse mingi osa kahest massiivist comm.send(massiiv1, dest=1, tag=1) ja comm.send(massiiv2, dest=1, tag=2). Kui nende massiivide peal kavatsetakse teha operatsioone, kus järjekord on tähtis (ehk pole
kommutatiivsed, nt \( a / b != b / a \) iga a ja b puhul), siis on tähtis, et õiged andmed satuvad õigesse kohta. Nüüd üks näide natuke keerukamast programmist.

**JAGAMINE**

```python
# coding: utf-8
# division.py

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    # Loome massiivid
    massiiv1 = [i for i in range(size*2)]
    massiiv2 = [i+1 for i in range(size*2)]
    # Arvutab osa suuruse
    osaSuurus = len(massiiv1)/size
    # Saadab töö laiali. Iga protsess saab len(massiiv)/size, ehk praegu 2 elementi.
    for i in range(1, size):
        comm.send(massiiv1[i*osaSuurus:(i*2)*osaSuurus], dest=i, tag=1)
        comm.send(massiiv2[i*osaSuurus:(i*2)*osaSuurus], dest=i, tag=2)

    # Protsess teeb oma töö.
    osa1 = massiiv1[:osaSuurus]
    osa2 = massiiv2[:osaSuurus]
    tulemus = []
    for i in range(len(osa1)):
        tulemus.append(float(osa1[i])/osa2[i])

    # Küsime teistelt töö tagasi.
    for i in range(1, size):
        tulemus = tulemus + comm.recv(source=i, tag=3)

    print tulemus
else:
    # Tagidega kindlustame, et osa1 on massiiv1'st ja osa2 on massiiv2'st
    osa1 = comm.recv(source=0, tag=1)
    osa2 = comm.recv(source=0, tag=2)
    tulemus = []
    for i in range(len(osa1)):
        tulemus.append(float(osa1[i])/osa2[i])
```
Üleval olev programm loob kaks massiivi, mis omavahel jagatakse. Ilma märgistamata võivad tekkida probleemid, nagu näiteks nulliga jagamine ja valed lahendused.

Vaheküsimus 2: Miks jookseb sama programm kiiremini ühe protsessi kui mitme protsessi peal?
Sellisel juhul jääb protsess 0 sinna barjääri taha kinni. Ainuke võimalus välja saamiseks on siis, kui teised protsessid ka sinnamaani jõuavad, mis antud olukorras on võimalu.

**BROADCAST**

Kui meil on vaja anda andmed kõikidele protsessidele samasugused andmed. Üks võimalus on anda esimeselt protsessilt ükskaival kõigile teistele need andmed. Teine võimalus on kasutada sisseehitatud meetodeid.

MPI4PY kommunikaatoril on oluline meetod bcast(data, source=0), mis võtab endale kaks argumenti. Esimene argument data – andmed, mildest iga protsess saab identse koopia, ja source, et milliselt protsessilt need andmed pärsinud. Broadcast kasutab puul põhinevat algoritmi, mis on tundvat kiirem kui algortim, mis jagab ühelt paljudele.

**ILMA BROADCASTITA**

```python
#!/usr/bin/env python
# coding: utf-8
# nobroadcast.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = 1
    for i in range(1, size):
        comm.send(data, dest=i, tag=1)
else:
    data = comm.recv(source=0, tag=1)

comm.Barrier()
print data, "protsessilt", rank
```

**BROADCASTIGA**

```python
#!/usr/bin/env python
# coding: utf-8
# broadcast.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Kõigil protsessidel väljaarvatud järjekorranumbriga 0 andmed puuduvad.
```
data = None
if rank == 0:
    data = [1,2,3,4,5,6,7,8,9,10]
# Broadcastiga
data = comm.bcast(data, root=0)
print data

Selle programmi töö tulemusena antakse igale protsessile andmed, mis on protsessil järjekorranumbriga 0. data = comm.bcast(data, root=0). Iga protsess print`ib töö tulemusena [1,2,3,4,5,6,7,8,9,10].

SCATTER JA GATHER

Scatter ja gather on põhimõtteliselt vastandlikud operatsioonid, mis on mõeldud ühelt protsessilt teistele protsessidele andmete jaotamiseks ja tagasi kokku kogumiseks.

SCATTER
Scatter jaotab andmete kõikide kommunikaatori küljes olevatele protsessidele laiali. Näiteks, kui meil on andmed massiivis [1,2,3,4,5] ja meil on 5 protsessi, siis protsessid saavad endale järjest 1, 2, 3, 4 ja 5.

