Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação

Projeto 2: Processamento de Imagens Utilizando OpenMP e OpenMPI

Programação Concorrente
Prof. Julio Cezar Estrella

Lucas Vieira Costa Nicolau 8517101
Lucas Peres Nunes Matias 8531633

São Carlos
2015

Índice

Introdução

O trabalho com processamento de imagens é muitas vezes custoso. O processamento de imagens pequenas pode ser simples e rápido, porém, a leitura de grandes imagens e armazenamento em memória principal muitas vezes leva tempo e muito custo computacional. Uma das alternativa para melhorar o desempenho é a divisão do trabalho em threads, aproveitando ao máximo o uso do processador.

Neste trabalho usamos duas bibliotecas para dividir o trabalho do processamento de imagens, OpenMP e OpenMPI. A OpenMP divide o trabalho em threads, enquanto a OpenMPI divide o trabalho em processos auxiliando na troca de mensagens entre esses processos. Nossa metodologia divide a matriz com os pixels da imagens em diferentes números de processos e em cada processo são usadas threads para paralelizar o trabalho.

Apesar de ambos os métodos, sequencial e paralelo, funcionarem o paralelo não obteve melhor desempenho pois não conseguimos dividir o processamento em vários processos.

Implementação

Neste projeto fizemos duas implementações para trabalhar com o processamento de imagens. Uma implementação sequencial e outra paralelo ambas em C. Para executar a versão sequencial usamos o comando *gcc main.c image.c* e depois rodamos o executável. Para a versão em paralelo usamos um comando diferente *mpicc main.c image.c -fopenmp*. Após a execução do programa ele espera o nome da imagem, na pasta principal do programa temos duas pastas *Images* e *outImages*, as imagens serão lidas diretamente da pasta *Images* e ao final do processamento salvas na pasta *outImages*. É calculado o tempo de execução do algoritmo. É importante ressaltar que o cálculo do tempo é iniciado antes de entrar na função de *smoothing* e encerrado quando a função retorna para a main. Focamos em calcular o tempo de execução do algoritmo e não da leitura da imagens e demais fatores.

Sequencial

A implementação foi feita utilizando laços para percorrer toda a matriz de pixels. A imagem é lida e salva na memória principal, primeiro é lido o *magic number* para saber se a imagem é .ppm ou .pgm. Sabendo isso, é lida a quantidade de pixels nas linhas e nas colunas, com as linhas e colunas é alocada a memória para guardar a matriz. Alocada a matriz, a imagem é lida e colocada na matriz

Em seguida a função *smoothImage* recebe a imagem lida e executa o algoritmo de *smooth*, são setados os mesmos valores para a dimensão da imagem e para o *magic number*, dentro do laço são feitas várias verificações para, caso o pixel esteja em algum dos cantos da imagem, os pixels que não existam na imagem sejam setados como 0 (Pixel[0][0] = 0 + 0 + 0 + 0 + Pixel[0][0] + Pixel[0][1] + 0 + Pixel[1][0] + Pixel[1][1]). A cada iteração era verificado se o pixel não estava no canto, calculado o novo valor do pixel e então colocado na nova imagem.

O retorno da função é a nova imagem com o *smooth* calculado com os parâmetros iniciais para leitura da imagem iguais à imagem inicial.

Paralela

A implementação paralela foi inteira baseada na implementação sequencial, através do código sequencial fizemos a paralelização. Infelizmente não conseguimos paralelizar do modo como queríamos a tempo, mas a idéia de paralelização foi a de divisão do processamento em processos e em cada processamento utilizar threads para dividir o trabalho.

Todos os processos teriam acesso à imagem inteira, a imagem seria dividida pela quantidade de processos, assim saberíamos quantas linha cada processos deveria processar, calculando o ínicio através da fórmula $início = pid * (lines/nprocs)$ e o final do processamento por $fim = (pid + 1) * (lines/nprocs)$. Com as linhas divididas entre cada processo, cada processo criaria suas threads com um número único de threads. Assim o toda a imagem ficaria dividida em blocos de processamento, divididos por processos e threads como é possível ver na figura 1.

