Trabalho 3

Felipe Podolan Oliveira

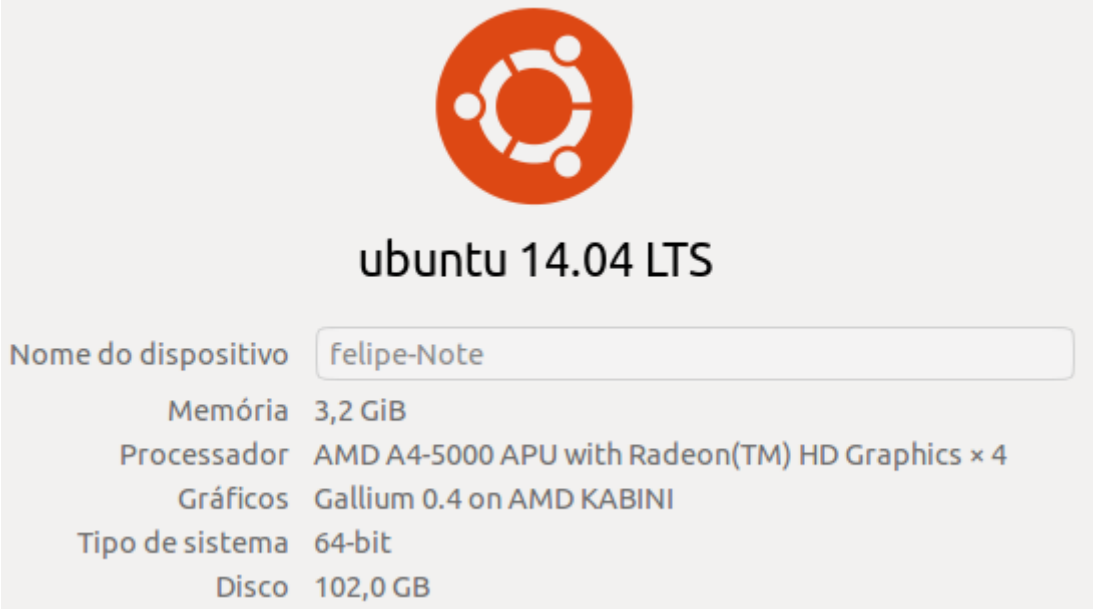
11 de julho de 2015

Computação de Alto Desempenho

Conteúdo

1 Especificação da máquina

Todos os códigos foram compilados e executados em uma máquina cujo processador é um AMD Quad Core A4, com 4GB de ram e no sistema operacional Ubuntu 14.04. A imagem a seguir mostra essas informações:



