

# Paralelização para memória distribuída com MPI

Bruno C. P. D. Rosa, Cristofer A. Oswald

<sup>1</sup>Departamento de informática  
Centro de Tecnologia – Universidade Estadual de Maringá (UEM)  
Maringá – PR – Brasil

**Resumo.** *A busca continua por mais poder computacional e eficiência levou a inevitável ascensão das arquiteturas paralelas, que atualmente dominam o mercado e estão presentes em todo tipo de dispositivo. Estas mudanças também afetaram profundamente o modelo de programação, causando uma troca de paradigma. Programar paralelamente se tornou crucial para o desenvolvimento de aplicações capazes de utilizar o hardware moderno. Neste trabalho, fazemos uso da interface de passagem de mensagens MPI, que será utilizada para realizar experimentos com memória distribuída em diversos computadores, utilizando o algoritmo Fast da CAP benchmark como objeto de testes.*

## 1. Introdução

Na história da ciência da computação os avanços no hardware sempre foram os maiores impulsionadores do aumento do poder computacional, o mesmo software simplesmente roda mais rápido à medida que os processadores apresentam uma maior velocidade de processamento, durante anos cientistas e desenvolvedores de software contaram com isto para atingirem seus objetivos. Esse impulso, porém, vem diminuindo desde 2003 devido a questões de consumo de energia e dissipação de calor, que limitaram o aumento da frequência do clock e o nível de tarefas que podiam ser realizadas em cada período de clock dentro de uma única CPU [Kirk and Wen-Mei 2011]. A indústria de microprocessadores respondeu a estas limitações com a mudança do paradigma dominante nas arquiteturas de computadores, que agora são as arquiteturas paralelas, principalmente sob a forma de processadores multinúcleos.

Um dos métodos de paralelização mais antigos é a paralelização para execução em cluster, onde cada fluxo de execução é alocado em uma máquina diferente num cluster de computadores. Assim cada máquina se torna responsável por parte do processamento, esta estratégia já era utilizada muito antes do surgimento dos processadores multinúcleos. Este modelo, denominado programação paralela com memória distribuída, apresenta vantagens e desvantagens quando comparado ao modelo de paralelização com memória compartilhada. Neste trabalho exploramos os aspectos deste modelo utilizando a técnica de passagem de mensagens utilizando MPI como solução para comunicação entre os dispositivos, analisando os pontos fortes e fracos desta solução.

## 2. Projeto

O programa de alto desempenho *Fast* escrito na linguagem C, foi escolhido para ser usado como objeto de experimentação deste trabalho. A versão original do programa é uma implementação paralela do algoritmo utilizando a técnica de paralelização de laços com a ferramenta *OpenMP*. *Fast* faz parte do CAP benchmark [Souza et al. 2017], um benchmark desenvolvimento com o objetivo principal de testar o desempenho e o gasto de

energia em processadores multinúcleos. O número de fluxos de execução concorrentes e o tamanho do problema são parâmetros de execução do programa.

*Fast*, acrônimo do inglês *Features from Accelerated Segment Test* implementa um método para detecção de cantos em imagens seguindo o padrão estêncil paralelo. É comumente usado para extrair características e mapear objetos em tarefas de visão computacional. Em sua execução, um círculo de 16 pixels é utilizado para testar se um ponto  $p$  candidato é um canto ou não. Um valor inteiro de 1 a 16 é atribuído para cada pixel no círculo em ordem horária. Se todos os  $N$  pixel contínuos do círculo são mais claros que a intensidade do pixel candidato  $p$  somado a um valor limiar  $t$  ou todos mais escuros que a intensidade de  $p - t$ , então  $p$  é classificado como um canto.

Para avaliar o desempenho do programa no modelo paralelo com memória distribuída foi implementada uma versão que utiliza a técnica de passagem de mensagens através de MPI. Uma implementação MPI (*Message Passing Interface*) é uma API para envio de mensagens juntamente de protocolos e especificações semânticas sobre seu funcionamento [Lusk et al. 1996]. A implementação MPI utilizada é a Open MPI, uma versão desenvolvida e mantida por um consórcio de parceiros acadêmicos, pesquisadores e outros interessados. A API Open MPI disponibiliza funções para inicialização do ambiente de execução, envio e recebimento de mensagens entre outras funções como diretivas de sincronização da execução. Além disso, a biblioteca MPIP (*MPI Profiling*) foi utilizada para avaliar os gastos de tempo com a passagem de mensagens entre os computadores, dados estes que serão apresentados neste trabalho.

