Meccanismo Complesso
-www.meccanismocomplesso.org

Cluster e programmazione in parallelo con MPI e Raspberry Pi

2015-05-01 18:05:51 Fabio Nelli

Introduzione

Sei uno sviluppatore e hai sempre desiderato poter implementare del codice in parallelo e provarlo su un cluster di processori? Bene, adesso è possibile a basso costo e comodamente a casa tua. Grazie al bassissimo costo delle schede Raspberry Pi è possibile realizzare un piccolo cluster casalingo su cui fare pratica e prendere dimestichezza con la programmazione in parallelo.

In questo articolo, vedrai come è facile realizzare un piccolo cluster composto da due Raspberry Pi. Come linguaggio di programmazione utilizzeremo Python e per quanto riguarda la programmazione in parallelo, vedremo come sia facile poter implementare del codice parallelo grazie al modulo MPI. Una serie di esempi ti introdurranno poi ai concetti base che ti saranno utili per sviluppare qualsiasi tuo progetto.

La programmazione in parallelo

Finora, molto probabilmente, hai sempre visto il codice come una serie di comandi che vengono eseguiti uno dopo l'altro, in modo seriale, e per seriale non si intende dall'inizio alla fine del codice, bensì un comando alla volta consecutivamente. Quindi quando implementi del codice per risolvere un problema o in generale per effettuare un'operazione specifica, ragionerai in tale modo creando alla fine quello che generalmente viene chiamato un **algoritmo seriale**.

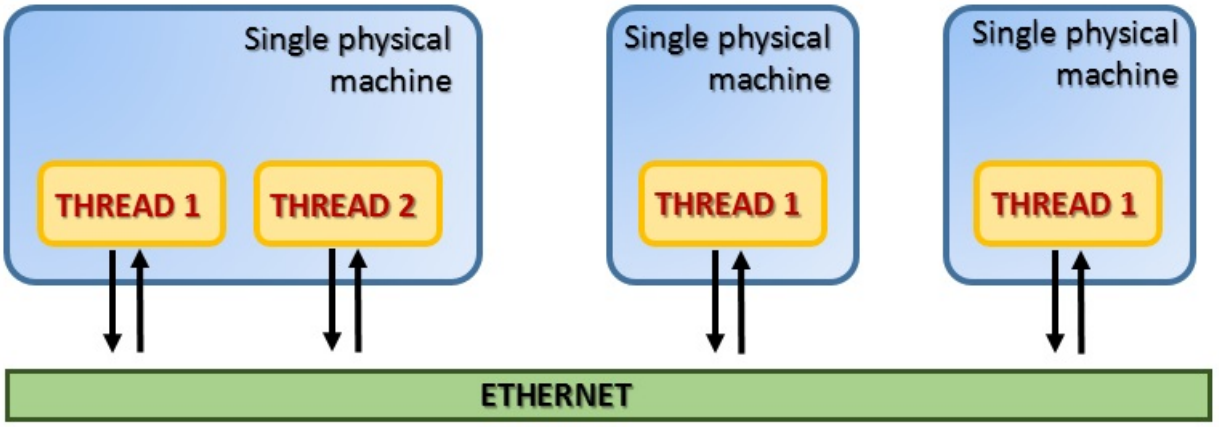
In realtà questo non è certo l'unico modo di ideare degli algoritmi. Quando si ha a che fare con dei sistemi in parallelo allora dobbiamo cominciare a ragionare diversamente, e qui appunto che viene la **programmazione in parallelo**. In realtà, quando si ha a che fare con il calcolo parallelo bisogna cominciare a ragionare in maniera diversa. In effetti, quando sviluppo del codice, dovrai cominciare a pensare se qualche sua porzione è più efficiente se viene calcolato in maniera parallela, cioè da più sistemi contemporaneamente, o se invece conviene calcolarla serialmente. Alla fine quello che realizzerai sarà un **algoritmo parallelo**.

Quindi durante la stesura del codice, sarà tuo compito pensare a quali comandi possono essere distribuiti su più sistemi, in che modo e come devono essere calcolati. Potresti ragionare in modo che più sistemi si suddividano il problema in porzioni più semplici e alla fine ogni loro risultato possa essere riunito (**in sinergia**), oppure che un sistema assegni un particolare compito ad un altro (**master-slave**) o persino che essi entrino in qualche modo in concorrenza tra di loro (**concorrente**).

In effetti, questo modo di pensare può risultare abbastanza complesso e a volte alcune soluzioni possono portare ad effetti imprevisti. Generalmente, se sviluppato correttamente, un algoritmo parallelo si dimostra molto più efficiente di uno seriale, in modo particolare quando si devono effettuare calcoli complessi o estesi su una base di dati enorme.

Il Message Passing

Fra i diversi modelli computazionali paralleli, il **Message Passing** si è dimostrato uno dei più efficienti. Questo paradigma è specialmente adatto per architetture a memoria distribuita. In questo modello i thread si trovano su una o più macchine e comunicano tra di loro attraverso Ethernet tramite messaggi.



Questo modello si basa su un protocollo specializzato chiamato appunto **MPI (Message Passing Interface)**. Consiste principalmente in un sistema di message-passing portabile e standardizzato e attualmente è diventato lo standard de facto per la comunicazione in parallelo all'interno di clusters. Dal suo primo rilascio la libreria standard MPI è diventata la più diffusa.

Questo protocollo standard ha raggiunto la versione 3.0 (2012) (vedi qui) e nel tempo, è stato implementato da diverse librerie. La libreria più utilizzata è **MPI (OpenMPI)** che è proprio quella che utilizzeremo nei nostri esempi.

