Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

SSC0143 – Programação Concorrente

Trabalho 2 – Hospital em ICMCTown

Paralelização com OpenMP

Elias Italiano Rodrigues – 7987251
Rodolfo Megiato de Lima – 7987286
Vinicius Katata Biondo – 6783972

São Carlos, 17 de outubro de 2014

Repositório: `https://code.google.com/p/pc2014-02-grupo2-turmab/`

Sumário

1 Introdução **2**

1.1 Hospital em ICMCTown? 2

1.2 Os Algoritmos 2

1.2.1 Floyd-Warshall 2

1.2.2 Dijkstra 2

2 Desenvolvimento **3**

2.1 Floyd-Warshall – Sequencial 3

2.2 Floyd-Warshall – Paralelo 3

2.3 Dijkstra – Sequencial 3

2.4 Dijkstra – Paralelo 4

2.5 Cálculo das Estatísticas 4

2.6 Compilação e Execução dos Programas 4

3 Resultados **5**

3.1 Floyd-Warshall 6

3.2 Dijkstra 7

4 Conclusão **8**

Referências **9**

1 Introdução

1.1 Hospital em ICMCTown?

Este trabalho implementa algoritmos para encontrar uma melhor localização para a instalação de um hospital na fictícia cidade de ICMCTown, assim como foi sugerido no enunciado do trabalho 2 da disciplina. A modelagem da cidade é feita em um grafo, em que os vértices representam as junções (esquinas) e as arestas representam as ruas que ligam as junções.

Os algoritmos implementados foram o de Floyd-Warshall [1] e o de Dijkstra [2]. Além da versão sequencial, também foi implementada uma versão paralela de cada um dos algoritmos utilizando recursos oferecidos pela OpenMP [3].

Este relatório faz parte de um conjunto de arquivos referentes ao trabalho. A documentação do código-fonte encontra-se nos arquivos `.c` dentro de `./src` a partir do diretório raiz.

1.2 Os Algoritmos

1.2.1 Floyd-Warshall

O algoritmo de Floyd-Warshall é uma análise feita em cima de um grafo valorado e direcionado para determinar menores caminhos. Em uma única execução, esse algoritmo encontra o menor caminho entre todos os pares de vértices do grafo. O caminho é determinado somente pelo seu custo – soma dos pesos de todas as arestas presentes em seu percurso.

Para resolver o problema da instalação de um hospital em ICMCTown, é requerido que a localização seja de modo que a maior distância entre uma casa da cidade e o hospital seja a mínima possível. A execução do Floyd-Warshall resulta na distância de cada esquina da cidade para cada outra esquina, sendo assim, seleciona-se como resultado a distância da esquina que possui a mínima distância entre as maiores dessas distâncias.

1.2.2 Dijkstra

O algoritmo de Dijkstra é utilizado em problemas em que se deseja encontrar caminhos mínimos em grafos.

Como o problema da instalação de um hospital em ICMCTown foi modelado como um grafo, tem-se a melhor localização desse hospital quando a sua distância entre o vértice mais longínquo for a menor possível.

Dessa forma, o algoritmo de Dijkstra, executado para cada um dos vértices do grafo, encontra a distância de cada junção para todas as outras. Tendo esses dados avaliados, uma solução do problema torna-se imediata selecionando-se como resultado a distância menor entre as maiores dessas distâncias.

2 Desenvolvimento

O programa foi desenvolvido em linguagem `C` em sistema operacional `GNU/Linux`. Além das bibliotecas convencionais `stdlib.h` e `stdio.h`, foi utilizada também a `omp.h` do OpenMP.

2.1 Floyd-Warshall – Sequencial

A implementação do código sequencial do algoritmo Floyd-Warshall foi uma tradução direta do pseudo-código para o programa em `C`. Para a execução desse algoritmo, o grafo foi modelado como uma matriz de adjacências $n \times n$.

Optou-se por desprezar o vetor de nós predecessores que não foi requerido na resolução do problema, já que a saída é apenas a distância em que o hospital estaria da casa mais longínqua.

Como a saída do algoritmo é uma tabela de distância entre todos os vértices, foi necessário buscar nesse vetor a menor distância entre os maiores caminhos de cada um das junções.

2.2 Floyd-Warshall – Paralelo

A paralelização do algoritmo Floyd-Warshall, utilizando OpenMP, foi implementada dividindo-se o primeiro laço de repetição `for` executando partes dele em cada uma das *threads* inicializadas. Para fazer isso, utilizou-se a diretiva OpenMP:

```
#pragma omp parallel for private(k, i, j)
```

Com isso, o `for` foi dividido pelo número de *threads*, mantendo as variáveis de iteração i, j e k como *private* para **tentar** garantir a integridade dos dados. Porém, ao serem efetuados testes com a versão paralela, constatou-se que em uma pequena parcela das execuções, o resultado retornado era diferente do esperado. Devido à alta dependência entre os índices dos laços, a paralelização utilizando as diretivas OpenMP não consegue garantir o resultado determinístico do algoritmo.

