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 presente trabalho tem como objetivo implementar e analisar o desempenho de um algoritmo paralelo para o cálculo da soma de prefixos utilizando **PThreads**. O foco está na utilização de um modelo de *pool de threads*, em que um conjunto fixo de threads coopera na execução de um vetor global, buscando equilibrar a carga de trabalho e reduzir o tempo de execução.

A implementação foi desenvolvida em linguagem C, utilizando três barreiras de sincronização para coordenar o início, a transição entre as fases e o término da execução. O estudo busca avaliar o impacto do paralelismo no desempenho, comparando os resultados obtidos com diferentes quantidades de threads, de modo a observar o comportamento de escalabilidade e os limites impostos pelo hardware. Essa análise permite compreender como a divisão de tarefas e a sincronização entre threads influenciam diretamente o ganho de desempenho e a eficiência da solução paralela.

2 Implementação do Algoritmo Paralelo

A versão paralela do algoritmo de soma de prefixos foi desenvolvida com base em um modelo de pool de threads (piscina de *threads*), onde um grupo fixo de *threads* opera de forma cooperativa sobre um vetor global `Vector`. O objetivo principal é garantir um paralelismo equilibrado e seguro, coordenado por três barreiras de sincronização que definem o início, a transição entre as fases e o término conjunto da execução.

2.1 Visão Geral e Estrutura de Duas Fases

O núcleo do processamento paralelo reside na função `ps_worker`, executada por todas as *threads* do pool. A divisão do trabalho é feita pela função `ps_compute_chunk`, que calcula o intervalo contíguo de índices do vetor `Vector` que cada *thread* deve processar, garantindo uma distribuição justa da carga.

O algoritmo se desenrola em duas fases principais, essenciais para a corretude da soma de prefixos paralela:

1. Fase 1: Cálculo das Somas Parciais. Cada *thread* trabalha de forma independente, somando apenas os elementos de sua fatia local do vetor `Vector`. O resultado, a

soma parcial da fatia, é armazenado em um outro vetor global (`partialSum`). Uma barreira de sincronização é crucial aqui para garantir que todas as *threads* completem seus cálculos e atualizem o vetor `partialSum` antes que qualquer uma prossiga.

2. Fase 2: Aplicação do Deslocamento Global (Prefixos). Após a barreira, o vetor `partialSum` está completo e pode ser lido. Cada *thread* calcula o deslocamento (prefixo) que deve ser aplicado à sua fatia. Este deslocamento é a soma de todos os `partialSum` das *threads* que a precedem. Em seguida, a *thread* percorre sua fatia de `Vector` e aplica o cálculo da soma de prefixos localmente, adicionando o deslocamento acumulado. Uma última barreira assegura que todas as atualizações *in-place* no vetor principal sejam concluídas antes do retorno.

2.2 Orquestração e Configuração do Pool

A função `ParallelPrefixSumPth` é o ponto de controle do processo. Ela define o número de *threads* (`ps_nThreads`), inicializa as três barreiras (`ps_poolBarrier`, `ps_algoBarrier`, `ps_doneBarrier`) e cria as *threads* secundárias em modo *"detached"* para que seus recursos sejam liberados automaticamente e para que seja evitada a utilização de `thread joins` .

É crucial destacar que a thread principal (Main) também atua como a thread de índice 0 do pool, executando a função `ps_worker`. Essa abordagem garante que todas as participantes sejam igualmente sincronizadas pelas barreiras, simplificando a coordenação e permitindo a correta destruição das barreiras ao final.

2.3 Particionamento Balanceado do Vetor

A função auxiliar `ps_compute_chunk` é responsável por dividir o vetor total, de tamanho N, em P fatias contíguas e balanceadas. Ela calcula o tamanho base de cada fatia e, caso N não seja múltiplo de P (ou seja, existe um resto), distribui os elementos extras um por um para as primeiras *threads*.