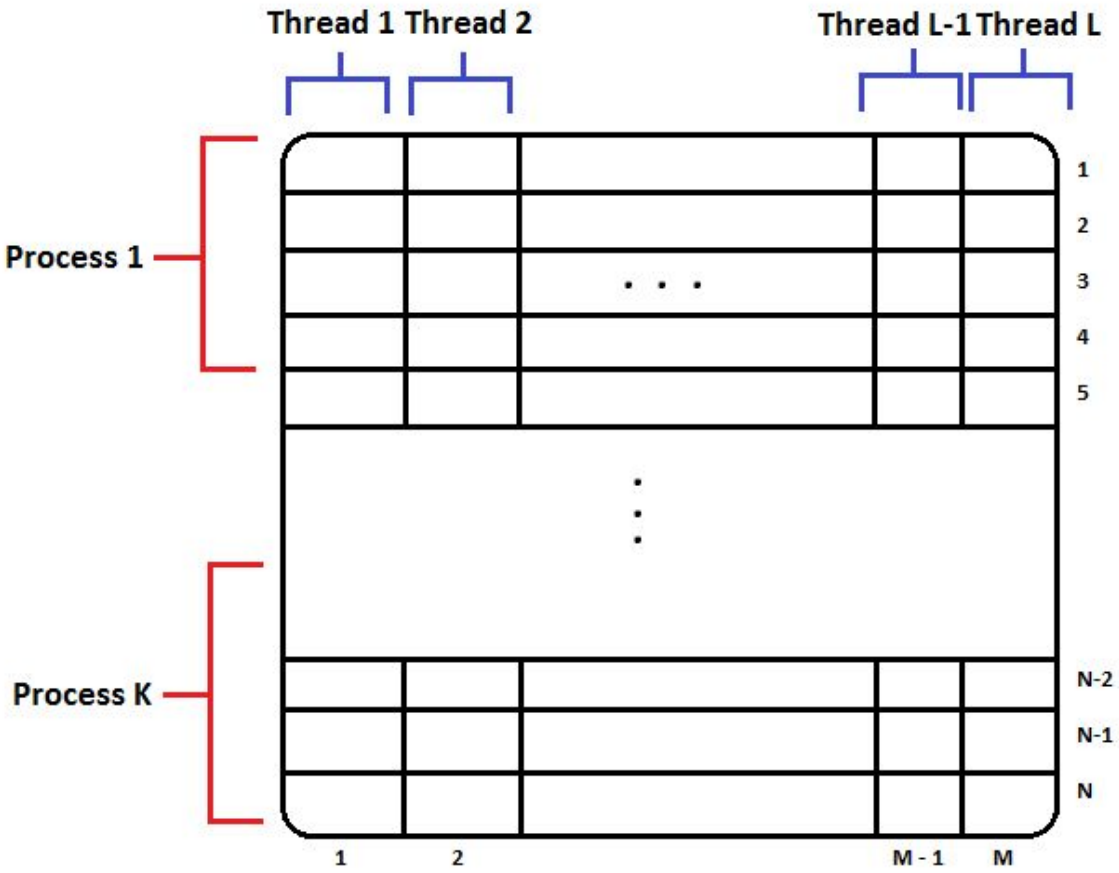

Figura 1. Divisão do trabalho entre processos e threads.

Em cada processo as threads dividem o trabalho entre si dividindo a quantidade de colunas pela quantidade de threads. O cálculo do inicio e o fim das colunas que cada thread deve processar é similar ao cálculo para os processos, $início = tid * (columns/nthreads)$ e o fim é calculado por $fim = (tid + 1) * (columns/nthreads)$. É importante ressaltar que fazemos a verificação para atribuir as linhas ou colunas restantes (caso o número de linhas ou colunas não sejam divisíveis pelo número de processos ou threads, respectivamente) ao último processo ou thread, evitando que alguma linhas ou coluna não passe pelo algoritmo de *smooth*. Com os testes que fizemos, o melhor número de threads encontra foram 8 threads.

