Análise do problema do caixeiro viajante sobre diferentes ambientes de programação concorrente

Elvio Viçosa Junior, Jeronimo M. Medina, Rafael Pereira, Gerson Geraldo H. Cavalheiro

Departamento de Informática
Universidade Federal de Pelotas (UFPel)
Pelotas – RS – Brasil

{evicosa_ifm, jmadruga.ifm, rpereira.ifm, gerson.cavalheiro}@ufpel.edu.br

Resumo. *O problema do caixeiro viajante é uma aplicação conhecida por possuir alto custo computacional. Neste trabalho foi realizado um estudo do desempenho desta aplicação em arquiteturas dual-core utilizando diferentes ambientes de execução: Pthreads, Cilk e Athreads. Os resultados de desempenho indicam ganho de desempenho em todas as versões concorrentes.*

1. Introdução

A popularização dos processadores *multi-core* no mercado aumentou a demanda por programação concorrente. Em linhas gerais, o objetivo é utilizar recursos de ferramentas de programação concorrente para implementação de aplicações sobre tais arquiteturas com vistas a obter melhores índices de desempenho. A academia e o mercado apresentam diversas opções de ferramentas. Este trabalho analísa a implementação de uma aplicação que apresenta alto custo computacional em três destas ferramentas: Pthreads [Dre05], Cilk [Joe96] e Athreads [Cav06].

O estudo de caso é aplicado ao algoritmo do caixeiro viajante, cujo custo computacional está associado ao número de cidades que um caixeiro viajante necessita visitar. Os experimentos foram conduzidos em arquiteturas *dual-core* para avaliar o desempenho da implementação considerando as ferramentas selecionadas.

O restante deste trabalho encontra-se organizado como segue. O problema do caixeiro viajante é detalhado na Seção 2. Os ambientes de programação concorrente são apresentados na Seção 3. A estratégia que foi adotada para obter a concorrência no algoritmo do caixeiro viajante, juntamente com os dados obtidos através das execuções e a análise dos resultados, são apresentados na Seção 4. A Seção 5 conclui o trabalho.

2. O problema do caixeiro viajante

O caixeiro viajante é um problema clássico no conjunto dos problemas que possuem uma carga computacional elevada. A formulação do problema é a seguinte: suponha que um caixeiro viajante deseja visitar N cidades de uma certa localização e que, entre alguns pares de cidades existem rotas, permitindo a viagem de uma cidade para outra. Cada rota tem um número associado, que pode representar a distância ou o custo necessário para percorrê-la. Assim, o caixeiro viajante deseja encontrar um caminho que passe por cada uma das N cidades apenas uma vez, e além disso que tenha o menor custo possível; onde o custo do caminho é a soma dos custos das rotas percorridas [Mol04].

Tabela 1. Crescimento no número de caminhos para avaliação

Quantidade de cidades	Números de caminhos
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200

Este trabalho implementou o algoritmo do caixeiro viajante com a avaliação do problema utilizando a abordagem ”força-bruta”. Esta avaliação consiste na verificação de todos os caminhos existentes, para obter o menor caminho. A Tabela 1 mostra o crescimento de caminhos possíveis, entre 5 e 14 cidades, verificando um grande aumento de caminhos mesmo com um baixo número de cidades. Como o objetivo do algoritmo é encontrar o caminho de menor custo, sua complexidade está relacionada com o número de cidades a serem visitadas, e analisando o crescimento do número de caminhos possíveis de acordo com o crescimento de número de cidades, nota-se que a computação desse problema mesmo com poucas cidades pode ser uma tarefa difícil e de longa duração.

O desempenho computacional não é o único motivo do estudo do problema do caixeiro viajante. Inúmeros problemas reais são modelados a partir deste problema ou suas variantes. Um exemplo disto é o processo de entrega de produtos por um fornecedor em diferentes super-mercados. A obtenção de uma rota ótima para entrega dos produtos, resulta em um menor tempo de entrega e em um menor gasto com combustível.

3. Ambientes de programação

A escolha dos ambientes de programação foi baseada na idéia de utilizar ferramentas que oferecem uma camada de abstração *multithread* para descrição da concorrência da aplicação. Destas ferramentas, Athreads e Cilk representam propostas da academia, ambas disponibilizam uma interface de programação que possibilita a manipulação de tarefas concorrentes de forma independente dos recursos da arquiteturas, facilitando o desenvolvimento do código. Pthreads representa uma ferramenta popular comercialmente.

O objetivo de Athreads é oferecer recursos para a exploração do processamento de alto desempenho, provendo tanto uma interface de programação concorrente de alto nível, como um núcleo executivo. Com isso, o programador é capaz de descrever a concorrência de sua aplicação independente dos recursos computacionais disponíveis na arquitetura. Além disso, a atribuição das tarefas aos nós reais de processamento, a criação e sincronização de tarefas e o controle de acesso aos dados compartilhados passam a ser atribuições do ambiente, não mais do programador [Cav06].

Cilk é uma linguagem paralela multithread para programação de multiprocessadores com memória compartilhada. Cilk é uma extensão à linguagem C que fornece uma

estrutura para controle e sincronização do paralelismo, tendo um baixo nível de overhead e possibilitando a criação de uma thread com um custo somente de 2 a 6 vezes maior do que a invocação de uma função em C. Muitos programas Cilk que rodam em um processador acabam tendo o desempenho com quase nenhuma degradação em comparação a um programa equivalente em C [Joe96].

PThreads é uma interface de manipulação de threads padronizada em 1995 pelo Institute of Electrical and Electronic Engineers(IEEE). Foi definida como um conjunto de tipos de dados e procedimentos em C, definida na biblioteca pthreads.h. A idéia básica é a de ser uma API para o gerenciamento de threads [Dre05].

