Acelerando uma Aplicação de Detecção de Pistas com MPI

Gabriel B. Justo[1], **Renato B. Hoffmann**[1], **Adriano Vogel**[1], **Dalvan Griebler**[1,2], **Luiz G. Fernandes**[1]

[1] Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP), Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil.

[2]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC) Faculdade Três de Maio (SETREM), Três de Maio, Brasil.

{gabriel.justo,dalvan.griebler}@edu.pucrs.br

Resumo. *Aplicações de stream de vídeo demandam processamento de alto desempenho para atender requisitos de tempo real. Nesse cenário, a programação paralela distribuída é uma alternativa para acelerar e escalar o desempenho. Neste trabalho, o objetivo é paralelizar uma aplicação de detecção de pistas com a biblioteca MPI usando o padrão Farm e implementando duas estratégias de distribuição de tarefas. Os resultados evidenciam os ganhos de desempenho.*

1. Introdução

O paradigma de processamento de *stream* consiste em computar um fluxo contínuo de dados, usualmente infinito e/ou irregular. Algumas dessas aplicações exigem respostas em tempo real [Vogel et al. 2020], como por exemplo, a detecção de pistas. Neste caso, não é possível atender esse requisito através da execução sequencial. Essa aplicação foi originalmente paralelizada para um ambiente *multi-core* em [Griebler et al. 2017]. Porém, em uma arquitetura *multi-core* a escalabilidade é limitada à uma única máquina. Por outro lado, os *clusters* fornecem um número maior de máquinas que quando devidamente programadas e utilizadas, as aplicações permitem maior escalabilidade.

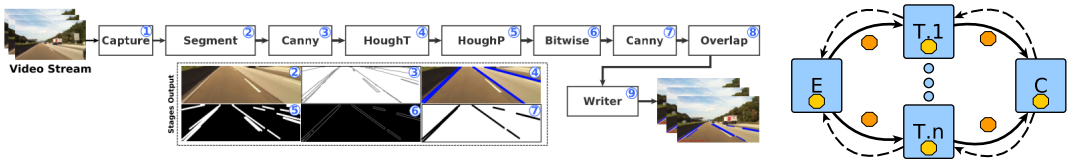
Na literatura, alguns esforços já foram feitos na área de processamento de *stream* com MPI. *MIStream* [Peng et al. 2015] é um exemplo que divide os processos MPI em dois grupos, *data producers* que recebem um *stream* e o dividem entre os processos *data consumers*, que recebem e processam os dados. O resultado do processamento é armazenado localmente e ao final da execução uma função *Reduce* unifica os resultados gerados pelos processos *data consumers*. Sendo assim, o resultado final apenas é obtido após o processamento do último elemento de *stream*. Porém, algumas aplicações, como *Detecção de Pistas* podem ter *streams* infinitos como entrada, o que impossibilita o uso do *MIStream*. Além disso, *MPIStream* não garante a ordem dos elementos de *stream* ao final da execução, dificultando a implementação de aplicações de vídeo.

No trabalho anterior [Justo et al. 2019], foram propostas e implementadas estratégias de paralelismo similares na aplicação *Person Recognition*. Essa aplicação reconhece faces em vídeos. Diferentemente, neste trabalho foi feita uma implementação mais genérica para paralelização da aplicação. Além disso, foram usados dois vídeos de entrada com diferentes cargas de trabalho. Dessa forma, é possível identificar qual o impacto de diferentes cargas nos resultados devido a forma com que as tarefas são distribuídas. O objetivo desse trabalho é a implementação do paralelismo de *stream* com distribuição de tarefas de forma dinâmica e estática na aplicação de detecção de pistas com MPI.

O trabalho está organizado da seguinte forma. Na Seção 2 é explicado o funcionamento da aplicação *Detecção de Pistas* e a estratégia de paralelismo implementada. A Seção 3 apresenta os resultados obtidos e a Seção 4 conclui o trabalho.

2. Implementações

O funcionamento da aplicação *Detecção de Pistas* é demonstrado na Figura 1(a). A função *Capture* recebe um vídeo de entrada, fragmentando-o em uma sequência de *frames*. Três algoritmos de visão computacional são aplicados a essa sequencia. O primeiro divide o *frame* horizontalmente em três partes, sendo que as divisões inferiores da imagem recebem mais foco pois é onde se localizam as faixas da pista. Essa etapa é chamada de *Segment*. Então, é aplicado um filtro para detecção das bordas da imagem, chamado *Canny*. A partir das bordas encontradas, é aplicado um filtro *HoughT* para detectar linhas retas. Nesta etapa também é executado o *Probabilistic Hough Transform* para detectar o início e o fim das faixas.