Ao fim do cálculo dos novos valores do pixel, a nova imagem é retornada. Quando a função de *smooth* retorna para a main, o cálculo do tempo de execução é parado, a nova imagem salva em um novo arquivo e o tempo de execução do algoritmo é mostrado na tela.

Problemas

Na implementação sequencial não tivemos grandes problemas, porém, na implementação paralela encontramos diversos problemas. Inicialmente fizemos somente uma implementação utilizando OpenMP, a divisão das threads não parecia correta e o arquivo era salvo mas com muitos pixels sendo iguais a zero. O problema foi que estávamos fazendo a alocação da memória para a nova imagem, dentro das threads, fazendo com que alguns valores dos pixels fossem calculados antes de ter a memória para salvá-la. Com a implementação em OpenMP integramos o código ao OpenMPI.

Apesar de funcionar, o tempo de execução da implementação com OpenMPI paralelo era um pouco maior que a implementação sequencial.

O problema era que não estávamos conseguindo criar mais de um processo fazendo com a divisão da imagem fosse feita entre processos. Por falta de tempo para testes não conseguimos recolher os dados com o código utilizando mais de um processo, porém, as fórmulas para dividir o trabalho em mais de um processo foram implementadas, os processos não são adicionados, mas os cálculos são válidos caso tenhamos mais de um processo.

O número de processos e threads afetam diretamente o tempo de execução, somente com um processo e 8 threads o tempo de execução ficou debilitado pois seria necessário dividir a imagem entre mais processos, assim, o tempo de criação de processos e threads não afetaria tanto o tempo final de execução.

Outro problema foi para colocarmos as imagens no cluster. As imagens selecionadas foram colocadas como .JP2, porém nossa implementação suportava somente .ppm e .pgm com *magic number* P5 ou P6. Para evitar perda de tempo para converter as imagens selecionadas, utilizamos as imagens do grupo 22a no mesmo formato que precisávamos e que já estavam no cluster.

Execução

Para executar os códigos, sequencial e paralelo, utilizamos o cluster do Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC). Fizemos o teste com 8 imagens, no formato .ppm para imagens RGB e .pgm para imagens *Gray Scale*. Os testes foram em quatro tipos de tamanho de imagem, pequeno, médio, grande e enorme, em cada tamanho testamos imagens RGB e *Gray Scale*. As mesmas imagens foram usadas para a implementação sequencial e paralelo e cada imagem foi executada 10 vezes.

Na tabela abaixo temos o tempo de execução para imagens na implementação sequencial, bem como a média e o desvio padrão das execuções:

Sequencial							
P - PPM	P - PGM	M - PPM	M - PGM	G - PPM	G - PGM	E - PPM	E - PGM
0.12770	0.12784	0.33169	1.21852	2.02007	8.77121	4.73776	21.0466
0.12803	0.13862	0.33164	1.18241	1.97548	8.77257	4.74101	21.0408
0.12775	0.16411	0.33173	1.18235	1.97540	8.77067	4.73576	21.0330
0.13602	0.13405	0.33172	1.18213	1.97589	8.77188	4.73610	21.0315
0.12766	0.13467	0.33192	1.18231	1.97590	8.77065	4.73571	21.0307
0.12767	0.13098	0.33182	1.18207	1.97550	8.76975	4.73507	21.0303
0.12781	0.15569	0.33184	1.18215	1.97604	8.77137	4.73676	21.0260
0.12782	0.13218	0.33193	1.18238	1.97589	8.77222	4.73643	21.0305
0.12778	0.12791	0.33202	1.18224	1.97569	8.77241	4.73553	21.0431
0.12774	0.13454	0.33201	1.18209	1.97541	8.77123	4.73786	21.0308
Média: 0.12859	Média: 0.13805	Média: 0.33183	Média: 1.18586	Média: 1.98012	Média: 8.77139	Média: 4.73679	Média: 21.0343
Desvio: 0.00247	Desvio: 0.01149	Desvio: 0.00012	Desvio: 0.01088	Desvio: 0.01331	Desvio: 0.00084	Desvio: 0.00165	Desvio: 0.00636