Além disso, durante a programação foi encontrado um outro problema. Antes foi utilizado o comando `continue` da linguagem `C` dentro dos laços do algoritmo para ignorar os vértices que não tem arestas entre si. A utilização desse comando ocasionava também erros no resultado do programa, que foi averiguado após vários testes. Esse problema com os comandos `continue` e `break` na paralelização de laços com OpenMP foi discutido no fórum dos desenvolvedores. [4]

2.3 Dijkstra – Sequencial

Para Dijkstra, o grafo teve que ser transformado em uma implementação de lista de adjacências. Essa abstração de um grafo utilizando lista de adjacências, contém uma lista de arestas que representa todos os vértices ao qual um determinado vértice está conectado. Além disso, foi necessário o desenvolvimento de uma lista para salvar os custos para cada vértice e assim escolher qual será o próximo vértice a ser visitado.

Nesse caso, também não foi necessário armazenar o vetor de vértices predecessores, pois não é requerido para a resolução do problema.

2.4 Dijkstra – Paralelo

A paralelização do algoritmo de Dijkstra, utilizando OpenMP, foi implementada dividindo-se o laço de repetição `for` que realiza a chamada da função do Dijkstra. Dessa forma, cada uma das *threads* inicializadas é responsável pela chamada de uma quantidade de vezes do Dijkstra, um para cada vértice que ficou designada. Para realizar essa operação, foi utilizada a seguinte diretiva:

```
#pragma omp parallel for
```

2.5 Cálculo das Estatísticas

Além dos algoritmos para a resolução do problema, foi implementado um outro programa para auxiliar o cálculo das média, desvios padrão e intervalo de confiança dos tempos das execuções. Esse programa `./bin/stats` recebe como entrada o número de execuções seguido dos tempos de cada execução de um dos algoritmos. O arquivo dos tempos de execução é gerado através de um *shell script* `./run.sh` que roda os programas uma quantidade informada de vezes.

2.6 Compilação e Execução dos Programas

Estando no diretório raiz dos arquivos deste trabalho, é possível compilar e executar os programas seguindo as instruções:

- Valores de entrada:
  ```
  #algoritmo : 0 para Dijkstra e 1 para Floyd-Warshall
  #nthreads  : quantidade de threads (só tem efeito para versão paralela)
  #arquivo   : nome do arquivo de entrada
  #vezes     : número de vezes que os programas serão executados
  ```
- Para compilar todos programas, execute:
  ```
  make
  ```
- Para rodar a versão sequencial, execute:
  ```
  ./bin/sequential #algoritmo < #arquivo
  ```
  Por exemplo:
  ```
  ./bin/sequential 0 < input01
  ```
- Para rodar a versão paralela, execute:
  ```
  ./bin/parallel #algoritmo #nthreads < #arquivo
  ```
  Por exemplo:
  ```
  ./bin/parallel 0 4 < input01
  ```
- Para limpar todos os arquivos compilados, execute:
  ```
  make clean
  ```
- Para rodar vários testes, paralelo e sequencial, execute:
  ```
  ./run.sh #algoritmo #nthreads #arquivo #vezes
  ```
  Por exemplo:
  ```
  ./run.sh 0 4 input01 10
  ```

- Para conferir por vazamento de memória, execute:
  `make memcheck ALG=#algoritmo`

Caso não consiga compilar e rodar os programas, confira por dependências da OpenMP, assim como dos programas usados `make`, `gcc` e `valgrind` no seu sistema operacional.

3 Resultados

Os programas foram compilados e executados de modo automatizado por *shell scripts* no computador de um dos integrantes do grupo do trabalho. Seguem as especificações:

```
Phoronix Test Suite v5.2.1
System Information

Hardware:
Processor: Intel Core i7-3612QM @ 3.10GHz (8 Cores),
Motherboard: Dell 0DNMM8, Chipset: Intel 3rd Gen Core DRAM, Memory: 8192MB,
Graphics: Intel HD 4000 2048MB (1100MHz)

Software:
OS: Fedora 20, Kernel: 3.15.10-201.fc20.x86_64 (x86_64),
Compiler: GCC 4.8.3 20140624, File-System: ext4
```

Os resultados de tempos a seguir são provenientes de 10 execuções consecutivas dos programas no computador citado acima. Foi considerado para os testes o `input08` que acompanha a especificação do trabalho. Os arquivos completos dos resultados aqui apresentados podem ser consultados no diretório `./test` deste trabalho.