Per il nostro clustering utilizzeremo proprio il modello **Message Passing** utilizzando il protocollo MPI e la libreria OpenMPI che in Python è concretizzata nel modulo **MPI for Python**.

mpi4py – MPI for Python

Il modulo **mpi4py** fornisce un approccio object oriented al message passing da integrare nella programmazione Python. Questa interfaccia è stata progettata seguendo la sintassi e la semantica MPI definita dallo standard **MPI-2 C++ bindings**. Un'ottima documentazione al riguardo la puoi trovare in questa pagina scritta proprio dall'autore di questo modulo **Lisando Dalcin**. Gli esempi di questo articolo sono infatti tratti dalla sua documentazione.

Installazione

Per installare questo modulo su Linux

```
$ sudo apt-get install
python-mpi4py
```

```
1 $ sudo apt-get install python-mpi4py
```

Oppure se preferisci usare **PyPA** per installare il package, immetti il seguente comando.

```
$ [sudo] pip install
mpi4py
```

```
1 $ [sudo] pip install mpi4py
```

Comunicare oggetti Python con i Communicator

L'interfaccia **MPI for Python** implementa il modello Message Passing attraverso il protocollo MPI. Quindi la caratteristica fondamentale di questa interfaccia è quella di poter comunicare dati e comandi (oggetti Python) tra i vari sistemi. Alla base di questo funzionamento vengono sfruttate le funzionalità del modulo **pickle**. Questo modulo permette di costruire rappresentazioni binarie (**pickling**) di qualsiasi oggetto Python (sia built-in che definito dall'utente). Tali rappresentazioni binarie potranno essere così inviate tra i vari sistemi scambiando così i dati e le istruzioni necessarie per il calcolo parallelo. Le rappresentazioni binarie una volta raggiunto il sistema vengono ripristinate nella loro forma originale, cioè l'oggetto Python di partenza (**unpickling**).

Questa scelta inoltre permette la comunicazione tra sistemi eterogenei, cioè cluster composti da sistemi aventi architetture differenti. Infatti gli oggetti Python inviati sotto forma di rappresentazioni binarie possono essere ricostruiti dal sistema ricevente in concordanza con la sua particolare architettura.

Nella libreria **MPI**, i vari thread vengono organizzati in gruppi chiamati **communicators**. Tutti i thread all'interno di un determinato communicator possono parlare tra di loro, ma non con thread esterni ad esso.

L'oggetto del modulo MPI for Python che svolge tutte le funzionalità di comunicazione in modo implicito è l'oggetto **Comm**, che sta per **communicator**. Esistono due istanze predefinite di questa classe:

- COMM_SELF
- COMM_WORLD

Attraverso di essi potrai creare tutti i nuovi communicator di cui avrai bisogno. Infatti, all'attivazione del protocollo MPI, un communicator di default viene creato contenente tutti i thread disponibili. Per far partire una MPI è sufficiente importarla

```
from mpi4py import MPI
```

```
1 from mpi4py import MPI
```

Per poter inviare i messaggi devi poter indirizzare i thread all'interno del Communicator. Questi thread sono riconoscibili tra di loro attraverso due numeri associati ad essi:

- il **numero di processo**, ottenibile con **Get_size()**
- il **rank**, ottenibile con **Get_rank()**

Quindi una volta creato un oggetto Communicator, il passo successivo è quello di sapere quanti thread esistono e quale sia il loro rank.

```
comm =
MPI.COMM_WORLD
```

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

Tipologie di comunicazione

Ogni comunicazione ha con sé dei **tag**, questi forniscono uno strumento di riconoscimento da parte del ricevente in modo che esso possa capire se considerarlo o meno, e in che modalità agire.

Prima di cominciare a sviluppare il codice, analizzando l'algoritmo parallelo che ci permetterà di svolgere il compito da noi prefissato è necessario pensare a quale tipologia di comunicazione mettere in atto. Infatti, esistono due tipologie di comunicazione:

- Blocking
- Non-blocking

Queste due tipologie di comunicazione si differenziano a seconda di come vogliamo che i due sistemi che comunicano si comportino.

Le comunicazioni Blocking

Generalmente le comunicazioni che intercorrono tra i vari sistemi di un cluster sono di tipo **blocking**. Queste funzioni infatti bloccano il chiamante finché i dati coinvolti nella comunicazione in parallelo non vengono restituiti dal ricevente. Il programma quindi non proseguirà finché non si riceveranno i risultati necessari per proseguire. I metodi **send()**, **recv()** e **sendrecv()** sono i metodi che permettono di comunicare oggetti Python generici.Mentre i metodi **Send()** e **Recv()** servono a permettere la trasmissione di grandi array di dati come per esempio i NumPy array.

Le comunicazioni Nonblocking

Alcune tipologie di calcolo possono richiedere un incremento di performance e questo si può ottenere facendo in modo che i sistemi comunichino in modo autonomo, cioè inviano i dati e ripartono con il calcolo senza attendere alcuna risposta da parte del ricevente. MPI fornisce per la comunicazione anche funzioni nonblocking. Ci sarà quindi una funzione di chiamata-ricezione che inizia l'operazione richiesta e una funzione di controllo che permetterà di scoprire se tale richiesta è stata completata. Quindi come funzioni di chiamate-ricezione abbiamo **Isend()**, **Irecv()**. Questi metodi restituiscono un oggetto **Request**, che identifica in modo univoco l'operazione richiesta. Una volta che tale operazione è stata inviata si può controllare il suo andamento attraverso i metodi di controllo **Test()**, **Wait()** e **Cancel()** della classe Request.

Le Comunicazioni Persistenti

Esiste anche un'altra tipologia di comunicazione, anche se in realtà rientra tra le comunicazioni di tipo non-blocking. Spesso durante la stesura del codice ti potresti accorgere che qualche richiesta di comunicazione venga eseguita ripetutamente poiché si trova all'interno di un loop. In questi particolari casi, si può ottimizzare la performance del codice utilizzando le comunicazioni persistenti. Le funzioni del modulo MPI for Python che svolgono questa mansione sono **Send_init()** e **Recv_init()**, appartengono alla classe Comm e creano una richiesta persistente restituendo un oggetto di tipo **Prequest**.