Scatter tahab, et andmemassiivis oleks täpselt sama palju elemente, kui on protsessse, millele need jaotatakse. Näiteks, kui meil on rohkem andmeid kui protsessse, siis tuleks andmed jagada tükideks, ning siis need tükidena edasi anda. Kui meil on 2 protsessi ja meil on massiiv [1,2,3,4,5] siis enne scatter`i tegemist tuleks nad viia kujule [[1,2,3], [4,5]] ja siis protsess järjekorranumbriga 0 omandab [1,2,3] ja protsess järjekorranumbriga 1 omandab endale [4,5]

GATHER
Gather kogub kõikidelt protsessideelt andmed kokku. Näiteks kui protsessidel on järgnevalt 5, 4, 3, 2, 1 siis protsess, millele kutsutakse gather välja, omandab endale andmed [5,4,3,2,1].

Vaatame eelnevalt kirjeldatud operatsioonide kohta näidet:

SCATTER JA GATHER NÄIDE

#!/usr/bin/env python
# coding: utf-8
# scattergather.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Kõigil protsessidel väljaarvatud järjekorranumbriga 0 andmed puuduvad.
data = None
# Loome protsessil järjekorranumbriga 0 massiivi, mis on pikkusega size (ehk protsesside arv)
if rank == 0:
    data = [i*2 for i in range(size)]
    print "algsed andmed", data
    data = comm.scatter(data, root=0)
    # Lahutame kõigil protsessidel saadud andmetest maha 1
    data = data - 1
    # Saadame töödeldud andmed tagasi. Protsess 0 saab endale töödeldud andmed.
    data = comm.gather(data, root=0)

if rank == 0:
    print "lõplikud andmed", data

Programm tekitab protsessil järjekorranumbriga 0 massiivi, mis on protsesside arvuga sama pikk. Siis jaotab programm andmed kõigile protsessidele laiali ja iga protsess lahutab andmetest 1. Lõpptulemusena saadakse väljund:

    algsed andmed [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
    lõplikud andmed [-1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

ÜLESANNE 3

1) Muuta Scatter ja Gather programmii nii, et massiivi pikkus on kaks korda suurem kui protsesside arv. Vihje: kuna scatter operatsioon peab saama andmetemassiivina argumendiks sama palju andmeid, kui on protsesse, siis tuleb andmed jaotada paaridesse.
2) Miks on kollektiivsete operatsioonide kasutamine kasulikum kui nee d operatsioonid ise teha?
3) Uurida broadcasti kohta, et miks on broadcast kiirem kui esimeselt protsessilt ükskaaval teistele saatmine.
4) Millistes uurimisvaldkondades kasutatakse hajusarvutusi?

MUUD OPERATSIOONID

Reduce on kollektiivne operatsioon, mis kogub kõigilt protsessidelt, mis on kommunikaatoriga ühendatud, andmed kokku ja teeb nendega vastavat operatsiooni.

Näiteks kui igal protsessil on üks täisarv, siis kasutades näiteks reduce() operatsiooni on võimalik kõik need arvud üheks kokku liita.

Näiteks on 3 protsessi. Neil on vastaval arvud 1, 2 ja 3. Siis kasutades reduce operatsiooni ja tahtes liita, liidab reduce operatsioon need 1+2+3 = 6 ja tagastab sellele ühele ühele protsessile.

Allreduce on selline kollektiivne operatsioon, mis kogub kõikidel protsessidel kokku, teeb nendega mingi tehte ja siis saadab tulemuse kõikidele protsessidele tagasi.

Ehk siis ülevalpool arvutatud 6 saadetakse kõikidele protsessidele.


EFKTIIVSEMPY
Üleval pool kasutatud kollektiivsed operatsioonid töötavad kõigi Pythoni objektidega, aga need ei ole jõudluse kohalt kõige efektiivsemad. Selleks, et kirjutada efektiivset Pythoni MPI programmi, oleks tarvilik kasutada näiteks NumPy objekte. NumPy on Pythoni laiendus, mis lubab efektiivselt teha erinevaid tehteid massiivide, maatriksitega jne.

