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

Pedro Barbiero
Universidade Estadual de Maringá
pedrobarbiero@gmail.com

26 de junho de 2016

Abstract

This report describes the process of parallelizing the Ant Colonization Optimization algorithm used to solve the Multiple-Constraint Knapsack problem. The process included separating the ant-structures into different threads, and it was noted that the speedup of the application was affected not only by the number of threads, but also by the memory organization and modelling of the problem. It was also noted that there was a smaller variance in execution time the more threads were used, but that increasing the number of threads does not necessarily mean in a smaller execution time. This report does not go into detail of methods of solving the Knapsack problem nor does it go into detail of different Ant Systems, and uses a very simplified algorithm.

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 paralelizando alguns de seus passos, de tal forma que ele continue resolvendo o problema da mesma forma porém tirando vantagem das arquiteturas multi-processadoras atuais.

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.

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**
        $\tau_i \leftarrow \tau_i * \Delta\tau(S)$;

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 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 de Algoritmos

Algoritmos sequenciais podem tomar vantagem das arquiteturas atuais de processadores e executar suas operações de forma concorrente com a ajuda de certas bibliotecas de programação. Neste trabalho foi-se utilizada a bilbioteca `pthreads.h` na linguagem `C`. Nesta seção, serão explicados alguns mecanismos que são necessários para paralelizar um algoritmo que normalmente seria sequencial.

4.1 Threads

Em um fluxo de execução comum, o *processo* irá chamar uma função inicial(normalmente `main`) e então seguir o código sequencialmente a partir da chamada da função até que seu fluxo de execução seja finalizado - via uma chamada de retorno da função `main`, chamada da função `exit` ou mesmo por uma interrupção externa. Entretanto, utilizando-se de uma biblioteca de paralelização e chamadas específicas ao sistema operacional, é possível criar *threads*, que são fluxos de execução que não são necessariamente executados em sequencia. Estas threads irão então seguir seu fluxo de execução independentes entre si, aproveitando as arquiteturas de multi-processadores para que o código ocorra em paralelo.

4.2 Memória compartilhada e condiçoes de corrida

Quando se trata de um código paralelo, a memória pode - ou não - ser compartilhada, dependendo da implementação do mesmo. Com `pthreads`, a memória do *stack* do programa é compartilhada entre si, o que permite a diferentes Threads se comunicarem ou acessarem informações globais entre si. Existe, porém um problema quando a memória entre threads é compartilhada: condições de corrida(*race conditions*).

Quando duas threads tentam acessar e manipular uma mesma variável ao mesmo tempo, condições de corrida podem ocorrer. Suponha que duas threads, T_0 e T_1, tentem executar a operação a_{++}. Esta operação pode ser expandida desta forma:

$$\begin{aligned} i &\leftarrow a \\ i &\leftarrow i + 1 \\ a &\leftarrow i \end{aligned}$$

Onde i é um registrador do processador. Se, por exemplo, T_0 e T_1 executam estas instruções ao mesmo tempo, quando chegarem na terceira linha $a \leftarrow i$, a variável a receberá o valor $a + 1$, mesmo que duas Threads tenham executado a operação de incremento. Para que isto não ocorra, torna-se necessário o uso de instrumentos de controle de concorrência, chamados de **locks**.

4.3 Locks

Lock ou **Mutually exclusive lock**(`mutex_lock`) é uma instrução que determina uma *região crítica* para que duas threads não possam acessar esta região crítica ao mesmo tempo. Assim, no exemplo acima, o operador a_{++} poderia ser implementado como

$$\begin{aligned} &lock() \\ &\quad i \leftarrow a \\ &\quad i \leftarrow i + 1 \\ &\quad a \leftarrow i \\ &unlock() \end{aligned}$$

Neste exemplo, se T_0 começar a executar a instrução, ele irá adquirir a trava da região crítica, enquanto T_1 irá pausar sua execução na função `lock()`, até que T_0 libere a trava com a função `unlock()` e permita que T_1 adquiria a trava da região crítica, o que irá impedir outra thread de acessar esta região até que T_1 a libere.

