Computación Avanzada

Práctico Final:

Simplex Paralelo

4° Licenciatura En Sistemas de Información

2020

Universidad: UADER FCyT Concepción del Uruguay

Profesores: María Fabiana Piccoli, Carlos Casanova

Alumnos: Cepeda Leandro, Gonzalez Exequiel

Práctico Final: Computación Avanzada

1. Estudie el código del algoritmo simplex revisado.
2. Identifique partes del algoritmo que puedan ser paralelizables y proponga alternativas para hacerlo.
3. Escriba una versión paralela del algoritmo simplex y compárela con la versión secuencial utilizando al menos 5 instancias de PL distintas. Recomendamos que las ejecuciones sean locales en la misma PC para hacer la comparación lo más justa posible.

2. Identifique partes del algoritmo que puedan ser paralelizables y proponga alternativas para hacerlo.

A. El producto de matrices de la librería numpy.

```
zj_cj = np.round(np.matmul(c_p[base], np.matmul(B_1, A)) - c_p, 10)
A_ve = np.matmul(B_1, A[:, ve])
b_p = np.matmul(B_1, b)
B_1 = np.matmul(E_1, B_1)
zj_cj = np.round(np.matmul(c_p[base], np.matmul(B_1, A)) - c_p, 10)
z_opt = np.matmul(c_p[base], b_p)
```

Para paralelizar esta parte del problema definimos una función que se encarga de realizar la multiplicación de matrices en paralelo, si el proceso es el 0, le envía a cada uno de los procesos restantes un índice que indica a partir de que columna de la matriz A (se utiliza también para las filas de la matriz B) le corresponde calcular a cada proceso (en el caso que la cantidad de columnas sea múltiplo de la cantidad de procesos se le asigna a todos la misma cantidad de columnas, caso contrario al último proceso se le asignan las columnas sobrantes).
Luego cada uno de los procesos excepto el 0, realizan los cálculos en paralelo del producto de matrices y le envían los resultados al proceso 0.
Finalmente el proceso 0 recibe de todos los demás procesos el resultado de cada uno, y lo agrupa todo en la matriz resultado.

```
#Se calcula la cantidad de columnas que le corresponde a cada proceso
cols_per_proces = cols_A // (total_processes)

#Para el caso en el que los datos a repartir y los procesos no son múltiplos
rest = cols_A % (total_processes)
if (rank == (total_processes-1)) and rest != 0:
  cols_per_proces += rest

for i in range (rows_A):
  for j in range (cols_B):

    suma = 0

    """
    ********************************************************************************************
    ################### START MULTIPLICATION OF MATRICES IN PARALLEL ####################
    ********************************************************************************************
    """

    if rank == 0:

      #Se reparten las columnas de A que van a trabajar los procesos
      process = 1
      for col in range(cols_per_proces,cols_A,cols_per_proces):

        if (process == (total_processes)) and (cols_A % (total_processes) != 0):
          break
        else:
          comm.send(col,dest=process)
        process += 1

      #El proceso 0 hace su parte

      for k in range(0,cols_per_proces):
        suma += A[i,k]*B[k,j]

      #El proceso 0 recibe los resultados y acumula todo en la matriz resultado
      for proc in range(1,total_processes):
        suma += comm.recv(None,source=proc)

      C[i,j] = suma

    else:

      #Cada proceso recibe su parte y resuelve el calculo
      col = comm.recv(None,source=0)
      for k in range(col,col+cols_per_proces):
        suma += A[i,k]*B[k,j]

      comm.send(suma,dest=0)

    """
    ********************************************************************************************
    ##################### END MULTIPLICATION OF MATRICES IN PARALLEL ####################
    ********************************************************************************************
    """
```

B. Primer ciclo for de la matriz extendida.

```
for i in range(cant_lt):
  A[:, n+i] = [(1. if i == j else 0) for j in range(m)]
  base.append(n+i)
```