Selle kohta saab samuti rohkem lugeda [http://mpi4py.scipy.org/docs/usrman/tutorial.html](http://mpi4py.scipy.org/docs/usrman/tutorial.html)

**PROGRAMMIDE KIIRUSE MÕÕTMINE**

MPI4PY programmide efektiivsust on võimalik mõõta näiteks käsitsi, lisades koodiosadesse erinevaid taimereid. Üks meetod, mis lubab seda teha, on WTime()

```
wt = MPI.Wtime()
```

ProgrammiOsa.

```
wt = MPI.Wtime() - wt
```

if rank == 0:

    print wt

**LAHENDUSED**

1.1. helloworld.sh muuta #SBATCH -N 2 -> #SBATCH -N 3 ja #SBATCH --ntasks-per-node=2 -> #SBATCH --ntasks-per-node=4

1.2. 12 rida Tere, Maailm! .. jne

1.3. Kuna mõni protsess võib lõppeda enne teist protsessi (näiteks võib mõni arvuti olla kiirem kui mõni teine), siis ei pruugi olla faili trükitud tekstid arvu suhtes kasvavas järjekorras.

1.4. 

```
#!/usr/bin/env python
# coding: utf-8
# helloworld.py

# Importime vajaliku mpi4py teegi.
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank % 2 == 0:
    print "Tere, Maailm! Protsessi nr: %d \n" % (rank)
else:
    print "Head aega, Maailm! Protsessi nr: %d \n" % (rank)
```

2.1. Saame programmi alguses küsida size = comm.Get_size() ja sellele vastavalt kasutada if lauseid. Üks lahendus on näiteks selline:

```
#!/usr/bin/env python
# coding: utf-8
# rebane.py
```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = "Mida rebane ütleb?"
    print data, rank
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source = 0)
    print data, rank
    if size > 2:
        comm.send(data, dest = 2)
elif size > 2 and rank == 2:
    data = comm.recv(source = 1)
    print data, rank

2.2. Program jääb rea comm.recv(source=0) peal lõputult ootama.

3.1.

#!/usr/bin/env python

# coding: utf-8
# scatterjagather2.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Kõigil protsessidel väljaarvatud järjekorranumbriga 0 andmed puuduvad.
data = None
# Looome protsessil järjekorranumbriga 0 massiivi, mis on pikkusega size (ehk protsesside arv)
if rank == 0:
    algAndmed = [i*2 for i in range(size*2)]
data = []
    for i in range(0, len(algAndmed), 2):
        data.append([algAndmed[i]]+[algAndmed[i+1]])
    print "algsead andmed", algAndmed
data = comm.scatter(data, root=0)
# Lahutame kõigil protsessidel saadud andmetest maha 1
for i in range(len(data)):
    data[i] = data[i] - 1
# Saadame töödeldud andmed tagasi. Protsess 0 saab endale töödeldud andmed.
data = comm.gather(data, root=0)

if rank == 0:
    tulemus = [y for x in data for y in x]
    print "lõplikud andmed", tulemus

3.2. Kuna kollektiivsed operatsioonid kasutavad endas keerulisemaid algoritme, mis on keerukuselt efektiivsemad.

3.3. Kasutatakse puualgoritmi, mis on asümptootiliselt parema efektiivsusega algoritm.

Vaheküsimus 1: Järjest arvud alates 5, 6 kuni protsesside arvuni. Iga numbri kõrval protsessi järjekorranumber.

Vaheküsimus 2: Kuna teistele protsessidele andmete andmine teadete teel tekitab ajakulu – seetõttu on

**LISA JA KASUTATUD MATERJAL**

Hajusarvutus: https://et.wikipedia.org/wiki/Hajusarvutus

OpenMPI koduleht: [http://www.open-mpi.org/](http://www.open-mpi.org/)

MPI4PY dokumentatsioon: [http://mpi4py.scipy.org/](http://mpi4py.scipy.org/)


Python 2 ja Python 3 süntaksi erinevused: [https://docs.python.org/release/3.0.1/whatsnew/3.0.html](https://docs.python.org/release/3.0.1/whatsnew/3.0.html)


Materjalina on kasutatud Indrek Jentson’i tööd [https://courses.cs.ut.ee/2015/tatar/spring/Main/Project](https://courses.cs.ut.ee/2015/tatar/spring/Main/Project) nimega Paralleelarvutuste programmeerimine keeles Python. Sealt on võimalik ka lugeda estnlk abil tekstitöötlmise kohta

High Performance Center: [http://www.hpc.ut.ee/](http://www.hpc.ut.ee/)

Kasulikke MPI4PY näiteprogramme leiab aadressilt: [https://bitbucket.org/dalcinl/mpi4py-tutorial/src/b55f546244e1?at=default](https://bitbucket.org/dalcinl/mpi4py-tutorial/src/b55f546244e1?at=default)