Análise de benefícios do paralelismo por comunicação unilateral em aplicações com grades não estruturadas

Pedro Pais Lopes

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
Obtenção do título
de
Mestre

Programa: Ciência da Computação

Orientador: Prof. Dr. Siang Wun Song

São Paulo, 25 de junho de 2010

Análise de benefícios do paralelismo por comunicação unilateral em aplicações com grades não estruturadas

Esta versão definitiva de dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante defesa realizada
por Pedro Pais Lopes em 3/9/2010.

Comissão Julgadora:

- Prof. Dr. Siang Wun Song (orientador) - IME-USP
- Prof. Dr. Saulo Rabello Maciel de Barros - IME-USP
- Prof. Dr. Jairo Panetta - CPTEC-INPE

Agradecimentos

O autor desta dissertação gostaria de agradecer:

- primeiramente a sua esposa, América Murguía Espinosa, pelo calor e amor especial. Não há melhor sensação neste mundo que sentir-se apoiado a todo momento por alguém que nos ama, nos quer e nos dá energia para seguir adiante;
- aos seus familiares (geograficamente próximos ou distantes, do outro lado do mundo!), que dão a base para o crescimento pessoal, emocional, cultural e intelectual;
- ao orientador professor Siang Wun Song, que indicou diretrizes e sempre apresentou-se entusiasmado com o tema e o trabalho como um todo;
- ao grande mestre Jairo Panetta, que atuou como co-orientador (mesmo que não oficialmente, infelizmente), guia, amigo e chefe, apoiando financeiramente e intelectualmente para a realização (e finalização) deste trabalho;
- aos colegas e amigos do CPTEC/INPE, em especial a Álvaro Luis Fazenda e Eduardo Hidenori Enari, companheiros de diversas discussões acerca de aspectos intrigantes deste trabalho;
- aos amigos do laboratório MASTER do IAG/USP (Bruno, Enzo, Fabricio e Daniel);
- ao professor e amigo Pedro Leite da S. Dias e também a sua secretária Inês Massumi Iwashita (a "tia");
- ao professor Edmilson Dias de Freitas, por permitir a realização de testes em seus equipamentos;
- a Cray Inc., representado por Claude Paquette, por disponibilizar uma conta em um equipamento próprio localizado nos EUA. Sem esta conta boa parte dos testes de desempenho ficariam prejudicados;
- ao professor Robert Walko, autor do modelo OLAM, por permitir utilizar figuras, trechos do código do modelo e por ter sanado diversas dúvidas que surgiram ao se estudar o modelo;

E por fim a Deus, em todas as suas formas, cores, paisagens e sensações.

Resumo

A computação paralela, empregada no meio científico para resolução de problemas que demandam grande poder computacional, teve nos últimos anos o surgimento de um novo tipo de comunicação entre instâncias do paralelismo. Trata-se da Comunicação Unilateral (CUL), onde somente uma instância realiza a operação de transferência de informações, e esta ocorre em segundo plano, ao contrário da Comunicação Bilateral (CBL), onde uma instância envia a informação e a outra recebe. Neste contexto se buscou analisar os benefícios que a CUL agrega ao paralelismo de um programa que se utiliza de uma grade não estruturada em memória. Duas formas de apoio ao paralelismo foram utilizadas: uma biblioteca, a "*Message Passing Interface*" (MPI) (especificamente a sua parte que descreve a CUL), e uma extensão a linguagem Fortran, o *Coarray* Fortran (CAF). A semântica do MPI CUL é mais complexa que a do CAF, mas a do CAF é mais restritiva. Para analisar a semântica e desempenho da CUL foi realizada uma ambientação utilizando MPI CUL e CAF no paralelismo de um programa simples, denominado jogo da Vida (*Game of Life*), com grade estruturada e com ótimo desempenho paralelo através do MPI CBL. Na programação o MPI CUL se mostrou verborrágico (aumento do número de linhas de código) e complexo, principalmente quando se utiliza um controle refinado de sincronismo entre as imagens. Já o CAF reduziu o número de linhas de código (entre 20% e 40%), e o sincronismo é muito mais simples. Os resultados mostraram uma piora no desempenho no caso do MPI CUL, mas para o CAF o desempenho absoluto foi melhor que a implementação original até o número de núcleos de processamento que compartilham a mesma memória. Para grades não estruturadas se utilizou o "*Ocean Land Atmospheric Model*" (OLAM), um modelo de simulação do sistema terrestre com grade baseada em prismas triangulares, paralelizado através de MPI CBL. A implementação da comunicação por MPI CUL na estrutura do paralelismo existente mostrou que esta semântica possui alguns pontos que podem prejudicar a programação, como o tratamento da exposição de memória (cada instância tem uma memória exposta de tamanho diferente) e como é realizado o sincronismo entre as instâncias. Em termos de desempenho as curvas de *speed-ups* mostraram que o MPI CUL prejudicou o OLAM independentemente da implementação das bibliotecas ou do equipamento utilizado, com redução de pelo menos 20% no *speed-up* para sete ou mais processadores. Assim como no jogo da Vida o MPI com comunicação unilateral penalizou o desempenho.

Abstract

Parallel computing is used to solve many scientific problems that demand intensive computing power. Recently a new paradigm of communication between instances of the parallelism has appeared, called the one-sided communication (OSC), where only one instance performs the operation of information transfer, occurring in the background, as opposed to the two-sided communication (TSC), where one instance sends the information and the other receives it. In this context we analyze the benefits that OSC aggregates to the parallelism of a program that uses an unstructured grid in memory. Two OSC implementations were used: the "Message Passing Interface" (MPI) library (specifically the part that describes OSC), and Coarray Fortran (CAF), an extension of the Fortran language. The semantics of MPI OSC is more complex than that of CAF, but the semantics of CAF is more restrictive. To analyze the semantics and performance of OSC a simple program called Game of Life is used in a structured grid, giving very good parallel performance through MPI TSC. The MPI OSC program was verbose (increase in the number of lines of code) and complex, especially when using a more refined control to synchronize the parallel instances. On the other hand, CAF has reduced the number of lines of code (between 20% to 40%), and the synchronization is very simple. The results showed a worse performance in the case of MPI OSC, but for the CAF the absolute performance was better than the original implementation up to the number of processor cores that share the same memory. For unstructured grids we used the "Ocean Land Atmospheric Model" (OLAM), an earth simulation model on a grid based on triangular prisms, and parallelized with MPI TSC. The implementation with MPI OSC showed that this semantics has some points that may affect the coding of the communication structure, as in the treatment of memory exposure (each instance has an exposed memory of different size) and the way to treat the synchronization among instances. In terms of performance, the speedup curves showed that MPI OSC penalized OLAM, independently of the MPI implementation or the equipment used, with a reduction of at least 20% in speedup for seven or more processors. As in the Game of Life, MPI OSC degrades the performance.

Sumário

Nomenclaturas

Lista de Abreviaturas

BRAMS Brazilian Regional Atmospheric Model System

CAF Coarray Fortran

CBL Comunicação Bilateral

CCE Cray Compiling Environment

CFD Computational Fluid Dinamics

CLE Cray Linux Environment

CPTEC Centro de Previsão de Tempo e Estudos Climáticos

CPU Central Processing Unit

CUG Cray Users Group

CUL Comunicação Unilateral

E/S Entrada/Saida

EUA Estados Unidos da América

GCC GNU Compiler Collection

GHz Gigahertz

GNU GNU Not Unix

GPL GNU Public License

GZ Ghostzone

HDF5 Hierarchical Data Format (versão 5)

ICS Image Control Statements

INPE Instituto Nacional de Pesquisas Espaciais

ISO International Organization for Standardization

LAM Local Area Multicomputer

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MM5 Fifth-Generation NCAR / Penn State Mesoscale Model

MPI Message Passing Interface

MPMD Multiple Program Multiple Data

NCAR National Center for Atmospheric Research

NCARG NCAR Graphics

NML Namelist

OLAM Ocean Land Atmospheric Model

PC Personal Computer (Computador Pessoal)

PGI Portland Group Inc.

PICL Portable Instrumented Communication Library

PVM Parallel Virtual Machine

RDMA Remote Direct Memory Access

RMA Remote Memory Access

SE Segmento de Execução

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SPMD Single Program Multiple Data

UPC Unified Parallel C

Lista de Símbolos

ϵ Espaço de tempo pequeno

N	Número de imagens no paralelismo via CAF
p	Número de processadores ou instâncias
S	Speed-up
s	Segundos
T	Tempo de processamento

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Desde a invenção do computador busca-se a melhoria de seu desempenho através do desenvolvimento de equipamentos, peças e com o emprego de tecnologias mais avançadas. Esta busca é consolidada na clássica lei de Moore (Moore [1965]), afirmação feita em 1965 pelo fundador da Intel, Gordon E. Moore, de que o número de transistores de uma pastilha (*chip*) dobra a cada 18 meses. Como geralmente o aumento do número de transistores é utilizado para aumentar o desempenho de um processador esta lei mostra justamente que os computadores cresceram muito em termos de desempenho de processamento.

A lei de Moore em si continuará vigente, e por muitos anos: o número de transistores em uma CPU, ou mais genericamente, de um circuito integrado crescerá exponencialmente, segundo Kanellos [2005]. Novas tecnologias estão sendo desenvolvidas, a densidade de transistores tende a aumentar, novos elementos químicos estão sendo empregados e principalmente novas técnicas de fabricação de *chips* estão sendo utilizadas, a um custo competitivo, segundo o mesmo artigo. Mas a lei não está mais vigente quando consideramos a velocidade (que é aproximadamente proporcional ao número de ciclos de *clock* de uma CPU) das unidades de processamento dos computadores, expressa em *hertz*. Segundo Sutter [2005] é claro que deixamos de ter o aumento exponencial na velocidade: desde 2004 não encontramos uma CPU comum com velocidade maior que 3,4GHz.

O número de transistores cresceu e crescerá exponencialmente, mas com estagnação na velocidade de processamento. Para contornar a estagnação da velocidade de processamento, e continuar aproveitando do aumento do número de transistores, é necessário explorar o paralelismo que a construção de *chips* está utilizando. É o emprego de múltiplas unidades de processamento (comumente chamadas de núcleos) em um único *chip*.

O paralelismo é uma forma largamente utilizada para obtenção de alto desempenho em computação, principalmente na área de computação científica. Busca-se realizar a execução concomitante de duas ou mais operações, e assim realizar uma tarefa de forma mais rápida. Apesar do conceito ser simples inúmeros empecilhos existem para se conseguir o desempenho teórico, empecilhos estes que estão cristalizados na lei de Amdahl.

Além das questões inerentes ao *hardware* o programa precisa ser construído para permitir

a divisão de trabalho entre diversas unidades de processamento. A codificação de programas para execução em paralelo geralmente é mais complicada, e muitas vezes é necessária a comunicação entre processadores para atualização de memórias, regiões de sobreposição de domínio, etc.

Diversos esforços existiram no passado para dar suporte ao paralelismo. O principal, que atualmente é dito como o padrão *de facto* de programação paralela, é uma biblioteca, denominada *Message Passing Interface* (MPI). É constituída de diversas funções para realizar a transferência de dados entre duas instâncias de um programa, com interfaces para linguagens C, C++ e Fortran.

Na mesma linha existem também esforços para levar o paralelismo à linguagem de programação, buscando benefícios que uma biblioteca externa a linguagem não pode proporcionar. Podemos listar alguns exemplos, como o *Titanium* para o Java, *Unified Parallel* C (UPC) para o C++ e *Coarray* Fortran (CAF), para a linguagem Fortran.

Juntamente com os esforços, que geralmente damos conta de sua existência quando é lançada uma nova biblioteca ou versão de um compilador, há um grande trabalho para a padronização. O MPI foi fruto de diversas tentativas de padronização de bibliotecas anteriores a ele, e o CAF é considerado para inclusão no padrão Fortran 2008, assim como as outras formas de paralelismo nas linguagens de programação listadas. Este trabalho de padronização, além de definir uma sintaxe para uso das funções ou operadores também padronizam a semântica, que em um nível bem abstrato é a maneira com que uma operação é realizada.

Um caso onde o paralelismo é peça fundamental para a resolução de problemas computacionais complexos é na área de simulação da atmosfera terrestre, através de programas de dinâmica dos fluídos e parametrizações de processos físicos. Os programas que fazem esta simulação, comumente denominados de modelos atmosféricos, empregam técnicas de programação e utilizam bibliotecas de apoio ao paralelismo (principalmente a MPI) para usufruir, não somente da existência de múltiplos núcleos em um único processador, como também de equipamentos que empregam diversas máquinas interconectadas entre si. Estas aplicações também oferecem um desafio para medição da robustez da semântica e implementação das bibliotecas de paralelismo, pois a divisão das operações nem sempre é trivial.

1.1 Objetivos específicos e definição do problema

Realizar envios ou recebimentos de dados entre instâncias de um programa com execução paralela é o mínimo necessário para que uma biblioteca ou extensão da linguagem de programação consiga auxiliar no paralelismo, quando este estiver inserido em um contexto de programas independentes, com dados independentes. Uma grande quantidade de atividades, grades, *stencils* e operações em um programa podem ser realizadas em paralelo de diversas formas, mas todo momento que estas atividades precisarem ter conhecimento de outras

atividades que estão ocorrendo em outra instância do programa se faz necessário transferir dados.

Mais especificamente estes dados podem ser referentes a informações do programa, como, por exemplo, parte de um domínio geral que foi dividido. Durante a execução uma instância pode precisar de informações que ela não possui, e se faz necessário transferir dados de outra instância que possui as informações para ela. Mas a forma da transferência desses dados pode ser ou não ser interessante e o programa pode ou não providenciar todas as informações necessárias para chamar uma função ou invocar uma operação para realizar a transferência. A semântica tem papel importante neste caso. Buscar uma forma mais adequada de transferência de dados pode significar o aporte de benefícios que uma semântica revisada pode trazer.

Uma forma clássica de transferência de dados é baseada na paridade: uma instância do paralelismo envia, e outra recebe, com as duas partes ativamente participando (com chamadas de funções, por exemplo). Pode haver modificações como envio e recebimento assíncronos (em tempos distintos), com uso de meios intermediários, várias enviam para uma instância receber, mas a base é a mesma.

Porém outra forma foi apresentada pelos padrões de paralelismo, sejam da biblioteca MPI como das extensões às linguagens, que é baseado na unilateralidade: uma instância envia a informação para outra instância, mas somente uma delas (quem envia ou recebe) sabe da ocorrência da operação. A outra parte, a que desconhece a comunicação, tem sua memória, variáveis, entidades acessadas de forma passiva, tanto para leitura como para escrita.

O presente trabalho busca os benefícios que a semântica baseada na unilateralidade pode proporcionar. Estes benefícios poderão ser no esforço da programação, ou seja, se a semântica é mais indicada para o tipo de programa em questão, levando em consideração a facilidade de programação, a satisfação dos requisitos para a transferência dos dados, ou mesmo no esforço da escrita do código da transferência dos dados. Mas também poderão ser relativos ao desempenho, quando uma semântica diferente pode dar suporte a transferir dados em equipamentos diferenciados, tecnologias de acesso a memória, etc.

A busca, portanto, por benefícios, precisa centrar-se como ponto de partida no entendimento da semântica, tanto a bilateral como a unilateral. Os padrões que foram analisados são o do MPI, tanto nas suas funções de comunicação bilateral como na unilateral, e também o do *Coarray* Fortran.

O estudo dos padrões das bibliotecas e extensões às linguagens é uma boa maneira para entender sua semântica, e ater-se primeiramente a sua padronização é uma boa forma de não tornar-se tendencioso. O padrão da biblioteca MPI possui pelo menos 16 anos de existência (primeira versão oficial data de 5 de maio de 1994) e se tornou o padrão *de facto* de programação paralela (ao menos na área científica e de computação de alto desempenho). Possui uma forma largamente utilizada de transferência de dados baseada na paridade, e em 1998 uma nova maneira, uma nova semântica foi adicionada ao padrão, porém baseada

na unilateralidade. Já o *Coarray* Fortran se mostra como uma extensão muito simples ao Fortran trazendo os paradigmas de endereçamento global, e em algumas plataformas seu desempenho relativo é significativo, tornando-se uma alternativa ao MPI.

Mas também é necessário verificar sua aplicabilidade em programas que as utilizam, e a forma encontrada foi de analisar a sua usabilidade em um programa simples. Desta forma podemos abstrair informações como facilidade no uso, detalhes da sintaxe, garantias de sincronismo e satisfação das informações para a transferência dos dados.

Partimos com o uso das rotinas de comunicação unilateral em um programa simples chamado "Jogo da Vida". Foram analisados os dois aspectos listados anteriormente: o aporte de benefícios que a semântica unilateral do MPI e do CAF trouxeram ao programa e a mudança de desempenho adquirida. Este programa simples serviu para ambientação com as bibliotecas e funções, e devido a este utilizar uma grade estruturada em memória também serviu como ponto de partida para verificar se esta "estruturação" é bem aceita pelas semânticas (se estas possuem funções e estruturas para auxiliar na codificação da comunicação e divisão de trabalhos).

Após a implementação de comunicação unilateral em um programa simples expandimos a análise dos benefícios para um programa mais complexo. Este programa é um modelo global de simulação do sistema terrestre denominado *Ocean Land Atmospheric Model* (OLAM). O OLAM é um programa que discretiza o globo terrestre em volumes finitos, no formato de prismas com base triangular, e sua distribuição na memória de um computador é realizada de forma não-estruturada. Ele já possui paralelismo baseado na comunicação bilateral com a biblioteca MPI, através da divisão da grade.

O intuito foi de aplicar a nova semântica, a unilateral, em suas rotinas de transferência de informações entre instâncias do paralelismo. Foram analisados os dois aspectos relacionados anteriormente, aquele que trata da parte semântica (se traz benefícios para a programação) e da parte do desempenho computacional (se explora o *hardware* e os sistemas envolvidos no processamento).

Ao final foi possível comparar as impressões em programar o paralelismo em dois tipos de grades (estruturadas e não-estruturadas), como também verificar se a semântica de comunicação unilateral trouxe benefícios de desempenho nas duas grades.

1.2 Estrutura desta dissertação

O presente trabalho está estruturado da seguinte maneira. No capítulo 2 temos uma descrição simplificada do padrão MPI, tanto de suas funções de comunicação bilateral (comumente denominada de MPI versão 1) como unilateral (comumente denominada MPI versão 2), ao final do mesmo são apresentadas observações comparando as duas semânticas.

A seguir é colocado, no capítulo 3, o padrão *Coarray* Fortran, com ênfase em sintaxe e operações de sincronismo. Ao final do mesmo são apresentadas observações relativas a este

padrão e sua aplicabilidade.

O capítulo 4 descreve o jogo da Vida, o programa simples citado anteriormente. Após a sua descrição temos a implementação do mesmo tanto em MPI com comunicação bilateral, unilateral e com uso de *Coarray* Fortran. Temos, após a implementação, os resultados das execuções destas implementações em comparação à implementação através da comunicação bilateral. Ao final são colocadas impressões relativas à usabilidade e programabilidade das funções do MPI e do CAF, bem como conclusões relativas ao desempenho.

