Relatório: Avaliação da Escalabilidade com Afinidade de Threads em Simulação Numérica

Aluno: Cristovão Lacerda Cronje

1. Introdução

Nesta atividade, foi realizada uma simulação numérica baseada na forma simplificada da equação de Navier-Stokes tridimensional. A implementação utilizou paralelismo via OpenMP, e o foco foi avaliar como diferentes configurações de **afinidade de threads** impactam no desempenho da simulação. O experimento foi executado no **nó da partição amd-512** do **NPAD (Núcleo de Processamento de Alto Desempenho)**, com **128 CPUs por tarefa** e tempo de execução limitado a **20 minutos**.

2. Objetivos

- Avaliar o impacto da configuração de afinidade de threads no desempenho paralelo.
- Comparar a escalabilidade do código ao variar o número de threads (32, 64, 128) sob diferentes políticas de afinidade.
- Compreender o comportamento do sistema NUMA utilizado (memória não uniforme) e o papel da afinidade na utilização de caches e acesso à memória.

3. Metodologia

- Código paralelo implementado com **#pragma omp parallel for collapse(3) proc_bind(…) schedule(static).**
- Malhas simuladas: 100×100×100, 200×200×200, 300×300×300.
- Métricas coletadas por passo de tempo: tempo de execução e valor central da malha.
- Afinidades testadas: f**alse, true, close, spread, master.**
- Execuções realizadas em um nó com –cpus-per-task=128.

Script de Submissão (SLURM):

```
#SBATCH --time=0-0:10
#SBATCH --partition=amd-512
#SBATCH --cpus-per-task=128
#SBATCH --job-name=013_tarefa_job
#SBATCH --output=013_tarefa_%j.out
#SBATCH --error=013_tarefa_%j.err

module load compilers/gnu/14.2.0

# Exporta variáveis para OpenMP
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_PLACES=cores

# Compila o código
gcc -fopenmp 013_tarefa.c -o 013_tarefa -lm

# Executa a simulação
./013_tarefa
```

3.1 Cláusulas de Afinidade (proc_bind)

A configuração de afinidade foi feita via diretiva OpenMP proc_bind(...) e variável de ambiente OMP_PROC_BIND. Abaixo, a descrição das políticas testadas e sua relação com uso de cache e memória:

- **false:**
  Desativa a afinidade explícita. As threads podem ser migradas livremente entre núcleos, o que permite balanceamento de carga, mas pode causar perda de localidade de dados, *cache flushes*[*2], e maior penalidade em sistemas NUMA[*1].
- **true:**
  Ativa afinidade automática. As threads tendem a permanecer nos núcleos onde foram criadas, reduzindo migração e melhorando o uso de cache. Ainda assim, não há controle preciso sobre agrupamento ou dispersão.
- **close:**
  Agrupa threads em núcleos próximos, otimizando o uso de caches compartilhados (L2, L3). Ideal quando há comunicação frequente entre threads ou reutilização de dados em cache[*3]. Pode levar a **thrashing**[*4] se muitas threads competem por cache L1/L2 nos mesmos núcleos.
- **spread:**
  Distribui as threads o mais distante possível entre os núcleos disponíveis, evitando competição por caches L1/L2 e reduzindo **thrashing**. É benéfico para códigos com pouca comunicação entre threads e acesso independente à memória.
- **master:**
  Agrupa threads em torno do núcleo principal (onde a thread mestre está). Útil em aplicações com forte interação entre threads e a thread mestre. Pode causar gargalo de cache se o número de threads for elevado.

3.2 Controle via OMP_PLACES

Utilizado OMP_PLACES=cores para mapear explicitamente as threads a núcleos físicos, garantindo granularidade no controle de alocação de threads e reduzindo ambiguidades do escalonador do sistema.