Para resolver esto, definimos que el proceso 0 le envía a cada uno de los procesos restantes un índice que indica a partir de que fila le corresponde calcular a cada proceso (en el caso que la cantidad de filas sea múltiplo de la cantidad de procesos se le asigna a todos la misma cantidad de filas, caso contrario al último proceso se le asignan las filas sobrantes).
Cada uno de los procesos excepto el 0, calcula las filas correspondientes agregando los 1 y 0 respectivamente, y envían los resultados al proceso 0.
Por úĺtimo el proceso 0 recibe de todos los demás procesos el resultado de cada uno, agrega a las filas de la matriz extendida los vectores con 1 y 0, y almacena en la base los índices correspondientes.

```
#Se calcula la cantidad de filas que le corresponde a cada proceso
 amount_per_process = amount_lt // (total_processes)
 rest = amount_lt % (total_processes)

 #Para el caso en el que las filas a repartir y los procesos no son múltiplos
 if (rank == (total_processes-1)) and rest != 0:
  amount_per_process += rest

 """
 ********************************************************************************************
 ################## START FIRST FOR OF MATRIX EXTENDED IN PARALLEL ##################
 ********************************************************************************************
 """

 if rank == 0:

  #Se envia a cada proceso su parte para calcular
  process = 1
  for i in range(amount_per_process,amount_lt,amount_per_process):
   if (process == (total_processes)) and (amount_lt % (total_processes) != 0):
    break
   else:
    comm.send(i,dest=process)
   process += 1

  for x in range(0,amount_per_process):
   A[:,n+x] = [(1. if x == j else 0) for j in range(m)]
   base.append(n+x)

  #Agrupa los vectores soluciones en la base y la matriz A
  for i in range(1,total_processes):
   result = comm.recv(None,source=i)
   for tupla in result:
    pos = tupla[1]
    A[:,n+pos] = tupla[0]
    base.append(n+pos)

  #En estos for, la cantidad de iteraciones son menor que la cantidad de procesos en los que se
  #puede repartir, por lo tanto no los paralelizamos y dejamos que el proceso 0 realice los calculos
  for i in range(amount_lt, amount_lt + 2*amount_gt, 2):
   A[:, n+i] = [(-1. if i == j else 0) for j in range(m)]
   A[:, n+i+1] = [(1. if i == j else 0) for j in range(m)]
   c_p[n+i+1] = -M
   base.append(n+i+1)

  for i in range(amount_lt + 2*amount_gt, amount_lt + 2*amount_gt + amount_eq):
   A[:, n+i] = [(1. if i == j else 0) for j in range(m)]
   c_p[n+i] = -M
   base.append(n+i)

 else:

  #Cada proceso recibe y guarda en una lista las tuplas con el vector y la posición correspondiente en cada una
  i = comm.recv(None,source=0)
  result = []
  for x in range(i,i+amount_per_process):
   result.append(([(1. if x == j else 0) for j in range(m)],x))
  comm.send(result,dest=0)

 """
 ********************************************************************************************
 ################### END FIRST FOR OF MATRIX EXTENDED IN PARALLEL ###################
 ********************************************************************************************
 """
```

C. Calculo de los cocientes de las titas.

titas = [(b_p[i]/A_ve[i] if A_ve[i] > 0 else np.nan) for i in range(m)]