A estratégia *Mestre-Escravo* foi adotada como modelo de comunicação no desenvolvimento do programa. Neste contexto, o processo principal, intitulado como Mestre, se encarrega de executar tarefas de inicialização do programa como alocação da matriz de pixels, preenchimento aleatório da mesma, envio dos dados para os processos Escravos e recebimento dos resultados computados por eles. Os demais processos, intitulados Escravos, são os processos que irão realizar, de maneira coletiva, a computação efetiva do programa. Estes iniciam sua execução aguardando o recebimento da mensagem do processo Mestre contendo os dados os quais irão computar, executam o método de detecção de cantos sob a sub-área recebida e retornam, na forma de mensagem, o resultado obtido, neste caso o número de cantos encontrados. Todos os envios e recebimentos de dados realizados entre o Mestre e os Escravos ocorrem através de métodos MPI para envio de mensagens, citados anteriormente. Nota-se que no modelo de comunicação adotado a troca de mensagens ocorre exclusivamente entre o processo Mestre para os demais, ou seja, processos Escravos não se comunicam entre eles.

Todos os códigos foram compilados com o compilador *mpicc* parte do Open MPI com otimização O3. As execuções do experimento foram realizadas no laboratório Lin 2 do Departamento de Informática da Universidade Estadual de Maringá em oito computadores com configurações semelhantes executando sistema operacional Ubuntu. Os dados coletados para a avaliação compreendem os resultados de execuções do programa com o problema configurado com o maior tamanho disponível.

### 3. Avaliação

A metodologia de avaliação adotada compreende o uso da ferramenta MPIP para avaliação do tempo gasto na passagem de mensagens e a utilização de dados de tempo

de execução providos pelo próprio algoritmo. Para garantir resultados mais precisos e evitar possíveis falsas conclusões, a avaliação de speedup contempla execuções com quantidades diferentes quantidades de processos (de 2 a 16), sendo que cada uma das configurações foi executada 11 vezes e com a entrada de maior tamanho disponível, assim, a primeira, que tem a finalidade de aquecer a cache, o melhor e o pior resultados são descartados, dessa maneira, os dados utilizados na avaliação são as médias das 8 execuções restantes. Já para a passagem de mensagens, foi realizada uma única execução com o executável sendo linkado com o MPIP.

Sobre a quantidade de processos, é importante notar que o número de máquinas usadas foi diferente para cada uma das configurações e que as execuções sempre contaram com um processo a mais, pois o processo Mestre não é contabilizado na quantidade de processos. As quantidades de máquinas e processos são:

- **2 processos:** 2 máquinas, uma com 2 processos (Mestre + Escravo) e outra com um Escravo.
- **4 processos:** 4 máquinas, uma com 2 processos (Mestre + Escravo) e as outras com um Escravo.
- **8 processos:** 8 máquinas, uma com 2 processos (Mestre + Escravo) e as outras com um Escravo.
- **16 processos:** 8 máquinas, uma com 3 processos (Mestre + dois Escravos) e as outras com dois Escravos.

