Lab 1 – Soma de Prefixos com Pthreads

CI1316 – Programação Paralela (2º semestre/2025)

Luíza Diapp – GRR20221252
João Marcelo Caboclo – GRR20221227
Professor: W. Zola

03 de Outubro de 2025

1 Introdução

O objetivo deste trabalho foi desenvolver uma implementação paralela do algoritmo de **soma de prefixos** (*prefix sum*) sobre vetores de inteiros de 64 bits. O desafio principal foi otimizar a solução para execução em **CPUs multicore** utilizando **PThreads** e **pool de threads**.

Um *pool de threads* consiste em criar um conjunto fixo de threads que permanecem ativas durante toda a execução. As threads recebem tarefas para executar, evitando a criação e destruição repetida de threads — operações custosas em tempo. No contexto deste laboratório, o pool permite dividir o vetor em **faixas de elementos** que são processadas em paralelo, sincronizando-se apenas uma vez por meio de uma barreira.

A versão implementada segue a especificação **v1.2**, com as seguintes características:

- Execução **in place**, usando apenas um vetor auxiliar `PartialSum` com uma célula por thread.
- Divisão equilibrada do trabalho entre as threads.
- Sincronização mínima, com apenas **uma barreira**.

2 Implementação

2.1 Visão geral

O algoritmo realiza a soma de prefixos *in place* com **pool de threads** e apenas uma barreira. Cada thread processa uma faixa do vetor em duas fases:

1. **Antes da barreira:** calcula a soma parcial da sua faixa e escreve em `partialSum[tid]`;
2. **Após a barreira:** calcula seu deslocamento (`myPrefixSum`) a partir das somas parciais das threads anteriores e reescreve sua faixa com os valores finais.

2.2 Estrutura de contexto

A estrutura **ThreadCtxPS** reúne os parâmetros de cada thread: identificador (`tid`), limites da faixa (`begin` e `end`), ponteiro para o vetor global e número de threads. Isso organiza as informações de forma clara e evita dependência de variáveis globais adicionais.

2.3 Particionamento

O particionamento divide o vetor em faixas contíguas e equilibradas. Quando a divisão não é exata, os elementos excedentes são distribuídos entre as primeiras threads, garantindo bom balanceamento de carga e localidade de memória.

2.4 Worker das threads

O **ps_worker** executa as duas fases descritas: primeiro calcula a soma parcial da faixa, depois da barreira computa o deslocamento e reescreve sua parte do vetor. Cada thread escreve apenas na sua própria faixa, evitando disputas, e o uso de apenas uma barreira reduz o overhead de sincronização.

2.5 Função principal

A **ParallelPrefixSumPth** inicializa a barreira, cria o conjunto de threads com seus contextos, aguarda a conclusão (*join*) e finaliza a barreira. Dessa forma, todo o fluxo do algoritmo fica encapsulado dentro das threads, mantendo simplicidade e eficiência.

Topologia (lstopo)

```
Machine (7830MB total)
  Package L#0
    NUMANode L#0 (P#0 7830MB)
    L3 L#0 (8192KB)
      L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#4)
      L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#5)
      L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
        PU L#4 (P#2)
        PU L#5 (P#6)
      L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
        PU L#6 (P#3)
        PU L#7 (P#7)
```

3 Experimentos e Metodologia

- Vetor inicial preenchido com valores aleatórios (`rand() % 10`).
- Teste com 8M elementos.

- Execução com 1 a 8 threads, 10 vezes cada, via script `roda-v1.sh`.
- Tempo médio calculado com `chrono.c`, descontando `memcpy`.

4 Resultados do teste com 8M

Nos experimentos com **8 milhões de elementos**, o tempo médio de execução com uma única thread foi de aproximadamente 14,3 segundos. Ao aumentar para duas threads, o tempo caiu para cerca de 10 segundos, representando um ganho expressivo de desempenho. A partir de três e quatro threads, os tempos estabilizaram em torno de 9,5 segundos, mantendo-se semelhantes até oito threads. Isso mostra que o paralelismo trouxe ganhos claros em relação à execução sequencial, mas a escalabilidade não foi linear: a velocidade praticamente saturou a partir de quatro threads. Esse comportamento pode ser explicado pelo gargalo de memória e pelo limite da largura de banda, já que cada thread continua acessando intensivamente o vetor global. Ainda assim, a redução de tempo em relação à versão sequencial demonstra que o uso de múltiplos núcleos foi eficaz para vetores grandes, consolidando a eficiência do algoritmo com apenas uma barreira de sincronização.

