Relatório Projeto 3 - CUDA - Supercomp 2019.2

Frederico Vilela Curti

Descrição do problema tratado

O problema tratado nesse projeto é o problema do Caixeiro Viajante, no qual, de forma lúdica, um vendedor possui uma lista de empresas que ele deverá visitar ao longo do dia, e não existe uma ordem para tal. Desde que todas as empresas sejam visitadas, o objetivo do dia está cumprido. Para otimizar esse percurso e passar o maior tempo possível com seus clientes, ele precisa encontrar uma rota que resulta no menor caminho percorrido ao final do dia. Nesse problema, as empresas serão representadas por pontos em um plano. O desafio desse projeto era implementar uma solução capaz de utilizar o OpenMPI para computar esse percurso de maneira extremamente eficiente através da comunicação entre processos. Essa arquitetura é expansível para clusters, onde pode ser feito proveito do imenso poder de paralelismo oferecido por tais dispositivos, especialmente quanto considerado a escalabilidade horizontal.

Organização em alto nível do projeto.

O projeto foi escrito em C++. Foram desenvolvidas três soluções para efeito de comparação: A solução `2opt-mpi`, a solução `exaustive-bb` e a solução `exaustive-mpi`. As soluções com o prefixo exaustive utilizam da técnica de enumeração exaustiva, que embora pouco eficiente, retorna a solução ótima. A solução 2opt-mpi pode não retornar a solução ótima, mas muitas vezes se aproxima desta com qualidade e alto desempenho.

O OpenMPI é um protocolo de comunicação entre processos que também funciona na rede. Dessa forma, processos locais ou remotos podem abstrair as dificuldades de IPC (Interprocess communication) através de uma API simplificada, para resolverem problemas de maneira coordenada e distribuída.

Para otimizar o problema, assim como no Projeto 3, foi feito um pré-calculo das distâncias entre cada ponto em vetor N * N, no qual o indice de cada linha representa um ponto e a coluna outro. Isso evita que a distância entre os pontos seja computada em cada iteração. Esse cálculo é feito pelo processo zero, apelidado de máquina `MASTER`. Inclusive, essa keyword `MASTER` é um macro criado e está presente no código para indicar que o próximo bloco de código entre as macros `MASTER` e `END` só será executado pelo processo 0, ou raiz. O mesmo foi feito com a macro `WORKERS`.

Na solução `2opt-mpi`, o processo `MASTER` lê o arquivo de entrada, precomputa as distâncias entre os pontos lidos no vetor `distances`. Então ocorre um broadcast, passando para todos os outros processos a quantidade de pontos do problema (`N`) e as distâncias. Então, todos os processos criam um pseudo RNG (`random number generator`), usando como `seed` seu próprio `rank` e o tempo atual. Então, cada processo cria `SOLUTIONS` (100000) soluções aleatórias, e vai "desfazendo" os cruzamentos entre os pontos dessa solução, com o intuito de minimizar o custo. Cada processo então armazena seu melhor custo e caminho em um vetor, que são recolhidos com o método `gather` do MPI. A `MASTER` então, por fim, encontra a menor dessas soluções e as exibe no formato esperado no terminal, junto com seu respectivo custo.

As soluções `exaustive-bb` e `exaustive-mpi` são similares àquelas propostas no projeto 2, no entanto, ao invés de usar múltiplos processos na mesma máquina, com o MPI eles serão distribuídos em um cluster. A solução `exaustive-bb` é a com Branch and Bound, porém sem uso do OpenMPI. Escolhi o Branch and Bound como refência pois a solução inocente levaria muito tempo e seria desproporcional. A solução `exaustive-mpi` é bem similar, porém o primeiro nível das chamadas recursivas será distribuído em várias máquinas. Dessa forma, o problema pode ser dividido de maneira bastante justa entre todos os processos disponíveis, e deve ter seu tempo diminuído de maneira drástica.

As entradas desse problema representam um conjunto de *N* pontos e seguem o seguinte formato:

```
N
x_0 y_0
x_1 y_1
....
x_(N-1) y_(N-1)
```

Onde:

- `N` é o número de pontos do problema.
- Cada linha subsequente contém um ponto com as seguintes propriedades:
  - coordenada no eixo x `x_N`
  - coordenada no eixo y `y_N`

Quando lidos, as coordenadas de cada ponto são armazenados em um `point`, que nada mais é do que um `std::set` cuja chave é um `char 'x'` ou `'y'` e os valores são `doubles` lidos no arquivo. Isso é necessário pois o MPI/Boost por padrão só consegue serializar objetos da biblioteca padrão do C++ para serem distribuídos na rede.