En esta parte del código lo que hacemos es, si es el proceso 0 se le envía a cada uno de los procesos restantes un índice que indica a partir de que fila deberá calcular los cocientes de las titas (en el caso que la cantidad de filas sea múltiplo de la cantidad de procesos se le asigna a todos la misma cantidad de filas, caso contrario al último proceso se le asignan las filas sobrantes).
Luego cada proceso exceptuando el 0, realiza la operación que le corresponde y envía al proceso 0 una lista con los resultados.
El proceso 0 recibe los resultados de cada uno de los procesos restantes, y por último los agrupa en la lista de titas.

```
#Se calcula la cantidad de filas que le corresponde a cada proceso
  amount_per_process = m // (total_processes)
  rest = m % (total_processes)

  #Para el caso en el que las filas a repartir y los procesos no son múltiplos
  if (rank == (total_processes-1)) and rest != 0:
    amount_per_process += rest

  titas = []

  """
  ********************************************************************************************
  ######################## START CALCULATION OF TITAS IN PARALLEL ########################
  ********************************************************************************************
  """

  if rank == 0:

    #Se envia a cada proceso su parte para calcular las titas
    process = 1
    for i in range(amount_per_process,m,amount_per_process):

      if (process == (total_processes)) and (m % (total_processes) != 0):
         break
      else:
         comm.send(i,dest=process)
      process += 1

    #El proceso 0 hace su parte
    for x in range(0,amount_per_process):
      if A_ve[x] > 0:
        calculo = b_p[x]/A_ve[x]
      else:
        calculo = np.nan
      titas.append(calculo)

    #Agrupa los resultados en el vector de titas
    for i in range(1,total_processes):
      result = comm.recv(None,source=i)
      titas.extend(result)

  else:

    #Cada proceso calcula el valor de la tita en el caso que sea mayor a 0 y lo guarda en una lista
    i = comm.recv(None,source=0)
    result = []
    for x in range(i,i+amount_per_process):
      if A_ve[x] > 0:
        calculo = b_p[x]/A_ve[x]
      else:
        calculo = np.nan
      result.append(calculo)
    comm.send(result,dest=0)

  """
  ********************************************************************************************
  ######################## END CALCULATION OF TITAS IN PARALLEL ########################
  ********************************************************************************************
  """
```

3. Escriba una versión paralela del algoritmo simplex y compárela con la versión secuencial utilizando al menos 5 instancias de PL distintas. Recomendamos que las ejecuciones sean locales en la misma PC para hacer la comparación lo más justa posible.

Test N°1

	Modelo PL	Simplex Paralelo (procesos)	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.7232654094696045	1
2	Variables: 50, Restricciones: 70	1.5151245594024658	2
3	Variables: 70, Restricciones: 90	38.67264461517334	3
4	Variables: 90, Restricciones: 110	29.333220958709717	4

Test N°2

	Modelo PL	Simplex Paralelo (procesos)	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.6944544315338135	1
2	Variables: 50, Restricciones: 70	1.4137802124023438	2
3	Variables: 70, Restricciones: 90	43.80835819244385	3
4	Variables: 90, Restricciones: 110	39.923725843429565	4

Test N°3

	Modelo PL	Simplex Paralelo (procesos)	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.7717733383178711	1
2	Variables: 50, Restricciones: 70	2.7528836727142334	2
3	Variables: 70, Restricciones: 90	39.423641204833984	3
4	Variables: 90, Restricciones: 110	28.92477059364319	4

Test N°4

	Modelo PL	Simplex Paralelo (procesos)	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.6609573364257812	1
2	Variables: 50, Restricciones: 70	1.8954312801361084	2
3	Variables: 70, Restricciones: 90	37.13983750343323	3
4	Variables: 90, Restricciones: 110	31.997485637664795	4

Test N°5

	Modelo PL	Simplex Paralelo (procesos)	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.6215803623199463	1
2	Variables: 50, Restricciones: 70	1.6848390102386475	2
3	Variables: 70, Restricciones: 90	37.13983750343323	3
4	Variables: 90, Restricciones: 110	30.67760682106018	4

Promedio

	Modelo PL	Promedio	Cantidad de procesos
1	Variables: 30, Restricciones: 50	0.694406176	1
2	Variables: 50, Restricciones: 70	1.852411747	2
3	Variables: 70, Restricciones: 90	43.912871647	3
4	Variables: 90, Restricciones: 110	32.171361971	4

En el siguiente gráfico podemos observar que el paralelismo realizado bajo el rendimiento de la ejecución del programa en comparación con el algoritmo secuencial que empleaba las librerías de numpy.