4. Estratégias adotadas

Para critério de validação do algoritmo do caixeiro viajante, a primeira versão foi desenvolvida de forma seqüencial, avaliando casos com 10,11 e 12 cidades. A definição da concorrência no algoritmo foi obtida pela divisão de tarefas que possuem execução independente dentro da aplicação. A distância completa de um caminho foi separada para ser computada concorrentemente. O caminho encontrado é computado a partir das somas dos percursos das cidades que o caixeiro viajante visita a partir de sua origem até a cidade destino. Cada thread criada calcula independentemente o melhor caminho em um intervalo. Um exemplo é uma aplicação que possui 100 caminhos e duas threads. Neste exemplo cada thread irá calcular 50 caminhos. A primeira thread irá calcular o menor caminho no intervalo de 1..50 e a segunda thread irá calcular o melhor caminho no intervalo de 51..100. Durante a sincronização é escolhido o menor caminho retornando por cada thread.

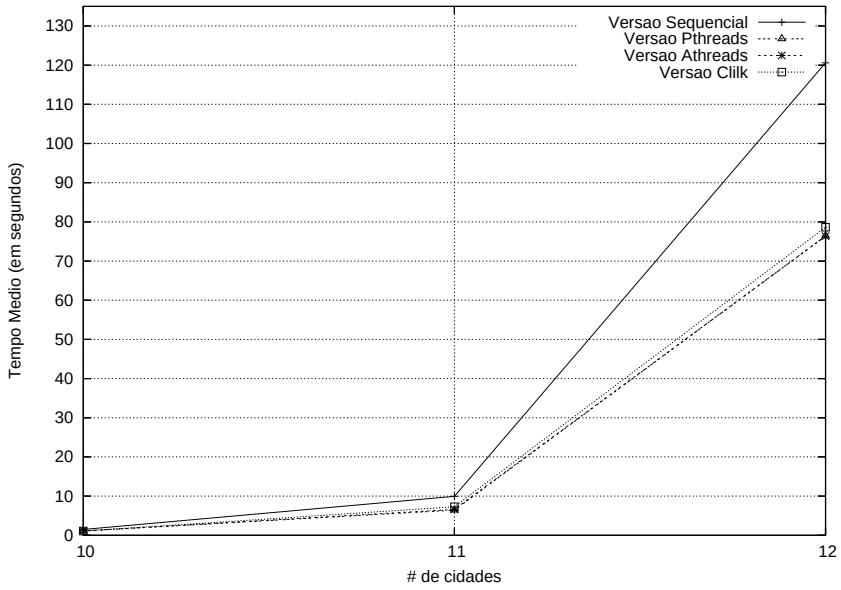

Figura 1. Tempos obtidos durante execuções

Os testes de execução para este trabalho, foram feitos em um Pentiun 1,73 GHZ Dual-Core, possuindo 512 de memória RAM e sistema operacional Debian GNU Linux.

Através da Figura 1 é mostrado o tempo de execução do caixeiro viajante com 10, 11 e 12 cidades de forma seqüencial, em Pthreads, Athreads e em Cilk, sendo possível notar a diferença nos tempos obtidos entre a versão seqüencial e as versões concorrentes. Os resultados mostram um ganho considerável de desempenho a medida que a carga computacional aumenta. Os resultados obtidos com as execuções no ambiente Athreads, Cilk e Pthreads foram muito semelhantes, conseguindo praticamente o mesmo ganho de desempenho em ambas as situações, quando comparado ao tempo de execução apresentado pela versão seqüencial .

5. Conclusão

Através do desenvolvimento do algoritmo do caixeiro viajante de forma concorrente, pode-se obter um aumento de desempenho considerável. A diferença entre os tempos seqüenciais e concorrentes mostra que a efetiva utilização dos recursos disponibilizados pela arquitetura permite obter uma diminuição do tempo de execução e, com isto, um aumento no desempenho da aplicação. A projeção dos tempos de execução para um número de cidades maior do que o número testado, permite observar que versões concorrentes obterão tempos até 50% mais rápidos do que os tempos atingidos pela aplicação em sua versão seqüencial. Este ganho de desempenho pode ser explicado, pois a versão seqüencial computa a distância de apenas um caminho por vez. Desta forma, a computação dos caminhos restantes fica bloqueada. A versão concorrente consegue calcular a distância entre caminhos concorrentemente, conseguindo então, computar mais de um caminho por vez. Tendo em vista os diversos problemas reais que podem ser modelados de forma similar ao caixeiro viajante, a obtenção de melhores resultados no seu algoritmo ocasiona uma melhoria em processos executados diariamente em casos reais.

Como trabalho futuro, deseja-se implementar o algoritmo do caixeiro viajante em um ambiente distribuído, explorando tanto a concorrência intra-nó utilizando os ambientes testados, como também a concorrência através da distribuição de tarefas entre diferentes nós de um agregado de computadores.

Referências

CAVALHEIRO, G. G. H. et al Anahy: A programming environment for cluster computing In VII High Performance Computing for Computational *Science*, Berlin 2006 Springer-Verlag(LNCS 4395)

DREPPER, U. e MOLNAR, I. The native POSIX thread library for Linux. http://people.redhat.com/drepper/nptl-design.pdf(05/dez/2007)

MOLE, V. L. D. et al Caixeiro Viajante - Abordagem Baseada no Algoritmo Simulated Annealing In IV Congresso Brasileiro de Computação, pages 18–21, Brasil, 2004

JOERG, C. F. The Cilk System for Parallel Multithreaded Computing In MIT Press, Massachusetts, Jan. 1996