Aplicação de Processamento Paralelo na Solução do Problema da Soma dos Subconjuntos

Fernando Concatto[1]

[1]Bacharelado em Ciência da Computação – Universidade do Vale do Itajaí (UNIVALI)
Caixa Postal 360 – CEP 88302-202 – Itajaí – SC – Brasil

fernandoconcatto@edu.univali.br

Resumo. *O Problema da Soma dos Subconjuntos é um problema computacionalmente difícil, demandando tempo exponencial para ser resolvido. Este trabalho buscou aplicar técnicas de processamento paralelo na busca de soluções para o problema, com a intenção de identificar o ganho de desempenho por thread utilizada. Através da análise dos dados experimentais, foi possível constatar que duas threads ofereceram um desempenho aproximadamente X vezes melhor, enquanto quatro threads ofereceram um ganho de Y vezes.*

1. Introdução

Um problema computacional pode ser interpretado como uma questão a ser respondida, geralmente possuindo *parâmetros*, ou *variáveis*. O problema deve ser definido a partir da descrição de todos os seus parâmetros e do estabelecimento de quais propriedades a resposta ou *solução* deve ser composta para ser considerada uma resposta válida para o problema. Uma *instância* do problema é obtida ao atribuir valores a todos os seus parâmetros [Garey and Johnson 1979].

Algoritmos são procedimentos passo-a-passo que resolvem problemas. Dado um problema, um algoritmo *resolve* tal problema se ele sempre produz uma solução para qualquer uma de suas instâncias. Um objetivo bastante comum na busca de soluções para um problema é o desenvolvimento de um algoritmo eficiente, que resolve o problema no menor tempo possível. O campo da Teoria da Complexidade Computacional busca estudar e classificar os algoritmos, identificando a quantidade de recursos computacionais necessária para executar um algoritmo. Geralmente, a eficiência de um algoritmo é definida pela quantidade de operações básicas que o mesmo demanda para resolver o problema. Esta quantidade é usualmente estabelecida em termos do tamanho da instância do problema, denotado pelo símbolo n [Arora and Barak 2009, Garey and Johnson 1979].

Uma das principais classes de problemas identificadas pela Teoria da Complexidade é a classe NP-completo, um subconjunto da classe NP, que significa *polinomial não determinístico*. A classe NP contém todos os problemas cujas soluções podem ser verificadas em tempo polinomial, enquanto a classe NP-completo é composta por problemas onde todos os problemas em NP podem ser reduzidos para eles em tempo polinomial [Garey and Johnson 1979]. Redução, nesse contexto, significa transformar um problema em outro de forma com que uma solução para o segundo problema também possa ser utilizada para resolver o primeiro [Sipser 1996]. Uma classe adicional de problemas estabelecida pela Teoria da Complexidade é a classe P, de *polinomial*; esta classe contém problemas que podem ser solucionados em tempo polinomial ou inferior. P é um subconjunto de NP.

Apesar de que as soluções para problemas NP-completos podem ser verificadas em tempo polinomial, nenhum algoritmo para resolver um problema NP-completo em tempo polinomial foi encontrado até hoje; todos demandam tempo exponencial ou superior. Apesar disso, não há nenhuma prova de que não existe um algoritmo eficiente para resolver problemas NP-completo. Esta condição é um dos principais pontos da questão "P = NP?", um dos maiores problemas abertos no campo da Ciência da Computação [Sipser 1996].

Entre os problemas pertencentes à classe NP-completo está o Problema da Soma dos Subconjuntos. Sua NP-completude foi comprovada por Richard Karp, juntamente com diversos outros problemas, em seu artigo de 1972, intitulado "Reducibility Among Combinatorial Problems" [Karp 1972]. Por ser um problema NP-completo, não se conhece um algoritmo capaz de resolvê-lo em tempo polinomial. Este trabalho se propôs a analisar este problema aplicando técnicas de processamento paralelo, com a intenção de acelerar a velocidade de busca pela solução do problema utilizando um algoritmo de força bruta, que demanda tempo exponencial, e analisar o ganho de desempenho obtido em função da quantidade de unidades de processamento executando simultaneamente.

