[[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:

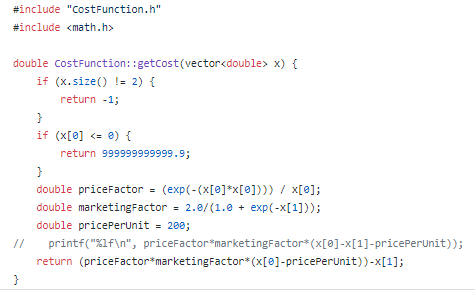
1. Inicialmente calcula-se a nova velocidade da partícula através de uma combinação linear entre a velocidade atual, a direção que leva ao máximo pessoal da partícula e a direção que leva ao máximo global encontrado. As constantes definidas anteriormente são utilizadas para ponderar a combinação linear. 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.



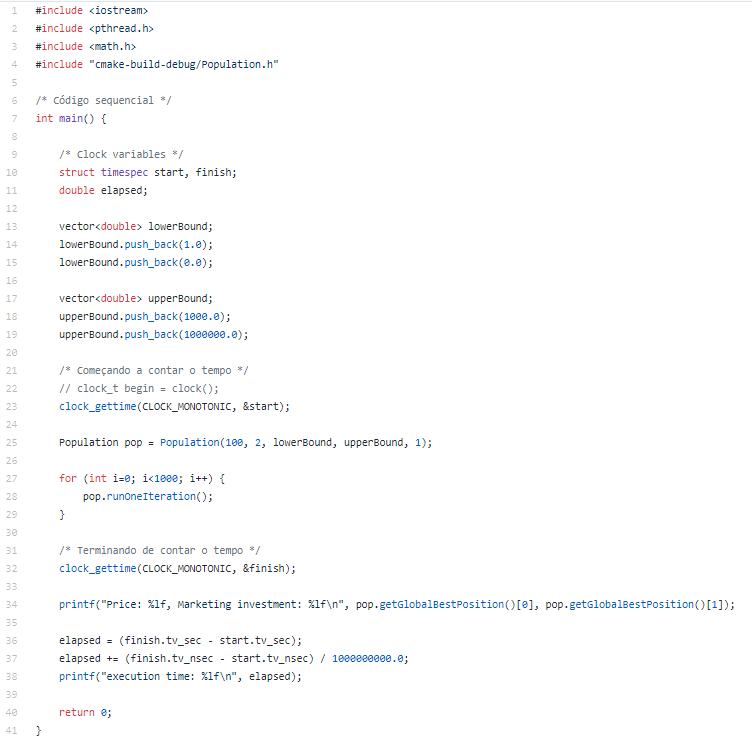
*Figura 3: porção da classe Population que faz a inicialização de todas as partículas.*

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***

# Análise dos resultados

Para a validação dos resultados, deve-se comparar os resultados obtidos ao final das execuções de cada uma das versões do programa. Ademais, serão medidos os tempos para a execução do programa com e sem o uso de *multithread* para que se possa ter um comparativo mais abrangente de qual das soluções é a mais recomendada a ser adotada.

Como o algoritmo utilizado – *Particle Swarm Optimization* – possui um fator aleatório, esperam-se medidas próximas para os resultados das diferentes versões do programa, mas não necessariamente serão os mesmos valores encontrados. Inicialmente deseja-se determinar o a quantidade de partículas e iterações necessárias para que o algoritmo convirja para um resultado. Para isso foi adotado o seguinte critério: utilizando-se o algoritmo sequencial após se escolher uma quantidade de partículas e de iterações rodou-se o programa dez vezes com diferentes *seeds* para a geração dos números pseudoaleatórios e registrou-se o maior e o menor preço sugerido (assim como o maior e o menor investimento em marketing sugerido). Veja-se a **Tabela 1** com os resultados encontrados.