Especificação da máquina

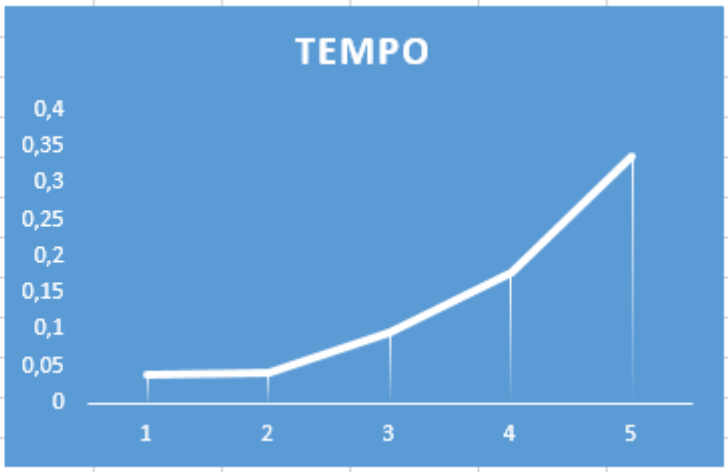
2 Exercício 1

Inicialmente foram alocados quatro vetores A, B (tamanho $N = 10^7$) e C, E (tamanho $n = N/p$). Onde p é o número de processos.
Para simplificar, supôs-se que N/p é uma divisão perfeita. Em seguida, os vetores A e B foram populados com a função rand() e passaram a receber valores pseudo-aleatórios.
Então, foi ultilizada a rotina $MPI\_Scatter()$ para distribuir os elementos do vetor A entre os vetores C de cada processo e fez-se o mesmo com os vetores B e D.
Com isso, cada processo passou a possuir vetores C e D contendo uma fração de tamanho n dos vetores A e B respectivamente.
Depois, realizou-se o cálculo do produto interno entre os vetores C e D e o resultado foi armazenado em uma variável chamada s.
O produto interno dos vetores A e B consiste na soma de todos os produtos internos locais de C e D. Existe uma rotina MPI que faria este cálculo de maneira eficiente e armazenaria o resultado em todos os processos, a rotina $MPI\_Allreduce()$.
Entretanto, foi solicitado que os envios fossem realizados com um pseudo-código fornecido. Portanto, foram utilizadas rotinas $MPI\_Send()$ para eviar as variáveis

s de todos os processos para todos os processos, exceto o próprio, que receberam o valor numa variável t utilizando a rotina $MPI\_Recv()$.
A cada recebimento de um novo valor em t, somou-se t ao s. No final, todos os processos passaram a ter o resultado do produto interno entre A e B em suas variáveis s.
Foi utilizada a rotina $MPI\_Wtime()$ para medir o tempo das trocas de mensagens. Os resultados estão na tabela a seguir:

#Processo	Tempo
2	0,037262
4	0,040175
8	0,09686
16	0,176598
32	0,335056

Obs. Tempo calculado para a troca de mensagens!!!!



Tempo para toca de mensagens do produto interno

3 Exercício 2

Esse exercício consistia em realizar um produto matricial em blocagem, onde cada processo calcularia os elementos finais de um bloco da matriz resultante.
O exercício foi bastante difícil e trabalhoso, pois as linhas e colunas enviadas do processo 0 para os demais processos deve se repetir, ou seja, não é uma distribuição uniforme das linhas e colunas.
Inicialmente, abre-se o arquivo fornecido no primeiro argumento da linha de comando e obtém-se a dimensão das matrizes. Esse valor é armazenado na variável dimention.
Então, são alocadas três matrizes A, B (tamanho dimention) e C de tamanho $n = dimention/sqrt(nTasks)$ onde nTasks é o número de processos.
Todos os processos populam a matriz C com 0.
Depois disso, o processo 0 popula as matrizes A e B com o conteúdo dos arquivos utilizando a função populate(), que basicamente lê cada inteiro do arquivo e vai adicionando nas matrizes.
A função foi construída de forma a popular a matriz B com a transversa da matriz do arquivo.
Em seguida, o processo 0 deve enviar n linhas das matrizes A e B para cada pro-

cesso. Porém, como mencionado anteriormente, essas n linhas devem ser enviadas de forma que os processos contenham as linhas necessárias para realizar o produto matricial.

Para resolver o problema, foram criadas duas variáveis $start\_row$ e $start\_col$ inicializadas com 0 e que contêm a primeira linha enviada da matriz A e a primeira linha/coluna enviada da matriz B respectivamente. Na verdade envia-se linhas da matriz B (já que ela está transposta) mas para um entendimento mais fácil, as linhas da matriz B serão chamadas de colunas.

O processo 0 itera sobre os demais processos. As linhas da matriz A enviadas serão as mesmas a cada sqrt(nTasks) vezes. Ou seja, nas sqrt(nTasks) primeiras iterações (que envia para os primeiros sqrt(nTasks) processos), serão enviadas linhas começando na linha 0 e acabando na linha n e, portanto, $start\_row$ contém o valor 0. Nas próximas sqrt(nTasks) iterações, serão enviadas linhas da matriz A começando em n e indo até 2n ($start\_row = n$).

Então, se o iterador for multiplo de sqrt(nTasks), $start\_row$ é incrementada de n.

Já para a matriz B, serão enviadas n colunas começando em $start\_col$. Porém, as colunas enviadas devem ser incrementadas de n a cada iteração até que se atinja o final da matriz B, ou seja, $start\_col + n = dimention$. Quando esse valor for atingido, as colunas devem começar a ser enviadas a partir da coluna 0 novamente. Atingir essa condição é equivalente a atingir a condição de o iterador ser um multiplo de sqrt(nTasks).