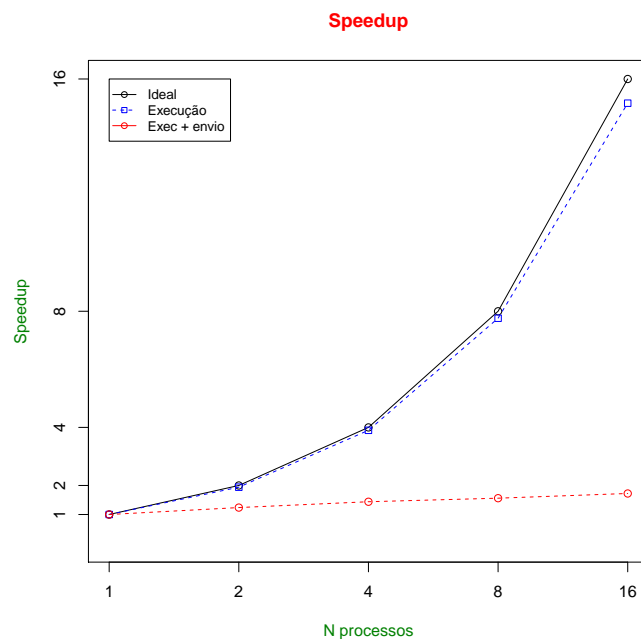
### 3.1. Speedup

Como foco principal deste trabalho, utilizou-se os resultados dos experimentos para avaliar o impacto da paralelização com memória distribuída. O gráfico de speedup da figura 1 é composto por três componentes: Speedup ideal, Speedup obtido desconsiderando o overhead e por fim Speedup real considerando overhead, este overhead será explicado posteriormente.

O speedup de execução, mostrado na figura 1, é bastante expressivo e pode ser justificado através de uma análise no algoritmo do programa. A estrutura de execução garante que cada fluxo de execução compute parte do problema de maneira independente, isto é, sem interrupções destinadas a comunicação entre os processos, o que caracteriza um problema sem condições de corrida. Esta característica do algoritmo possibilita este grande ganho de desempenho quando desconsiderado o custo do overhead.

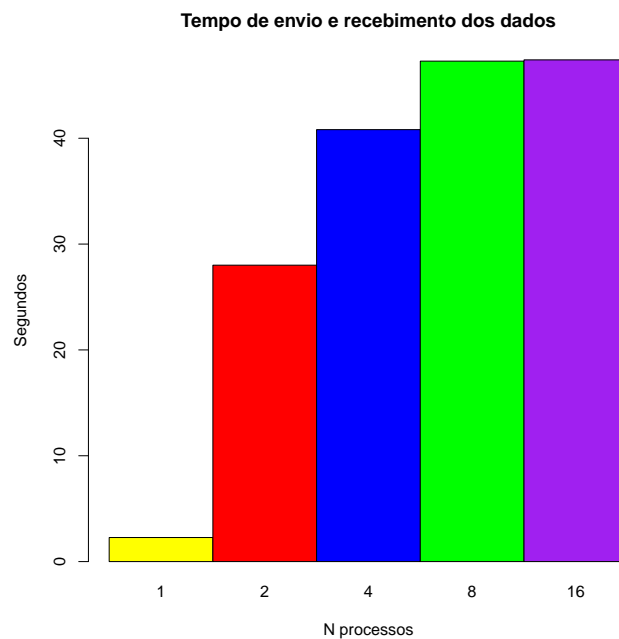
Percebe-se entretanto que o speedup real obtido é baixo, isto por que quando se soma o tempo de overhead do envio e recebimento de mensagens entre os processos, o tempo total de execução se torna muito alto. Na figura 2 temos a relação do tempo de inicialização da execução com o número de fluxos de execuções. O tempo da inicialização é a soma do tempo total gasto com alocação dos dados, geração do problema do tamanho dado como parâmetro e o tempo de overhead de envio das mensagens durante a execução do programa, sendo esta a parte mais longa da inicialização.

Na execução sequencial, o tempo de inicialização é perto de 0 pois overhead de envio das mensagens é inexistente neste caso. Observa-se que a partir de 2 processos este tempo se torna muito alto e esta é a causa do speedup real baixo obtido. Apesar do algoritmo possibilitar um speedup semi-ideal a paralelização com memória distribuída acarreta custos adicionais de comunicação e discutiremos tal implicação e seus efeitos no