Adesso vediamo come codificare una comunicazione blocking. Abbiamo visto che per inviare dei dati possiamo utilizzare la funzione **send()**.

```
data = [1.0, 2.0,
3.0, 4.0]
```

```
data = [1.0, 2.0, 3.0, 4.0]
comm.send(data, dest=1, tag=0)
```

con questo comando i valori contenuti all'interno di data vengono inviati al thread di destinazione con *rank=1* (definito dall'opzione *dest* passata come secondo argomento). Il terzo parametro è una **tag** che può essere utilizzata nel codice per etichettare diverse tipologie di messaggio. In questo modo il thread ricevente può comportarsi in maniera diversa a seconda del tipo di messaggio.

Per quanto riguarda il sistema ricevente scriveremo la funzione **recv()**.

```
data =
comm.recv(source =
```

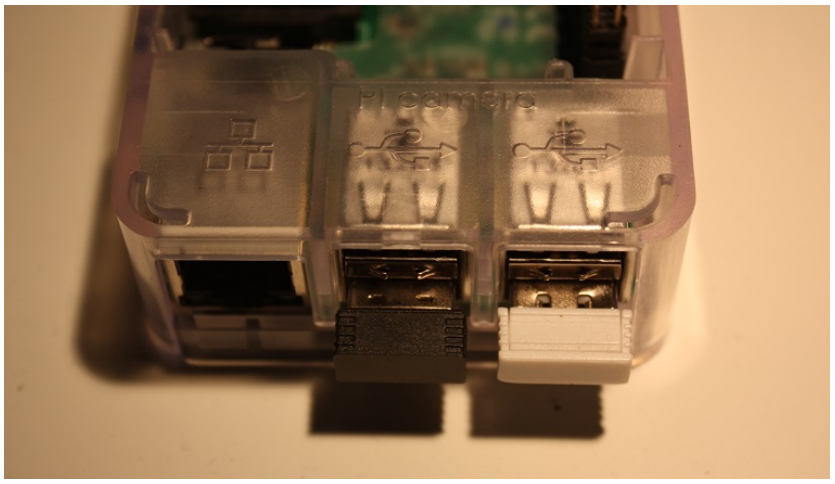
```
data = comm.recv(source = 0, tag = 0)
```

Realizziamo il cluster di Raspberry

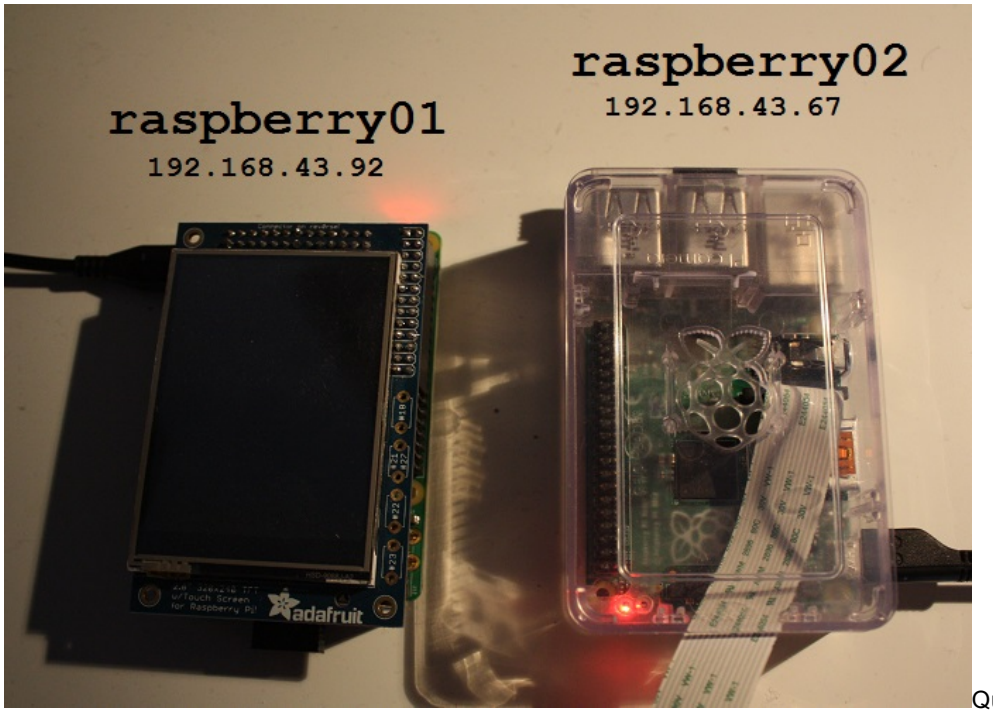
In questo articolo mi illustrerò il mio semplice cluster composto da due Raspberry Pi B+, comunque i principi e i comandi che utilizzerò saranno gli stessi per qualsiasi sistema cluster voi implementiate, per quanto esteso sia.

Al momento attuale è consigliabile creare un cluster con coppie di Raspberry Pi 3 Quad Core CPU 1.2 GHz , si avrà così una maggiore potenza di calcolo, al medesimo prezzo!