Notas Técnicas

[*1] **NUMA (Non-Uniform Memory Access):** Arquitetura de memória em que núcleos de processadores possuem regiões de memória local. Acesso remoto (a memórias de outros nós) é mais lento, impactando negativamente o desempenho se a thread for migrada.

[*2] **Cache Flushes:** Ocorrem quando uma thread muda de núcleo e o conteúdo do cache precisa ser invalidado ou recarregado. Isso afeta o desempenho ao aumentar o tempo de acesso aos dados.

[*3] **Perda de Localidade de Dados:** Quando uma thread deixa de acessar dados que estavam próximos (em cache), os dados precisam ser buscados novamente na RAM, aumentando a latência. Políticas de afinidade ajudam a minimizar esse efeito.

[*4] **Thrashing:** Situação em que múltiplas threads competem pelo mesmo espaço de cache, causando substituições frequentes e reduzindo a eficiência. Ocorre principalmente quando muitas threads compartilham o mesmo núcleo ou socket sem considerar a hierarquia de memória cache.

4. Resultados e Análise

4.1 Tabelas de Tempo de Execução (em segundos)

Malha 100x100x100:

Afinidade	32 threads	64 threads	128 threads	Speedup
false	1.12	0.70	0.60	1.87
true	1.12	0.70	0.60	1.87
close	1.12	0.70	0.60	1.87
spread	1.12	0.70	0.61	1.86
master	1.12	0.70	0.60	1.86

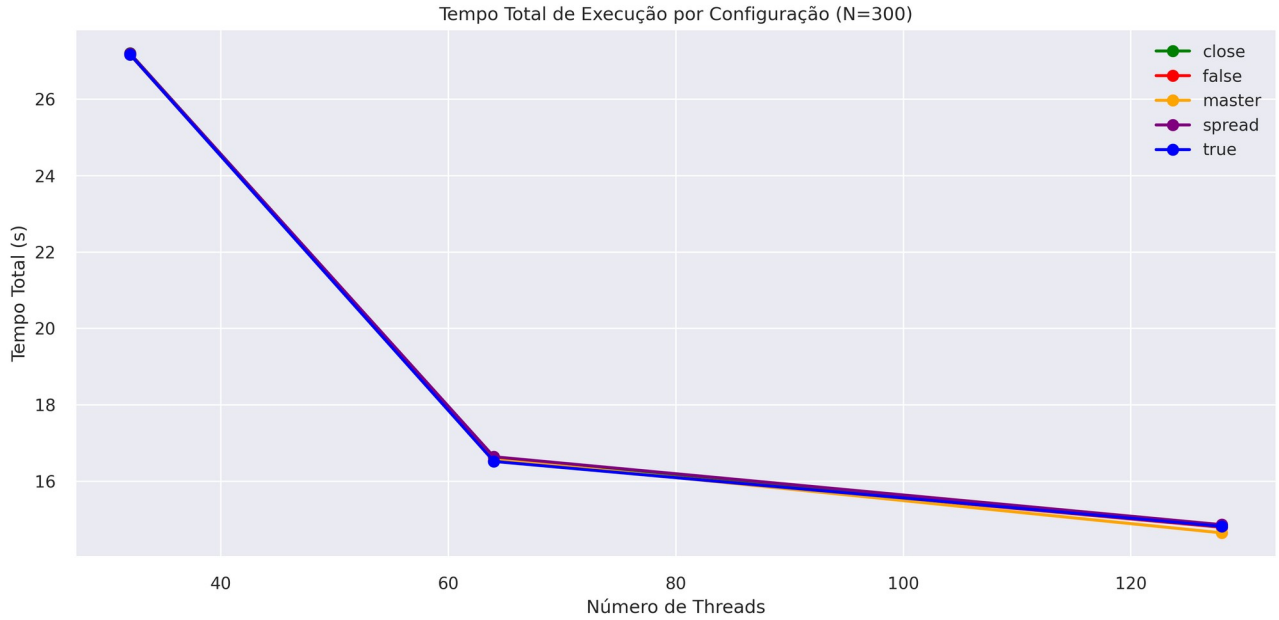
Malha 200x200x200:

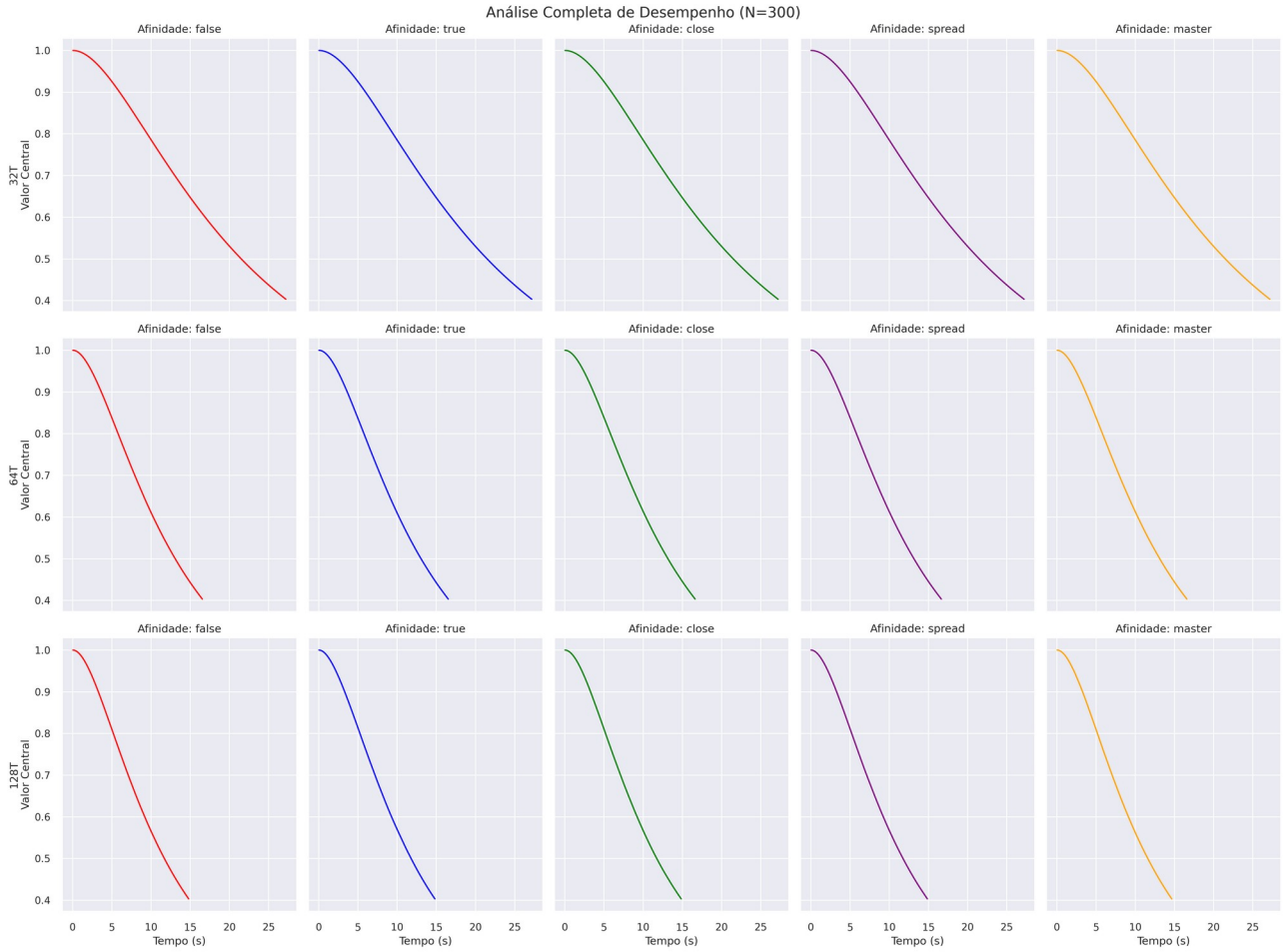
Afinidade	32 threads	64 threads	128 threads	Speedup
false	8.36	5.00	4.46	1.87
true	8.37	5.00	4.46	1.87
close	8.38	5.01	4.46	1.88
spread	8.40	5.04	4.46	1.87
master	8.41	5.04	4.47	1.87

