|  |  |
| --- | --- |
|  | Escuela de Informática.  Departamento de Informática y Computación. |

Anexos adicionales.

Modificaciones realizadas al proyecto.

|  |
| --- |
| Asignatura: Computación Paralela.  Profesor: Oscar Magna Veloso.  Alumnos:  Sergio Abarca Flores.  Rosa González Soto.  Diego Hernández García.  Roberto Oñate Piedras.  Martes, 28 de abril de 2015. |

# Introducción

El informe entregado el jueves 23 de abril del presente año contempla una de las dos soluciones desarrolladas para este proyecto, debido al tiempo, se entregó aquella única solución. Luego de la entrega, los avances en la solución no desarrollada se lograron alcanzar, dando como resultado esta modificación al informe entregado, el cual mostrara los tiempos, el código de la solución y las conclusiones.

La solución entregada en el informe se describe en breves palabras como:

*“Dividir los paquetes de datos en la cantidad máxima de procesadores disponibles, para que así cada procesador trabaje con una medida igualitaria entre todos, reduciendo el tiempo de espera y de comunicación”*

Tal solución entregó buenos resultados, bajando el tiempo a 14,5 segundos en promedio, pero al no existir otra solución que sirviera de comparación, no fue factible determinar qué tan bueno fue la solución entregada. Otra limitación de la solución anterior es sobre la cantidad de procesadores se utilizaran, solo sirve para 40 procesadores, al existir menos procesadores disponibles se generan esperas ya que la división de los paquetes de datos se realizó de manera física (es decir, se tomó el universo de datos y se dividió en 40, generando 40 archivos CSV de datos),

Con esta nueva solución se espera mantener un bajo tiempo de procesamiento y corregir las limitantes que presenta la solución anterior.

Cabe mencionar que para nombrar los anexos siguientes, se mantendrá la numeración utilizada en el informe entregado. El último anexo escrito en aquel informe fue

*“Anexo E: Ejemplo de código paralelo”*

Por lo tanto, el primer anexo comenzara con la letra F.

# Tabla de contenidos

[Introducción 2](#_Toc417947825)

[Anexo F: Código segunda solución. 4](#_Toc417947826)

[1. Explicación código 7](#_Toc417947827)

[Anexo G: Tiempos. 8](#_Toc417947828)

[1. Primera medición. 8](#_Toc417947829)

[2. Segunda medición. 9](#_Toc417947830)

[3. Tercera medición. 10](#_Toc417947831)

[4. Observaciones. 10](#_Toc417947832)

[Anexo H: Métricas. 11](#_Toc417947833)

[1. Speed Up 11](#_Toc417947834)

[2. Eficiencia. 12](#_Toc417947835)

[Anexo I: Análisis y conclusiones. 13](#_Toc417947836)

# Anexo F: Código segunda solución.

from mpi4py import MPI

import csv

import random

import sys

from time import time

comm = MPI.COMM\_WORLD

rank = comm.rank

size = comm.size

name = MPI.Get\_processor\_name()

# Funcion que genera un voto en base a la probabilidad que tiene la comuna

# es por esto que se entregan como rango1 el porcentaje que tiene el

# partido de izquierda y rango2 como las suma de la probabilidad de

# partido de izquierda junto con la probabilidad del partido de derecha.

def Voto(rango1, rango2):

probabilidad = random.randint(1, 100)

voto = 0

# rango perteneciente a partido de izquierda.

if probabilidad <= rango1:

voto = 1

# rango perteneciente a partido de derecha.

elif probabilidad > rango1 and probabilidad <= rango2:

voto = 2

# rango perteneciente a partido independiente.

else:

voto = 3

return voto

# Funcion que lee el archivo csv que contiene las comunas y

# sus probabilidades de cada partido politico. Estos datos se

# trasladan a una lista para reducir el tiempo de lectura de

# los datos al generar los datos con la funcion Voto

def LeerComunas():

# Apertura de archivo.

reader = csv.reader(open('Data/comunas.csv', 'rb'))

lista = []

for i, row in enumerate(reader):

