Paralelização do Algoritmo de Colonização de Formigas com o Problema da Mochila de Múltiplas Restrições

Pedro Barbiero
Universidade Estadual de Maringá
pedrobarbiero@gmail.com

5 de agosto de 2016

Abstract

This report describes the process of parallelizing the Ant Colonization Optimization algorithm, used to solve the Multiple-Constraint Knapsack problem, through a distributed memory model. It was noted that, given a sufficiently small message size that the overhead from this model is negligible, achieving very high speedup factors. It was also noted that at certain instances, there was a super-linear speedup and it was hipotesized that this happens due to the hyper-threading technology provided on the Intel i7 processor, but no conclusion was reached from this phenomenom.

1 Introdução

O algoritmo de otimização da colônia de formigas(sigla em inglês, ACO) é uma heurística baseada em probabilidade no qual se baseia no conceito de formigas que buscam alimentos e deixam trilhas de feromônio para que outras formigas encontrem este mesmo alimento[2]. Em termos de computacionais, considera-se que as *formigas* são elementos que irão atravessar um grafo de um problema, enquanto a *comida* será uma solução válida para este problema, tal que as formigas irão encontrar uma solução boa(e possívelmente, ótima) para o dado problema.

O problema em questão é nomeado *Problema da Mochila de Múltiplas Restrições*, em inglês *Multiple Knapsack Problem*. Este problema é uma variação do problema da mochila onde cada nó do grafo do problema representa um estado da mochila, e cada aresta representa a introdução de um novo item na mochila. Este problema foi mostrado como NP-Completo por Gens, G[4].

Neste trabalho, iremos utilizar o *ACO* para resolver o problema em questão, e então melhorar o tempo de solução do problema criando múltiplas instâncias do programa de solução, tais que elas se comuniquem e resolvam o problema da mesma forma que em um programa sequencial, podendo ser escalada para diversas máquinas ou mesmo em diversos processadores de uma única máquina.

2 Problema da Mochila de Múltiplas Restrições

2.1 O Problema da Mochila

O Problema da Mochila tradicional é um problema de otimização combinatória, normalmente descrito como **"Dado um conjunto de possíveis itens, cada um com um certo valor e um certo peso, e uma mochila com uma capacidade de peso limite, qual é a melhor combinação de itens que a mochila consegue suportar, tal que o valor seja maximizado?"**.

Matematicamente, o problema é descrito como:

$$\max \sum_{i=1}^{n} v_i x_i$$

$$tal\ que \sum_{i=1}^{n} w_i x_i \leq W$$

Onde

- n é a quantidade de itens que podem ser postos na mochila,
- v_i é o valor do item i,
- w_i é o peso do item i,
- W é a capacidade da mochila e
- x_i é a quantidade de vezes em que o item i é colocado na mochila.

Em particular, a variação mais comum do problema a ser encontrado é a *Mochila Binária*, onde $x_i \in \{0, 1\}$, ou seja, existe apenas um item de cada tipo que pode ser posto na mochila, e este item não pode ser fracionado.

2.2 Mochila Binária com Múltiplas Restrições

Outra variação do problema da mochila binária determina que a mochila possui mais do que uma restrição - por exemplo, além da capacidade de peso existe um limite de volume. O problema é então descrito na forma

$$\begin{array}{rll} \max & \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij} & \\ \textit{tal que} & \sum_{j=1}^{n} w_j x_{ij} \leq W_i, & \forall\ 1 \leq i \leq m, \\ & \sum_{i=1}^{m} x_{ij} \leq 1, & \forall\ 1 \leq j \leq n, \\ & x_{ij} \in \{0,1\}, & (\forall\ 1 \leq j \leq n) \wedge (\forall\ 1 \leq i \leq m) \end{array}$$

3 Sistema de Colonização de Formigas

O Algoritmo de Colonização de Formigas, ou *ACO*, é uma meta-heurística de otimização aplicada em diversas áreas e é útil quando um tempo curto é necessário para resolver um problema [3]. É um sistema onde múltiplos agentes(formigas) agem de uma forma relativamente simples porém interagem entre si(através de um feromônio) resultando em um comportamento complexo.

