[[1]](#footnote-1)

**3º Laboratório de Sistemas Operacionais**

**CES-33**

Prof. Cecília de Azevedo Castro César

Davi Grossi Hasuda, Eduardo Henrique Ferreira Silva, 17/04/2018

*Resumo*— O presente relatório apresenta os resultados encontrados na terceira atividade laboratorial de CES-33, no qual se estudou a aplicação de Threads em uma implementação do algoritmo de otimização PSO.

# Introdução

A aplicação escolhida foi a implementação do algoritmo de otimização PSO (*Particle Swarm Optimization*). Tal algoritmo consiste em simular um enxame de partículas que busca o máximo ou mínimo de uma função custo a ser definida de acordo com o problema, em um domínio restrito.

Cada partícula tem acesso ao seu melhor resultado encontrado e ao melhor resultado global e, com isso, decide qual será sua velocidade, ou seja, qual região da função custo irá explorar em seguida.

Esse é um modelo de solução bastante genérico, mas para os propósitos do laboratório, será aplicado no contexto de precificação de produtos. Consideremos, por exemplo, que a quantidade q de celulares vendidos varie de acordo com a escolha do preço p e com o valor m investido em *marketing*, seguindo a seguinte função:

Assim o lucro é dado pelo número de celulares vendidos multiplicado pelo lucro individual de cada celular, dado pela diferença entre seu preço e o custo de fabricação c.

Dessa forma o lucro – que é a função custo a ser maximizada - é dado por:

O PSO se mostra bastante útil em conseguir encontrar o valor, uma vez que a solução analítica para este problema não pode ser encontrada.

# Implementação de Aplicação

***Solução sequencial***

O primeiro passo da solução foi implementar o algoritmo sequencialmente. Para isso, definiu-se três classes:

1. Classe Particle: essa classe representa uma partícula individualmente, que consegue fornecer sua posição, sua velocidade, e a posição e valor do melhor ponto que ela encontrou. A partícula consegue se mover caso lhe seja fornecida uma nova velocidade.
2. Classe Population: essa classe é responsável por controlar todas as partículas. Ela, por exemplo, retorna e atualiza o máximo global e roda uma iteração do processo de busca. Além disso, a classe cria todas as partículas inicialmente, posicionando elas em um ponto aleatório do domínio da função. A cada nova iteração, a classe Population calcula a nova velocidade de cada partícula e passa esse valor para a partícula, que executa a movimentação.
3. Classe CostFunction: representa a função a ser maximizada no intervalo. Dado um ponto do domínio, retorna o valor da função naquele ponto.

Além das classes acima descritas, utilizamos também uma função main para cada solução apresentada, para coordenar o funcionamento do algoritmo como um todo, além de mostrar os resultados obtidos. A função main1.cpp é a solução sequencial, a main2.cpp é a solução paralela sem exclusão mútua e a main3.cpp é a solução paralela com exclusão mútua.

Explicitaremos agora cada uma das classes citadas, além da main da primeira solução.

A classe que representa as partículas é consideravelmente simples: inicializa a partícula com velocidade zero e, dado uma velocidade, faz a movimentação. Abaixo, podemos ver tal classe.



*Figura 1: classe que representa uma partícula*

Um dos pontos mais fundamentais para que as partículas encontrem o ponto máximo corretamente é o modo como a nova velocidade de cada partícula é calculada, durante cada iteração. Abaixo, vemos a fração de código que comanda uma nova iteração. Após, explicaremos seu funcionamento.



*Figura 2: fração do código de comanda uma nova iteração*

Entre as linhas 33 e 35, são definidas três constantes que serão utilizadas adiante. Após, entre as linhas 37 e 63, iteramos sobre todas as partículas realizando o seguinte procedimento:

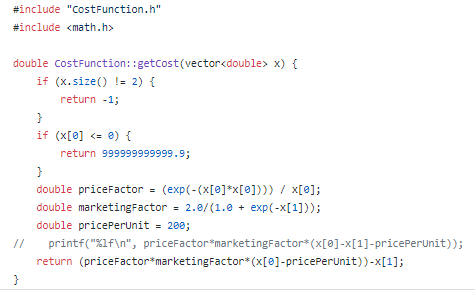
1. Inicialmente calcula-se a nova velocidade da partícula através de uma combinação linear entre a velocidade atual, a direção que leva ao máximo pessoal da partícula e a direção que leva ao máximo global encontrado. As constantes definidas anteriormente são utilizadas para ponderar a combinação linear. Observemos também que utilizamos uma componente aleatória na ponderação das duas direções.
2. Após, entre as linhas 48 e 53, verifica-se se a velocidade calculada anteriormente gera uma nova posição válida. Caso não, corrige-se para a posição valida mais próxima.
3. Depois, a partícula é movida, na linha 57.
4. Finalmente, atualiza-se o máximo global.

Importante citar que a fração anterior faz parte da classe Population. Além disso, a classe em questão também inicializa todas as partículas em um ponto aleatório do domínio. Tal inicialização pode ser vista abaixo.



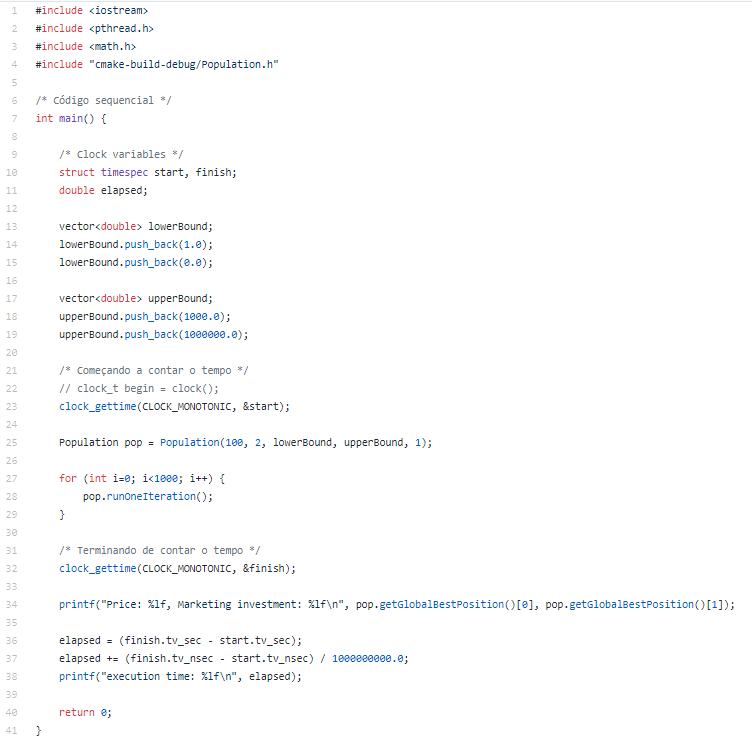
*Figura 3: porção da classe Population que faz a inicialização de todas as partículas.*

Já a classe CostFunction somente retorna o valor da função dado um ponto do domínio, e pode ser vista abaixo.



*Figura 4: classe CostFunction.*

Agora, vejamos o funcionamento da função principal, que coordena a execução como um todo. Abaixo, o código da função main.cpp.



*Figura 5: função principal, main1.cpp*

Inicialmente, definimos os dois pontos que serão os limites do domínio analisado e as variáveis que serão utilizadas na medição do tempo. Na linha 23, começamos a contagem do tempo. Após, na linha 25, inicializamos a população. Os parâmetros são, respectivamente, o número de partículas (100, no exemplo), o número de dimensões do domínio (2), os limites do domínio, definidos anteriormente, e uma semente para o gerador de números pseudoaleatórios utilizado (1). Após, imprimimos o melhor valor encontrado globalmente e o tempo tomado na execução.

Assim, explica-se a implementação da solução sequencial. Nas duas próximas duas sub-seções, explicaremos mais sobre as soluções paralelas. É fundamental ressaltar que as três classes explicadas acima serão utilizadas em todas as três soluções. A única coisa que mudará sera a main a ser utilizada.

***Solução paralela sem mecanismo de exclusão mútua***

***Solução paralela com mecanismo de exclusão mútua***

# Validação da solução

# Discussão

# Conclusão

Realizada a atividade, consegue-se observar na prática o efeito da implementação correta de threads no desempenho do programa. Ademais, percebeu-se o quão cuidadoso deve-se ser ao trabalhar com tarefas que serão executados em paralelo e manejam dados em comum, para que não exista conflitos.

No geral, reforçou-se bem os conceitos aprendidos em sala, além de aumentar a familiarização com o sistema operacional que será utilizado nas trilhas, mais adiante no curso.

1. [↑](#footnote-ref-1)