Relatório do Terceiro Trabalho Prático
Computação de Alto Desempenho

Pedro Hollanda Boueke {phboueke@poli.ufrj.br}
Erik Fernandes Tronkos {erik.tronkos@poli.ufrj.br}
UFRJ

25/07/2016

Os dados obtidos e utilizados para a realização desse relatório serão entregues junto ao mesmo. Ademais, os códigos fonte completos também estarão disponíveis em https://github.com/pboueke/CAD-COC472, no diretório "/3". Os experimentos e iterações foram realizados por meio de scripts bash e python, que também serão disponibilizados, responsáveis pela execução ordenada dos programas e geração de tabelas a partir dos dados obtidos.

OS	Ubuntu 14.04 LTS 64-bit
Processor	Intel Core i7-337U CPU @ 2.00GHz (cache size: 4096 KB; line size: 64B;) x 4
Memory	2 x 4GiB SODIM DDR3 Synchronous 1600 MHz (0,6 ns)

Tabela de especificações de hardware. Informações de hardware do ambiente

Exercício 1

Conforme o enunciado, a implementação desse exercício foi feita de forma simples, de modo que todo processo inicializado imprima no console uma mensagem com sua identificação. Além disso, o processo mestre imprime o número total de processos. O código se encontra em "3/1/mpi_hello_world.c".

Exercício 2

Seguindo o enunciado, a implementação desse exercício se focou na criação de um programa envio de mensagens "hello" entre processos de forma não bloqueante ("3/2/helloBsend.c") e bloqueante ("3/2/helloNBsend.c"). A principal diferença entre os dois programas está na chamada das funções MPI de envio e recebimento de mensagens. Para chamadas bloqueantes usando MPI_Send e MPI_Recv e para chamadas não bloqueantes usando MPI_Isend e MPI_Irecv. O código referente aos programas se encontra nos arquivos mencionados acima.

Exercício 3

A implementação desse exercício segue o pedido pelo enunciado. Cada processo transmite dados ao próximo de forma a gerar uma topologia anelar. A comunicação entre processos é feita por meio de envio de mensagens não bloqueante, com todos os processos se comunicando concorrentemente, havendo um MPI_Waitall ao fim das transmissões para que possa-se garantir que o retorno (impressão dos dados) seja acurado em todos os processos. O código referente a esse exercício se encontra em "3/3/ring.c".

Exercício 4

Esse execício, por fim, mistura elementos dos anteriores para criarmos um processo que execute a soma circular dos identificadores de todos os processos. Nele, utilizando uma topologia anelar, transmitimos, a cada iteração, o identificador recebido pelo seu vizinho anterior ao vizinho posterior, somando sempre o valor a uma soma local.

A implementação depende de um laço no qual as transmissões, para cada iteração, do identificador recebido (a iniciar do seu próprio identificador) são iniciadas e concluídas para todos os vizinhos. Se o identificador recebido é o seu próprio, então a execução é concluída. O código referente a esse exercício se encontra em "3/4/cyclicsum.c"

Exercício 5

A implementação desse exercício se encontra no arquivo "3/5/ex5.c". O programa inicializa uma matriz A e outra B no processo mestre (taskid = 0) e faz a distribuição delas para os demais processos. Idealmente, faríamos um scatter de A e da transposta de B, de forma que fosse possível que cada processo consiga realizar os cálculos necessários sem a necessidade de buscar os pedaços que faltariam de B. Na implementação realizada, contudo, a fim de facilitar o desenvolvimento, é feito um broadcast de B e um scatter de A. Cada processo, então realiza os cálculos referentes à sua fração de A.

Ao final dos cálculos ocorre um gather na matriz resultado C, localizada no processo mestre. Abaixo, a tabela que relaciona N ao número de processos.

N# processos	1	2	4	8
tempo(s) para N=1000	9.607	6.350	5.992	6.030
tempo(s) para N=5000	1825.378	1992.875	2004.770	1094.504