O ACO consiste em repetir diversas vezes um conjunto de operações:

1. Buscar solução baseado no feromônio atual
2. Atualizar o Feromônio
3. (opcional) Operações de *Daemon*

Estas operações são repetidas várias vezes até que uma condição de parada seja satisfeita: um limite no número de iterações, tempo ou mesmo uma interrupção externa. Em geral, espera-se que o algoritmo encontre uma melhor solução se este ciclo for repetido mais vezes.

3.1 Construção da Solução

Como é comum para o ACO tratar o problema como um grafo, a construção da solução pode ser feita pelo Algoritmo 3.1.

Embora o algoritmo pareça relativamente simples, o método de seleção da aresta que será avançada é o diferencial do *ACO*. Neste sistema, atribuem-se a cada aresta dois valores chamados *feromônio*(τ) e *atratividade*(μ). Estes valores podem, ou não, ser relativos ao estado atual da solução no momento de sua construção, e relativos aos atributos das arestas; A seleção é feita então probabilisticamente: Dado um conjunto A_i de possíveis arestas para

construir a solução em um passo i, a chance de selecionar uma aresta j é dada por

$$p_j = \begin{cases} \dfrac{\tau_j^\alpha \mu_j^\beta}{\sum_{k \in A_i} \tau_k^\alpha \mu_k^\beta}, & \text{se } j \in A_i, \\ 0 & \text{se } j \notin A_i. \end{cases}$$

O valor de μ_j é dado por uma combinação de fatores do problema; no problema da mochila, por exemplo, pode se referir a uma relação entre os atributos de um item e a capacidade total da mochila. Este valor deve ser determinado como uma heurística para ajudar a selecionar as melhores arestas para a solução. As constantes α e β determinam o peso que o feromônio τ e a atratividade μ terão, respectivamente.

3.2 Atualização do Feromônio

Em cada iteração do sistema, o feromônio será atualizado de acordo com a melhor solução encontrada por uma formiga nesta iteração, assim aumentando a probabilidade de seguir um caminho de uma boa solução para a próxima iteração. Além de existir a soma do feromônio, ele também irá *evaporar*, isto é, o valor numérico do feromônio será multiplicado por uma constante $0 < \rho < 1$. Assim, a atualização do feromônio é dada pelo algoritmo 3.2.

Note que $\Delta\tau$ é um valor que pode, ou não, depender de atributos da solução S, mas é comum que este valor seja relativo à qualidade da solução encontrada - assim, soluções melhores colocam mais feromônio no sistema.

3.3 Operações de Daemon

São operações dependentes da implementação. Por exemplo, o cálculo de $\tau_j^\alpha \mu_j^\beta$ é computacionalmente custoso, e pode ser útil atualizá-lo em cada

Algorithm 3.1: Algoritmo de construcão de solução

```
Solucao ← ∅;
while true do
    A ← conjunto de arestas que levam a uma solução válida;
    if A = ∅ then
        return Solucao;
    Solucao avança em uma aresta de A;
```

Algorithm 3.2: Atualização de Ferômonio

Input: S_k: Conjunto de soluções encontradas por todas as formigas
ρ: Taxa de evaporação do feromônio, $0 \leq \rho \leq 1$
foreach j : *aresta no sistema* **do**
    $\tau_j \leftarrow \tau_j * \rho$;
foreach $S \in S_k$ **do**
    foreach i: *Aresta* $\in S$ **do**
        $\tau_i \leftarrow \tau_i * \Delta\tau(S)$;

aresta apenas uma vez por iteração ao invés de calculá-lo em todas as chamadas, assim otimizando a velocidade de execução do programa.

4 Paralelização do Algoritmo em Memória Distribuída

Programas sequenciais podem ser programados, com a ajuda de uma biblioteca específica, de tal forma que eles possam ser executados em múltiplas instâncias através de diversas máquinas interligadas em rede(local ou internet). A **Message Passing Interface**, ou MPI, é o modelo de bibliotecas para um sistema de passagem de mensagens; Neste trabalho em específico foi-se utilizada a biblioteca `mpich`, que implementa este modelo.

4.1 Conceitos básicos do MPI

Alguns conceitos precisam ser compreendidos para entender como o MPI funciona, e como o programa pode tirar vantagem destes conceitos para resolver problemas sobre múltiplas máquinas ou processadores. Primeiramente, deve-se entender como **nó** uma instância de um programa que pode se comunicar com outros nós de mesmo tipo; Esta comunicação não é detalhada pois pode ocorrer internamente(na mesma máquina) ou externamente(entre diferentes máquinas), e ela pode ocorrer de acordo com o protocolo de comunicação definido pelo MPI - para o programa e o algoritmo em questão, esta comunicação é abstraída e tratada pela biblioteca.

