CI1316: Re-entrega do Trabalho 2

Fabiano A. de Sá Filho
Departamento de Informática
Universidade Federal do Paraná – UFPR
GRR20223831
fabiano.filho@ufpr.br

***Resumo*—Este trabalho teve como objetivo a paralelização de um programa de particionamento de vetores em C. Para tal foi utilizado um pool de threads com a biblioteca pthreads. Foram rodados dois experimentos, o primeiro com mil elementos no vetor de posições, e o segundo com 100 mil. Conforme planilhas em anexo, a paralelização foi um sucesso, de forma que ao aumentar o número de threads, diminui-se o tempo de execução e aumenta-se o throughput.**

I. Observações sobre o trabalho

O arquivo enviado contém os códigos fonte e makefile. O README contém as instruções para compilar e rodar o programa. Também tem a saída que foi rodada no cluster w00, e que foi utilizada na planilha para os dois experimentos, inclusa no subdiretório `resultados`. Vale pontuar algumas ressalvas sobre o desenvolvimento do trabalho:

- ~~Impactos da Cache: Modifique o script slurm fornecido e fiz meu proprio script run.sh, que produzem os resultados utilizados pela planilha. Dessa forma, o script slurm chama o executável novamente cada uma das NTIMES vezes, o que significa que a cada chamada do executavel os vetores são gerados novamente, não havendo portanto impacto da Cache nas medições de tempo.~~
- Apesar do meu algoritmo escalar bem com paralelismo, eu acredito que minha solução anterior tinha complexidade alta, talvez $O(n*np)$. Agora, com as modificações descritas nas seções a seguir, acredito que minha solução esteja mais próxima de $O(n*log(np))$.

II. Discussão de Resultados

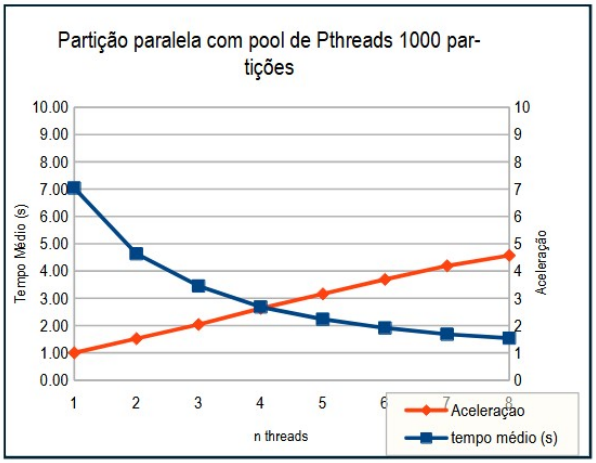
Após as modificações de código descritas na seção III, a experiência B pôde ser rodada no cluster com NTIMES interno do programa na casa dos 5 a 6. Mais do que isso, e o programa excede o tempo de 5 minutos cluster. Portanto, tanto os testes da parte A quanto B foram rodados com NTIMES interno igual a 5 e NEXECS no script igual a 5 também, o que garantiu um tempo de execução inferior a 3 minutos. Apesar desse NTIMES não ser ideal, é uma melhora considerável em relação à primeira entrega, onde a parte B não rodava nem uma vez sequer com NTIMES = 1.

Como pode ser observado pela planilha, o algoritmo de partição proposto obteve sucesso ao ser paralelizado. Por exemplo, para 1000 posições (parte A), obtive um tempo médio de 7,04 segundos sem threads utilizando as 8 threads, este tempo cai para 1,54, representando uma aceleração de 4,57x Veja os resultados na tabela abaixo:

Threads	1	2	3	4	5	6	7	8
Tempo médio (s)	7.04	4.63	3.45	2.68	2.23	1.91	1.68	1.54
Aceleração	1.00	1.52	2.04	2.63	3.16	3.69	4.19	4.57