Já o capítulo 5 descreve o modelo OLAM, principalmente suas rotinas de divisão de domínio e envio e recebimento de informações estre as instâncias. É colocada a modificação do código para uso das rotinas de comunicação unilateral do MPI. Posteriormente temos os resultados das execuções do OLAM com comunicação unilateral, e comparação com o desempenho apresentado com implementação original. Ao final são mostradas observações do uso destas rotinas no código, com comparação com o que foi constatado no jogo da Vida.

No capítulo 6 são colocadas as conclusões gerais, com os pontos mais pertinentes das observações levantadas.

1.3 Revisão bibliográfica

Em computação o paralelismo pode ser encontrado em diversos estudos. Flynn [1972] propôs quatro tipos de fluxos de dados em arquiteturas de computadores: SISD (*Single Instruction Single Data*); SIMD (*Single Instruction Multiple Data*); MISD e MIMD (Multiple *Instruction* Multiple *Data*).

Esta taxonomia teve, nos últimos anos, uma ramificação para tratar alguns novos casos, entre eles o SPMD (*Single Program Multiple Data*) e o MPMD (*Multiple Program Multiple Data*). Conforme Darema [2001] foi proposto o modelo SPMD, onde um único programa é utilizado para processar vários conjuntos de dados. Cada programa está "encapsulado" em processos, um para cada conjunto de dados, e estes processos, em conjunto, processam em paralelo. O segundo novo caso é o MPMD, onde cada conjunto de dados pode ser processado por um programa distinto, com a divisão de programas para processar dados independentes.

Uma possível mudança de abstração da categoria SPMD nos permite chegar a um tipo específico de paralelismo, denominado "Paralelismo de tarefas" (do inglês "*Task Parallelism*"). Podemos observar um "fluxo de instruções" como o processamento de uma CPU, e de forma mais abstrata ainda um "processo" em um sistema operacional, e o "conjunto de dados" como variáveis de um programa, série de funções a serem executadas, área de memória virtual alocada, etc.

A base deste tipo de paralelismo é de dividir uma tarefa em diversas tarefas menores, que podem ser executadas de maneiras e em momentos diferentes, possuindo uma execução relativamente "solta" (comparado ao SIMD, onde um fluxo de instruções só entra em cena quando o anterior terminar, causando um sincronismo muito "rígido"). Estas tarefas são

então distribuídas para serem executadas por CPUs distintas, e ao final algum mecanismo une os resultados destas tarefas a fim de gerar o resultado final. Esta divisão costuma ser explícita, isto é, o programa tem conhecimento de como é dividido o domínio/dados, e como cada um destes domínios serão processados (quais funções ou operações serão executadas).

Diversas bibliotecas de apoio ao paralelismo foram criadas com intuito de auxiliar a programação paralela. Além disso, com a criação destas bibliotecas, ficam padronizados ao programador funções de comunicação e sincronismo de processos, o que ajuda muito na portabilidade.

Segundo Meglicki [2004] ao redor dos anos 80 o surgimento de algumas máquinas com muitos processadores (como o N-cube) impulsionou o desenvolvimento de bibliotecas de apoio ao paralelismo. Alguns esforços foram específicos para arquiteturas proprietárias (como a do próprio N-cube), e outros foram para redes de estações de trabalho UNIX, como a biblioteca P4 de *Argonne National Laboratory* (Butler e Lusk [1994]) e sua interface Fortran PARMACS da *Gesellschaft für und Datenverarbeitung mbH* e descrita em Hempel [1991], PICL () descrita em Geist *et al.* [1990] e PVM (*Parallel Virtual Machine*) de *Oak Ridge National Laboratory*, descrita em Sunderam *et al.* [1994], e a biblioteca de passagem de mensagens LAM (*Local Area Multicomputer*) de *Ohio Supercomputer Center*, descrita em Gaglianello *et al.* [1989].

Porém como havia muitos esforços com um mesmo objetivo ficou claro que a união de trabalhos era iminente (Meglicki [2004]). Foi então que em 1992 a idéia do padrão MPI (*Message Passing Interface*) foi criada. Constituída por algumas empresas, centros de pesquisa e universidades dos EUA e Europa o "Forum MPI" tem como objetivo definir, corrigir e manter o padrão MPI. O padrão MPI versão 1.0 (MPI-Forum [1994]) surgiu em 1994, e o padrão MPI versão 2.0 (MPI-Forum [1998]) em 1998.

Desde então o padrão MPI versão 1 teve grande aceitação, pelo menos na área científica e suas aplicações que demandam grande poder de processamento paralelo. Ele não é simples de usar, mas não é seu objetivo ser simples. Seu objetivo é fornecer um modelo completo de programação paralela, e, segundo um dos principais autores da biblioteca MPICH, W. Gropp[1]:

[1] Gropp [2001], pág. 87 (tradução livre).

> "Outra maneira de olhar para isso (codificações paralelas cada vez mais complexas) é que enquanto muitos programas talvez não sejam simples com o uso do MPI, nenhum programa é impossível (de ser programado com o uso desta biblioteca). MPI é às vezes chamado de "linguagem *assembly*" da programação paralela. Aqueles que fazem esta afirmação esquecem que C e Fortran também foram descritos como linguagens *assembly* portáveis."

É possível observar o sucesso do MPI nos livros que foram publicados a seu respeito. Existem vários, mas foram escolhidos dois por se mostrarem próximos do padrão e possuírem as chamadas das funções nas três linguagens (C, C++ e Fortran): O *Using MPI*, de Gropp *et al.* [1999a] e o *Using MPI-2*, de Gropp *et al.* [1999b]. São imparciais em termos de implementação, não mostrando tendência para detalhes e melhorias das mesmas. O último possui alguma informação quanto a aplicabilidade da semântica do MPI versão 2 *versus* a semântica da versão 1.

As implementações de MPI existentes são inúmeras. Cada fabricante de computadores de alto desempenho possui implementação desejada para o seu equipamento, como é o caso da Cray, SGI, IBM, NEC, etc. Devido ao presente trabalho não desejar ater-se a alguma plataforma específica se deu preferência para as implementações portáveis. Uma discussão mais específica sobre as implementações MPI pode ser encontrada no capítulo 2.

Além destas bibliotecas vários esforços são encontrados em criar extensões a linguagens de programação como forma de levar o paralelismo a um nível mais próximo do código. As extensões são mudanças na sintaxe da linguagem (um exemplo é inclusão de colchetes, como no caso do *Coarray* Fortran), dialetos (como em Java), módulos (como no Co-array Python), entre outros.

O *Coarray* Fortran, um objeto de nosso estudo neste trabalho, foi descrito primeiramente como F^{--} em Numrich *et al.* [1998], e que posteriormente foi colocado em discussão no comitê gestor do padrão Fortran pelo artigo Numrich e Reid [1998]. Diversas revisões se passaram até a revisão mais atual (Reid [2009]). Está em constante discussão para sua inclusão no padrão Fortran, mas a tendência é mantê-lo fora, como uma extensão à linguagem, com a credibilidade da padronização do mesmo comitê da ISO.

Um problema que o CAF apresenta é a falta de compiladores que tenham suporte a ele. O mais usado e que apresenta melhores resultados é parte do *Cray Compiling Environment* (CCE), da Cray Inc. Existem compiladores para arquiteturas e sistemas operacionais não proprietários, e uma descrição sobre estes compiladores pode ser encontrada no item 3.2.

Diversos trabalhos podem ser encontrados descrevendo a migração de programas paralelos baseados em MPI para CAF, porém todos utilizando processamento vetorial e sistemas

proprietários. O trabalho de Dawson [2004], apesar de estar em um periódico pouco expressivo é bastante interessante e compara duas implementações de troca de bordas de um programa com uma grade estruturada, uma com a biblioteca MPI e outra com CAF. São analisados pontos como *deadlock*, sintaxe e semântica, divisão e localização das partições da grade, a técnica de sobreposição destas partições para minimizar a necessidade de comunicação e são comparados entes pontos entre MPI e CAF. É próximo ao assunto deste trabalho, porém não trata de uma grade complicada, não estruturada e a versão do MPI é a 1, enquanto o presente trabalho busca comparar a versão 2, baseada na unilateralidade, assim como CAF.

O trabalho de Barrett [2006] ilustra as experiências de se programar em CAF. É próximo da realidade dos programas das ciências da terra, pois trabalha com métodos de diferenças finitas (trabalha com um *stencil* sobre uma matriz, da mesma classe de problemas do trabalho analisado no parágrafo anterior). Inicia sua análise com a descrição do paralelismo de seu problema com MPI, com particionamento da grade (não divergindo do trabalho de Dawson). Apresenta então três formas diferentes de transferência das bordas: uma com total integração da notação do CAF; outra com a troca de bordas através da notação CAF; uma terceira com uso do CAF somente quando o dado envolvido no cálculo não estiver na memória local. As questões específicas das três implementações são melhor explicadas no artigo.

Outro artigo pertinente é Ashby e Reid [2008], relatando as experiências da migração de uma aplicação de *Computational Fluid Dinamics* (CFD) de MPI para CAF. Da mesma forma que os outros dois anteriores este artigo chega a conclusão de que o código com CAF ficou mais claro, com redução de linhas de código de 50%, mas utilizou MPI versão 1 (funções com paridade). Uma observação importante: este trabalho coloca o tempo de processamento da aplicação com *coarrays*, e este é menor que o tempo de execução com MPI, mostrando então que *coarrays* é competitivo em termos de desempenho, ao menos na arquitetura utilizada (Cray X1E).

Estes trabalhos dão embasamento ao poder do CAF, pois mostram que o uso de *coarrays* facilita na programação, com redução do esforço, sem perda de desempenho. E também são de uma área correlata ao presente trabalho. Todavia foram realizados em uma plataforma fechada e específica (processadores vetoriais e rede de comunicação dedicada). São trabalhos apresentados em conferências do *Cray Users Group (*CUG), grupo de usuários de equipamentos da Cray, o único compilador com bom desempenho conhecido. O ambiente utilizado está descrito no item 3.2.

Capítulo 2

A biblioteca *Message Passing Interface*

Neste capítulo vamos descrever o padrão *Message Passing Interface* (MPI). As aplicações no jogo da Vida e no OLAM estão descritas nos capítulos 4 e 5, respectivamente.

O padrão MPI é constituído de diversas rotinas que descrevem operações como definição de tipos de dados, sincronismo, criação e destruição do ambiente paralelo e operações de troca de mensagens. Ele é um padrão para comunicação de dados em computação paralela, que, em termos de programação, pode ser denominado como uma biblioteca, pois providencia ao programador uma coleção de funções ou métodos para auxiliar e permitir a computação paralela.

Ao utilizar o MPI um programa conta com informações para possibilitar a divisão de tarefas entre diversas instâncias. Todo o paralelismo ocorre em um ambiente paralelo que é criado, por funções da biblioteca MPI, geralmente no início do programa. Ao final da computação o mesmo ambiente deve ser destruído.

A mais básica das informações é um número que o processo possui dentro da computação paralela, comumente denominado de *rank*. Também é denominado comumente de "processador", e para o sistema operacional é um "processo"[1].

De maneira geral o *rank* varia entre zero e o número total de instâncias do programa menos 1, número este especificado no momento da execução do programa. Com *ranks* diferentes para instâncias diferentes o programador tem a possibilidade de dividir tarefas de maneira direta, como descrito no pseudocódigo abaixo:

```
se rank = 4
    (executa tarefas do rank 4)
se rank = 7
    (executa tarefas do rank 7)
senão
    (executa tarefas de outro rank)
```

[1]Neste texto, para simplificar, usaremos a palavra *rank* para denotar "processo identificado por *rank*" ou "processador identificado por *rank*".

Os *ranks* podem ser distribuídos em grupos, e estes grupos distribuídos em comunicadores[2]. O padrão MPI fornece operações para gerenciar e dar mais robustez ao ambiente paralelo, propiciando maior flexibilidade.

Porém, ainda seguindo o mesmo exemplo, pode haver a necessidade de que o *rank* 7 precise de informações que estão no *rank* 4. O mesmo MPI possui operações para envio e recebimento de informações entre *ranks*. Este tipo de operações – envio e recebimento de dados entre *ranks* – é a classe de operações MPI que será analisada no presente trabalho.

Existem basicamente duas formas de tratamento da transferência da informação em MPI. A primeira (em termos históricos) é a que descreve o paradigma denominado "Comunicação Bilateral", ou CBL. Nesta forma os envolvidos na comunicação invocam dois tipos de operações. O *rank* que envia a informação invoca a operação de envio, com as informações do envio pertinentes (posição de memória do dado a ser enviado, tamanho, tipo e *rank* de destino). Já aquele que recebe a informação invoca a operação de recebimento, com informações de recebimento pertinentes (*rank* de origem, tamanho do dado, tipo e posição de memória a ser gravado o dado).

A outra forma de transferência descreve o paradigma denominado "Comunicação Unilateral", ou CUL. Diferentemente da CBL somente uma das partes da comunicação invoca a operação. Todas as informações pertinentes à transferência do dado são fornecidas à operação de CUL, portanto posição de memória de origem, tamanho, tipo, *rank* remoto, posição de memória de alvo, tamanho da memória do alvo e tipo do dado no alvo são conhecidos no momento da operação da transferência.

Historicamente as operações de transferências bilaterais, por terem sido definidas no padrão MPI versão 1, são chamadas de "MPI1". Dizer "este programa utiliza comunicação via MPI1" é próximo a dizer que o programa utiliza a CBL. Já a CUL foi definida na segunda versão do padrão MPI, e dizer "o programa utiliza comunicação MPI2" é próximo a dizer que o programa utiliza a CUL. Mas não é possível dividir a CBL e CUL em dois padrões diferentes, pois elas fazem parte de um padrão único, denominado MPI, que atualmente está a versão 2.2 (lançado em 4 de setembro de 2009).

Em termos de implementação a mais clássica e difundida é a MPICH1[3], com implementação do padrão MPI versão 1, e que atualmente está "congelada". Sua descrição está no artigo Gropp *et al.* [1996]. Já para a versão 2, que também implementa as funções da versão 1, é a MPICH2[4]. Outra implementação portável e bastante completa é a OpenMPI[5].

A seguir são discutidas mais a fundo as semânticas da CBL e CUL, com posterior colocação de observações comparando a CUL frente a CBL.

[2]Com a criação de grupos ou comunicadores os *ranks* poderão estar distribuidos de outra maneira, além daquela distribuição linear de *ranks* colocada anteriormente.
[3]Site: www.mcs.anl.gov/research/projects/mpi/mpich1/
[4]Site: www.mcs.anl.gov/research/projects/mpich2/
[5]Site: www.open-mpi.org/

2.1 CBL: Comunicação Bilateral

Este tipo de operação é a responsável pelo envio da informação de um processo para ser recebido por outro, com a particularidade de ser realizado de forma explícita. Uma instância envia o dado, a outra recebe, e cada uma efetua chamadas a funções respectivas para envio e recebimento. O padrão fornece diversas formas de transferência, mas sempre baseado na mesma semântica.

A análise das operações de envio e recebimento de informações é baseada nas garantias que são asseguradas pelo retorno das mesmas. Este tipo de análise é importante para se ter conhecimento de quatro características:

1. se o conteúdo da variável a ser transferida pode ser alterado, após o envio, no *rank* de envio;

2. se o conteúdo da variável a ser recebida pode ser lido, após o recebimento, no *rank* de recebimento;

3. se o retorno da função de envio assegura que a função de recebimento, se existir, já terminou;

4. se o retorno da função de recebimento assegura que a função de envio remoto, se existir, já terminou.

Estes quatro pontos descrevem o sincronismo entre duas instâncias, ou, em nível mais alto, se é possível controlar a ordenação do ponto do processamento entre duas instâncias.

2.1.1 Comunicação bloqueante

Neste item será analisada uma classe de funções que realizam a transferência de dados entre *ranks* de forma bloqueante. O termo bloqueante se refere ao fato de as funções bloquearem a execução da instância que a invoca para assegurar que a operação termine com sucesso, não permitindo, portanto, sobreposição de comunicação e computação.

A maneira genérica para enviar um dado de um processador para outro é através da função MPI_Send. Ela recebe como argumentos principais a posição de memória local do dado a ser enviado, seu tamanho, seu tipo e o *rank* de destino. Em contrapartida, a função utilizada para recebimento no *rank* remoto é MPI_Recv. Recebe como argumentos a posição de memória local onde o dado deve ser armazenado, o tamanho do mesmo, o tipo e o *rank* que envia o dado. A figura 2.1 ilustra o envio de uma variável, em Fortran do tipo *integer*, denominada I.

O exemplo da figura possui uma ordenação inerente entre *ranks* envolvidos, pois as operações só retornam quando ocorre o envio ou o recebimento, e são comumente denominadas de bloqueantes. Ao analisar as características listadas acima, temos que:

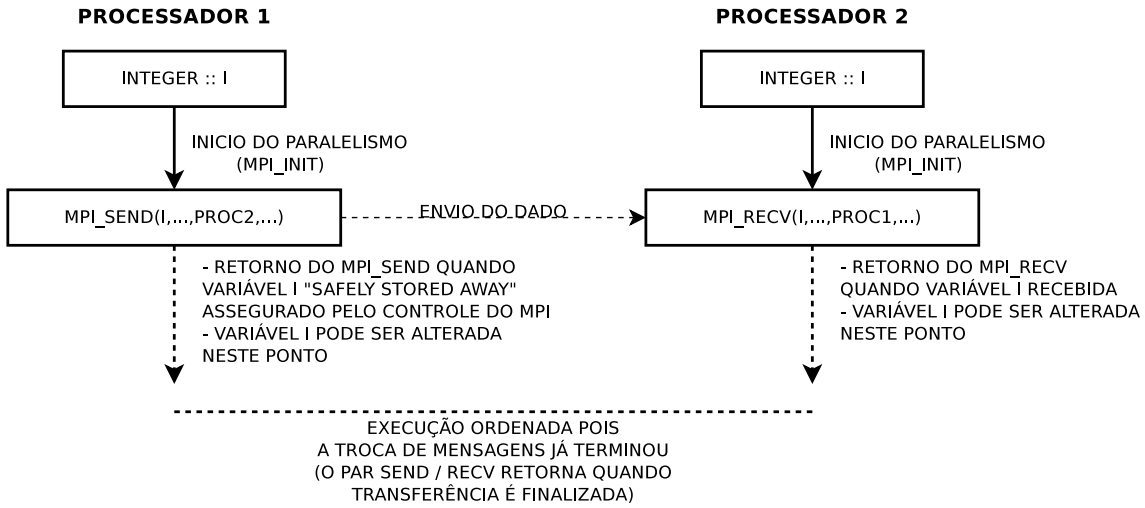

Figura 2.1: Diagrama indicativo do envio de uma variável, tipo INTERGER, "I" do processador 1 ao processador 2, com uso de funções bloqueantes.