4.1.1 Comunicadores

No MPI, um **comunicador** é um conjunto de um ou mais nós que fazem parte, de forma lógica, de um mesmo grupo de comunicação. Comunicadores

são utilizados para determinar a topologia dos nós em execução assim como organizar as passagens de mensagem(principalmente as do tipo *broadcast*). Por padrão, um comunicador global chamado `MPI_COMM_WORLD` é definido na inicialização dos nós, e este comunicador irá englobar todos os nós do sistema.

Neste projeto, foi-se utilizada uma topologia do tipo **estrela**; Isto significa que todos os nós irão passar mensagens entre todos os outros nós, significando que o comunicador `MPI_COMM_WORLD` será o único criado.

4.1.2 Rank & Size

A identificação de um dado nó é dada por seu **rank** dentro de um comunicador; O comunicador, por sua vez, possui um número de nós variável que é dado por **size**, e cada nó pertencente a um comunicador obedece à lei de $0 \leq rank_{no} < Size_{comm}$.

Neste projeto, um rank em particular, o rank 0, é especificado como o nó que irá produzir a saída de dados; Além de imprimir na tela(ou arquivo de saída) o resultado do programa, o nó com rank 0 não difere dos outros nós em execução.

4.1.3 Send, Receive, Broadcast

Em um sistema baseado em passagem de mensagens, alguns comandos devem ser definidos para que estas mensagens sejam, de fato, passadas. A função `Send` envia uma quantidade de bytes para um certo nó alvo com uma *tag*, sendo que estes bytes contém alguma informação útil(a mensagem) para o programa alvo. Para cada chamada de `Send` de uma fonte a um destino, deve haver uma chamada, no destino, da função `Receive` que indica a fonte da mensagem assim como o tamanho da mensagem e o tipo da mesma. Estruturas complexas(`struct`, em C) devem ser registradas em cada um dos nós que executam estas funções para que estes mantenham uma sincronia entre suas comunicações.

Já o comando `Broadcast` recebe como parâmetro uma variável, o tamanho dela, um nó fonte e um comunicador. Esta função irá então recuperar a informação contida nesta variável no nó fonte e enviá-la para todos os nós que fazem parte deste comunicador, que irão receber a informação e registrá-la na variável que eles possuem(lembrando que cada nó possui seu próprio conjunto de memória e de variáveis). Assim, é possível enviar uma informação a diversos nós de forma eficiente(sem a necessidade de múltiplas chamadas de `Send` e `Receive`).

4.2 Distribuição de Instâncias e Execução

A execução do algoritmo será feita em auxílio do programa `mpirun`, que recebe como parâmetros o número de nós a ser executado, os *hosts*, ou máquinas, que irão executar o programa, além do próprio programa com seus próprios argumentos.

As instâncias de execução podem ser distribuídas de duas formas: por máquina ou por *slot*. Na distribuição por máquina, cada máquina alvo receberá uma quantidade balanceada de instâncias, independente do hardware que a máquina executa; Já na distribuição por slot, pode-se determinar que uma máquina receba uma certa quantidade de instâncias antes de começar a criar novas instâncias na próxima máquina. Esta segunda forma de distribuição pode ser vantajosa, pois a comunicação entre nós em uma mesma máquina é mais rápida do que a comunicação entre nós de máquinas distintas(afinal, a comunicação é local), porém deve-se configurar manualmente exatamente quantos *slots* uma máquina deve possuir antes de ser considerada "cheia".

4.3 Leituras Adicionais

O tópico de paralelização é muito mais extenso do que o escopo deste trabalho. Recomenda-se a leitura de outras fontes como `https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage/`[1].

5 Aplicando o Problema da Mochila ao Ant System

No Problema da Mochila, soluções válidas podem ser interpretados como estados em que a mochila se encontra. Assim, uma mochila começa em um estado inicial onde suas capacidades C_k estão em seu valor máximo, e nenhum item está adicionado à mochila.

A transição de estados da mochila é dada pela adição de um novo item na mesma; Ao adicionar um novo item, as restrições C_k são reduzidas de acordo com os atributos do item i, e o valor da mochila, e portanto qualidade da solução, cresce de acordo com v_i(o valor de i). Se um item possui uma restrição $w_{ik} > C_k$, então a mochila não pode adicionar este item nela - o que significa que o estado em que a mochila se encontra não é ligado à aresta a qual este item é representado.