2. Definição do problema

O Problema da Soma da Soma dos Subconjuntos, como especificado por [Garey and Johnson 1979] a partir de uma transformação do problema "Knapsack" de Richard Karp, é definido da seguinte forma: dado um conjunto $A = \{a \mid a \in \mathbb{Z}^+\}$ e um número inteiro b, existe um subconjunto $A' \subseteq A$ onde a soma de todos os elementos de A' é exatamente b?

No escopo deste trabalho, o problema será ligeiramente simplificado com a intenção de facilitar sua análise no contexto de processamento paralelo. O parâmetro b será tratado como 0, e consequentemente, os valores do conjunto A pertencerão a $\mathbb{Z}$ (isto é, poderão ser negativos ou zero) e $|A| > 0$ (ou seja, o conjunto não poderá ser vazio). Desta forma, o Problema da Soma dos Subconjuntos será tratado da seguinte forma:

$$\sum_{i=1}^{n} a_i x_i = 0 \tag{1}$$

Onde $a_i \in A$, $x_i \in \{0, 1\}$ e $\sum x_i > 0$. Um algoritmo que resolve este problema deve determinar se existe uma sequência de valores de x que soluciona a equação 1. Esta formulação é equivalente a encontrar um subconjunto não-vazio de A cuja soma é igual a zero. Utilizando técnicas de análise combinatória, é possível estabelecer que a quantidade de possíveis sequências de x é igual a $2^n - 1$, onde n representa a quantidade de elementos no conjunto. Este valor é equivalente à quantidade de subconjuntos não-vazios de um conjunto qualquer, definida por:

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1 \tag{2}$$

O algoritmo aplicado neste trabalho tentará verificar todas as combinações de x até que encontre uma combinação que solucione a equação ou até que as combinações se

esgotem. Portanto, o algoritmo irá demandar tempo exponencial, pois no pior caso, todas as $2^n - 1$ combinações deverão ser testadas.

3. Processamento paralelo

Para realizar a execução do algoritmo em mais de uma unidade de processamento simultaneamente, o conceito de *multithreading* foi aplicado. *Thread* é um conceito abstrato que descreve a sequência de execução de um programa, ou o trabalho sendo realizado pelo computador. Threads são similares à processos, porém são entidades muito mais leves, que carregam apenas informações dos registradores, da pilha e alguns outros dados, enquanto processos contém diversas outras informações, como mapas de memória, *file descriptors* e o código e dados do programa propriamente dito. Todas essas informações que compõem um processo são compartilhadas entre suas threads, cuja quantidade pode variar de apenas uma para múltiplas por processo [Lewis and Berg 1996].

Crucialmente, todas as threads de um processo podem acessar o código e os dados daquele programa. Desta forma, as threads podem executar qualquer subrotina globalmente visível no programa; o mesmo é aplicável no acesso à variáveis. Portanto, threads podem cooperar na resolução de problemas, desde que o mesmo possa ser dividido em múltiplos subproblemas independentes, para que possam ser executados paralelamente pelas threads. Com isso, desconsiderando o impacto dos procedimentos de divisão de trabalho, espera-se que um problema sendo resolvido paralelamente por n threads seja solucionado em n vezes menos tempo do que sua versão sequencial.

4. Experimentos

Para a realização dos experimentos, foi escrito um programa na linguagem C utilizando a biblioteca POSIX Threads (*pthreads*) para realizar a divisão do trabalho entre múltiplas threads. Os experimentos foram executados conforme os seguintes parâmetros:

- Quantidade de threads, com valores 1, 2 e 4;
- Tamanho do conjunto (n), variando de 4 a 52 com incrementos de 4;
- Quantidade de bits utilizados para representar os valores no conjunto, dada pela expressão $2^{0.5n}$.

Para os experimentos envolvendo múltiplas threads, um parâmetro adicional foi estabelecido para definir a quantidade de subconjuntos que cada thread deve testar ininterruptamente. Este parâmetro foi chamado de *tamanho do trabalho*, e recebeu como valores 10, 100 e 1000.