Ou seja, na primeira iteração, serão enviadas n colunas de B, começando em 0 ($start\_col = 0$) até n. Na segunda iteração, serão enviadas n colunas começando em n ($start\_col = n$) até 2n. Quando já tiverem sido enviadas n colunas sqrt(nTasks) vezes, $start\_col$ volta a ser 0.

Além disso, devem ser enviados os valores de $start\_row$ e $start\_col$ a cada iteração.

Então, o processo 0 envia para cada processo, n linhas da matriz A utilizando a rotina $MPI\_Send()$ que serão armazenadas nas n primeiras linhas da matriz A local de cada processo e envia o valor atual de $start\_row$ do processo 0 também que é armazenado no $start\_row$ local de cada processo.

O processo 0 também envia n colunas da matriz B (envia as linhas na verdade) que são armazenadas nas n primeiras linhas da matriz B local de cada processo utilizando a mesma rotina e o valor atual de $start\_col$ do processo 0 que é armazenado no $start\_col$ local de cada processo.

Todos os recebimentos são realizados utilizando a rotina $MPI\_Rec()$.

Após o recebimento, todos os processos terão nas n primeiras linhas das suas matrizes A e B as linhas necessárias das matrizes A e B originais para realizar o produto matricial do seu bloco que terá dimensão nxn.

Como a matriz B está transposta, o produto foi calculado de uma maneira um pouco

diferente: C[i][j] += A[i][k] * B[j][k] (e não B[k][j]) para k variando de 0 até dimention.
O produto foi armazenado na matriz C local de cada processo.
Depois, cada processo (exceto o 0) envia sua matriz C para o processo 0 e envia também os valores de suas variáveis $start\_row$ e $start\_col$. O processo 0 recebe esses valores e armazena na matriz A da seguinte forma: o C[0][0] recebido é colocado na posição A[$start\_row$][$start\_col$]. As n linhas seguintes são recebidas a partir dessa posição.
Os envios e recebimentos são realizados com as rotinas $MPI\_Send()$ e $MPI\_Recv()$.
Por fim, o processo 0 coloca seu C local no bloco 0x0 da matriz A, ou seja A[0][0] = C[0][0] e as próximas n linhas e colunas são armazenadas a partir dessa posição.
Depois de tudo isso feito, a matriz A do processo 0 conterá o resultado, que é então escrito no arquivo do terceiro argumento fornecido pela linha de comando.

4 Exercício 3

O objetivo desse exercício era paralelizar uma função de estimar número π comparando quantos pontos aleatórios estariam dentro de um circulo de raio 1 inscrito num quadrado de aresta 1. O número de pontos dentro do circulo é count e o número total de pontos é $num\_samples$.
Para paralelizar, bastava adicionar as inicializações do MPI que o programa seria rodado em paralelo, com algumas adaptações.
A primeira adaptação foi para fazer com que cada processo obtivesse valores diferentes de pontos, pois o mesmo conjunto de pontos produzem o mesmo valor de count e consequentemente, o valor estimado do π seria o mesmo que rodando sequencialmente.
Para isso, cada processo deve ter um seed diferente, então bastou multiplicar o seed pelo rank do processo dentro da função srand(). Com isso, a função rand() gera conjuntos diferentes de pontos para cada processo.
Agora cada processo vai encontrar valores diferentes de pontos e consequentemente, terá um count local diferente. Para um número de pontos igual a $num\_samples$.
A ideia é que o processo mestre some ao seu count os count dos demais processos. Para isso, todos os processos enviam seu count para uma variável ($received\_count$) do processo 0, que então, soma esse valor recebido à sua variável count.
Além disso, a variável $num\_samples$ do processo 0 deve ser multiplicada pelo número de processos.
No final, o processo 0 calcula o $\pi = count/num\_samples$.
O objetivo é que cada processo trabalhe com $num\_samples$ pontos, já que, quanto

maior o número de pontos distintos, menor o erro esperado, pois quanto mais pontos aleatórios houver, maior a área coberta.
O valor de π é exato para (área do circulo)/(área do quadrado).
$(\pi * R^2)/L^2$. Como $R = L$, esse valor é π.
Como esperado, o valor estimado de π se aproxima do valor real com o aumento do $num\_samples$.