relat_chen

October 2, 2019

1 Projeto 1 - Supercomputação

1.0.1 Lucas Chen Alba

1.1 Descrição do problema

Este projeto consiste em realizar simulações do conhecido problema do caxeiro-viajante (https://en.wikipedia.org/wiki/Travelling_salesman_problem) e analisar o ganho de velocidade com a utilização de técnicas de computação paralela e otimizações. A simulação consiste em encontrar o caminho ótimo ou sub-ótimo para um dado número de nós na rede do caxeiro. A primeira técnica para encontrar o caminho ótimo é a enumeração exaustiva, que consiste em testar todos os caminhos possíveis e elegir o melhor dentre eles.

1.2 Medições de tempo

Para as medições do tempo gasto nas simulações foi utilizada a biblioteca *chrono*. O tempo é medido e enviado para arquivos *.json*. Tendo os executáveis prontos, estes são executados 5 vezes neste arquivo *.ipynb* para entradas aleatórias (de 10 à 12 pontos no plano) geradas pelo arquivo *generator.py*. Assim calcula-se a média das diferentes configurações de otimização.

A simulação utiliza-se de duas configurações de *flags* de compilação distintas:
-O3 (apenas para o sequencial)
-O3 -fopenmp

1.3 Organização geral do código

1.3.1 Sequencial

O código possui 4 funções não incluindo a *main()*. Segue a explicação básica de cada função:

- **double dist(std::vector p1, std::vector p2)**: Recebe dois vetores com a posição de dois pontos e calcula a distância entre eles.
- **double path_dist(std::vector sol, std::vector<std::vector> points)**: Recebe dois vetores, o primeiro com uma sequência de pontos e o segundo com as posições dos pontos e calcula a distância total para percorrer todos os pontos no primeiro vetor.
- **double backtrack(std::vector<std::vector> points, int idx, double curr_cost, double best_cost, std::vector curr_sol, std::vector used, std::vector &best_sol)**: Recebe todos os parâmetros necessários para realizar a enumeração exaustiva. Realiza chamadas recursivas para percorrer caminhos a partir de um nó, assim, esse nó "inicial" é trocado quando o

código passa para a próxima iteração do loop de chamadas recursivas, assim percorrendo em cascata todos os caminhos trocando-se o "nó inicial".

Quando o código é executado, primeiramente são coletados todos os inputs da saída do arquivo *generator.py* passados como *cin*, assim, todos os pontos são gerados e gravados em um vetor de vetores(x, y).

Após a geração dos pontos, é chamada a função *backtrack()*, no final de sua execução, o vetor *best_sol* automaticamente contém a melhor sequência.

1.3.2 Paralela

Para a versão paralela do código, continua-se executando a função *backtrack()* múltiplas vezes, porém agora dividimos o trabalho entre *tasks* criadas na função *main()*, a diferença entre elas será que cada task irá iniciar suas chamadas recursivas a partir de um nó diferente, podendo realizar a enumeração exaustiva em paralelo.

1.3.3 Local Search

Para a otimização discreta "Local Search" foram implementadas duas novas funções: **void local_search(std::vector<std::vector> points, double &best_cost, std::vector<std::vector> &best_sol)**: Esta função irá receber um caminho aleatório e irá otimizá-lo até chegar em uma solução sub-ótima, significando um mínimo local. Ela faz isso a partir da checagem de "cruzamentos" entre dois pares de pontos quaisquer do caminho, ela então realiza um "swap" dos pontos, assim, quando todos os cruzamentos forem resolvidos, este será o mínimo local. **bool check_intersec(std::vector p1, std::vector p2, std::vector q1, std::vector q2)**: Esta função recebe uma sequência de 4 pontos e checa se há alguma interseção entre os segmentos de reta formados pelos dois pares de pontos.

Na função *main()* serão gerados N caminhos aleatórios, caminhos estes que serão passados para chamadas da funçãoeste número define a quantidade de caminhos iniciais aleatórios seráo otimizados pelo Local Search

1.3.4 Branch and Bound

Este código é muito parecido com o código de enumeração exaustiva em paralelo, porém o seu conceito é muito mais perspicaz. Sempre que uma nova chamada da função *backtrack()* é feita, é realizada a comparação do custo do caminho até o momento, caso o caminho seja maior, a execução da chamada atual é cancelada.

```
In [223]: import os
          import json
          import pandas as pd
          import matplotlib.pyplot as plt
          import statistics
          import subprocess
          import math
```

```
In [179]: # Build simulations

          subprocess.run(['cmake', '..'], cwd='../build')
          subprocess.run(['make'], cwd='../build')
```

```
Out[179]: CompletedProcess(args=['make'], returncode=0)
```

```
In [180]: # Build standard inputs with generator.py
          for i in range(2, 15):
              inp = subprocess.run(['python3', 'generator.py'], input=bytearray(str(i), 'utf-8
              inp = inp.stdout.decode()
              with open('../inputs/in'+str(i), 'w') as file:
                  file.write(inp)

          # Build standard outputs with tsp.py
          for i in range(2, 5):
              out = subprocess.run(['python3', 'tsp.py'], input=bytearray(open('../inputs/in'+s
              out = out.stdout.decode()
              with open('../outputs/out'+str(i), 'w') as file:
                  file.write(out)
```

```
In [181]: # Validate answers from the simulations
          simulations = ['seq-opt', 'par1-opt', 'locsea-opt', 'bb-opt']
          # 'locsea-bb-opt'

          flag = True
          for _ in range(10):
              for size in range(2, 5):
                  for sim in simulations:
                      process = subprocess.run(['./'+sim], input=bytearray(open('../inputs/in'+
                      process = process.stdout.decode()
                      out = open('../outputs/out'+str(size)).read()
                      # Check total distance and sequence
                      if (process != out and sim in ['seq']):
                          print(sim+': wrong')
                          print('real:', repr(process))
                          print('ideal:', repr(out))
                          flag = not flag
                      # Test total distance, but not sequence due to paralellism
                      else:
                          process = process.split('\n')[0].split(' ')[0]
                          out = out.split('\n')[0].split(' ')[0]
                          if (process != out):
                              print(sim+': wrong')
                              print('real:', repr(process))
                              print('ideal:', repr(out))
                              flag = False
          if (flag):
              print('All tests are ok')
```

```
All tests are ok
```

```
In [214]: # Run simulations and get mean times
```

```
        # ['seq-opt', 'par1-opt', 'locsea-opt', 'bb-opt']
        times = {}
        simulations = ['seq-opt', 'par1-opt', 'locsea-opt', 'bb-opt']

        for sim in simulations:
            times[sim] = []

        for sim in simulations:
            for size in range(10, 13):
                temp = []
                for _ in range(5):
                    process = subprocess.run(['./'+sim], input=bytearray(open('../inputs/in'+
                    process = process.stderr.decode()
                    temp.append(float(process.split('\n')[0]))
                times[sim].append(sum(temp) / float(len(temp)))
```