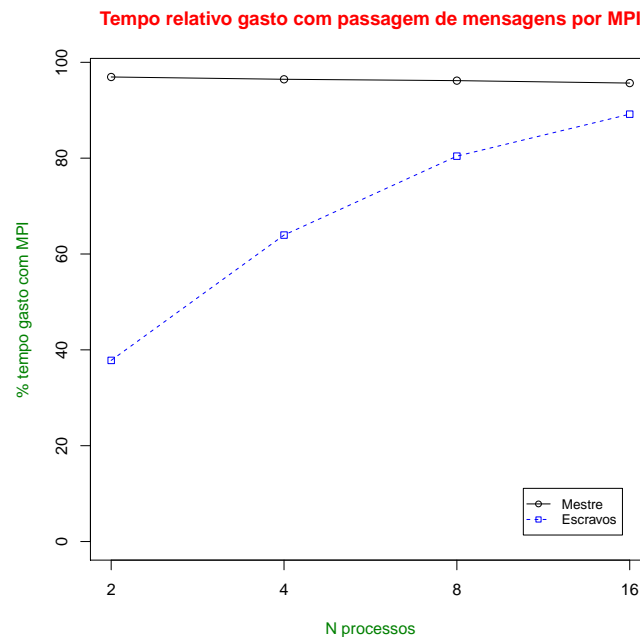


**Figura 1. Gráfico de speedup**

tópico seguinte. É importante citar que o tempo de inicialização para a execução com 8 e 16 fluxos paralelos é quase o mesmo pois para ambas configurações foram utilizadas oito computadores, sendo que na segunda o número de processos por computador é dobrado, porém o custo do overhead de envio das mensagens é praticamente o mesmo já que o número de computadores é o mesmo.



**Figura 2. Custo do overhead**



**Figura 3. Tempo relativo gasto com o MPI**

Na figura 3, podemos ter a real noção do impacto do custo das trocas de mensagens. Vemos que, conforme a quantidade de processos e, conseqüentemente, a de computadores aumenta, o custo das trocas chega a mais de 90% do tempo. Na última execução, onde a quantidade de processos era de 16, os dados de tempo de execução do programa mostraram que cerca de 5 segundos do tempo de execução foi de computação efetiva e o tempo da troca de mensagens foi de mais de 40.

O mestre aparece em todos os casos com quase 100% de utilização do MPI pois a sua parte no programa, como já foi citado, é apenas de gerar e enviar os dados que serão utilizados pelos escravos, além de receber o resultado final, assim, grande parte do tempo utilizado por ele é esperando a resposta da computação ou enviando os dados iniciais, o que cai um pouco conforme a quantidade de processos aumenta, pois, com mais processos, o tempo de computação efetiva diminui, logo o tempo de espera pelas respostas também diminui.

Outra coisa que podemos notar avaliando as figuras 2 e 3, é que o tamanho das mensagens não influencia no tempo gasto para as trocas. O quantidade de dados utilizados pelo algoritmo como um todo é o mesmo em todas as execuções, visto que a entrada é a mesma, mas a quantidade de dados utilizados por cada processo diminui conforme a quantidade de fluxos aumenta, pelo fato de que, com mais processos dividindo o trabalho, menos dados são necessários por cada um. Com isso em mente, podemos afirmar que, mesmo com mensagens maiores nas execuções com menos fluxos, isto não é um fator determinante para custo de envio das mensagens, por outro lado, o maior determinante para esse custo é a quantidade de computadores envolvidos na computação.

## 4. Conclusão

Com as análises feitas na seção anterior, podemos afirmar que há um grande ganho no tempo de execução do algoritmo, mostrado pelo gráfico de speedup, mas o gasto com a passagem das mensagens, isto é, dos dados, é muito grande, tornando o ganho real mínimo. Mesmo assim, não podemos considerar a implementação deste problema utilizando memória compartilhada totalmente ineficiente, pois o ambiente utilizado para os experimentos não é ideal e o tamanho máximo da entrada é pequeno para uma avaliação real do tempo de execução.

Como explicado anteriormente, o aumento do custo das trocas de mensagens ocorre com o aumento na quantidade de computadores e não no aumento dos dados. Assim, com entradas maiores, o tempo de envio dos dados pode se tornar baixo comparado ao tempo de execução real, visto que algoritmo tem aumento quadrático para com sua entrada, fazendo com que o ganho no tempo de execução seja efetivo.

Por fim, nota-se que, se utilizado um cluster de computadores específico para este modelo de paralelização, poderíamos obter melhores resultados, visto que neste ambiente a comunicação entre os computadores é otimizada para este paradigma paralelo, reduzindo os custos da troca de mensagens.

## Referências

- Kirk, D. B. and Wen-Mei, W. H. (2011). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.
- Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22:789–828.
- Souza, M. A., Penna, P. H., Queiroz, M. M., Pereira, A. D., Góes, L. F. W., Freitas, H. C., Castro, M., Navaux, P. O., and Méhaut, J.-F. (2017). Cap bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience*, 29(4).