Insper

Super Computação

Igor Montagner, Luciano Soares

2019/2

Projeto 3 - Multi-core

Neste projeto continuaremos trabalhando com o problema do Caixeiro Viajante:

> Um vendedor possui uma lista de empresas que ele deverá visitar em um certo dia. Não existe uma ordem fixa: desde que todos sejam visitados seu objetivo do dia está cumprido. Interessado em passar o maior tempo possível nos clientes ele precisa encontrar a sequência de visitas que resulta no menor caminho.

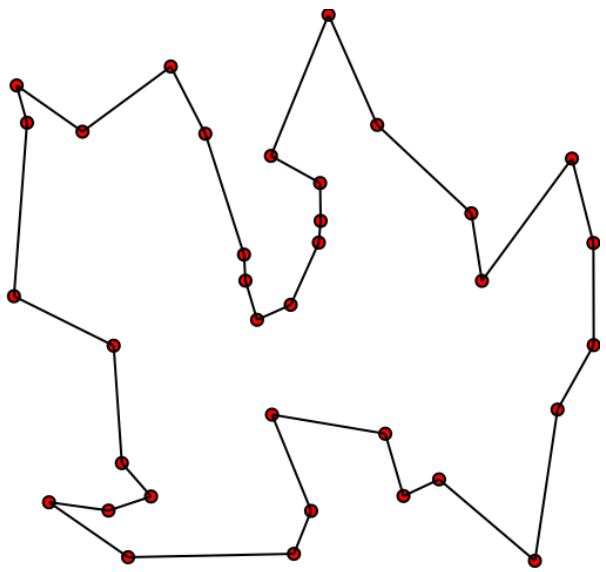

Figure 1: Cada ponto vermelho é uma empresa a ser visitada e a linha preta é o melhor trajeto

No projeto 2 trabalhamos com implementações em multi-core e conseguimos desempenho muito superior a implementação ingênua em Python ao escrever o programa em C++ e paralelizá-lo. Neste projeto iremos conseguir ganhos ainda maiores ao trabalhar com GPU.

Parte 1 - implementacão

Iremos trabalhar neste projeto com os conceitos de otimização discreta vistos na rubrica **B** do projeto anterior. Isto é necessário pois **o caixeiro viajante não é um problema que tem implementação inocente em GPU**. Ou seja, se quisermos obter ganhos precisaremos entender melhor o problema que estamos trabalhando e tentar encontrar em quais pontos ele pode se valer do tipo de paralelismo existente em GPUs.

Você poderá usar as tecnologias vistas em sala de aula e outras que estejam disponíveis publicamente **desde que elas não resolvam diretamente o problema**. Se estiver em dúvida consulte o professor.

Métodos usados

A ideia deste projeto é implementar métodos incrementalmente mais sofisticados de resolução do problema, de maneira que quanto maior sua nota mais rápidos serão (ou deveriam ser) os programas. Em especial, gostaríamos

de chegar cada vez mais perto de encontrar a solução ótima do problema. Para facilitar os testes, iremos gerar um executável diferente para cada método.

Conceito D

Neste conceito iniciaremos com a solução mais simples possível: sortearemos 100.000 soluções aleatoriamente e pegaremos a que tive menor custo. Esta solução é simplória, mas é extremamente portável para GPU: todos os roteios são independentes e terei uma solução boa simplesmente pelo fato de ter testado muitas coisas.

Seu executável deverá se chamar **random-sol** e funcionar exatamente igual ao programa do projeto anterior, mas desta vez ele colocará `0` no campo `opt` da saída.

É obrigatório neste conceito pré-calcular todas as distâncias entre pontos antes de iniciar os sorteios.

Conceito C

A partir do programa acima, implemete a heurística 2-opt descrita no curso de Otimização Discreta. A qualidade da sua solução será sempre menor ou igual a anterior (a probabilidade de ser exatamente igual é desprezível).

Seu executável deverá se chamar **2opt-sol** e funcionar exatamente igual ao programa do projeto anterior, mas desta vez ele colocará `0` no campo `opt` da saída.

Conceito C+

Melhore seu programa usando técnicas de balanceamento de carga entre threads e de otimização de memória. Se quiser, também pode explorar o uso de operações atômicas para compartilhamento de dados dentro de um bloco e entre blocos. O resultado deverá ser o mesmo da seção anterior, mas o tempo de execução deverá ser menor.

Você não precisa, necessariamente, criar um novo executável neste conceito. Se seu **2opt-sol** tiver as otimizações acima ele já será considerado para este conceito. Porém, se quiser criar duas etapas isto tornará seu relatório mais completo.

A partir deste ponto você pode escolher dois caminhos. Cada um vale até 2 pontos se implementado corretamente. Fazer ambos implica em **A+** neste projeto.

Caminho 1 - branch and bound

Agora que já temos um programa que devolve soluções (provavelmente muito boas), vamos tentar encontrar a solução ótima. Seu objetivo neste conceito será reimplementar o algoritmo de **enumeração exaustiva** em GPU.

Seu executável deverá se chamar **best-sol** e funcionar exatamente igual ao programa do projeto anterior, mas desta vez ele colocará `1` no campo `opt` da saída.