In [228]:
```
mean_times = [[], [], []]
for mean in times.values():
    mean_times[0].append(mean[0])
    mean_times[1].append(mean[1])
    mean_times[2].append(mean[2])
print(mean_times)
```

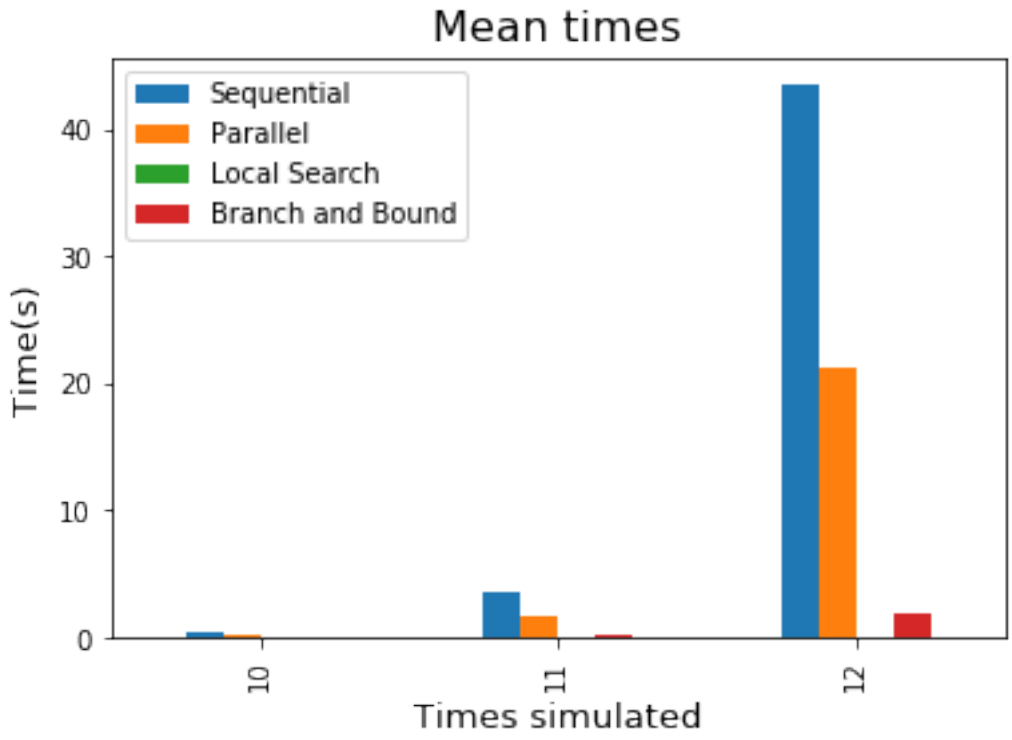
```
[[0.34347380000000005, 0.1478142, 0.004195659999999999, 0.04028274], [3.653792, 1.702168, 0.003
```

In [229]:
```
# Mean times
groups = mean_times
group_labels = ['10', '11', '12']

# Convert data to pandas DataFrame.
df = pd.DataFrame(groups, index=group_labels).T

# Plot.
pd.concat(
    [df.loc[0].rename('Sequential'),
     df.loc[1].rename('Parallel'),
     df.loc[2].rename('Local Search'),
     df.loc[3].rename('Branch and Bound')],
    axis=1).plot.bar()
plt.xlabel('Times simulated', fontsize=13)
plt.ylabel('Time(s)', fontsize=13)
plt.title('Mean times', fontsize=16)
```

Out[229]:
```
Text(0.5, 1.0, 'Mean times')
```



1.4 Resultados e análises

Os ganhos nas otimizações discretas são bastante visíveis, sendo o mais rápido de todos, o Local Search, isso se da por ele não depender do número de nós da entrada, apenas do número de iterações iniciais (número de caminhos aleatórios iniciais), isto pode ser bom no caso de poucos nós, onde a variação de caminhos não é alta, porém em casos maiores, o caminho sub-ótimo encontrado pode ser bastante impreciso. O segundo é o Branch and Bound, este é bem mais rápido do que o paralelo pois ele realiza basicamente os mesmos trajetos do que a enumeração exaustiva porém com condições de parada, não precisando gastar tempo em caminho que não serão melhores que o melhor até o momento.

Não foram implementadas melhorias no sistema de comparação do Branch and Bound, que poderiam potencialmente reduzir os tempos de execução.

```
In [ ]:
```