O algoritmo utilizado na realização dos experimentos em paralelo é descrito a seguir. Como entrada, o algoritmo recebe o tamanho do conjunto n, a quantidade de bits dos valores do conjunto, a quantidade de threads e o tamanho do trabalho. Inicialmente, o tempo atual é gravado para calcular a duração do algoritmo. Em seguida, um conjunto é gerado aleatoriamente com base nos parâmetros recebidos. Então, os identificadores das threads são criados para controlá-las durante a execução do programa. Na sequência, as operações que serão executadas em paralelo são especificadas.

Cada thread permanece em um *loop* que executa até que todos os subconjuntos sejam testados ou até que alguma thread encontre uma solução, realizando as operações

Algoritmo 1: Problema da Soma dos Subconjuntos em Paralelo

Entrada: $n, bits, nThreads, trabalho$
Saída : $solucao, duracao$

```
1  inicio ← tempo()
2  conjunto ← gerarConjuntoAleatorio(n, bits)
3  threads ← criarIdentificadores(nThreads)
4  foreach thread ∈ threads do
5      Paralelamente em thread
6          while houver trabalho e término não for sinalizado do
7              inicio ← obterTrabalho(trabalho)
8              for i = inicio to inicio + trabalho do
9                  if i ≥ 2^n − 1 then
10                     finalizarThread(thread, ∅)
11                 subconjunto ← gerarSubconjunto(conjunto, i)
12                 soma ← somarElementos(subconjunto)
13                 if soma = 0 then
14                     sinalizarTermino()
15                     finalizarThread(thread, subconjunto)
16         finalizarThread(thread, ∅)
17     iniciarExecucao(thread)
18 solucao ← ∅
19 foreach thread ∈ threads do
20     resultado ← aguardarExecucao(thread)
21     if resultado ≠ ∅ then
22         solucao ← resultado
23 duracao ← tempo() − inicio
24 return {solucao, duracao}
```

detalhadas a seguir. O ponto de partida do trabalho é gerado pela função *obterTrabalho*, que mantém o estado atual do trabalho, partindo do número 1 e sendo incrementado pelo parâmetro *trabalho* (o estado deve ser protegido por uma *mutex*, pois será acessado por várias threads [Lewis and Berg 1996]). Em seguida, um loop é iniciado partindo do ponto gerado pela função descrita anteriormente e executando *trabalho* vezes. Para cada iteração, um subconjunto é gerado a partir da representação em binário do estado atual (por exemplo, para $3_{10} = 111_2$, apenas os três primeiros elementos do conjunto original serão inclusos). Caso o valor do estado atual ultrapasse a quantidade de subconjuntos, a thread é finalizada com resultado vazio. Caso contrário, a soma dos elementos do subconjunto é computada e, caso seja zero, todas as threads serão sinalizadas para interromperem sua execução e a thread atual será finalizada com o subconjunto atual como resultado. Por fim, caso todos os subconjuntos tenham sido testados, a thread é finalizada com resultado vazio.

Enquanto as threads buscam a solução para o problema, o programa principal declara a solução do problema, inicialmente vazia, e fica aguardando o término das mesmas. Cada uma dessas threads conterá um resultado, indicado pelas chamadas da função *finalizarThread*. Caso o resultado de uma thread seja vazio, a solução fica inalterada; caso contrário, a solução recebe o subconjunto que a thread encontrou. Quando todas as threads forem finalizadas, o algoritmo é terminado e retorna a solução encontrada (ou o conjunto vazio se não houver solução) e a duração do procedimento, que é calculada a partir da diferença entre o tempo atual e o tempo inicial.

A versão sequencial do algoritmo é bastante similar à versão paralela, exceto pela ausência das threads. Desta forma, não é necessário criar identificadores, separar o trabalho em blocos, nem iniciar, sincronizar e aguardar o término das threads. O valor cuja representação em binário indica o subconjunto a ser gerado é simplesmente inicializado em 1 e incrementado cada vez que a solução não é encontrada, até atingir $2^n - 1$.