No padrão `C11`, a *keyword* `_Atomic` determina que uma variável é uma região crítica, e quaisquer operações feitas sobre esta variável são implicitamente atômicas - efetivamente agindo como se existissem *locks* sobre estas variáveis. Com a biblioteca `pthreads`, o uso de funções como `pthread_mutex_lock` e `pthread_mutex_unlock` permite um controle maior de regiões críticas, não se limitando apenas a operações de uma variável.

4.4 Barreiras

Além dos locks, outra forma de controle de concorrência é a barreira. Barreiras são estruturas que determinam um ponto do programa em que uma thread deve esperar até que n threads atinjam este ponto, onde n é um número especificado para a barreira, normalmente sendo igual ao número de Threads que são relacionadas àquele ponto em específico.

Barreiras podem ser implementadas de diferentes formas; no programa anexado a este trabalho, uma forma particular de barreira chamada **Tree Barrier** foi implementada, onde a a barreira é organizada na forma de uma árvore, em que cada thread está em um nó-folha da árvore e quando esta thread chega no ponto da barreira, ela irá informar o nó pai. Quando o pai das folhas verifica que todas as folhas informaram que a barreira terminou, este irá informar o seu pai até que a raiz note que todas as folhas tenham chegado no ponto de execução desejado, então liberando-as automaticamente.

4.5 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/`[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*

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.

6.1 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.

6.2 Seleção de Formigas

Dado um número qualquer de formigas, pode-se distribui-las entre as threads quase-igualmente(com uma exceção para a última thread, caso $ants \mod T \neq 0$) com duas operações:

$$ini_{ant} \leftarrow threadid * \lceil \frac{ants}{T} \rceil$$
$$fin_{ant} \leftarrow \min(ini_{ant} + \lceil \frac{ants}{T} \rceil, ants)$$

Assim, enquanto um programa que atinge todas as formigas irá iterar sobre as formigas $[0, ants)$, as threads irão iterar igualmente sobre $[ini_{ant}, fin_{ant})$. Além disso, as operações que ocorrem sobre os n itens são analogamente distribuídas entre as threads, onde cada thread irá processar os itens $[ini_{item}, fin_{item})$

6.3 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 notebook **Dell Inspiron 14z 5423**, que possui um processador **i5 3317U 1.7GHz**. Este processador possui dois núcleos físicos e dois virtuais, totalizando quatro núcleos de processamento, com um cache de 128KB/512KB/3072KB para L1/L2/L3 respectivamente. A execução ocorreu no sistema operacional Elementary OS "freya", baseado no Ubuntu 14.04.

Para a coleta de dados de desempenho, o programa foi executado utilizando 1, 2, 4 e 8 threads cada um executados 10 vezes, e mais 10 vezes com uma versão não paralela do programa(ou seja, sem a inclusão da biblioteca `pthreads` ou qualquer mecanismo de paralelismo). O tempo reportado foi coletado com um cronometro implementado dentro do programa, e não inclui o tempo de processamento não relevante à execução do algoritmo(por exemplo, leitura de dados de entrada). A tabela 1 mostra o resultado da coleta de dados.

Nota-se que existe uma perda de desempenho relativo ao aumentar o número de Threads; Idealmente, a coluna de *Speedup* deveria se aproximar ao número de threads. Entretanto, no caso de teste existem alguns problemas que causam uma perda de desempenho. Primeiramente, em uma arquitetura com apenas 4 núcleos de processamento, é natural que, ao passar desse número de threads, exista uma perda de processamento. Isto é visualizado com o fato de 8 threads serem mais lentas do que 4 threads.

Além disso, existe uma grande perda de desempenho com o *overhead* do paralelismo - isto é verificado com a diferença de tempo entre o programa Sequencial e de 1 Thread. Em particular, a existência de um lock para cada τ_j causa um overhead notável.