(a) Aplicação *Detecção de Pistas*. Extraída de [Griebler et al. 2017]. (b) Padrão *Farm*.

Figura 1. Aplicação e paralelismo

A implementação com MPI da aplicação demonstrada na Figura 1(b) utiliza o padrão paralelo *farm* [Griebler et al. 2017, Vogel et al. 2020] em três etapas: Emissor (E), Trabalhadores (T), e Coletor (C) conforme ilustrado na Figura 1(b). No caso do Emissor, foi usado um processo MPI responsável por receber a *stream* de dados e particioná-la para os Trabalhadores. Por sua vez, os Trabalhadores processam uma sequência das funções *Capture*, *Segment*, *Canny*, *HoughT* e *Probabilistic Hough Transform*. Por fim, o Coletor recebe todos os *frames* processados pelos trabalhadores e os reordena para formar o vídeo de saída da aplicação. Nesse estágio, também foi usado um processo MPI.

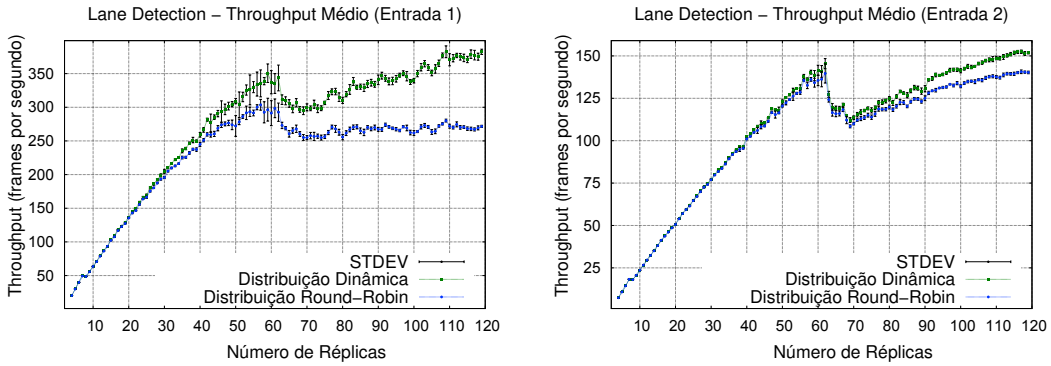
Foram usadas duas estratégias para dividir os *frames* entre os Trabalhadores. A primeira foi a Distribuição *Round-Robin* de tarefas, que divide os *frames* entre os trabalhadores em ordem crescente através de um identificador de cada processo. Já a segunda foi a distribuição dinâmica de tarefas, que distribui os *frames* sob demanda para os trabalhadores. Sendo assim, sempre que um trabalhador termina a execução de sua tarefa, ele envia uma mensagem para o emissor com seu identificador como conteúdo. Dessa forma, o emissor utiliza o identificador recebido para enviar o próximo *frame* de acordo com o método de escalonamento. Nos dois casos, foi usada a função *MPI_Send*. Por isso, tem-se uma comunicação bloqueante entre os processos. Assim, garante-se que o trabalhador apenas receberá a mensagem quando finalizar a execução de sua tarefa atual.

Para enviar os *frames* entre os processos, é necessário dividir a aplicação em duas partes. A primeira parte é uma estrutura caracterizada por informações como o número de linhas, colunas, tipo dos dados armazenados na matriz da imagem e um contador da ordem da imagem no vídeo. A outra parte é um vetor chamado *data* que armazena o

valor de cada *frame* da imagem. Contudo, o tamanho do vetor *data* pode variar pois ele depende da resolução da imagem. Sendo assim, antes de enviá-lo é necessário encaminhar mais uma mensagem com seu tamanho, totalizando três mensagens. Cada *frame* possui um tempo de processamento diferente, necessitando reordenamento dos *frames* gravados no arquivo de saída. Para isso, foi usado o algoritmo proposto em [Griebler et al. 2018]. Nesse caso, a medida em que o Coletor recebe os *frames*, o algoritmo reordena os *frames* usando como base o contador armazenado na *struct*.

