Trabalho 2

Lucas Emanuel de Oliveira Santos – GRR20224379
CI1316 – Programação Paralela
Universidade Federal do Paraná – UFPR
Curitiba, novembro de 2024

1. Implementação

A implementação realiza o particionamento do vetor de números inteiros (long long) Input em intervalos definidos pelo vetor P. O processo se baseia em duas etapas principais: primeiramente, uma busca binária é realizada no vetor Input para determinar a quantidade de elementos que pertencem a cada intervalo especificado por P. Esses valores são armazenados em um vetor que serve como contador para cada faixa. Além disso, a faixa do elemento tambem é armazenada em um vetor auxiliar para ser usada posteriormente no calculo do Output. Na segunda etapa, é construído o vetor Pos, que contém as posições iniciais de cada faixa dentro do vetor resultante Output. Essas posições são calculadas acumulando as contagens obtidas na etapa anterior, permitindo definir os limites exatos de cada intervalo no vetor final.

Com o vetor Pos inicializado, o programa utiliza o vetor auxiliar onde as faixas estão armazenadas para identificar onde cada elemento do Input deve ser alocado. Cada elemento é então posicionado no vetor Output de acordo com sua faixa correspondente, respeitando as posições iniciais definidas por Pos. Durante esse processo, um contador é utilizado para rastrear os índices disponíveis dentro de cada faixa.

Para assegurar a eficiência, as threads dividem o vetor Input em partes iguais, de forma que cada uma processa apenas uma subseção dos dados. Esse particionamento é realizado por meio de cálculos de índices que definem os limites de trabalho de cada thread. A função **thread_worker** é responsável por determinar essas faixas de trabalho com base no número total de elementos e no número de threads disponíveis. A sincronização entre as threads é gerenciada por barreiras (pthread_barrier_wait), garantindo que todas completem suas tarefas de contagem e posicionamento antes de avançar para as próximas etapas. Durante a execução, a contagem dos elementos por faixa é realizada de forma local e depois incrementada ao resultado das outras threads utilizando atômicos.

A função **multi_partition** coordena a execução de todo o processo. Inicialmente, ela cria e configura as threads, compartilhando estruturas de dados como vetores de entrada, partições, contagem e índices. A thread principal (thread 0) atua tanto como coordenadora quanto como participante, processando sua própria porção dos dados enquanto sincroniza as demais threads. Após o cálculo das contagens e o ajuste dos índices iniciais, a função executa a redistribuição dos elementos, completando o particionamento.