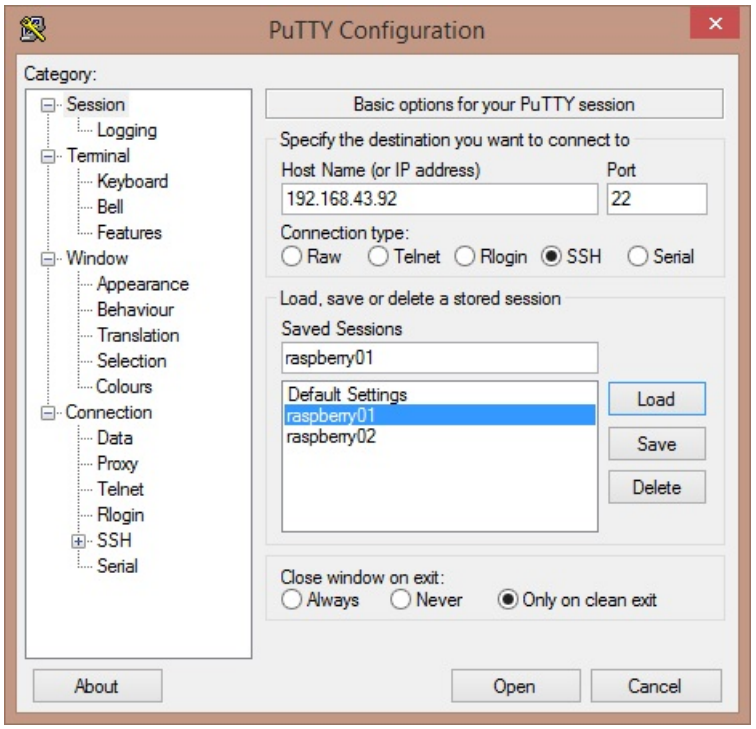
Per prima cosa ho aggiunto una Wireless USB a ciascuno dei due Raspberry Pi,



e li ho aggiunti alla mia rete casalinga.



Quindi dopo aver acceso i due raspberry con il mio portatile mi sono collegato tramite SSH ai due sistemi. Se siete su Windows potete utilizzare l'applicazione **Putty** (scaricabile da qui).



Se invece disponete di una versione di Linux, sarà sufficiente inserire questa riga di comando (se il vostro utente è pi):

```
ssh pi@192,168.43.92
```

e poi inserire la password.

Se per caso non dovesse funzionare, è molto probabile che non avete abilitato l'SSH sui due Raspbìan. Per abilitarlo scrivi

```
sudo raspi-config
```

e poi puoi abilitare SSH selezionando la voce corrispondente nel menu di configurazione

A questo punto vi voglio ricordare che per proseguire con gli esempi dovete aver già installato MPI for Python su entrambe i Raspbian dei due Raspberry.

Adesso dovresti avere tutto per poter cominciare. Ogni codice Python seguente deve essere copiato su entrambe le macchine e poi eseguito su una delle due lanciando il comando **mpiexec**

```
mpiexec -n 2 --host raspberry01,raspberr
```

Nel mio caso sto usando un cluster composto da due sistemi, quindi ho specificato 2 come argomento dell'opzione **n**. Poi si inseriscono gli IP o gli hostname delle macchine componenti il cluster, il primo nome deve essere quello della macchina su cui si sta lanciando il comando mpiexec. Alla fine si inserisce il comando da lanciare contemporaneamente su tutti i sistemi. Il file contenente il codice Python deve essere presente su tutte le macchine e deve trovarsi nello stesso path di directory.

Nel mio esempio ho creato una directory MPI che conterrà tutti i file Python.

Primi esempi di trasferimento dati

Il primo esempio sarà quello di trasferire un oggetto Python built-in come as esempio un ***dictionary*** dal sistema 0 al sistema 1. Questa è l'attività più basilare che si ha nel calcolo parallelo ed è un semplice esempio di comunicazione Point-to-Point. Scrivi il codice seguente e salvalo come **first.py.**

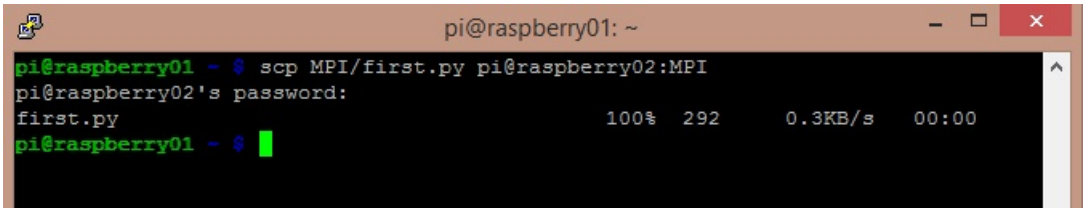
```
from mpi4py import
MPI
```

```
from mpi4py import MPI

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

if rank == 0:
data = {'a': 7, 'b': 3.14}
comm.send(data, dest=1, tag=11)
print('from rank 0: ')
```

```
print(data)
else:
data = comm.recv(source=0, tag=11)
print('from '+str(rank)+': ')
print(data)
```

Io personalmente edito questo file su uno dei due sistemi, per esempio *raspberry01* con il comando **nano** e poi lo copio sul sistema *raspberry02* attraverso SSH con il comando **scp**.

```
scp MPI/first.py pi@raspberry02:MPI
```



Se avete creato il file in un path diverso che dalla directory MPI, modificate la riga descritta sopra. E adesso eseguiamo su raspberry 01, il file appena creato.