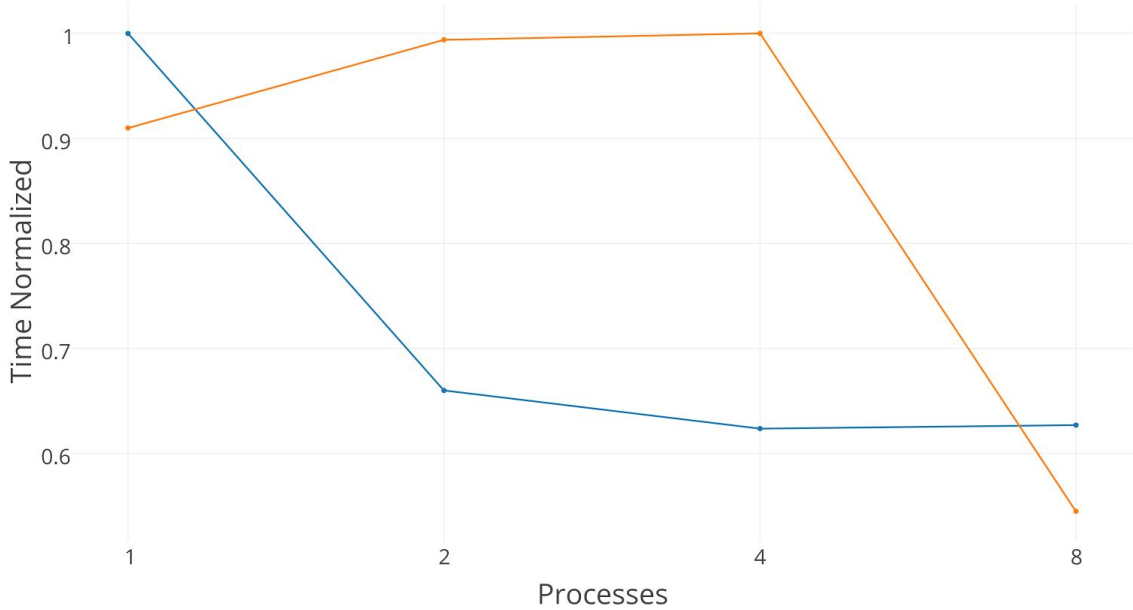

Gráfico 1. Tempos normalizados. LARANJA: N=5000. AZUL: N=1000.

Observa-se que a eficiência da implementação com MPI é superior à da implementação serical (número de processos igual a 1) quando N vale 1000, porém o comportamento para N igual a 5000 é diferente, só apresentando bons resultados para u número de processos igual a 8. O resultado obtido para N igual a 5000 parece contra-intuitivo e contradiz o comportamento esperado, acreditamos que parte desse fato se deve à instabilidade do sistema operacional no qual as execuções foram realizadas.

Exercício 6

A implementação desse exercício se encontra nos arquivos "3/6/prodserial.c" (algoritmo serial) e "3/6/prodmpi.c" (algoritmo com mpi). O programa inicializa uma matriz A e outra B no processo mestre (taskid = 0), que são distribuídas aos demais processos por meio da função MPI_Scatter. Após o scatter, cada processo realiza os cálculos referentes aos pedaços obtidos das matrizes originais e acumulam o resultado em uma variável que, então, sofre um MPI_Reduce, trazendo o resultado final distribuído ao processo mestre.

A tabela abaixo relaciona a versão serial com a distribuída para diferentes valores de N e de processos.

N# processos	serial	2	4	8
tempo(s) para N=10000000	0.086	1.08	0.073	0.087
tempo(s) para	0.198	0.128	0.131	0.188

N=20000000				
tempo(s) para N=40000000	0.299	0.213	0.290	0.286
tempo(s) para N=200kk	0.893	0.723	0.780	1.217

Os resultados obtidos apontam para uma melhora de eficiência com o aumento do número de processos, mesmo que pequena. Nota-se que para um número muito grande de processos há perda de eficiência.

Exercício 7

O primeiro pedido, de simplesmente rodar o programa inserindo o MPI_Init() e MPI_Finalize(), foi realizado sem dificuldades, replicando o mesmo processamento para diferentes processos. A distribuição entre processos em si foi realizada, por meio da criação de células fantasmas que são atualizadas a cada iteração pelo envio de mensagens não bloqueantes. Cada processo recebe um número de linhas a serem processadas equivalente a o número total de linhas dividido pelo número de processos, sendo que o último processo recebe a diferença entre o número total de linhas pelo número total de linhas utilizado pelos outros processos.

Durante o passo do algoritmo, é realizado:

- cálculo das condições de contorno para faces da direita e da esquerda;
- envio da posição 1 do trecho local para o vizinho de cima;
- recebimento da posição 0 do trecho local do vizinho de cima;
- envio da posição n-2 do trecho local para o vizinho de baixo;
- recebimento da posição n-1 do trecho local do vizinho de baixo;
- cálculo da soma dos vizinhos entre as posições 2 e n-2 do trecho local;
- MPI_Waitall para a troca de mensagens;
- cálculo das condições de contorno para esquinas do domínio;
- cálculo da soma dos viznhos para as posições restantes (1 e n);
- cópia do estado para o array ‘novo’;

Um detalhe a ser notado é que, enquanto as trocas de mensagens não foram concluídas, ocorre o processamento das posições que não dependem daquelas que estão sendo recebidas. O processamento das posições que dependem dos dados a serem recebidos só ocorre após o MPI_Waitall.

Por fim, cada processo faz o somatório do número de células vivas e esse valor é acumulado em uma variável no processo principal. Essa redução é feita por meio do MPI_Reduce.

O código referente a esse exercício se baseia no código fornecido para o trabalho, e se encontra em "3/7/game_of_life-mpi.c". Na tabela abaixo encontra-se a relação entre o tempo de execução (real time em segundos) e o número de processos escolhido.

N# processos	1	2	4	8	16
Tempo (s)	4.605	5.128	3.290	3.564	4.252