5 Gráficos Obtidos com base nos resultados:

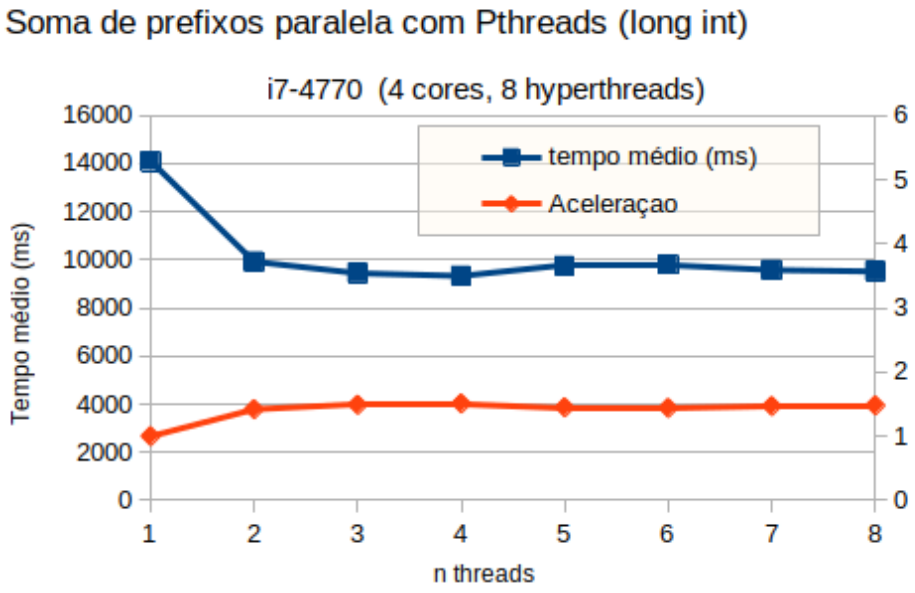

Figura 1: Aceleração e Tempo Médio por thread

Os gráficos de **tempo médio** e **vazão média** em função do número de threads evidenciam um ganho de desempenho significativo ao paralelizar o algoritmo de soma de prefixos, principalmente na transição de 1 para 2 threads. A partir de 3 ou 4 threads, tanto a vazão quanto a aceleração se estabilizam, indicando que a aplicação atingiu um limite prático de escalabilidade.

Esse comportamento está alinhado com o número de núcleos físicos do processador (4 cores): acima desse ponto, o uso de *hyperthreading* não traz ganhos adicionais expressivos, e o gargalo de memória passa a dominar. A vazão média atinge seu pico próximo de 55

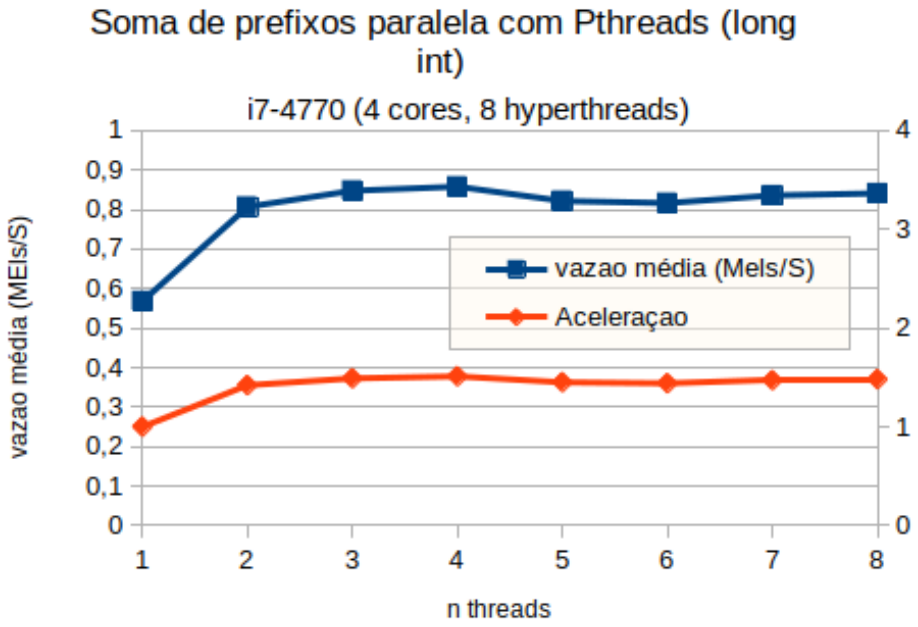

Figura 2: Vazão média e Aceleração por thread

Mels/s com quatro threads, mantendo-se quase constante até oito threads, enquanto a aceleração máxima fica em torno de 2,2×. Isso mostra que a implementação paralela aproveita bem o paralelismo disponível, mas é limitada por fatores de hardware e acesso à memória.

6 Conclusão

A implementação paralela de soma de prefixos com **PThreads** e **pool de threads** obteve ganhos significativos em vetores grandes. A estratégia de usar apenas uma barreira e o vetor auxiliar `PartialSum` se mostrou eficiente. O limite de desempenho pode estar ligado ao gargalo de memória e ao custo da barreira.