Levando então a consideração de que o estado da mochila é um nó, e o item é uma aresta, basta determinar uma fórmula para a atratividade μ dos

itens.

Como o intuito deste trabalho é mostrar a paralelização do *ACO*, não foi feito uma análise extensa sobre qual μ provém os melhores resultados em menos iterações. Usaremos então uma fórmula baseada no trabalho de Schiff, K[5], a qual é simples de calcular:

$$\mu_j = \frac{v_j}{\prod r_{ij}}$$

Onde r_{ij} são todas as restrições de um item j e v_j é o valor do item j.

Dado então um número de iterações *iter*, um número de formigas *ants* e um número de itens n onde cada item possui m restrições e um valor v_j, um valor inicial de feromônio φ e um valor máximo de feromônio $\varphi_{\max}$, assim como capacidade máxima da mochila C_k, $0 < k < m$, constrói-se o algoritmo final no Algoritmo 5.1.

6 Paralelização do *ACO* no modelo de memória distribuída

Uma das vantagens de um modelo de memória distribuída é que precisa-se se preocupar apenas com a informação que *deve* ser distribuída entre diferentes nós do processo. No caso do *ACO*, se assumirmos que todos os nós em execução possuem acesso a todos os itens do universo, e que todos os nós são capazes de construir uma solução dada um feromônio, nota-se que a única váriavel é de fato compartilhada e modificada entre as formigas é o feromônio em si: τ_j depende de cada solução encontrada por cada formiga, porém todas as outras variáveis, inclusive o composto entre Feromônio e Desejabilidade $\tau_j^\alpha \mu_j^\beta$, podem ser calculadas antes ou depois de se determinar o valor do feromônio em uma dada iteração.

6.1 Lista de atualização de feromônio

Quando um conjunto de formigas termina sua iteração - isto é, encontram uma solução válida e estão prontas para atualizar o feromônio - é possível criar uma lista $\Delta\tau_j \forall j$ tal o índice j indica qual será a diferença linear do feromônio após a evaporação com o feromônio após o término da atualização. Esta lista é relativamente pequena - cada elemento pode ser dado por um par `{int, double}` que tem um tamanho de 12 bytes na maioria dos sistema; e uma lista de atualização iria conter apenas os itens que irão, de fato, receber um aumento de feromônio além da evaporação; Isto é especialmente útil pois

no problema da mochila, apenas um subconjunto pequeno de itens é incluso na mochila em um grande universo.

6.2 Envio da Mensagem

Pode-se então utilizar a função `Broadcast` para enviar a lista de feromônios para todos os nós do sistema, onde cada nó possui um subconjunto de formigas(em geral, $num\_ants \div num\_nos$). Assim, cada nó irá receber as listas de todos os outros nós, onde então todos os nós irão somar as partes das listas para atualizar suas listas locais de feromônio - minimizando, assim, o tamanho e número de mensagens a serem enviadas no sistema.

Finalmente, ao término do programa(neste caso, do número de iterações), os nós podem então enviar seus melhores resultados ao nó de rank 0 para que este possa finalizar a execução do programa com o resultado geral final.

A forma mais intuitiva de se paralelizar o ACO é em paralelizar as formigas, ou seja, enviar as formigas em *threads* separadas para que trabalhem de forma concorrente. Durante a construção de solução, não existem problemas de concorrência: variáveis globais como τ_i não são modificadas.

6.3 Regiões críticas

Nota-se também que o algoritmo sequencial(5.1) possui 4 estágios bem definidos: Construção de soluções, Evaporação de feromônio, Atualização de feromônio e Atualização da probabilidade de escolha $\tau_j^\alpha \mu_j^\beta$. Existe também uma ordem específica em que cada um destes estágios precisam ocorrer: Não se pode evaporar o feromônio antes de terminar de construir as soluções, pois estas são dependentes do feromônio. Também não se pode atualizar o feromônio antes da evaporação pois isto mudará o valor final da operação; e com certeza não se pode atualizar o feromônio *durante* a evaporação. Também não se pode atualizar $\tau_j^\alpha \mu_j^\beta$ antes que o feromônio esteja completamente atualizado, e não se pode construir uma nova solução antes que $\tau_j^\alpha \mu_j^\beta$ esteja atualizado em cada item.