Essa estratégia de divisão contígua é fundamental para o desempenho, pois maximiza a localidade de referência, permitindo que cada *thread* trabalhe primariamente em sua própria região de memória cache.

2.4 Barreiras de Sincronização

As três barreiras são o mecanismo chave de controle do pool:

- `ps_poolBarrier` (B1): Garante que todas as *threads* estejam prontas e alinhadas para iniciar o processamento da Fase 1 ao mesmo tempo.
- `ps_algoBarrier` (B2): Separa a Fase 1 da Fase 2. Garante que o vetor `partialSum` esteja completamente escrito por todas as *threads* antes que qualquer uma comece a lê-lo para calcular o deslocamento global.
- `ps_doneBarrier` (B3): Garante que todas as *threads* tenham concluído a Fase 2 (atualização final do vetor) antes que a função `ParallelPrefixSumPth` retorne e as estruturas sejam destruídas.

2.5 Corretude e Propriedades de Segurança

A corretude do algoritmo é garantida por um princípio de exclusividade de escrita: cada *thread* somente escreve em sua própria fatia do vetor principal (`Vector`) e em seu índice correspondente no vetor de somas parciais (`partialSum`). O acesso compartilhado ocorre apenas por meio de leitura. Como as leituras críticas são protegidas pelas barreiras (a leitura de `partialSum` só ocorre após a escrita completa da Fase 1, garantida pela B2), condições de corrida são eliminadas, assegurando a consistência e o resultado correto da soma de prefixos.