3.1 Floyd-Warshall

Comparando os resultados obtidos nas execuções do programa Floyd-Warshall, pode-se constatar que a paralelização foi eficiente e as execuções ficaram cada vez mais rápidas de acordo com a quantidade de *threads*, alcançando um *speed up* de até 2.60. Percebe-se também que o *speed up* tende a se estabilizar conforme a quantidade de *threads* aumenta, pois o sistema operacional gasta cada mais tempo com o gerenciamento das *threads*.

Analisando-se o algoritmo sequencial, é possível notar que sua complexidade é $O(n^3)$, pois o maior processamento encontra-se nos 3 laços aninhados que iteram com relação à quantidade total n de vértices.

Threads	FWS	FWP	Stddev	ConfInt	*Speedup*
1	6.827	—	0.040012	[6.802200, 6.851800]	—
2	—	5.647	0.585219	[5.284278, 6.009722]	1.21
4	—	4.591	0.609778	[4.213056, 4.968944]	1.49
8	—	3.289	0.448206	[3.011199, 3.566801]	2.08
16	—	2.727	0.372345	[2.496218, 2.957782]	2.50
32	—	2.628	0.310895	[2.435305, 2.820695]	2.60

Tabela 1: Médias dos tempos (s) de 10 execuções do algoritmo Floyd-Warshall e *speed-up*

FWS: Floyd-Warshall Sequencial / FWP: Floyd-Warshall Paralelo
Stddev: Desvio padrão / ConfInt: Intervalo de Confiança

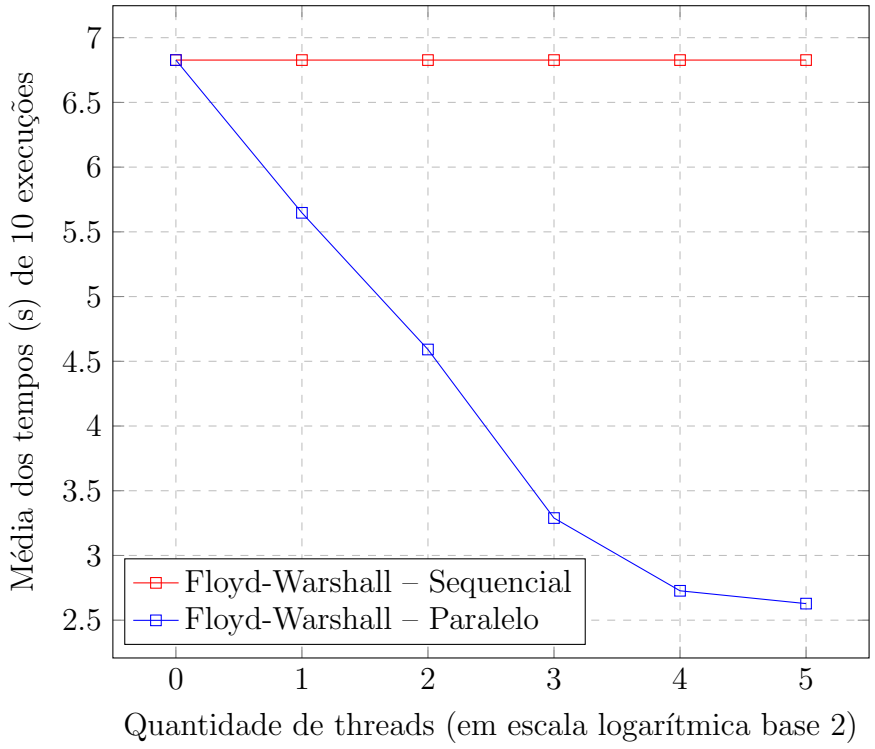

Figura 1: Comparativo das médias de execução do Floyd-Warshall sequencial e paralelo

3.2 Dijkstra

As estruturas de dados utilizadas na implementação do Dijkstra (listas de adjacências), implicaram em uma performace inferior em comparação à implementação com Floyd-Warshall (matriz de adjacências) apesar de possuirem uma mesma complexidade. Porém com relação ao *speed up* da paralelização, obteve-se uma maior aceleração nos tempos de execução do Dijkstra como pode ser observado nos resultados. Percebe-se também uma tendência a piorar o *speed up* com um número de *threads* muito grande.