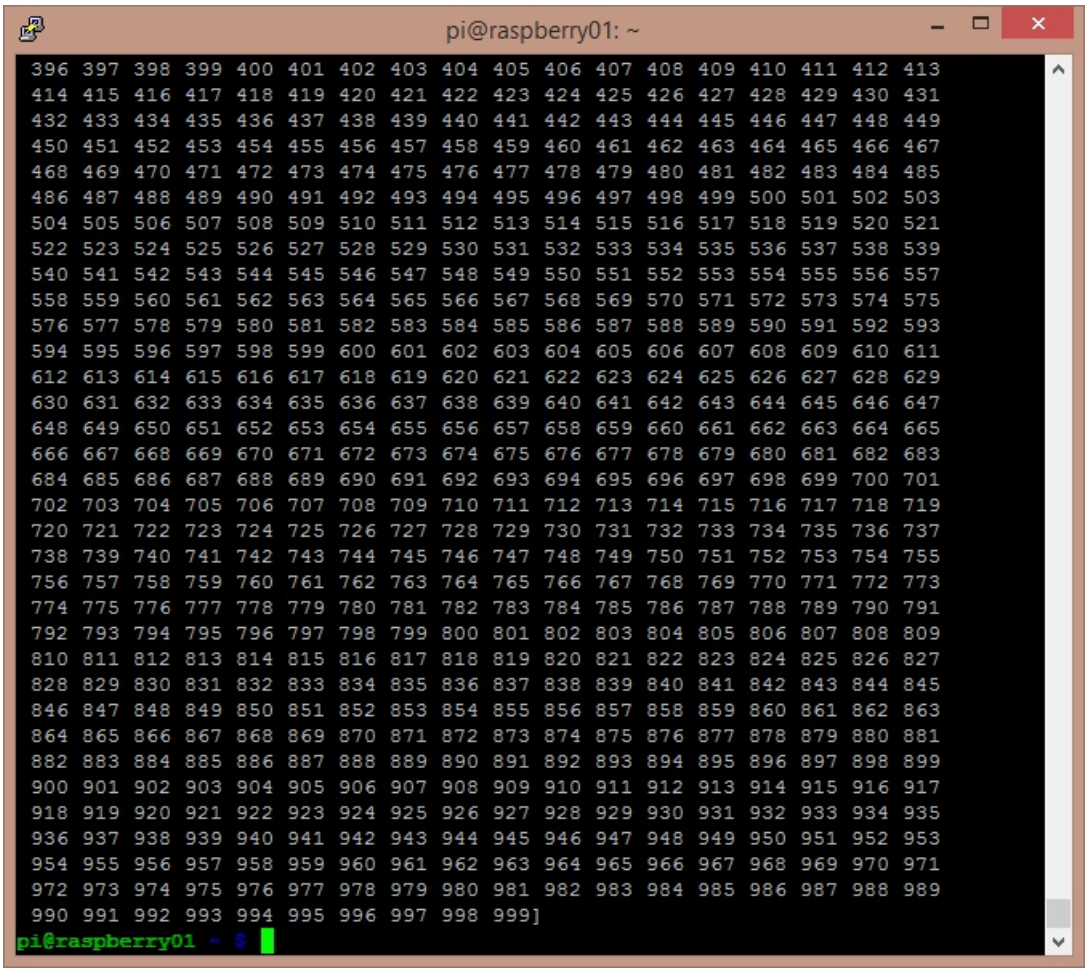
Come possiamo vedere dalla figura sopra, l'oggetto dictionary è stato trasferito correttamente al secondo sistema raspberry02. L'oggetto inviato è stato convertito in forma binaria dal sistema raspberry01 ed è stato ricostruito dal sistema ricevente raspberry02.

In un secondo esempio, scambieremo tra i due sistemi un array NumPy.In questo caso a differenza del caso precedente utilizzeremo le funzioni **Send()** e **Recv()** (riconoscibili dal fatto che la prima lettere è in maiuscolo). Queste funzioni utilizzano un sistema di buffering molto efficiente che rende la trasmissione paragonabile al sistema MPI sviluppato in C. Copia il codice seguente su uno dei sistemi del cluster e poi copialo su tutti gli altri. Salvalo come **numpy_array.py**.

```
from mpi4py import
MPI
```

```
from mpi4py import MPI
import numpy

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

if rank == 0:
data = numpy.arange(1000, dtype='i')
comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
data = numpy.empty(1000, dtype='i')
comm.Recv([data, MPI.INT], source=0, tag=77)
print('from '+str(rank))
print(data)
```

Il codice fa inviare dal sistema raspberry01 al sistema raspberry02 un array NumPy contenente 1000 numeri interi in sequenza crescente.



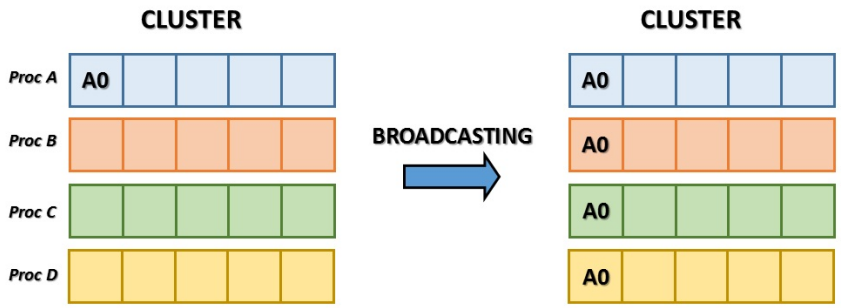
Altri esempi sulla comunicazione collettiva

Negli esempi precedenti abbiamo visto che il rank 0 viene assegnato al **master node**, che è quel sistema su cui viene lanciato il comando mpiexec, raspberry01 per internderci. Il master code inviava messaggi specifici ad un preciso sistema specificando il suo rank nella chiamata.

Adesso vedremo tre esempi che coprono tre aspetti diversi della comunicazione collettiva che può avvenire tra più sistemi contemporaneamente. Purtroppo il mio cluster è composto da soli due elementi, mentre in realtà gli esempi seguenti sarebbero più consoni se eseguiti in un cluster maggiore di 4 elementi.

Broadcasting

In questo esempio eseguiremo invece un broadcast, cioè il master node invierà lo stesso messaggio di dati contemporaneamente a tutti gli altri sistemi.



Per prima cosa tutti i dati sugli altri nodi devono essere impostati come **None**. Poi il master node invia i dati a tutti gli altri tramite la funzione **bcast()**. Il codice seguente esegue tutte queste operazioni. Salvalo come ***broadcasting.py*** e poi copialo su tutti gli altri sistemi.