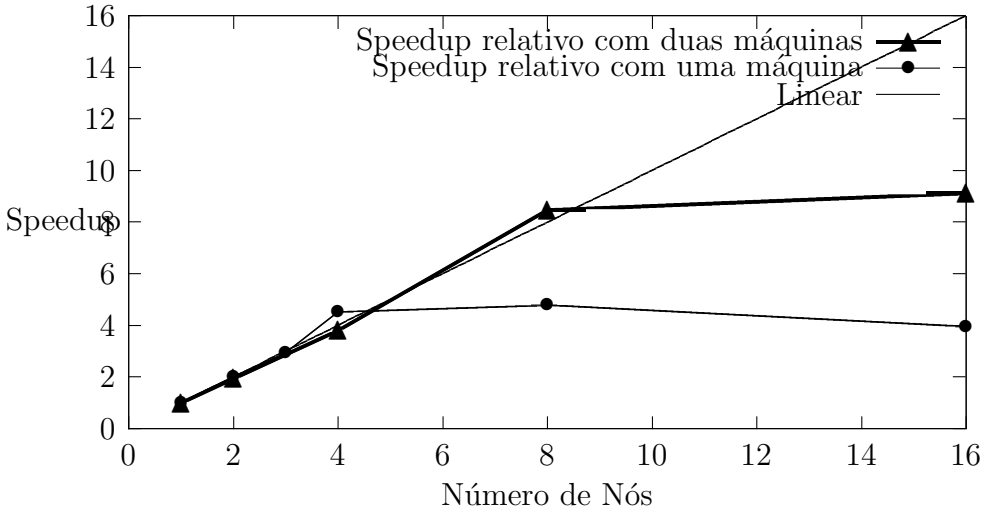
Isto indica que o algoritmo irá se beneficiar do uso de **barreiras**, impedindo que cada thread continue sua execução antes que todas as threads terminem o estágio específico em que se encontram. Além disso, um lock será necessário para impedir que τ_i seja modificado por várias threads ao mesmo tempo. O algoritmo 6.1 mostra como ele irá funcionar para uma Thread. Para o programa final, basta verificar qual thread produziu a melhor solução.

6.4 Análise de desempenho

O programa foi implementado na linguagem `C`, padrão `C11`, compilado com gcc versão 5.3.0 sob otimização `-O2`, e executado em um e duas máquinas com processador **i7 @ 3.5GHz**. A execução ocorreu no sistema operacional Ubuntu 14.04.

Para a coleta de dados, o programa foi executado em 1, 2, 3, 4, 8 e 16 nós, sendo testado o desempenho com uma distribuição de instâncias por máquina e por *slot*, limitando cada máquina a 4 slots. Um gráfico do *speedup* de cada instância pode ser verificado na figura 6.4.

Fig. 6.4: Speedup relativo



Imediatamente nota-se que existe uma *Superlinearidade* no *speedup* dos programas em uma instância em particular: Quando o número de nós é igual ao número total de núcleos físicos de processamento. A hipótese deste fenômeno ter ocorrido se dá ao modo em que o processador i7 se comporta quando sobre uma grande carga de processos ou com o modo em que o sistema operacional escalona os processos quando nesta instância em específica. Testes de outras instâncias não registrados mostram um padrão neste fenômeno em que o speedup superlinear é diretamente relacionado a este número de processos(4 por máquina).

Nota-se também o fenômeno mais esperado de que, ao passar do número de processadores físicos, ocorre uma queda na eficiência de processamento pois o sistema estará passando uma grande quantidade de tempo escalonando os processos - ocorrendo um slowdown perceptível quando o número de processos começa a ficar muito maior do que o número de núcleos de processamento.

Atribui-se o alto desempenho ao fato de que as mensagens passadas são

pequenas e rapidamente processadas entre nós; Além disso, os computadores se encontravam fisicamente próximos e é esperado que uma rede mais congestionada ou esparsa, ou mesmo um aumento no número de máquinas, causaria uma perda de eficácia em relação às instâncias testadas. Mesmo assim, o ganho de tempo no processamento distribuído para uma instância suficientemente grande sobrepõe qualquer perda de eficácia gerada por uma rede esparsa.

A implementação do projeto está disponível em `https://github.com/Barbiero/KnapsackAntSystem/tree/mpibranch`.