3. Resultados

Os teste foram realizados em um *cluster* com 7 máquinas. Cada máquina possui um processador Intel(R) Xeon(R) CPU 2.40 GHz (12 cores-24 threads) e 32 GB de memória RAM. Estas máquinas estão conectadas através de uma rede *gigabit*. O sistema operacional usado foi Ubuntu Server, G++ versão 5.3.0, OpenCV versão 2.4.13 e OpenMPI versão 1.4.5. Os processos Emissor e Coletor foram executados em nodos dedicados. os processos Trabalhadores executados nos 5 nodos restantes (totalizando 60 cores e 120 *Hyper-threads*), o número de processos Trabalhadores variou de 4 a 120 processos distribuídos entre os nodos. Além disso, duas entradas de vídeo distintas foram utilizadas. A entrada 1 tem resolução de 640x360 *pixels* e duração de 61 segundos. Já a segunda entrada tem uma duração total de 142 segundos e simula uma carga desbalanceada, que é uma característica representativa no processamento de *streams*. A variação de carga na Entrada 2 é causada pelos *frames* possuírem diferentes resoluções, simulando o processamento em tempo real de *feed* de vídeo proveniente de câmeras com diferentes especificações ou de um número diferente de dispositivos produzindo dados. O tempo de execução sequencial em um *core* da máquina descrita acima é de 180 e 1160 segundos para as entradas 1 e 2 respectivamente. A métrica usada para avaliar o desempenho das implementações foi o *throughput*, sendo a métrica mais importante no processamento de *streams* [Vogel et al. 2020]. O *throughput* foi calculado considerando o número de tarefas processadas (*frames*) dividido pelo tempo.



(a) *Throughput* médio da *entrada 1*. (b) *Throughput* médio da *entrada 2*.

Os resultados apresentados são uma média e desvio padrão de 10 execuções referente a cada implementação, entrada e número de processos. Os resultados da entrada 1 são demonstrados na Figura 2(a), pode-se observar que a implementação com distribuição dinâmica de tarefas foi mais performática, principalmente quando usando *hyper-threading* no nodos que executaram os processos Trabalhadores. A concorrência excessiva causada

pelo uso de *hyper-threading* resultou em variações no tempo de processamento de cada *frame*. Nesse cenário, a distribuição dinâmica de tarefas foi melhor pois reduz o desbalanceamento de carga através da distribuição de tarefas sob demanda. Na Figura 2(b), os testes com a *entrada 2* mostram uma diferença menor entre o *throughput* das duas estratégias de distribuição de tarefas, principalmente com o número de réplicas inferior a 40. Por outro lado, quando os processos Trabalhadores usaram *hyper-threading* novamente a distribuição dinâmica teve ganhos de desempenho.

Um aspecto que dificultou a implementação paralela da aplicação foi o envio das imagens por mensagens do MPI. Desta forma, é necessário lidar com a transformação dos dados em fluxos de *bytes* e vice-versa, além de lidar com detalhes do objeto `Mat` da biblioteca OpenCV para a correta manipulação. Outra dificuldade foi o desenvolvimento das políticas de distribuição das tarefas, pois requer muita atenção na sincronização das mensagens entre Emissor, Trabalhador e Coletor. Por fim, a dependência de dados que há entre as fases da aplicação sequencial precisaram ser bem estudas para permitir a paralelização.

4. Conclusão

Neste trabalho, foi apresentada uma solução para o processamento de *stream* de vídeo em *clusters*. Foram implementadas duas versões distribuídas com duas estratégias de distribuição de tarefas para os trabalhadores na aplicação *Detecção de Pistas*. Apesar de enviar mais mensagens que a distribuição estática, a implementação com distribuição dinâmica de tarefas tem um desempenho superior. O ganho de desempenho é ainda maior quando a carga é desbalanceada e com o uso de *hyper-threading*. Como trabalho futuro, pretende-se criar uma biblioteca de abstração para o MPI, uma vez que a serialização de dados e a implementação do padrão Farm requer muito esforço de programação paralela.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e FAPERGS.

Referências

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2017). Higher-Level Parallelism Abstractions for Video Applications with SPar. In *International Conference on Parallel Computing*, pages 698–707, Italy. IOS Press.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018). Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. *Journal of Supercomputing*, 75(8):4042–4061.

Justo, G. B., Vogel, A., Griebler, D., and Fernandes, L. G. (2019). Acelerando o Reconhecimento de Pessoas em Vídeos com MPI. In *Escola Regional de Alto Desempenho (ERAD/RS)*, page 4, Três de Maio, BR. SBC.

Peng, I. B., Markidis, S., Laure, E., Holmes, D., and Bull, M. (2015). A Data Streaming Model in MPI. In *Workshop on Exascale MPI*, pages 1–10, USA.

Vogel, A., Rista, C., Justo, G., Ewald, E., Griebler, D., Mencagli, G., and Fernandes, L. G. (2020). Parallel Stream Processing with MPI for Video Analytics and Data Visualization. In *CCIS-High Performance Computing Systems*, pages 102–116. Springer.