1. no *rank* de envio a variável I pode ser alterada após retorno da função de envio;

2. no *rank* de recebimento a variável I pode ser lida após retorno da função de recebimento;

3. A operação MPI_Send retorna quando ocorre o envio, mas não é assegurado que ocorreu o recebimento[6];

4. A operação MPI_Recv retorna quando ocorre o recebimento e, devido a ordem natural, assegura também que ocorreu o envio (mas não o seu término).

As garantias dos retornos das funções não modificam a ordenação do processamento entre as duas instâncias. Apesar de, "no relógio", não ser possível inferir se as duas instâncias estão realizando o mesmo processamento, é assegurado que I foi enviado com segurança, e que ele foi recebido também com segurança. Isso é suficiente para que o processador 1 continue seu processamento podendo alterar I, e o 2 continue seu processamento podendo ler I, atualizado.

Existem outras funções de envio e recebimento, possibilitando alteração do comportamento das garantias do retorno das funções, sincronização e conhecimento, da instância do envio, de que a instância do recebimento já terminou o recebimento. Mas as operações acima descritas são genéricas, permitindo que a implementação escolha, em tempo de execução, se a transferência gera sincronismo ou não. Mas elas garantem que a transferência ocorra com sucesso no retorno da função.

[6] A informação que será enviada está, como descrito no padrão, *safely stored away*. Isso assegura que o processo que envia pode acessar de forma segura a memória do que foi enviado, pois o dado já está guardado em lugar seguro.

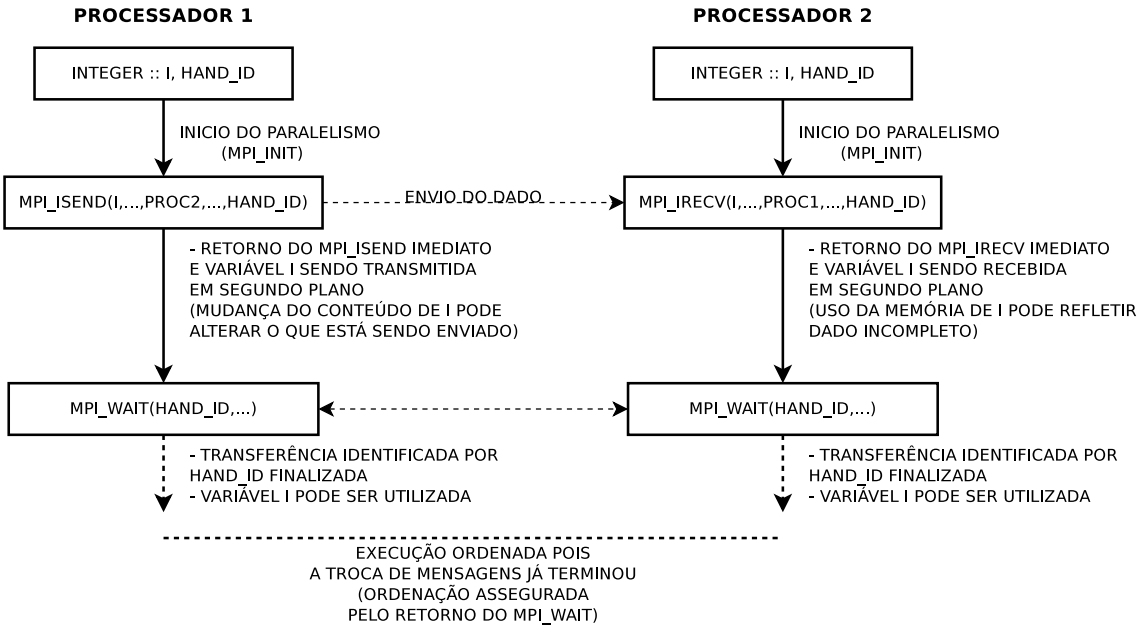

Figura 2.2: Diagrama indicativo do envio de uma variável, tipo *integer*, I do processador 1 ao processador 2, com uso de funções de não-bloqueantes.

2.1.2 Comunicação não-bloqueante

A outra classe de operações de envio e recebimento de informações é a não-bloqueante, diferentemente da anteriormente descrita, que é bloqueante, e retorna imediatamente sem assegurar que a operação respectiva – envio ou recebimento – foi terminada com sucesso. São as operações MPI_Isend e MPI_Irecv. A figura 2.2 ilustra, de forma similar a figura 2.1, como é o envio da variável I do processador 1 ao 2.

As quatro características para estas funções são listadas a seguir:

1. no *rank* de envio a variável I não pode ser alterada após o retorno da função de envio;
2. no *rank* de recebimento a variável I não pode ser lida após o retorno da função de recebimento;
3. A operação MPI_Isend retorna quando ocorre a sinalização ao controle do paralelismo (controle do MPI) que o envio deve ser iniciado, sem conhecimento do recebimento;
4. A operação MPI_Irecv retorna quando ocorre a sinalização ao controle do paralelismo que o recebimento deve ser iniciado, sem conhecimento do envio.

As características mostram, então, que não há qualquer garantia ao utilizar-se de funções não-bloqueantes. Não há garantia quanto a modificação ou leitura da variável, e não se pode inferir quanto a ordenação da execução, menos ainda quanto ao sincronismo.

O padrão fornece um tipo de operação para assegurar que o envio, ou recebimento, deve terminar antes de seu retorno, e a genérica é a MPI_Wait. Ela assegura que a operação

não-bloqueante à qual é atrelada termine, ou seja, se o MPI_Wait for realizado por quem envia assegura que I pode ser alterado. Se for realizado por quem recebe, que I pode ser lido. Não há conhecimento sobre o sincronismo entre as imagens, mas o MPI_Wait garante que a operação – envio ou recebimento – terminou com sucesso.

Ainda sobre o MPI_Wait fica a critério da implementação o comportamento dos pontos 3 e 4 das características da transferência de dados. Não é assegurado que a espera atrelada ao envio, ao retornar, garante que a espera atrelada ao recebimento terminou.

Cada mensagem possui uma etiqueta (*tag*). Com estas etiquetas e com operações de verificação o programador possui as condições para assegurar a ordenação do processamento entre instâncias distintas no processamento, utilizando etiquetas sequenciais, por exemplo. Além disso, de forma direta, existem operações do tipo barreira, como $MPI\_Barrier$, onde todos os processos de um comunicador invocam esta operação, e só neste instante ela retorna, forçando assim um sincronismo.

2.2 CUL: comunicação unilateral

Esta forma de transferência de dados muda o paradigma da comunicação. A CBL parte da paridade envio-recebimento, com operações distintas para cada parte. Já a CUL não possui esta distinção, pois quem envia descreve o recebimento, ou quem recebe descreve o envio.

De maneira semelhante a descrita no padrão, para facilitar, vamos definir dois conceitos: a origem (*origin*) e o alvo[7] (*target*). Origem é o processo que realiza a chamada da função, e alvo é o processo cuja memória é acessada a partir do processo origem, acesso este realizado de forma passiva, sem conhecimento do mesmo, geralmente em segundo plano, "em paralelo" a execução do programa.

O esquema descrito na figura 2.3 ilustra a disposição da origem e do alvo na comunicação.

Podemos observar que a CUL é realizada por duas funções equivalentes, a MPI_Put e a MPI_Get. A MPI_Put envia dados do *rank* que a invoca ao *rank* remoto, o alvo. Já a MPI_Get faz o caminho inverso, recebendo, do *rank* remoto, o alvo, dados e o armazena na origem, o *rank* que invoca a função. Note que uma delas é suficiente, não sendo necessário que o alvo execute alguma função recíproca para a comunicação.

Porém, para que o alvo tenha conhecimento que a comunicação (a transferência) ocorreu é necessária uma operação distinta à comunicação. O padrão a denomina de "sincronização". A seguir são colocadas as funções e operações para a iniciar a comunicação, a comunicação propriamente dita e a sincronização entre os *ranks* envolvidos na comunicação.

[7] Dizer que a contrapartida de origem em CUL é o destino é impreciso. Destino costuma ser aquele que recebe alguma coisa, mas, em CUL, o "alvo" pode ser aquele que envia algo. Portanto o termo utilizado, "alvo", é o mais correto, apesar de estranho.

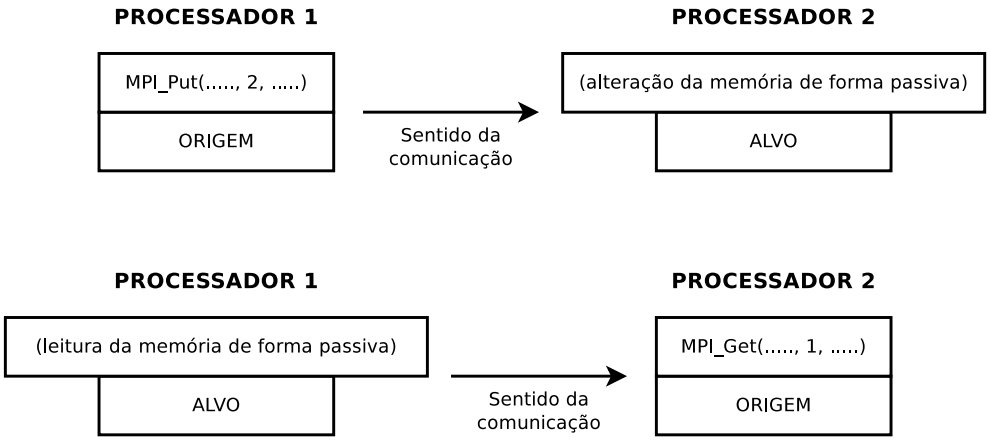

Figura 2.3: Diagrama explicativo do significado de origem e alvo na CUL.

2.2.1 Operações de Comunicação

As operações da CUL possuem todos os argumentos para sua execução declarados por somente uma das partes da comunicação, ou seja, devido ao fato de as CUL serem realizadas por somente um participante da comunicação, então este participante deve ter todo o conhecimento da comunicação. Isso significa que este participante saberá de onde obter a informação e para onde esta informação deverá ser enviada.

Existem duas operações que realizam esta comunicação: a MPI_Put e a MPI_Get. A MPI_Put recebe como *origem* a memória do processo que executa esta função, e envia o conteúdo desta memória para o processo *alvo*, ou seja, a "fonte" da informação é o processo que realiza a chamada, e o "destino" é o processo remoto. Já na operação MPI_Get ocorre exatamente o inverso. Como colocado anteriormente todas as informações pertinentes à transferência do dado são conhecidas pela origem e fornecidas na chamada da função: posição de memória de origem, tamanho, tipo, *rank* remoto, posição de memória de alvo, tamanho da memória do alvo e tipo do dado no alvo.

2.2.2 Janelas para comunicação

Todas as operações de CUL em MPI tem como base em sua semântica a transferência de dados entre processos. Neste contexto somente um deles executa a operação, e o outro, de forma passiva, tem sua memória acessada, e se for desejado, modificada.

A região de memória que é acessada no processo *alvo* é exposta por janelas (*windows*), e a mesma é acessada pela origem usando operações da CUL. A operação MPI_Win_create é responsável pela criação das janelas, e recebe como argumento a região de memória que se deseja expor. Esta região é identificada com um ponto inicial e um tamanho, portanto a região é contígua. A operação MPI_Put, por exemplo, tem como origem o processo que a executa, enviando dados da memória local para o processo *alvo*, armazenando este dado na janela do processo destino. Já a MPI_Get envia dados do janela do processo *alvo* para a

memória local do processo *origem*, aquele que executa a função[8]. É inerente a operação da CUL o acesso a um dado no *alvo* através das janelas, e o dado local, na *origem*, não precisa estar exposto por uma janela.

As janelas são coletivas, isto é, todos os processos no comunicador MPI *comm* devem possuir a janela. Mas a área de memória exposta por elas não necessariamente precisa ser a mesma. O *rank* 1, por exemplo, pode expor uma variável qualquer X na sua janela *win*, e o *rank* 2, por sua vez, expor a variável qualquer Y na janela *win*. Podem ter tamanhos diferentes e tipos diferentes. Mas a janela – no caso, *win* – deve ser a mesma.

No mesmo exemplo, se for necessário o envio de um dado de 1 para 2, a memória a ser escrita em 2 está exposta por *win*. A janela *win* em 1 não é lida nem escrita, e portanto terá papel somente na sincronização. Será denominada de "janela tipo *dummy*"[9].

As operações da CUL que transferem dados de memória local para janela remota, ou o inverso, não precisam ter a área de memória local, envolvida na operação, exposta via janela. Isto, por sua vez, corrobora com a idéia de janelas *dummy* como descrito anteriormente. E também significa que as operações da CUL, ao realizarem a comunicação, poderão trocar dados entre janela remota e área de memória local, exposta ou não em outra janela. Neste segundo caso é importante tomar cuidado ao tratamento de sincronismo, pois mesmo não havendo sobreposição de janelas, a operação cuja parte local está exposta por outra janela pode estar escrevendo na memória na janela temporal de exposição[10] que outra operação com o acesso a janela da área de memória local da primeira operação da CUL.

O padrão MPI coloca uma maneira mais clara de ilustrar o conceito de janela. A abstração descrita é de se imaginar uma cópia do dado local e uma cópia para cada janela que a memória está exposta (lembrando que é possível expor uma mesma região de memória através de várias janelas). A figura 2.4 ilustra os acessos da CUL (RMA na figura).

A figura mostra que atualizações em qualquer parte da janela – seja parte local através de operações LOAD/STORE, seja parte remota – não é assegurado o sincronismo das informações a cada operação (seja ela da CUL ou locais). E podemos observar a abstração colocada pelo padrão, onde, no caso, existem três cópias do conteúdo da memória: uma para cada uma das duas janelas e uma cópia local. Com isso podemos verificar que não existem garantias de atualizações entre cópias enquanto não houver alguma operação de sincronismo (a ser explicada mais adiante).

Em um nível mais baixo estas cópias podem estar implementadas de diversas maneiras. O tratamento mais próximo da figura anterior é através de *buffers*, onde estes são as cópias da memória para cada janela que a expõe. Outra forma é através do mapeamento global

[8]Existe uma terceira função, denominada $MPI\_Accumulate$. Esta tem semântica diferente (realiza operações com dados nas janelas de um ou mais processos destino) e não será englobada neste documento.

[9]Um exemplo interessante é em linguagem C, onde o programador pode criar uma janela expondo a memória de NULL.

[10]Trecho entre duas barreiras, entre um post e um wait, entre um start e um complete, funções estas explicadas mais adiante.

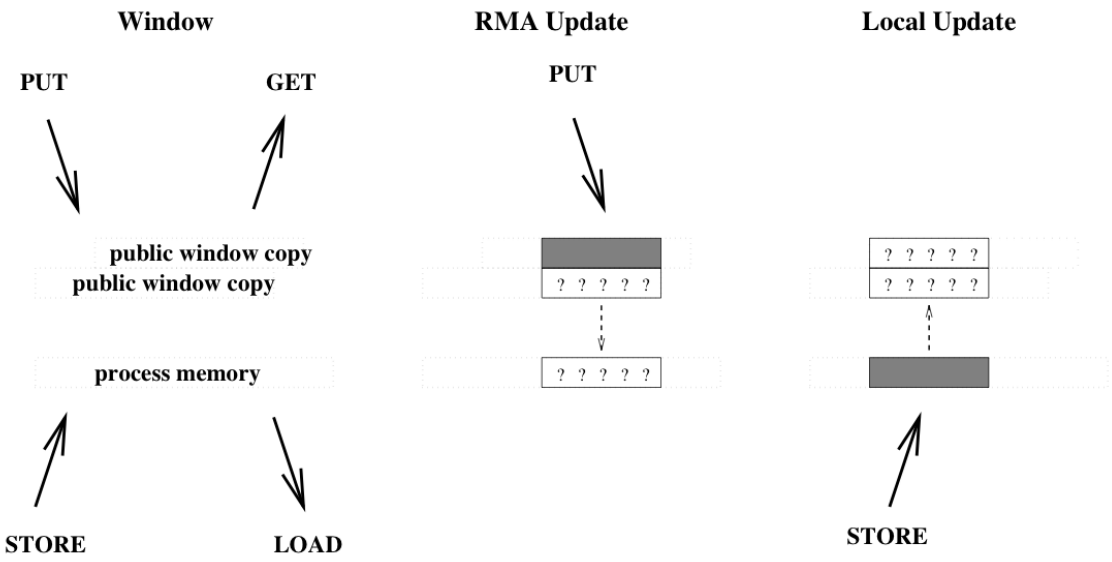

Figura 2.4: Descrição esquemática de janelas com uma cópia local e uma cópia para cada janela que expõe o dado (fonte: MPI-Forum [1998], pág. 350).

de memória e o tratamento das janelas fica a cargo da implementação, de funcionalidades do *hardware*, sistema operacional, etc. Com tecnologias de acesso direto a memória remota (*Remote Direct Memory* Access, RDMA) estas regiões podem ser mapeadas e a memória local (a cópia local) pode ser acessada diretamente pelo processo origem, e, neste caso, uma atualização na janela atualizaria a memória local do processo remoto. O padrão, porém, não trata deste caso especial, e as garantias de sincronismo – ou a falta delas – entre memória local e janelas levam para o caso mais geral, de falta de sincronismo entre estas memórias.

A liberação das janelas, através da operação MPI_Win_free, possui um detalhe interessante: ela é bloqueante, significando que todos os processos do comunicador *comm* precisam liberar "ao mesmo tempo", e somente depois que todos os processos invocarem a liberação é que esta função libera a execução do programa em todos os *ranks*. Este detalhe existe para não permitir escrita em janela remota quando a mesma já foi fechada, o que ocorreria sem o bloqueio da liberação da janela.

2.2.3 Operações de Sincronização

As comunicações da CUL são livres de sincronismo implícito. Isso ocorre pois o padrão MPI não impõe restrições a arquitetura do computador ou a peculiaridades do sistema operacional, e as operações de comunicação possuem retorno imediato, sem garantias de término da comunicação no retorno das mesmas. Portanto é necessário que ocorram operações de sincronização para assegurar que um determinado dado seja transferido.

Diferentemente do que foi descrito em 2.1.2, onde há uma operação de espera atrelada a cada parte da comunicação (atrelada ao *send* e ao *recv*), a CUL não permite este tipo de abordagem. A maneira escolhida pelo padrão é de sincronizar as partes envolvidas na comu-

nicação, ou seja, criar um mecanismo onde a origem tenha conhecimento que a comunicação terminou e o alvo tenha conhecimento que as modificações em sua janela foram realizadas, permitindo acesso à mesma.

A fim de ilustrar a necessidade do sincronismo, digamos que em uma computação estão envolvidos dois processos, o processo A e o B. Cada um tem a sua variável I, sendo em A $I^A = 10$ e em B $I^B = 20$. O processo A executa duas operações de comunicação, uma Put de I^A para B e outra Get logo após, mas com o sentido inverso. A variável I em B já está previamente exposta por uma janela. O processo B não sabe que estão ocorrendo tais comunicações em sua janela, pois estas operações são unilaterais e este processo é alvo em ambas as operações. A fim de facilitar, vamos supor que o resultado ao final deve ser 10.