7 Conclusões

O *ACO* é um algoritmo que pode ser facilmente e eficazmente acelerado em sistemas de memória distribuída. Uma distribuição inteligente de processos pode gerar ganhos de velocidade muito grande - possivelmente mais rápida do que a execução sequencial de uma versão proporcionalmente menor da mesma instância.

Referências

[1] Barney, B. Introduction to parallel computing, 2016.

[2] Dorigo, M., and Stützle, T. *Ant Colony Optimization.*

[3] Fidanova, S. *Ant Colony Optimization and Multiple Knapsack Problem.*

[4] Gens, G., and Levner, E. *Complexity of approximation algorithms for combinatorial problems: a survey.*

[5] Schiff, K. Ant colony optimization algorithm for the 0-1 knapsack problem.

Algorithm 5.1: Algoritmo sequencial para o ACO

Input:
$iter$: Número de iterações do sistema
$ants$: Número de formigas do sistema
n: Número de itens
m: Número de restrições
$C_k, 0 < k < m$: Capacidade da mochila
$v_j, 0 < j < n$: Valor do item j
$r_{ij}, 0 < j < n, 0 < i < m$: Restrição i do item j
$\mu_j = \frac{v_j}{\prod r_{ij}}$: Atratividade do item j
$\varphi, \varphi_{\max}$: Valor inicial e máximo de feromônio
ρ: Coeficiente de evaporação do feromônio
α: Peso que o feromônio tem sobre a seleção dos itens
β: Peso que a atratividade tem sobre a seleção dos itens
Output: S: estado da mochila na solução final

```
best = ∅;
∀_{0<j<n} τ_j ← φ;
while iter−− > 0 do
    best_x ← ∅      ∀_{0<x<ants};
    //Buscar uma solução;
    for x ← 0 to ants do
        solucao ← constroi_solucao();
        if (solucao > best_x) then
            best_x ← solucao
        if (solucao > best) then
            best ← solucao
    //Atualizar feromonio;
    for j ← 0 to n do
        τ_j ← τ_j * ρ;
    for x ← 0 to ants do
        foreach i: Item ∈ best_x do
            τ_i ← Δτ(best_x);
    Atualizar τ_j^α μ_j^β para cada item;
return best;
```

Algorithm 6.1: Algoritmo de uma thread para o ACO

Input:
N: O numero total de nós
$rank$: O identificador do nó no comunicador global, $1 \leq rank \leq N$
$iter$: Número de iterações do sistema
$ants$: Número de formigas por nó, onde $ants * N$ dá o número total de formigas
n: Número de itens
m: Número de restrições
$C_k, 0 < k < m$: Capacidade da mochila
$v_j, 0 < j < n$: Valor do item j
$r_{ij}, 0 < j < n, 0 < i < m$: Restrição i do item j
$\mu_j = \frac{v_j}{\prod r_{ij}}$: Atratividade do item j
$\varphi, \varphi_{\max}$: Valor inicial e máximo de feromônio
ρ: Coeficiente de evaporação do feromônio
α: Peso que o feromônio tem sobre a seleção dos itens
β: Peso que a atratividade tem sobre a seleção dos itens
Output: S: estado da mochila na solução final

$\forall_{0<j<n} \tau_j \leftarrow \varphi$;
while $iter-- > 0$ **do**
    $best_x \leftarrow \emptyset \quad \forall_{0<x<ants}$;
    for $x \leftarrow 1$ ***to*** $ants$ **do**
        $solucao \leftarrow constroi\_solucao()$;
        if $(solucao > best_x)$ **then**
            $best_x \leftarrow solucao$
    $\Delta\tau_j \leftarrow \Delta\tau(best_x) \quad \forall_j \in best$;
    $\Delta\tau' \leftarrow \Delta\tau$
    for $i \leftarrow 1$ ***to*** N **do**
        $\Delta\phi < -\emptyset$;
        if $rank = i$ **then**
            $\Delta\phi < -\Delta\tau'$;
        `Broadcast`$(\Delta\phi, i,$ `MPI_COMM_WORLD`$)$;
        $\Delta\tau \leftarrow \Delta\tau + \Delta\phi$;
    for $j \leftarrow 1$ ***to*** n **do**
        $\tau_j \leftarrow \tau_j * \rho + \Delta\tau_j$;
        Atualizar $\tau_j^\alpha \mu_j^\beta$;
return $best$;