Tabela I
Tempo médio e aceleração para a Parte A

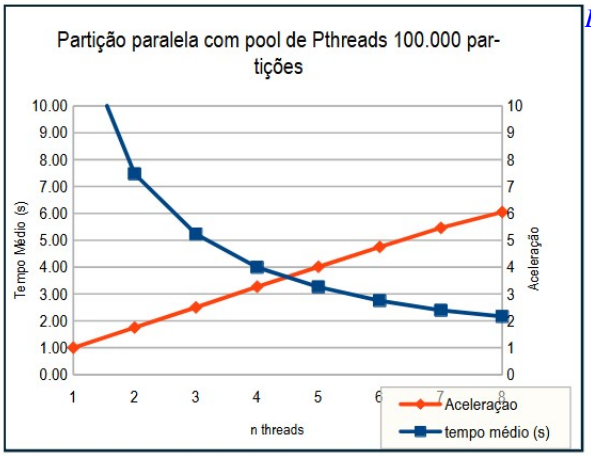
O gráfico à seguir mostra esta tabela em termos visuais:



Na experiência B, com 100,000 elementos, o resultado não foi diferente. Obtive uma aceleração de 6,05x utilizando 8 threads. Os resultados são resumidos na tabela e gráfico abaixo:

Threads	1	2	3	4	5	6	7	8
Tempo médio (s)	13.06	7.47	5.23	3.99	3.26	2.75	2.39	2.16
Aceleração	1.00	1.75	2.50	3.27	4.01	4.75	5.46	6.05

Tabela II
Tempo médio e aceleração para Parte B



Em termos da vazão, obtive resultados na ordem de 3 a 19 Mega Elementos Particionados por segundo (MEP/s), com boa escalabilidade de threads. Veja a tabela de resultados para a parte B:

Threads	1	2	3	4	5	6	7	8
MEPS	3.06	5.35	7.65	10.03	12.27	14.55	16.74	18.52
Aceleração	1.00	1.75	2.50	3.27	4.01	4.75	5.46	6.05

Tabela III
VAZÃO MÉDIA (MEPS) E ACELERAÇÃO PARA DIFERENTES NÚMEROS DE THREADS.

III. ALTERAÇÕES

A. No Código

- Modifiquei a função `thread_func`, que é passada para as threads, para utilizar uma nova função auxiliar de busca binária para contar o número de elementos em cada faixa. Isto melhorou o tempo de execução.
- Modifiquei a variável nTotalElements. Como reutilizei parte do código fornecido para o problema de `reduce_parallel`, esta variável ficava confusa no código uma vez que ela armazenava o tamanho do vetor P, e não o número total de elementos no vetor a ser particionado (vetor `Input`), que foi fixado em 8 milhões neste trabalho.
- Para ordenar o vetor P, mudei o algoritmo de ordenação do BubbleSort para MergeSort. Acredito que era isso estava contribuindo para o tempo de excução elevado.
- Adicionei o vetor de cópias, tanto para o vetor Input quanto para o vetor P, a fim de eliminar os efeitos de cache nas medições. Desta forma, cada vez que é chamada, a função `multi_partition_parallel` utiliza uma cópia diferente do vetor para evitar armazenamento na cache. O número de cópias a ser feito depende de uma variável `CACHE_SIZE`, a qual fixei em 32 MiB, a potência de 2 mais próxima do tamanho da cache da máquina do cluster (24 MiB, de acordo com o lscpu).

B. Nas medições, scripts e cálculos

- Mudei a métrica reportada no código para a unidade de MEP/s, ficando assim na mesma unidade utilizada nas planilhas, coisa que não foi feito na primeira entrega.
- Mudei, tanto no código quanto na planilha, o cálculo de throughput. Antes, eu estava utilizando o tamanho do vetor P vezes NTIMEs. Agora alterei para o tamanho do vetor Input vezes NTIMES, que no nosso caso é de 8 milhões x 5, ou seja, 40 milhões de elementos particionados a cada execução.
- Fiz um script separado para cada parte, os quais devem ser rodados individualmente. Antes, tinha feito tudo em um script só, possivelmente contribuindo para estourar o tempo alocado no cluster.

C. Na Planilha

- Adicionei as saídas do lscpu e do programa lstopo do cluster à planilha.
- Melhorei os gráficos, consertando as legendas e a escala do gráfico de tempo. Na primeira entrega, a linha de tempo médio ficava encima do zero, pois o gráfico estava configurado com a escala de centenas de segundos ao invés de segundos.
- Consertei os cálculos e unidades de tempo e vazão nas tabelas.