O padrão deixa claro que se não ocorrerem sincronizações não há nenhuma garantia que a primeira operação Put termine a tempo, no alvo em B, da segunda operação Get começar. Poderá ocorrer que a operação Get realizada por A obtenha em B o valor original (sem atualização da Put), o que não é desejado.

O padrão introduz a existência do conceito de "fatias de tempo", ou *epochs*, justamente para possibilitar a manutenção do sincronismo entre processos. Isto é, a comunicação ocorre em trechos temporais onde em uma *epoch* algumas operações relativas a comunicação podem ocorrer, sem garantia alguma da ordem das mesmas, nem mesmo quando se trata da ordem das operações realizadas pelo mesmo *rank*. A *epoch* pode garantir que o valor de I^A do exemplo anterior seja, ao final da execução do programa e de forma definitiva, igual a 10, se a operação Put for realizada em um *epoch* anterior a operação Get.

Estas *epochs* de comunicação no decorrer da execução de um programa são gerenciados por diversas operações de sincronismo. A seguir são listadas aquelas apresentadas no padrão MPI.

Barreira

A função MPI_Win_fence em MPI possui um conceito muito similar ao de barreira. Deve ser invocada por todos os processos que possuem a mesma janela no comunicador *comm* e seu retorno é realizado quando todos os processadores do comunicador invocam a barreira. Ela assegura que todos os processos chamaram esta função, e que as operações CUL ocorridas antes dela terminaram. Simples mas muito proibitiva, pois pode forçar que uma aplicação tenha que ter trechos de sincronismo e de computação, bem separados. Caso um dos processos seja muito lento para terminar suas contas antes da próxima barreira os outros ficarão ociosos esperando o término, o que não é eficiente.

Trechos de exposição da janela

Uma janela pode ser exposta, em um controle mais refinado de sincronismo através das operações MPI_Win_start, $MPI\_Win\_complete$, MPI_Win_post e MPI_Win_wait.

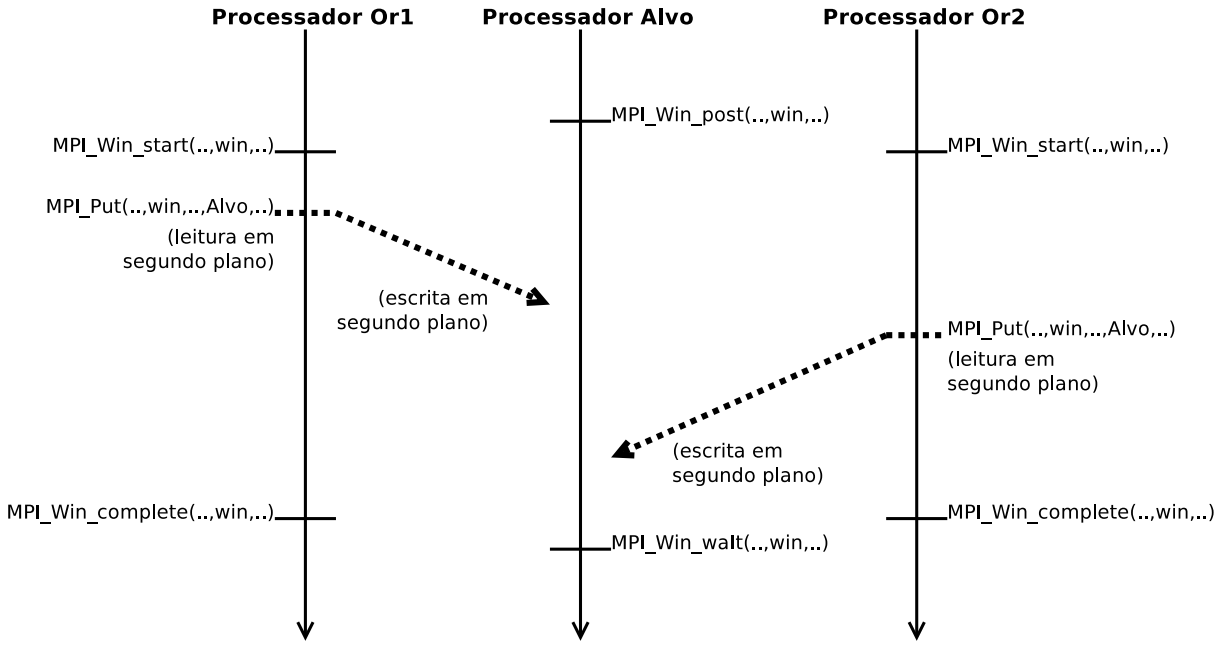

Figura 2.5: Descrição esquemática da ordem de chamadas das funções de controle refinado de sincronismo em MPI CUL.

Estas operações não são coletivas. As operações MPI_Win_start e seu complemento, a $MPI\_Win\_complete$, são invocadas pelo processo origem. Antes da operação de CUL é aberto o trecho temporal de exposição da janela via MPI_Win_start. Após a operação de CUL o trecho é finalizado via $MPI\_Win\_complete$. No processo *alvo* primeiro é aberto o trecho temporal de acesso a janela remota via MPI_Win_post, e finalizado via MPI_Win_wait.

O controle do sincronismo é mais refinado pois é possível que a origem e o alvo façam a exposição da janela de maneira independente. Este tipo de sincronismo permite, em uma mesma janela comum a todos os processos de *comm*, que a sincronização seja somente com aqueles envolvidos nas operações de CUL.

A ordem temporal natural das chamadas das operações entre origem e destino é a seguinte: o *alvo* expõe a janela via *post*; a *origem* começa o trecho temporal via *start*; a *origem* executa a operação de CUL; a *origem* fecha o trecho via *complete*; o *alvo* fecha a exposição via *wait*. Com respeito a duração de cada trecho, o *start complete* ocorre entre a execução do *post* e do *wait*. A figura 2.5 ilustra a ordem das chamadas.

Em outras palavras, o *alvo* "posta", via MPI_Win_post, a janela para que operações da CUL ocorram nela. Qualquer origem "começa" o acesso a janela (*start*) do alvo, via MPI_Win_start. Uma vez feito isso – *post* da janela pelo *alvo* e *start* da janela pela(s) origem(ns) – operações da CUL podem ocorrer. Elas podem ser realizadas por diversas origens na mesma janela, já que estas operações não impõem criação de pares de processos (ao contrário do travamento de memória, a ser explicado mais adiante). A finalização da *epoch* de acesso a janelas é "completada" pela(s) origem(ns) com o $MPI\_Win\_complete$.

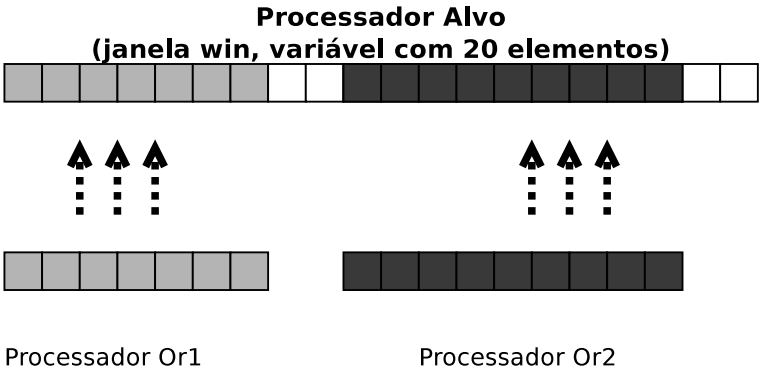

Figura 2.6: Escrita por MPI CUL em um vetor exposto pela janela *win*. Vale notar que não há sobreposição de endereços na escrita.

No *alvo* a janela é colocada em "espera" via MPI_Win_wait.

Pode-se observar, na figura 2.5, que ocorrem dois acessos na mesma janela. Como não existe forma de identificar qual dos dois acessos – do processador $Or1$ ou do $Or2$ – irá escrever primeiro em *win* no processador *alvo* poderá ocorrer uma *race condition*. Caso a variável exposta por *win* no *alvo* seja um vetor de E elementos, e as escritas forem realizadas em trechos distintos do vetor não ocorrerá a *race condition*. A figura 2.6 coloca este caso.

Mas, caso a variável seja um escalar, não há como evitar a *race condition*, estando as duas chamadas MPI_Put na mesma *epoch*. Porém o mecanismo de travamento de memória, a ser explicado no próximo item, auxilia neste caso.

Travamento de memória

A terceira forma de sincronismo e organização das operações de CUL é através de travamento, ou "pinagem", da memória.

A origem utiliza da operação MPI_Win_lock para ter exclusividade de acesso a uma janela de um determinado *alvo*. Outras operações de travamento a esta janela por outros processos ficam proibidas de ocorrer, e estarão em espera até o destravamento da janela pela origem que a travou, através do MPI_Win_unlock. Ao se travar uma janela, a origem pode realizar operações de CUL na mesma. Qualquer acesso a esta janela por operações de CUL por outro processo que não a tenha travado é proibido.

A garantia da atualização de um determinado dado na memória local do *alvo* (transferência de dados entre janela e memória local) é assegurada com o retorno do MPI_Win_unlock. Para o *alvo*, como ele não possui maneiras de saber se a origem liberou a janela ela não sabe quando sua cópia local será atualizada (a janela local conforme explicado anteriormente), e só será assegurada a atualização no retorno do MPI_Win_free. A figura 2.7 ilustra o travamento.

O travamento de memória, em virtude do que foi colocado no parágrafo anterior, não parece ser muito indicado para quando o processo que tem sua janela travada/destravada

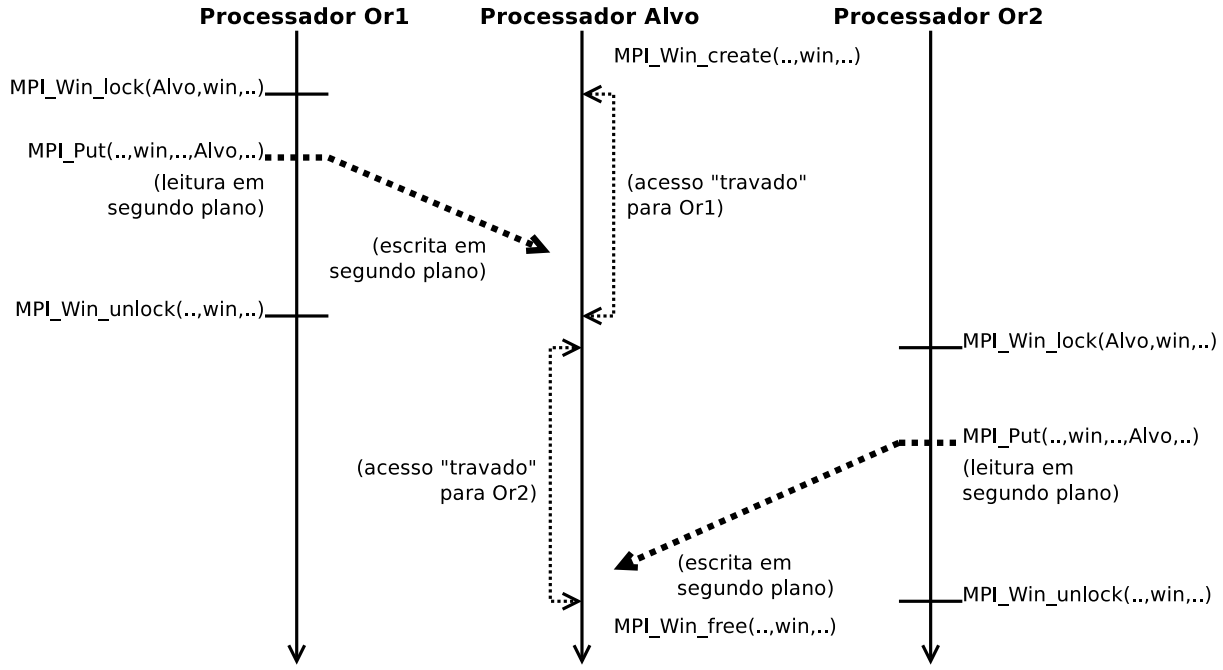

Figura 2.7: Esquema de travamento de memória. O processo *alvo* tem sua janela travada e alterada de forma passiva. Uma vez travada por $Or1$ não é possível acesso de outro processador. O MPI_Win_free só libera a execução quando todos liberarem a janela (esta situação não está colocada na figura).

precisa ter acesso ao conteúdo desta janela. Mas pode ser interessante quando houver alguma estrutura de dados central (todos os processos acessando um conjunto de dados para serem processados em paralelo sem cópia entre processadores, por exemplo).

2.2.4 Semântica do funcionamento da comunicação e sincronização

O padrão deixa bem claro o seguinte: as operações de transferência não respeitam a ordem temporal de execução quando dentro de um mesmo *epoch*. Com isso algumas regras são impostas para permitir que este tipo de decisão seja mantida.

Uma regra direta é aquela que diz sobre o acesso a mesma janela de comunicação, no mesmo processo, por duas operações – *Put* e *Get* – no mesmo *epoch*. O padrão não permite isso. Da mesma forma, duas operações *Put* na mesma janela no mesmo processo não é permitida. Mas duas operações *Get*, na mesma janela, no mesmo processo, é permitida, pois é bastante natural a leitura concorrente.

Toda operação da CUL termina, e o momento exato de ela terminar não é passível de ser conhecido dentro de um *epoch*. O padrão, porém, é enfático: estas operações terminam no sincronismo. E isso também limita o uso concomitante de operações de escrita seja via *store* local (por exemplo, uma atribuição) seja via um MPI_Put ou MPI_Get. No exemplo do início do item 2.2.3, se o processo 1 recebe, via MPI_Get, um valor do processo 2, e devido ao retorno imediato das funções, o processo 1 escrever na variável outro valor via *store* local,

na mesma *epoch*, não haverá nenhuma garantia de qual das operações - MPI_Get ou *store* - terá seu valor mantido no fim da sincronização subsequente. O mesmo ocorre com um MPI_Put e um *store* em seguida (indefinição do valor a ser enviado ao processo remoto: valor antes ou depois do *store*) ou ainda um MPI_Get e um *load* (qual valor será lido pelo *load*: o proveniente da operação de CUL ou o valor anterior ao MPI_Get? Este valor não é determinado *a priori*).

Outra regra importante é relativa a escrita local (através de operações do tipo *load*/*store*, intrínsecas a linguagem) em uma área exposta por uma janela. Boa parte da seção 6.7 do padrão MPI descreve as regras relativas a estas operações e às operações de comunicação. Estas escritas em geral não podem ocorrer quando houver, em um *epoch*, alguma operação de comunicação sendo invocada, não importando quando.

Estas regras ditam o funcionamento do processo *origem*, onde a operação da CUL foi realizada. Já no processo *alvo* a semântica é semelhante, seguindo o sincronismo/fechamento da *epoch* pelo processo *origem*.

2.3 Observações

A mudança radical de semântica de MPI CBL e CUL significou que a tarefa de transferência de informações entre processos deve ser feita somente por um deles, aquele que executa a operação de comunicação. Isso significa que este processo, o executor, deve saber, *a priori*, a localização do dado localmente e no processo remoto. E toda operação de comunicação unilateral não possui paridade, ou seja, somente um dos envolvidos na comunicação sabe o instante exato do início da operação.

Estes detalhes, apesar de simples, são poderosos no que diz respeito a liberdade de comunicação. O momento exato para a comunicação é conhecido somente por quem executa a operação de comunicação. O processo remoto tem sua memória alterada (lida ou escrita) de forma passiva sem necessidade de "paridade" ou "correspondência" das operações. Um dado é alterado sem conhecimento pelo detentor da memória remota. A *unilaterialidade* da comunicação está cristalizada nestas proposições.

Em termos de programação o dado ser atualizado de forma passiva em uma área de memória pode significar esforços reduzidos de escrita de código para controle e distribuição de trabalhos, uma vez que a paridade de operações (um processo envia explicitamente; o outro recebe explicitamente) não é necessária. Como processos diferentes podem significar sequência de operações diferentes esta não-paridade é natural: em qualquer equipamento que não tenha coerência de *cache*, ou outra forma de controle forte para a sequência de operações em todas as instâncias de processos paralelos, ela ocorre. A linguagem permitir em sua semântica a não-paridade possibilita poder explorar esta não-paridade natural.

Outro ponto importante é o fato de o dado ser lido ou escrito em memória remota já com seu formato e tipo de dado. Portanto a transformação de um tipo de dado derivado, por

exemplo, pode ser realizada pelo controle MPI (ele permite que se envie um dado cujo tipo é derivado de uma forma para o processo origem e de outra forma para o processo de destino, a conversão é feita automaticamente), o que também pode significar redução no esforço de programação.

Em termos tecnológicos a CUL possibilita redução de *buffers* intermediários, uma vez que um pacote (um determinado dado e localização local, remota, processo de origem e de destino) possui informações para acesso direto, à memória remota, pelo processo origem. A tecnologia RDMA é um exemplo disso. Como uso de *buffers* significa cópia de memória, e mesmo sendo feita em segundo plano, pode acarretar em penalidades de tempo ao programa, ou, em nível mais superior, ao sistema operacional.

A CUL no padrão MPI, segundo Dobbelaere e Chrisochoides [2007], pode ser realizada de forma simulada com uso de rotinas da CBL. Mas como a CUL possui em sua semântica a falta de paridade nas operações ela traz maior especificidade e robustez, sem qualquer simulação de rotinas de outro tipo. Mesmo com a simulação da CUL por rotinas da CBL não se consegue ter a robustez proporcionada por uma semântica padronizada de transferência unilateral como a CUL do padrão MPI.

A transferência de dados com endereço completo de origem e destino pode significar um esforço significativo para localização da informação na memória remota. Se o paralelismo empregado permitir que a distribuição dos dados nos processos forem dísparos (grades não estruturadas com divisão não homogênea, como pode ser visto no item 5.2) inferir o mapeamento remoto da memória pode ser impossível sem consultar a outra parte. Se houver esta consulta de mapeamento o problema fica parcialmente resolvido, mas se o mapeamento for dinâmico fica mais complicado. O problema se concentra em predizer, ou conhecer, *a priori*, a posição dos dados na memória remota. Isso não ocorre no MPI CBL, pois não há esta necessidade de se saber exatamente o local. No capítulo 5 iremos definir melhor este problema e verificar se esta nova forma de comunicação é interessante para grades não estruturadas.

Capítulo 3

A extensão *Coarray* Fortran

O presente capítulo trata de descrever a extensão *Coarray* Fortran. Os capítulos 4 e 5 mostram sua aplicação em no jogo da Vida e OLAM.

A linguagem Fortran, uma das primeiras linguagens de programação, tem na computação científica e nas engenharias a sua principal área de utilização. Estas áreas, por sua vez, demandam enorme poder de processamento, que pode ser conseguido através do uso de várias unidades de processamento, sendo utilizadas em paralelo.

O esforço mais proeminente para trazer o poder do paralelismo para a linguagem é o *Coarray* Fortran (CAF). Segundo Reid [2009], "Qual é a menor mudança requerida para converter o Fortran em uma robusta e eficiente linguagem paralela?. Nossa resposta é uma simples extensão na sintaxe."[1]. Esta extensão permite a operação básica para o paralelismo: a execução concomitante de duas ou mais instâncias do programa. Já na execução paralela ela também possui mecanismos de controle de ordenação das operações entre instâncias e operações ou notação para permitir comunicação de dados.

A seguir é descrito, em linhas gerais, o CAF, seus operadores, nomenclatura e sua sintaxe. Também é colocada uma análise da semântica de suas operações, principalmente daquelas envolvidas com a comunicação e sincronismo. Segue também uma visão dos compiladores existentes que possuem suporte a esta extensão, seguido de algumas observações sobre o *Coarray* Fortran.

3.1 Principais operadores do CAF e nomenclatura

A seguir são colocados os principais operadores de CAF pertinentes ao contexto do trabalho[2]. Primeiramente são explanadas as referências às instâncias do paralelismo. Logo após é colocada a referência e especificação a objetos de dados (variáveis), com atenção especial a

[1] Reid [2009], página 4, tradução livre.

[2] O padrão CAF, colocado por Reid [2009], possui toda a descrição da sintaxe e semântica, garantias, proibições e comportamento do programa que usa a notação e seus operadores. O que é colocado a seguir é, de certa forma, um resumo do padrão, com ênfase no contexto do presente trabalho. Esta descrição não pretende esgotar as informações colocadas no padrão, pois caso contrário se trata de redundância.

notação, que utiliza colchetes [ ]. Segue com a explicação dos operadores de sincronismo, e finalizamos com um resumo dos demais pontos do padrão.

3.1.1 Referenciando imagens

O CAF leva à linguagem Fortran o modelo de paralelismo "*Single Program Multiple Data*" (SPMD). Desta forma o programa é replicado N vezes, e cada replicação do mesmo possui um conjunto de dados próprio. A denominação para cada replicação, em CAF, recebe o nome de "Imagem" (do inglês *Image*). O número de imagens sendo utilizadas na computação paralela é obtido pelo operador $num\_images()$, e este número é especificado em tempo de execução. Porém o padrão não descreve operadores para controlar o número de imagens envolvidas na computação, ficando, portanto, a cargo da implementação este controle (via linha de comando, por exemplo).

As imagens no contexto do paralelismo possuem um índice, que varia de 1 (um) até N, sendo N o número total de imagens. A obtenção deste número é através do operador $this\_image()$[3].

3.1.2 Especificando e acessando objetos e variáveis

Cada imagem possui seu conjunto de dados. Porém o CAF permite que uma variável possua uma "codimensão", inserindo-a no contexto do paralelismo. A fim de facilitar o entendimento deste conceito-chave vamos lançar mão a um exemplo.

Tomemos o seguinte trecho de código em Fortran (somente declaração de variáveis):

```
integer, dimension(30), codimension[*] :: x
integer :: i1, i2, i3, i4
character :: c1, c2, c3, c4
character :: z1(50,60)[*], z2(50)
```

Podemos observar que as variáveis x, $i1$ até $i4$, $c1$ até $c4$, $z1$ e $z2$ são declaradas. Possuem seus tipos (inteiro, real, caractere) e um tamanho (30 elementos para x, 1 elemento para $i1$ até $i4$, $c1$ até $c4$, 50 por 60 elementos para $z1$ e 50 elementos para $z2$).

As variáveis x e $z1$ possuem um operador diferente: é o uso de uma "codimensão", que é conseguida através do uso de colchetes [ ] ou, de forma explícita, com o atributo "*codimension*". Este novo operador é proveniente do CAF.

O padrão CAF deixa claro que estas variáveis, x e $z1$, também são locais. A imagem que as declara pode referenciá-las de forma direta, com o uso das operações que o padrão Fortran descreve (atribuição, envolvimento das mesmas em operações aritméticas, como argumento a funções ou sub-rotinas, etc). Porém estas variáveis podem ser referenciadas por outras

[3]Estes dois operadores, $num\_images()$ e $this\_image()$, são funções "intrínsecas", e não "sub-rotinas". Desta forma não são chamados via comando *call* do Fortran.

imagens, de forma simples, com o uso de notação *coarray*. Inclusive não é interessante ao programador entender que uma variável com a "codimensão" tenha "cópia local" e "cópia externa (ou própria ao paralelismo)". Um *coarray* é uma região de memória, em nível mais abstrato uma variável, que pode ser acessada por outras imagens, com o uso da notação *coarray*.

Vamos supor que temos quatro imagens sendo executadas no programa paralelo com a declaração de variáveis anteriormente colocada. Em um determinado ponto do programa existem as seguintes operações:

```
(1)  ! lendo conteúdo de x da imagem 3 e armazenando em i3
(2)  i3 = x(17)[3]
(3)  ! lendo conteúdo da décima coluna de z1 da imagem 1
(4)  ! e armazenando em z2
(5)  z2(:) = z(:,10)[1]
(6)  ! escrevendo em i2 da imagem 3
(7)  i2[3] = 123
```

Vamos analisar a sintaxe das linhas 2, 5 e 7. A linha 2 realiza a atribuição do conteúdo da variável x, posição 17, da imagem 3, para a variável $i3$ na imagem que executa esta operação. Já a linha 5 efetua o mesmo porém com notação implícita, copiando conteúdo da variável $z1$ na posição $(1...50, 10)$ da imagem 1 para a variável $z1$ na imagem que executa esta operação. A linha 7, por sua vez, executa a operação de atribuição do valor inteiro 123 para a variável $i2$ na imagem 3, e esta operação é realizada pela imagem que contém esta linha de código.

Temos um problema com a linha 7. Ela resultará em falha pois $i2$ não foi declarada com o atributo de "codimensão", não podendo ser referenciada como sendo um *coarray*. As operações de atribuição que envolve o uso de [ ] ficam proibidas. Já as outras linhas (2 e 4) serão executadas com sucesso. E, como o exemplo não possui nenhuma condicional envolvendo o índice, todas as imagens executarão a atribuição.

De forma clara podemos observar os pontos em que ocorre comunicação. Estes pontos são identificados pela notação CAF, o uso de colchetes. Sempre que são utilizados, exceto na declaração de variáveis, ocorre comunicação, "de" ou "para" a imagem referenciada entre os colchetes. Já nas outras operações, onde os colchetes não são empregados, temos o acesso a memória local da variável.

Um ponteiro não é permitido ser um *coarray*, ou, em outras palavras, ter o atributo da "codimensão". Para que um ponteiro faça parte da comunicação com *coarrays* a variável deve ser de um tipo derivado, por exemplo T, sendo que T tenha um ponteiro. A "codimensão" é na variável do tipo T, e não nos elementos que o tipo T venha a ter. Entre estes elementos um ponteiro pode existir, e o padrão permite operações de atribuição envolvendo este ponteiro. O código a seguir mostra melhor esta proposição.

```
(1) type T
(2)   integer, pointer :: ptr
(3)   integer, allocatable :: arr
(4) end type

(5) type(T) :: var[*]
(6) integer :: dest, result
(...)
(7) dest = this_image()
(8) var%ptr => dest
(9) result = var[2]%ptr
```

As linhas 1 a 4 mostram a declaração do tipo T, que possui o ponteiro para inteiro *ptr*. A variável *var* é um *coarray*, do tipo T, como visto na linha 5. Na linha 6 são declaradas duas variáveis tipo inteiro *dest* e *result*. Na linha 7 há a atribuição do índice da imagem à variável *dest*. Supondo duas imagens, temos em 1 o valor de *dest* igual a 1. Em 2, o valor é 2. A linha 8 aponta o ponteiro *var%ptr* para a variável *dest*.

A atribuição da linha 9 é: "vá para a imagem 2, e, no componente *ptr* da variável *var* naquela imagem e copie o conteúdo deste componente (no caso, o destino que o ponteiro apresenta) para a variável *result*". Portanto, *result* em qualquer das duas imagens será igual a 2.

Coarrays podem ser alocáveis (*allocatable*), com a restrição de que, quando forem alocados, tenham o mesmo tamanho em todas as imagens, de forma a ser aderente com a exigência de que os *coarrays* sejam idênticos entre todas as imagens. Como forma de permitir alocação diferente entre as imagens, na mesma idéia apresentada anteriormente, um elemento do tipo T pode ser alocável (conforme linha 3), e a alocação é realizada de forma independente para cada imagem, com referência a variável, *var%arr*. A referencia ao tamanho da variável alocável é semelhante a referência a ponteiros, sendo necessário, antes da atribuição, verificar o tamanho da variável na imagem desejada.

Estas restrições são resilientes ao fato de um *coarray* ser semelhante, quase idêntico, entre as imagens. Não é permitido, por exemplo, algumas imagens declararem um *coarray*, e outras não. Todo *coarray* é global, de mesmo tamanho, ordem, com os mesmos limites e forma.

3.1.3 Sincronismo

As operações de sincronismo existem em CAF para permitir que a execução de cada imagem seja a mais independente possível, não sendo necessário, para a execução de cada linha de código, independente do uso da notação CAF, o conhecimento do que está ocorrendo em outra imagem.

Desta forma quando há acesso a memória de outra imagem não há garantias que se acessará uma memória atualizada. Se a imagem remota estiver alterando o conteúdo desta memória poderá ocorrer acesso durante a alteração, o que não é desejado. Caso o programa tenha este tipo de situação o programa pode se tornar incorreto pois possibilita a ocorrência de "*race conditions*".

Vamos supor que duas imagens, P de índice 1, e Q de índice 2, executam o seguinte código:

```
(1) integer :: x[*]
(...)
(2) x = this_image()
(3) if (this_image() == 1) then
(4)     x = x[2]
(5) end if
(6) if (this_image() == 2) then
(7)     x = x[1]
(8) end if
```

Devido ao fato de este trecho de código ser executado por duas instâncias em paralelo a linha 2 em Q pode ser executada depois da linha 4 em P, o que resultará que a atribuição da linha 4 seja errônea. A manipulação de *coarrays* poderia ser extremamente complicada caso o padrão não criasse o conceito de "segmento de execução (SE)" (do inglês *Execution segments*). Estes segmentos são delimitados por operadores chamados "*Image Control Statements* (ICS)".

Um SE é um conjunto de operações da linguagem Fortran (linhas de código) delimitado por ICSs (ou, no início, entre a linha *program* e a primeira ICS). A fim de possibilitar ao programador, e também ao compilador, ferramentas para ordenar a execução do programa no contexto do paralelismo, foi introduzido o conceito de ordenação dos SEs. A figura 3.1 ilustra o conceito de SE.

Segundo o padrão[4]:

[4] Reid [2009], página 16, tradução livre.

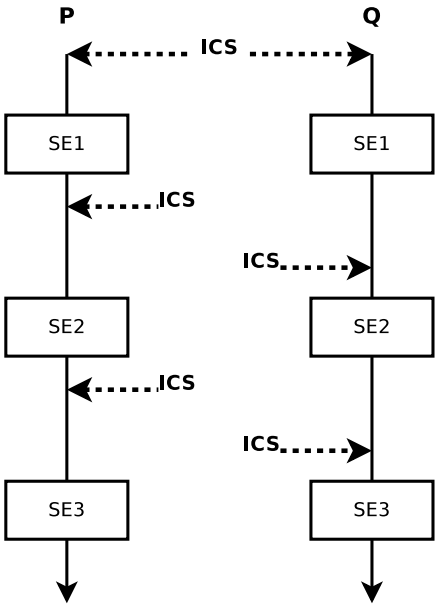

Figura 3.1: Ilustração do conceito de segmentos de execução. É importante atentar-se que as ICS ocorrem em instantes diferentes, e podem não ser o mesmo operador na mesma divisão de SEs.

> "Em cada imagem P, a ordem de execução das ICSs determina a ordem dos segmentos, P_i, $i = 1, 2, ...$. Entre imagens, a execução das ICSs correspondentes sobre as imagens P e Q no final de segmentos de P_i e Q_j pode assegurar que P_i precede $Q_j + 1$, ou Q_j precede $P_i + 1$, ou ambos.
> A consequência é que o conjunto de todos os segmentos em todas as imagens são parcialmente ordenados: o segmento P_i precede segmento Q_j se e somente se existe uma sequência de segmentos a partir de P_i e terminando com Q_j de tal forma que cada segmento da sequência precede a próxima, quer porque eles estão na mesma imagem ou por causa da execução de instruções de controle correspondente da imagem."

A fim de ilustrar a proposição anterior, o seguinte código é executado pelas mesmas duas imagens, P e Q:

```
( 1) integer :: x[*]
(...)
( 2) x = this_image()
( 3) sync all ! barreira: todas as imagens a executam
              ! e retornam quando todas "entrarem" na barreira
( 4) if (this_image() == 1) then
( 5)    x = x[2]
( 6) end if
( 7) sync all ! outra barreira
( 8) if (this_image() == 2) then
```

```
( 9)     x = x[1]
(10) end if
(11) sync all ! outra barreira
```

O programa agora está correto, sem a *race condition* que estava ocorrendo ao não utilizar as barreiras (que serão explicadas logo a seguir). A operação *sync all*, que é uma ICS, permite que o primeiro segmento SE_1 do programa (as linhas 1 e 2) esteja ordenado com o segundo segmento SE_2 (linhas 4, 5 e 6), e que SE_2 esteja ordenado com o terceiro SE_3 (linhas 8, 9 e 10). O SE_1 em P está ordenado também com SE_2 em Q, e SE_2 nesta imagem está ordenado com SE_3 em P. Porém SE_2 em P está não-ordenado com SE_2 em Q. Desta forma as atribuições das linhas 5 e 9 estarão com sua atribuição com correção assegurada.

Apesar de o padrão não deixar claro as operações de leitura, as atribuições que contêm a notação [ ] do lado direito da atribuição terminam com a comunicação realizada – a transferência ocorre e o dado é transferido da imagem remota para a imagem que executa a atribuição. A ordenação das SEs, neste caso, tem o objetivo de assegurar a ordem das operações e não permitir *race conditions*, por exemplo.

Porém a escrita em um *coarray* (quando a notação está do lado esquerdo da atribuição) pode ocorrer de forma assíncrona, através de uso de *buffers* ou *cache*. Este tipo de comportamento não está descrito no padrão, mas é completamente possível acontecer, já que ele assegura que as SEs estão ordenadas com o uso das ICSs. Entre SEs, desordenadas, em imagens distintas, a implementação do CAF está livre para realizar as otimizações que achar necessária.

As ICSs podem ser divididas em duas classes: aquelas que realizam algum tipo de sincronismo e aquelas que não realizam.

Sincronismo via barreira

O sincronismo via barreira possui a semântica de travar a execução do programa até que todas as imagens invoquem a barreira. Se um programa tem N imagens, e $N-1$ invocam a barreira, ela somente será "liberada" (e as imagens que a invocaram continuarão o processamento) se a imagem que ainda não invocou a barreira assim o fizer. Decorre que o número de barreiras é o mesmo em todas as imagens (não é possível, por exemplo, que algumas imagens executem X barreiras e outras Y, com $X \neq Y$). A figura 3.2 ilustra o conceito de barreira.

A barreira, por tratar de sincronizar globalmente no mesmo instante temporal, é plausível dizer que o tempo entre duas barreiras é maior ou igual ao tempo de processamento da instância que mais demora em processar. A tabela 3.1 ilustra este comportamento.

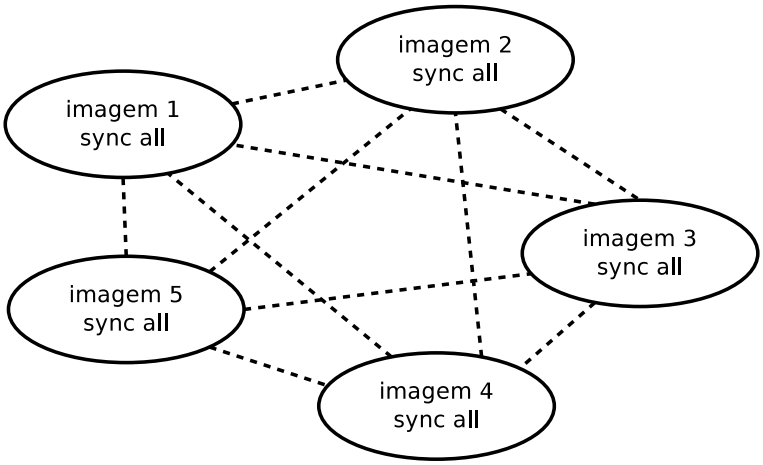

Figura 3.2: Sincronismo de barreira via *sync all*. O traço pontilhado é a ligação do sincronismo entre as imagens.

Instância	Tempo entre barreiras (excluindo-as) (s)
A	3
B	4
C	3
D	5
Tempo entre barreiras (incluindo as mesmas)	$5 + \epsilon$

Tabela 3.1: Sendo A a D instâncias de um programa paralelo fictício, é colocado o tempo de processamento entre o instante imediatamente após uma barreira e imediatamente antes da seguinte. Na última linha está o tempo entre duas instâncias consecutivas, incluindo-as. O tempo será 5 (o tempo da instância mais demorada, a D) e ϵ é o tempo que as duas barreiras demoram em seu processamento.

Trata-se de um mecanismo muito importante para ordenação do código. Toda operação executada antes da barreira na imagem P será executada antes da mesma barreira na imagem Q. O padrão coloca um detalhe interessante: não é necessário que a mesma barreira (a mesma linha de código) seja executada em todas as imagens, possibilitando assim flexibilidade na programação.

Sincronismo via grupos de imagens

Outro mecanismo de sincronismo é apresentado pelo padrão, e se trata de um sincronismo local, ou, de forma mais abstrata e imprecisa, uma barreira local envolvendo algumas imagens. O *sync images*(Z) é um operador que recebe como argumento Z um inteiro escalar com índice de uma imagem, um vetor de inteiros com vários índices ou asterisco ($*$) para se especificar todas as imagens.

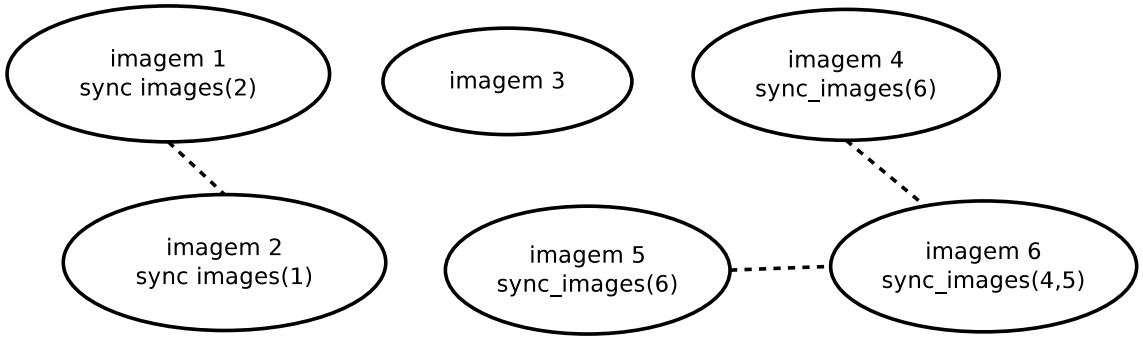

Figura 3.3: Sincronismo via *sync images*. O traço pontilhado é a ligação do sincronismo entre as imagens.

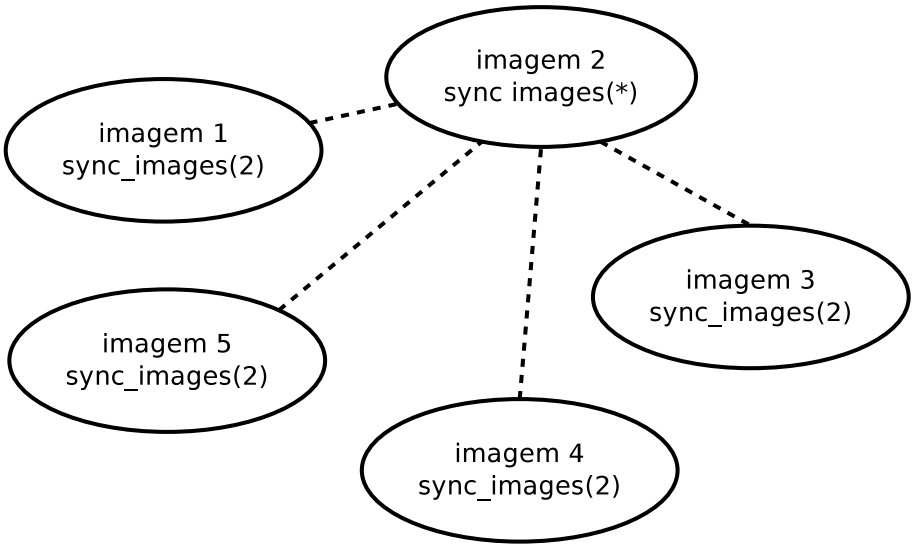

Figura 3.4: Sincronismo via *sync images* com a imagem 2 sincronizando com todas as outras imagens. As outras, por sua vez, somente sincronizam com a imagem 2. O traço pontilhado é a ligação do sincronismo entre as imagens.

Sua semântica permite mais flexibilidade. Um grupo de imagens pode realizar sincronismo e outro grupo continuar sua computação, sincronizando mais adiante, se desejado. A figura 3.3 ilustra este conceito.

As imagens 1 e 2 sincronizam entre si, com a 1 invocando *sync images* com argumento 2, e a 2 invocando *sync images* com argumento 1. A imagem 3 não está em momento de sincronização, portanto não invoca o operador. Já as imagens 4, 5 e 6 sincronizam de forma diferente. A imagem 4 sincroniza com a imagem 6, igualmente com a imagem 5. A imagem 6 sincroniza com a 4 e a 5. Porém a imagem 4 não sincroniza com a 5. O operador *sync image* permite esta flexibilidade.

Algo semelhante ocorre quando uma imagem invoca *sync images*$(*)$. A figura 3.4 mostra que a ligação do sincronismo é das imagens 1, 3, 4 e 5 com a 2, sendo que a 2 sincroniza com todas as outras.

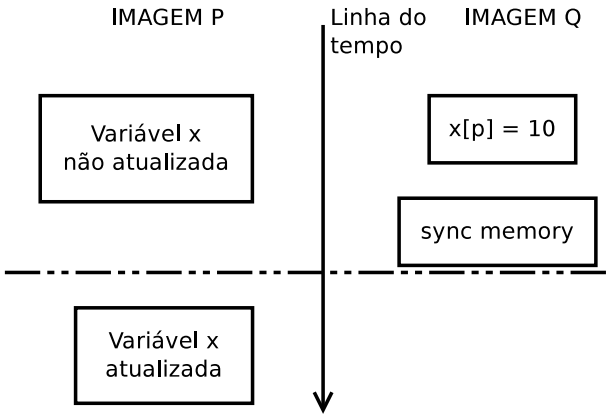

Figura 3.5: Ilustração, em termos temporais, da garantia de atualização de memória remota com operador *sync memory*. A variável x está declarada, e é do tipo inteiro.

O operador de sincronismo de memória

Segundo o padrão[5]:

> "A execução da *sync memory* define um limite de uma imagem entre dois segmentos, cada um dos quais podem ser ordenados de alguma forma definida pelo usuário em relação aos segmentos em outras imagens. Ao contrário das outras ICSs, não tem qualquer efeito de sincronização embutida. No caso de haver alguma ordem definida pelo usuário entre imagens, o compilador provavelmente vai evitar otimizações envolvendo movimentação de declarações entre antes e depois da *sync memory* e assim garantirá que qualquer dado modificado que a imagem tem na memória temporária, como *cache* ou registos ou até mesmo pacotes em trânsito entre imagens, torna-se visível para outras imagens. Além disso, todos os dados a partir de outras imagens que são mantidos na memória temporária serão tratados como indefinidos até que ele é recarregado a partir de sua imagem de acolhimento."

Denominada *sync memory*, esta operação não possui o objetivo de sincronizar duas ou mais imagens, mas sim assegurar que ocorra sincronização das estruturas internas da implementação (*cache*, *buffers* ou mesmo mecanismos de envio de dados através de operações como mensagens) com a memória remota (segundo Vaught [2008]). Funciona como uma sinalização à implementação, sem semântica direta ligada a comunicação ou sincronismo.

Desta forma, se uma variável na imagem P é alterada remotamente (com notação CAF do lado esquerdo da atribuição) a operação *sync memory* irá assegurar que esta operação termine e que o conteúdo da variável em P esteja atualizado, na imagem remota. A figura 3.5 ilustra esta situação.

Entretanto, devido ao paralelismo, pode ocorrer que *sync memory* não sincronize esta operação com a memória remota a tempo de a mesma ser acessada, o que é representado no retângulo "Variável x não atualizada" (na parte superior esquerda da figura 3.5).

[5] Reid [2009], página 20, tradução livre.

Demais ICSs

O padrão coloca como ICSs, sem efeito de sincronismo, os seguintes operadores:

- *lock* e *unlock* (travamento da memória);
- *allocate* e *deallocate* (presentes no Fortran);
- *critical* e *end critical* (criação de seção crítica);
- *end* e *return* (presentes no Fortran) quando ocorrer desalocação implícita de um *co-array*;
- *stop* e *end program* (presentes no Fortran).

3.2 Plataformas e compiladores com suporte a CAF

O padrão *Coarray* Fortran existe há bastante tempo. Como visto teve sua origem como uma extensão informal ao Fortran 95 no ano de 1998. Ainda segundo o autor do padrão desde o princípio dos anos 90 se tem notícia de algo parecido ao *Coarray* Fortran implementado em compiladores Fortran 77 das máquinas CRAY-T3D.

Desde estes anos a fabricante de supercomputadores Cray Inc. possui, em seus compiladores proprietários, suporte ao CAF (ou suas variações, como visto na extensão para Fortran 77). O *Cray Compiling Environment* (CCE) possui um desempenho competitivo comparado a outras alternativas de programação paralela, como MPI por exemplo. Mais detalhes sobre trabalhos realizados neste equipamento foram explicados no item 1.3.

As alternativas ao ambiente da Cray são poucas. Em Dotsenko *et al.* [2004] houve o desenvolvimento do compilador $cafc$, com implementação de partes do padrão CAF 2004 e que realizava uma tradução *source-to-source* de códigos em Fortran 90 introduzindo no mesmo chamadas a funções e operações para realização das transferências e sincronismo. Utilizava a biblioteca *mpich*1 e bibliotecas de comunicação unilateral (ARMCI descrito por Nieplocha e Ju [1999] ou GASNet descrito por Bonachea [2002]). Mostrou-se com bom desempenho em alguns *benchmarks* (Coarfa *et al.* [2005]).

O $cafc$ foi utilizado no começo desta pesquisa mas as experiências não foram boas. O compilador não possuía suporte a construções básicas do Fortran 90 (uso de módulos, por exemplo), o que impossibilitou a programação de códigos mais estruturados. Foram realizados alguns testes e o compilador gerava códigos pouco estáveis, e se optou por não mais trabalhar com este compilador.

Seu desenvolvimento está congelado desde 2006. Atualmente existem sinais que um novo compilador está sendo desenvolvido, mas sem nenhum *release* público até abril de 2010.

Em outubro de 2008 o compilador de código aberto G95[6] começou a suportar *coarray* como uma extensão ao padrão Fortran 95. O autor do compilador, em Vaught [2008], descreve a extensão CAF sob outra ótica, com exemplos que podem ser executados com o G95. Como o compilador já suportava o padrão Fortran 95 então a compilação de programas mais sofisticados com uso de notação CAF é realizada com sucesso, sendo necessário, porém, o uso de um "console" denominado *cocon*. O suporte ao *coarray*, portanto, permite investigar o funcionamento, desempenho e usabilidade do CAF em quase a sua totalidade, algo que era muito deficitário no $cafc$.

Em abril de 2010 o G95 possibilitou o uso de CAF em ambientes de memória compartilhada, sem uso do *cocon*, mas sim utilizando *threads*.

3.3 Observações

O CAF foi criado para trazer ao Fortran características de um modelo de programação paralela, com o uso de uma semântica que possibilite o acesso a memória remota de forma simples e clara.

Todas as imagens possuem o mesmo *coarray* com os mesmos atributos, tanto em forma, tipo e tamanho. Como ele é sempre conhecido – o *coarray* em um processo remoto é do mesmo formato do *coarray* local – é simples referenciá-lo, pois não há necessidade de se acessar a imagem remota para conhecer o *coarray*. E também não é necessário inferir sobre como o *coarray* remoto está descrito. Durante a execução de um programa o número de *coarrays* não varia. A alocação de *coarrays* de tamanhos dinâmicos é sempre conhecida pois é realizada de forma igual por todas as imagens. Esta semântica simples e pouco flexível cria um ambiente de endereços globais, mas particionado entre as imagens.

Decorre desta identidade entre imagens que o mapeamento de memória do programa é conhecido *a priori* pelo compilador, no mesmo grau e de uma maneira semelhante ao conhecimento de memória alocada dinamicamente. A parte local já é explorada por otimizações existentes dos compiladores atuais, com o embasamento de já estar existente no código e seu dinamismo ser dependente dos dados de entrada, alocação de vetores e matrizes, etc. Já a parte remota, referente àquelas vinculadas aos *coarrays*, que é dinâmica no número de processadores (o CAF é flexível no número de processadores utilizados), os *coarrays* são conhecidos independentemente do número de imagens, pois são todos iguais em todas as imagens. Portanto esta fato permite maior flexibilidade para as implementações do CAF tratarem o mapeamento de memória e assim da otimização.

As outras partes da semântica do CAF são simples também. A execução das imagens com o CAF é assíncrona, com os *coarrays* sendo acessados de forma semelhante. A execução do programa é marcado por "segmentos de execução" (SE), e a semântica deixa claro que

[6]Site: www.g95.org

a ordem é entre SE consecutivos. Operações que ocorrem dentro de um SE se supõe que estão desordenadas com relação as operações dentro da mesma SE. Este fato dá subsídios para o compilador alterar a ordem de execução das operações para a otimização. Estas otimizações, modificações no mapeamento de memória, implementações diversas de memória, comunicação, sincronismo são possíveis somente devido a semântica clara, pouco flexível e principalmente pelo fato de o CAF ser uma extensão a uma linguagem de programação, e não uma biblioteca, módulos ou qualquer agente externo.

Em termos de usabilidade do CAF se observa uma potencial clareza e facilidade na sua sintaxe. Programas codificados em CAF são claros e as comunicações são expressas de forma direta, pois estão descritas nas atribuições. Inclusive alguns trabalhos mostram redução de número de linhas de código em até 40%, o que também foi constado na implementação do jogo da vida.

As mesmas limitações, premissas para otimização, limitam bastante a programação em situações com divisões heterogêneas de domínio. Se um *coarray* for utilizado para permitir a troca de bordas, ele deve ter pelo menos o tamanho da borda. Em divisões heterogêneas as bordas podem ser de diferentes tamanhos, sendo necessário inferir o maior tamanho da borda por todas as imagens. Esta tarefa pode ser complicada quando forem empregadas muitas imagens.

Outro problema, devido a mesma limitação, ocorre quando as divisões, além de heterogêneas, forem dinâmicas, ou seja, mudam durante o processamento. Um *coarray* é estático e portanto seu uso neste caso fica comprometido.

A forma que o padrão encontrou para suprir este tipo de problema é um *coarray* com um tipo derivado, e este tipo ter elementos que são ponteiros ou dinâmicos (alocáveis). Segundo o padrão a comunicação de elementos desses tipos trocam informações sobre o tamanho, local da memória a ser acessada, limites do vetor antes da comunicação desejada. Neste caso pode haver penalidades para a otimização e para a implementação, a não ser que um controle central, com um mapeamento global de todos os *coarrays* entre em cena. Esta área importante do padrão CAF pode ser explorada neste caso.

Capítulo 4

O jogo da vida paralelizado

Neste capítulo apresentamos o jogo da Vida, suas implementações paralelas em MPI CBL, MPI CUL e em CAF. Os resultados experimentais serão apresentados no item 4.3.

Criado pelo matemático britânico John Conway em 1970, o jogo da vida (Gardner [1970]) é um autômato celular com o objetivo de reproduzir, em regras simples, alterações em grupos de seres vivos.

Conway estava interessado em um problema apresentado nos anos 40 pelo matemático John von Neumann. A idéia era criar uma máquina hipotética que conseguisse criar cópias de si mesma. Este conseguiu encontrar um modelo matemático em uma grade retangular, porém com regras muito complicadas. O jogo da Vida, por ter regras muito simples (quatro no total), possui idéia muito similar a de von Neumann, e desta forma ganhou fama quase que imediatamente.

Este jogo, ou melhor, o conceito apresentado por Conway logo atraiu muito interesse pois abriu um enorme campo: o de Autômatos Celulares. Inclusive ultrapassou as barreiras da matemática computacional com interesses de outras áreas, como biologia, economia, física e filosofia.

Ele parte de um tabuleiro de células, representado na figura 4.1, onde cada uma pode estar "viva" ou "morta". Este tabuleiro será preenchido com um estado inicial (uma distribuição de células "vivas" e "mortas") e uma vez preenchido ele avançará no tempo, ou seja, dado o estado inicial quatro regras serão aplicadas a cada célula, que gerará um novo estado. Este novo estado será então o estado inicial para a próxima iteração, e assim por diante.

Cada célula possui 8 vizinhos, e o estado de cada vizinho comanda o futuro da célula, através das seguintes regras:

1. Qualquer célula viva com 0 ou 1 vizinhos vivos morre (de "solidão");
2. Qualquer célula viva com 4, 5, 6, 7 ou 8 vizinhos vivos morre (de "superpopulação");
3. Qualquer célula viva com 2 ou 3 vizinhos vivos se mantém viva;
4. Qualquer célula morta com 3 vizinhos vivos se torna viva.

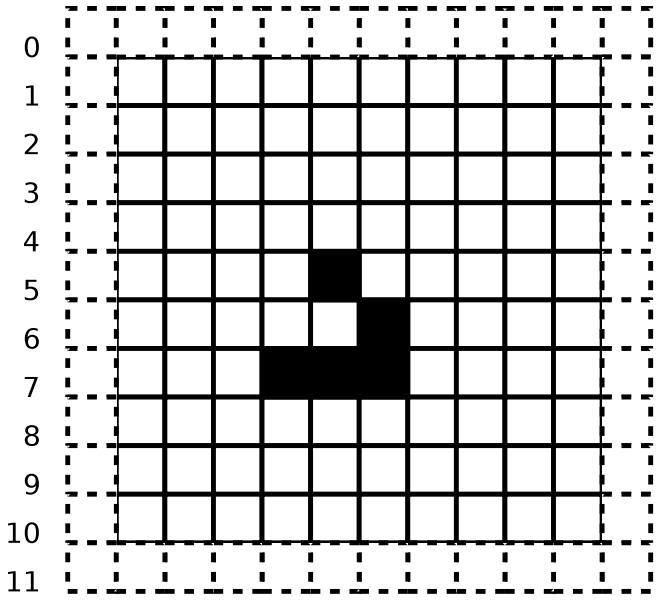

Figura 4.1: Tabuleiro de jogo da vida com dez linhas e dez colunas. As células pretas estão "vivas" e brancas "mortas". As células tracejadas são a borda do problema, onde não serão computadas as regras do jogo (sempre estarão "mortas"). O padrão desenhado, de nome "avião", parte do canto superior esquerdo e termina no canto inferior direito, conforme se passam as iterações.

Atualmente existem diversos programas que implementam o jogo da Vida. O mais notável é o Golly[1], que possui licença GPL. Além das regras originais o Golly implementa outras regras e tabuleiros, vem com a funcionalidade do chamado "*Hashlife*" (permite avançar nas iterações em qualquer velocidade, independente do equipamento) e possui uma enorme quantidade de exemplos. A figura 4.2 ilustra o que é possível fazer com o programa.

A implementação do presente trabalho é mais modesta, porém fixada nas 4 regras anteriores.

4.1 Divisão de domínio e paralelismo

A divisão de domínio utilizada para este problema visa explorar um problema comum: o de trocar bordas entre processadores sem que estas estejam contíguas em memória.