Tabela 1. Tempo de execução Sequencial.

Na tabela seguinte temos o tempo de execução da implementação em paralelo para as mesmas imagens:

Paralelo							
P - PPM	P - PGM	M - PPM	M - PGM	G - PPM	G - PGM	E - PPM	E - PGM
0.21733	0.21354	0.45346	1.52689	2.52734	11.1813	6.10725	26.9270
0.21383	0.21646	0.45465	1.52541	2.52510	11.1138	6.09309	26.7475
0.22123	0.21785	0.45546	1.51846	2.58321	11.2136	6.22589	26.7112
0.21739	0.21192	0.49820	1.53023	2.52349	11.1136	6.12090	26.7503
0.21586	0.21487	0.45062	1.54927	2.52616	12.9393	6.13752	26.6770
0.23749	0.19102	0.45066	1.52034	2.65058	12.9371	6.14683	26.7671
0.21372	0.21620	0.45177	1.52640	2.52203	11.1818	6.09165	26.7077
0.21868	0.18709	0.45628	1.52238	2.53092	11.0965	6.97944	26.6690
0.19953	0.19242	0.44988	1.52141	2.52789	11.1216	6.11696	26.7116
0.19028	0.19197	0.44902	1.53286	2.57683	11.0980	6.05976	26.6907
Média: 0.21453	Média: 0.20533	Média: 0.45700	Média: 1.52736	Média: 2.54935	Média: 11.4996	Média: 6.20792	Média: 26.7359
Desvio: 0.01192	Desvio: 0.01217	Desvio: 0.01393	Desvio: 0.00843	Desvio: 0.03996	Desvio: 0.72026	Desvio: 0.26055	Desvio: 0.07058
Speedup 1.66832	Speedup 1.52616	Speedup 1.37721	Speedup 1.28797	Speedup 1.28747	Speedup 1.31103	Speedup 1.31057	Speedup 1.27106

Tabela 2. Tempo de execução Paralelo.

Como citado, a implementação em paralelo não foi mais rápida que a sequencial por não conseguirmos dividir o trabalho entre diferentes processos. O *Speedup* da execução em paralelo foi maior do que 1 justamente por levar mais tempo para executar do que a implementação sequencial.

Conclusão

O processamento de imagens é algo muito utilizado atualmente, porém, para imagens muito grandes, tal tarefa pode se tornar custosa para computadores comuns, para isso podemos utilizar computadores mais bem preparados para trabalhar com o processamento de imagens grandes. Para facilitar tal trabalho o uso de threads e a divisão do trabalho entre diferentes processos se torna essencial para realizar o processamento de forma eficiente e aproveitar ao máximo o uso do processador.

Neste trabalho, comparamos uma implementação sequencial e outra em paralelo no processamento de imagens grandes. A divisão incorreta da imagem entre os processos, afetou drasticamente o tempo de execução final do implementação paralelo, apesar do algoritmo funcionar. Foi possível ver que a divisão do trabalho em threads e processos deve ser feito de modo correto e bem pensado para que, a implementação paralela, seja válida e menos custosa que a sequencial.

Bibliografia

http://netpbm.sourceforge.net/doc/pgm.html

http://www.drdobbs.com/architecture-and-design/faster-image-processing-with-openmp/184405586

http://staff.city.ac.uk/~sbbh653/publications/OpenMP_SPM.pdf