O formato de saída será:

```
dist opt
0 e_1 ... e_(N-1)
```

Onde:

- `dist` é o comprimento do caminho encontrado usando 5 casas decimais.
- `e_1 ... e_(N-1)` é a sequência de empresas visitadas no trajeto
- `opt` é `1` se a solução encontrada é a ótima e `0` se foi usada outra estratégia

Benchmark

Foram criadas 3 versões, compiladas com o mpicxx para as seguintes versões do algoritmo:

- 2 Opt (MPI)
- Exaustive (MPI)
- Exaustive

Cluster

O cluster utilizado é um conjunto de 3 instâncias do EC2 da AWS. Uma das máquinas, chamada de A, é a mestra que irá invocar o `mpiexec`.

Para realizar esse benchmark, foi necessário configurar o ambiente dos clusters manualmente, permutando as chaves ssh entre as 3 instâncias. Além disso, foi configurado um servidos NFS para compartilhar os binários necessários entre as instâncias. A pasta montada esta no diretório `/home/mpi/shared`

- CPU: Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz Dual Core
- GPU: Não especificada
- RAM: 1GB
- SO: Ubuntu 18.04.3 LTS

In [53]:
```
KEY_PATH = "../keys/clustermpi.pem"
MASTER_PUBLIC_IP = 'ec2-3-17-190-206.us-east-2.compute.amazonaws.com'
```

In [80]:
```
# Importando dependências
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
import paramiko
```

In [81]:
```
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(MASTER_PUBLIC_IP, username='mpi', pkey=paramiko.RSAKey.from_privat
stdin, stdout, stderr = client.exec_command("echo hello from $USER!")
stdout.readline().strip()
```

Out[81]:
```
'hello from mpi!'
```

Descrição dos testes feitos

Os testes abaixo foram realizados com um pipe ssh na máquina remota. Todas as entradas foram geradas com o gerador do projeto 2 `gerador.py`. Como a solução que só usa CPU leva muito mais tempo, as entradas são menores (8, 12, 14 e 16 pontos).

Por questão de tempo, só foi realizada uma única execução de cada programa, o que pode ser alterado na constante `TESTS_PER_EXECUTABLE`.

O tempo em todas as versões é mensurado com a biblioteca `chrono`, usando o `high_resolution_clock`, que ao término da simulação é impresso na saída de erros *(stderr)*, para ser capturado pelo código de benchmark.

Todos os testes são armazenados em um DataFrame do Pandas para análise grafica.

In [82]:
```
df = pd.DataFrame()
```

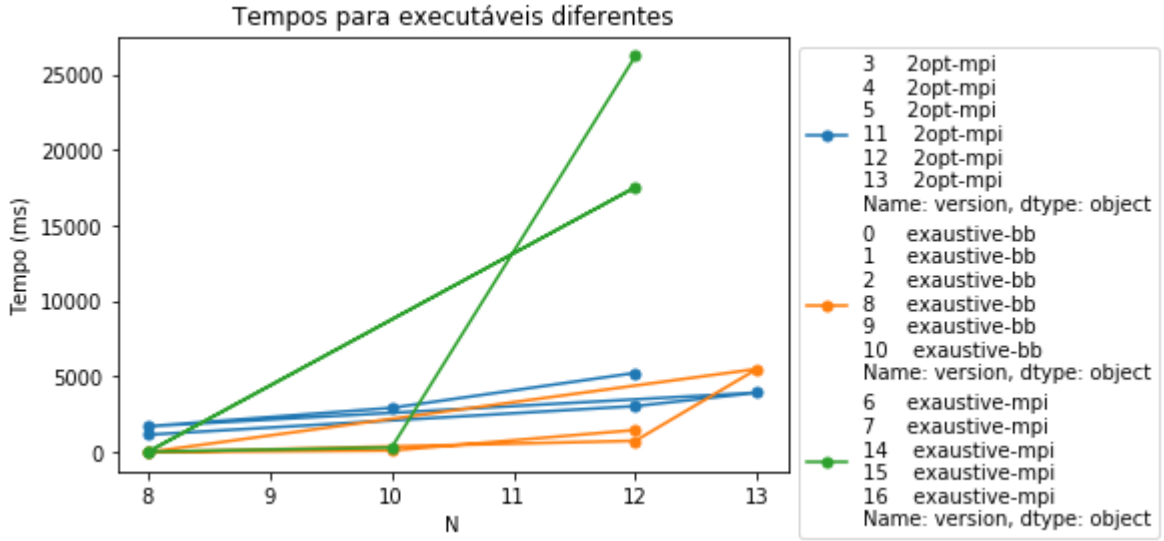
In [85]:
```
def cmd(exe, test, bb = False):
    supress_warning = "--mca btl_base_warn_component_unused 0"
    if bb:
        return f'/home/mpi/shared/{exe} < /home/mpi/shared/{test}'

    return f'mpiexec {supress_warning} -n 6 -hostfile /home/mpi/shared/hostfiles
```