Os experimentos foram realizados em um computador com processador Intel Core i3-3240 com clock de 3.4 GHz e 4 núcleos e com o sistema operacional Windows 10 Pro 64 bits. O programa foi compilado utilizando GCC versão 4.8.1 no ambiente MinGW. Um aspecto importante para melhorar a qualidade dos dados foi estabelecer a mesma *seed* para o gerador de números aleatórios para diferentes testes. Cada teste foi composto por 30 execuções do algoritmo para cada tamanho de conjunto.

5. Análise dos resultados

O primeiro passo tomado para analisar os resultados obtidos foi investigar as diferenças entre os diferentes tamanho de trabalho para 2 e 4 threads, observando qual tamanho de trabalho oferece o melhor desempenho, medido pela média de tempo entre todos os tamanhos de conjunto. Através do gráfico da figura 1 foi possível perceber que a diferença de desempenho é bastante pequena, mas o tamanho de trabalho 1000 se mostrou superior aos outros em ambos os casos.

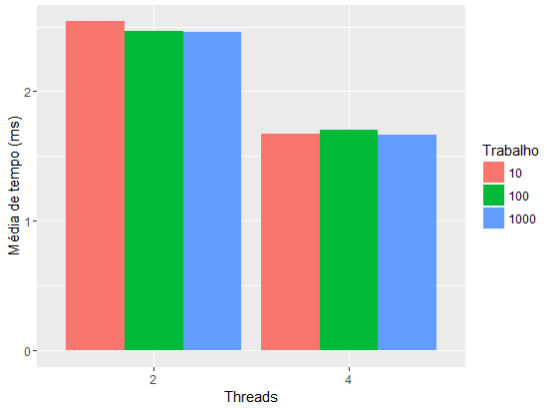

Figura 1. Média de tempo para diferentes tamanhos de trabalho

Com isso, o tamanho de trabalho 1000 será utilizado nas análises a seguir. O próximo passo efetuado foi a realização da principal proposta deste trabalho: análisar o ganho de desempenho em função da quantidade de threads utilizadas. O gráfico da figura 2 apresenta as médias de tempo que o algoritmo demandou para solucionar o problema

em função do tamanho do conjunto, para 1, 2 e 4 threads. É possível perceber um ganho significativo de performance, especialmente quando $n \geq 44$?.

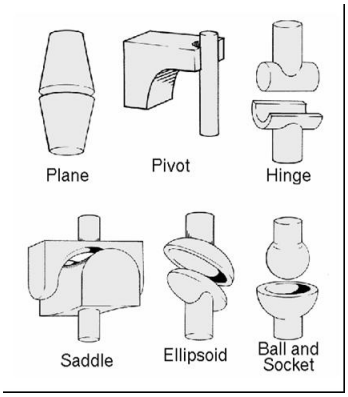

Figura 2. This figure is not good at all

Para visualizar detalhadamente o ganho de performance, os dados foram dispostos de forma tabular na tabela 1. As durações apresentadas foram calculadas a partir da média de duração de todas as 30 iterações.

Tabela 1. Variables to be considered on the evaluation of interaction techniques

	Value 1	Value 2
Case 1	1.0 ± 0.1	$1.75 \times 10^{-5} \pm 5 \times 10^{-7}$
Case 2	0.003(1)	100.0

Através da tabela 1 é possível observar uma melhora de 10 vezes para duas threads e 20 vezes para quatro threads quando $n = 44$, 11 vezes para duas e 22 vezes para quatro quando $n = 48$ e por fim 12 vezes para duas e 23 vezes para quatro quando $n = 52$. Portanto, em média, o tempo de execução para duas threads 11 vezes mais veloz, enquanto para quatro threads a melhora é de 22 vezes.

6. Conclusões

Concluímos que threads são boas.

Referências

Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA.

Lewis, B. and Berg, D. (1996). *Threads Primer: A Guide to Multithreaded Programming*.

Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition.