MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21c/2018/05.25.12.46-TDI

AVALIAÇÃO DE TÉCNICAS AVANÇADAS DE COMUNICAÇÃO MPI NA EXECUÇÃO DO MODELO BRAMS

Carlos Renato de Souza

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Stephan Stephany, e Jairo Panetta, aprovada em 28 de maio de 2018.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/3R6QKR8>

INPE
São José dos Campos
2018

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GBDIR)
Serviço de Informação e Documentação (SESID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
E-mail: pubtc@inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21c/2018/05.25.12.46-TDI

AVALIAÇÃO DE TÉCNICAS AVANÇADAS DE COMUNICAÇÃO MPI NA EXECUÇÃO DO MODELO BRAMS

Carlos Renato de Souza

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Stephan Stephany, e Jairo Panetta, aprovada em 28 de maio de 2018.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/3R6QKR8>

INPE
São José dos Campos
2018

Dados Internacionais de Catalogação na Publicação (CIP)

Souza, Carlos Renato de.

So89a Avaliação de técnicas avançadas de comunicação MPI na execução do modelo BRAMS / Carlos Renato de Souza. – São José dos Campos : INPE, 2018.

xvi + 92 p. ; (sid.inpe.br/mtc-m21c/2018/05.25.12.46-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2018.

Orientadores : Drs. Stephan Stephany, e Jairo Panetta.

1. BRAMS. 2. MPI. 3. Shared Memory. 4. Modelo numérico de previsão. I.Título.

CDU 551.511.61

Aluno (a): *Carlos Renato de Souza*

Título: "AVALIAÇÃO DE TÉCNICAS AVANÇADAS DE COMUNICAÇÃO MPI NA EXECUÇÃO DO MODELO BRAMS"

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Mestre* em *Computação Aplicada*

Dr. Celso Luiz Mendes

Presidente / INPE / São José dos Campos - SP

( ) Participação por Video - Conferência

Dr. Stephan Stephany

Orientador(a) / INPE / SJCampos - SP

( ) Participação por Video - Conferência

Dr. Jairo Panetta

Orientador(a) / ITA / São José dos Campos - SP

( ) Participação por Video - Conferência

Dr. Haroldo Fraga de Campos Velho

Membro da Banca / INPE / São José dos Campos - SP

( ) Participação por Video - Conferência

Dr. Álvaro Luiz Fazenda

Convidado(a) / UNIFESP / São José dos Campos - SP

(X) Participação por Video - Conferência

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

São José dos Campos, 28 de maio de 2018

RESUMO

O Centro de Previsão de Tempo e Estudos Climáticos (CPTEC) do Instituto Nacional de Pesquisas Espaciais (INPE) executa operacionalmente modelos numéricos de previsão de tempo e clima. Em particular, utiliza o modelo regional BRAMS (*Brazilian developments on the Regional Atmospheric Modeling System*). Esse modelo foi paralelizado com a biblioteca de comunicação por troca de mensagens *Message Passing Interface* (MPI) e é executado no supercomputador Tupã do CPTEC, o qual é composto de mais de um milhar de nós de processamento. Cada nó possui dois processadores multinúcleos numa arquitetura de memória compartilhada. Na execução paralela do modelo BRAMS, seu domínio de cálculo é dividido em subdomínios entre processos executados nos núcleos dos muitos nós. Eventuais dependências de dados entre subdomínios diferentes implicam na troca de mensagens MPI entre processos sejam eles do mesmo nó ou não. O BRAMS utiliza troca de mensagens MPI bilateral no modo assíncrono e sem bloqueio, disponível desde a primeira versão do padrão MPI. O padrão MPI tem evoluído, oferecendo novas técnicas para otimizar a comunicação entre processos. Assim, visando otimizar o desempenho da comunicação, o padrão MPI-2 introduziu a chamada comunicação unilateral por acesso remoto à memória, que permite a um processo pode fazer leituras ou escritas por meio de funções MPI na memória de outro, seja do mesmo nó ou não, permitindo a troca de dados entre processos, sem que o processo alvo participe da comunicação explicitamente. A comunicação unilateral foi aperfeiçoada no MPI-3, mas uma nova funcionalidade foi introduzida, a comunicação unilateral por memória compartilhada, que permite a processos MPI executados num mesmo nó definirem uma janela comum de memória local e efetuar leituras e escritas diretas na área da janela de outros processos locais. Este trabalho visa avaliar o desempenho da execução paralela do modelo regional BRAMS ao utilizar a comunicação unilateral de memória compartilhada na comunicação intra-nó e mantendo a comunicação bilateral assíncrona e sem bloqueio na comunicação inter-nó e preservando a mesma divisão de domínio de sua versão paralela original.

Palavras-chave: BRAMS. MPI. *Shared Memory*. Modelo Numérico de Previsão.

EVALUATION OF ADVANCED TECHNIQUES FOR MPI COMMUNICATION IN THE EXECUTION OF THE BRAMS MODEL.

ABSTRACT

The Center for Weather Forecasts and Climate Studies (CPTEC) of the Brazilian National Institute for Space Research (INPE) executes several climate and weather numerical forecast models on an operational basis, specifically using the regional model nominated BRAMS (Brazilian developments on the Regional Atmospheric Modeling System). This model was parallelized using the Message Passing Interface (MPI) communication library, being executed by the CPTEC's "Tupã", a supercomputer composed of hundreds of processing nodes. Each node has two multi-core processors in a shared memory architecture. In the paralell execution of BRAMS, its calculation domain is divided among processes executed in the cores of the many nodes. Data dependencies between different subdomains require the exchange of MPI messages between the processes, either intra-node or inter-node. BRAMS employs asynchronous non-blocking point-to-point communication, available since the first version of the MPI standard. The MPI standard has evolved through the years bringing new techniques to optimize the communication between processes. Thus MPI-2 introduced the one-sided communication by remote memory access. It allows a process to execute reads or writes using MPI functions to the memory of other process, either in the same node or not, exchanging data betweem processes without the explicit cooperation of the target process. One-sided communication was improved in the MPI-3 standard, but a new technique was added, the shared memory one-sided communication. MPI processes executed in the same computational node may define a common shared memory window and execute direct reads and writes in the window part of another process. The purpose of this work is to evaluate the parallel performance of the BRAMS model using the shared memory one-sided communication for the intra-node communication while keeping the asynchronous non-blocking point-to-point inter-node communication, and preserving the domain decomposition of its original parallel version.

Keywords: BRAMS. MPI. Shared Memory. Numerical Model Forecast.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

SUMÁRIO

Pág.

1 INTRODUÇÃO

O Centro de Previsão de Tempo e Estudos Climáticos (CPTEC) do Instituto Nacional de Pesquisas Espaciais (INPE) é o principal centro brasileiro de meteorologia, gerando previsões de tempo e clima de alta qualidade em caráter operacional. Atua também em pesquisa e desenvolvimento com o objetivo de aprimorar suas previsões.

Os modelos numéricos de previsão de tempo/clima são a principal ferramenta utilizada pelos meteorologistas na geração de previsões. Os modelos de previsão são programas que implementam modelos matemáticos baseados em equações diferenciais de forma a simular o estado da atmosfera, bem como fenômenos atmosféricos de interesse. Em conjunto com dados observacionais no presente, essas simulações permitem ao meteorologista prever as condições atmosféricas futuras.

Existem modelos de previsão de cobertura espacial global e aqueles de cobertura espacial regional. O CPTEC utiliza o MCGA (Modelo de Circulação Global da Atmosfera) (BONATTI, 1996) como principal modelo de previsão de tempo e de clima global. E, dentre outros, executa modelo BRAMS (*Brazilian developments on the Regional Atmospheric Modeling System*) (BRAMS, 2017) como modelo regional de previsão de tempo e também ambiental.

Os modelos globais possuem como cobertura espacial o globo terrestre inteiro, então não necessitam de condições de contorno laterais nas equações, apenas as condições iniciais, configurando um problema de valor inicial (PVI). Entretanto, um modelo regional possui, além do problema de valor inicial, também o problema de valor de contorno (PVC), pois sua cobertura espacial é limitada a uma determinada região (um sub-domínio do globo terrestre). Dessa forma, as bordas do domínio dos modelos regionais precisam ser atualizadas periodicamente por dados de outros modelo globais.

Os modelos de previsão de tempo são programas computacionalmente custosos, uma vez que a precisão da previsão exige resoluções espaciais altas e, consequentemente, para garantir sua estabilidade numérica, resoluções temporais também altas. Assim, em sua grande maioria são paralelizados e exigem supercomputadores para executá-los em tempo hábil para divulgação das previsões. A maioria dos modelos numéricos foi paralelizada utilizando-se a biblioteca de comunicação por troca de mensagens *Message Passing Interface* (MPI),

que implica na execução concorrente de vários processos. No caso de um supercomputador, os processos são executados pelos núcleos dos processadores que compõem os nós computacionais, os quais são interligados por uma rede de comunicação que permite altas taxas de transmissão, ou seja, com alta largura de banda. Cada nó computacional envolve um espaço de endereçamento único de memória, caracterizando-se como uma máquina de memória compartilhada. Por outro lado, os diversos nós que compõem o supercomputador constituem uma máquina de memória distribuída. Cada processo é executado independentemente, sendo que as trocas de mensagens devidas a dependências de dados entre processos, bem como a sincronização entre eles, demandam o uso de funções MPI, as quais podem retardar sua execução. Trocas de mensagens entre processos do mesmo nó são efetuadas empregando sua arquitetura de memória, enquanto que trocas de mensagens entre processos de nós diferentes fazem uso da rede de interconexão entre nós. No caso de um modelo numérico, o domínio é particionado em subdomínios, a cargo dos diversos processos.

O BRAMS foi paralelizado com a versão MPI 1.0 (MPI 1.0, 1994), que dá suporte à comunicação (troca de mensagens) entre os processos na forma bilateral (ou ponto-a-ponto) ou coletiva (envolvendo vários processos). A comunicação bilateral implica em ações de envio e recebimento da mensagem exigindo a chamada conjunta de funções MPI em ambos os processos envolvidos na comunicação. Por outro lado, comunicações coletivas visam otimizar a troca de mensagens envolvendo vários processos. Além da comunicação bilateral, a partir de sua versão 2.0 (MPI 2.0, 1998), o MPI oferece a comunicação unilateral, na qual um processo pode acessar a memória remota de outro processo, sendo também conhecida como *Remote Memory Access* (RMA), em que apenas esse processo participa diretamente da comunicação. A comunicação unilateral foi implementada para tentar diminuir atrasos causados pela comunicação bilateral convencional.

A versão 3.1 (MPI 3.1, 2015) do MPI aprimorou a comunicação unilateral RMA, mas introduziu uma nova forma de comunicação unilateral específica para processos executados num mesmo nó computacional de memória compartilhada, visando otimizar a comunicação entre processos para essa arquitetura de memória. A comunicação unilateral *Shared Memory* (SHM) permite definir e tornar visível aos processos locais ao nó uma janela de memória compartilhada, a qual pode incluir parte da memória de cada processo local. Assim, processos locais podem acessar parte da memória de outro(s) processo (s) diretamente por meio de leituras e escritas, sem a necessidade do uso de funções MPI, a um custo

teoricamente menor que aquele da comunicação bilateral convencional.

Desde 2010, o CPTEC conta com o supercomputador Tupã CRAY-XE6, uma máquina massivamente paralela composta por 1.304 nós computacionais de processamento, cada um com 2 processadores 12-*core* de 2,1 GHz cada. Cada nó computacional tem 32GB de memória compartilhada e a rede proprietária (Cray) Gemini interliga todos os processadores de todos os nós por uma topologia Torus-3D, numa interligação que trata de maneira similar a comunicação entre processadores do mesmo nó ou de nós distintos.

Recentemente, o CPTEC recebeu uma extensão do supercomputador CRAY Tupã, composta por nós do modelo CRAY XC-50, que já foram montados e instalados fisicamente, mas ainda encontram-se atualmente em fase de testes. O XC-50 constitui assim um novo conjunto de 104 nós computacionais de processamento, cada um com 2 processadores Intel Xeon Gold 1648 Skylake de 2,4 GHz de 20 *cores* cada, somando 40 *cores* por nó, e 4.160 *cores* no total, sendo que cada nó novo tem 192 GBytes de memória DDR4-2666. Além disso, foi adicionado um *Storage* do tipo *Lustre File System*, que representa um acréscimo de 986 Tbytes de armazenamento para o Tupã.

1.1 Objetivo

Este trabalho, visa avaliar a potencial melhora de desempenho paralelo do modelo BRAMS ao se utilizar a comunicação unilateral MPI *Shared Memory* para processos executados num mesmo nó computacional, mas preservando a estrutura MPI bilateral paralela original de comunicação existente para processos executados em nós computacionais diferentes. Essa avaliação foi feita considerando-se um subconjunto de dados e programas relativos ao módulo da dinâmica do modelo BRAMS, constituindo o que aqui se denomina por dinâmica isolada do BRAMS, de forma a permitir avaliar o desempenho de uma forma mais objetiva, sem requerer a execução completa do modelo.

De maneira geral, modelos numéricos de previsão de tempo e de clima simulam a evolução temporal do estado da atmosfera considerando uma grade tridimensional (latitude, longitude e altitude/altura), sendo paralelizados com a divisão do domínio horizontal (latitude, longitude). A dinâmica do modelo envolve a resolução de equações diferenciais parciais da Mecânica dos Fluidos que determinam o movimento das massas de ar, enquanto que as parametrizações permitem simular fenômenos de pequena escala ou então fenômenos muito complexos. En-

quanto que as parametrizações são calculadas independentemente para cada ponto da grade horizontal, isto é, para a coluna de ar correspondente ao ponto de grade, na dinâmica ocorrem dependências de dados entre pontos de grade adjacentes, devido ao movimento das massas de ar, sendo que os cálculos não podem ser feitos independentemente. Essa foi a razão de se escolher a dinâmica do modelo BRAMS, que exige comunicação entre processos a cargo de subdomínios adjacentes.

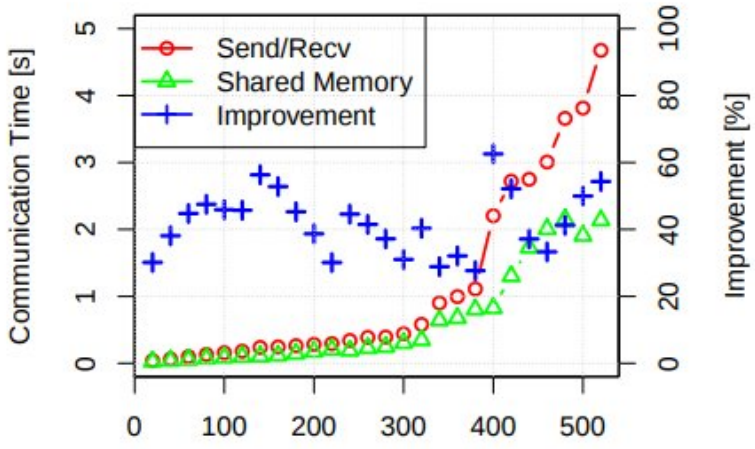
Neste trabalho, refere-se à comunicação MPI bilateral como sendo convencional, e à comunicação unilateral MPI SHM como sendo avançada, por ser mais recente e por ser potencialmente menos custosa.

1.2 Revisão bibliográfica

Há publicações referentes à comunicação MPI unilateral RMA, mas a comunicação MPI unilateral SHM ainda é relativamente recente, sem muitas publicações na área de processamento de alto desempenho. Uma possível razão, discutida adiante neste trabalho, é que a implementação corrente dessa comunicação ainda não é eficiente.

Hoefler et al. (2013) utilizaram duas versões de um programa que implementa um problema exemplo para comparar o desempenho paralelo da comunicação MPI unilateral SHM com o da comunicação MPI bilateral convencional. Ambas as versões do programa foram escritas na linguagem C e executadas numa máquina de memória compartilhada. O problema exemplo calcula um estêncil de 5 pontos oriundo da aplicação do método de diferenças finitas para resolução da equação de Poisson bidimensional na simulação da propagação de calor numa malha discretizada. Os autores consideraram uma malhar quadrada de N x N pontos decomposta em subdomínios iguais em ambas as dimensões. A cada passo de tempo, a atualização dos pontos de grade na fronteira entre 2 subdomínios quaisquer requer a comunicação entre os processos escalonados para esses subdomínios. A máquina em questão tem um único processador AMD Opteron de 2,2 GHz de 6 núcleos, e a paralelização utilizou apenas 2 processos MPI. A Figura 1.1 apresenta os resultados de desempenho da execução paralela dessas versões do programa, demonstrando que houve um ganho de desempenho significativo ao utilizar-se a comunicação MPI unilateral SHM.

Figura 1.1 - Tempos de comunicação da versão do programa exemplo com comunicação bilateral Send/Recv e da versão com comunicação unilateral *Shared Memory*.

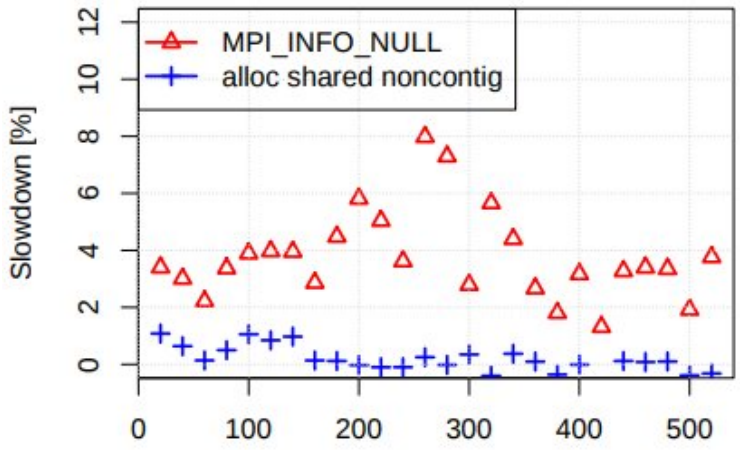


Eixo vertical esquerdo: Tempos de comunicação. Eixo vertical direito: Diminuição porcentual do tempo da segunda versão em relação à primeira. Eixo horizontal: Tamanho de grade N.

Fonte: Hoefler et al. (2013).

A Figura 1.1 mostra os tempos de comunicação da versão com comunicação MPI bilateral (Send/Recv), representada pela linha pontilhada em vermelho, e os tempos da versão com comunicação *Shared Memory* em linhas verdes com triângulos; as cruzes em azul representam a melhoria da versão *Shared Memory* sobre a versão Send/Recv. O eixo horizontal refere-se ao tamanho da malha N, enquanto que o eixo vertical à esquerda mostra os tempos de comunicação total em segundos (válidos para as curvas vermelhas e verdes) e o eixo vertical à direita representa a diminuição porcentual do tempo de comunicação da versão *Shared Memory* em relação à versão Send/Recv (valores válidos para as cruzes em azuis), a qual variou entre 30% e 60%. Os autores atribuem essa diminuição ao acesso direto à memória compartilhada e à não necessidade das chamadas de funções MPI relativas à comunicação bilateral convencional.

Figura 1.2 - Diminuição porcentual do *speedup* da versão com comunicação unilateral *Shared Memory* em relação à versão com comunicação bilateral Send/Recv na atualização dos pontos interiores de cada subdomínio da grade para as opções de alocação contígua (`MPI_INFO_NULL`) e não-contígua (`alloc shared noncontig`).



Fonte: Hoefler et al. (2013).

Adicionalmente, os autores compararam os tempos de computação da versão *Shared Memory* para atualização dos pontos interiores dos subdomínios ao se utilizar a opção de alocação não-contígua de memória (opção `alloc shared noncontig`) ou a opção de alocação contígua (opção `MPI_INFO_NULL`, *default*). Considerando-se o "*speedup*" obtido pela versão *Shared Memory* em relação à versão Send/Recv relativo a esses cálculos, a Figura 1.2 ilustra a desaceleração porcentual desse *speedup* ao se utilizar alocação de memória contígua (triângulos vermelhos), que chegou a ser até 8% pior do que com alocação não-contígua. Analogamente à figura anterior (1.1), o eixo horizontal refere-se ao tamanho de grade N. Nessa mesma figura, a desaceleração porcentual com a opção de alocação não-contígua deveria ser nula, uma vez que não poderia haver desaceleração em relação a ela própria (cruzes azuis), mas os autores atribuem isso a ruído nas medidas de tempo, citando que esse ruído chega a 1,7%. Os autores concluem que o pior desempenho da alocação contígua deve-se a *false sharing* e ao fato da janela estar alocada no processo com rank 0. Conclui-se que a opção de alocação não-contígua de memória para a comunicação unilateral *Shared Memory* permite melhor desempenho pois otimiza o acesso à memória.

Deve-se notar que o programa exemplo do trabalho acima comparou a comunicação MPI unilateral SHM com a comunicação bilateral convencional que utiliza as funções `MPI_Send()` e `MPI_Recv()`, que são com bloqueio, enquanto que o BRAMS, bem como programas MPI em geral, utilizam a comunicação assíncrona e sem bloqueio com as funções `MPI_Isend()` e `MPI_Irecv()`, que permitem a sobreposição de computação e comunicação, possibilitando um ganho de desempenho em relação às primeiras.

1.3 Organização do texto

Nesta seção, será apresentada uma breve descrição dos demais capítulos desta dissertação de mestrado, mostrando sua organização:

- Capítulo 2: Biblioteca de comunicação por troca de mensagens MPI
  Neste capítulo, é apresentada uma descrição geral da biblioteca MPI, descrevendo as forma de comunicação suportadas: a comunicação bilateral convencional, a comunicação unilateral por acesso remoto à memória e a recente comunicação unilateral de memória compartilhada, objeto deste trabalho.

- Capítulo 3: O modelo meteorológico BRAMS
  Descrevem-se as características gerais do modelo BRAMS: organização, etapas de execução, paralelização MPI, bem como as principais estruturas de dados e funções usadas na comunicação.

- Capítulo 4: Implementação da comunicação SHM no BRAMS
  Neste capítulo, são mostradas todas as modificações feitas na dinâmica isolada do BRAMS, as quais foram necessárias para a implementação da comunicação unilateral de memória compartilhada.

- Capítulo 5: Resultados de desempenho paralelo
  Considerando-se um problema exemplo e também a dinâmica isolada do BRAMS, são apresentadas as análises de desempenho comparando-se a comunicação bilateral com a comunicação unilateral de memória compartilhada, aplicável apenas a processos do mesmo nó computacional.

- Capítulo 6: Conclusão
  São expostas as conclusões e comentários finais do trabalho, além de algumas sugestões para futuras continuações desta pesquisa.

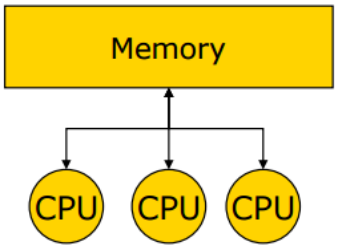
2 BIBLIOTECA DE COMUNICAÇÃO POR TROCA DE MENSAGENS MPI

2.1 A biblioteca MPI

Os fabricantes de computadores começaram a produzir máquinas com mais de um processador na década de 1980. Essas máquinas utilizavam a arquitetura de memória central, onde todos os processadores endereçavam a mesma memória. O paralelismo nessas máquinas era explorado por meio de tarefas, acessíveis ao programador por meio de bibliotecas produzidas pelo fabricante. As bibliotecas forneciam rotinas para que um programa Fortran (uma tarefa), executando em um processador, criasse outra tarefa e a executasse em outro processador, gerando paralelismo (CINDY; WEEKS, 1986). Também haviam rotinas que sincronizavam tarefas em execução e rotinas que aguardavam o término de tarefas anteriormente criadas, destruindo o paralelismo.

Ao longo dos anos, as invocações das rotinas da biblioteca de tarefas foram substituídas por diretivas no compilador Fortran. Cabia ao compilador Fortran transformar as diretivas em invocações das rotinas da biblioteca. Essas diretivas a princípio eram proprietárias dos fabricantes. Ao longo do tempo, surgiram diretivas padronizadas (como OpenMP) que foram aceitas pela comunidade de programadores. Também ao longo do tempo, as tarefas em computadores de memória central foram implementadas por *threads* e surgiram bibliotecas padronizadas para *threads* (como POSIX *pthreads*). Esse modelo de programação é hoje chamado de "*fork-join*".

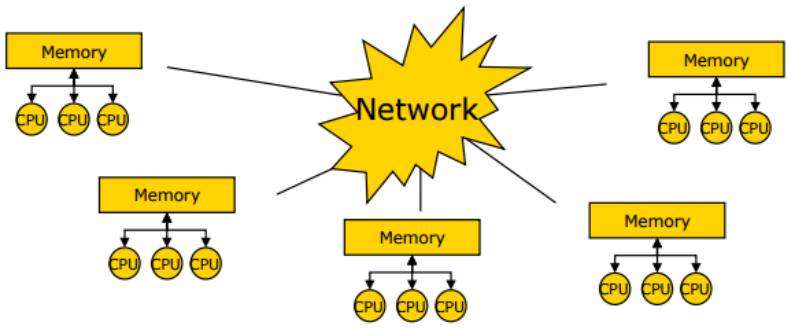
Figura 2.1 - Multiprocessador com vários processadores que utilizam uma memória compartilhada.



Fonte: Gropp (2016).

Com o avanço da tecnologia e dos anos, surgiram computadores com vários processadores e com memórias distribuídas, chamados de "*clusters*". Os *clusters* eram constituídos basicamente de várias ilhas de processamento, chamados de "nós". No final dos anos 90, tipicamente, cada nó possuía um único processador com sua própria memória e os nós eram interligados por uma rede rápida. Nos *clusters*, o processador de um determinado nó não acessa as memórias de outros nós. Assim, na programação paralela, a eventual dependência de dados entre processos executados em nós diferentes exige a troca de mensagens entre esses processos. Obviamente, *clusters* e supercomputadores atuais são formados de centenas ou milhares de nós computacionais interligados por uma rede rápida, sendo cada nó constituído de dois ou mais processadores multi-núcleos numa arquitetura de memória compartilhada, como ilustrado na Figura 5.7.

Figura 2.2 - *Cluster* de memória distribuída, composto por vários nós computacionais multiprocessados de memória compartilhada.



Fonte: Gropp (2016).

Ao longo dos anos, surgiram diversas bibliotecas proprietárias de troca de mensagens, dificultando a portabilidade de programas paralelos. Os desenvolvedores de software a cargo dessas bibliotecas uniram-se aos usuários e à comunidade acadêmica em um fórum específico do qual resultou a criação da biblioteca de comunicação por troca de mensagens chamada de *Message Passing Interface* (MPI). Essa biblioteca acabaria se tornando um padrão, sendo a mais utilizada até os dias de hoje para programação paralela.

O MPI não é uma linguagem de programação, mas constitui uma API (*Application*

Programming Interface) e inclui uma biblioteca de funções para comunicação e sincronização entre processos. Criada em Maio de 1994, sua versão 1.0 (conhecida como MPI-1) (MPI 1.0, 1994) aplica-se exclusivamente a programas escritos em linguagem Fortran 77, e C.

Várias versões do MPI foram publicadas posteriormente à versão 1.0. Em 1997/8 a versão 2.0 (MPI 2.0, 1998) foi publicada e em 2015 a última versão 3.1 (MPI 3.1, 2015). As novas versões MPI são retro-compatíveis com as versões anteriores, ou seja, um programa escrito com MPI 1.0 pode ser executado com MPI 3.0.

Desde sua primeira versão, o MPI fornece comunicações que envolvem apenas dois processos (denominadas ponto a ponto ou bilaterais) e comunicações que envolvem todos os processos de um comunicador, ou seja, grupo específico de processos, as quais são denominadas comunicações coletivas. A seguir, nas Seções 2.2, 2.3 e 2.4 comentam-se as versões principais do MPI (1.0, 2.0 e 3.0, respectivamente) e as formas de comunicação que foram adicionadas incrementalmente a essas versões: a comunicação bilateral, a comunicação unilateral por acesso remoto à memória e a comunicação unilateral de memória compartilhada. A última, objeto deste trabalho, permite leituras/escritas diretas na memória compartilhada pelos processos executados localmente num mesmo nó computacional. A seção 5.2 apresenta um exemplo de comunicação unilateral de memória compartilhada e também um exemplo com comunicação híbrida, que utiliza a comunicação bilateral e também a comunicação unilateral de memória compartilhada. A primeira aplica-se a processos de nós distintos, enquanto que a segunda, a processos executados localmente num mesmo nó.

2.2 Padrão MPI 1.0 e a comunicação bilateral

Na comunicação bilateral, existem diferentes formas de envio e recebimento das mensagens: síncrono ou assíncrono, e com ou sem bloqueio.

Na comunicação síncrona o processo que envia a mensagem é suspenso até que haja uma confirmação de recebimento da mensagem pelo processo destinatário. No modo assíncrono o processo que envia a mensagem não espera por essa confirmação do destinatário, passando a executar a instrução seguinte à chamada da função MPI que enviou a mensagem.

Tanto a comunicação síncrona como a assíncrona, podem ser com ou sem bloqueio. A comunicação assíncrona com bloqueio (*blocking*) permite que o pro-

cesso que enviou a mensagem passe à instrução seguinte assim que haja uma garantia de que os dados foram enviados ou copiados com segurança para um *buffer* do MPI. Na comunicação assíncrona sem bloqueio *non-blocking*, o processo passa a executar imediatamente a instrução seguinte àquela da chamada da função MPI que enviou a mensagem, mas fica a cargo do programador verificar se os dados foram enviados ou copiados com segurança por meio de funções MPI específicas.

Um conjunto mínimo de 6 funções MPI torna possível paralelizar boa parte dos programas existentes, desde que sejam paralelizáveis. Essas principais funções e suas respectivas funcionalidades estão descritas abaixo:

`MPI_Init()`	::	inicializa o ambiente de execução MPI;
`MPI_Finalize()`	::	finaliza o uso da biblioteca MPI;
`MPI_Comm_size()`	::	retorna o número de processos existentes;
`MPI_Comm_rank()`	::	retorna um identificador do processo (rank);
`MPI_Send()`	::	envia mensagem para rank destino;
`MPI_Recv()`	::	recebe mensagem de rank origem;

Na execução paralela de um programa MPI, delimitada a chamada nas duas primeiras funcões listadas acima, além de identificar o número de processos e o rank de cada processo, é necessário também definir um comunicador que agrupa os processos envolvidos numa dada comunicação. O comunicador `mpi_comm_world` inclui todos os processos, sendo possível definir subcomunicadores especificos para grupos de processos.

As funções `MPI_Send()` e `MPI_Recv()` são responsáveis por executar o envio e o recebimento das mensagens em comunicações ponto a ponto (bilaterais). Ambas são com bloqueio, podendo acontecer de forma síncrona ou assíncrona, conforme determinado pelo padrão MPI em tempo de execução, ependendo da existência ou não e do tamanho do *buffer* provido pelo próprio MPI para armazenamento de mensagens em trânsito. Assim, o processo que executa a função `MPI_Send()` fica bloqueado até a confirmação de recebimento do processo destinatário. Similarmente, outro processo, que executa a função `MPI_Recv()` pode ficar numa espera indefinida pelo recebimento de uma determinada mensagem. Assim, fica a cargo do programador garantir a sincronização entre processos, bem como a reciprocidade de chamadas a funções MPI de envio e recebimento

na comunicação bilateral, bem como a necessária sincronização entre eles. Obviamente, cada processo tem o seu próprio espaço de memória independente, sejam os processos executados num mesmo nó de memória compartilhada ou em nós distintos.

Abaixo segue a sintaxe da função `MPI_Send()`:

`MPI_Send (buf, count, datatype, dest, tag, comm)`, onde:

`buf`	::	argumento de entrada	::	endereço inicial do dado a ser enviado;
`count`	::	argumento de entrada	::	quantidade de elementos do dado a ser enviado;
`datatype`	::	argumento de entrada	::	tipo do dado a ser enviado;
`dest`	::	argumento de entrada	::	rank destino;
`tag`	::	argumento de entrada	::	identificador opcional da mensagem;
`comm`	::	argumento de saída	::	comunicador;

As funções `MPI_Recv()` e `MPI_Send()` possuem lista de argumentos similares, sendo ambas com bloqueio e podendo ser executadas pelos processos simultaneamente ou em instantes distintos. O MPI também oferece variações dessas funções para os outros modos de comunicação, como síncrono, sem bloqueio, (etc.) modificando apenas uma letra de prefixo da função (por exemplo: `MPI_Isend() / MPI_Irecv()`, `MPI_Ssend()`, etc.) (MPI 1.0, 1994).

As desvantagens de se usar comunicação bilateral são relativas à necessidade de sincronização e cooperação entre processos na comunicação entre êles, tornando a programação paralela mais complexa em programas de grande porte. Outra desvantagem é devida à reciprocidade necessária, isto é, para cada chamada à função `Send` num processo deve existir uma chamada correspondente `Recv` em outro processo, aumentando a complexidade do código e dificultando o entendimento do algoritmo original (sequencial). Há também a latência devida ao envio e recebimento de mensagens não ser simultâneo nos processos envolvidos na comunicação bilateral.

2.3 Padrão MPI 2.0 e a comunicação unilateral

O MPI 2.0 apresentou três novas funcionalidades em relação ao MPI 1.0: (1) criação e extinção dinâmica de processos, (2) I/O (input/output) paralelo aperfeiçoado e (3) comunicação unilateral por acesso remoto à memória. Este trabalho visa estudar mais profundamente esta última funcionalidade, não abordando as demais.

O conceito de comunicação unilateral (*One Sided Communication*) é relativo à participação direta de apenas um processo na comunicação entre processos. O processo que executa a comunicação acessa uma janela na área de memória do processo alvo utilizando RMA (*Remote Access Memory*) para efetuar uma leitura ou escrita por meio de funções MPI específicas. Assim, o processo alvo não executa nenhuma função MPI e não há necessidade de sincronização nessa comunicação. Essa janela de memoria torna-se acessível aos demais processos do comunicador por meio da chamada da função MPI `MPI_Win_create()`. A descrição da função MPI `MPI_Win_create()` é listada logo a seguir.

`MPI_Win_create (base, size, disp_unit, info, comm, win)`, onde:

`base`	::	argumento de entrada	::	endereço inicial da janela;
`size`	::	argumento de entrada	::	tamanho da janela em bytes;
`disp_unit`	::	argumento de entrada	::	tamanho unitário do tipo de dado, em bytes;
`info`	::	argumento de entrada	::	informação adicional;
`comm`	::	argumento de entrada	::	comunicador;
`win`	::	argumento de saída	::	objeto janela retornado pela função;

Essa função é executada de forma coletiva, ou seja, cada processo cria uma janela de memória própria e a torna visível aos demais processos do comunicador. Um dado processo acessa a janela de memória de outro processo por RMA utilizando as funções `MPI_Put()` e `MPI_Get()`, respectivamente, para escrever ou ler o dado desejado.

A função `MPI_Put()` transfere a quantidade de `origin_count` elementos sucessivos do tipo especificado em `origin_datatype` a partir do endereço inicial indicado por `origin_addr` do processo origem para o processo destino (`target_-`

rank). Os dados são escritos na *window* do processador destino no endereço target_addr, que é calculado assim:

```
target_addr = Window_base + target_disp * disp_unit.
```

A função MPI_Get() funciona similarmente à função MPI_Put(), mas permitindo o acesso à janela RMA do processo alvo para leitura.

A seguir, descreve-se a interface da função MPI_Put():

MPI_Put (origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win), onde:

origin_addr	::	argumento de entrada	::	endereço inicial do buffer original ;
origin_count	::	argumento de entrada	::	quantidade de elementos contidos no buffer original ;
origin_datatype	::	argumento de entrada	::	datatype de cada elemento do buffer original;
target_rank	::	argumento de entrada	::	rank do processo alvo;
target_disp	::	argumento de entrada	::	deslocamento do início da janela para o buffer de destino;
target_count	::	argumento de entrada	::	quantidade de elementos contidos no buffer destino;
target_datatype	::	argumento de entrada	::	datatype de cada elemento do buffer destino;
win	::	argumento de entrada	::	janela utilizada para comunicação;

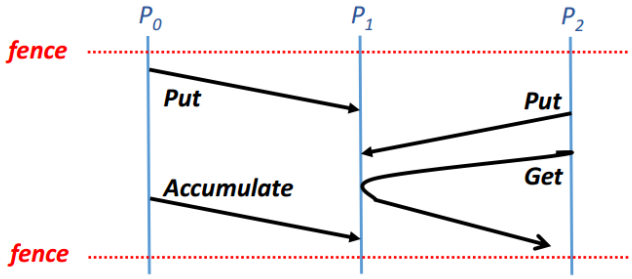
Quando um ou mais processos expõem uma parte de sua memória para os outros processos acessarem via RMA, podem ocorrer condições de corrida (*Race Conditions*), nas quais não há garantia de sincronização entre leituras e escritas de uma ou mais variáveis pelos vários processos. Essa sincronização, no escopo da comunicação unilateral por RMA é feita pela definição de "épocas" (*epochs*). Uma época é definida por um par de chamadas MPI_Win_fence() que delimita um trecho do programa em que são feitos acessos por meio de MPI_Put() e MPI_Get() à janela RMA de um ou mais processos. A Figura 2.3 ilustra a comunicação unilateral ocorrendo dentro de uma época.

Os argumentos da função `MPI_Win_fence()` são expostos abaixo.

`MPI_Win_fence (assert, win)`, onde:

`assert`	::	argumento de entrada	::	parâmetro para otimizar a sincronização;
`win`	::	argumento de entrada	::	objeto janela de memória compartilhada;

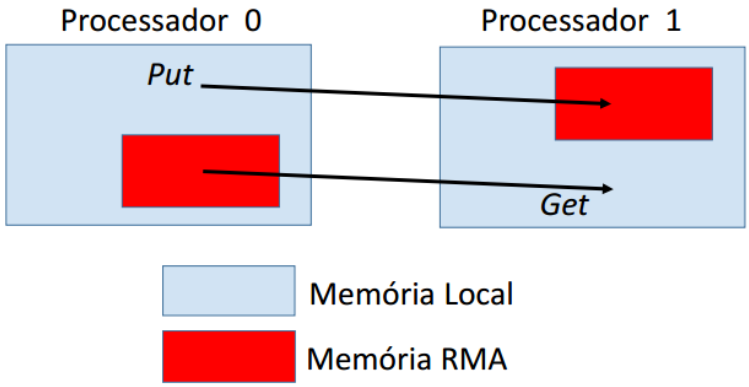
Figura 2.3 - Sincronização da comunicação RMA entre processos através da `MPI_Win_fence()`.



Fonte: Mendes e Stephany (2016).

A Figura 2.4 mostra como acontece o acesso à memória remota na comunicação unilateral usando as funções `MPI_Put()` e `MPI_Get()`. São ilustrados dois processos, o processo 0 e o processo 1, cada um com sua respectiva área de memória local representada pela área cinza. Cada processo também possui uma sub-área em vermelho representando a janela que é acessível remotamente pelos demais processos. Assim, a janela em vermelho do processo 0 pode ser vista e acessada pelo processo 1, e vice-versa.

Figura 2.4 - Esquema ilustrativo do funcionamento das funções `MPI_Put()` e `MPI_Get()`.

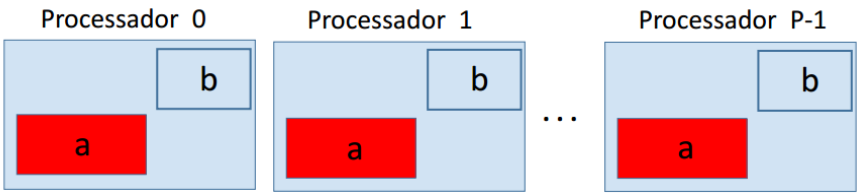


Fonte: Mendes e Stephany (2016).

A Figura 2.4 mostra o processo 0 executando um `MPI_Put()` com destino à janela de memória do processo 1. O processo 0 está escrevendo um dado que estava em sua memória local na janela de memória compartilhada do processo 1, sem a intervenção deste, ou seja, o processo 1 não participou desta escrita. O processo 1 está executando um `MPI_Get()` para ler um dado da janela de memoria do processo 0 e armazená-lo em sua memória local, sem a participação do processo 0. Os processos que participam da comunicação unilateral não precisam estar no mesmo nó, nem precisam compartilhar fisicamente a mesma memória para comunicação unilateral.

A função `MPI_Win_create()` cria a janela de memória que é acessível remotamente pelos demais processos. A Figura 2.5 mostra a criação dessas janelas de memórias. A mesma figura representa "P" processadores (de 0 a P-1), onde cada processador possui uma vetor "`b`" alocado em sua memória local (área em cinza), e um o vetor "`a`" na sua janela de acesso remoto. A execução da função `MPI_Win_create()` torna visível o vetor "`a`" de cada processo a todos os demais processos, permitindo leitura e escritas remotas através das funções `MPI_Put()` ou `MPI_Get()`.

Figura 2.5 - Esquema ilustrativo do funcionamento da função `MPI_Win_create()` no RMA.



Fonte: Mendes e Stephany (2016).

O processo 0 pode executar um `MPI_Put()` colocando, por exemplo, a variável `b[7]`, que está originalmente em sua memória local, escrevendo na janela `a[7]` de memória remota do processo 1. Mas o processo 0 não pode acessar as variáveis `b[]` de outros processos, pois estas estão em suas memórias locais (MENDES; STEPHANY, 2016).

2.4 Padrão MPI 3.0 e a comunicação unilateral de memória compartilhada

A versão MPI 3.0 apresentou novas funcionalidades em relação ao MPI 2.0, tais como funções para comunicação coletiva sem bloqueio e a comunicação unilateral de memória compartilhada *Shared Memory* (SHM), aplicável somente a processos executados dentro de um mesmo nó computacional de memória compartilhada. A comunicação unilateral por RMA introduzida pelo padrão MPI 2.0 aplica-se tanto a processos executados no mesmo nó quanto em nós distintos. A comunicação unilateral MPI *Shared Memory* visa otimizar a comunicação entre processos executados num mesmo nó de memória compartilhada por meio da criação de uma janela comum visível a todos esses processos, e que permite leituras e escritas diretas, isto é, sem uso de funções MPI.

A versão MPI 3.0 oferece funções específicas para identificar o *rank* global de cada processo executado num mesmo nó e mapeá-lo para um *rank* local. A função `MPI_Comm_split_type()` possibilita dividir o comunicador global em subcomunicadores disjuntos, que são específicos de cada nó computacional de memória compartilhada quando o argumento `split_type` é do tipo `MPI_COMM_TYPE_SHARED`. Os demais argumentos dessa função são listados em seguida.

`MPI_Comm_split_type (comm, split_type, key, info, newcomm)`, onde:

`comm`	::	argumento de entrada	::	grupo comunicador global;
`split_type`	::	argumento de entrada	::	`MPI_COMM_TYPE_SHARED`;
`key`	::	argumento de entrada	::	chave para manter a ordem dos *ranks*;
`info`	::	argumento de entrada	::	Info;
`newcomm`	::	argumento de saída	::	novo comunicador;

Outra função importante é a `MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2)`, que mapeia uma lista de *ranks* do comunicador global (`group1`) para a lista de *ranks* de um sub-comunicador (`group2`). Assim é possível obter os *ranks* do sub-comunicador específico de cada nó de memória compartilhada. Os argumentos da função `MPI_Group_translate_ranks()` são listados abaixo.

`MPI_Group_translate_ranks (group1, n, ranks1, group2, ranks2)`, onde:

`group1`	::	argumento de entrada	::	comunicador 1;
`n`	::	argumento de entrada	::	número de `ranks` correspondentes dos comunicadores 1 e 2;
`ranks1`	::	argumento de entrada	::	vetor com os `ranks` do comunicador 1;
`group2`	::	argumento de entrada	::	grupo comunicador 2 ;
`ranks2`	::	argumento de saída	::	vetor com os `ranks` do comunicador 2 (retorna `MPI_UNDEFINED` quando não há correspondência);

Uma vez identificados quais os processos que compartilham do mesmo nó, pode-se alocar uma janela de memória compartilhada e torná-la visível a esses processos. A função `MPI_Win_allocate_shared()` cria e aloca uma área da memória para a janela de memória compartilhada, a qual é visível para todos os processos do mesmo nó. Os parâmetros da função `MPI_Win_allocate_shared()` são listados logo a seguir.

`MPI_Win_allocate_shared (size, disp_unit, info, comm, baseptr, win)`, onde:

`size`	::	argumento de entrada	::	tamanho da janela de memória a ser alocada;
`disp_unit`	::	argumento de entrada	::	tamanho unitário do tipo do dado, para o cálculo do deslocamento na memória;
`info`	::	argumento de entrada	::	Informação opcional;
`comm`	::	argumento de entrada	::	comunicador local;
`baseptr`	::	argumento de saída	::	endereço de memória do início do segmento de memória compartilhada do processo;
`win`	::	argumento de saída	::	endereço de memória da janela criada;

Uma outra função fundamental na comunicação unilateral SHM é a `MPI_Win_shared_query()`, que retorna o endereço da parte da janela compartilhada de outro processo do sub-comunicador local (ao nó computacional) para o processo que a executa. O endereço retornado serve para o processo acessar (ler ou escrever) as posições de memória da janela com dados que os demais processos inicializaram ou escreveram. Os parâmetros da função `MPI_Win_shared_query()` são listados logo abaixo.

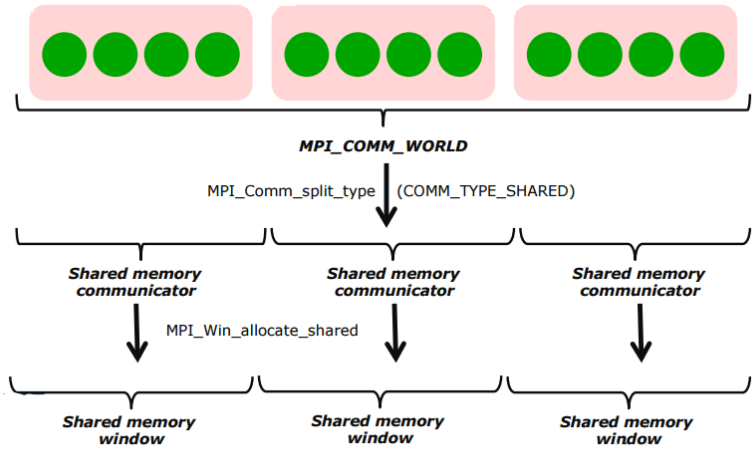
`MPI_Win_shared_query (win, rank, size, disp_unit, baseptr)`, onde:

`win`	::	argumento de entrada	::	endereço de memória da janela compartilhada;
`rank`	::	argumento de entrada	::	*rank* do processo cujo segmento da janela compartilhada se deseja acessar;
`size`	::	argumento de saída	::	tamanho da janela de memória;
`disp_unit`	::	argumento de saída	::	tamanho unitário do tipo do dado;
`baseptr`	::	argumento de saída	::	endereço para acesso à parte da janela compartilhada do processo definido em `rank`;

O acesso concorrente dos processos locais de um nó à janela de memória com-

partilhada pode também ocasionar *Race Conditions*, sendo necessário definir uma "época" de forma a sincronizar esses acessos, à semelhança de como é feito na comunicação unilateral RMA.

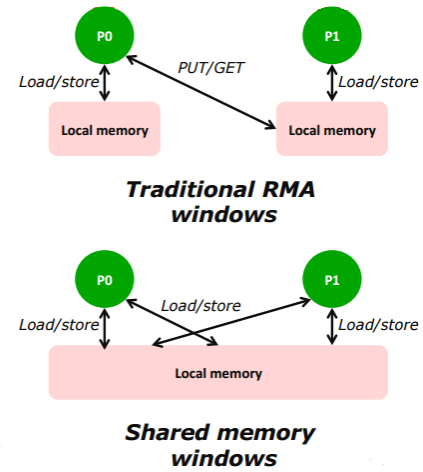
Figura 2.6 - Divisão do comunicador global em subgrupos de comunicadores específicos de cada nó de de memória compartilhada.



Fonte: Gropp (2016).

A Figura 2.6 mostra a divisão do comunicador global em subgrupos de comunicadores disjuntos, e a criação da janela de memória compartilhada em cada nó. Os círculos em verde representam os processos circulados pela área rosa que representam cada nó. Cada nó contém quatro processos. Todos os processos disponíveis fazem parte do grupo comunicador global (`MPI_COMM_WORLD`). A mesma figura mostra também a alocação da janela compartilhada acessível a todos os processos do mesmo nó.

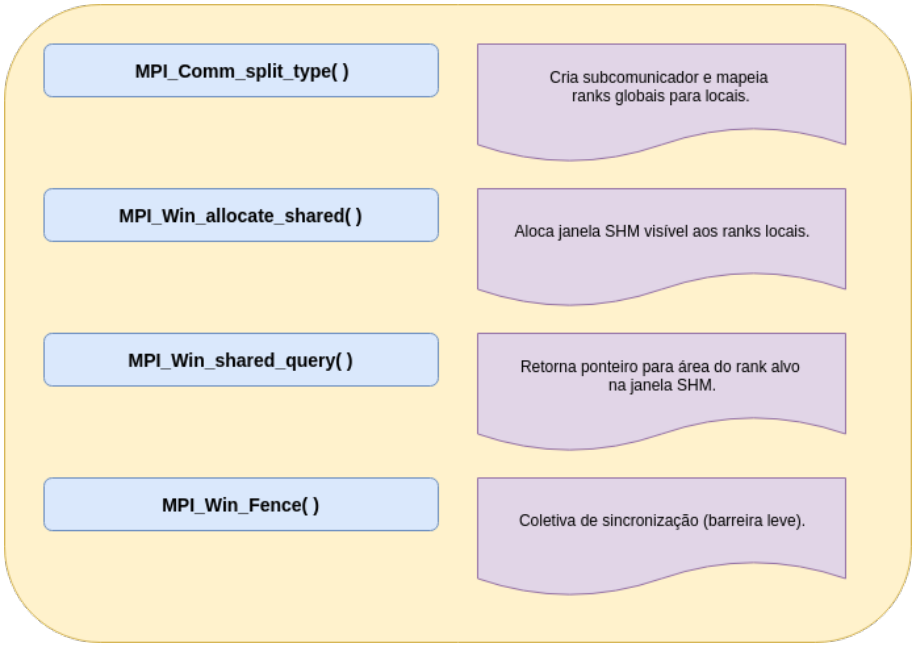
Figura 2.7 - Comparação da comunicação MPI unilateral com janela RMA e com janela SHM.



Fonte: Gropp (2016).

A Figura 2.7 ilustra a diferença entre a comunicação unilateral por RMA e unilateral de SHM. Na comunicação RMA, o dado somente pode ser acessado (escrita e leitura) através das funções `PUT/GET`. Na comunicação *Shared Memory* os processos executam leituras e escritas diretamente na janela de memória compartilhada.

Figura 2.8 - Resumo das funções MPI utilizadas na comunicação unilateral SHM.



A Figura 2.8 ilustra as funções MPI necessárias à comunicação MPI SHM, que já foram detalhadas nesta seção. São elas: (1) `MPI_Comm_split_type()` que cria um sub-comunicador local e mapeia *ranks* globais para locais; (2) `MPI_Win_allocate_shared()` que cria e aloca a janela SHM tornado-a visível aos processos do sub-comunicador local; (3) `MPI_Win_shared_query()` que retorna um ponteiro relativo ao endereço de memória da parte da janela SHM de outro processo local, e (4) `MPI_Win_fence()` que executa a sincronização dos processos na comunicação unilateral SHM.

Segue abaixo um trecho de código do programa exemplo no qual é utilizada a comunicação unilateral SHM:

```
1  program exemplo
2
3    USE, INTRINSIC :: ISO_C_BINDING
4    USE            :: MPI
5
6    implicit none
7    (...)
8    call MPI_Init(ierror)
9    call MPI_Comm_rank(MPI_COMM_WORLD, r, ierror)
10   call MPI_Comm_size(MPI_COMM_WORLD, p, ierror)
11   (...)
```

```
12    !divide o comm_world em comm_sh:
13    call MPI_Comm_split_type(MPI_COMM_WORLD,       &
14                             MPI_COMM_TYPE_SHARED, &
15                             0,                    &
16                             MPI_INFO_NULL,        &
17                             shmcomm,              &
18                             ierror)
19    call MPI_Comm_size(shmcomm, sp, ierror)
20    call MPI_Comm_rank(shmcomm, sr, ierror)
21    (...)
22    call MPI_Win_allocate_shared(size,           &
23                                 disp,           &
24                                 info_noncontig, &
25                                 shmcomm,        &
26                                 mem1,           &
27                                 win1,           &
28                                 ierror)
29    call c_f_pointer(mem1, anew, memshape)
30    (...)
31    !obtendo os enderecos das SHM vizinhas ao NORTE do meu rank:
32    if ( north /= MPI_PROC_NULL ) then
33       call MPI_Win_shared_query(win1,      &
34                                 north,     &
35                                 size,      &
36                                 disp_unit, &
37                                 memnorth,  &
38                                 ierror)
39       call c_f_pointer(memnorth, northptr, northptrshape)
40    endif
41    (...)
42    !laco principal das iteracoes:
43    do iters=1, niters
44       call MPI_Win_fence(0, win2, ierror) ! INICIO de uma epoca-----------|
45       if ( north /= MPI_PROC_NULL ) then
46          aold(1, 2:bx+1) = northptr(bx+1, 2:bx+1)
47       endif
48       (...)
49       !update grid points
50       do j=2, by+1
51          do i=2, bx+1
52             anew(i, j) = aold(i, j)/2.0 + &
53                          (aold(i-1, j)  + &
54                          aold(i+1, j)   + &
55                          aold(i, j-1)   + &
56                          aold(i, j+1))/4.0/2.0
57          enddo
58       enddo
59       (...)
60       aold=anew
61    enddo
62    (...)
63    call MPI_Finalize(ierror)
64 end
```

3 O MODELO METEOROLÓGICO BRAMS

O BRAMS (*Brazilian developments on the Regional Atmospheric Modeling System*) é um modelo meteorológico de previsão numérica regional de tempo. Foi desenvolvido baseado no RAMS (*Regional Atmospheric Modeling System* (FREITAS et al., 2016)) que foi originalmente criado por cientistas da Universidade do Estado do Colorado, nos Estados Unidos da América. Aqui no Brasil, o RAMS foi adaptado para tratar os problemas ambientais e sistemas atmosféricos locais da América do Sul, dando origem ao BRAMS.

A primeira versão do BRAMS foi desenvolvida pelo CPTEC/INPE em conjunto com ATMET/USA (*ATmospheric, Meteorological, and Environmental Technologies*), IME/USP (Instituto Matemática e Estatística da Universidade de São Paulo) e IAG/USP (Instituto de Astronomia, Geofísica e Ciências Atmosféricas da Universidade de São Paulo). Atualmente o BRAMS é mantido, atualizado e distribuído pelo CPTEC/INPE sendo também é utilizado nas previsão de tempo e ambiental, bem como em suas atividades de pesquisa (FREITAS et al., 2016).

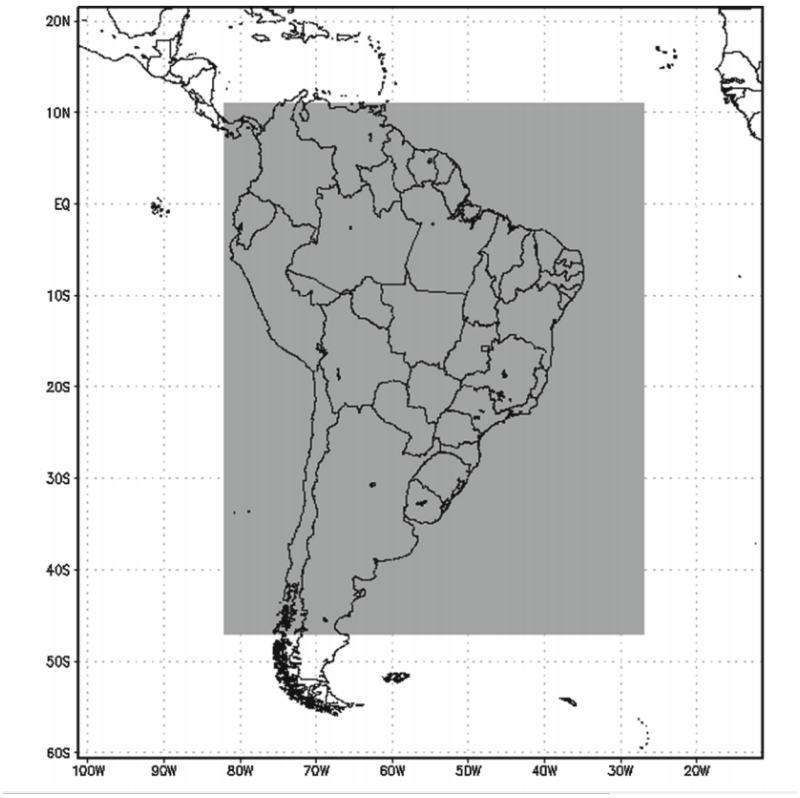
O CPTEC/INPE desenvolveu e utiliza o modelo ambiental CCATT-BRAMS (*Coupled-Chemical, Aerosol and Tracer Transport*) que acopla um modelo de transporte de gases traço ao modelo regional BRAMS. Esse modelo ambiental permite o monitorar operacionalmente o transporte atmosférico de emissões antropogênicas e de queimadas sobre os continentes da América do Sul e África e também sobre o Oceano Atlântico Sul.

No modelo BRAMS, cada variável é representada por uma matriz tridimensional. Os pontos da matriz representam os valores médios do domínio físico discretizado em cada célula da grade computacional. Essa representação é denominada de "campo". Para um domínio de tamanho fixo, quanto maior a quantidade de pontos menor será o tamanho da célula e melhor representada será a atmosfera, mas o custo computacional será maior. O tamanho da célula nasce da distância entre os pontos da malha computacional, definindo a resolução espacial do modelo.

No CPTEC/INPE, o BRAMS era utilizado operacionalmente até maio de 2018, com resolução espacial de 5km e sobre o domínio da América do Sul para previsão de tempo (PANETTA, 2012). Sua execução utiliza 9.600 processadores no Supercomputador Cray-XE6 tupã, demandando aproximadamente 6 horas de execução para prever 168 horas à frente (7 dias), com um passo de tempo de

12 segundos nas equações, saídas prognósticas geradas a cada 6 horas e com uma grade tridimensional com pouco mais 90 milhões de pontos (1360x1480x45 - x, y, z). A Figura 3.1 mostra seu domínio espacial horizontal.

Figura 3.1 - Cobertura do domínio espacial do BRAMS.



O modelo numérico BRAMS foi escrito em sua maior parte na linguagem Fortran-2003 e com alguns módulos escritos em linguagem C, com aproximadamente 350.000 linhas de código. O BRAMS foi paralelizado com a biblioteca de comunicação MPI, sendo a decomposição de domínio feita no plano horizontal apenas, ou seja, cada processo MPI abrange um subdomínio composto de pontos da grade horizontal, incluindo para cada ponto todos seus níveis verticais.

O BRAMS é executado em três etapas:

1. Primeira etapa: `MAKESFC`
   Esta etapa converte dados da superfície terrestre (topografia, cobertura do solo, temperatura de superfície do mar, entre outros) no formato necessário para as próximas etapas, considerando a grade a ser utilizada.

2. Segunda etapa: `MAKEVFILE`
   Nesta etapa, o BRAMS converte os arquivos de condições iniciais e de contorno (oriundos de um modelo global) na própria resolução espacial, deixando-os prontos para os cálculos.

3. Terceira etapa: `INITIAL`
   É nesta etapa que o BRAMS executa o cálculo das equações atmosféricas simulando o estado da atmosfera para os próximos instantes. É nesta etapa também que o BRAMS produz os arquivos denominados de análises. As análises são arquivos contendo valores aproximados dos campos físicos da atmosfera num determinado instante de tempo. Na versão anterior do BRAMS existia uma quarta etapa que convertia as análises para o formato binário IEEE-754 (IEEE-754, 2017). Isso tornava possível a visualização dos dados com um programa de visualização gráfica, como por exemplo, o GrADS (GRADS, 2017) apenas ao final de sua execução. Agora, esse recurso acontece durante o tempo de execução, simultaneamente com o cálculo das integrações do modelo. Este recurso é especialmente importante para os centros operacionais pois permite gerar produtos operacionais automáticos durante sua execução.

A sequência abaixo mostra as 4 fases que compõem o BRAMS na etapa `INITIAL`:

1. Inicialização;

2. Leitura dos dados de entrada e leitura periódica de dados de condição de contorno;

3. Processamento principal (rotina *TimeStep*);

4. Saída (escrita periódica).

Na Inicialização (fase 1), é criada toda estrutura de dados utilizada no BRAMS, bem como a alocação dinâmica de memória para os campos. É também nesta fase que a leitura das condições iniciais da atmosfera é executada. Um único processo principal lê cada campo das condições iniciais e os envia para todos os outros processos via *broadcast*. Cada processo recebe o campo inicial atmosférico integral e extrai apenas o pedaço correspondente ao seu subdomínio.

A fase 2 é responsável por atualizar o estado da atmosfera nas bordas do domínio do BRAMS ao longo do tempo. Assim como em todos os modelos regionais, o BRAMS recebe periodicamente dados de um modelo global atmosférico como condição de contorno em volta de seu domínio principal. Dessa maneira, o BRAMS mantém o estado da atmosfera na borda do seu domínio sempre atualizado. Um único processo lê os campos atmosféricos de contorno, distribui entre os processos responsáveis por computar as bordas. Cada um desses processos, por sua vez, recorta os campos recebidos e armazena apenas o trecho do seu subdomínio.

É na fase 3 (rotina *TimeStep*) que o cálculo numérico das equações diferenciais parciais atmosféricas é realizado, representando todo o transporte de fluidos da atmosfera (dinâmica) e também as forçantes das equações (processos físicos) como a radiação, a convecção, etc. Essa fase é a que mais demanda capacidade de processamento e operações aritméticas de ponto flutuante.

Finalmente a quarta fase do BRAMS escreve os resultados de sua simulação nos intervalos de tempo solicitados. A escrita de dados em disco é feita por um único processo, que recebe os dados dos subdomínios dos outros processos, compõe o domínio completo e escreve-o em disco, um campo por vez.

As quatro fases da etapa INITIAL do BRAMS descritas acima podem ser vistas no algoritmo mostrado na tabela 3.1. A tabela apresenta o algoritmo escrito numa pseudo-linguagem em duas versões: uma do ponto de vista do processo responsável por I/O (coluna esquerda da tabela), e outra para os processos que recebem os campos e executam a computação (coluna da direita).

Tabela 3.1 - Algoritmo geral em alto nível do BRAMS visto pelo processo responsável pelo I/O (coluna da esquerda) e pelos outros processos (coluna da direita).

I/O Process view	Ordinary process view
1. initialization (send to slaves);	1. initialization (receive from master);
2. for (until maximum time)	2. for (until maximum time)
3. if (read boundary conditions)	3. if (read boundary conditions)
4. read boundary conditions;	4.
5. send all fields to others processes;	5. receive all fields from I/O process;
6. endif	6. endif
7. perform timestep;	7. perform timestep;
8. evaluate CFL;	8. evaluate CFL;
9. receive CFL results from all processes;	9. send CFL results to I/O process;
10. if (output) then	10. if (output) then
11. receive sub-domain from all processes;	11. send my sub-domain to I/O processes;
12. mount application domain;	12.
13. output data;	13.
14. endif	14. endif
15. endfor	15. endfor

É importante ressaltar que na tabela 3.1, o algoritmo da esquerda diferencia do algoritmo executado pelos outros processos apenas no I/O e no cálculo do CFL. Apenas um processo é responsável por fazer a leitura dos dados iniciais e das condições de contorno e envia-los aos demais processos (tarefas executadas nas linhas 4 e 5 do algoritmo da esquerda). Do ponto de vista dos outros processos (algoritmo da direita) apenas a tarefa de receber os campos do processo I/O é executada (linha 5 do algoritmo da direita).

Em seguida, os processos entram no laço principal e integram as equações ao longo do tempo (rotina *TimeStep*). Todos os processos executam o processamento da rotina *TimeStep* (na linha 7). É na rotina *TimeStep* que é calculado o estado da atmosfera através das equações diferenciais. As equações permitem calcular o estado da atmosfera em qualquer instante de tempo. Em seguida é feito o cálculo da condição de estabilidade CFL (Courant–Friedrichs–Lewy).

Os critérios de estabilidade de integração numérica das equações diferenciais parciais CFL foram criados por Courant, Friedrichs e Lewy (CFL, 2017). Se esta condição de estabilidade for excedida, em qualquer célula da grade, o passo de tempo de integração precisa ser reduzido. No BRAMS, cada processo calcula o valor máximo da CFL em seu próprio domínio e envia-o para o processo respon-

sável por computar o valor máximo global.

A linha 10 verifica se naquele instante de tempo do cálculo do *timestep*, o programa deve escrever resultados em disco. Caso afirmativo, todos os campos solicitados naquele exato momento são escritos em disco. Nesse caso, todos os processos enviam os campos de seus respectivos subdomínios para o processo responsável por I/O. Este processo recebe os subdomínios remontando o domínio total. E depois escreve cada campo total em disco (linhas 11, 12 e 13 de ambos os algoritmos).

Toda a configuração do BRAMS é feita dinamicamente através de um arquivo de entrada clássico da linguagem Fortran do tipo *namelist*. Dessa forma, o BRAMS dispensa ser compilado sempre que for preciso alterar um parâmetro ou ajustar qualquer uma das opções de configuração do modelo pois todas as variáveis de configuração são lidas em tempo de execução (ALMEIDA; BAUER, 2012).

O arquivo de entrada *namelist* chama-se RAMSIN. Neste arquivo reúne-se todos os parâmetros que traçam o perfil das previsões desejadas pelo usuário. Dentre tantas variáveis fundamentais para definir o funcionamento do BRAMS, as principais são:

- `NNXP`, `NNYP` e `NNZP`: definem o número de células discretas nas direções `x`, `y` e `z` da grade de execução;
- `RUNTYPE`: define qual etapa do BRAMS será executada;
- `TIMEUNIT` e `TIMMAX`: juntas definem a unidade de tempo (`TIMEUNIT`) e por quanto tempo será a duração desta previsão;
- `DELTAX` e `DELTAY`: definem a variação espacial representada pela grade de pontos, trabalham em conjunto com as variáveis `NNXP` e `NNYP`;
- `DTLONG`: define o passo do tempo em segundos das equações do modelo.

3.1 Divisão de domínio do BRAMS

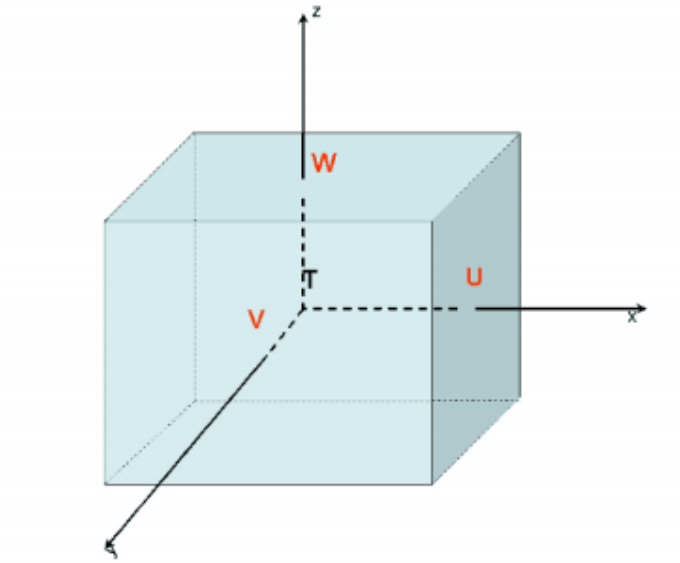
O BRAMS é executado com sua grade total divida em subdomínios (iguais ou não) entre os processos MPI, sendo que cada subdomínio é atribuído a um processo distinto. As equações diferenciais parciais do modelo são discretizadas

pelo método das Diferenças Finitas e resolvidas pelo cálculo de estênceis de 5 pontos aplicados em seu próprio subdomínio.

Em um RAMSIN com `NNXP = nx` e `NNYP = ny`, por exemplo, cada campo meteorológico do BRAMS é representado por um *array* Fortran com índices (`1:nx`, `1:ny`), desprezando a coordenada vertical. Dependendo do campo que está sendo representado pelo *array*, alguns elementos desse array representam valores no interior do domínio, outros representam valores na fronteira do domínio (ou seja, condição de fronteira) e, em alguns campos, há células sem utilidade.

A razão dessa variação é a grade deslocada *staggered grid* utilizada pelo BRAMS. Nessa grade, campos escalares são colocados no centro de cada célula, enquanto campos vetoriais são colocados na face de maior coordenada da célula. Como o valor do campo em cada célula é representada por um elemento do *array*, os valores de cada campo podem estar no interior de uma célula ou na borda dessa célula. Por exemplo, na Figura 3.2 as velocidades U, V e W estão colocadas nas faces de maior coordenada de x, y e z, respectivamente, enquanto o campo de temperatura T é colocado no ponto central da célula.

Figura 3.2 - Representação gráfica de um subdomínio físico do BRAMS: um paralelepípedo, e como são armazenadas computacionalmente.

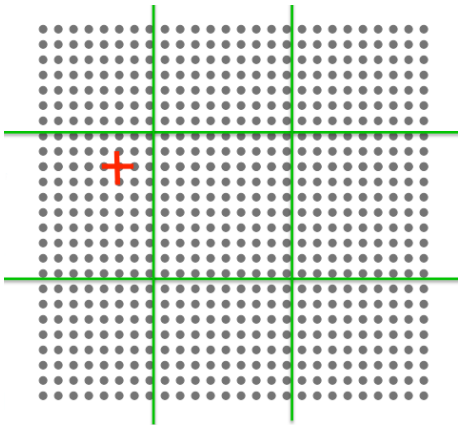


O domínio físico é um paralelepípedo que é discretizado por células de tamanho fixo. Para representar as condições de fronteira de um campo vetorial, é necessário utilizar todas as células externas do paralelepípedo. Logo, o domínio a ser dividido entre os processos MPI é composto pelos índices (`2:nx-1, 2:ny-1`).

Este modelo trabalha com colunas na vertical inteiras, ou seja, cada ponto da grade do sub-domínio de cada processador possui `NNZP` pontos em níveis verticais. Então quando se trabalha com campos 3D, cada processador precisa considerar não apenas seu subdomínio x-y, mas também suas devidas colunas em z.

A Figura 3.3 ilustra um exemplo do domínio horizontal de um campo físico do BRAMS discretizado. A cruz vermelha mostra os pontos envolvidos no cálculo do método estêncil de 5 pontos, que usa os vizinhos para calcular o novo ponto central. Esse método é muito utilizado para descrever resoluções discretizadas pelo método de diferença finitas. A cada iteração do laço principal (rotina *TimeStep*) cada ponto de grade da malha discretizada é atualizado pela combinação linear de seu próprio valor e dos valores dos 4 pontos vizinhos através de um estêncil de 5 pontos.

Figura 3.3 - Divisão da malha computacional entre os processos.



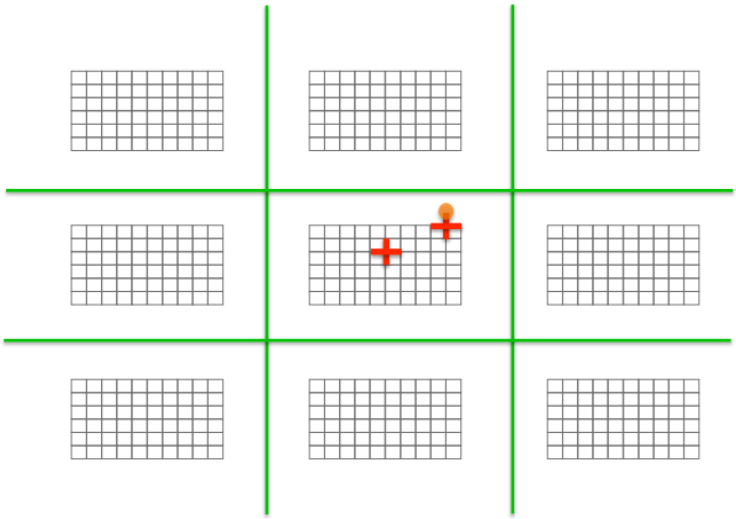
Fonte: Balaji et al. (2014)

Na atualização dos pontos nas bordas de cada subdomínio, cada processo pre-

cisa dos valores das fileiras de pontos vizinhos, que foram atualizadas pelos processos vizinhos, este é um problema clássico de atualização de bordas necessitando, dessa forma, trocar informações das bordas entre os processos vizinhos.

As linhas em verde definem os limites da divisão de domínio da malha entre os processadores, dessa forma, cada processo deverá receber um subdomínio. A Figura 3.4 mostra um exemplo do clássico problema de atualização das bordas quando se usa estêncil de 5 pontos. Cada processo precisa conhecer o ponto da borda de seu vizinho para calcular os valores centrais de sua borda.

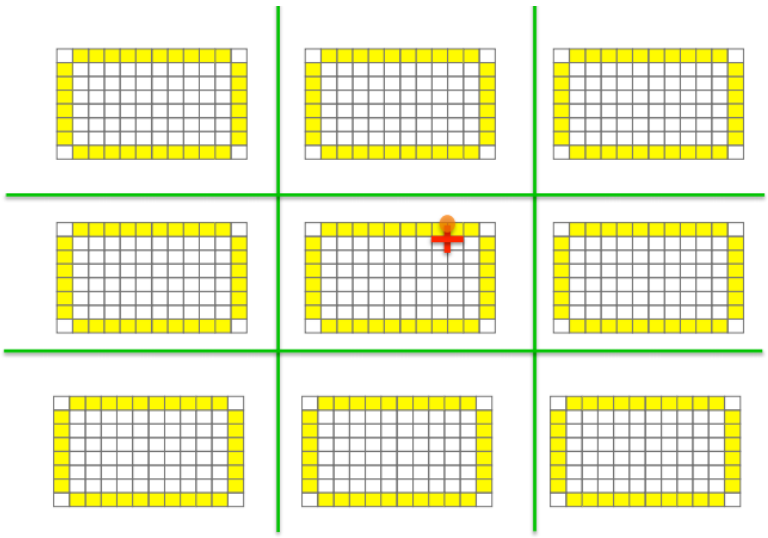
Figura 3.4 - Malha dividida em subdomínios. Cada processo precisa do ponto lateral de seu vizinho para calcular o seu ponto central nas bordas.



Fonte: Balaji et al. (2014)

Para resolver esse problema, é necessário transferir para cada processador uma fileira de dados a mais, além do seu subdomínio. A essa área de dados sobressalente transferida é dado o nome de *ghost zone* ou *halo zone*. A Figura 3.5 apresenta o esquema da malha dividida entre os processadores ilustrando o problema do *ghost zone*.

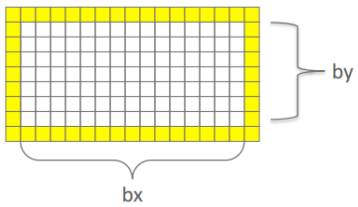
Figura 3.5 - Malha dividida em subdomínios. Cada subdomínio com sua *ghost zone*, o problema das bordas.



Fonte: Balaji et al. (2014)

O tamanho de cada subdomínio (x_{local}, y_{local}) ou (bx, by) deverá contemplar duas fileiras de pontos a mais em sua dimensão x, e na dimensão y (uma fileira de cada lado do subdomínio). Assim como apresentado na Figura 3.6, que mostra o tamanho exato de cada subdomínio dado por $(bx + 2, by + 2)$.

Figura 3.6 - Célula de subdomínio com as dimensões definidas.

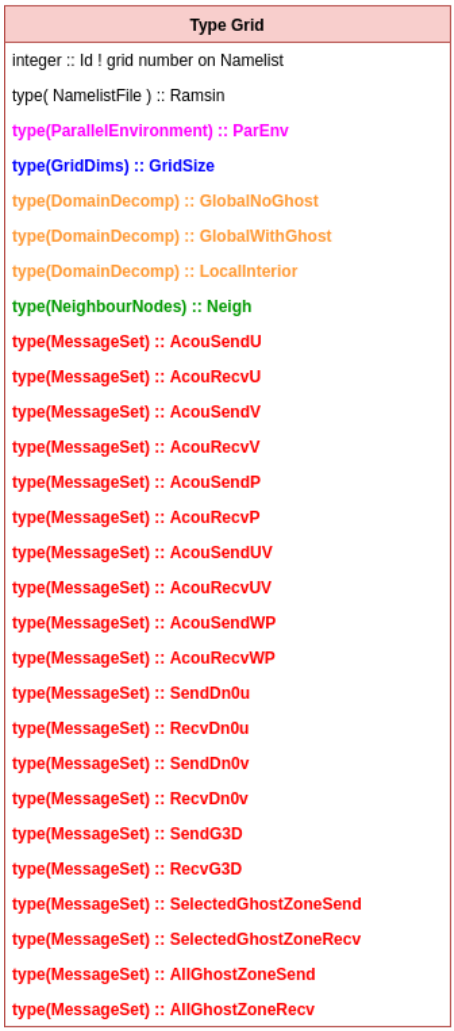


Fonte: Balaji et al. (2014)

3.2 Estrutura de dados do BRAMS

A Figura 3.7 mostra a estrutura de dados do tipo `Grid`. Dentro do BRAMS, esta estrutura representa um subdomínio de cada processo, reunindo nela outras sub-estruturas onde cada uma define um conjunto de parâmetros necessários para várias tarefas do modelo, especialmente na comunicação paralela.

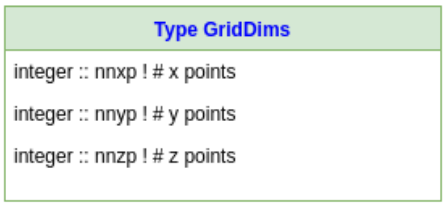
Figura 3.7 - Estrutura de dados do tipo `Grid`.



A estrutura de dados que armazena todas as informações sobre a dimensão da grade (`x`, `y` e `z`) é a `GridDims`, situada dentro da estrutura principal `Grid` com o

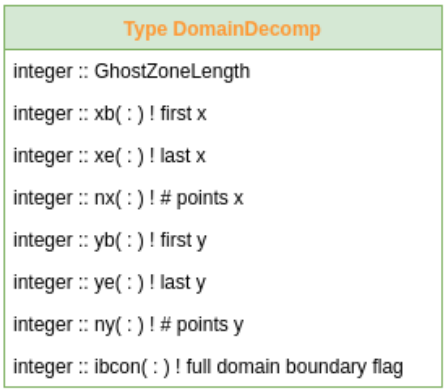
nome de `GridSize`. Esta estrutura é apresentada na Figura 3.8.

Figura 3.8 - Estrutura de dados do tipo `GridDims`.



A próxima sub-estrutura da estrutura `Grid` é a `DomainDecomp`, criada em três variáveis com os nomes de `GlobalNoGhost`, `GlobalWithGhost` e `LocalInterior`. Esta sub-estrutura é responsável por armazenar todos os índices do subdomínio em relação ao domínio principal: sem a *ghost zone* (`GlobalNoGhost`); com a *ghost zone* (`GlobalWithGhost`) e apenas os índices locais (`LocalInterior`). Assim como mostra a Figura 3.9.

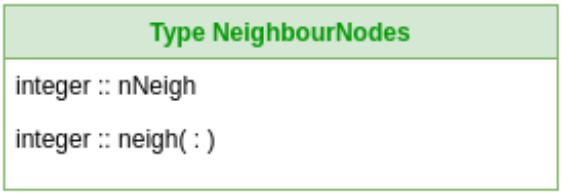
Figura 3.9 - Estrutura de dados do tipo `DomainDecomp`.



A sub-estrutura mostrada na Figura 3.10 é a `Neigh` do tipo `NeighbourNodes`, que armazena as informações dos vizinhos que cercam cada processo: `nNeigh` define

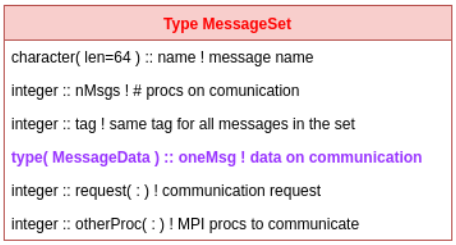
a quantidade de nós vizinhos, e o vetor `neigh(:)` que armazena todos os nós vizinhos.

Figura 3.10 - Estrutura de dados do tipo `NeighbourNodes`.



A sub-estrutura do tipo `MessageSet` talvez seja uma das mais importantes estruturas do ponto de vista da comunicação paralela do modelo BRAMS. Ela contém todas as informações necessárias para se definir uma mensagem a ser enviada: `name` que armazena o nome da mensagem; `nMsgs` informa a quantidade de mensagens a ser enviada ou recebida; `tag` contém a *tag* da mensagem MPI; `oneMsg` que é uma estrutura do tipo `MessageData` (apresentada logo a seguir) encarregada de armazenar o *buffer* da mensagem a ser enviada ou recebida; `request(:)` que armazena os *requests* das mensagens MPI e `otherProc(:)` que é o vetor onde são armazenados o número dos processos vizinhos. Esta estrutura pode ser vista com detalhes na Figura 3.11.

Figura 3.11 - Estrutura de dados do tipo `MessageSet`.



A estrutura do tipo `MessageData`, ilustrada na Figura 3.12, é uma sub-estrutura

da `MessageSet`. Ela armazena as informações sobre o *buffer* a ser enviado ou recebido no MPI: as *ghost zone* dos campos a serem enviadas ou recebidas são empacotadas no vetor `buf(:)`, com o tamanho definido em `bufSize`. A sub-estrutura `fieldList` do tipo `FieldSectionList` armazena uma lista dinâmicamente encadeada contendo o sequência dos campos físicos cujas *ghost zone* serão empacotadas no `buf(:)`. As estruturas que compõem esta lista podem ser observadas na Figura 3.13.

Figura 3.12 - Estrutura de dados do tipo `MessageData`.

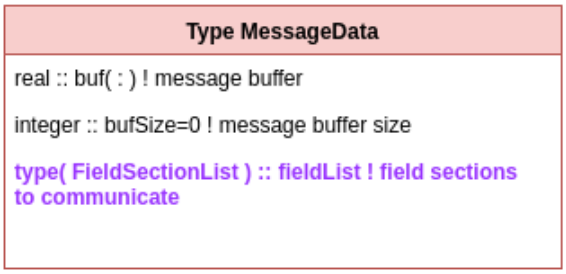
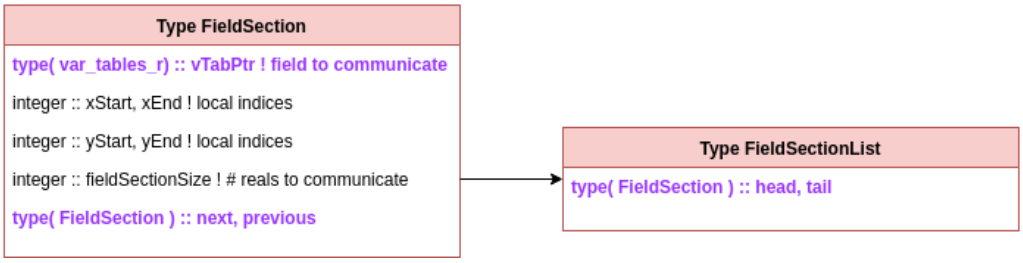


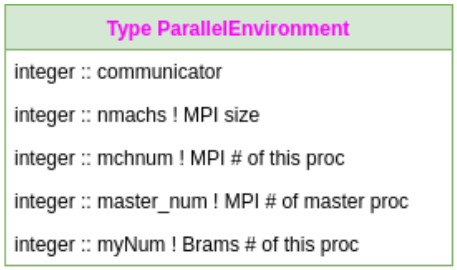
Figura 3.13 - Conjunto de estrutura de dados dos tipos `FieldSectinList` e `FieldSection`.



A Figura 3.14 ilustra a estrutura de dados do tipo `ParallelEnvironment` situada como uma sub-estrutura da `Grid` com o nome de `ParEnv`. Esta estrutura armazena todas as informações pertinentes ao paralelismo MPI: o comunicador em vigência (`communicator`), a quantidade de processos dentro deste comunicador

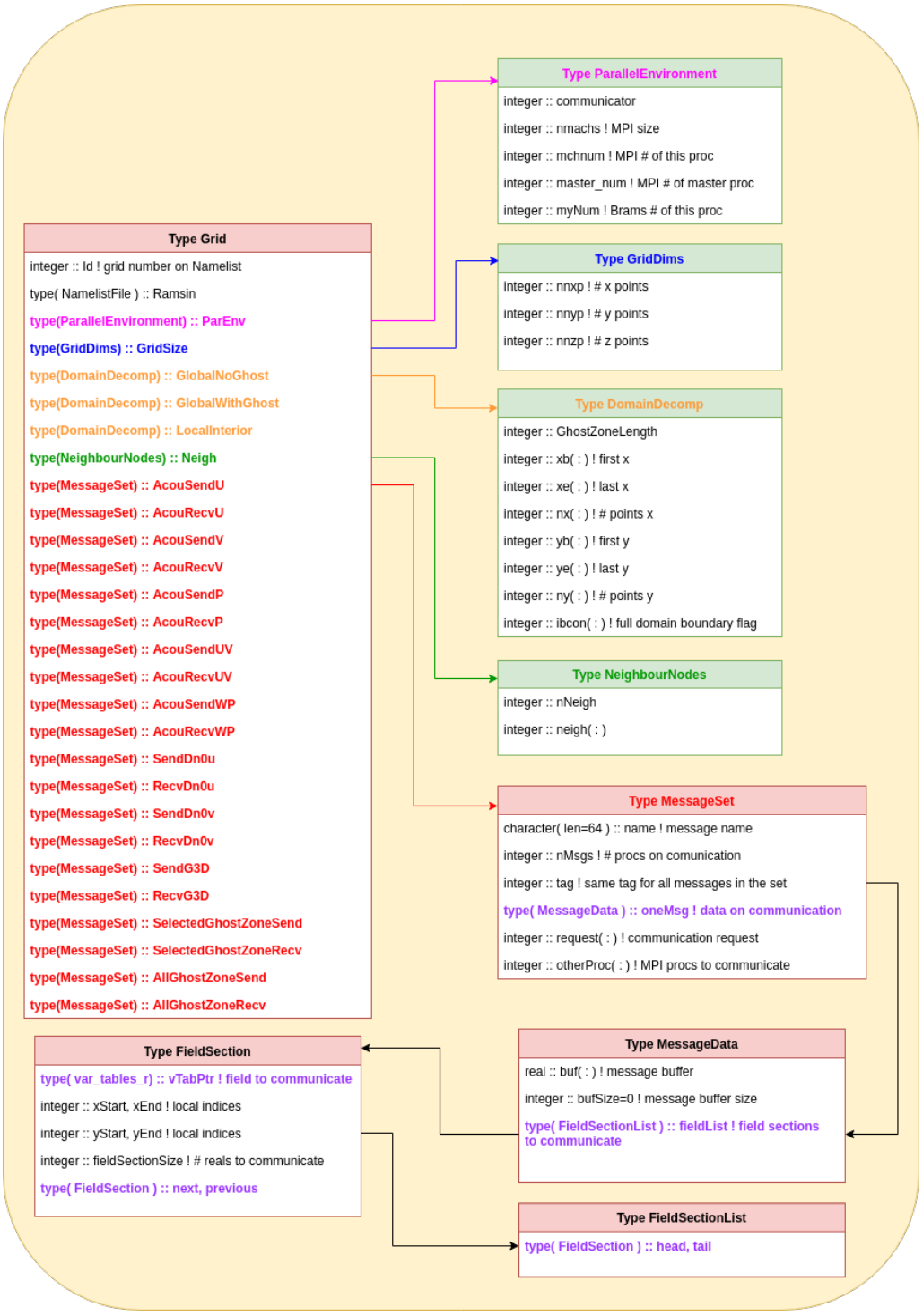
(nmachs), o número do "meu" processo (myNum), e o número do processo mestre (master_num) muito utilizado nas versões anteriores do BRAMS, quando ainda se utilizava o modelo mestre/escravo, extinto nesta última versão 5.0.

Figura 3.14 - Estrutura de dados do tipo ParallelEnvironment.



A Figura 3.15 mostra, de uma maneira geral, a organização de todas as estruturas de dados explicadas anteriormente juntas num único diagrama, mostrando onde cada estrutura está localizada e suas respectivas hierarquias.

Figura 3.15 - Diagrama de todas as estrutura de dados.

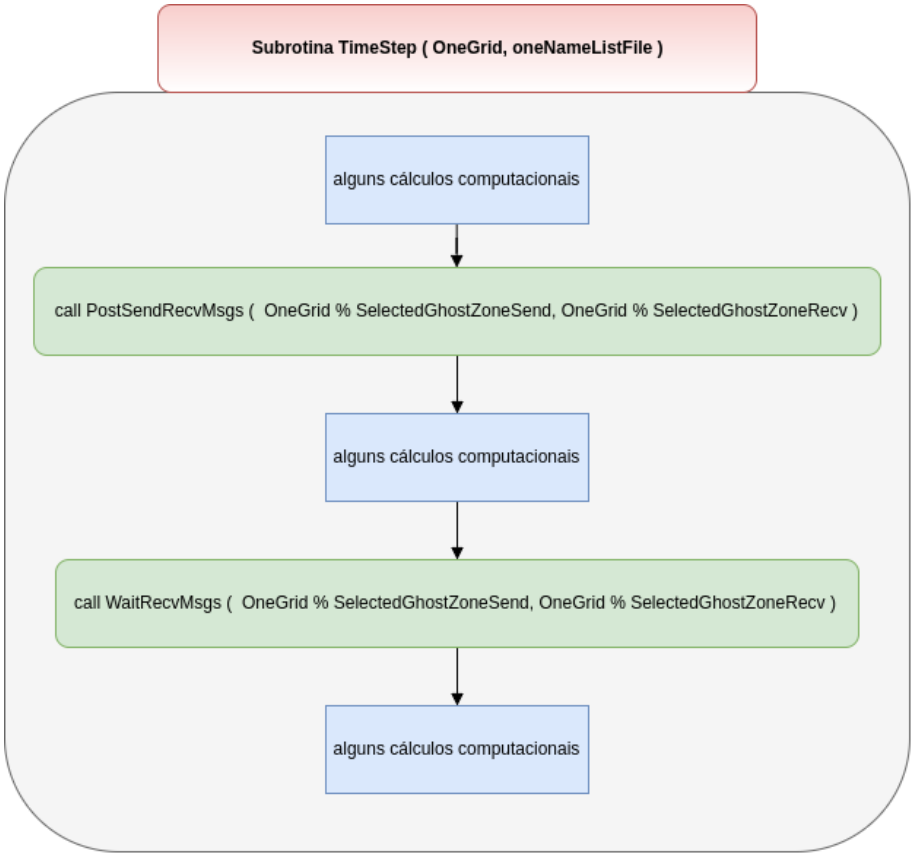


3.3 Estrutura das funções e sub-rotinas do BRAMS

A Figura 3.16 mostra um fluxograma resumido da sub-rotina `TimeStep()` do BRAMS. Esta sub-rotina executa, dentre muitas tarefas e cálculos computacionais, chamadas de sub-rotinas específicas para comunicação paralela das *Ghost Zones* entre os processos vizinhos.

A sub-rotina `PostSendRecvMsgs()` é responsável por postar as mensagens paralelas das *Gohst Zones* de envio e de recebimento. E a sub-rotina `WaitRecvMsgs()` realiza os `MPI_Wait()` e em seguida desempacota os *buffers* recebidos. As duas sub-rotinas recebem como argumento as estruturas `OneGrid%SelectedGhostZoneSend` e `OneGrid%SelectedGhostZoneRecv` do tipo `MessageSet`.

Figura 3.16 - Fluxograma resumido da sub-rotina `TimeStep()`.



A estrutura `OneGrid` é uma estrutra do tipo `Grid` que é a principal estrutura de dados do BRAMS, conforme descrita e mostrada na Figura 3.7 da Seção 3.2, ela armazena todas as informações e parâmetros relativos ao subdomínio, comunicação paralela, informações sobre a grade, índices dos vetores, etc...

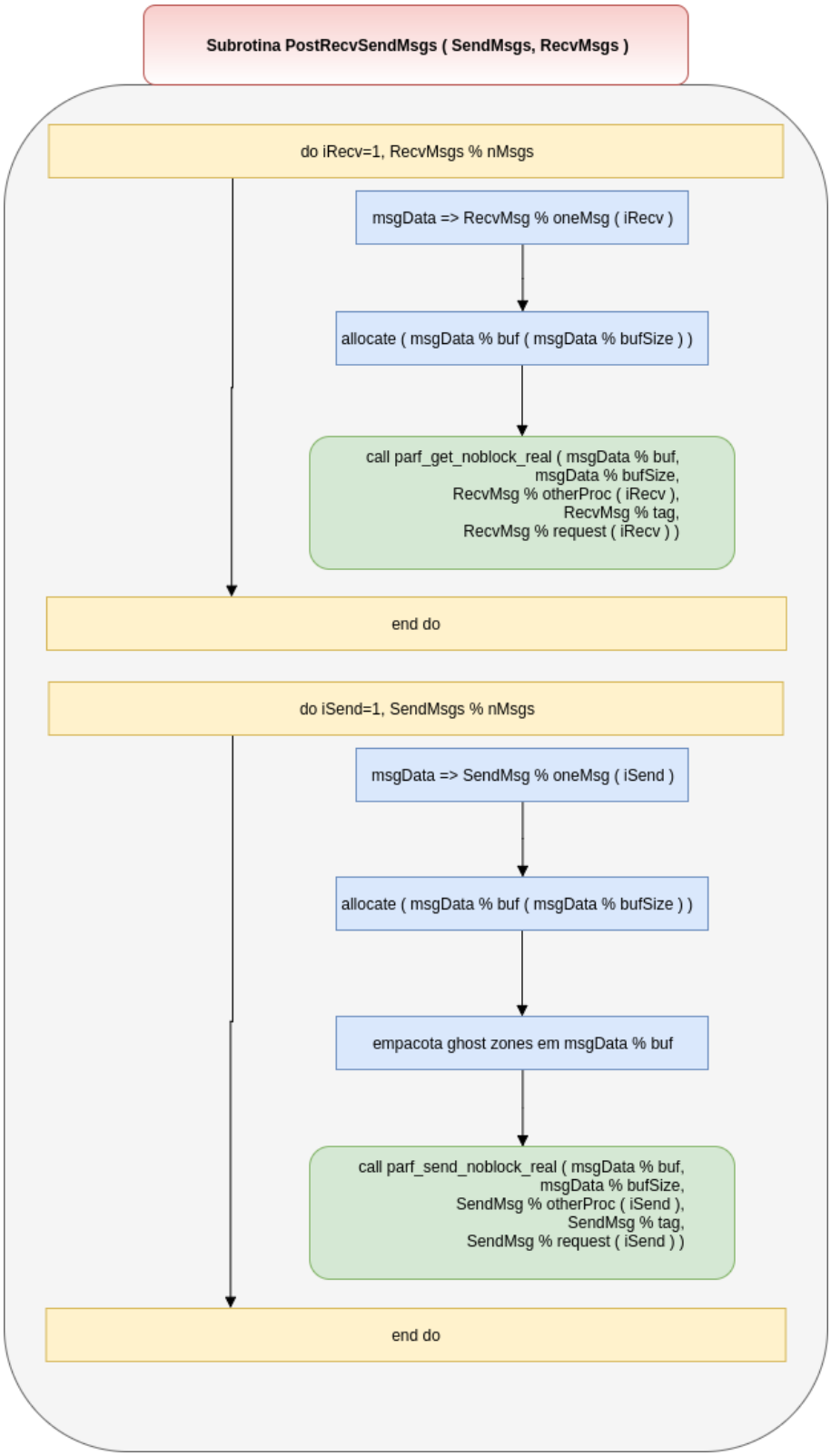
As sub-estruturas `OneGrid%SelectedGhostZoneSend` e `OneGrid%SelectedGhostZoneRecv` da `OneGrid` são estruturas (Figura 3.11) que armazenam a lista dos processos vizinhos que devem comunicar suas *Ghost Zones*, e também um vetor para armazenar todas as *Ghost Zones* a serem empacotadas para o envio/recebimento.

A Figura 3.17 mostra o fluxograma da `PostSendRecvMsgs()`. Esta sub-rotina recebe como argumento duas estruturas do tipo `MessageSet`, chamadas internamente de `SendMsgs` e `RecvMsgs`, com informações referentes ao envio e recebimento dos dados entre os processos, respectivamente.

Resumidamente, a sub-rotina `PostSendRecvMsgs()` pode ser divida em dois blocos. O primeiro bloco é delimitado por um laço de repetição `do`, que percorre o vetor de estruturas `oneMsg` (do tipo `MessageData`) que contem `nMsgs` mensagens a serem recebidas. É neste bloco que os recebimentos das mensagens MPI são feitos e os *buffers* são alocados para armazenar os dados recebidos. A função MPI `MPI_Irecv()` é executada dentro da sub-rotina `parf_get_noblock_real()` que recebe como parâmetros o *buffer* a ser recebido (`buf`), o tamanho do *buffer* (`bufSize`), o processo de quem está recebendo os dados (`otherProc(iRecv)`), e um campo para armazenar o *request* desta mensagem (`request(iRecv)`).

O segundo bloco da sub-rotina `PostSendRecvMsgs()` (segundo laço `do`) é onde ocorre o empacotamento das *Ghost Zones* em *buffers* para em seguida efetuar o envio MPI através da função `MPI_Isend()` que está embutida na sub-rotina `parf_send_noblock_real()`, que recebe basicamente os mesmos parâmetros que a sub-rotina de recebimento, sendo os mais importantes: o *buffer* a ser enviado (`buf`), seu respectivo tamanho (`bufSize`), o processo destino (`otherProc(iSend)`), dentre outros. Cada iteração do laço `do` executa o envio de um determinado conjunto de *buffers* para um determinado processo vizinho.

Figura 3.17 - Fluxograma resumido da subrotina `PostRecvSendMsgs()`.

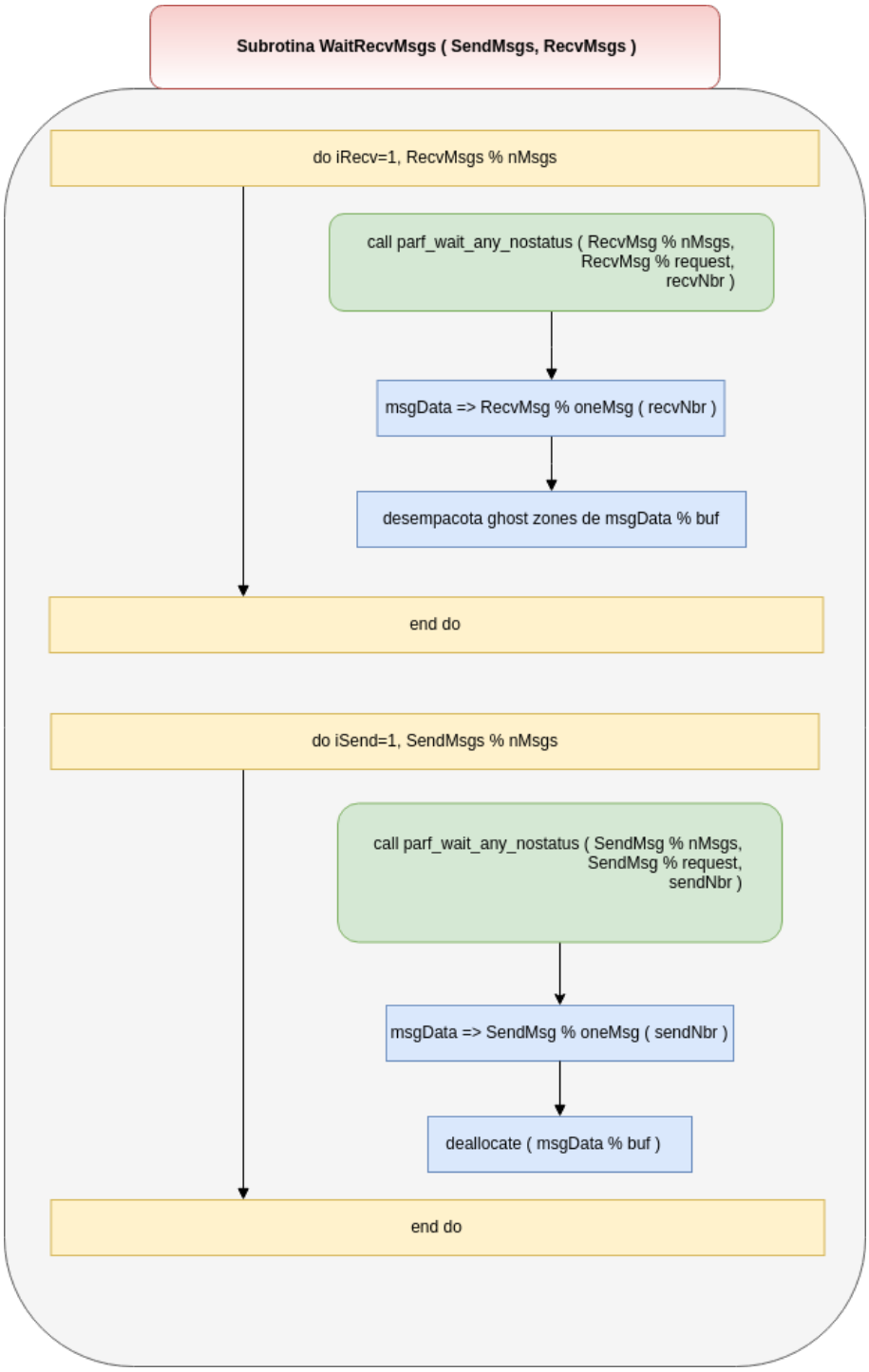


Depois que a sub-rotina `PostSendRecvMsgss()` é finalizada, alguns cálculos computacionais são executados (na rotina `TimeStep()`) e em seguida a sub-rotina `WaitRecvMsgs()` é chamada. Esta sub-rotina é responsável por executar as funções `MPI_Wait()` referentes aos envios e recebimentos que foram postados anteriormente pela `PostSendRecvMsgss()`.

A Figura 3.18 apresenta o fluxograma resumido das tarefas executadas na sub-rotina `WaitRecvMsgs()`, que também pode ser dividida em dois blocos. No primeiro bloco delimitado pelo primeiro laço `do`, igualmente ao primeiro bloco da sub-rotina `PostSendRecvMsgs()`, percorre todas as mensagens que supostamente foram enviadas. Este bloco é responsável por executar as funções `MPI_Wait()` tanto para as mensagens que foram recebidas quanto para as que foram enviadas, e desempacotar os *buffers* recebidos em suas *Ghost Zones* respectivas.

O segundo bloco percorre todas as mensagens que foram enviadas, executando a função MPI `MPI_Wait()` e desalocando os *buffers* que foram alocados para o envio.

Figura 3.18 - Fluxograma resumido da sub-rotina `WaitRecvMsgs()`.



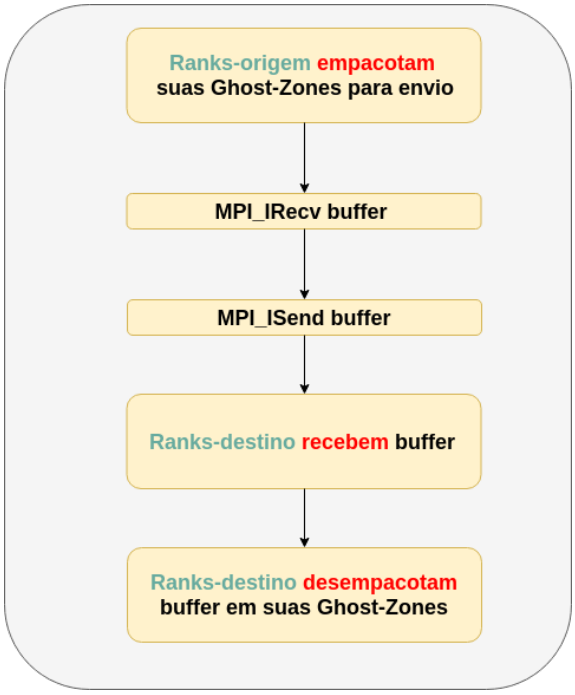
4 IMPLEMENTAÇÃO DA COMUNICAÇÃO SHM NO BRAMS

Conforme mencionado anteriormente, a avaliação do desempenho da comunicação unilateral SHM será feita para o modelo BRAMS, mas considerando-se um subconjunto de dados e programas relativos ao módulo da dinâmica do modelo BRAMS, chamado aqui de dinâmica isolada do BRAMS, de forma a não requerer a execução completa do modelo. Entretanto, essa dinâmica isolada inclui o esquema de comunicação MPI desenvolvido para paralelização do modelo como detalhado na Seções 3.1, 3.2 e 3.3). Assim, no restante do texto, eventuais referências ao modelo BRAMS devem ser entendidas como sendo feitas ao módulo da dinâmica isolada do BRAMS.

Neste capítulo, serão descritas as implementações adicionadas ao módulo da dinâmica isolada do BRAMS a fim de se obter uma comunicação paralela híbrida, ou seja, na troca de mensagens em que os processos são executados num mesmo nó computacional de memória compartilhada utiliza-se comunicação unilateral *Shared Memory* (SHM), enquanto que, para processos executados em nós computacionais diferentes, utiliza-se a comunicação MPI convencional bilateral já existente no BRAMS, `MPI_Isend() / MPI_Irecv()` (IS/IR). Dessa maneira, tem-se uma comunicação paralela MPI híbrida: MPI IS/IR + MPI SHM.

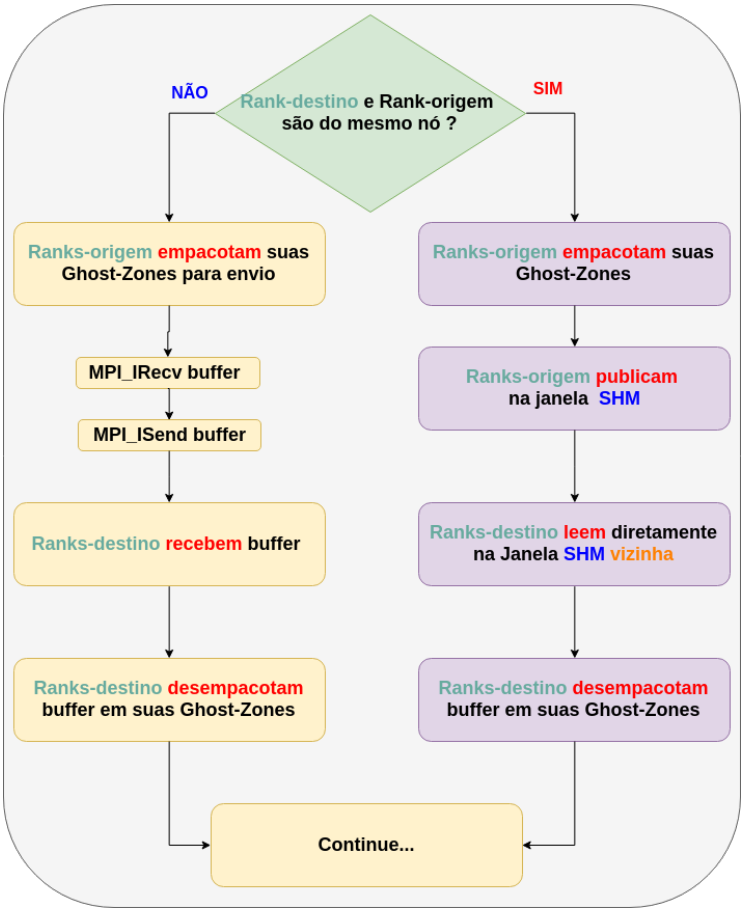
O modelo de programação paralela existente originalmente no BRAMS está ilustrado no fluxograma da Figura 4.1. A cada iteração da rotina `TimeStep()` cada processo seleciona as bordas de seus sub-domínios e as empacota num *buffer* para enviá-los aos seus respectivos processos vizinhos. Em seguida cada processo recebe esses *buffers* de seus vizinhos (`MPI_Irecv()`) e depois envia o seu *buffer* para seus respectivos processos vizinhos (`MPI_Isend()`). Somente depois que todos os envios e recebimentos foram concluídos é que cada processo desempacota seus respectivos *buffers* recebidos em suas *Ghost Zones*, para então seguir os cálculos computacionais.

Figura 4.1 - Fluxograma resumido da comunicação paralela no BRAMS.



A Figura 4.2 apresenta o fluxograma da nova comunicação híbrida MPI IS/IR + MPI SHM implementada no BRAMS. Para cada troca de mensagens MPI que deve ser executada, o processo remetente verifica se o processo-destinatário pertence ao mesmo nó computacional. Caso não seja, a troca de mensagens MPI ocorre da maneira original bilateralmente (IS/IR) (como descrito na Figura 4.1), seguindo-se o caminho à esquerda do fluxograma; Caso seja do mesmo nó, a troca de mensagens é feita via comunicação unilateral MPI *Shared Memory* (SHM), seguindo o caminho à direita do fluxograma. O processo em questão empacota as suas próprias *Ghost Zones* em *buffers*; publica-as na janela de memória SHM, e faz a leitura dos *buffers* correspondentes às *Ghost Zones* dos processos vizinhos nas partes correspondentes da janelas SHM. Finalmente, o processo considerado desempacota todos os *buffers* "recebidos"(lidos) em suas respectivas *Ghost Zones*.

Figura 4.2 - Fluxograma resumido da nova comunicação híbrida paralela no BRAMS.



Uma sub-rotina foi criada especialmente para classificar quais os processos que compartilham (ou não) do mesmo nó computacial, chamada de `translate_ranks()`. O código a seguir mostra esta sub-rotina em detalhes:

```
subroutine translate_ranks(wcomm, shmcomm, partners, partners_map, n_partners)
    integer                       , intent(in)  :: shmcomm, wcomm
    integer                       , intent(in)  :: n_partners
    integer, dimension(n_partners), intent(in)  :: partners
    integer, dimension(n_partners), intent(out) :: partners_map
    integer                                     :: world_group, shared_group
    integer                                     :: ierror, i
    call MPI_Comm_group(wcomm, world_group, ierror)
    call MPI_Comm_group(shmcomm, shared_group, ierror)
    call MPI_Group_translate_ranks(world_group,  &
                                   n_partners,   &
                                   partners,     &
                                   shared_group, &
                                   partners_map, &
```

```
15                                      ierror)
16     return
17   end subroutine translate_ranks
```

A sub-rotina `translate_ranks()` recebe como argumento o comunicador global (`wcomm`), o comunicador local SHM (`shmcomm`), um vetor contendo a lista dos processos globais (`partners()`), um segundo vetor *intent out* (`partners_map()`) que receberá a relação de cada processo identificando se compartilham ou não do mesmo nó que o processo corrente. Cada posição deste vetor terá o valor `MPI_UNDEFINED` se estiver em nós diferentes, e caso contrário terá o valor do número do processo local.

A comunicação SHM ocorre, de uma maneira resumida, inicialmente com cada processo empacotando suas bordas em *buffers* assim como na comunicação IS/IR. Porém, na comunicação convencional, cada processo envia e recebe um *buffer* de cada processo-destino vizinho, enquanto na comunicação SHM todos os *buffers* foram empacotados em um único *buffer* chamado de `SHMbuf`. Este `SHMbuf` é publicado em uma janela de memória compartilhada SHM e torna-se visível entre os processos-destinos vizinhos que pertencerem ao mesmo nó computacional.

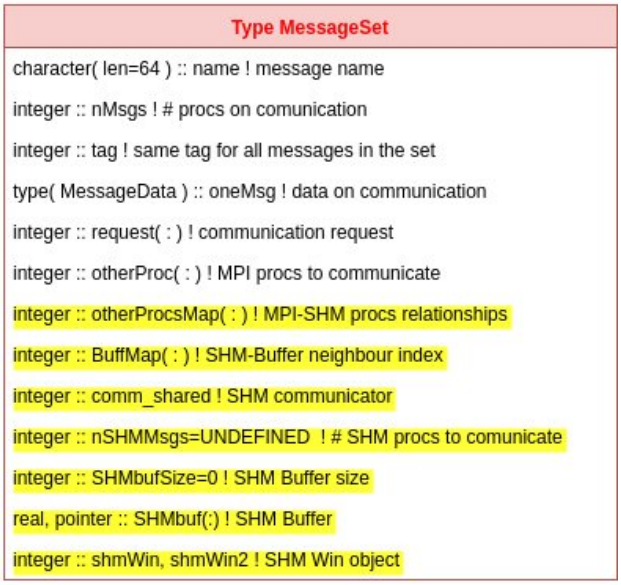
No próximo passo, cada processo consulta o endereço da janela SHM (`SHMbuf`) do processo vizinho, do qual deveria receber um *buffer*, na comunicação convencional IS/IR. Isso permite ao processo ler e/ou escrever diretamente à janela de memória SHM do processo vizinho, sem que ele participe dessa atividade. Neste caso, cada processo desempacota diretamente do `SHMbuf` vizinho suas *Ghost Zones*, evitando dessa forma transferir os dados via funções MPI (`MPI_Irecv()` / `MPI_Isend()`). Com isso a troca de mensagens MPI SHM é concluída.

Para que este novo modelo de comunicação híbrida fosse possível, foi necessário alterar algumas estruturas de dados do BRAMS (descritas na seção 3.2). As principais implementações executadas serão detalhadas nas seções a seguir.

4.1 Novas estruturas de dados para suportar comunicação unilateral SHM

A principal estrutura de dados do BRAMS que sofreu maiores alterações foi a `MessageSet`. A Figura 4.3 apresenta o diagrama desta estrutura de dados com as modificações destacadas em amarelo.

Figura 4.3 - Estrutura de dados do tipo `MessageSet` modificada.



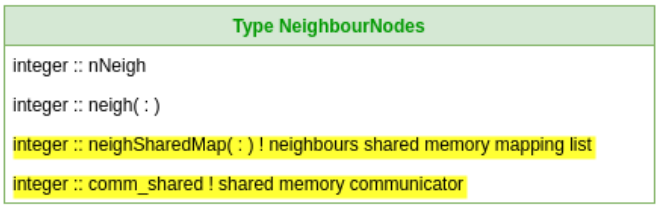
Outros campos também foram adicionados à estrutura de dados `NeighbourNodes`, para que fosse possível coletar dados da comunicação SHM em outro nível dentro do BRAMS.

Os novos campos adicionados à `MessageSet` foram:

`otherProcsMap(:)`	::	vetor de inteiros que armazena em cada posição um mapeamento do processo vizinho em relação a estar ou não no mesmo nó que o processo corrente;
`BuffMap(:)`	::	vetor (SHM) que armazena a posição de cada processo vizinho dentro do `SHMBuf`;
`comm_shared`	::	novo comunicador local SHM;
`nSHMMsgs`	::	variável contendo a quantidade total de mensagens a serem trocadas via SHM;
`SHMbufSize`	::	variável indicando o tamanho total do `SHMbuf`;
`SHMbuf(:)`	::	vetor que armazena todos os *buffers* juntos em um janela de memória compartilhada SHM;

A Figura 4.4 mostra o diagrama da `NeighbourNodes` modificada.

Figura 4.4 - Estrutura de dados do tipo `NeighbourNodes`.



Foram adicionados dois novos campos na estrutura:

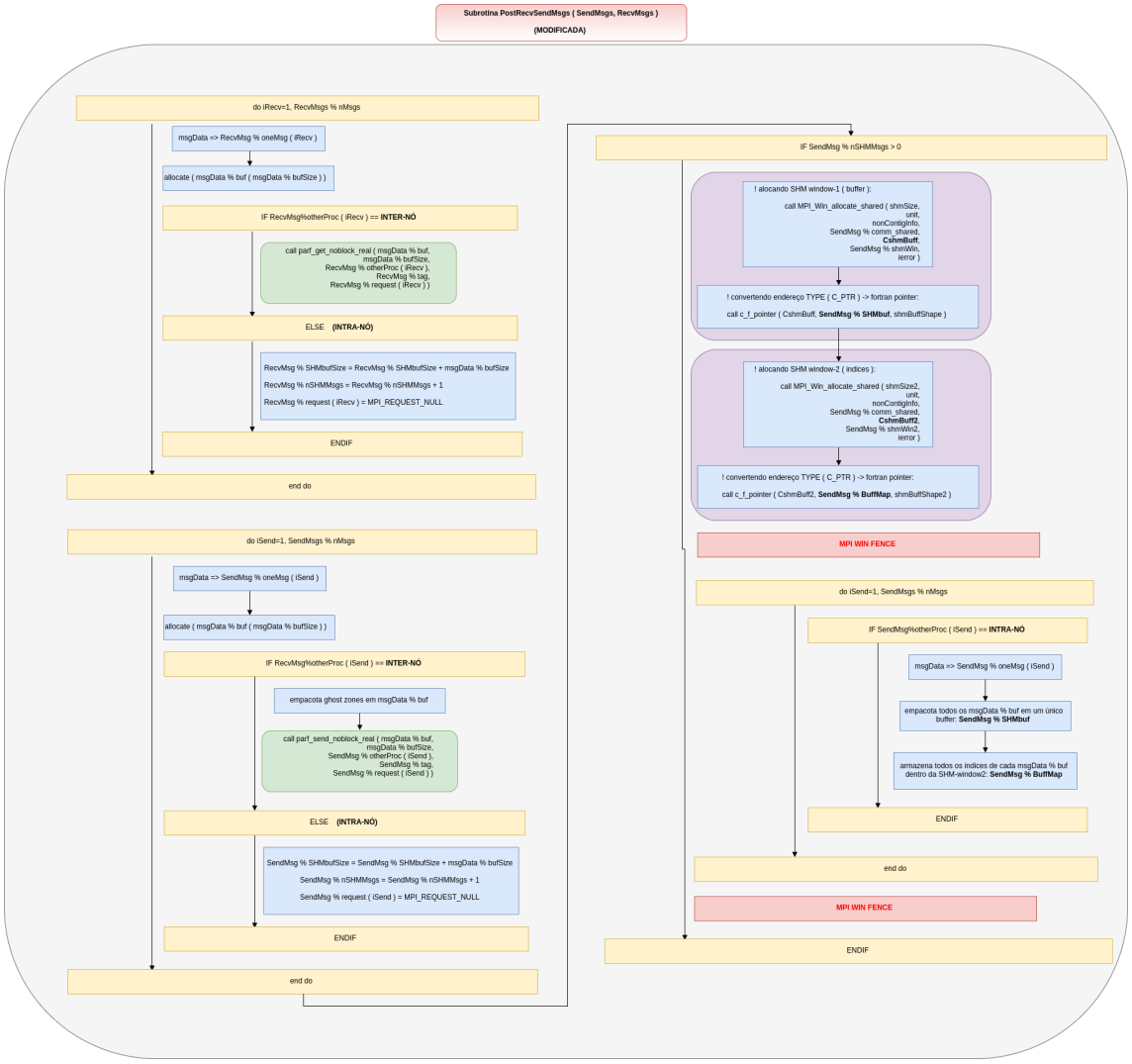
`neighSharedMap(:)`	::	vetor que armazena a relação dos processos vizinhos quanto a estarem ou não no mesmo nó;
`com_shared`	::	novo comunicador local SHM;

Os novos campos da `NeighbourNodes` coletam na etapa inicial do BRAMS as suas respectivas informações, de forma que a estrutura `MessageSet` coleta esses dados a partir desta estrutura. Isso ocorre na sub-rotina `CreateMessageSet()`.

4.2 Novas funções e subrotinas para suportar comunicação unilateral SHM

A Figura 4.5 apresenta o fluxograma simplificado da nova sub-rotina `PostRecvSendMsgs()` modificada para postar as novas comunicações SHM além das comunicações IS/IR originais. Na primeira parte do fluxograma (à esquerda) podemos notar que há dois blocos de laços `do` onde são feitas as comunicações MPI `MPI_Irecv()` e `MPI_Isend()`, respectivamente, assim como na sub-rotina original.

Figura 4.5 - Fluxograma resumido da nova sub-rotina `PostRecvSendMsgs` com as implementações SHM.



No entanto, como mencionado acima, antes que um processo envie uma mensagem é feita uma verificação se o processo destinatário está ou não no mesmo nó computacional de forma a se optar pela comunicação unilateral SHM ou pela comunicação bilateral assíncrona e sem bloqueio. Essa verificação é feita pelo seguinte condicional:

```
IF RecvMsg%otherProcMap(iRecv) == INTER-NÓ.
```

Este `IF` verifica se o processo destinatário não está no mesmo nó computaci-

onal que o processo corrente e, no caso afirmativo, executa-se a comunicação original do BRAMS (IS/IR); no caso negativo, ou seja, se o processo destino estiver no mesmo nó computacional que o processo corrente (ELSE INTRA-NÓ), então calcula-se o tamanho total do SHMbuf(:) acumulando os tamanhos dos *buffers* destinados à comunicação SHM, e armazena em RecvMsg%SHMbufSize. Isso ocorre tanto no primeiro quanto no segundo bloco das comunicações MPI_Irecv() e MPI_Isend(), respectivamente.

A segunda parte do fluxograma (à direita) mostra resumidamente as ações referentes à comunicação SHM apenas. Primeiramente, cada processo publica seu SHMbuf numa janela de memória compartilhada SHM e em seguida converte o endereço de memória desta janela do tipo TYPE(C_PTR) em um ponteiro em Fortran, através da função c_f_pointer() da biblioteca ISO_C_BINDING. Em seguida, cada processo empacota todos os buffers em um único SHMbuf(:). Uma segunda janela SHM foi usada para armazenar as posições dos *buffers* de cada processo vizinho dentro do SHMbuf(:).

A Figura 4.6 ilustra cada um dos *buffers* empacotados em vetores buf(:) (representados pelos retângulos em azul), destinados ao seus respectivos processos-destinos no caso da comunicação MPI IS/IR. Cada *buffer* fica armazenado no vetor buf(:) em uma posição do vetor da sub-estrutura (oneMsg(:)), cada posição de oneMsg(:) refere-se a uma mensagem a ser enviada e/ou recebida.

No caso da comunicação ocorrer via SHM, a Figura 4.7 mostra como os *buffers* são posicionados em uma primeira janela SHM, e como os índices de cada *buffer* são armazenados em uma segunda janela SHM, para que os vizinhos possam localizar seu respectivo *buffer* dentre todos que estão na janela SHM.

A primeira janela SHM (SHMbuf(:)) armazena todos os *buffers*, de todos os processos-vizinhos que compartilham do mesmo nó, sequencialmente. A segunda janela SHM (BuffMap(:)) armazena, para cada processo vizinho, o índice do início do *buffer* seguido do tamanho deste *buffer*. Dessa forma, cada processo vizinho conhece onde inicia e termina o seu respectivo *buffer* para acessá-lo.

Figura 4.6 - Esquema dos *buffers* empacotados para envio na comunicação MPI bilateral IS/IR.

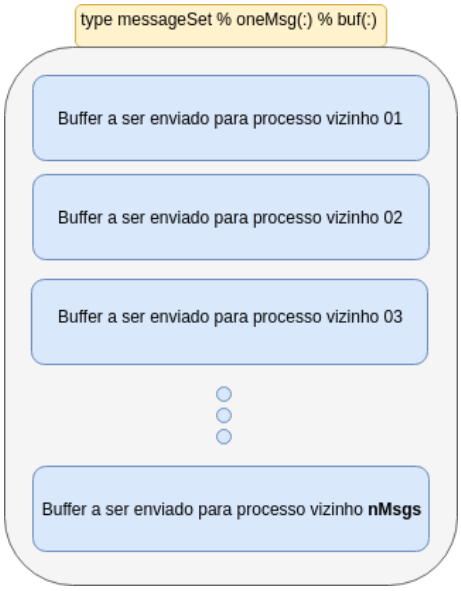
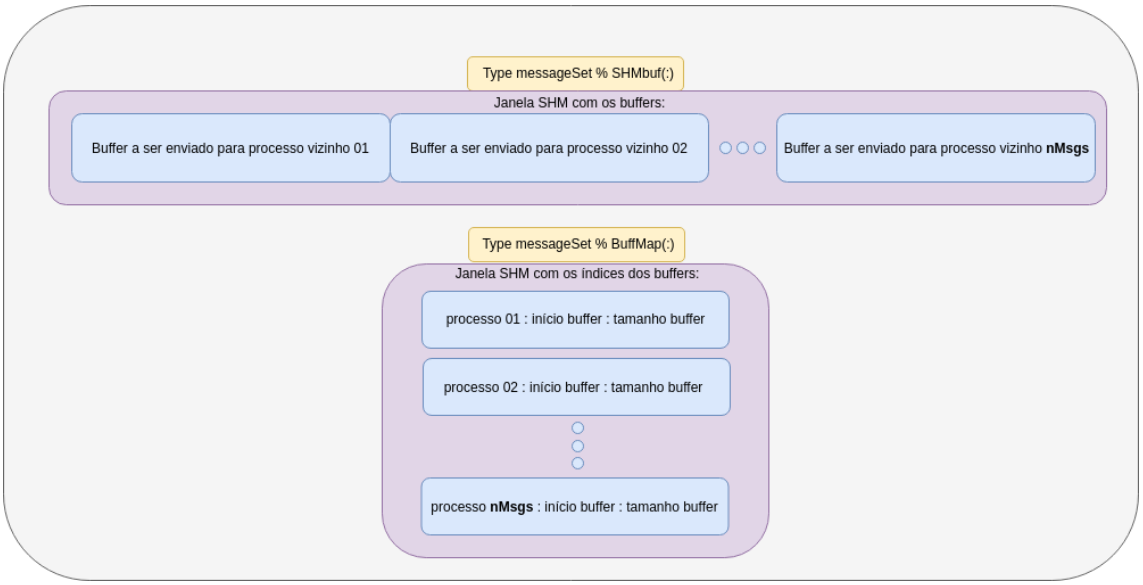


Figura 4.7 - Esquema das duas janelas SHM. A primeira com os *buffers* empacotados para envio na comunicação MPI unilateral SHM, e a segunda com os índices de cada *buffer*.

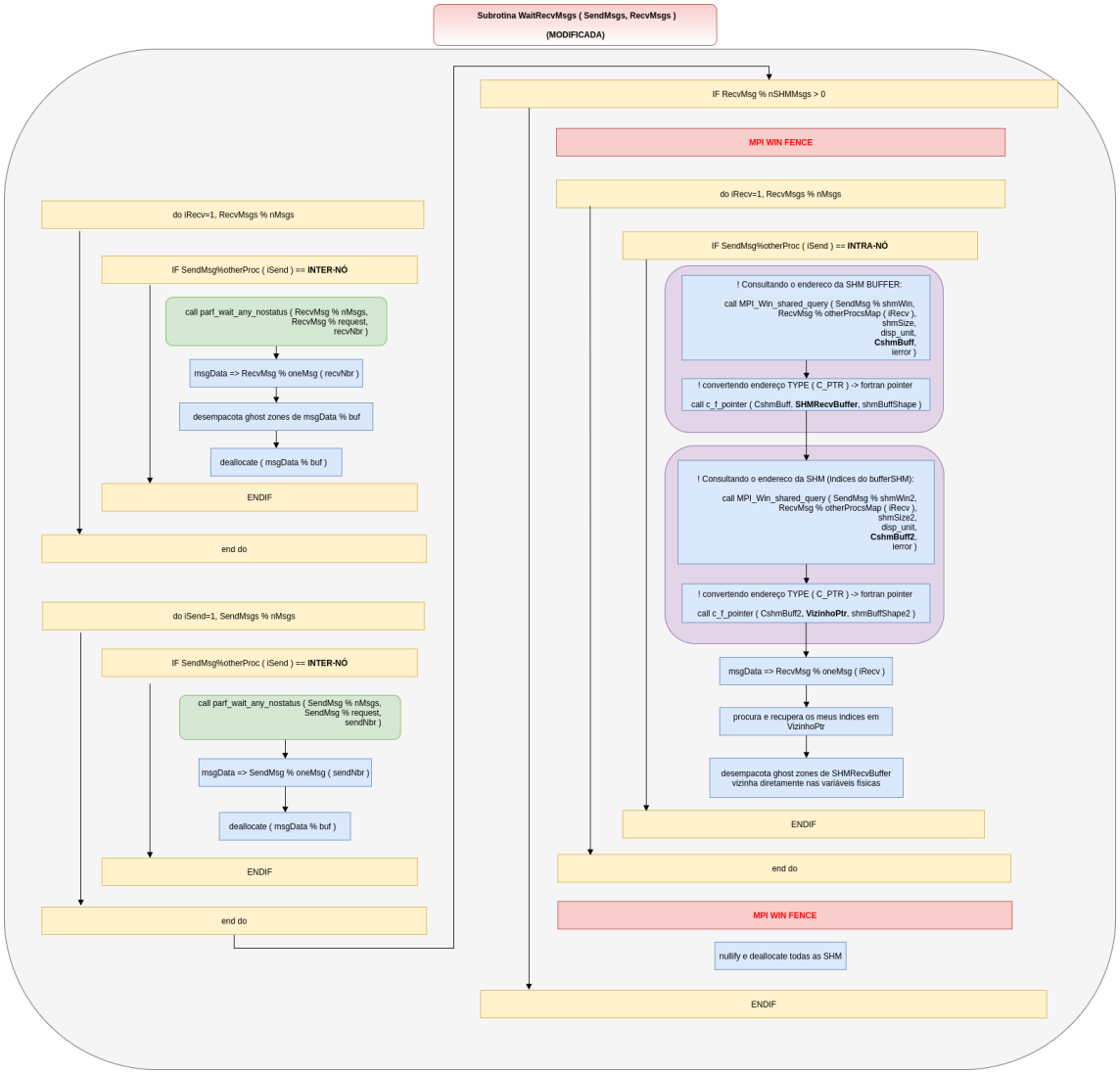


O trecho de código a seguir, retirado da nova sub-rotina `PostRecvSendMsgs()` mostra, com mais detalhes, a criação e publicação das duas janelas SHM sendo realizada, e em seguida o empacotamento dos `buffers`:

```
subroutine PostRecvSendMsgs(SendMsg, RecvMsg)
    (...)
    integer                         :: shmBuffShape(1), shmBuffShape2(1)
    integer(kind=MPI_ADDRESS_KIND)  :: shmSize, shmSize2
    TYPE(C_PTR)                     :: CshmBuff, CshmBuff2
    (...)
    !SHM Non contig mode:
    call MPI_Info_create(nonContigInfo, ierror)
    call MPI_Info_set(nonContigInfo, "alloc_shared_noncontig", "true", ierror)

    !alocando o shape do ponteiro F a ser convertido:
    shmBuffShape(1)=SendMsg%SHMbufSize

    !alocando SHM window:
    call MPI_Win_allocate_shared(shmSize,               &
                                 unit,                  &
                                 nonContigInfo,         &
                                 SendMsg%comm_shared,   &
                                 CshmBuff,              &
                                 SendMsg%shmWin,        &
                                 ierror)
    call c_f_pointer(CshmBuff, SendMsg%SHMbuf, shmBuffShape)

    shmSize2=((SendMsg%nSHMMsgs*3)+1)*c_sizeof(iSend)
    shmBuffShape2(1)=((SendMsg%nSHMMsgs*3)+1)
    !publicando a SHM2 do vetor com as infos do mapeamento dos buffers-vizinhos:
    call MPI_Win_allocate_shared(shmSize2,              &
                                 unit,                  &
                                 nonContigInfo,         &
                                 SendMsg%comm_shared,   &
                                 CshmBuff2,             &
                                 SendMsg%shmWin2,       &
                                 ierror)
    call c_f_pointer(CshmBuff2, SendMsg%BuffMap, shmBuffShape2)
    call FieldSection2Buffer(node%vTabPtr%var_p_4D,     &
                             node%vTabPtr%idim_type,    &
                             node%xStart,               &
                             node%xEnd,                 &
                             node%yStart,               &
                             node%yEnd,                 &
                             SendMsg%SHMbuf,            &
                             lastBuffer)
    (...)
end subroutine PostRecvSendMsgs
```

Na Figura 4.8, é apresentado o fluxograma simplificado da nova sub-rotina `WaitRecvMsgs()` modificada para executar além das comunicações IS/IR originais, também a transferência dos dados referentes às comunicações SHM.

Figura 4.8 - Fluxograma resumido da nova sub-rotina `WaitRecvMsgs` com as implementações SHM.



Na primeira parte deste fluxograma (à esquerda da Figura 4.8), onde são executadas as comunicações MPI IS/IR originais do BRAMS, continua o mesmo que o original, porém dentro de um bloco de `if` confirmando se os processos envolvidos na comunicação não compartilham do mesmo nó computacional.

Na segunda parte do fluxograma (à direita da Figura 4.8), pode-se observar os novos procedimentos implementados referentes à comunicação SHM. Como na primeira parte todas as comunicações MPI IS/IR já foram feitas, agora só resta-

ram as comunicações entre processos que compartilham do mesmo nó. Então todo este novo bloco é executado dentro de um `if` onde apenas comunicações entre processos intra-nós são realizadas.

Primeiramente, cada processo consulta o endereço da janela de memória SHM do processo vizinho, o qual supostamente deveria enviar uma *Ghost Zone* se fosse feita via IS/IR. Em seguida este endereço precisa ser convertido para ponteiro em Fortran, isso é feito através da função `c_f_pointer()`. Depois disso, o mesmo processo se repete mas agora para uma segunda janela SHM, a qual contem informações de onde inciam e terminam cada *buffer* de cada processo vizinho dentro do `SHMbuf`. Uma vez que cada processo conhece onde exatamente incia e termina seu respectivo *buffer* dentro do `SHMbuf`, dá-se início ao desempacotamento de suas *Ghost Zones* diretamente da janela SHM vizinha para suas bordas locais.

É muito importante notar que os acessos às janelas SHM vizinhas (leitura) são feitas dentro de um mesmo par de *Fences*. Isso garante que no momento em que os processos vizinhos acessarem essa área de memória remota, todos os processos já terminaram suas respectivas escritas, evitando assim tanto o problema de *Race Conditions* quanto o problema de acessar um dado remoto desatualizado.

O trecho de código a seguir, retirado da nova sub-rotina `WaitRecvMsgs()` mostra, com mais detalhes, o procedimento que cada processo realiza para consultar o endereço das janelas SHM vizinhas e desempacotar suas *Ghost Zones* em suas bordas locais:

```
1  subroutine WaitRecvMsgs(SendMsg, RecvMsg)
2      (...)
3      call MPI_Win_fence(0, SendMsg%shmWin, ierror)  !========== EPOCH
4      call MPI_Win_fence(0, SendMsg%shmWin2, ierror) !========== EPOCH
5      !CR: Consultando o endereco da SHM vizinha (indeces do bufferSHM):
6      call MPI_Win_shared_query(SendMsg%shmWin2,              &
7                                RecvMsg%otherProcsMap(iRecv), &
8                                shmSize2,                     &
9                                disp_unit,                    &
10                               CshmBuff2,                    &
11                               ierror)
12     shmBuffShape2(1)=RecvMsg%nSHMMsgs*3
13     call c_f_pointer(CshmBuff2, VizinhoPtr, shmBuffShape2)
14     !CR: Consultando o endereco da SHM BUFFER:
15     call MPI_Win_shared_query(SendMsg%shmWin,               &
16                               RecvMsg%otherProcsMap(iRecv), &
17                               shmSize,                      &
```

```
                              disp_unit,                    &
                              CshmBuff,                     &
                              ierror)
    shmBuffShape(1)=RecvMsg%SHMbufSize
    call c_f_pointer(CshmBuff, SHMRecvBuffer, shmBuffShape)
    msgData => RecvMsg%oneMsg(iRecv)
    (...)
    do iVizinhoPtr=2, (VizinhoPtr(1)*3)+1, 3
        if (VizinhoPtr(iVizinhoPtr) == myidlocal) then
            ! extract field sections from incoming SHM buffer
            call Buffer2FieldSection(node%vTabPtr%var_p_2D,                    &
                                     node%vTabPtr%idim_type,                   &
                                     node%xStart,                              &
                                     node%xEnd,                                &
                                     node%yStart,                              &
                                     node%yEnd,                                &
                                     SHMRecvBuffer(VizinhoPtr                  &
                                     (iVizinhoPtr+1):VizinhoPtr(iVizinhoPtr+2)),&
                                     lastBuffer)
            (...)
        endif
        (...)
    enddo
    (...)
end subroutine WaitRecvMsgs
```

5 RESULTADOS DE DESEMPENHO PARALELO

Apresentam-se aqui os resultados de desempenho paralelo referentes à comparação entre o uso da comunicação MPI bilateral convencional assíncrona e sem bloqueio com a comunicação MPI unilateral de memória compartilhada (SHM) para dois estudos de caso: o problema exemplo adotado e o módulo da dinâmica isolada do modelo BRAMS.

O problema exemplo adotado refere-se a um programa implementado por Hoefler e Balaji que resolve equações diferenciais numa malha 2D pelo método das diferenças finitas. Essa forma de resolução das equações diferenciais aparece no âmago de modelos de previsão numérica de tempo em geral, embora numa escala muito maior, tal como no caso do BRAMS. As comunicações MPI bilateral e unilateral*Shared Memory* foram implementadas e testadas primeiramente neste problema-exemplo (SOUZA et al., 2017) para depois serem portadas para o módulo da dinâmica isolada do BRAMS (SOUZA et al., 2018).

Escolheu-se utilizar o conjunto de compiladores PGI nos experimentos do problema exemplo, mas também para o módulo da dinâmica isolada do BRAMS, à semelhança do modelo BRAMS utilizado no CPTEC/INPE em modo operacional Entretanto, outros compiladores disponíveis na máquina Cray foram utilizados nos experimentos do problema exemplo, sendo que os resultados de desempenho paralelo aparecem em Anexo A.

Para quantificar o ganho de desempenho dos programas paralelos sobre as versões sequenciais, utilizou-se como base o *speedup* e a eficiência.

O *speedup* (S_p) (ou aceleração) é a relação entre o tempo gasto para executar uma tarefa com um único processo e o tempo gasto com p processos. Quanto mais próximo de p for o *speedup*, melhor, pois se aproxima do *speedup* linear (p), considerado ideal. É definido como:

$$S_p = \frac{t_1}{t_p} \tag{5.1}$$

Onde:
S_p : *speedup* utilizando p processos;
t_1 : tempo sequencial;
t_p : tempo utilizando p processos.

A eficiência (E_p) é o *speedup* normalizado pelo número de processos, ou seja, quanto mais próximo da unidade (ou 100%) melhor, à semelhança do *speedup* linear se aproximando de (p). A eficiência é definida por:

$$E_p = \frac{S_p}{p} \tag{5.2}$$

Onde:
E_p : eficência utilizando p processos;
S_p : *speedup* utilizando p processos;
p : número de processos.

Nesta seção, apresentam-se os resultados dos experimentos realizados com o problema exemplo (5.2) e com o módulo da dinâmica isolada do BRAMS (5.3), bem como uma breve descrição do ambiente computacional utilizado.

5.1 O ambiente computacional

Os resultados de desempenho paralelo aqui apresentados foram obtidos utilizando-se uma máquina CRAY. A tabela 5.1 mostra os tipos de processadores, a quantidade em cada nó, a quantidade de nós disponível e a quantidade de memória em cada nó.

Tabela 5.1 - Configuração geral da máquina CRAY. Fonte: Cray (2017)

Nodes		**CPUs on each Compute Node**		
Type	**Count**	**Type**	**Clock**	**Memory**
CPU	8	2 Ivy Bridge 12-core Intel Xeon	2.7 GHz	64 GB DDR3-1866
CPU	104	2 Broadwell 22-core Intel Xeon	2.2 GHz	128 GB DDR4-2400
CPU	4	2 Haswell 16-core Intel Xeon	2.3 GHz	128 GB DDR4-0213

A máquina possui 3 tipos de processadores: *Ivy Bridge* de 2,7GHz (12 *cores*), *Broadwell* de 2,2GHz (22 *cores*) e *Haswell* de 2,3GHz (16 *cores*). Cada nó disponibiliza dois processadores totalizando: 24, 44 e 32 *cores* por nó, respectivamente.

neste trabalho, foram escolhidos nós com processadores intel xeon broadwell, em razão de seu maior numero, e tambem por disporem de mais nucleos por no.

Assim como o supercomputador Tupã do CPTEC, essa máquina CRAY possui um gerenciador de filas do tipo PBS. E cada fila destina-se a executar o *job* submetido num determinado tipo de (nó) processador. A tabela 5.2 apresenta a configuração das filas disponíveis:

Tabela 5.2 - Configuração geral das filas da máquina CRAY. Fonte: Cray (2017)

Queue	Wall Time Limit	Node Limit	Max Queued	Max Running	Node Type
small	24h	16 nodes	30 jobs	4 jobs	Broadwell
large	24h	86 nodes	2 jobs	1 job	Broadwell
ivb12	24h	8 nodes	3 jobs	1 job	Ivy Bridge
hsw16	24h	4 nodes	3 jobs	1 job	Haswell

A CRAY oferece, nessa máquina, vários tipos de compiladores, como mostra a tabela 5.3:

Tabela 5.3 - Ambientes de compilação disponíveis na máquina CRAY. Fonte: Cray (2017)

Package	Module	Compilers	Version
Cray Compiling Environment (CCE)	PrgEnv-cray	craycc, crayftn	8.7.0
Intel XE Compilers	PrgEnv-intel	icc, icpc, ifort	17.0.4.196
GNU Compiler Collection (GCC)	PrgEnv-gnu	gcc, g++, gfortran	7.3.0
PGI Compilers	PrgEnv-pgi	pgcc, pgfortran	17.10.0

Os testes de desempenho foram executados nos nós computacionais bi-processados com processadores Intel Xeon *Broadwell* do tipo E5-2699.v4 de 2,2 GHz e 22 *cores* (44 por nó) com cache L1 de 64KB por *core*, L2 de 256KB por *core* e L3 compartilhado de 55MB. O ambiente de compilação escolhido foi o conjunto de compiladores PGI (PGI, 2017) e a biblioteca MPICH instalada nessa máquina.

5.2 Problema exemplo adotado

Muitas aplicações científicas, tais como modelos de previsão numérica de tempo, envolvem a solução de equações diferenciais parciais (EDPs). Existem muitos algoritmos na literatura que resolvem as EDPs aproximando uma solução pela discretização de grandezas contínuas da natureza, como por exemplo, aqueles

que implementam os métodos de diferenças finitas, elementos finitos e volumes finitos.

O método de diferenças finitas é um método de resolução de equações diferenciais, que se baseia na aproximação de derivadas (FINITE DIFFERENCE METHOD, 2018). O método dos elementos finitos é um método que trata de problemas de valores de contorno (PVC), subdividindo o domínio de um problema em partes menores, por isso o motivo do nome: elementos finitos. O método dos volumes finitos é largamente utilizado na resolução de problemas envolvendo transferência de calor ou massa e em mecânica dos fluidos, evoluiu do método das diferenças finitas, baseado na resolução de balanços de massa, energia e quantidade de movimento a um determinado volume de meio contínuo (FINITE VOLUME METHOD, 2018).

Modelos numéricos simulam a evolução do estado da atmosfera em passos de tempo discretos. A cada passo de tempo, de maneira geral, as operações envolvidas em sua execução paralela em MPI são sempre as mesmas: cálculos efetuados no subdomínio de cada processador, comunicação devida a dependências de dados entre subdomínios, atualização de cada subdomínio a partir dos dados referentes a subdomínios vizinhos.Isso também se aplica ao problema exemplo considerado.

A abordagem apresentada por Balaji et al. (2014) mostra um problema da equação de Poisson em duas dimensões. A equação de Poisson é uma equação de derivadas parciais de segunda ordem muito utilizada em dinâmica dos fluidos. Neste problema exemplo, a equação de Poisson foi utilizada para representar a distribuição de calor numa superfície, dada por:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = f(x, y) \tag{5.3}$$

O campo de temperaturas U é definido numa malha discreta em duas dimensões (x, y) de variação uniforme $\Delta x = \Delta y = h$. Aproximando-se as derivadas parciais de segunda ordem pelo método de diferenças finitas centradas, tem-se:

$$\frac{\partial^2 U}{\partial x^2} \approx \frac{U(x+h, y) - 2U(x, y) + U(x-h, y)}{h^2} \tag{5.4}$$

Simplificando-se essa equação por $U(x, y) = U_{i,j}$, $U(x+h, y) = U_{i+1,j}$, tem-se:

$$\frac{\partial^2 U}{\partial x^2} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{(\Delta x)^2} \tag{5.5}$$

Finalmente, a Equação 5.3 pode ser aproximada pelas duas derivadas parciais discretizadas e aproximadas pelo método de diferenças finitas pelo seguinte par de equações:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{(\Delta x)^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{(\Delta y)^2} \tag{5.6}$$

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \approx \frac{U_{i+1,j} + U_{i,j+1} - 4U_{i,j} + U_{i-1,j} + U_{i,j-1}}{h^2} \tag{5.7}$$

Essa discretização espacial bidimensional, é similar àquela do modelo BRAMS, apresentada na Figura 3.3 e utilizada para ilustrar sua divisão de domínio. A cruz vermelha mostra os pontos envolvidos no cálculo de um estêncil de 5 pontos, que usa os valores do campo nos pontos vizinhos para atualizar o valor no ponto central. A cada iteração do laço principal (*timestep*) cada ponto de grade da malha discretizada é atualizado pela média de seu próprio valor e dos valores dos 4 pontos vizinhos através do cálculo de um estêncil de 5 pontos, assim como explicado na seção 3.1.

Na atualização dos pontos nas bordas de cada subdomínio, cada processo precisa dos valores das fileiras de pontos vizinhos, que foram atualizadas pelos processos vizinhos, configurando um problema clássico de atualização de bordas em MPI, o qual demanda comunicação entre processos a cargo de subdomínios vizinhos.

5.2.1 Análise do desempenho para execução do programa exemplo em um único nó

O problema da Equação de Poisson em 2D foi modelado, implementado e disponibilizado por Hoefler e Balaji (BALAJI et al., 2014) originalmente em Linguagem C. Três versões foram implementadas e testadas neste trabalho: a versão sequencial, a versão paralela MPI com comunicação bilateral com as funções de envio e recebimento de mensagens com bloqueio `MPI_Send()` e `MPI_Recv()` (denotadas aqui por S/R) e a versão paralela MPI comunicação unilateral *Shared Memory* com alocação de memória contígua (SHM). Uma quarta versão, derivada desta

última, foi criada com a janela de memória compartilhada não-contígua (nomeada por SHM-NC). Esta última versão foi adotada com o objetivo de avaliar seu desempenho em relação à versão com janela de memória compartilhada alocada de forma contígua. Foi utilizada uma malha bidimensional discreta de 4.800 x 4.800 pontos, com 500 iterações no tempo e adicionando-se uma unidade de energia a cada passo de tempo em três posições da grade escolhidas aleatoriamente e mantidas fixas ao longo das iterações.

As versões paralelas foram compiladas com o compilador PGI 16.10.0 com opções *default*, exceto pela opção *-O3*, sendo executadas com 1, 4, 9, 16, 25 e 36 processos num único nó computacional, pois as codificações admitem que cada eixo do domínio seja particionado pelo mesmo número de processos MPI.

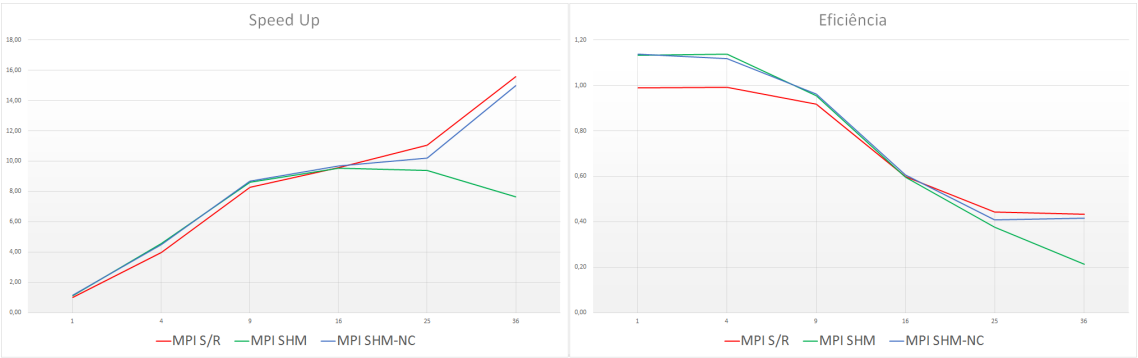
A Tabela 5.4 mostra os tempos de execução dessas versões em linguagem C num único nó (média de 5 execuções para cada caso) em função do número de processos, notando-se que as versões C MPI SHM são mais rápidas para até 16 processos e que à medida que se aumenta o número de processos, a versão SHM-NC tende a ficar mais rápida que a SHM. Entretanto, a versão SHM-NC ainda demanda um tempo ligeiramente maior que a versão MPI S/R. A versão SHM-NC é mais rápida que a SHM porque utiliza uma janela de memória não contígua na qual cada segmento é alocado exatamente no início de uma página da memória virtual.

Um fato curioso, é que as versões SHM executadas com 1 único processo foram mais rápidas que a própria versão sequencial. A mesma tabela e também a Figura 5.1 apresentam os correspondentes *speedup's* e eficiências, calculados em relação ao tempo da versão sequencial. Notam-se *speedup's* ligeiramente superlineares para as versões SHM e SHM-NC executadas com até 4 processos, mas o desempenho destas se degrada com o aumento do número de processos, com ênfase para a versão SHM, enquanto que a versão SHM-NC teve desempenho paralelo próximo da versão S/R. Nota-se uma degradação do desempenho de todas as versões paralelas C com o aumento do número de processos.

Tabela 5.4 - Tempos de execução em segundos, *speedup's* e eficiências para as versões paralelas em **linguagem C** MPI S/R, MPI SHM e MPI SHM-NC compiladas com **PGI** e executadas com 1 (1P), 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4.800 x 4.800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 43,14s	**1P**	**4P**	**9P**	**16P**	**25P**	**36P**
MPI S/R	43,31s	10,88s	5,22s	4,51s	**3,90**s	**2,77**s
MPI SHM	38,08s	**9,48**s	5,02s	4,52s	4,59s	5,64s
MPI SHM-NC	**37,95**s	9,64s	**4,98**s	**4,45**s	4,23s	2,88s
			Speedup			
MPI S/R	0,99	3,97	8,26	9,56	11,06	15,58
MPI SHM	1,13	4,55	8,59	9,53	9,39	7,65
MPI SHM-NC	1,14	4,47	8,67	9,70	10,19	14,98
			Eficiência			
MPI S/R	0,99	0,99	0,92	0,60	0,44	0,43
MPI SHM	1,13	1,14	0,95	0,60	0,38	0,21
MPI SHM-NC	1,14	1,12	0,96	0,61	0,41	0,42

Figura 5.1 - Desempenho paralelo das versões em **linguagem C** MPI S/R , MPI SHM e MPI SHM-NC.



Versão MPI S/R em vermelho, MPI SHM em verde e MPI SHM-NC em azul; compiladas com **PGI** e executadas com 1 (1P), 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo. Gráficos do*speedup* (à esquerda) e da eficiência (à direita) em função do número de processos.

Essas versões, foram convertidas para Fortran 90 com o objetivo de comparar o desempenho do MPI SHM em ambas as linguagens e porque modelos numéricos de previsão de tempo e clima, como o modelo BRAMS por exemplo, são

escritos em Fortran 90 (FREITAS et al., 2016). Assim, foram também geradas as quatro versões correspondentes àquelas em C (Fortran 90 sequencial, MPI S/R, MPI SHM e MPI SHM-NC).

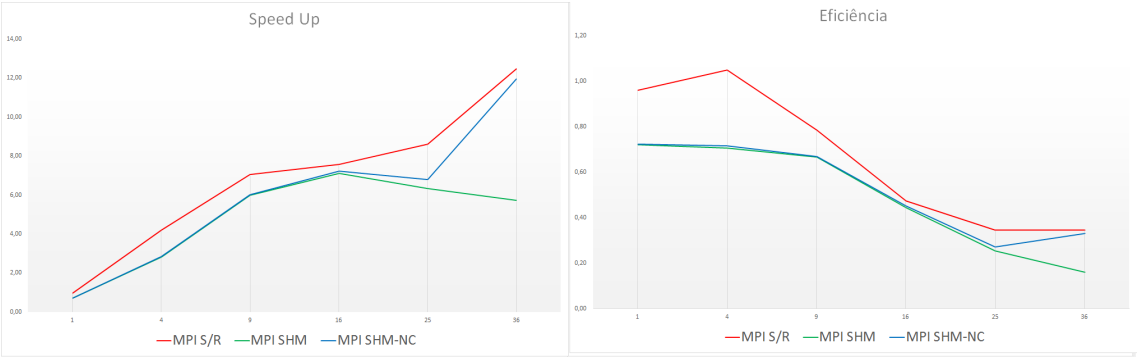
Todas essas versões paralelas, foram compiladas com o compilador PGI com opções *default*, exceto pela opção *-O3*, sendo executadas com 1, 4, 9, 16, 25 e 36 processos num único nó computacional (exceto pela versão híbrida) para o mesmo domínio de 4.800 x 4.800 pontos.

Tabela 5.5 - Tempos de execução em segundos, *speedup's* e eficiências para as versões em linguagem **Fortran 90** MPI S/R, MPI SHM e MPI SHM-NC compiladas com **PGI** e executadas com 1 (1P), 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4.800 x 4.800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 33,30s	**1P**	**4P**	**9P**	**16P**	**25P**	**36P**
MPI S/R	**34,67**s	**7,93**s	**4,72**s	**4,40**s	**3,87**s	**2,67**s
MPI SHM	46,18s	11,79s	5,56s	4,68s	5,26s	5,81s
MPI SHM-NC	46,12s	11,65s	5,54s	4,62s	4,90s	2,79s
Speedup						
MPI S/R	0,96	4,20	7,06	7,56	8,61	12,46
MPI SHM	0,72	2,82	5,99	7,12	6,33	5,73
MPI SHM-NC	0,72	2,86	6,01	7,21	6,80	11,94
Eficiência						
MPI S/R	0,96	1,05	0,78	0,47	0,34	0,35
MPI SHM	0,72	0,71	0,67	0,44	0,25	0,16
MPI SHM-NC	0,72	0,71	0,67	0,45	0,27	0,33

A Tabela 5.5 mostra os tempos de execução dessas versões Fortran 90 num único nó (média de 5 execuções para cada caso) em função do número de processos, notando-se que as versões Fortran 90 S/R são mais rápidas que as versões SHM e SHM-NC para qualquer número de processos, embora à medida que se aumenta o número de processos, a versão SHM-NC tende a melhorar seu desempenho em relação à versão S/R, a ficar mais rápida que a SHM e a ter um tempo ligeiramente pior que a versão MPI S/R. A mesma tabela e também a Figura 5.2 apresentam os correspondentes *speedup's* e eficiências, calculados em relação ao tempo da versão sequencial. Nota-se uma degradação do desempenho de todas as versões paralelas Fortran 90 com o aumento do número de processos.

Figura 5.2 - Desempenho paralelo das versões em linguagem **Fortran 90** MPI S/R, MPI SHM e MPI SHM-NC.



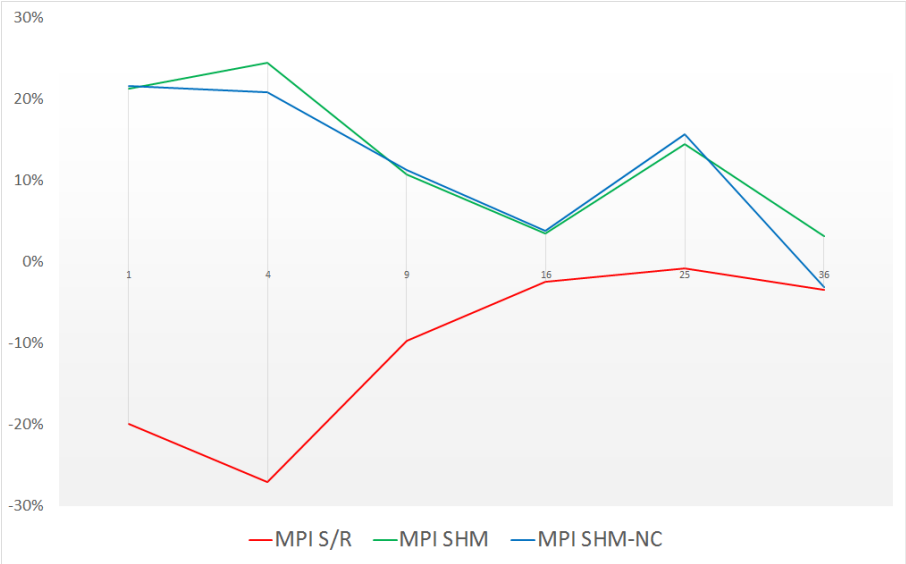
Versão MPI S/R em vermelho, MPI SHM em verde e MPI SHM-NC em azul; compiladas com **PGI** e executadas com 1 (1P), 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4.800 x 4.800 pontos com 500 passos de tempo. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos.

Finalmente, a Figura 5.3 mostra a diferença percentual do tempo de execução das versões Fortran 90 em relação às correspondentes versões C em função do número de processos, expressa por $((T_f/T_c) - 1) \times 100$. As versões Fortran 90 S/R mostram valores negativos dessa diferença porcentual por terem apresentado tempos de execução menores que as correspondentes versões C, embora a diferença fique menor à medida que se aumenta o número de processos. Por outro lado, as versões Fortran 90 MPI SHM e SHM-NC apresentam valores positivos dessa diferença porcentual pois têm tempos de execução maiores que as correspondentes versões em C.

Em tese, as versões com a comunicação MPI SHM deveriam apresentar melhor desempenho tanto em C quanto em Fortran 90, uma vez que essa funcionalidade foi criada para explorar a memória compartilhada na comunicação unilateral.

Os testes indicaram que as versões C MPI SHM eram mais rápidas que as correspondentes versões S/R, para um número baixo de processos por nó, porém isso não aconteceu com as versões Fortran 90. Esse fato levou à investigação das diferenças de desempenho entre as versões das duas linguagens.

Figura 5.3 - Diferença porcentual das versões Fortran 90 em relação às correspondentes versões em C em função do número de processos (MPI S/R em vermelho, MPI SHM em verde e MPI SHM-NC em azul).



Versão MPI S/R em vermelho, MPI SHM em verde e MPI SHM-NC em azul; compiladas com **PGI** e executadas com 1 (1P), 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4.800 x 4.800 pontos com 500 passos de tempo. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos.

O tempo de execução da versão Fortran 90 S/R executada com um processo (1P) é muito próximo da versão Fortran 90 sequencial (4% maior), mas as versões Fortran 90 SHM 1P demandam quase 40% mais tempo. Uma simples comparação entre o código das versões MPI SHM C e Fortran 90 mostra que esse tempo adicional deve-se à conversão de ponteiros C para ponteiros Fortran 90, que obviamente não ocorre na versão C, detalhada a seguir.

5.2.2 Comparação de versões sequenciais Fortran 90

As janelas de memórias compartilhada do MPI SHM são criadas pela função `MPI_Win_allocate_shared()` que retorna um endereço de memória por meio de um ponteiro, mas este ponteiro é do tipo C (`TYPE(C_PTR)`), o qual precisa ser convertido para um ponteiro Fortran 90 (`pointer`) utilizando-se a função `c_f_pointer()`, como ilustrado no trecho de programa a seguir para a matriz `avector`. Por outro lado, as versões Fortran 90 sequencial e MPI S/R não necessitam usar

esse esquema de conversão de ponteiros.

Em consequência disso, foram feitos testes com diferentes versões sequenciais com o objetivo de identificar qual o tempo adicional demandado ao se utilizar ponteiros do tipo `TYPE(C_PTR)` em Fortran 90 em comparação com a utilização usual de alocação dinâmica da matriz, ou ainda de um ponteiro para matriz. Assim, foram desenvolvidas 3 versões, detalhadas a seguir, nas quais diferentes formas de declaração da matriz `avector` foram utilizadas. O trecho de código mostrado abaixo mostra a primeira versão sequencial do programa que emprega uma matriz alocada dinamicamente, denominada de versão "F-Alloc":

```
! Parte 1: Inicializacao-------------------------------
double precision, dimension(:, :), allocatable  :: avector, aold
! Alocando a matriz:
allocate(avector(n, n))
allocate(aold(n, n))
avector=0.0
aold=0.0
!-------------------------------------------------------

do iters=1, niters

! Parte 2: Leitura de valor corrente--------------------
   do j=2, size-1
      do i=2, size-1
         avector(i, j) = aold(i, j)/2.0 + &
         (aold(i-1, j) + aold(i+1, j) +   &
         aold(i, j-1) + aold(i, j+1))/4.0/2.0
      enddo
   enddo
!-------------------------------------------------------

! Parte 3: Escrita de valores atualizados--------------
   aold=avector
!-------------------------------------------------------

enddo
```

Uma segunda versão sequencial do código foi criada com o uso de ponteiro Fortran 90, sendo denominada versão "F-Point":

```
! Parte 1: Inicializacao-------------------------------
double precision, dimension(:, :), pointer  :: avector, aold, atmp
! Alocando a matriz:
allocate(avector(n, n))
allocate(aold(n, n))
allocate(atmp(n, n))
avector=0.0
aold=0.0
atmp=0.0
!-------------------------------------------------------

```

```
do iters=1, niters

! Parte 2: Leitura de valor corrente-------------------
   do j=2, size-1
      do i=2, size-1
         avector(i, j) = aold(i, j)/2.0 + &
         (aold(i-1, j) + aold(i+1, j) +   &
         aold(i, j-1) + aold(i, j+1))/4.0/2.0
      enddo
   enddo
!-------------------------------------------------------

! Parte 3: Escrita de valores atualizados--------------
   atmp=>anew
   avector=>aold
   aold=>tmp
!-------------------------------------------------------

enddo
```

Finalmente, a terceira versão sequencial do código, que utiliza o conversor de ponteiros `c_f_pointer()` de C para Fortran 90, sendo denominada versão "F-Cptr":

```
! Parte 1: Inicializacao-------------------------------
USE, INTRINSIC                              :: ISO_C_BINDING
TYPE(C_PTR)                                 :: mem1
TYPE(MPI_Win)                               :: win1
integer, dimension(:), allocatable          :: memshape
double precision, dimension(:, :), pointer  :: avector, aold, atmp
double precision                            :: heat=0.0
integer                                     :: size, n
! Atribuindo o shape dos ponteiros Fortran:
allocate(memshape(2))
! Atribuindo o tamanho de cada dimensao dos ponteiros Fortran:
memshape(1)=n
memshape(2)=n
size=n*n*c_sizeof(heat)
! Alocando a janela SHM :
call MPI_Win_allocate_shared(size, 1, MPI_INFO_NULL, shmcomm, mem1, win1, ierror)
call MPI_Win_allocate_shared(size, 1, MPI_INFO_NULL, shmcomm, mem2, win2, ierror)
! Convertendo o endere o SHM de ponteiro C para Fortran:
call c_f_pointer(mem1, avector, memshape)
call c_f_pointer(mem2, aold, memshape)
allocate(atmp(n, n))
avector=0.0
aold=0.0
atmp=0.0
!-------------------------------------------------------

do iters=1, niters

! Parte 2: Leitura de valor corrente-------------------
   do j=2, size-1
      do i=2, size-1
```

```
32          avector(i, j) = aold(i, j)/2.0 + &
33          (aold(i-1, j) + aold(i+1, j) +   &
34          aold(i, j-1) + aold(i, j+1))/4.0/2.0
35        enddo
36     enddo
37 !-----------------------------------------------------
38
39 ! Parte 3: Escrita de valores atualizados--------------
40     atmp=>anew
41     avector=>aold
42     aold=>tmp
43 !-----------------------------------------------------
44
45 enddo
```

Essas 3 versões sequenciais, foram então executadas para 100 iterações simulando passos no tempo, sendo a matriz `avector` declarada com dimensão de 10.000 x 10.000 elementos. Cada versão foi temporizada de forma a se medir o tempo de alocação e inicialização dessa matriz, o tempo de acesso a matriz para leitura dos valores correntes e o tempo de acesso para escrita dos valores atualizados, a cada iteração correspondente a um passo de tempo. Esses tempos foram somados resultando nos tempos acumulados mostrados na Tabela 5.6, denominados respectivamente, de tempos de "inicialização", de "leitura de valor corrente"e de "escrita de valores atualizados"para as três versões F-Alloc, F-Point e F-Cptr.

Tabela 5.6 - Tempos acumulados de execução para 100 iterações no tempo (em segundos) das versões sequencias implementadas em Fortran 90, com a matriz `avector` de dimensão 10.000 x 10.000 alocada dinamicamente (F-Alloc), alocada com ponteiro Fortran 90 (F-Point) e alocada com uso do conversor `c_f_pointer()` (F-Cptr).

Versões	**F-Alloc**	**F-Point**	**F-Cptr**
Inicialização	3	3	2
Leitura de valor corrente	95	162	218
Escrita de valores atualizados	70	$\approx$0	$\approx$0
Total	168	165	220

A Tabela 5.6 mostra os tempos acumulados referentes à execução das 3 versões sequenciais (F-Alloc, F-Point e F-Cptr) compiladas com o compilador da PGI, opções *default*, exceto pela opção -O3. Nota-se que as versões que utilizam ponteiros demandam muito tempo na leitura da matriz, mas realizam sua escrita/atualização num tempo quase nulo, sendo que a versão F-Cptr gasta 35% a

mais de tempo que a versão F-point, provavelmente devido ao uso do conversor `c_f_pointer()`. Entretanto, comparando-se as versões F-Alloc e F-point, nota-se que a primeira é mais rápida na leitura da matriz, porém mais lenta na escrita/atualização, sendo que o total de tempos acumulados de ambas é praticamente igual. Essa comparação de tempos acumulados dessas 3 versões sequenciais foi também efetuada com os compiladores Intel, GNU e CCE (suíte de compiladores da Cray), com resultados similares. Uma vez que a comunicação MPI unilateral SHM em Fortran 90 exige o uso desse conversor de ponteiros, isso pode explicar seu pior desempenho paralelo, mas não permite ainda excluir outros fatores inerentes a essa forma de comunicação.

5.2.3 Análise do desempenho para execução do programa exemplo em vários nós

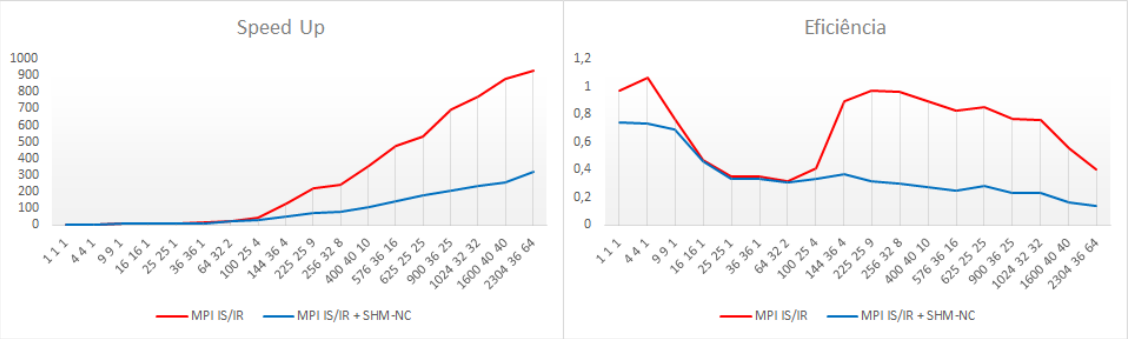
Os códigos referentes ao mesmo problema de diferenças finitas foram executados utilizando, para estes testes, vários nós computacionais da mesma máquina e do mesmo tipo que os empregados na Seção 5.2.1, com processadores Intel Xeon Broadwell de 2.2 GHz, sendo empregada a versão Fortran 90 S/R com comunicação convencional S/R e a versão Fortran 90 denominada híbrida, que combina as comunicações inter-nó S/R com a melhor das versões de comunicação intra-nó, com alocação de memória não contígua SHM-NC (MPI S/R + SHM-NC). O domínio do problema também foi de 4.800 x 4.800 pontos mas agora com 5.000 iterações no tempo, sendo utilizados até 64 nós computacionais com o número máximo de *cores* por nó (até 44), num total de 2.304 *cores* (na configuração de com 36 cores por nó, alocando 64 nós).

A Tabela 5.7 mostra os tempos de execução dessas duas versões aumentando-se o número de processos (N1), com o consequente número de processos por nó (N2) e de nós computacionais (N3) sendo esses tempos correspondentes à média de 5 execuções para cada caso. Nessa mesma tabela, aparecem os correspondentes *speedup's* e eficiências, calculados em relação à versão sequencial Fortran 90. Analogamente à Seção 5.2.1, foram usadas as opções *default* do compilador PGI, exceto pela opção *-O3*.

Tabela 5.7 - Tempos de execução (em segundos) e correspondentes *speedup's* e eficiências das versões **Fortran 90** paralelas MPI S/R e MPI S/R + SHM-NC compiladas com **PGI** e executadas com N1 processos, alocados em N2 *cores* de N3 nós computacionais para o domínio de 4.800 x 4.800 pontos com 5.000 passos de tempo. (para cada caso, o menor tempo aparece em negrito)

[N1-N2-N3]	**MPI S/R**			**MPI S/R + SHM-NC**		
Seq.= 333,59s	Tempo (s)	*Speedup*	Eficiência	Tempo (s)	*Speedup*	Eficiência
1-1-1	**343,84**	0,97	0,97	450,87	0,74	0,74
4-4-1	**78,36**	4,26	1,06	113,66	2,93	0,73
9-9-1	**48,07**	6,94	0,77	54,05	6,17	0,69
16-16-1	**44,36**	7,52	0,47	44,85	7,44	0,46
25-25-1	**38,56**	8,65	0,35	40,41	8,26	0,33
36-36-1	**26,82**	12,44	0,35	28,47	11,72	0,33
64-32-2	**15,26**	20,52	0,32	16,77	19,89	0,31
100-25-4	**8,17**	40.83	0,41	10,06	33,16	0,33
144-36-4	**2,61**	127,81	0,89	6,24	53,46	0,37
225-25-9	**1,53**	218,03	0,97	4,60	72,52	0,32
256-32-8	**1,36**	245,29	0,96	4,33	77,04	0,30
400-40-10	**0,94**	354,88	0,89	3,14	106,24	0,27
576-36-16	**0,70**	476,56	0,83	2,28	146,31	0,25
625-25-25	**0,63**	529,51	0,85	1,88	177,44	0,28
900-36-25	**0,48**	694,98	0,77	1,61	207,20	0,23
1024-32-32	**0,43**	775,79	0,76	1,43	233,28	0,23
1600-40-40	**0,38**	877,87	0,55	1,29	258,60	0,16
2304-36-64	**0,36**	926,64	0,40	1,05	317,70	0,14

Figura 5.4 - Desempenho paralelo das versões em linguagem **Fortran 90** MPI S/R e MPI S/R + SHM-NC.



Versão MPI S/R em vermelho e MPI S/R + SHM-NC em azul; compiladas com **PGI** e executadas com N1 processos, alocados em N2 nós com N3 *cores* por nós para o domínio de 4.800 x 4.800 pontos com 5.000 passos de tempo. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos.

Nota-se que, em todos os casos, a versão Fortran 90 MPI S/R foi mais rápida do que a Fortran 90 S/R + SHM-NC, demonstrando que esta última é penalizada seja pela conversão de ponteiros C para Fortran 90 seja pela saturação do uso da memória em cada nó, ou seja, a alocação da janela de memória SHM diminui o restante da memória compartilhada disponível aos processos, ocasionando uma contenção pelo seu acesso. A eficiência da versão S/R é alta para 4 e 9 processos, decai entre 16 e 100 processos para valores abaixo de 50% e sobe significativamente ao passar de 100 para 144 processos, provavelmente devido ao tamanho do cache L3 para o problema considerado. A eficiência permanece acima de 80% até 1024 processos, para decair para 1600 e 2304 processos, provavelmente devido à baixa granularidade decorrente para o problema considerado. Em comparação, a versão S/R + SHM-NC teve um desempenho pior para qualquer número de processos, especialmente a partir de 144 processos, em que suas eficiências foram menos que a metade daquelas da outra versão.

Observando os resultados de desempenho obtidos com outros compiladores: Intel, GNU e Cray (apresentados no Anexo A), nota-se um cenário semelhante ao apresentado nesta seção com compilador da PGI. Rodou-se para 1 nó computacional os códigos do problema exemplo compilados em Linguagem C e em Fortran 90 com outros compiladores disponíveis na máquina Cray, compilados apenas com a opção -O3".

Em todos os casos para Linguagem C, as versões MPI SHM-NC apresentaram tempos superiores à versão MPI S/R, exceto para os casos em que se utilizou 9 processos. Observando os resultados de desempenho dos códigos executados em Fortran 90, a versão MPI SHM-NC apresenta um resultado superior à versão MPI S/R apenas com o compilador da Cray com 16, 25 e 36 processos, mas em todos os outros casos persiste com resultados de desempenho inferior à sua versão concorrente bilateral.

5.3 Análise do desempenho do módulo da dinâmica isolada do BRAMS

Similarmente ao problema exemplo da seção anterior, implementou-se no módulo da dinâmica isolada do modelo BRAMS a comunicação híbrida, a qual combina a comunicação MPI unilateral *Shared Memory* entre processos executados no mesmo nó, com a comunicação MPI bilateral assíncrona e sem bloqueio, entre processos executados em nós computacionais diferentes. Assim, as execuções

do módulo da dinâmica isolada do BRAMS foram realizadas utilizando duas versões de códigos: a versão original com a comunicação paralela exclusivamente com MPI `MPI_Isend()` e `MPI_Irecv()` (IS/IR), e a versão híbrida com a comunicação paralela híbrida com MPI IS/IR + MPI SHM. No caso da comunicação unilateral SHM utilizou-se sempre a alocação de memória não-contígua (SHM-NC) para a janela de memória compartilhada, mais vantajosa que a alocação contígua, em função dos resultados obtidos para o problema exemplo da seção anterior.

Nesta comparação do desempenho paralelo entre versões da dinâmica isolada do BRAMS com comunicação bilateral assíncrona sem bloqueio e comunicação unilateral de memória compartilhada, para execução num único nó computacional, foram utilizadas duas configurações do BRAMS, sempre compiladas com PGI:

- Configuração operacional correspondente à versão operacional do CPTEC/INPE que utiliza 9.600 processos ou 400 nós computacionais do supercomputador Tupã, de forma a se reproduzir o subdomínio que é atribuído a cada nó do supercomputador Tupã num nó único da máquina Cray. Essa configuração corresponde a uma grade espacial horizontal de 111 x 111 pontos, cada ponto com 35 níveis na vertical, correspondendo a uma resolução espacial de 5 km e a *timesteps* de 8 segundos, gerando previsões para até 24 horas (essa limitação é específica do experimento).

- Configuração alternativa, mais leve computacionalmente que a operacional, com grade espacial horizontal de 100 x 100 pontos, cada ponto com 35 níveis na vertical, correspondendo a uma resolução espacial de 30 km e a *timesteps* de 30 segundos, gerando previsões de até 24 horas (essa limitação também é específica do experimento).

A configuração operacional foi utilizada num único nó computacional, para ambas versões do BRAMS (comunicação convencional e híbrida) de forma a poder se fazer a comparação entre ambas formas de comunicação MPI de maneira mais objetiva. Assim, tem-se unicamente comunicação MPI unilateral *Shared Memory* numa versão ou comunicação MPI bilateral assíncrona sem bloqueio na outra, entre os processos executados localmente nesse nó. Note-se que, no primeiro caso, embora a versão do BRAMS seja aquela de comunicação híbrida, utiliza-

se somente a comunicação unilateral *Shared Memory* por se tratar de um único nó computacional.

5.3.1 Análise do desempenho da execução do módulo da dinâmica isolada do BRAMS em um único nó

Comparam-se aqui as versões MPI da dinâmica isolada do BRAMS com comunicação unilateral *Shared Memory* e com comunicação bilateral assíncrona sem bloqueio para ambas as configurações do BRAMS (operacional e alternativa) na execução num único nó computacional da máquina Cray.

Os resultados do primeiro experimento aparecem na Tabela 5.8, que ilustra os tempos de execução das duas versões do BRAMS (com comunicação unilateral ou bilateral) na configuração operacional em função do número total de processos (N1), do número de processos alocados por nó (N2) e do número de nós computacionais utilizados (N3=1) executados num único nó computacional. Cada tempo representa a média de 5 execuções, e na mesma tabela encontram-se os valores dos desvios-padrão correspondente, *speedup's* e eficiência calculados para cada caso e os menores tempos estão destacados em negrito.

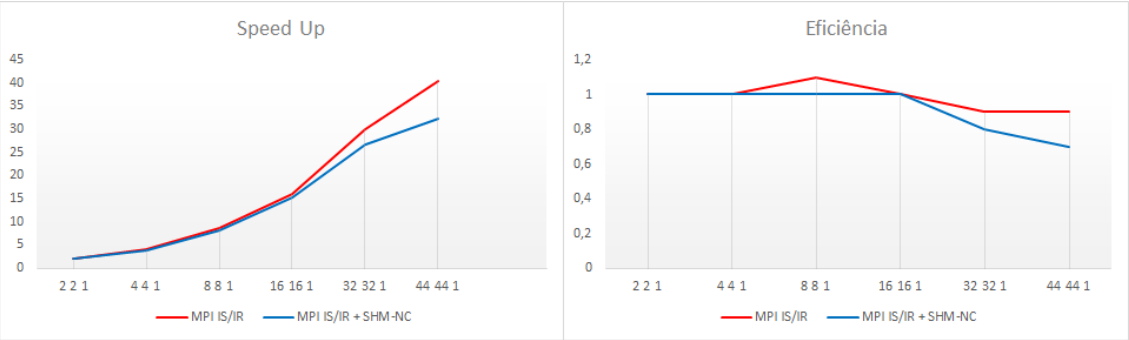
Tabela 5.8 - Tempos de execução (s) do BRAMS com a configuração operacional, desvios-padrão, *speedup's* e eficiências das versões MPI IS/IR original e híbrida (MPI IS/IR + SHM-NC), compiladas com PGI, executadas com N1 processos, utilizando N2 processos por nó e N3=1 nós computacionais, ou seja, um único nó (para cada caso, o menor tempo aparece em negrito).

[N1-N2-N3]	**MPI IS/IR**				**MPI IS/IR + SHM**			
Seq: 8.314s	Tempo (s)	Desv.	*Speedup*	Efic.	Tempo (s)	Desv.	*Speedup*	Efic.
2-2-1	**4.215**	1,9	2,0	1,0	4220	1,4	2,0	1,0
4-4-1	**2.055**	2,5	4,0	1,0	2.126	2,0	3,9	1,0
8-8-1	**954**	0,5	8,7	1,1	1.010	0,4	8,2	1,0
16-16-1	**517**	0,6	16,1	1,0	544	1,3	15,3	1,0
32-32-1	**277**	0,3	30,0	0,9	311	0,7	26,8	0,8
44-44-1	**206**	0,1	40,4	0,9	258	1,6	32,3	0,7

Nota-se que neste experimento, execução do BRAMS na configuração operacional num único nó, a versão convencional MPI IS/IR apresentou sempre melhor desempenho que a versão híbrida, sendo que esta corresponde neste caso (único nó) à versão com comunicação exclusivamente unilateral de memória compartilhada. A Figura 5.5 mostra os gráficos dos valores de *speedup* e eficiência da Tabela 5.8 de ambas as versões em função do número de processos.