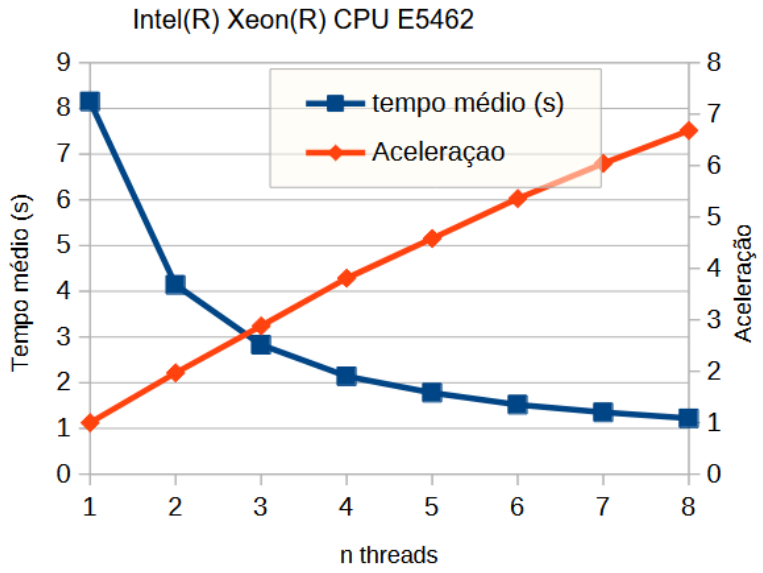
2. Resultados

2. A) Tabelas e gráficos

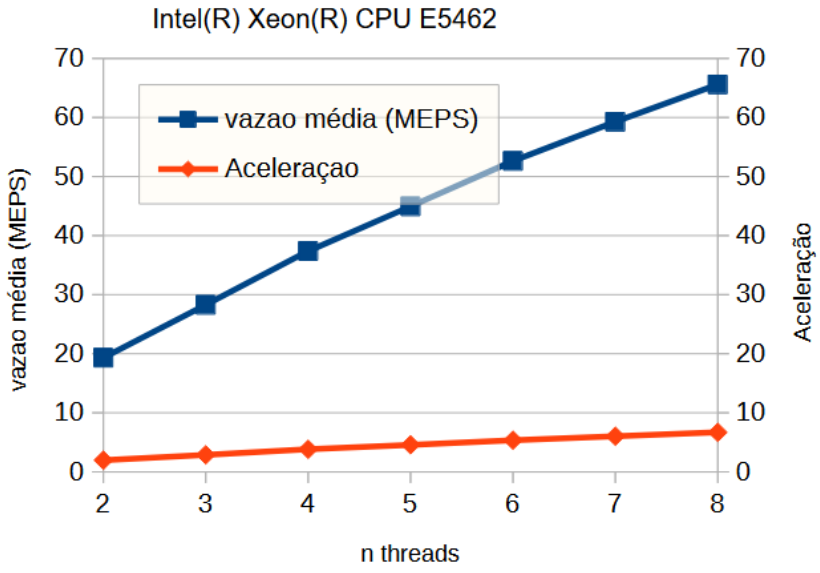
Parte A:

1 threads:	2 threads:	3 threads:	4 threads:	5 threads:	6 threads:	7 threads:	8 threads:
8.129087	4.153394	2.831404	2.144126	1.778207	1.514946	1.349320	1.218393
8.166638	4.174536	2.830195	2.141534	1.788767	1.499552	1.377601	1.249658
8.186347	4.097161	2.823615	2.149300	1.791928	1.520946	1.338434	1.191293
8.124162	4.151723	2.826584	2.132163	1.779245	1.513462	1.356251	1.193620
8.134270	4.108670	2.827640	2.141236	1.776796	1.538106	1.345354	1.242935

multi_partition com pool de Pthreads



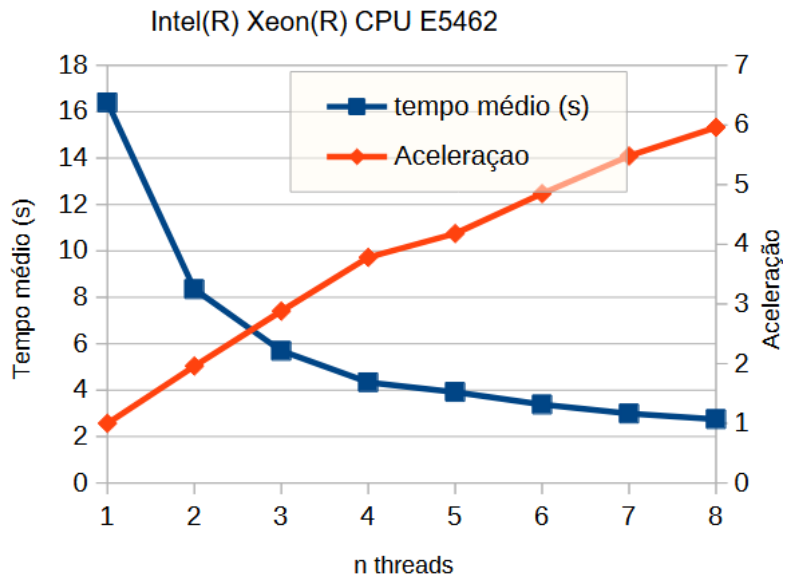
multi_partition com pool de Pthreads



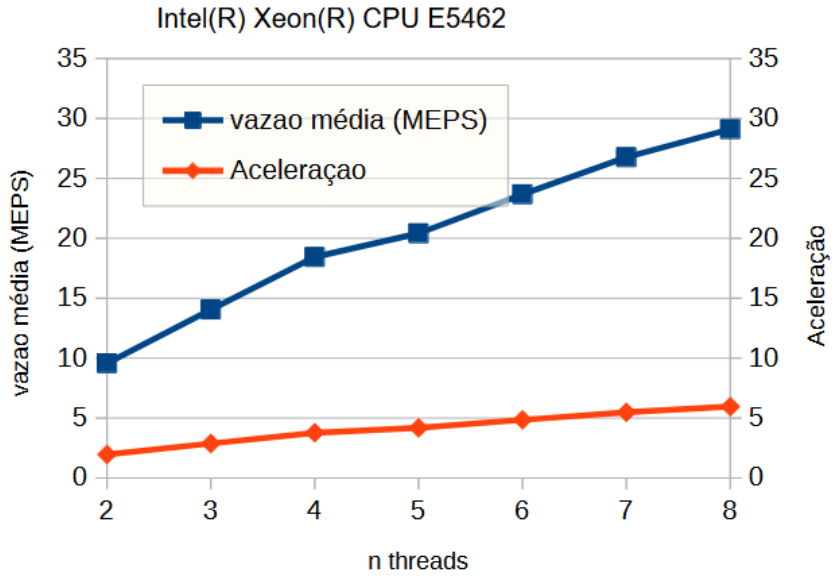
Parte B:

1 threads:	2 threads:	3 threads:	4 threads:	5 threads:	6 threads:	7 threads:	8 threads:
16.357971	8.406085	5.685368	4.313128	3.915742	3.373491	2.987675	2.836887
16.406169	8.350859	5.708623	4.332513	3.920572	3.379581	2.990815	2.696955
16.425525	8.369217	5.685369	4.331397	3.922195	3.362840	2.991226	2.747318
16.382369	8.323750	5.672891	4.357829	3.921059	3.384722	3.003467	2.762904
16.448089	8.312477	5.692945	4.359301	3.920205	3.394118	2.992709	2.715951

multi_partition com pool de Pthreads



multi_partition com pool de Pthreads



2. B) Análise dos resultados

Os resultados dos testes demonstram como o desempenho da implementação melhora significativamente com o aumento do número de threads, evidenciando a eficiência do paralelismo na divisão das tarefas no multi particionamento. Na Experiência A, que utiliza um número menor de partições, o tempo de execução reduziu de aproximadamente 8 segundos com 1 thread para aproximadamente 1 segundos com 8 threads, uma aceleração próxima de 7 vezes. Já na Experiência B, com um número maior de partições, o tempo diminuiu de aproximadamente 16 segundos para aproximadamente 2 segundos, indicando que a implementação mantém a escalabilidade mesmo em cenários mais complexos.

Apesar dos ganhos expressivos, observa-se que o aumento no número de threads apresenta retornos decrescentes. Por exemplo, o ganho de desempenho ao passar de 1 para 2 threads é muito maior do que ao passar de 7 para 8 threads. Esse comportamento é esperado devido ao overhead de gerenciamento de threads e à limitação imposta pela arquitetura do sistema, como o número de núcleos disponíveis e a eficiência do barramento de memória. Além disso, a Experiência B, que trabalha com mais partições, se beneficia mais do paralelismo devido ao maior número de operações realizadas, tornando o balanceamento de carga entre threads mais eficaz.

2. C) Características do processador

A máquina **w00** utilizada para os testes possui um processador Intel Xeon E5462 com 8 núcleos e 2,8 GHz. Com 8 núcleos físicos, a CPU é capaz de executar até 8 threads simultaneamente, aproveitando ao máximo o paralelismo da implementação. Cada núcleo é capaz de executar um único thread de cada vez, o que significa que, com até 8 threads, a carga de trabalho é bem distribuída entre os núcleos, permitindo que a execução ocorra de maneira eficiente e escalável.

Além disso, a arquitetura do processador também influencia a eficácia do paralelismo. Embora o número de threads adicionais possa continuar a distribuir a carga de trabalho, a máquina não pode executar mais threads do que os núcleos disponíveis sem recorrer à técnica de time-sharing, onde os núcleos alternam entre as threads, o que pode reduzir o desempenho devido à latência do processo de troca de contexto. Essa limitação é visível nos resultados dos testes, onde o ganho de desempenho é mais significativo ao passar de 1 para 2 threads do que ao passar de 7 para 8 threads, refletindo a saturação dos recursos da CPU.

3. Alterações e melhoramentos feitos nesse trabalho

3. A) Alterações e aperfeiçoamentos feitos no código

Criação de um vetor temporário **range_temp** para armazenar o resultado da busca binária realizada no calculo na contagem de elementos do intervalo. Assim, a busca binária só é realizada uma vez ao longo do código. Quando o vetor Output vai ser calculado o range é definido pelo resultado que foi armazenado em **range_temp.**

Criação de vetores locais **local_range_count** e **local_range_index** para efetuar os cálculos localmente durante as threads e só depois sincronizar os resultados com as demais threads. Atômicos ainda continuam sendo usados quando necessários.

Adição do cálculo da vazão em MEPS.

3. B) Alterações e aperfeiçoamentos feitas na metodologia de medições, scripts e cálculos

Não foi feita nenhuma mudança na metodologia de medições, scripts e cálculos de tempo e vazão.

3. C) Alterações e aperfeiçoamentos feitos nas planilhas

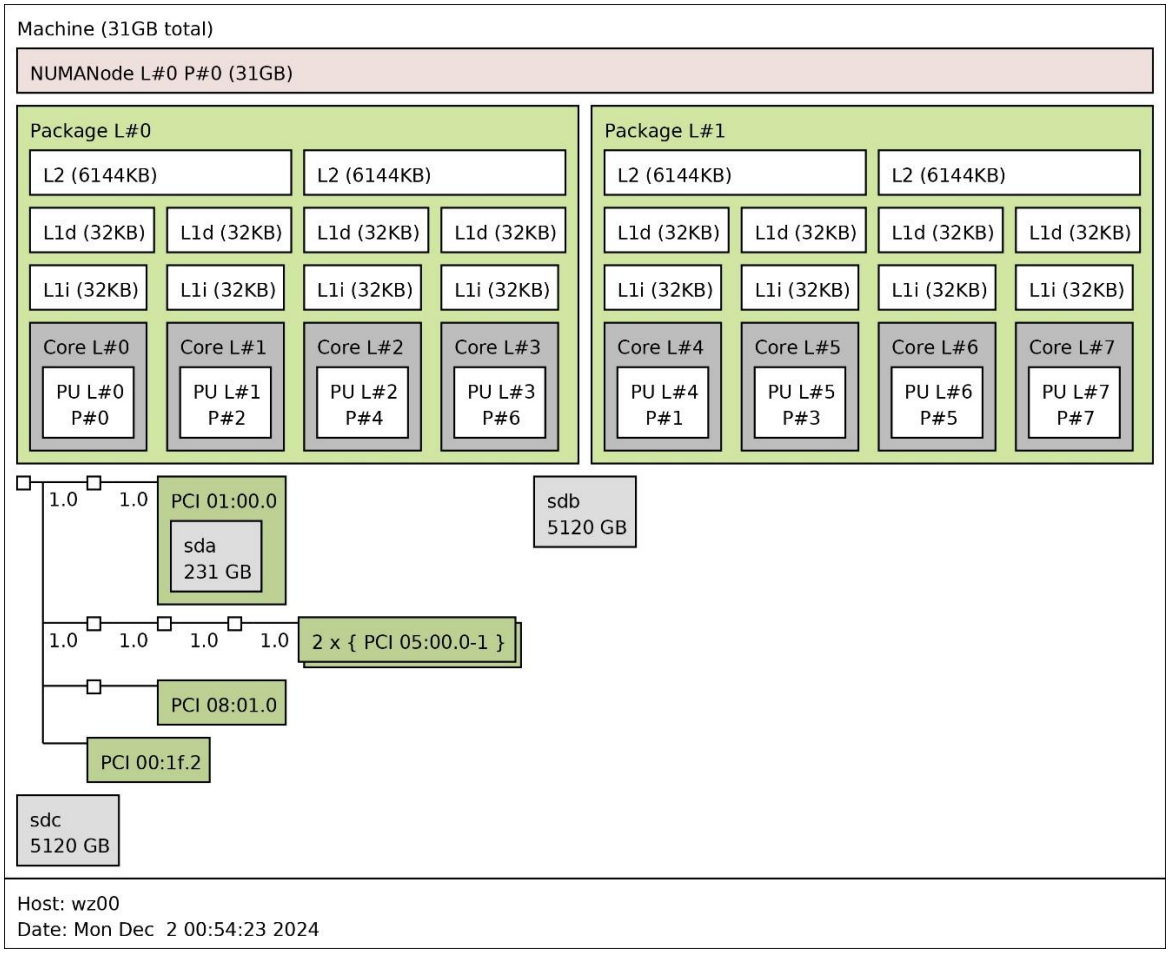
Substituição dos resultados anteriores pelos resultados dos novos testes realizados após as mudanças no código.