Tabela 1. Resultados com diferentes quantidades de partículas e número de iterações. Os valores apresentados são aproximações truncadas dos valores reais retornados pelo programa.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Total de partículas** | **Total de iterações** | **Menor preço final** | **Maior preço final** | **Menor investimento em MKT** | **Maior investimento em MKT** |
| 1000 | 1000 | 48 | 991 | 267 | 1706 |
| 1000 | 10000 | 48 | 991 | 267 | 1116 |
| 10000 | 1000 | 63 | 632 | 1 | 35 |

Nota-se que os resultados ainda não convergem, possuindo variações muito grandes entre o maior e menor resultados encontrados. Contudo, percebe-se a diminuição da amplitude dos valores encontrados, resultados que é mais significativo entre os valores para investimento em marketing. Isso sugere que o código possui um funcionamento adequado. Contudo, devido às limitações de tempo serão limitados o número de partículas e de iterações para os resultados apresentados a seguir.

Para que o código convirja de forma adequada, o tempo de espera para o programa funcionar excede o tempo hábil para se produzir este relatório, mas incentiva-se ao leitor que aumente o número de partículas e de iterações e deixe o programa rodando pelo tempo necessário para se obter um resultado coerente. Como referência, tendo 1000 partículas e 10000 iterações, o tempo para gerar um resultado foi de aproximadamente 35 segundos na máquina dos alunos. Lembra-se aqui também que é possível continuar havendo uma comparação justa entre as diferentes versões do algoritmo dado que é possível escolher a *seed* que gerará os números pseudoaleatórios.

Uma decisão tomada também para se acelerar o processo de convergência do resultado foi alterar o limite mínimo do preço do produto para seu preço de fabricação (200 unidades).

Para que sejam comparados os desempenhos nas diferentes versões, gerou-se a **Tabela 2**, em que se mostram os resultados encontrados em cada uma das versões do código de acordo com o total de partículas geradas e o número de iterações do algoritmo. O total de *threads* utilizadas nas versões em *multi thread* foram 4. Ressalta-se que a **Tabela 2** tem por intuito medir o desempenho de cada tipo de solução, e não a validade das respostas, assim as duas versões em *multithread* possuem diferentes *seeds* para seus geradores de números pseudoaleatórios.

Tabela 2. Tabela com o desempenho de cada uma das soluções apresentadas. Os resultados para os valores de retorno do programa foram aproximados por truncamento. O tempo de execução é medido em segundos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Versão** | **Total de partículas** | **Total de iterações** | **Preço final** | **Investimento em MKT** | **Tempo de execução** |
| main.cpp | 100 | 1000 | 736 | 7698 | 0,9435 |
| main2.cpp | 100 | 1000 | 985 | 81664 | 0,2037 |
| main3.cpp | 100 | 1000 | 703 | 38884 | 0,2752 |
| main.cpp | 100 | 10000 | 736 | 7698 | 3,9567 |
| main2.cpp | 100 | 10000 | 821 | 15716 | 2,1631 |
| main3.cpp | 100 | 10000 | 453 | 47044 | 2,5172 |
| main.cpp | 1000 | 100 | 838 | 275 | 0,655 |
| main2.cpp | 1000 | 100 | 488 | 7396 | 0,2595 |
| main3.cpp | 1000 | 100 | 353 | 3285 | 0,3919 |

Nota-se que o desempenho dos programas espelha o esperado. O mais rápido de todos é aquele que utiliza mais de uma *thread*,mas que não realiza nenhum tipo de sincronização. Enquanto que o mais lento é o programa sequencial

Finalmente, confrontam-se os resultados retornados pelo programa para os dois casos em que é *multi thread*, pois deseja-se saber se a falta de sincronização altera o resultado final. Para tal, o programa foi ajustado para que as *seeds* geradoras dos números pseudoaleatórios sejam as mesmas nas *threads* das diferentes versões (main2.cpp e main3.cpp). Na **Tabela 3** é possível observar os resultados obtidos dessa comparação.