Além do programa acima vamos também criar uma versão *Branch-and-Bound* em GPU. Segundo o que vimos nas aulas, é impossível fazer sincronização entre blocos. Ou seja, teremos que pensar como guardar a(s) solução(ões) ótima(s) para consultar e efetivamente cortar ramos de recursão de maneira eficiente.

Seu executável deverá se chamar **best-sol-bb** e funcionar exatamente igual ao programa do projeto anterior, mas desta vez ele colocará `1` no campo `opt` da saída.

Caminho 2 - cooperação CPU-GPU

Neste conceito você irá fazer um programa que aproveita 100% dos recursos de uma máquina ao trabalhar com CPU e GPU **ao mesmo tempo**. Seu programa deverá rodar buscas locais com **2opt** na GPU e um algoritmo eficiente de *Branch-and-Bound* em paralelo na CPU. Ao encontrar soluções boas na GPU elas devem ser "enviadas" para cortar mais ramos de recursão do Branch-and-Bound na CPU.

Importante: incluir `if (curr_sol >= best_sol) return best_sol` em seu programa não conta como *Branch-and-Bound* neste item. Você precisa pensar um pouco em como encontrar um limitante inferior para sua solução que seja (muito) mais justo que isto.

Requisitos de projeto

Se os requisitos de projeto abaixo não forem cumpridos sua nota máxima será **D**.

- [ ] CMakeLists.txt que gere um executável por método testado
- [ ] Relatório feito em Jupyter Notebook ou PWeave. Seu relatório deve conter as seguintes seções:
  - [ ] Descrição do problema tratado
  - [ ] Descrição dos testes feitos (tamanho de entradas, quantas execuções são feitas, como mediu tempo)
  - [ ] Organização em alto nível de seu projeto.
  - [ ] Comparação com projeto em CPU multi-core
- [ ] Versão já rodada do relatório exportada para *PDF*
- [ ] README.txt explicando como rodar seus testes
- [ ] Conjunto de testes automatizados (via script Python ou direto no relatório)
- [ ] Respeitar os formatos de entrada e saída definidos na seção anterior
- [ ] Seu programa deverá retornar sempre o mesmo resultado.

Como usamos máquinas remotas, não é necessário que seu relatório execute os testes propriamente ditos. Você pode entrar um script Python ou Bash que executa os programas e salva os resultados em um conjunto de arquivos que seu relatório lê e analisa.

Além disto, como estamos começando nosso programa com a busca local, a escala de tamanho de testes deverá aumentar. Veremos que conseguimos soluções muito rapidamente mesmo para os problemas grandes (`N=300`) do TSPlib. Levem isto em conta nos testes feitos.

Parte 2 - Relatório e testes

O relatório seguira uma rubrica contendo diversos itens. A nota final de relatório é a média das notas parciais, levando em conta a seguinte atribuição de pontos.

- **I** - 0 pontos: Não fez ou fez algo totalmente incorreto.
- **D** - 4 pontos: Fez o mínimo, mas com diversos pontos para melhora.
- **B** - 7 pontos: Fez o esperado. Não está fantástico, mas tem qualidade.
- **A+** - 10 pontos: Apresentou alguma inovação ou evolução significativa em relação ao esperado.

Rubrica para SuperComputação – Projetos

Critério	Conceito			
	I	D	B	A+
Descrição do problema e da realização dos testes.	O relatório descreve o problema de maneira errada ou que induz ao erro. A descrição da implementação não bate com o código entregue.	O problema é descrito de maneira sucinta e clara. A descrição da implementação omite dados importantes da execução dos testes (máquina usada, arquivos de entrada, etc)	O problema tratado é descrito de maneira sucinta e clara. Detalhes da execução dos testes são apresentados e a escolha dos testes é justificada.	Além do item anterior, é feita alguma tentativa de relacionar características da máquina usada nos testes com os resultados colhidos.
Tamanho das instâncias de teste e quantidade de cenários testados	Não apresenta o resultado de nenhum teste no relatório ou apresenta apenas de casos triviais.	Realiza um conjunto pequeno de testes mas consegue demonstrar alguma diferença significativa de desempenho.	Realiza um conjuto abrangente de testes, levando em conta o consumo de recursos do sistema.	Leva a capacidade de sua máquina próxima de seu limite, realizando também testes intermediários de desempenho.
Medidas de consumo de recursos	Não mede nenhuma medida de desempenho de maneira correta ou consistente.	São feitas medições de tempo imprecisas (comando time) ou misturando partes diferentes do programa.	É medido consumo de tempo com maior precisão (por exemplo, std::chrono).	São feitas várias medições de tempo da mesma tarefa e é os resultados são apresentados como média e desvio padrão dos tempos.
Facilidade de leitura do relatório	O relatório contém erros grosseiros de escrita, não segue nenhuma estrutura clara de organização nem respeita a norma culta de escrita.	Os resultados estão explicados diretamente no texto de maneira confusa. Se utiliza tabelas ou gráficos, o faz incorretamente.	Ilustra as diferenças de desempenho com gráficos e tabelas e comenta brevemente seus resultados.	Ilustra as diferenças de desempenho com tabelas e gráficos e os interpreta de maneira detalhada no texto.