Em seguida, nota-se uma perda de desempenho entre 2 e 4 threads. Enquanto um Speedup de 1.628 para 2 threads é *aceitável*(porém possivelmente melhorável, se o código implementado), o speedup de apenas 1.852 para quatro threads é extremamente baixo - esta perda de desempenho é uma consequencia do *overhead* em conjunto com o fato de que o computador utilizado possui dois núcleos *virtuais*, que dividem memória com seus núcleos físicos - e com a divisão de memória, existe um número maior de erros de cache.

6.4 Possíveis otimizações

O programa foi escrito com suporte a um número dinâmico de itens e de restrições. Em `C`, isto significa que não é possível alinhar precisamente os dados dos itens com os dados de suas restrições: como estes não são definidos a tempo de compilação, é necessário alocá-los em tempo de execução o que implica que a memória das restrições r_{ij} não ficará alinhada corretamente à estrutura do item i; Isto significa que todas as operações sobre um item que leva em consideração suas restrições causarão uma grande quantidade de misses de cache.

Outra possível otimização seria considerar o feromônio como uma matriz individual a cada thread. Isto significaria que não seria necessário correr os problemas de corrida para atualizar o feromônio, e que cada thread agiria como uma colônia de formigas independente.

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

7 Conclusões

A paralelização do *ACO* pode ser feita com o uso de barreiras; o desempenho da paralelização é completamente dependente de como o problema é

modelado e como a memória é gerenciada, assim como em como o problema toma vantagem da arquitetura interna do computador que irá executar o programa.

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:
$barreira$: Estrura de barreira inicializada antes da criação das Threads
T: O numero total de threads
$threadid$: O identificador da thread, $0 \leq threadid \leq T$
$iter$: 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
$ini_{ant} \leftarrow threadid * \lceil \frac{ants}{T} \rceil$;
$fin_{ant} \leftarrow \min(ini_{ant} + \lceil \frac{ants}{T} \rceil, ants)$;
$ini_{item} \leftarrow threadid * \lceil \frac{n}{T} \rceil$;
$fin_{item} \leftarrow \min(ini_{item} + \lceil \frac{n}{T} \rceil, n)$;
$\forall_{0<j<n} \tau_j \leftarrow \varphi$;
while $iter-- > 0$ **do**
    $best_x \leftarrow \emptyset \quad \forall_{0<x<ants}$;
    `buscar_solucao(`$best_x, best$`)(6.2)`;
    barreira_esperar($barreira$);
    `evap_feromonio()(6.3)`;
    barreira_esperar($barreira$);
    `atualiza_feromonio(`$best_x$`)(6.4)`;
    barreira_esperar($barreira$);
    for $j \leftarrow ini_{item}$ ***to*** fin_{item} **do**
        Atualizar $\tau_j^\alpha \mu_j^\beta$;
    barreira_esperar($barreira$);
return $best$;

Algorithm 6.2: Busca de solução no algoritmo paralelo

```
for x ← ini_ant to fin_ant do
    solucao ← constroi_solucao();
    if (solucao > best_x) then
        best_x ← solucao
    if (solucao > best) then
        best ← solucao
```

Algorithm 6.3: Evaporar Feromonios no algoritmo paralelo

```
for j ← ini_item to fin_item do
    τ_j ← τ_j * ρ;
```

Algorithm 6.4: Atualizar feromonios no algoritmo paralelo

```
for x ← ini_ant to fin_ant do
    foreach i: Item ∈ best_x do
        lock(τ_i);
        τ_i ← Δτ(best_x);
        unlock(τ_i);
```

Tabela 1: Tempo de execução médio (10 execuções)

Nº de Threads	**Média** de tempo	**Desvio Padrão** (σ)	**Speedup** (vs sequencial)	**Speedup** (vs 1 thread)
Sequencial	4'42"114	12"070	1	–
1	6'3"115	1"853	0.777	1
2	3'23"765	10"734	1.260	1.628
4	3'16"29	7"888	1.439	1.852
8	4'11"818	1"579	1.120	1.442