In [86]:
```
TESTS_PER_EXECUTABLE = 1
INPUTS = ['in8.txt', 'in10.txt','in12.txt']

EXECUTABLES = ['exaustive-bb', '2opt-mpi', 'exaustive-mpi']

for exe in EXECUTABLES:
    for test in INPUTS:
        for i in range(TESTS_PER_EXECUTABLE):
            print('RUNNING:', exe, test)
            command = cmd(exe, test, exe == 'exaustive-bb')
            stdin, stdout, stderr = client.exec_command(command)
            time = float(stderr.readline().split()[0])
            cost = float(stdout.readline().split()[0])
            path = stdout.readline().strip()
            print(cost, time, path)

            df = df.append({
                'version': exe,
                'N': int(test.replace('in', '').replace('.txt', '')),
                'duration': time,
                'cost': cost,
                'path': path
            }, ignore_index=True)
```

```
RUNNING: exaustive-bb in8.txt
26312.11872 3.0 0 2 7 6 3 5 1 4
RUNNING: exaustive-bb in10.txt
7050.70882 109.0 0 6 7 1 3 5 8 9 4 2
RUNNING: exaustive-bb in12.txt
25658.30445 1466.0 0 10 11 4 1 6 9 3 8 2 7 5
RUNNING: 2opt-mpi in8.txt
26312.11872 1713.0 0 4 1 5 3 6 7 2
RUNNING: 2opt-mpi in10.txt
7050.70882 2934.0 0 6 7 1 3 5 8 9 4 2
RUNNING: 2opt-mpi in12.txt
25658.30445 5248.0 0 10 11 4 1 6 9 3 8 2 7 5
RUNNING: exaustive-mpi in8.txt
26312.11872 27.0 0 2 7 6 3 5 1 4
RUNNING: exaustive-mpi in10.txt
7050.70882 312.0 0 6 7 1 3 5 8 9 4 2
RUNNING: exaustive-mpi in12.txt
25658.30445 26225.0 0 10 11 4 1 6 9 3 8 2 7 5
```

Resultado e Conclusão

In [87]:
```
durations = df.groupby('version')
fig, ax = plt.subplots()
for name, group in durations:
    ax.plot(group.N, group.duration, marker='o', linestyle='-', ms=5, label=group

plt.title('Tempos para executáveis diferentes')
plt.ylabel('Tempo (ms)')
plt.xlabel('N')
plt.legend(loc='best', bbox_to_anchor=(1, 1))
plt.show()
```



In [96]:
```
df[df.version == 'exaustive-mpi'].mean().duration
```

Out[96]:
```
8824.8
```

In [93]:
```
df[df.version == '2opt-mpi'].mean().duration
```

Out[93]:
```
3006.8333333333335
```

In [95]:
```
df[df.version == 'exaustive-bb'].mean().duration
```

Out[95]:
```
1307.8333333333333
```

Dados os resultados, podemos supor que para entradas pequenas, o MPI não parece fazer muito sentido, em especial pelo overhead envolvido com a comunicação entre processos, porém, para uma solução mais elaborada com a 2opt distribuída, foi possível observar tempos muito melhores, em especial com um número N de pontos maior. Além disso, mesmo com o OpenMPI, a solução exaustiva ainda cresce seu tempo de maneira muito abrupta com o tamanho da entrada, então mesmo que seu caminho seja o ótimo, o custo temporal é muito grande.