Threads	DS	DP	Stddev	ConfInt	*Speedup*
1	38.359	—	1.050176	[37.708094, 39.009906]	—
2	—	20.101	0.578921	[19.742181, 20.459819]	1.91
4	—	12.072	0.072774	[12.026894, 12.117106]	3.18
8	—	8.462	0.127891	[8.382733, 8.541267]	4.53
16	—	8.667	0.515811	[8.347297, 8.986703]	4.43
32	—	9.255	0.503632	[8.942846, 9.567154]	4.14

Tabela 2: Médias dos tempos (s) de 10 execuções do algoritmo Dijkstra e *speed-up*

DS: Dijkstra Sequencial / DP: Dijkstra Paralelo
Stddev: Desvio padrão / ConfInt: Intervalo de Confiança

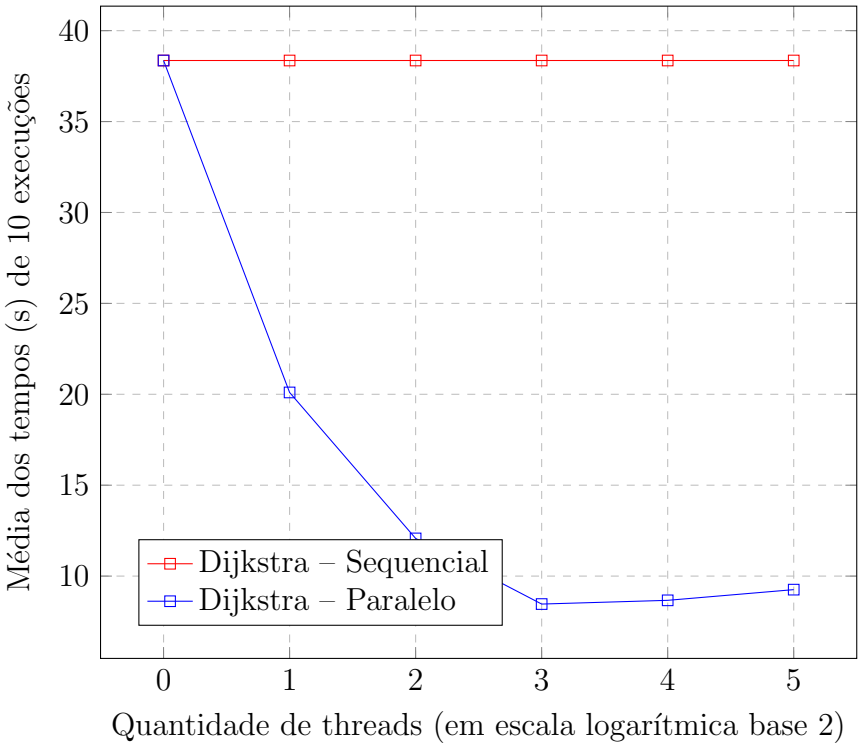

Figura 2: Comparativo das médias de execução do Dijkstra sequencial e paralelo

4 Conclusão

Na a implementação desse trabalho, não houve desafios evidentes quanto ao encontrar algoritmos para a solução do problema, uma vez que tais algoritmos são amplamente conhecidos. Uma parte da dificuldade dá-se à modelagem do problema como um grafo para então aplicar o algoritmo e buscar a menor distância de um hospital e uma esquina na fictícia cidade de ICMCTown. Além disso, a implementação dos algoritmos deveriam ser possíveis de paralelizar e produzir a solução de forma mais eficiente.

Então, com a utilização dos dois algoritmos – Floyd-Warshall e Dijkstra – a paralelização ficou trivial de se fazer, pois utilizando os recursos da OpenMP, apenas foi necessário acrescentar uma diretiva de paralelização para cada algoritmo.

Mas é importante notar também, que nem sempre é possível conseguir uma paralelização total que garanta a integridade dos dados quando estes são muito dependentes, como foi visto no caso do algoritmo Floyd-Warshall no tópico 2.2. Nessas situações, uma análise detalhada do algoritmo é necessária e talvez o uso de `pthread` para se ter controle mais preciso do comportamento do paralelismo ao invés usar recursos mais abstratos como as diretivas do OpenMP.

Referências

[1] Floyd-Warshall algorithm
Disponível em: <http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm>
Acesso em: 17 de outubro de 2014.

[2] Dijkstra's algorithm
Disponível em: <http://en.wikipedia.org/wiki/Dijkstra's_algorithm>
Acesso em: 17 de outubro de 2014.

[3] OpenMP.org
Disponível em: <http://openmp.org/>
Acesso em: 17 de outubro de 2014.

[4] OpenMP Forum - loops with break or continue statements
Disponível em: <http://openmp.org/forum/viewtopic.php?f=3&t=458>
Acesso em: 17 de outubro de 2014.