Laboratório 3 (CSC-33)

Threads

Antônio Gustavo Silveira Dantas
Lucca Rangel Haddad

10 de junho de 2024

Engenharia de Computação

Professor(a): Cecilia de Azevedo Castro Cesar

Instituto Tecnológio de Aeronáutica - ITA
Departamento de Sistemas de Computação

1 Introdução

O problema escolhido para este estudo é o *Subset Sum Problem*, que consiste em determinar se um conjunto de números possui algum subconjunto cuja soma é igual a um valor especificado previamente. Esta questão é um desafio clássico de otimização e combinação que se torna computacionalmente intensivo conforme o tamanho do conjunto aumenta.

A aplicação de técnicas de computação paralela, através do uso de múltiplas threads, é particularmente eficaz para este problema, pois permite a divisão do espaço de busca em subespaços menores e independentes. Cada thread pode então explorar um segmento do problema em paralelo, potencialmente reduzindo significativamente o tempo total de execução ao utilizar os múltiplos processadores disponíveis no sistema. Esta abordagem não apenas acelera a computação, mas também torna viável a resolução de instâncias maiores do problema que seriam praticamente impossíveis para um algoritmo sequencial.

2 Solução e Implementação

2.1 Descrição do Problema

O problema da Soma do Subconjunto (*Subset Sum Problem*) consiste na verificação da existência de um subconjunto de um determinado conjunto original cuja soma de seus elementos seja igual a um valor requisitado.

Embora tenha lógica simples, o problema apresenta complexidade exponencial, visto que, para um conjunto qualquer de tamanho n, há 2^n subconjuntos possíveis, fator que torna a verificação de subsomas computacionalmente extensa.

2.2 Arquitetura de Solução

2.2.1 Algoritmo Sequencial

```cpp
bool subsetSumSequential(const std::vector<int>& array, int target) {
    int n = array.size();
    unsigned int numberOfSubsets = 1 << n;
    for(unsigned int i = 0; i < numberOfSubsets; i++) {
        int sum = 0;
        for(int j = 0; j < n; j++) {
            if(i & (1 << j))
                sum += array[j];
        }
        if(sum == target) return true;
    }
    return false;
}

```

Listing 1: Algoritmo Sequencial em C++

No algoritmo sequencial de resolução do problema do subconjunto de soma, exploramos todos os possíveis subconjuntos de um conjunto dado. A verificação é realizada através de um loop que itera desde 0 até $2^n - 1$, onde n é o número de elementos no conjunto.

Utilizamos uma técnica conhecida como bitmasking para determinar quais elementos estão presentes em cada subconjunto gerado durante a iteração. Para cada subconjunto, somamos os elementos selecionados pela bitmask. Se a soma desses elementos corresponder ao valor alvo especificado, o processo é interrompido imediatamente, indicando que uma solução foi encontrada.

Esse método é direto, mas pode tornar-se computacionalmente intensivo à medida que o tamanho do conjunto aumenta, devido à necessidade de examinar todos os subconjuntos possíveis.

2.2.2 Algoritmo Paralelo

```
void findSubset(const std::vector<int>& array, unsigned int start,
    unsigned int end, int target, std::atomic<bool>& found) {
     int n = array.size();
     for(unsigned int i  = start; i < end && !found.load(); i++) {
         int sum = 0;
         for(int j = 0; j < n; j++) {
             if(i & (1 << j))
                 sum += array[j];
         }
         if(sum == target) {
             found.store(true);
             return;
         }
     }
}

```

Listing 2: Algoritmo Auxiliar

```
bool subsetSumParallel(const std::vector<int>& array, int target, int
    numberOfThreads) {
     int n = array.size();
     unsigned int numberOfSubsets = 1 << n;
     std::vector<std::thread> threads;
     std::atomic<bool> found(false);

     unsigned int segmentSize = numberOfSubsets / numberOfThreads;
     for(int i = 0; i < numberOfThreads; i++) {
         unsigned int start = i * segmentSize;
         unsigned int end = (i == numberOfThreads - 1) ? numberOfSubsets
    : start + segmentSize;
         threads.emplace_back(findSubset, std::cref(array), start, end,
    target, std::ref(found));
     }

     for(auto& thread : threads) {
         thread.join();
     }

     return found.load();
}

```

Listing 3: Algoritmo Paralelo em C++

No algoritmo paralelo, a tarefa de verificar subconjuntos é dividida entre várias threads, permitindo que múltiplas partes do espaço de busca sejam exploradas simultaneamente. Assim como na abordagem sequencial, utilizamos bitmasking para identificar os elementos de cada subconjunto. No entanto, ao invés de um único loop, o espaço de busca total, que ainda varia de 0 a $2^n - 1$, é segmentado em partes iguais de acordo com o número de threads disponíveis.