Observa-se que as curvas são similares até 16 processos, sendo que para 32 e 44 processos, *speedup* e eficiência decaem para a versão híbrida (linhas em azul).

Figura 5.5 - Desempenho paralelo do BRAMS com configuração operacional.



Versão MPI IS/IR em vermelho e MPI IS/IR + SHM-NC em azul; compiladas com **PGI** e executadas num único nó computacional. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos. As triplas referem-se a (i) número de processos (N1), (ii) número de processos por nó (N2), que no caso são idênticos, e número de nós (N3=1).

No segundo experimento de desempenho paralelo, comparam-se as versões MPI da dinâmica isolada do BRAMS com comunicação unilateral *Shared Memory* e com comunicação bilateral assíncrona sem bloqueio utilizando-se a configuração alternativa do BRAMS para execução num único nó computacional da máquina Cray.

A Tabela 5.9 mostra os tempos de execução dessas versões do BRAMS na configuração operacional em função do número total de processos (N1), do número de processos alocados por nó (N2) e do número de nós computacionais utilizados (N3=1) executados num único nó computacional. Cada tempo representa a média de 5 execuções, e na mesma tabela encontram-se os valores dos desvios-padrão correspondentes, *speedup's* e eficiência calculados para cada caso e os menores tempos estão destacados em negrito.

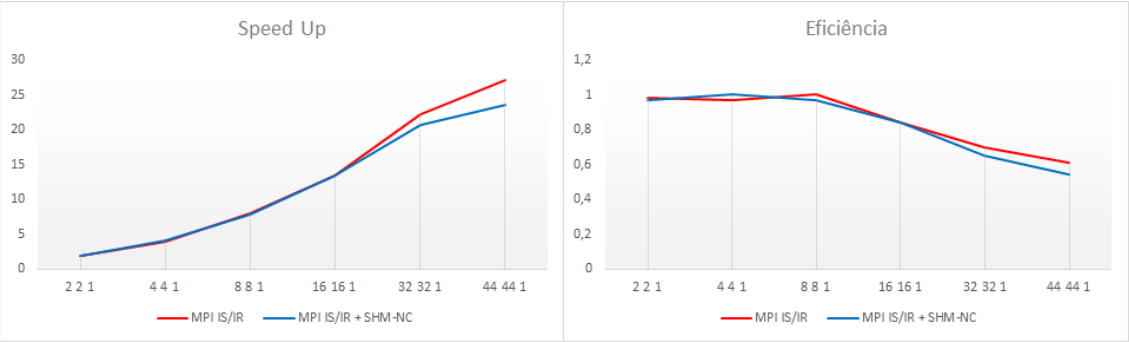
Pode-se observar, que utilizando-se um único nó para a configuração alternativa do BRAMS, o desempenho da versão MPI original com comunicação convencional é melhor, exceto para o caso com 4 processos, em que o tempo de execução

Tabela 5.9 - Tempos de execução (s) do BRAMS com a configuração alternativa, desvios-padrão, *speedup's* e eficiências das versões MPI original e híbrida (com SHM) executadas com N1 processos, utilizando N2 processos por nó e N3=1 nós computacionais, ou seja, um único nó (para cada caso, o menor tempo aparece em negrito).

[N1-N2-N3]	**MPI IS/IR**				**MPI IS/IR + SHM**			
	Tempo (s)	Desv.	*Speedup*	Efic.	Tempo (s)	Desv.	*Speedup*	Efic.
2-2-1	**459,46**	0,98	1,95	0,98	462,25	0,47	1,94	0,97
4-4-1	231,94	1,91	3,87	0,97	**223,89**	0,63	4,01	1,00
8-8-1	**112,11**	0,04	8,01	1,00	115,26	0,29	7,79	0,97
16-16-1	**67,13**	0,20	13,38	0,84	67,17	0,20	13,37	0,84
32-32-1	**40,27**	0,15	22,27	0,70	43,39	0,13	20,70	0,65
44-44-1	**33,15**	0,10	27,05	0,61	38,14	0,63	23,54	0,54

da versão híbrida (no caso, SHM) foi aproximadamente 3 % menor. A Figura 5.6 apresenta os gráficos de desempenho dos *speedup's* e eficiências informados na Tabela 5.9 onde é possível notar que as curvas de *speedup's* são semelhantes até 16 processos, havendo degradação das curvas da versão híbrida (SHM) para 32 e 44 processos (linhas em azul).

Figura 5.6 - Desempenho paralelo do BRAMS com configuração alternativa.



Versão MPI IS/IR em vermelho e MPI IS/IR + SHM em azul; compiladas com **PGI** e executadas num único nó computacional. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos, as triplas referem-se a (i) número de processos (N1), (ii) número de processos por nó (N2), que no caso são idênticos, e número de nós (N3=1).

Conclui-se que, para execução num único nó computacional, a versão do BRAMS com comunicação unilateral SHM-NC apresentou desempenho pior que a ver-

são original com comunicação MPI IS/IR, tanto para a configuração operacional quanto para a configuração alternativa. Vale lembrar que, embora tenha sido utilizada a versão híbrida desenvolvida neste trabalho, esta envolveu apenas a comunicação unilateral SHM-NC por se tratar de um único nó computacional.

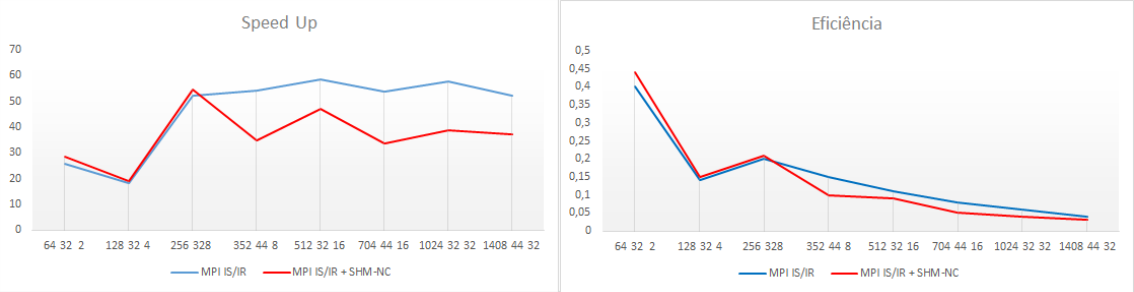
5.3.2 Análise do desempenho da execução do módulo da dinâmica isolada do BRAMS em vários nós

Comparam-se aqui as versões MPI da dinâmica isolada do BRAMS com comunicação híbrida (IS/IR + SHM-NC) e com comunicação bilateral assíncrona sem bloqueio (IS/IR) utilizando-se a configuração alternativa do BRAMS na execução com vários nós computacionais da máquina Cray. A Tabela 5.10 mostra os tempos de execução das duas versões do BRAMS com vários nós computacionais, onde N1 corresponde ao número total de processos utilizados, N2 é o número de processos alocados em cada nó e N3 é a quantidade de nós utilizados. Vale ressaltar que quanto mais processos intra-nó forem alocados mais comunicação SHM ocorre proporcionalmente, e quanto mais distribuídos dentre os nós estiverem os processos, isto é, menos processos por nó, mais comunicação IS/IR ocorre. Procurou-se executar as versões do BRAMS utilizando o máximo de processos no mesmo nó possível, para que dessa maneira, a comunicação SHM pudesse apresentar alguma vantagem sobre a comunicação IS/IR.

Tabela 5.10 - Tempos de execução (s) do BRAMS com a configuração alternativa, desvios-padrão, *speedup's* e eficiências das versões MPI original e híbrida (com SHM) executadas com N1 processos, utilizando N2 processos por nó e N3 nós computacionais (para cada caso, o menor tempo aparece em negrito).

[N1-N2-N3]	**MPI IS/IR**				**MPI IS/IR + SHM**			
	Tempo (s)	Desv.	*Speedup*	Efic.	Tempo (s)	Desv.	*Speedup*	Efic.
64-32-2	34,86	0,37	25,76	0,40	**31,61**	0,10	28,41	0,44
128-32-4	48,87	0,71	18,38	0,14	**46,93**	0,80	19,13	0,15
256-32-8	17,18	0,29	52,27	0,20	**16,41**	3,53	54,71	0,21
352-44-8	**16,62**	1,95	54,03	0,15	25,81	0,79	34,79	0,10
512-32-16	**15,36**	0,27	58,62	0,11	19,07	0,11	47,09	0,09
704-44-16	**16,64**	0,65	53,97	0,08	26,56	1,63	33,81	0,05
1024-32-32	**15,60**	1,25	57,57	0,06	23,17	3,76	38,77	0,04
1408-44-32	**17,23**	0,98	52,12	0,04	24,04	0,91	37,35	0,03

Figura 5.7 - Desempenho paralelo do BRAMS com configuração alternativa.



Versão MPI IS/IR em vermelho e MPI IS/IR + SHM-NC em azul compiladas com **PGI** e executadas com N1 processos, alocados em N2 *cores* por nó executados com N3 nós. Gráficos do *speedup* (à esquerda) e da eficiência (à direita) em função do número de processos. As triplas referem-se a (i) número de processos (N1), (ii) número de processos por nó (N2) e número de nós (N3).

Fonte: Gropp (2016).

De maneira geral, observando-se a mesma tabela, o desempenho paralelo da versão MPI híbrida foi pior do que a versão MPI original, mas podem-se observar casos, como os casos 64-32-2, 128-32-4 e 256-32-8, em que a versão híbrida foi mais rápida, sugerindo que a vantagem ou desvantagem da versão híbrida possa depender da granularidade do problema ou da arquitetura de memória específica da máquina. O desempenho paralelo referente a valores de *speedup* e eficiência aparecem na Figura 5.7, mostrando curvas similares para as duas versões até 256 processos (sempre com 32 processos por nó), mas com as curvas da versão híbrida piorando acima deste número de processos.

6 CONCLUSÕES

O objetivo deste trabalho foi avaliar o desempenho da comunicação unilateral MPI de memória compartilhada (SHM), proposta no padrão MPI 3.0 com o objetivo de otimizar a troca de mensagens entre processos executados num mesmo nó computacional de memória compartilhada. Neste trabalho, implementaram-se versões híbridas de programas paralelos, que utilizam a comunicação MPI unilateral SHM entre processos de um mesmo nó, e a comunicação MPI bilateral convencional assíncrona e sem bloqueio (IS/IR). Um programa considerado refere-se à implementação de problema exemplo relativo que utiliza cálculo de estências de 5 pontos num domínio bidimensional discretizado para resolução de equações diferenciais parciais pelo método das diferenças finitas. A implementação dessa nova forma de comunicação nesse problema exemplo permitiu sua implementação no módulo da dinâmica isolada do modelo BRAMS, que utiliza extensivamente diferenças finitas e cujo código é extremamente mais extenso e complexo, especialmente na implementação da comunicação MPI.

Essas versões, permitiram comparar o desempenho paralelo das versões híbridas (comunicação IS/IR + SHM) e convencionais (IS/IR) do programa exemplo e do módulo da dinâmica isolada do modelo BRAMS. As execuções paralelas, realizadas numa máquina paralela Cray com vários nós computacionais, demonstraram que o desempenho das versões originais IS/IR foi superior ao da versão híbrida. Foi possível constatar que a comunicação unilateral SHM teve desempenho ligeiramente melhor para programas na linguagem C do que para programas Fortran, mas ainda inferior à comunicação bilateral convencional IS/IR. Isso pode ser atribuído à implementação da comunicação MPI SHM ainda não estar suficientemente aperfeiçoada.

Aparentemente, a criação de uma janela de memória compartilhada comum aos processos locais gera uma contenção no acesso à memória que penaliza o desempenho paralelo mais do que as inúmeras trocas de mensagens na comunicação MPI bilateral convencional assíncrona e sem bloqueio. Deve-se levar em conta a especificidade do compilador, da máquina e de sua arquitetura de memória e processadores, além do sistema operacional Linux, que possivelmente favorecem a comunicação bilateral convencional.

A proposta da comunicação MPI unilateral de memória compartilhada foi prover uma otimização da comunicação intra-nó por meio do uso da memória compartilhada, a exemplo do padrão OpenMP. Haveria a vantagem de poder se portar

programas MPI para essa comunicação híbrida (IS/IR + SHM) visando explorar as vantagens da memória compartilhada sem o inconveniente de ter que reescrever códigos para torná-los *thread-safe*, que ocorre quando se porta o código para o padrão OpenMP. Isso seria muito interessante para o caso de códigos legados, tais como os de modelos numéricos de previsão de tempo, que são muito extensos, tipicamente com dezenas de milhares de linhas. Além disso, outra vantagem da comunicação MPI unilateral de memória compartilhada é permitir leituras e escritas diretas por parte do processo que as executa, sem necessidade de chamadas a funções MPI específicas, tornando a programação mais simples.

6.1 Trabalhos futuros

1. Utilizar ferramentas de *profiling* mais avançadas que permitam analisar melhor a influência do uso do `c_f_pointer()` na programação Fortran para comunicação MPI unilateral SHM.

2. Executar o módulo da dinâmica isolada do BRAMS para outras configurações (tamanhos de grade, etc.), outras máquinas paralelas, outros compiladores, de forma a verificar se as conclusões deste trabalho continuam válidas.

3. Analisar os poucos casos em que a comunicação híbrida teve desempenho paralelo melhor que a comunicação bilateral convencional IS/IR para identificar possíveis contextos favoráveis à comunicação híbrida.

4. Investigar se existe alguma opção de compilação, dentre os compiladores disponíveis, que possa minimizar o atraso da conversão das variáveis de `TYPE(C_PTR)` para *pointers* Fortran.

5. Comparar o desempenho das comunicações MPI avaliadas neste trabalho com a comunicação MPI unilateral por RMA.

REFERÊNCIAS BIBLIOGRÁFICAS

ALMEIDA, E. S. de; BAUER, M. Reducing time delays in computing numerical weather models at regional and local levels: a grid-based approach. **International Journal of Grid Computing & Applications**, v. 3, n. 4, p. 1, 2012. 30

BALAJI, P.; GROPP, W.; HOEFLER, T.; THAKUR, R. **Advanced MPI programming tutorial**. 2014. Disponível em: <http://www.mcs.anl.gov/~thakur/sc14-mpi-tutorial>. Acesso em: jan. 2017. 32, 33, 34, 64, 65

BONATTI, J. P. Modelo de circulação geral atmosférico do cptec. **Climanálise Especial (Edição Comemorativa de 10 anos)**, 1996. 1

BRAMS. **Brazilian developments on the Regional Atmospheric Modeling System (BRAMS)**. 2017. Disponível em: <http://brams.cptec.inpe.br>. Acesso em: jan. 2017. 1

CFL. **Courant–Friedrichs–Lewy (CFL) condition**. 2017. Disponível em: <https://en.wikipedia.org/wiki/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition>. Acesso em: jan. 2017. 29

CINDY; WEEKS. Concurrent extensions to the fortran language for parallel programming of computational fluid dynamics algorithms. In: . [S.l.: s.n.], 1986. 9

CRAY. **Tutorial about cray systems**. 2017. Disponível em: <https://partners.cray.com/restricted-swan/intro.html>. Acesso em: jan. 2017. xi, 62, 63

FINITE DIFFERENCE METHOD. **Finite Difference Method**. 2018. Disponível em: <https://en.wikipedia.org/wiki/Finite_difference_method>. Acesso em: maio de 2018. 64

FINITE VOLUME METHOD. **Finite Element Method**. 2018. Disponível em: <https://en.wikipedia.org/wiki/Finite_volume_method>. Acesso em: maio de 2018. 64

FREITAS, S. et al. The brazilian developments on the regional atmospheric modeling system (brams 5.2): an integrated environmental model tuned for tropical areas. **Geoscientific Model Development Discussion**, v. 10, 2016. 25, 68

GRADS. **Grid Analysis and Display System**. 2017. Disponível em: <http://cola.gmu.edu/grads/>. Acesso em: jan. 2017. 27

GROPP, W. **MPI + MPI: Using MPI-3 Shared Memory as a Multicore Programming System**. 2016. Disponível em: <http://www.caam.rice.edu/~mk51/presentations/SIAMPP2016_4.pdf>. Acesso em: jan. 2017. 9, 10, 21, 22, 82

HOEFLER, T.; DINAN, J.; BUNTINAS, D.; BALAJI, P.; BARRETT, B.; BRIGHTWELL, R.; GROPP, W.; KALE, V.; THAKUR, R. Mpi + mpi: a new hybrid approach to parallel programming with mpi plus shared memory. **Computing**, v. 95, n. 12, p. 1121–1136, 2013. 4, 5, 6

IEEE-754. **Instituto de Engenheiros Eletricistas e Eletrônicos**. 2017. Disponível em: <https://pt.wikipedia.org/wiki/Instituto_de_Engenheiros_Eletricistas_e_Eletr%C3%B4nicos#IEEE_754>. Acesso em: jan. 2017. 27

MENDES, C.; STEPHANY, S. **Tópicos especiais em computação aplicada: construção de aplicações massivamente paralelas**. 2016. Disponível em: <http://www.lac.inpe.br/~celso/cap387-2016/aulas.html>. Acesso em: jan. 2017. 16, 17, 18

MPI 1.0. **Message-Passing Interface Standard. Version 1.0.** 1994. Disponível em: <http://mpi-forum.org/docs/mpi-1.1/mpi1-report.pdf>. Acesso em: jan. 2017. 2, 11, 13

MPI 2.0. **Message-Passing Interface Standard. Version 2.0.** 1998. Disponível em: <http://mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf>. Acesso em: jan. 2017. 2, 11

MPI 3.1. **Message-Passing Interface Standard. Version 3.1.** 2015. Disponível em: <http://mpi-forum.org/>. Acesso em: jan. 2017. 2, 11

PANETTA, J. **Histórico de desenvolvimento do CCATT-BRAMS.** 2012. Disponível em: <https://projetos.cptec.inpe.br/attachments/237/JairoPanetta-1.pdf>. Acesso em: jan. 2017. 25

PGI. **Current PGI Compilers Documentation**. 2017. Disponível em: <http://www.pgroup.com/resources/docs.php>. Acesso em: jan. 2017. 63

SOUZA, C. R.; PANETTA, J.; STEPHANY, S. Desempenho da comunicação MPI shared memory no modelo meteorológico BRAMS. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP). **Anais...** São José dos Campos - SP., 2018. Disponível em: <https://eradsp2018.lsc.ic.unicamp.br>. 61

SOUZA, C. R.; STEPHANY, S.; PANETTA, J. Análise do desempenho de comunicação usando a funcionalidade de memória compartilhada do MPI 3.0. In: ENCONTRO NACIONAL DE MODELAGEM COMPUTACIONAL (ENMC). **Anais...** Nova Friburgo - RJ., 2017. Disponível em: <http://nbcgib.uesc.br/enmc2017>. 61

ANEXO A - COMPARAÇÃO DOS TEMPOS DE EXECUÇÃO DO PROBLEMA-EXEMPLO NUM ÚNICO NÓ COMPUTACIONAL PARA DIFERENTES COMPILADORES

A execução paralela do problema exemplo num único nó computacional para comparação de suas versões MPI com comunicação unilateral de memória compartilhada (SHM) e com comunicação bilateral convencional assíncrona e sem bloqueio (IS/IR) foi feita para ambas as versões compiladas com PGI e unicamente com a opção -O3, conforme mostrado na Tabela 5.5 da Seção 5.2.1.

Este apêndice objetiva reproduzir essa mesma comparação para as versões compiladas com outros compiladores (Intel, Gnu e Cray), todos também unicamente com a opção -O3. Entretanto, não houve mudanças significativas nessa comparação em relação àquela mostrada na Seção 5.2.1, ou seja, o uso de outros compiladores nesse problema não permitiu demonstrar um melhor desempenho da comunicação MPI unilateral SHM em relação à comunicação bilateral convencional assíncrona e sem bloqueio (IS/IR).

As Tabelas A.1, A.2 e A.3 deste anexo apresentam os tempos de execução, *speed ups* e eficiência do problema-exemplo utilizando os compiladores Intel, GNU e Cray com a opção de compilação *-O3*, nas versões em C.

As Tabelas A.4, A.5 e A.6 deste anexo apresentam os tempos de execução, *speed ups* e eficiência do problema-exemplo utilizando os compiladores Intel, GNU e Cray com a opção de compilação *-O3*, para as versões implementadas em Fortran.

Tabela A.1 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **linguagem C** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **Intel** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 34,75s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	**8,5**s	4,83s	**4,46**s	**3,91**s	**2,17**s
MPI SHM	10,05s	5,17s	4,62s	47,51s	20,15s
MPI SHM-NC	9,93s	**4,07**s	4,56s	4,14s	2,93s
Speed Up					
MPI IS/IR	4,1	7,2	7,8	8,9	12,8
MPI SHM	3,5	6,7	7,5	0,7	1,7
MPI SHM-NC	3,5	8,5	7,6	8,4	11,8
Eficiência					
MPI IS/IR	1,0	0,8	0,5	0,4	0,4
MPI SHM	0,9	0,7	0,5	0,0	0,0
MPI SHM-NC	0,9	0,9	0,5	0,3	0,3

Tabela A.2 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **linguagem C** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **GNU** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 34,20s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	8,37s	4,95s	**4,49**s	**3,92**s	**2,73**s
MPI SHM	**8,17**s	4,95s	4,59s	47,50s	21,35s
MPI SHM-NC	8,32s	**3,91**s	4,51s	3,99s	2,79s
Speed Up					
MPI IS/IR	4,1	6,9	7,6	8,7	12,5
MPI SHM	4,2	6,9	7,5	0,7	1,6
MPI SHM-NC	4,1	8,7	7,6	8,6	12,3
Eficiência					
MPI IS/IR	1,0	0,8	0,5	0,3	0,3
MPI SHM	1,0	0,8	0,5	0,0	0,0
MPI SHM-NC	1,0	1,0	0,5	0,3	0,3

Tabela A.3 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **linguagem C** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **Cray** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 35,63s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	**8,99**s	4.89s	4,53s	**3,94**s	**2,76**s
MPI SHM	9,14s	4,99s	4,63s	47,49s	27,30s
MPI SHM-NC	9,01s	**3,93**s	**4,53**s	4,11s	2,93s
Speed Up					
MPI IS/IR	4,0	7,3	7,9	9,1	12,9
MPI SHM	3,9	7,1	7,7	0,8	1,3
MPI SHM-NC	4,0	9,1	7,9	8,7	12,2
Eficiência					
MPI IS/IR	1,0	0,8	0,5	0,4	0,4
MPI SHM	1,0	0,8	0,5	0,0	0,0
MPI SHM-NC	1,0	1,0	0,5	0,3	0,3

Tabela A.4 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **Fortran** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **Intel** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 33,21s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	**8,11**s	**4,84**s	**4,43**s	**3,89**s	**2,69**s
MPI SHM	10,87s	5,35s	4,67s	47,59s	5,81s
MPI SHM-NC	10,93s	5,33s	4,56s	4,22s	2,95s
Speed Up					
MPI IS/IR	4,1	6,9	7,5	8,5	12,3
MPI SHM	3,1	6,2	7,1	0,7	5,7
MPI SHM-NC	3,0	6,2	7,3	7,9	11,2
Eficiência					
MPI IS/IR	1,0	0,8	0,5	0,3	0,3
MPI SHM	0,8	0,7	0,4	0,0	0,2
MPI SHM-NC	0,8	0,7	0,5	0,3	0,3

Tabela A.5 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **Fortran** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **GNU** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo (para cada caso, o menor tempo aparece em negrito).

SEQ= 33,65s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	**8,31**s	**4,86**s	**4,45**s	**3,87**s	**2,71**s
MPI SHM	8,71s	5,00s	4,62s	47,54s	5,76s
MPI SHM-NC	8,36s	4,96s	4,55s	4,04s	2,85s
Speed Up					
MPI IS/IR	4,0	6,9	7,6	8,7	12,4
MPI SHM	3,9	6,7	7,3	0,7	5,8
MPI SHM-NC	4,0	6,8	7,4	8,3	11,8
Eficiência					
MPI IS/IR	1,0	0,8	0,5	0,3	0,3
MPI SHM	1,0	0,7	0,5	0,0	0,2
MPI SHM-NC	1,0	0,8	0,5	0,3	0,3

Tabela A.6 - Tempos de execução em segundos, *speed ups* e eficiências para as versões paralelas em **Fortran** MPI IS/IR, MPI SHM e MPI SHM-NC compiladas com **Cray** e executadas com 4 (4P), 9 (9P), 16 (16P), 25 (25P) e 36 processos (36P) num único nó computacional para o domínio de 4800 x 4800 pontos com 500 passos de tempo. (para cada caso, o menor tempo aparece em negrito)

SEQ= 35,83s	**4P**	**9P**	**16P**	**25P**	**36P**
MPI IS/IR	**8,76**s	**5,58**s	5,36s	4,55s	3,25s
MPI SHM	12,62s	5,94s	4,73s	47,58s	5,83s
MPI SHM-NC	12,64s	5,93s	**4,63**s	**4,36**s	**3,14**s
Speed Up					
MPI IS/IR	4,1	6,4	6,7	7,9	11,0
MPI SHM	2,8	6,0	7,6	0,8	6,1
MPI SHM-NC	2,8	6,0	7,7	8,2	11,4
Eficiência					
MPI IS/IR	1,0	0,7	0,4	0,3	0,3
MPI SHM	0,7	0,7	0,5	0,0	0,2
MPI SHM-NC	0,7	0,7	0,5	0,3	0,3