4. Apêndice

4. A) lscpu

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
Address sizes: 38 bits physical, 48 bits virtual CPU(s): 8
On-line CPU(s) list: 0-7 Thread(s) per core: 1 Core(s) per socket: 4 Socket(s): 2
NUMA node(s): 1 Vendor ID: GenuineIntel CPU family: 6
Model: 23
Model name: Intel(R) Xeon(R) CPU E5462 @ 2.80GHz Stepping: 6
CPU MHz: 2792.819
BogoMIPS: 5585.67
Virtualization: VT-x L1d cache: 256 KiB L1i cache: 256 KiB
L2 cache: 24 MiB
NUMA node0 CPU(s): 0-7
Vulnerability Itlb multihit: KVM: Mitigation: VMX disabled Vulnerability L1tf: Mitigation; PTE Inversion; VMX EPT disabled
Vulnerability Mds: Vulnerable: Clear CPU buffers attempted, no microcode; SMT disabled
Vulnerability Meltdown: Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2: Mitigation; Full generic retpoline, STIBP disabled, RSB filling
Vulnerability Srbds: Not affected
Vulnerability Tsx async abort: Not affected
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs bts rep_good nopl cpuid aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 lahf_lm pti tpr_shadow vnmi flexpriority vpid dtherm

4. B) lstopo


Machine (31GB total)
NUMANode L#0 P#0 (31GB)
Package L#0
L2 (6144KB)
L2 (6144KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#0
PU L#0 P#0
Core L#1
PU L#1 P#2
Core L#2
PU L#2 P#4
Core L#3
PU L#3 P#6
Package L#1
L2 (6144KB)
L2 (6144KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#4
PU L#4 P#1
Core L#5
PU L#5 P#3
Core L#6
PU L#6 P#5
Core L#7
PU L#7 P#7
1.0
1.0
PCI 01:00.0
sda
231 GB
sdb
5120 GB
1.0
1.0
1.0
1.0
2 x { PCI 05:00.0-1 }
PCI 08:01.0
PCI 00:1f.2
sdc
5120 GB
Host: wz00
Date: Mon Dec 2 00:54:23 2024