# Particionar la informacion con el delimitador ";"

particion = str(row[0]).split(";")

comuna = particion[0]

region = particion[1]

derecha = particion[2]

izquierda = particion[3]

independiente = particion[4]

dato = [comuna, region, derecha,

izquierda, independiente]

lista.append(dato)

return lista

# Funcion que lee la informacion entregada por SERVEL

# comparando una comuna con sus probabilidades y utilizando

# la funcion Voto. Teniendo el voto generado, se asigna al

# partido politico.

def contarVoto(comunas, inicio, fin, rank, name):

wt = MPI.Wtime()

suma = 0

sumb = 0

sumc = 0

f = open('Data/data\_total.csv', 'r')

p = f.readlines()

primera = p[inicio:fin]

# sw = 0

for l in primera:

particion = str(l).split(";")

comuna = particion[1].rstrip()

for j in range(len(comunas)):

# Se busca que la comuna leida en el archivo "data\_server\_'xx'"

# coincida con la lista de comunas, para asi, obtener sus

# probabilidades.

if comuna == comunas[j][0]:

# Generar rango partido Izquierda

rango1 = int(comunas[j][2])

# Generar rango partido Derecha

rango2 = (int(comunas[j][2]) + int(comunas[j][3]))

# Guardar el voto generado con la funcion Voto

voto = Voto(rango1, rango2)

# Acumular los votos de cada partido politico donde:

# suma guarda los votos de izquierda

# sumb guarda los votos de derecha

# sumc guarda los votos independiente.

if voto == 1:

suma = suma + 1

elif voto == 2:

sumb = sumb + 1

else:

sumc = sumc + 1

# sw = 1

# if sw == 0:

# print comuna + "hola"

sumtotal = suma + sumb + sumc

# print "Suma total: " + str(sumtotal)

wt = MPI.Wtime() - wt

lista = dict(

izquerda=suma, derecha=sumb, independiente=sumc, total=sumtotal)

comm.send(lista, dest=0)

sys.stdout.write("Termino Proceso %d en %s.\n" % (rank, name))

def main():

if rank == 0:

print "\*\*\*\*\*Paralelo Granularidad Fina de Codigo\*\*\*\*\*"

tiempo\_inicial = time()

total = 12905887

rango = int(total / (size-1))

resto = total - (rango \* (size-1))

process = 0

for i in range(size - 2):

inicio = rango \* i

fin = rango \* (i + 1)

process = process + 1

rangos = [inicio, fin]

comm.send(rangos, dest=process)

inicio = fin

fin = inicio + rango + resto

process = process + 1

rangos = [inicio, fin]

comm.send(rangos, dest=process)

if rank != 0:

archivo = comm.recv(source=0)

print archivo[0]

print archivo[1]

comunas = LeerComunas()

contarVoto(comunas, archivo[0], archivo[1], rank, name)

if rank == 0:

izquerda = 0

derecha = 0

independiente = 0

totales = 0

for i in range(1, size):

voto = comm.recv(source=i)

#print voto

izquerda = izquerda + voto["izquerda"]

derecha = derecha + voto["derecha"]

independiente = independiente + voto["independiente"]

totales = totales + voto["total"]

print " Izquierda: " + str(izquerda)

print " Derecha: " + str(derecha)

print " Independiente: " + str(independiente)

print " Votos totales: " + str(totales)

tiempo\_final = time() - tiempo\_inicial

print "Tiempo total de ejecucion: " + str(tiempo\_final)

print "Programa Finalizado"

return 0

main()

## Explicación código

El código mostrado, elimina la limitación de solo procesar en 40 nodos.

Cuando se ejecuta la instrucción:

$mpirun -np (cantidad procesador) –hostfile (ruta del archivo hostfile) python (archivo python).py

Posiblemente se disponga de una menor cantidad de procesadores, es por esto que la solución se encarga de dividir la información de manera lógica, pero primero para realizar esto, es necesario que todos los datos a procesar estén en un solo archivo CSV.