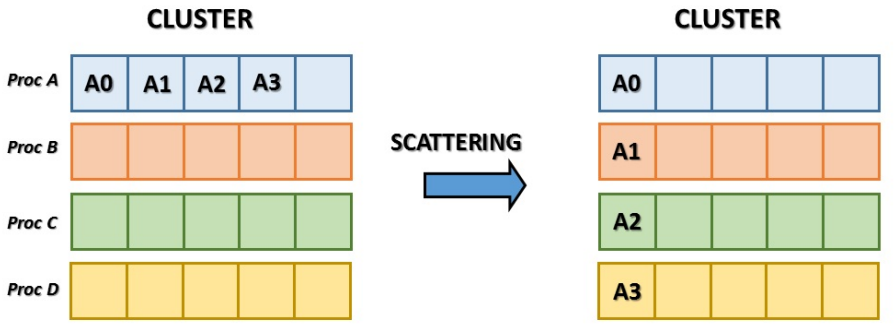
```
from mpi4py import
MPI
1 from mpi4py import MPI
2
3 comm = MPI.COMM_WORLD
4 rank = comm.Get_rank()
5
```

```
if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ('abc','xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
print('from' + str(rank))
print(data)
```

```
pi@raspberry01: ~
pi@raspberry01 ~ $ mpiexec -n 2 --host raspberry01,raspberry02 python MPI/broadc
asting.py
pi@raspberry02's password:
from 0
{'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
from 1
{'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
pi@raspberry01 ~ $
```

Scattering

Lo scattering è invece la modalità con cui il master node può suddividere una struttura di dati come una lista e poi distribuirla suddividendola in diverse porzioni ognuna trasmessa con un relativo messaggio ad uno dei sistemi che compone il cluster.



Questa operazione viene svolta dalla funzione **scatter()**. Anche in questo caso tutti gli altri sistemi prima di procedere allo scattering devono avere i dati impostati su None. Scrivi il codice seguente e salvalo come ***scattering.py***, poi copialo su tutti gli altri sistemi.

```
from mpi4py import
MPI
```

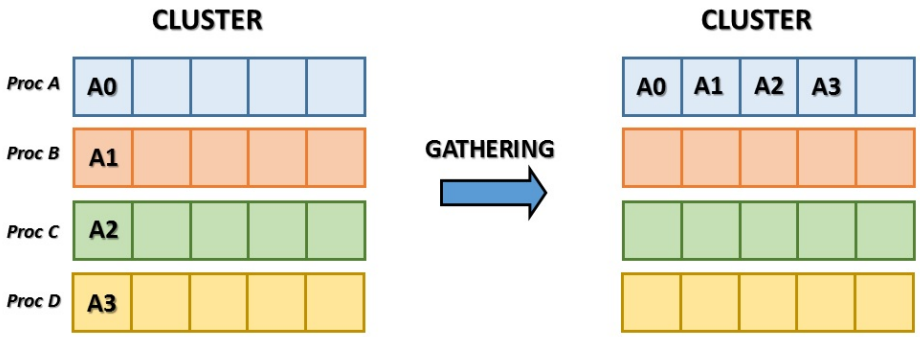
```
from mpi4py import MPI

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

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

```
pi@raspberry01: ~
pi@raspberry01 ~ $ mpiexec -n 2 --host raspberry01,raspberry02 python MPI/scatte
ring.py
pi@raspberry02's password:
from 0
1
from 1
4
pi@raspberry01 ~ $
```

Questo esempio fa calcolare ad ogni sistema il suo numero di rank aggiunto di una unita ed elevato al quadrato.

Gathering

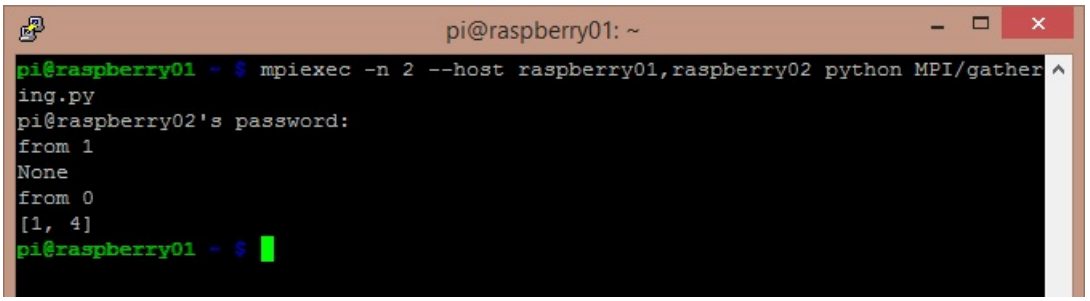
Il gathering è praticamente l'operazione opposta rispetto allo scattering. In questo caso è il master node ad essere inizializzato e a ricevere tutti gli elementi di dati dagli altri nodi componenti il cluster.



Inserisci il codice seguente e salvalo come **gathering.py**, poi copialo su tutti gli altri sistemi.

```
from mpi4py import
MPI
```

```
from mpi4py import MPI

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

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
   for i in range(size):
      assert data[i] == (i+1)**2
else:
   assert data is None
```



In questo esempio possiamo vedere come il master node riceve tutti i risultati di calcolo degli altri nodi componenti il cluster. I risultati sono gli stessi di quelli dell'esempio precedente, ottenuti calcolando il rank aggiunto di un valore e poi elevato al quadrato.

Esempio di Calcolo in parallelo: calcolo del π

Per concludere l'articolo prenderemo sotto esame il calcolo del π. Ho trovato in internet un bellissimo esempio per questo tipo di calcolo sviluppato da ***jcchurch***, e il cui codice è disponibile su GitHub.

Ho apportato qualche piccola modifica per misurare alcune caratteristiche come il tempo impiegato e l'errore sul calcolo del valore di π. Per comodità riporto il codice in questa pagina:

```
<span class="pl-
k">from</span>
```

```
<span class="pl-k">from</span> mpi4py <span class="pl-k">import</span> MPI
import time
import math

comm <span class="pl-k">=</span> MPI.COMM_WORLD
rank <span class="pl-k">=</span> comm.Get_rank()
<span style="font-family: Consolas, Monaco, 'Lucida Console', monospace; font-size: 12px;
font-style: normal;">size </span><span class="pl-k" style="font-family: Consolas, Monaco,
'Lucida Console', monospace; font-size: 12px; font-style: normal;">=</span><span style="font-
family: Consolas, Monaco, 'Lucida Console', monospace; font-size: 12px; font-style: normal;">
comm.Get_size()
</span><span style="font-family: Consolas, Monaco, 'Lucida Console', monospace; font-size:
12px; font-style: normal;">
# I will change these parameters for the performance table
slice_size </span><span class="pl-k" style="font-family: Consolas, Monaco, 'Lucida Console',
monospace; font-size: 12px; font-style: normal;">=</span> <span class="pl-c1" style="font-
family: Consolas, Monaco, 'Lucida Console', monospace; font-size: 12px; font-style:
normal;">1000000
</span>total_slices <span class="pl-k">=</span> <span class="pl-c1">50
```

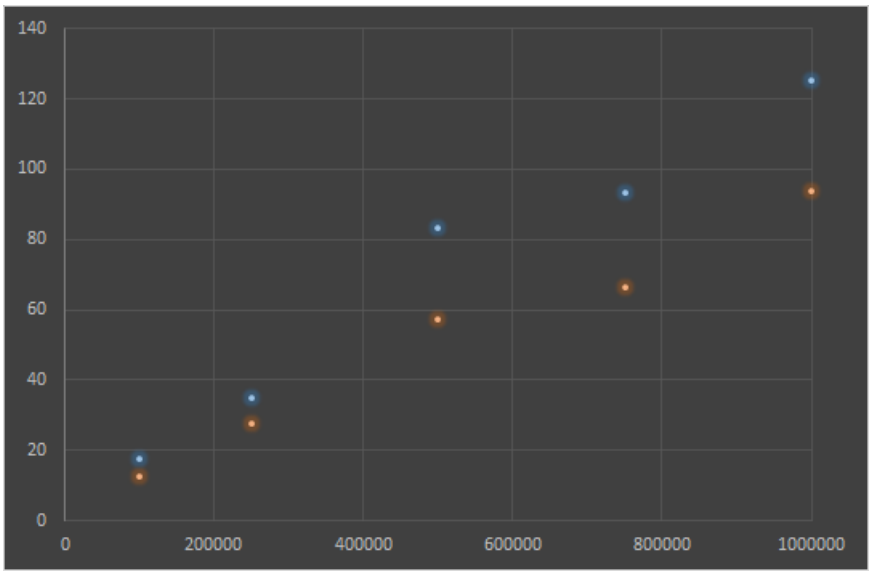
```
</span><span class="pl-c">
start = time.time()
pi4 = 0.0
# This is the master node.
</span><span class="pl-k">if</span> rank <span class="pl-k">==</span> <span class="pl-
c1">0</span>:
    pi <span class="pl-k">=</span> <span class="pl-c1">0
</span>    <span class="pl-s3">slice</span> <span class="pl-k">=</span>
    process <span class="pl-k">=</span> <span class="pl-c1">1
</span>
    <span class="pl-k">print</span> size

    <span class="pl-c"># Send the first batch of processes to the nodes.</span>
    <span class="pl-k">while</span> process <span class="pl-k">&lt;</span> size <span
class="pl-k">and</span> <span class="pl-s3">slice</span> <span class="pl-k">&lt;</span>
total_slices:
        comm.send(<span class="pl-s3">slice</span>, <span class="pl-vpf">dest</span><span
class="pl-k">=</span>process, <span class="pl-vpf">tag</span><span class="pl-k">=</span><span
class="pl-c1">1</span>)
        <span class="pl-k">print</span> <span class="pl-s1"><span class="pl-pds">"
</span>Sending slice<span class="pl-pds">"</span></span>,<span class="pl-
s3">slice</span>,<span class="pl-s1"><span class="pl-pds">"</span>to process<span
class="pl-pds">"</span></span>,process
        <span class="pl-s3">slice</span> <span class="pl-k">+=</span> <span class="pl-
c1">1</span>
        process <span class="pl-k">+=</span> <span class="pl-c1">1</span>

    <span class="pl-c"># Wait for the data to come back</span>
    received_processes <span class="pl-k">=</span> <span class="pl-c1">0</span>
    <span class="pl-k">while</span> received_processes <span class="pl-k">&lt;</span>
total_slices:
        pi <span class="pl-k">+=</span> comm.recv(<span class="pl-vpf">source</span><span
class="pl-k">=</span>MPI.ANY_SOURCE, <span class="pl-vpf">tag</span><span class="pl-
k">=</span><span class="pl-c1">1</span>)
        process <span class="pl-k">=</span> comm.recv(<span class="pl-vpf">source</span><span
class="pl-k">=</span>MPI.ANY_SOURCE, <span class="pl-vpf">tag</span><span class="pl-
k">=</span><span class="pl-c1">2</span>)
        <span class="pl-k">print</span> <span class="pl-s1"><span class="pl-pds">"
</span>Recieved data from process<span class="pl-pds">"</span></span>, process
        received_processes <span class="pl-k">+=</span> <span class="pl-c1">1</span>

        <span class="pl-k">if</span> <span class="pl-s3">slice</span> <span class="pl-
k">&lt;</span> total_slices:
            comm.send(<span class="pl-s3">slice</span>, <span class="pl-vpf">dest</span><span
class="pl-k">=</span>process, <span class="pl-vpf">tag</span><span class="pl-k">=</span><span
class="pl-c1">1</span>)
            <span class="pl-k">print</span> <span class="pl-s1"><span class="pl-pds">"
</span>Sending slice<span class="pl-pds">"</span></span>,<span class="pl-
s3">slice</span>,<span class="pl-s1"><span class="pl-pds">"</span>to process<span
class="pl-pds">"</span></span>,process
            <span class="pl-s3">slice</span> <span class="pl-k">+=</span> <span class="pl-
c1">1</span>

