Relatório
Trabalho 4 de Programação Paralela

GRR20190427 - João Lucas Cordeiro

1 Introdução

Este é um relatório do quarto trabalho de Programação Paralela, onde implementamos trocas de mensagens por meio do MPI que roda em máquinas diferentes. Temos 3 algoritmos: ping-pong, broadcast e a nossa própria implementação do broadcast. Os chamamos de A, B e C, respectivamente. Também temos o arquivo D, que é, basicamente, o algoritmo C, com um código mais limpo e com o verificador de implementação. Os algoritmos desenvolvidos serão detalhados, assim como o resultado dos experimentos usando os códigos.

2 Os Algoritmos A e B

Esses algoritmos são mais simples comparados ao C, por isso, explicaremos seu funcionamento em um único capítulo. Os dois algoritmos começam iguais: tratam a entrada, verificando os argumentos, iniciamos o MPI e guardamos o número de processos MPI em *nProc* e o *rank* do processo atual em *rankProc*. Depois, guardamos em *ni* quantos números terão em cada mensagem.

No algoritmo A, criamos a mensagem para os dois processos, que irão então ficar mandando a sua mensagem para o outro processo, e alocamos o buffer que irá receber essa mensagem. Já no B, criamos a mensagem para o processo raiz e alocamos espaço em que os outros processos receberão. Depois disso, sincronizamos os processos com uma barreira.

Então, caso o rank do processo for 0, ele começa a cronometrar a execução. No algoritmo A, o processo 0 e o 1 ficam mandando mensagens um para o outro, *nmsg* vezes, usando o **MPI_Ssend** e o **MPI_Recv**. Já no B, a raiz manda a mensagem para os outros processos *nmsg* vezes por meio do **MPI_Bcast**. Depois de mandar as mensagens, usamos uma barreira para sincronizar os processos.

Enfim, caso seja o processo 0, paramos o cronômetro, guardamos o tempo total na variável *tempoMS* em microsegundos e a vazão em *vazao* em bytes/microsegundos, e os imprimimos na tela. Por fim, o **MPI** é finalizado.

3 O Algoritmo C e D

Este algoritmo também começa de forma semelhante aos outros 2: tratamos a entrada, iniciamos o MPI e guardamos o número de processos, o rank do processo atual e a quantidade de números em cada mensagem nas respectivas variáveis. Agora, ele se assemelha ao algoritmo B: criamos uma mensagem para a raiz e alocamos espaço para os outros processos a receberem. Usamos uma barreira para sincronizar os processos.

Caso o rank do processo for 0, ele começa a cronometrar a execução. Aqui, no código D, a função **my_Bcast** é chamada e faz a mesma coisa que o C continuará fazendo. Calculamos quantas "fases" a implementação do broadcast terá baseada no número de processos usando a função **tetoLog**, e guardamos em *numFases*. Então, declaramos as variáveis *faseComeco*, *destinoMsg* e *origemMsg*, que serão úteis na troca de mensagens. Após isso, entramos no laço que envia *nmsg* mensagens.

Primeiramente, descobrimos em qual fase o processo atual receberá a sua primeira mensagem com a função **descobreFase** e guardamos em *faseComeco*. Caso o processo não seja a raiz, ele entra no *if*. Nele, calculamos a origem da mensagem que ele receberá e guardamos em *origemMsg*. Então, esperamos o recebimento de uma mensagem com o **MPI_Recv**.

Saindo do if, entramos no laço que itera entre as fases da transmissão. O número de vezes que o laço é executado depende da *faseComeco*, executando *numFases* - *faseComeco* vezes. Começando o laço, calculamos o destino da mensagem e guardamos em *destinoMsg*. Depois, verificamos se é a última fase da transmissão. Se não, apenas enviamos a mensagem para o destino. Se sim, calculamos a distância entre o processo atual e a raiz. Então, somamos isso com 2 elevado à *faseComeco*, que agora guarda a fase atual por ser o índice do *for*, calculado com a função **pow2**. Caso essa soma for menor que o número de processos, enviamos a mensagem. Essa verificação evita envio de mensagens para processos que já possuem a informação. O envio é feito com **MPI_Ssend**.

Aqui, no código D, retornamos da função **my_Bcast** e continuamos igual ao C.Depois disso, sincronizamos os processos usando uma barreira e recomeçamos o laço. Depois de fazermos *nmsg* broadcasts, também usamos uma barreira para sincronização. Enfim, caso seja o processo 0, paramos o cronômetro, guardamos o tempo total na variável *tempoMS* em microsegundos e a vazão em *vazao* em bytes/microsegundos, e os imprimimos na tela. Então, o código D chama a verificação de buffers, que testa se os broadcasts estão funcionando. Por fim, o **MPI** é finalizado.