3 Propriedades da máquina que rodou o programa

Os programa foi rodado na máquina j9 do dinf, a qual tem a seguinte 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)
```

4 Experimentos e Metodologia

- Vetor inicial preenchido com valores aleatórios.
- 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`.

5 Resultados do Teste com 8 Milhões de Elementos

Nos experimentos com um vetor de 8 milhões de elementos, com 1 thread levou em média 11,26 segundos. O uso do paralelismo trouxe uma melhoria imediata: com 2 threads, o tempo caiu para 9,54 segundos. O pico de desempenho foi alcançado com 4 threads, registrando 9,18 segundos. No entanto, o ganho de velocidade saturou rapidamente; o tempo de execução permaneceu estabilizado, variando entre 9,3 e 9,47 segundos, mesmo com o

aumento de threads até o limite de 8. Esta saturação indica que o gargalo do sistema não é mais a capacidade de processamento, mas sim a largura de banda da memória. Devido à natureza do algoritmo de soma de prefixos, que exige acesso e atualização intensiva do vetor global por todas as threads, o limite de transferência de dados da memória principal anula os benefícios de adicionar mais núcleos. Apesar disso, a redução no tempo de execução em relação à versão sequencial valida a eficácia do algoritmo paralelo.

6 Gráficos Obtidos com base nos resultados:

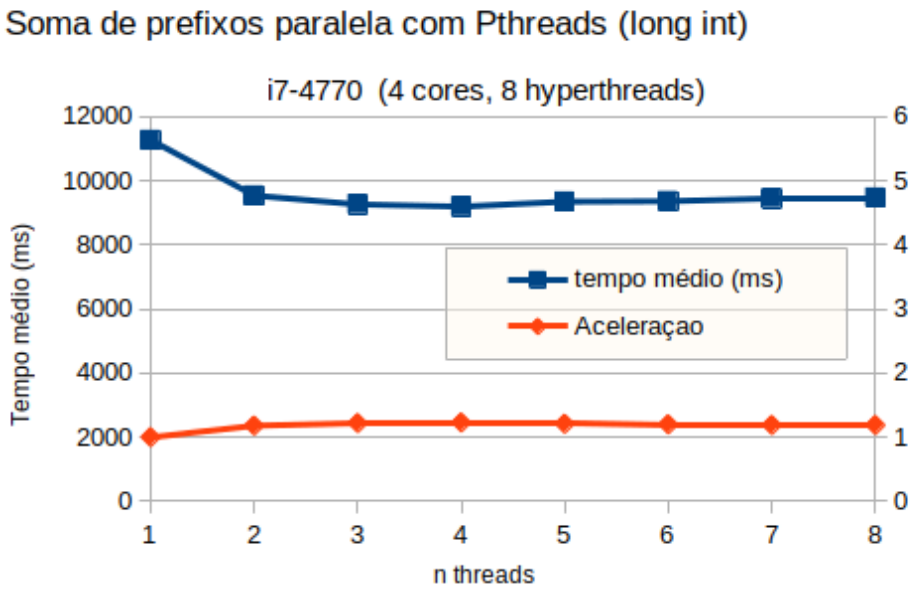

Figura 1: Aceleração e Tempo Médio por thread

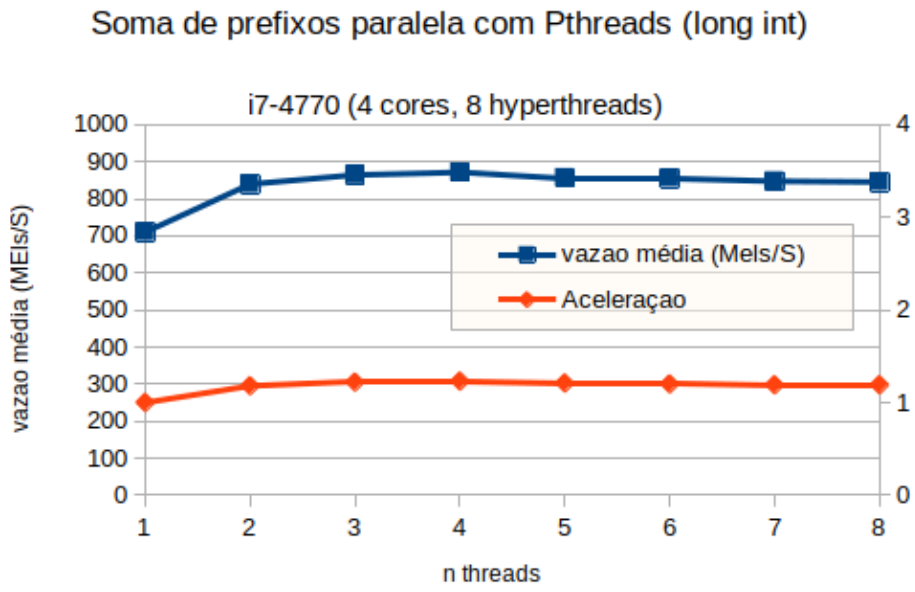

Figura 2: Vazão média e Aceleração por thread

Os gráficos de tempo médio, vazão média e aceleração mostram que o desempenho da soma de prefixos paralela melhora até cerca de quatro threads. O tempo médio reduz acentuadamente ao passar de uma para duas threads, e a partir de três ou quatro threads a redução se estabiliza, indicando o limite prático de escalabilidade. De maneira semelhante, a vazão cresce até atingir seu pico com quatro threads e permanece praticamente constante até oito, o que evidencia o aproveitamento eficiente do paralelismo disponível.

A aceleração também segue esse comportamento, aumentando rapidamente nas primeiras execuções paralelas e estabilizando após quatro threads. Esse resultado reflete o equilíbrio entre o ganho de processamento e o custo adicional de sincronização e acesso à memória. O comportamento observado está em conformidade com as características do processador i7-4770, que possui quatro núcleos físicos e oito lógicos por *hyperthreading*. Acima desse ponto, o aumento de threads não gera ganhos expressivos, pois o gargalo passa a ser o subsistema de memória e não a quantidade de núcleos. Assim, a implementação paralela obtém boa eficiência até o limite físico do hardware, com aceleração consistente e desempenho estável nas configurações com maior número de threads.

7 Conclusão

A implementação paralela de soma de prefixos com **PThreads** e **pool de threads** obteve ganhos 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.