Malha 300x300x300:

Afinidade	32 threads	64 threads	128 threads	Speedup
false	27.20	16.54	14.79	1.84
true	27.18	16.52	14.82	1.84
close	27.21	16.62	14.85	1.83
spread	27.20	16.64	14.86	1.83
master	27.17	16.58	14.65	1.86

Gráficos – Tempo de Execução por Afinidade (malha 300x300x300):





4.2 Análise Qualitativa

Os resultados com diferentes configurações de afinidade de threads (proc_bind) indicam impacto limitado no desempenho, o que pode ser atribuído a dois fatores principais:

- O escalonador do sistema já realiza boa alocação de threads, especialmente em nós com topologia simétrica.
- A simulação pode não ser suficientemente sensível à localidade de memória para que o *pinning* traga ganhos expressivos.

Diferenças observadas e fatores técnicos:

- **spread** teve desempenho ligeiramente superior com 64 e 128 threads, graças à ampla distribuição das threads, que reduz competição por cache (mitigando *thrashing*), saturação de barramentos e reutilização indesejada de L1/L2. Ideal para códigos com baixo acoplamento entre threads, como malhas tridimensionais.
- **close,** embora teoricamente vantajoso por agrupar threads em núcleos com cache compartilhado (L2/L3), teve ganhos marginais. O motivo é o baixo compartilhamento de dados entre threads — cada uma processa regiões independentes da malha. Possíveis estouros de cache L1/L2 não se mostraram relevantes.
- **false (**sem afinidade) apresentou desempenho estável, com migração de threads causando overhead mínimo. Em sistemas NUMA, migrações podem aumentar a latência de memória, provocar *cache flushes* e perda de localidade. No entanto, o

nó utilizado apresenta topologia regular e acesso quase uniforme à memória, o que suaviza esses efeitos.

- **true e master** apresentaram comportamento intermediário. true aplica afinidade automática, dependente da política do SO. master fixa as threads próximas à thread principal, útil quando há comunicação frequente, o que não se aplica a este caso.

Considerações Relevantes:
O impacto das políticas de afinidade tende a ser mais evidente nos seguintes contextos:

- **Códigos com alta reutilização de dados em cache**: Afinidades como close e master são vantajosas ao manter threads próximas, favorecendo o reaproveitamento de dados em caches L1/L2/L3 compartilhados.
- **Concorrência intensa por memória ou cache**: Estratégias como spread ajudam a distribuir threads entre núcleos distintos, reduzindo contenção, conflitos de cache e risco de *thrashing*.
- **Arquiteturas NUMA com latência assimétrica de memória**: Afinidades evitam migração de threads entre nós de memória, preservando localidade e reduzindo o custo de acesso remoto.
- **Algoritmos com comunicação frequente entre threads**: close e master minimizam a distância física entre threads que compartilham dados, otimizando latência de comunicação e uso de cache.

6. Conclusão

A variação de afinidades em um único nó do NPAD (amd-512) demonstrou que, para esta simulação numérica com carga bem distribuída e baixo compartilhamento de dados, os ganhos com afinidade explícita são marginais — mesmo em ambiente NUMA. Ainda assim, o experimento evidencia o valor da configuração manual com proc_bind e OMP_PLACES como ferramentas de ajuste fino em aplicações que dependem da localidade de memória, reutilização de cache e escalonamento estável, especialmente quando há risco de *thrashing* ou migração de threads em contextos de alta concorrência.