Relatório
Trabalho 5 de Programação Paralela

GRR20190427 - João Lucas Cordeiro

1 Introdução

Este é um relatório do quinto trabalho de Programação Paralela, onde implementamos um algoritmo que encontra os pontos de um conjunto P de D dimensões mais próximos de um conjunto Q com a mesma quantidade de dimensões, que pode ser rodado paralelamente entre vários processos. O algoritmo desenvolvido será detalhado, assim como o resultado dos experimentos usando o código.

2 O Algoritmo

O algoritmo inicia verificando a quantidade de argumentos que foram dados na chamada da execução. Caso a quantidade esteja correta, colocamos os argumentos em suas devidas variáveis (nQ, nP, nD e k). Depois disso, inicializamos o **MPI** e criamos e commitamos o tipo **MPI_T_PAR**, que será usado na troca de mensagens mais para frente. Então, alocamos espaço para os conjuntos Q e P.

Após isso, se for o processo raíz ($rank == 0$), alocamos espaço para a matriz resposta (R) e geramos os conjuntos Q e P. Temos então uma barreira para sincronizar os processos e inicializamos o cronômetro na raíz. Depois da sincronização, a raíz manda um pedaço da matriz P para cada outro processo. Cada processo será responsável pela sua parte da matriz. Mais uma barreira encerra essa fase de "inicialização" do algoritmo.

Enfim, chegamos na função *KNN*. Ela consiste de um laço que passa por todos os elementos de Q. À cada iteração, encontramos os pontos de P mais próximos desse elemento na função *calculaPontoQ*.

Começando essa função, a raíz envia a linha que representa o ponto de Q e os outros processos a recebem. Também alocamos espaço para um vetor que guardará os k pontos de cada pedaço de P mais próximos do ponto de Q que estamos lidando nessa iteração. Temos um vetor de *t_par*, um tipo que armazena um *float* (distância) e um *int* (índice do ponto). Na raíz alocamos um vetor maior, pois ela receberá os vetores dos outros processos futuramente. Após os *malloc's*, enchemos os vetores de **FLT_MAX**, o maior número que o *float* aguenta.

Depois disso, temos um laço que vai adicionando as menores distâncias e o índice do ponto em P no vetor de pontos próximos. Após esse cálculo, juntamos todos os vetores na raíz, usando mensagens com o tipo **MPI_T_PAR** citado anteriormente, que agora possui os pontos mais próximos de cada pedaço. Sincronizamos então com uma barreira.

Então, a raíz procura os k pontos menos distantes nessa junção de vetores e os coloca na linha da matriz resposta R. Então re-sincronizamos o programa com outra barreira e a função *calculaPontoQ* termina a execução. Depois de executarmos essa função nQ vezes, a função *KNN* termina e voltamos à *main*. Com isso, a raíz para o cronômetro, executamos da função *verificaKNN* e então imprimimos o tempo e a matriz resposta R. Finalizamos enfim o **MPI** e o programa.

3 Descrição do Processador

Os resultados citados neste relatório foram obtidos nas máquinas do laboratório 4 do DINF. Usando o comando *lscpu* temos estas informações do processador:

Arquitetura: x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes: Little Endian
Tamanhos de endereço: 39 bits physical, 48 bits virtual
CPU(s): 8
Lista de CPU(s) on-line: 0-7
Thread(s) per núcleo: 2
Núcleo(s) por soquete: 4
Soquete(s): 1
Nó(s) de NUMA: 1
ID de fornecedor: GenuineIntel
Família da CPU: 6
Modelo: 60
Nome do modelo: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Step: 3
CPU MHz: 3392.421
CPU MHz máx.: 3900,0000
CPU MHz mín.: 800,0000
BogoMIPS: 6784.91
Virtualização: VT-x
cache de L1d: 128 KiB
cache de L1i: 128 KiB
cache de L2: 1 MiB
cache de L3: 8 MiB
CPU(s) de nó0 NUMA: 0-7
Vulnerability Itlb multihit: KVM: Mitigation: VMX disabled
Vulnerability L1tf: Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds: Vulnerable: Clear CPU buffers attempted, no microcode; SMT vulnerable
Vulnerability Meltdown: Mitigation; PTI
Vulnerability Mmio stale data: Unknown: No mitigations
Vulnerability Retbleed: Not affected
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2: Mitigation; Retpolines, STIBP disabled, RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds: Vulnerable: No microcode
Vulnerability Tsx async abort: Not affected
Opções: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm abm cpuid_fault epb invpcid_single pti tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts

4 Resultados

O script que executa os experimentos onde os resultados foram adquiridos está em **script.sh**. Os experimentos são para: apenas 1 processo local, 4 processos locais e 4 processos remotos. Os resultados de cada um deles ficam, respectivamente, nos arquivos: **resultados-{1,4l,4r}.csv**. Os resultados citados aqui foram obtidos com os argumentos: $nQ = 128$, $nP = 75000$, $nD = 300$ e $k = 128$.

Gráficos de Tempo:

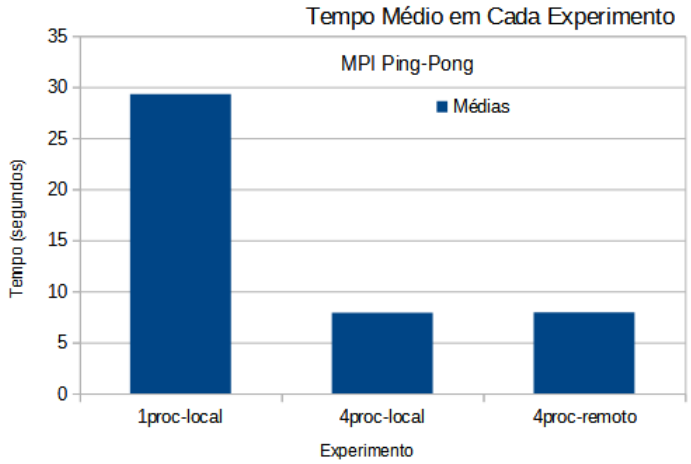
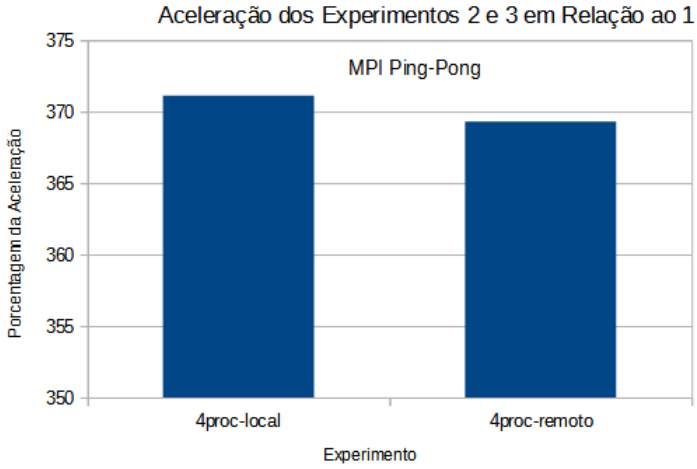


Gráfico de Aceleração:



Percebemos então que a divisão da execução em vários processos aumenta drásticamente a eficiência do programa, tanto localmente, quanto de forma remota. Também percebemos que, apesar de ser menos eficiente, a execução com processos remotos não fica muito atrás da versão com processos locais. Em comparação à execução com apenas um processo, temos essas acelerações:

Experimento	**Aceleração do Tempo Médio**
4 local	3.711258 (371,1% melhor)
4 remoto	3.692998 (369,2% melhor)

Revisando o código, acredito que, com mais tempo dedicado à implementação, alguns gargalos poderiam ser evitados/diminuídos, melhorando o desempenho do programa. Mas fora isso, a execução com vários processos possui uma aceleração bem decente comparada à execução com um processo único.