4 Descrição do Processador

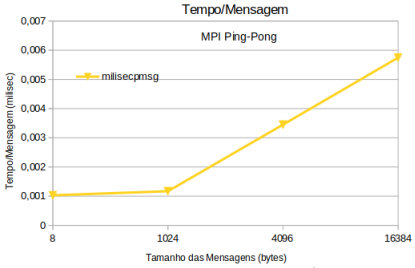
Os resultados citados neste relatório foram obtidos nas máquinas do laboratório 4 do DINF. Usando o comando *lscpu* temos estas informações do processador:

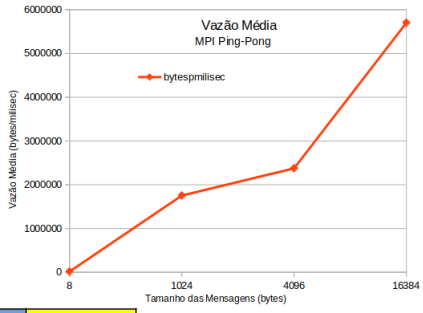
Arquitetura: x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes: Little Endian
Tamanhos de endereço: 39 bits physical, 48 bits virtual
CPU(s): 8
Lista de CPU(s) on-line: 0-7
Thread(s) per núcleo: 2
Núcleo(s) por soquete: 4
Soquete(s): 1
Nó(s) de NUMA: 1
ID de fornecedor: GenuineIntel
Família 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

5 Resultados

O script que executa os experimentos onde os resultados foram adquiridos está em **script.sh**. Os resultados são armazenados em **resultados-{a-b-c-d}.csv**. A execução foi feita com 8 processos, cada um em uma máquina diferente. Os gráficos foram gerados com uma planilha baseada na disponibilizada pelo professor. Também temos uma tabela que informa quantas mensagens foram usadas nos testes para cada tamanho. Estes são os gráficos gerados:

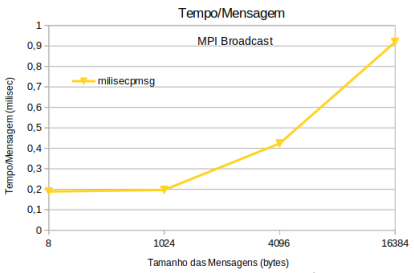
Gráficos para o algoritmo A:

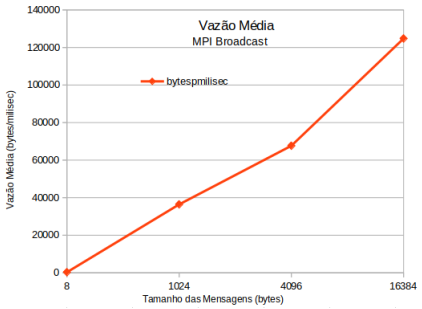




Número Mensagens	Tamanho
5000	8
2500	1024
1250	4096
625	16384

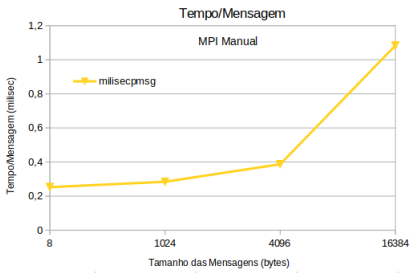
Gráficos para o algoritmo B:

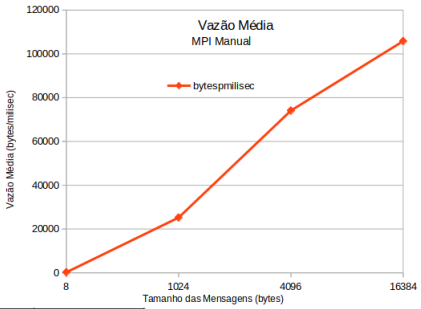




Número Mensagens	Tamanho
5000	8
2500	1024
1250	4096
625	16384

Gráficos para o algoritmo C e D:





Número Mensagens	Tamanho
5000	8
2500	1024
1250	4096
625	16384

Aqui, temos uma tabela que mostra a aceleração da latência examinada entre o **MPI_Bcast** e o **my_Bcast**, mostrando quanto o broadcast do MPI é mais rápido para cada mensagem que o implementado pelo aluno nos experimentos realizados.

Tamanho das Mensagens	**Aceleração da Latência**
8	1.3365808 (33,6% melhor)
1024	1.4433027 (44,3% melhor)
4096	0.9132705 (8,7% pior)
16384	1.1789195 (17,8% melhor)

Acredito que o mais interessante de se notar é a diferença entre o broadcast nativo do MPI e o implementado pelo aluno. Eles seguem um mesmo padrão de crescimento nos dois gráficos, mas o do MPI ainda é melhor que o do aluno, no geral. No caso das mensagens com tamanho 4096 bytes (4KBytes), todos os tempos do broadcast do MPI foram pior que o **my_Bcast**.