Dada uma matriz (tabuleiro) ela é armazenada em memória de alguma forma, e comumente é através de linearização por linha (*row major*) ou por coluna (*column major*). No primeiro caso, linearizada por linha, a seguinte matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

será armazenada em memória como:

[1]Site: golly.sourceforge.net

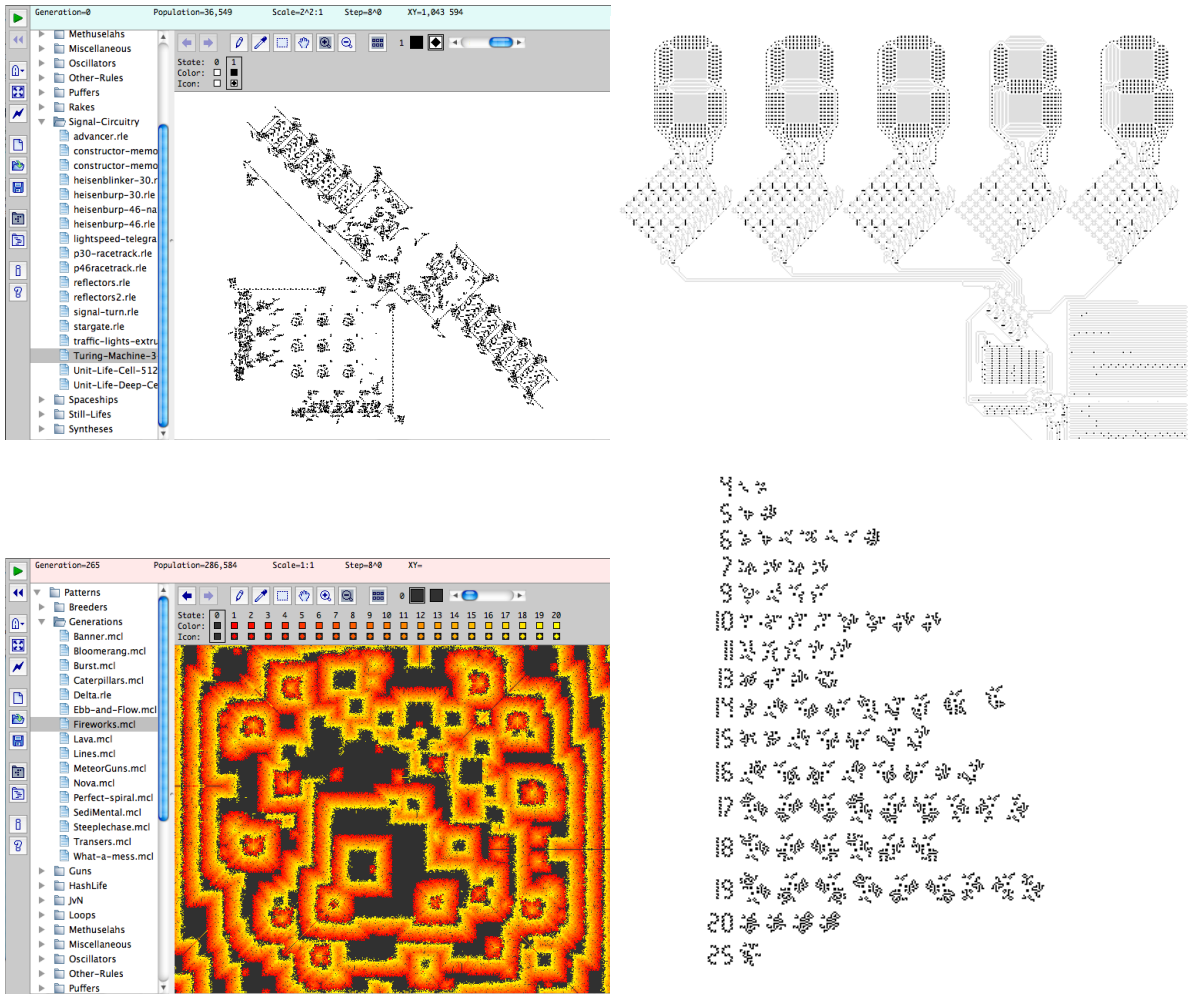

Figura 4.2: Exemplos de tabuleiros e formações do jogo da Vida no aplicativo Golly. Da esquerda para direita, de cima para baixo: Máquina de Turing; gerador de números primos (o número 43 está escrito); ambiente caótico; padrão estático (que não se modifica com o passar das iterações).

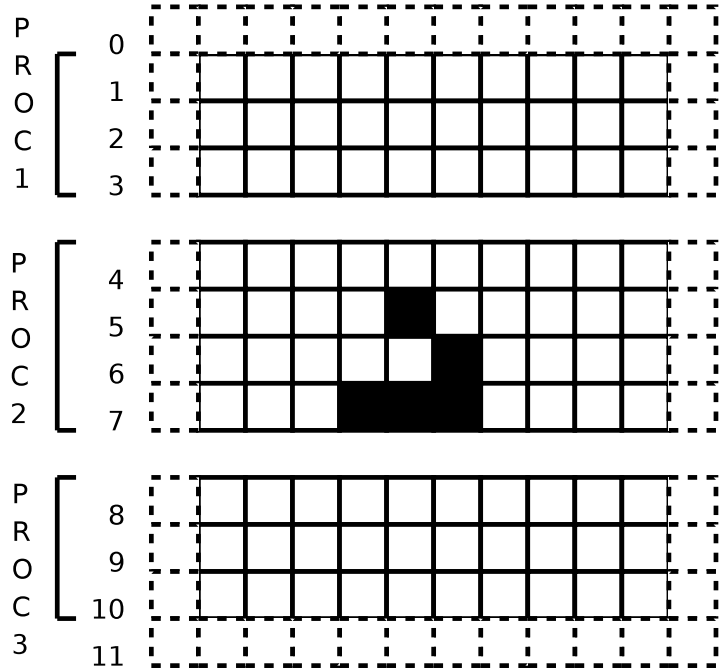

Figura 4.3: Divisão do tabuleiro entre os processadores do processamento paralelo.

1 2 3 4 5 6

Já o segundo caso, linearizado por coluna, a mesma matriz ficaria:

1 4 2 5 3 6

Em Fortran, linguagem em que este exercício foi implementado, a organização em memória é linearização por coluna.

A divisão do domínio do problema é realizada de forma a diminuir a linearização da borda a ser trocada, pois assim poderemos explorar ferramentas como tipos derivados em MPI e o poder da notação CAF. Pode parecer um contra-senso, pois dividindo-se o tabuleiro por colunas teremos a continuidade dos elementos a nosso favor, mas o objetivo é justamente trabalhar com as ferramentas que a biblioteca MPI e a extensão CAF fornecem para este caso.

Portanto o esquema é de dividir por linhas, ou seja, o tabuleiro é "cortado" dividindo o número de linhas igualmente entre os processadores, conforme a figura 4.3.

É importante ressaltar alguns termos importantes para tratamento do caso da referência a um conjunto de dados que possui características similares a uma linha em Fortran:

1. Dados contínuos (ou contíguos): são dados que em memória são ininterruptos, ou seja, entre dois elementos do conjunto de dados não há nenhum elemento entre eles;

2. Dados não-contínuos (ou não-contíguos): são dados que entre dois elementos do conjunto existe pelo menos um elemento, externo ao conjunto de dados, entre eles;

3. Espaçamento homogêneo: quando o número de elementos entre dois elementos em todo o conjunto de dados é constante (dados contínuos possui número de elementos entre eles igual a zero);

4. Espaçamento heterogêneo: quando o número de elementos entre dois elementos em todo o conjunto de dados não é constante.

O caso apresentado é do tipo 2, pois existem entre um elemento e outro vários elementos que não fazem parte do conjunto de dados a ser enviado, e também do tipo 3, pois há uma quantidade de elementos constante entre dois elementos da linha. Conforme explicado anteriormente, "*o jogo da vida possui quatro regras que dependem única e exclusivamente dos vizinhos de cada célula do tabuleiro*". Esta informação é muito importante para tratar a divisão de domínio.

As bordas de cada partição do tabuleiro possuem vizinhos que estão na memória do processador que as contém, mas não são processados por ele, pois são dominados por outro processador. Da mesma forma, estas bordas, que são processadas em um processador, são bordas vizinhas de apoio em outro processador. A figura 4.4 ilustra quem é proprietário de cada borda, e aquela que está na memória mas não é processada pelo processador, denominada daqui por diante de *ghostzone* (GZ).

A razão para existir as GZ é de permitir que quando um processador precisar de um dado (devido ao *stencil*) ele estará em sua memória local, diminuindo a necessidade de se acessar uma memória remota para ler o estado de um determinado vizinho de uma célula sendo processada. Estes dados não devem ser processados pelo processador que os possui dentro da GZ, pois é responsabilidade de outro processador atualizá-las. A atualização é realizada na sua totalidade, com o envio ou recebimento de toda a GZ, e não parte dela.

Como neste jogo cada iteração parte de um estado do tabuleiro para se chegar a outro, com aplicação das quatro regras, partes deste tabuleiro serão processadas de forma independente, porém as tais GZ estarão com dados desatualizados, surgindo a necessidade de se atualizar estas GZ com dados mais novos. A comunicação entre processadores, portanto, é necessária, e se utilizaram as operações descritas em MPI CBL, MPI CUL e CAF para realizar estas comunicações. Além da comunicação é necessária a consistência entre a situação do tabuleiro e a situação das GZ, e operações para assegurar o ordenamento do processamento entre as instâncias (inerentes ou não as operações de comunicação) são utilizadas a fim de assegurar a atualização das GZ.

Devido a divisão do tabuleiro ser realizada no eixo Y, a linha neste problema é que deve ser enviada a outro processador, e a informação não é contígua em memória. Existe, entre um elemento e outro da linha, posições de memória que não devem ser enviadas, o que caracteriza portanto uma comunicação de dados não contíguos.

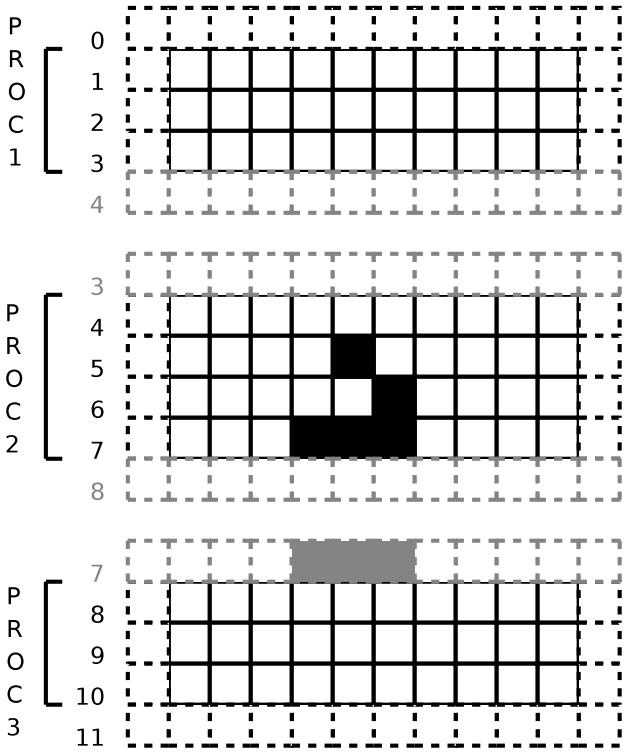

Figura 4.4: Ilustração com as bordas e os donos das mesmas. As bordas cinzas são *ghostzones*.

Não importando a escolha da biblioteca ou extensão à linguagem utilizada para apoio ao paralelismo é inerente ao problema o tratamento de dados não contíguos em memória para a operação em questão. Para a utilização de MPI foi necessário tratar esta falta de continuidade, conforme colocado a seguir. Já para o CAF a notação implícita do Fortran, na atribuição, foi utilizada.

4.2 Implementações

As implementações foram realizadas em Fortran 90, com uso de módulos, tipos derivados, mínima quantidade de argumentos passados nas chamadas às sub-rotinas, etc. Em linhas gerais o código está representado no diagrama 4.5.

Pode-se observar na figura que o programa possui diversas chamadas para iniciar e finalizar o trabalho (criar paralelismo, dividir as linhas entre os processadores, criar tabuleiro local, etc), mas todas elas são realizadas em paralelo por todas as instâncias. O processamento "pesado" fica por conta da parte central da figura – o avançar no tempo (aplicar as regras), trocar as bordas e imprimir o tabuleiro local e/ou global. As únicas operações que realizam E/S são as impressões do tabuleiro local e global, que podem ser "desligadas" (comentando as chamadas no código).

Conforme a esquematização as operações que realizam algum tipo de comunicação – seja explícita com operações de envio de bordas do MPI ou CAF, sejam implícitas com o uso de

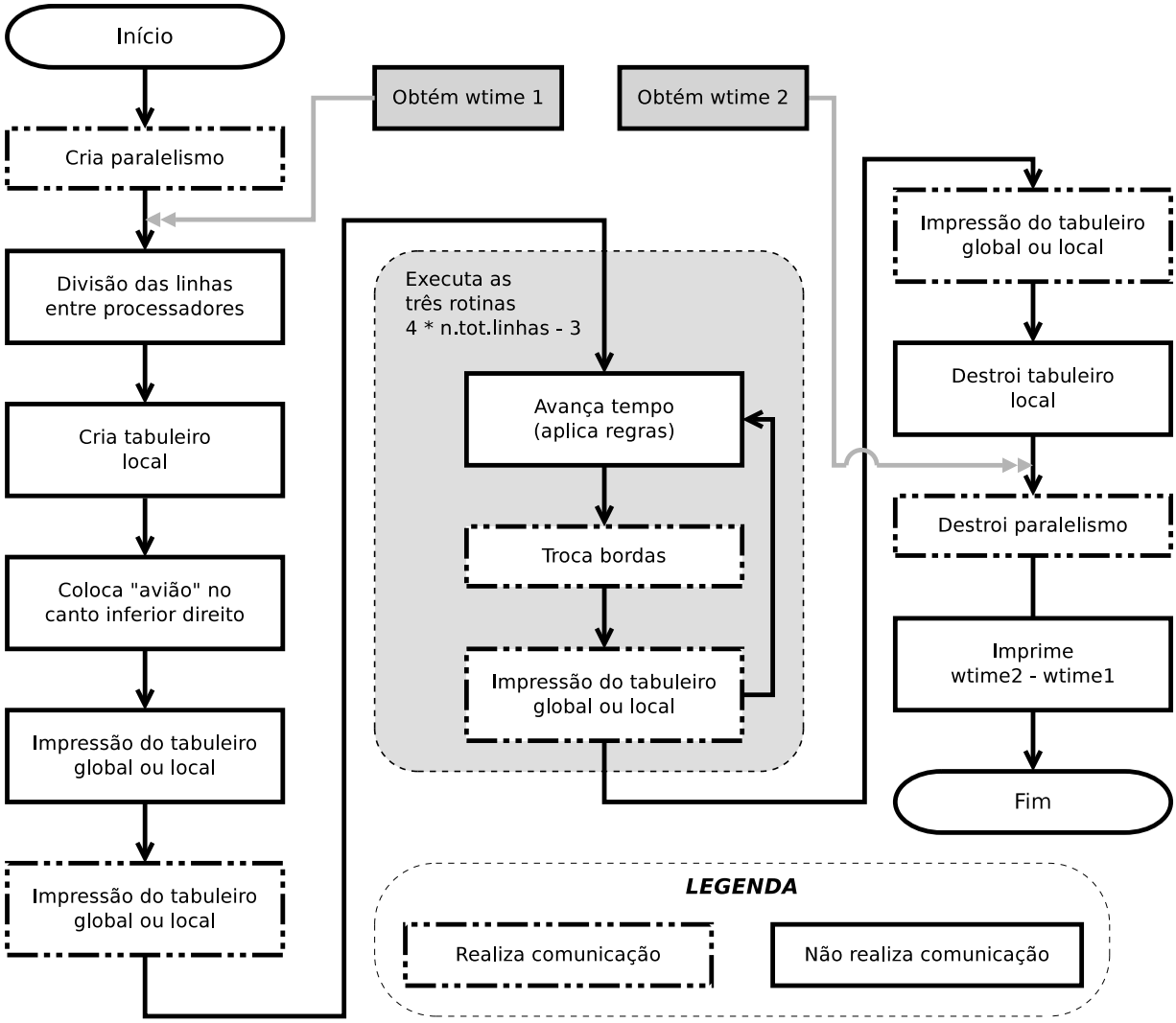

Figura 4.5: Diagrama das funções principais do jogo da vida.

inicialização ou finalização do paralelismo – são identificadas com linha hachurada. Portanto, se a impressão foi desligada as comunicações que ocorrem são a inicialização e finalização do paralelismo e a troca de bordas. Se supusermos que o tempo gasto pela inicialização e finalização for constante no número de processadores então a troca de bordas é a única operação de comunicação levada em consideração (inclusive, o mecanismo de medição de tempo utilizado exclui o tempo de inicialização e finalização do paralelismo).

Com relação ao esquema de descrição do tabuleiro, o mesmo é um tipo derivado, conforme o seguinte esquema:

```
– tabul é de um tipo com os elementos:
    – iPri: número da primeira linha do domínio (inteiro)
    – iUlt: número da última linha do domínio (inteiro)
    – jPri: número da primeira coluna do domínio (inteiro)
    – jUlt: número da última linha do domínio (inteiro)
    – tab: tabuleiro (logical, a célula está "viva" ou morta")
           com dimensões
        iUlt+1 – iPri–1 + 1 linhas
        jUlt+1 – jPri–1 + 1 colunas
```

A forma utilizada para atualizar este tabuleiro é de armazenar, em uma matriz temporária, o número de vizinhos de cada célula e aplicar as quatro regras baseando-se nesta tabela temporária, atualizando o tabuleiro. Desta forma – através da tabela de vizinhos – é assegurada a atualização correta do tabuleiro, não permitindo que o novo estado modifique o estado anterior.

Para a troca de bordas é necessário enviar uma linha em Fortran de um *rank* ao outro, ou de uma imagem a outra. Ao se utilizar as bibliotecas, foi necessário tratar a não continuidade dos elementos em memória. Para tanto se utilizou uma operação descrita no padrão MPI denominada MPI_Type_vector.

A operação constrói um objeto opaco (que o usuário não possui acesso a sua forma e tamanho, e é interno ao MPI), um tipo de dados derivado, que descreve uma replicação de um tipo de dados anterior em posições igualmente espaçadas. Em outras palavras, dada uma sequência igualmente espaçada de blocos de valores de um tipo de dados, a operação MPI_Type_vector constrói um novo tipo de dados baseado no número de blocos, no número de elementos de cada bloco, o espaçamento entre estes blocos de dados e o tipo de dado que existe em cada bloco. Esta operação é indicada para a linha em uma matriz em Fortran, onde o número de blocos é o número de colunas, o número de elementos por cada bloco é unitário, o espaçamento entre blocos é o número de linhas menos um e o tipo de dado de cada elemento do bloco é LOGICAL.