D. No Relatório

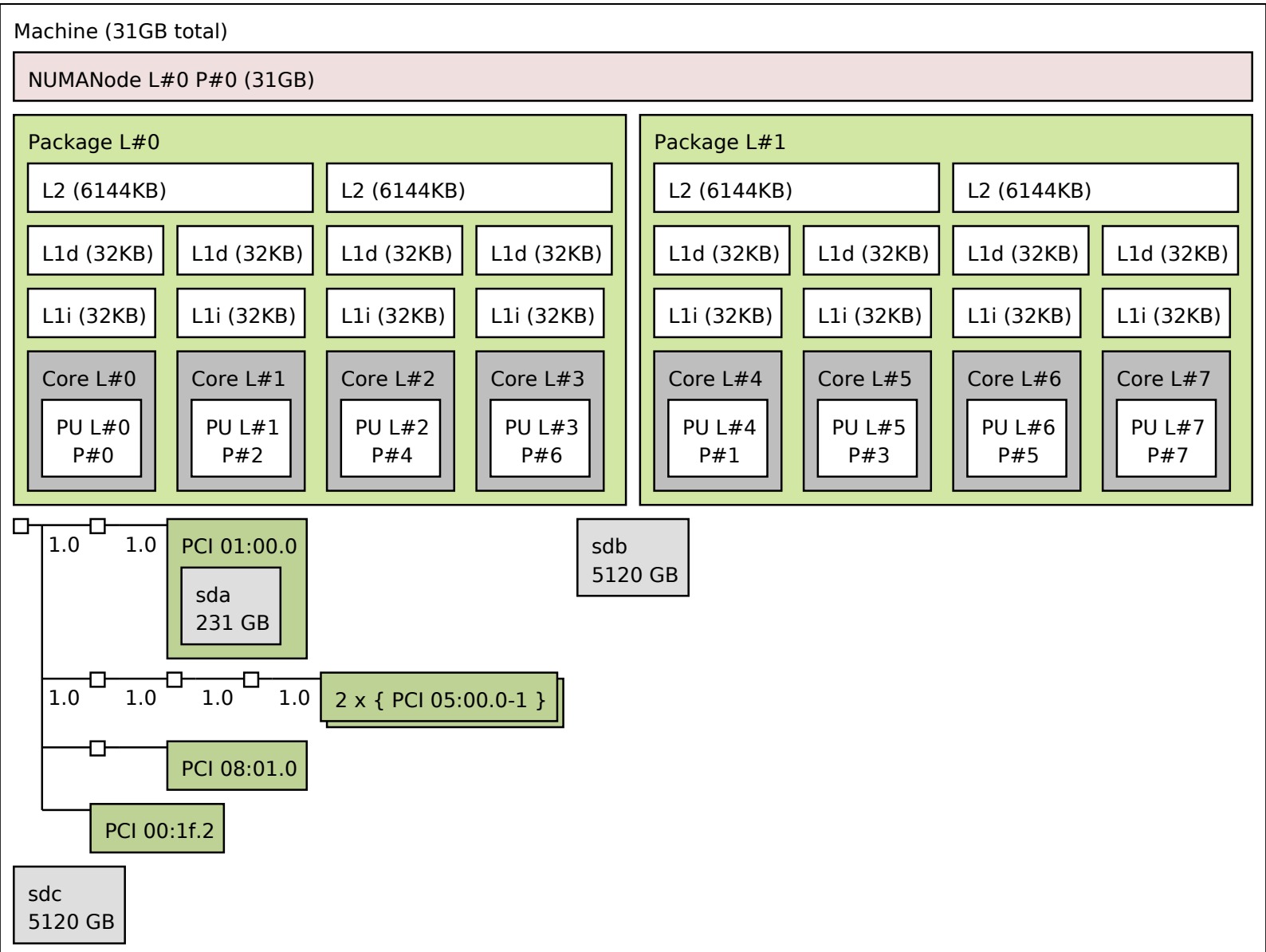
Adicionei esta seção de alterações, bem como apresentei uma análise mais detalhada dos resultados, com os gráficos gerados pela planilha. Todas as adições estão na cor azul. Seções “deletadas” foram riscadas em preto.

SAÍDA DO COMANDO 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  @ 2.80
   GHz
Stepping:                        6
CPU MHz:
   2792.839
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
```

IV. SAÍDA DO PROGRAMA TOPOLOGY



Host: wz00
Date: Mon Nov 25 21:21:15 2024