Cada thread é responsável por uma porção específica desse espaço, verificando independentemente os subconjuntos dentro de seu segmento. A comunicação entre threads é crucial: se uma thread encontrar uma solução — isto é, um subconjunto cuja soma dos elementos é igual ao valor alvo — ela deve notificar as outras threads para interromperem a busca. Isso é efetivamente realizado usando uma variável compartilhada $std :: atomic < bool >$, que todas as threads verificam regularmente. Uma vez que essa variável é marcada como verdadeira, todas as threads cessam suas operações.

Esse método paralelo pode reduzir significativamente o tempo necessário para encontrar uma solução em comparação com o método sequencial, especialmente quando o conjunto de dados é grande e há múltiplos núcleos de processamento disponíveis. No entanto, a eficiência dessa abordagem depende de uma distribuição equilibrada da carga de trabalho entre as threads e de uma gestão eficaz dos recursos de sincronização.

3 Análises

3.1 Metodologia de Teste

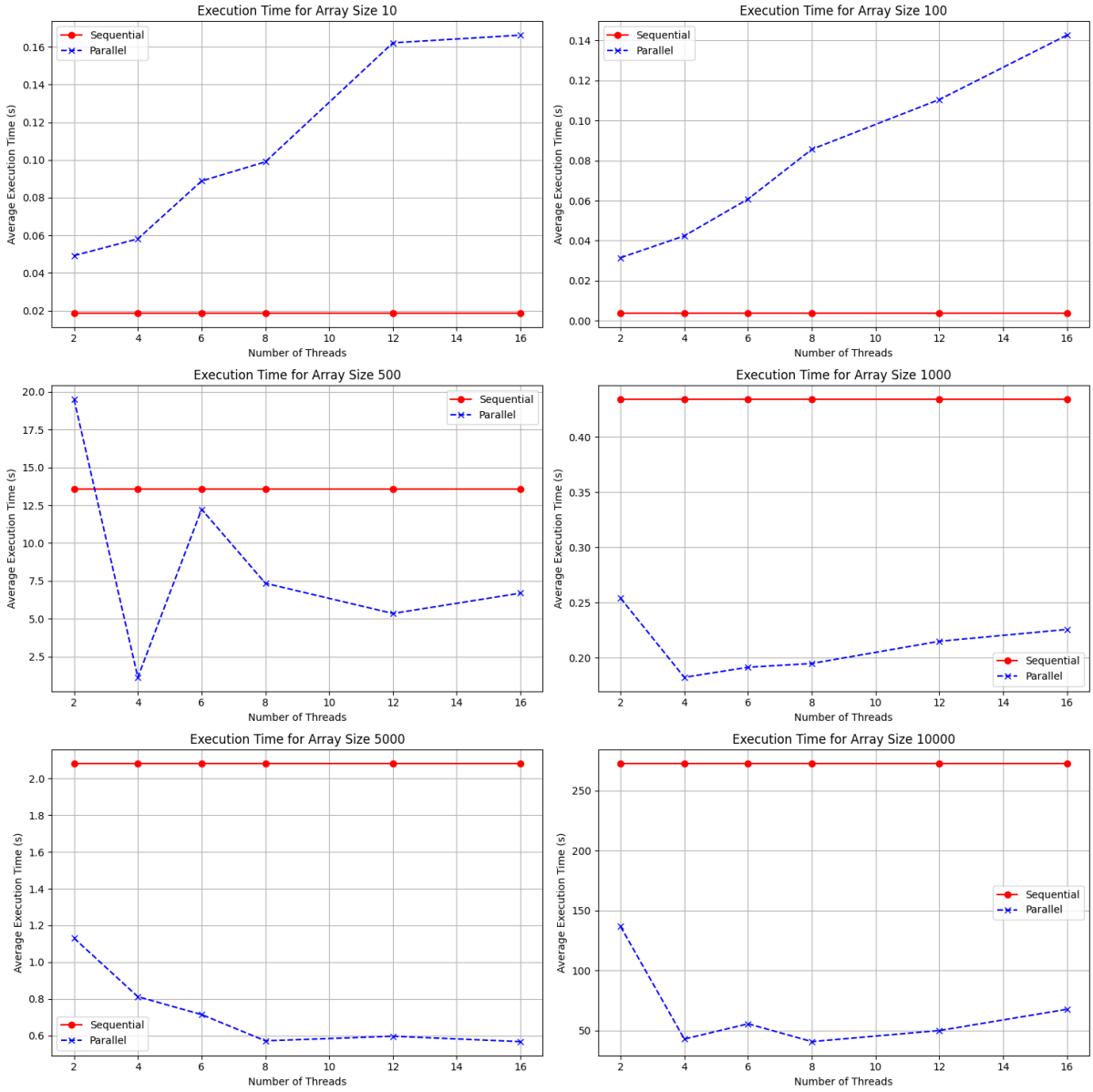
Para avaliar a eficiência dos algoritmos de soma de subconjunto implementados (tanto na versão sequencial quanto na paralela), adotamos uma metodologia de teste caracterizada por:

- **Variedade de Tamanhos de Conjuntos**: Testou-se os algoritmos com conjuntos de dados de diversos tamanhos, especificamente 10, 100, 500, 1000, 5000 e 10000 elementos, para investigar como o tamanho do conjunto afeta o desempenho dos algoritmos.
- **Diversidade de Configurações de Threads**: Para a versão paralela, explorou-se o impacto de diferentes quantidades de threads no desempenho, utilizando configurações de 2, 4, 6, 8, 12 e 16 threads.
- **Repetibilidade dos Testes**: Cada configuração foi testada 50 vezes para assegurar a robustez estatística dos resultados e minimizar as variações devido a fatores aleatórios.

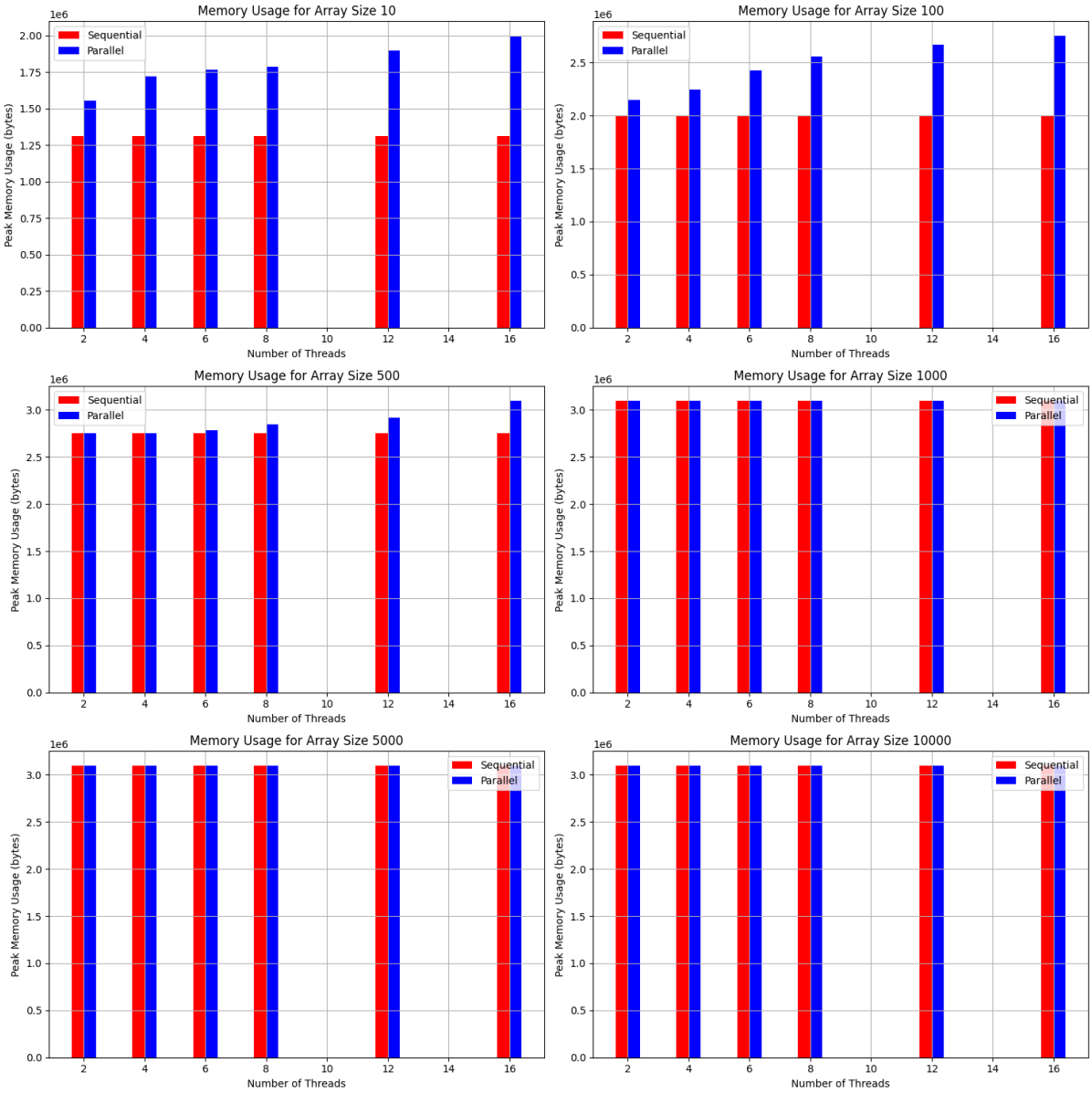
Os dados para os testes foram gerados aleatoriamente dentro de um intervalo de -50 a +50, garantindo uma ampla variação nos conjuntos testados. Para cada vetor, um número aleatório de seus elementos foi selecionado para formar um subconjunto, assegurando a existência de pelo menos uma solução válida para o problema do subconjunto de soma.