Tabela 3. Comparação de resultados com e sem sincronização utilizando as mesmas seeds. Os resultados para os valores de retorno do programa foram aproximados por truncamento.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Versão** | **Total de partículas** | **Total de iterações** | **Preço final** | **Investimento em MKT** |
| main2.cpp | 100 | 1000 | 736 | 7698 |
| main3.cpp | 100 | 1000 | 736 | 7698 |
| main2.cpp | 100 | 10000 | 736 | 7698 |
| main3.cpp | 100 | 10000 | 736 | 7698 |
| main2.cpp | 1000 | 100 | 200 | 878 |
| main3.cpp | 1000 | 100 | 200 | 878 |
| main2.cpp | 1000 | 1000 | 200 | 878 |
| main3.cpp | 1000 | 1000 | 200 | 878 |

Observa-se que não houve diferença nos resultados finais, mesmo abrindo-se mão da sincronização.

# Discussão

Na análise do problema, observou-se que a falta de sincronização não altera a saída. É razoável induzir que quando se aumentar consideravelmente o número de iterações e partículas, a diferença entre os retornos dos programas com e sem sincronização vai se manter pequena. Isso acontece devido ao fato de que a probabilidade de encontrar um novo valor de *globalBest* fica sempre menor, cada vez que essa mesma variável é atualizada. Isso faz com que os acessos a ela não sejam frequentes em dificilmente há disputa. Dessa forma, e considerando-se também as medidas de desempenho, pensando em um cenário real descarta-se a solução em que é realizada a sincronização.

Resta considerar a solução sequencial, em que o desempenho é bem pior, podendo demorar mais de quatro vezes mais quando comparada à solução com maior desempenho. Sabe-se que o custo de implementação de *multithread* não foi custoso aos programadores comparado com o restante do programa. Mas vale ressaltar também que em um cenário real, não há a necessidade de grande desempenho para se determinar o preço de um produto, pois não é algo que costuma ser atualizado com frequência (tome como referência o preço de um celular, por exemplo, em que muitas vezes o preço anunciado no lançamento se mantém por até um ano).

Assim, depende-se da urgência com que se precisa do resultado que, em grande parte das vezes, não é urgente. Reutilizando o exemplo do celular, os lançamentos de uma nova versão possuem o espaço de um ano entre lançamentos, e é necessário iniciar a produção dos celular com meses de adiantamento para suprir a demanda do lançamento. Tendo isso em mente, é possível assumir que há mais de um mês disponível para o programa retornar uma resposta, dando espaço, então, para uma solução sequencial.

Finalmente, a versão não sequêncial foi de fácil implementação comparada à sequencial. Considerando-se os desempenhos, essa solução é preferível e dá mais conforto ao usuário, sabendo que após retornado o resultado ele terá tempo de fazer uma segunda validação dos dados. Contudo, caso não seja possível utilizar a solução não sequencial, não há grandes perdas, apenas um atraso na entrega do resultado, que dificilmente será prejudicial.

# Máquina Virtual (Bônus)

Foi realizada uma comparação entre os tempos de execução do programa rodando em uma máquina virtual e na máquina hospedeira. Todas as execuções foram realizadas com 1000 partículas e 1000 iterações.

Tabela 4. Comparação de desempenho entre máquina hospedeira e máquina hóspede. Os tempos são medidos em segundos.

|  |  |  |
| --- | --- | --- |
| **Máquina** | **Versão** | **Tempo de execução** |
| Hospedeira | main.cpp | 4,2649 |
| Hospedeira | main2.cpp | 2,4464 |
| Hospedeira | main3.cpp | 2,1087 |
| Virtual | main.cpp | 4,9135 |
| Virtual | main2.cpp | 2,6871 |
| Virtual | main3.cpp | 2,5649 |

No caso apresentado, diferentemente do restante dos testes, mostrou a main3.cpp sendo mais rápida que a main2.cpp, muito embora essa diferença seja pequena. Quando comparam-se os respectivos tempos entre a máquina hospedeira e a hóspede, percebe-se que a hospedeira foi ligeiramente mais rápida em todos os testes. Isso faz sentido dado que a máquina hóspede está rodando em cima da hospedeira e, por causa disso, deve ter uma menor velocidade de processamento.

A diferença de tempos é pequena para os testes apresentados, contudo aumentando-se a quantidade de partículas e o número de iterações essa diferença tende a aumentar. Observa-se aqui que a máquina hospedeira utiliza macOS 10.13.4 enquanto que o sistema operacional da hóspede é ubuntu 16.04.

# 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)