    <span class="pl-c"># Send the shutdown signal</span>
    <span class="pl-k">for</span> process <span class="pl-k">in</span> <span class="pl-
s3">range</span>(<span class="pl-c1">1</span>,size):
        comm.send(<span class="pl-k">-</span><span class="pl-c1">1</span>, <span class="pl-
vpf">dest</span><span class="pl-k">=</span>process, <span class="pl-vpf">tag</span><span
class="pl-k">=</span><span class="pl-c1">1</span>)

    pi4 = 4.0 * pi
    #<span class="pl-k">print</span> <span class="pl-s1"><span class="pl-pds">"</span>Pi is
<span class="pl-pds">"</span></span>, <span class="pl-c1">4.0</span> <span class="pl-k">*
</span> pi

<span class="pl-c"># These are the slave nodes, where rank &gt; 0. They do the real
work</span>
<span class="pl-k">else</span>:
    <span class="pl-k">while</span> <span class="pl-c1">True</span>:
        start <span class="pl-k">=</span> comm.recv(<span class="pl-vpf">source</span><span
class="pl-k">=</span><span class="pl-c1">0</span>, <span class="pl-vpf">tag</span><span
class="pl-k">=</span><span class="pl-c1">1</span>)
        <span class="pl-k">if</span> start <span class="pl-k">==</span> <span class="pl-
k">-</span><span class="pl-c1">1</span>: <span class="pl-k">break</span>

        i <span class="pl-k">=</span> <span class="pl-c1">0</span>
        slice_value <span class="pl-k">=</span> <span class="pl-c1">0</span>
        <span class="pl-k">while</span> i <span class="pl-k">&lt;</span> slice_size:
            <span class="pl-k">if</span> i<span class="pl-k">%</span><span class="pl-
c1">2</span> <span class="pl-k">==</span> <span class="pl-c1">0</span>:
                slice_value <span class="pl-k">+=</span> <span class="pl-c1">1.0</span> <span
class="pl-k">/</span> (<span class="pl-c1">2</span><span class="pl-k">*</span>(start<span
class="pl-k">*</span>slice_size<span class="pl-k">+</span>i)<span class="pl-k">+</span><span
class="pl-c1">1</span>)
            <span class="pl-k">else</span>:
                slice_value <span class="pl-k">-=</span> <span class="pl-c1">1.0</span> <span
```

```
class="pl-k">/</span> (<span class="pl-c1">2</span><span class="pl-k">*</span>(start<span
class="pl-k">*</span>slice_size<span class="pl-k">+</span>i)<span class="pl-k">+</span><span
class="pl-c1">1</span>)
            i <span class="pl-k">+=</span> <span class="pl-c1">1</span>

        comm.send(slice_value, <span class="pl-vpf">dest</span><span class="pl-
k">=</span><span class="pl-c1">0</span>, <span class="pl-vpf">tag</span><span class="pl-
k">=</span><span class="pl-c1">1</span>)
        comm.send(rank, <span class="pl-vpf">dest</span><span class="pl-k">=</span><span
class="pl-c1">0</span>, <span class="pl-vpf">tag</span><span class="pl-k">=</span><span
class="pl-c1">2</span>)

if rank == 0:
   end = time.time()
   error = abs(pi4 - math.pi)
   print ("pi is approximately %.10f, "
          "error is %.10f" % (pi4,error))
   print ("elapsed time: %.2f" % (end - start))
```

Ho calcolato i tempi di calcolo mantenendo costante il numero delle slice ma variando la loro dimensione.

SLICES = 10

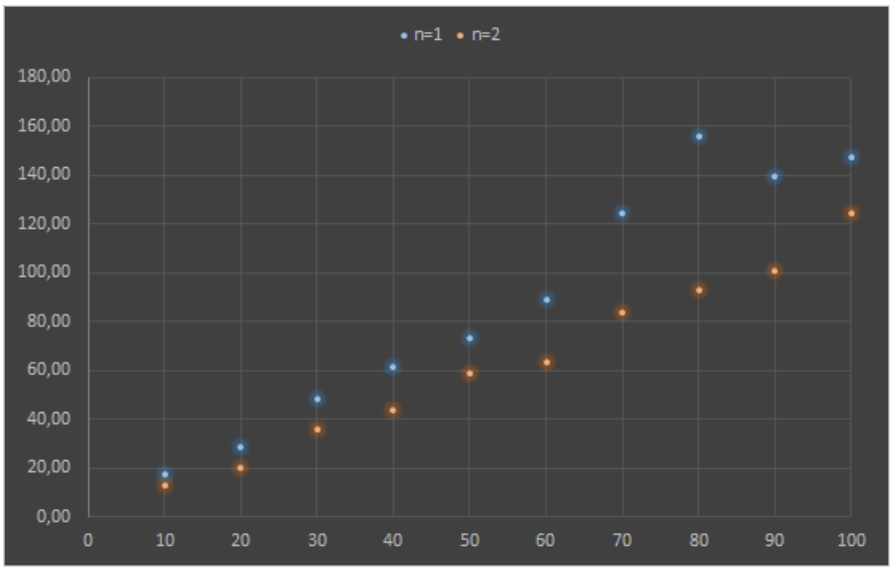
SLICE SIZE	CLUSTER NODES = 1 (time elapsed)	CLUSTER NODES = 2 (time elapsed)
100000	17.72	12.77
250000	35.15	27.50
500000	83.11	57.31
750000	93.16	66.39
1000000	125.07	93.55



Adesso vediamo i tempi richiesti per il calcolo mantenendo costante la dimensione delle slice ma incrementando il loro numero.

SLICESIZE = 100000

SLICES	CLUSTER NODES = 1 (time elapsed)	CLUSTER NODES = 2 (time elapsed)
10	17.72	12.77
20	28.46	20.27
30	48.49	35.49
40	61.53	43.72
50	73.11	58.64
60	89.08	63.52
70	124.47	83.56
80	156.14	92.92
90	139.66	101.02
100	147.11	124.36


n=1
n=2
180,00
160,00
140,00
120,00
100,00
80,00
60,00
40,00
20,00
0,00
0
10
20
30
40
50
60
70
80
90
100