3.2 Resultados Obtidos

3.2.1 Gráfico de Desempenho



3.2.2 Análise de Memória



3.3 Interpretação dos Resultados

3.3.1 Tempo de Execução

Para arrays pequenos (tamanho menor que 100, por exemplo), o uso de threads dificulta a resolução do problema, o que provoca um aumento no tempo de execução em relação ao algoritmo sequencial. Em contrapartida, para vetores com tamanho a partir de 500, a paralelização provoca reduções significativas no tempo de execução do programa.

Percebe-se também que há um limite de paralelização de funções executadas, visto que a partição ótima de tarefas ocorre para distribuições entre 4 e 8 núcleos e, a partir disso, a paralelização não provoca mais incrementos de performance, podendo até causar aumento do tempo de execução devido à dificuldade de gestão de maior número de threads.

3.3.2 Uso de Memória

O aumento no uso de memória no algoritmo paralelo, proporcional ao número de threads, é causado pelo overhead de gerenciamento dessas threads e pela necessidade de recursos extras para a sincronização e o gerenciamento de dados entre elas. Para conjuntos de dados pequenos, o tempo requerido para a inicialização e sincronização das threads frequentemente supera o tempo de processamento do trabalho propriamente dito, tornando a paralelização ineficaz. Além disso, conforme a Lei de Amdahl, a eficácia do paralelismo é limitada pela parte do programa que precisa ser executada de forma sequencial, o que tem um impacto mais significativo em conjuntos menores e reduz o ganho máximo de velocidade possível.

4 Conclusão

A implementação paralela do problema do subconjunto de soma provou ser uma estratégia eficaz para aprimorar o desempenho de tarefas computacionalmente exigentes. A capacidade de dividir o trabalho em múltiplas threads significativamente reduziu o tempo necessário para encontrar soluções, reforçando a eficácia da computação paralela em problemas de otimização combinatorial.

Embora a paralelização tenha trazido melhorias notáveis em eficiência, especialmente para problemas de maior escala, é importante notar que para conjuntos pequenos (com tamanho inferior a 100), a paralelização pode, de fato, diminuir a performance. Isso ocorre porque o overhead associado à criação e gerenciamento de threads, bem como à coordenação da memória, pode não justificar a substituição do algoritmo sequencial.

Além disso, a performance da paralelização é limitada pelo número de núcleos do processador. Portanto, aumentar o número de threads além do número de núcleos disponíveis tende a não resultar em ganhos adicionais de desempenho, levando a uma estabilização no desempenho à medida que o número de threads aumenta.

5 Link da Implementação

Repositório do Github