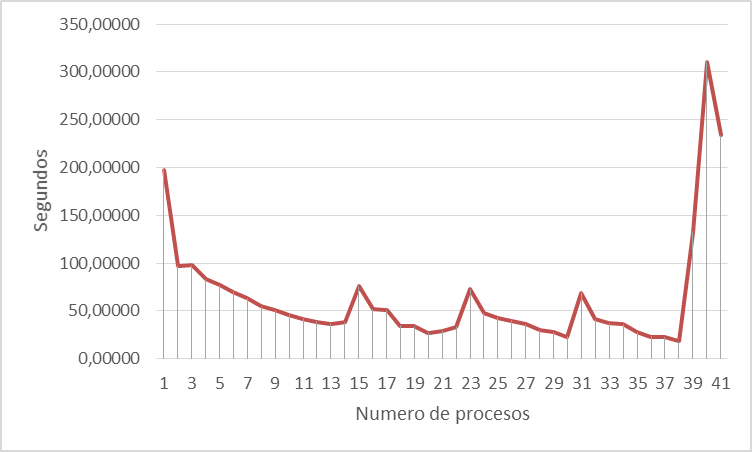
El programa lee la cantidad de procesadores y divide en n-1 el archivo CSV. Es en n-1 porque el primer procesador se encarga de distribuir y recopilar la información entregada por el resto de los procesadores, por ende, no realiza cálculos sobre el archivo CSV.

Otra gran ventaja de esta solución, es la posibilidad de medir cual es la cantidad óptima de nodos que se deben utilizar en base a los tiempos de mejora que se presentan.

Las funciones utilizadas para generar los votos, obtener las probabilidades y leer las comunas, se mantienen en su forma. Solo cambia la forma de paralelizar el programa.

# Anexo G: Tiempos.

## Primera medición.

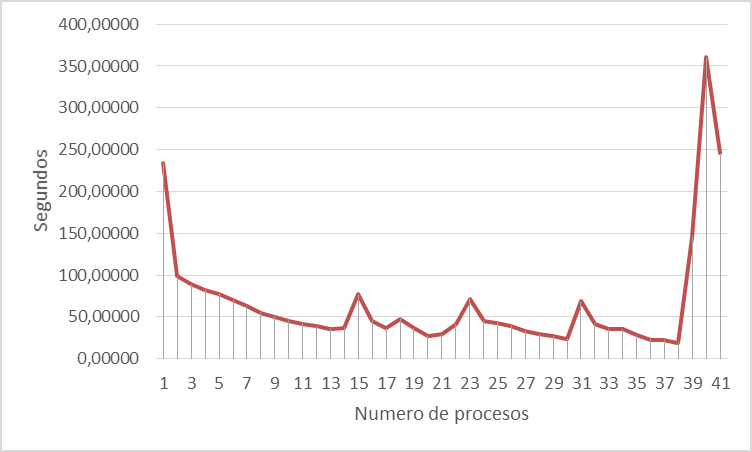


|  |  |
| --- | --- |
| Numero de procesos | Segundos |
| 3 | 197,36392 |
| 4 | 97,30442 |
| 5 | 97,76168 |
| 6 | 83,02806 |
| 7 | 76,82069 |
| 8 | 70,12196 |
| 9 | 63,35970 |
| 10 | 54,67005 |
| 11 | 50,44911 |
| 12 | 46,13908 |
| 13 | 41,76542 |
| 14 | 38,81648 |
| 15 | 36,25049 |
| 16 | 37,93338 |
| 17 | 75,76062 |
| 18 | 51,81760 |
| 19 | 50,70396 |
| 20 | 34,16224 |
| 21 | 33,95450 |
| 22 | 26,98362 |
| 23 | 28,91886 |
| 24 | 33,14123 |
| 25 | 72,68018 |
| 26 | 47,61401 |
| 27 | 42,74300 |
| 28 | 39,60700 |
| 29 | 35,83529 |
| 30 | 29,55056 |
| 31 | 27,49209 |
| 32 | 22,77029 |
| 33 | 69,11830 |
| 34 | 41,01511 |
| 35 | 36,83064 |
| 36 | 35,86724 |
| 37 | 27,56477 |
| 38 | 22,66339 |
| 39 | 22,37952 |
| 40 | 18,00140 |
| 45 | 132,54954 |
| 50 | 310,80487 |
| 60 | 233,88293 |

|  |
| --- |
| Grafico 1: Primera medición de tiempos, Elaboración Grupal |

Tabla 1: Primera medición de tiempos, Elaboración Grupal.

## Segunda medición.

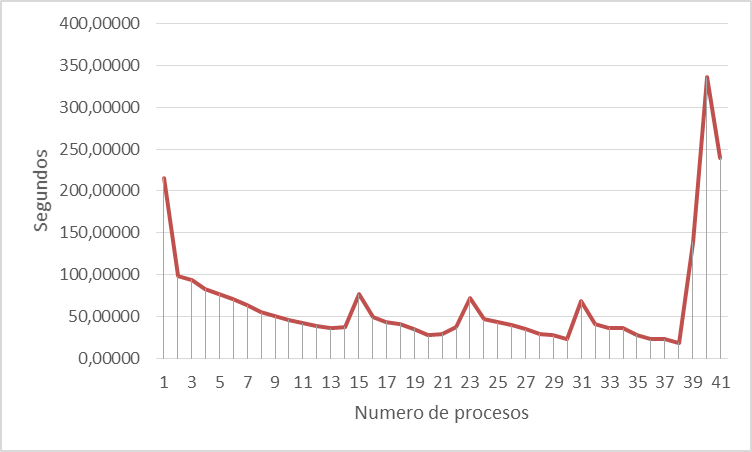


|  |  |
| --- | --- |
| Numero de procesos | Segundos |
| 3 | 233,89920 |
| 4 | 98,74000 |
| 5 | 89,28880 |
| 6 | 82,62490 |
| 7 | 77,51420 |
| 8 | 70,63350 |
| 9 | 62,73270 |
| 10 | 55,05000 |
| 11 | 49,96860 |
| 12 | 45,25210 |
| 13 | 41,86600 |
| 14 | 38,73130 |
| 15 | 35,82610 |
| 16 | 37,08910 |
| 17 | 77,16270 |
| 18 | 45,51340 |
| 19 | 36,60210 |
| 20 | 47,94790 |
| 21 | 36,18340 |
| 22 | 27,50930 |
| 23 | 29,52270 |
| 24 | 41,38040 |
| 25 | 71,52050 |
| 26 | 45,33050 |
| 27 | 43,09810 |
| 28 | 39,51090 |
| 29 | 33,50810 |
| 30 | 29,45220 |
| 31 | 27,11560 |
| 32 | 23,90020 |
| 33 | 68,48950 |
| 34 | 41,56120 |
| 35 | 35,77120 |
| 36 | 35,22890 |
| 37 | 28,26810 |
| 38 | 22,68910 |
| 39 | 22,37980 |
| 40 | 19,00120 |
| 45 | 146,14150 |
| 50 | 360,73764 |
| 60 | 245,86182 |

Tabla 2: segunda medición de tiempos, Elaboración grupal.

|  |
| --- |
| Grafico 2: Segunda medición de tiempos, Elaboración Grupal |

## Tercera medición.



|  |  |
| --- | --- |
| Numero de procesos | Segundos |
| 3 | 215,63156 |
| 4 | 98,02221 |
| 5 | 93,52524 |
| 6 | 82,82648 |
| 7 | 77,16744 |
| 8 | 70,37773 |
| 9 | 63,04620 |
| 10 | 54,86003 |
| 11 | 50,20886 |
| 12 | 45,69559 |
| 13 | 41,81571 |
| 14 | 38,77389 |
| 15 | 36,03830 |
| 16 | 37,51124 |
| 17 | 76,46166 |
| 18 | 48,66550 |
| 19 | 43,65303 |
| 20 | 41,05507 |
| 21 | 35,06895 |
| 22 | 27,24646 |
| 23 | 29,22078 |
| 24 | 37,26081 |
| 25 | 72,10034 |
| 26 | 46,47226 |
| 27 | 42,92055 |
| 28 | 39,55895 |
| 29 | 34,67169 |
| 30 | 29,50138 |
| 31 | 27,30384 |
| 32 | 23,33524 |
| 33 | 68,80390 |
| 34 | 41,28815 |
| 35 | 36,30092 |
| 36 | 35,54807 |
| 37 | 27,91643 |
| 38 | 22,67625 |
| 39 | 22,37966 |
| 40 | 18,50130 |
| 45 | 139,34552 |
| 50 | 335,77125 |
| 60 | 239,87237 |

Tabla 3: Tercera medición de tiempos, Elaboración grupal.

|  |
| --- |
| Grafico 3: Tercera medición de tiempos, Elaboración Grupal |

## Observaciones.

Si bien los tiempos de respuestas fueron muy similares en las tres tomas de muestras, la razón por la cual se presentan diferencias en una misma cantidad de nodos es debido al ruido en el switch generado por los demás computadores que se utilizaban en el laboratorio número 5 (lugar donde está montada la plataforma clúster) y también a la utilización de internet de los computadores pertenecientes a la red.

# Anexo H: Métricas.

Antes de realizar las mediciones correspondientes, es necesario rescatar la siguiente información que deriva del informe entregado (Pagina 29, 11.1 Speed-Up)

* Peor tiempo secuencial: 250,994252
* Mejor tiempo paralelo granularidad gruesa: 44,121425
* Peor tiempo paralelo granularidad gruesa: 49,7198160
* Mejor tiempo paralelo granularidad fina: 14,5342152

Los nuevos tiempos entregados por la granularidad gruesa son:

* Mejor tiempo paralelo granularidad gruesa: 18,00140 utilizando 40 procesos.
* Peor tiempo paralelo granularidad gruesa: 360,73764 utilizando 50 procesos.

## Speed Up

El primer speed Up queda representado de la siguiente manera:

Esto representa una mejora porcentualmente de 1394,30% equivalente a decir que la mejora es casi de 14 veces más comparado con el mejor tiempo encontrado en programa secuencial.

El segundo Speed Up, se compara el mejor tiempo que se tenía del algoritmo paralelo granularidad fina.

Esto representa una mejora porcentualmente de un 2481,24% equivalente a decir que la mejora es superior a 24 veces más comparado el peor tiempo que entrego esta nueva medida con el mejor tiempo de la solución anterior.

## Eficiencia.

*0,348576035*

Al comparar los resultados con los mostrados en el informe anterior (Pagina 30, 11.2 Eficiencia) podemos comprobar que este programa es más eficiente que el anterior, esto se explica por el hecho de manejar de mejor manera los datos que serán utilizados.

# Anexo I: Análisis y conclusiones.

Al revisar las tablas de los tiempos obtenidos por el nuevo programa en paralelo se observó que mientras se aumentaba la cantidad de procesadores ingresados, iba disminuyendo el tiempo de respuesta, hay casos excepcionales, como por ejemplo, en la cantidad de 17 procesadores. Con 16 procesadores el tiempo de respuesta es de 37,94 segundos, al utilizar 17 procesadores se rompe esa tendencia, haciendo que el tiempo de respuesta aumente a casi el doble, con un tiempo de 75,76 segundos, para luego utilizar 18 procesadores y vuelve a la tendencia de bajar los tiempos de respuesta a 51,82 segundos. Esto se repitió 3 veces, en la cantidad 17 (ya mencionada) 25 y 33 procesadores.

También se puede obtener de la tabla, que la cantidad óptima de procesadores a utilizar para este programa son 40. Sorpresivamente es la misma cantidad de procesadores utilizados en el código paralelo granularidad fina (página 16, 9.2.1 Granularidad Fina) pero en este caso se tiene una diferencia de 4 segundos, la razón de este aumento de tiempo, es la cantidad de procesamiento que se debe hacer para dividir de manera lógica un solo archivo CSV. A priori, pareciera ser mejor el algoritmo de granularidad fina, pero el algoritmo mencionado, no tiene la capacidad de procesar la información con menos o más procesadores, solo con 40. Esto representa una gran ventaja para el código desarrollado en este anexo, porque en tiempo de respuesta es mayor, pero tiene la ventaja de ser genérico, pudiendo ayudar en otros ámbitos que cumplan con el modelo de paralización SIMD (Single Instruction Multiple Data).

Al tener ambas soluciones desarrolladas se comprobó que la primera solución mostrada (informe anterior) no era la mejor alternativa, dando paso, por el momento, a la solución desarrollada en este anexo sea la mejor que hallamos desarrollado como grupo bajo las métricas de Speed Up y Eficiencia.