Em CAF não foi necessária a criação de um tipo derivado, pois realizar uma atribuição de uma linha de uma matriz para outra linha de outra matriz é direto (a notação do Fortran

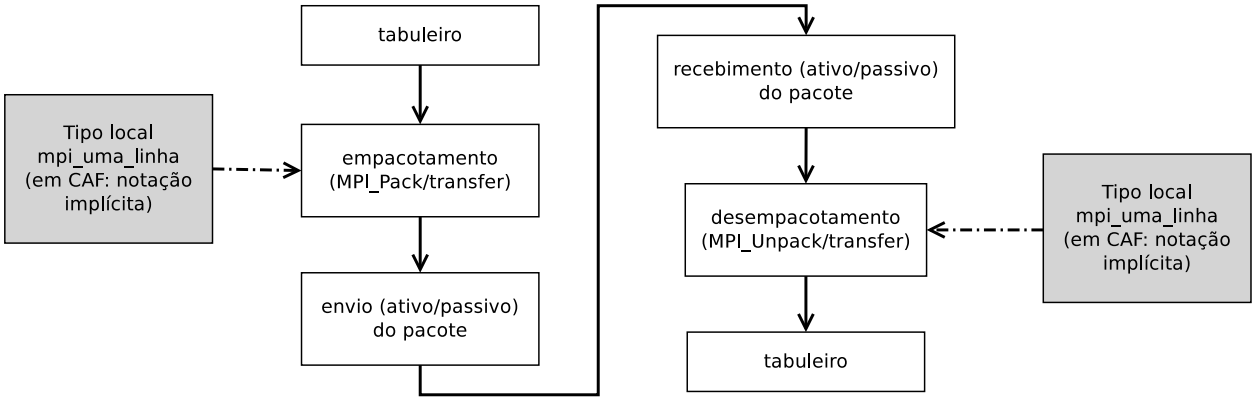

Figura 4.6: Ordem de operações para empacotamento dos dados. Em MPI pode-se utilizar a função MPI_Pack, já em CAF foi criado algo "similar" com o uso da função intrínseca $transfer$, transferindo os *bytes* de uma linha de valores lógicos para um vetor de caracteres.

permite isso):

```
! atribuição de uma linha de uma matriz para outra linha
! de outra matriz. As duas possuem o mesmo número de colunas
! independentemente do número de linhas

M1(4,:) = M2(7,:)

! outra forma, porém delimitando os limites da linha
M1(4,1:n) = M2(7,1:n)
```

Este caso é melhor representado no exemplo de atribuição do item 4.2.3.

Além da implementação da troca de bordas com tipo derivado ou de forma direta (como em CAF), foi criada outra forma, aquela que empacota dados, deixando-os assim contíguos em memória. A figura 4.6 coloca a ordem das operações para o empacotamento.

No empacotamento ocorre invariavelmente cópia de memória, introduzindo um aumento linear de tempo no processamento total. A seguir são colocados mais detalhes de cada implementação.

4.2.1 Implementação em MPI CBL

Em MPI CBL utilizou-se uma operação de envio e recebimento em uma mesma chamada (denominada $MPI\_Sendrecv$), para não permitir a ocorrência de dependências cíclicas e assim causar *deadlock*. Esta função é bloqueante, portanto não é necessário realizar operações de *wait* após o envio.

A seguir é apresentado um trecho do código que engloba a construção do tipo de dados, envio e recebimento na mesma função, e destruição do tipo de dados.

```
! construção do tipo que terá como handler mpi_uma_linha
```

```
! recebe count (número de blocos do tipo), 1 (tamanho do bloco)
! stride (espaço entre um bloco e o seguinte), mpi_logical
! (tipo de dado LOGICAL do MPI), mpi_uma_linha (handler para
! o tipo de dado criado) e retorna erro em ierr
call MPI_Type_vector(count, 1, stride, mpi_logical, &
  mpi_uma_linha, ierr)

! executa operação de criação do tipo
call MPI_Type_commit(mpi_uma_linha, ierr)

(...)

! - envia borda superior local para processador anterior
! atualizar sua GZ inferior
! - recebe borda inferior do processador posterior para
! atualizar GZ inferior local
call MPI_Sendrecv(&
  tabul%tab(tabul%iPri   ,tabul%jPri), 1, mpi_uma_linha, anterior,  10, &
  tabul%tab(tabul%iUlt+1,tabul%jPri), 1, mpi_uma_linha, posterior, 10, &
  mpi_comm_world, status, ierr)

(...)

! libera o tipo
call MPI_Type_free(mpi_uma_linha, ierr)
```

Também foi implementado o empacotamento de dados, via MPI_Pack, com a mesma forma de envio e recebimento, com devido ajuste nos argumentos para a rotina $MPI\_Sendrecv$.

4.2.2 Implementação em MPI CUL

Em MPI CUL foi definido que quem possui a borda que é GZ no processo remoto deve enviá-la (uso da operação MPI_Put). Da mesma forma foi criado um tipo para envio da linha. As janelas de comunicações, devido ao emprego da operação *put*, são criadas tendo como ponto de início o primeiro elemento da GZ (superior ou inferior) e seu tamanho se estende até o ultimo elemento, inclusive. Foram portanto duas janelas, uma para cada borda. Os elementos intermediários necessariamente fazem parte da janela, que é contínua (estão expostos para comunicação unilateral).

Foram escolhidas duas implementações de troca de bordas.

Sincronismo via barreira

A primeira implementação da troca de bordas foi realizada com barreiras. O sincronismo ocorre imediatamente antes e depois do envio. É colocado a seguir um exemplo de troca de bordas.

```
! similar a construção do tipo em MPI
call MPI_Type_vector(count, 1, stride, mpi_logical, &
  mpi_uma_linha, ierr)
call MPI_Type_commit(mpi_uma_linha, ierr)

(...)

! expõe região do tabuleiro da GZ inferior
! recebe como argumentos o início da janela, o tamanho,
! unidade de deslocamento, argumentos para criação da janela e
! o comunicador. Retorna o código de erro, via ierr
call MPI_Win_create(tabul%tab(tabul%iUlt+1,tabul%jPri), tamanho, 1, &
  MPI_INFO_NULL, MPI_COMM_WORLD, winGZi, ierr)

(...)

! Sincronismo antes do envio. Recebe zero (para o fence atuar como
! finalizador de uma "epoch"), a janela e retorna o código
! de erro via ierr
call MPI_Win_fence(0, winGZi, ierr)

! por convenção é enviado (MPI_Put) a primeira linha do domínio
! local para atualização da GZ inferior do processador anterior
! recebe como argumentos o primeiro elemento a ser enviado, o tamanho
! deste elemento, o tipo, o número do processador anterior, deslocamento
! do inicio da janela, tipo de dado a ser recebido no processador
! remoto, janela de comunicação. Retorna, via ierr, o código de erro
! da operação.
call MPI_Put(tabul%tab(tabul%iPri, tabul%jPri), 1, &
  mpi_uma_linha, anterior, 0, 1, mpi_uma_linha, winGZi, ierr)

(...)

! Sincronismo depois do envio, similar ao sincronismo anterior
call MPI_Win_fence(0, winGZi, ierr)

(...)

! dispõe da janela
call MPI_Win_free(winGZi, ierr)

! dispõe do tipo
call MPI_Type_free(mpi_uma_linha, ierr)
```

É importante ressaltar que o alinhamento entre o primeiro elemento da GZ e o primeiro elemento da janela de exposição da GZ, superior ou inferior, do tabuleiro auxilia na colocação, no lugar correto, do que se está sendo enviado via MPI_Put. As duas janelas –

GZ superior e inferior – se sobrepõem em grande parte do tabuleiro, e apesar de as duas operações de envio ocorrerem na mesma *epoch,* como são atualizadas linhas diferentes a execução correta da atualização é garantida. A semântica das operações deixa claro que estas operações são fora-de-ordem (as duas atualizações das GZs), porém cada uma atualiza uma região de memória diferente.

Existe, neste tipo de implementação, um problema relacionado ao tipo derivado. A divisão de linhas entre os processos pode não ser homogênea, havendo casos em que alguns processos ficarão com um número maior de linhas que outros. Por exemplo, ao se dividir um tabuleiro com 10 linhas entre três processos, dois ficarão com 3 e um com 4 linhas. Portanto, o tipo derivado nos processos com 3 linhas tem um salto de 4 elementos; já no processo com 4 linhas tem um salto de 5. Isso provoca inconsistência no envio (a escrita na janela remota ficará inconsistente). Foi utilizada, para este fim, a funcionalidade de *reshape* que o MPI fornece na função de envio por CUL. O tipo na origem é relativo aos saltos da origem, e o tipo do alvo relativo aos saltos do mesmo. Como consequência foi necessário um esforço maior de programação, o de inferir quantas linhas ficam para o processo superior e/ou inferior.

Sincronismo via *post/start/complete/wait*

Foram realizados também testes com implementação utilizando empacotamento. Desta forma o problema dos saltos entre elementos da linha é resolvido independentemente para cada processador, conforme figura 4.6.

A seguir é colocada a parte pertinente ao envio dos dados, empacotamento, criação de grupos, etc.

```
! construção do tipo
call MPI_Type_vector(count, 1, stride, mpi_logical, &
  mpi_uma_linha, ierr)
call MPI_Type_commit(mpi_uma_linha, ierr)
(...)

! expõe região do tabuleiro da GZ inferior
call MPI_Win_create(buffEi, tamanho, 1, &
  MPI_INFO_NULL, MPI_COMM_WORLD, winGZi, ierr)

! criação de grupo de processadores para acesso a janela
call MPI_Win_get_group(winGZi, group_world, ierr)
call MPI_Group_incl(group_world, 1, (/posterior/), gropoGZi, ierr)
call MPI_Group_free(group_world, ierr)

! expondo janela inferior
call MPI_Win_post(grupoGZi, 0, winGZi, ierr)
(...)
```

```
! comeca acesso a janela remota
call MPI_Win_start(grupoGZs, 0, winGZi, ierr)

! empacota e coloca na memoria remota a GZ
pos = 0
call MPI_Pack(tabul%tab(tabul%iPri, tabul%iPri), 1, mpi_uma_linha, &
  buffSs, pack_size, pos, mpi_comm_world, ierr)
call MPI_Put(buffSs, pack_size, MPI_PACKED, anterior, &
  int(0,MPI_ADDRESS_KIND), pack_size, &
  MPI_PACKED, winGZi, ierr)
(...)

! finaliza acesso a janela remota
call MPI_Win_complete(winGZi, ierr)
(...)

! fecha exposição da janela e desempacota
call MPI_Win_wait(winGZi, ierr)
pos = 0
call MPI_Unpack(buffEi, pack_size, pos, &
  tabul%tab(tabul%iUlt+1, tabul%jPri), 1, mpi_uma_linha, &
  mpi_comm_world, ierr)
(...)

! dispoe do grupo
call MPI_group_free(grupoGZi, ierr)

! dispõe da janela
call MPI_Win_free(winGZi, ierr)

! dispõe do tipo
call MPI_Type_free(mpi_uma_linha, ierr)
```

A implementação realiza o *post*, *start*, *complete* e *wait* em momento oportuno (inicialização, antes de acesso a GZ, etc).

4.2.3 Implementação em CAF

Já em CAF a troca é direta e simples como no exemplo a seguir.

```
- Pinferior é o índice da imagem inferior

! sincronizando antes da troca a fim de assegurar
! que o processo remoto terminou o processamento
sync all

! atualizando GZ inferior com a variável tabul do processo
```

```
! inferior e a primeira linha completa do elemento tab
tabul%tab(tabul%iUlt+1,tabul%jPri-1:tabul%jUlt+1) = &
        tabul[Pinferior]%tab(tabul[Pinferior]%iPri,
    tabul[Pinferior]%jPri-1:tabul[Pinferior]%jUlt+1)
```

É importante observar que se utiliza largamente a referência a variáveis de outro processador dentro da construção da atribuição. Em comparação a trechos de código similares com as bibliotecas MPI CBL e CUL em CAF a troca é escrita de forma diminuta e direta.

Não contando as implicações que a referência a uma linha em Fortran podem prejudicar no desempenho, o fato da falta de alinhamento de memória não é característica de primeira importância a ser levada em conta para a operação da transferência da GZ.

Foi realizada outra implementação com uso da função *sync images*( ) e "empacotamento" via $transfer$, mas tratando cada borda de maneira independente (assim como em MPI) com sincronismo independente para cada uma.

4.3 Testes de desempenho com CUL

Neste item iremos descrever os testes realizados com a comunicação unilateral no jogo da Vida. Estes testes possuem o objetivo de analisar os impactos (bons ou ruins) no desempenho computacional que a CUL gera no programa.

4.3.1 Ambientes e compiladores para o jogo da Vida

A análise dos impactos foi baseada no tempo de processamento, uma medida comum para análise de desempenho de programas paralelos. Esta medida foi realizada com a função intrínseca do Fortran *system_clock*. Mas, como esta rotina envolve a consulta ao relógio da máquina, e este pode não ser muito acurado (pode ocorrer um fenômeno chamado "*clock drift*", onde este pode atrasar ou adiantar com relação a outro), foram tomados alguns cuidados.

O primeiro foi o monitoramento da carga de processamento dos equipamentos. Todos estavam dedicados (ou livres) quando da execução dos testes, portanto não houve concorrência de CPU e comunicação com outros processos em execução.

Outro cuidado foi de analisar, através de algumas execuções sequenciais do jogo, o comportamento da medição de tempo via intrínseca *system_clock*. Os resultados desta medição estão no item 4.3.2.

Os ambientes utilizados foram três: dois *clusters* de computadores tipo PC e um equipamento da Cray Inc. A tabela 4.1 lista as características principais de cada um.

Nome	Local	nós	núcleos/nó	Modelo CPU	Conectividade	Memória/Nó
BREEZE	IAG/USP	4	2 (8 total)	Intel Core2Duo E8500 64 bits (Wolfdale) 3.16ghz / 6mb *cache*	Gigabit ethernet	2gb (1gb/núcleo)
SONNY1	IAG/USP	16	2 (32 total)	Intel Xeon "Pentium 4" 32 bits (Prestonia) 2.66ghz / 512kb *cache*	Gigabit ethernet	1gb (512mb/núcleo)
CROW	Cray Inc.	16	8 (128 total)	AMD Opteron 8000 64 bits (Shanghai) 2.7ghz / 6mb *cache*	Cray Seastar2+	32gb (4gb/núcleo)

Tabela 4.1: Descrição dos equipamentos utilizados para os testes de desempenho.

Em termos de *"pilha de software"* a tabela 4.2 descreve o sistema operacional, compiladores e biblioteca MPI utilizada em cada equipamento.

Nome	Sistema operacional	Compilador e versão	Biblioteca MPI
BREEZE	Ubuntu 9.04	Intel Suite 11.1 G95 0.9 (snapshot)	OpenMPI 1.3.3
SONNY1	Ubuntu 8.04	Intel Suite 11.1 G95 0.9 (snapshot)	OpenMPI 1.3.3
CROW	*Cray Linux Environment* (CLE)	PGI 10.4 Pathscale 3.2 CCE 7.2.3 GCC 4.2.0	MPICH2

Tabela 4.2: Descrição da "*pilha de software*" de cada equipamento utilizado.

Os compiladores que possuem suporte a CAF são o G95 e o CCE. No caso do G95 é necessário o ambiente de execução *cocon*, que possui código fechado[2]. O *cocon* funciona como um gerenciador de comunicação e console, controlando o número de imagens e a distribuição dos trabalhos. O mesmo G95 também possui a funcionalidade de dividir as instâncias em um ambiente de memória compartilhada sem uso do *cocon*.

Foram realizadas execuções com intuito de verificar o quanto a execução em paralela é mais rápida que a serial. Consiste em executar o código de forma serial (com um processador) e de forma paralela (com dois ou mais processadores), geralmente crescendo linearmente no número de processadores. Para o *speed-up* foi utilizada a relação da fórmula 4.1:

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

[2]O *cocon* é o *shareware* e é limitado para execução com mais de 5 processadores sem a compra de uma licença de uso. Mas o autor do programa, Andy Vaught, encarecidamente forneceu uma licença, sem custo, para ser utilizada neste trabalho. O autor desta dissertação gostaria de agradecer ao Andy.

onde p é o número de instâncias ou processadores utilizados, T é o tempo gasto para processamento e S é o *speed-up*. Em um programa perfeitamente paralelizado, $S_p = p$. Se, durante a execução, o tempo de processamento sofrer alguma penalidade causada por um agente externo (comunicação, acesso a memória, etc) o seu *speed-up* será mais distante do ideal. Esta penalidade é denominada de "*overhead*".

4.3.2 Resultados com o jogo da Vida

O paralelismo do jogo da Vida auxilia na transferência de linhas de uma matriz em Fortran. O domínio é dividido entre as instâncias do paralelismo, e durante a execução as vizinhanças de uma partição do domínio enviam e recebem linhas.

Para os testes em todos os equipamentos foi utilizada uma grade de 768 por 768 pontos, excluindo as bordas externas. A escrita dos resultados foi desabilitada, mas esta teve importante papel para verificação da corretude das implementações. Assim o impacto da escrita em disco, e antes disso, o processo de aglutinar todos os domínios em um só, foi eliminado, e o tempo medido é somente de processamento.

Primeiramente foi realizada uma análise do comportamento da medição de tempo do *system_clock*. O jogo da Vida foi executado dez vezes em cada equipamento, para cada compilador utilizado. A tabela 4.3 coloca o valor da média das medições e do desvio padrão para cada combinação de equipamento de compilador.

Equipamento	Compilador	Média (s)	Desvio padrão (s)
CROW	CCE	$166,00$	$0,04$
CROW	PGI	$180,56$	$0,02$
CROW	Pathscale	$233,17$	$0,01$
CROW	GCC	$189,56$	$0,03$
BREEZE	Intel	$140,68$	$0,05$
BREEZE	G95	$985,63$	$3,04$
SONNY1	Intel	$223,06$	$5,25$
SONNY1	G95	$3290,60$	$58,26$

Tabela 4.3: Análise do comportamento do erro da medição de tempo pela intrínseca *system_clock* nos equipamentos e compiladores utilizados.

De acordo com a tabela as medições de tempo possuem desvio padrão muito pequeno, mostrando pouca flutuação nas medições.

Foram feitas algumas implementações das transferências, sumarizadas na tabela 4.4. O objetivo foi de explorar três áreas: a troca da comunicação bilateral pela unilateral; o envio de uma região contínua de memória (*buffer*) após a cópia da linha (quando for o caso); diferentes tipos de sincronismo.

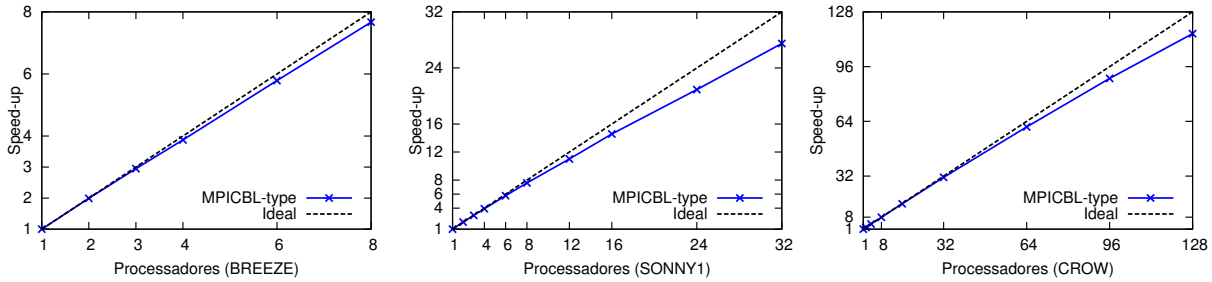

Figura 4.7: Curvas de *speed-up* da execução de controle (MPICBL-type) para os equipamentos BREEZE, SONNY1 e CROW.

Nome	Comunicação	Tratamento da linha	Sincronismo
CAF-direct	Notação CAF	Acesso direto com notação Fortran	Barreira
CAF-pack	Notação CAF	Cópia para buffer (uso do $transfer$)	Barreira
CAF-pack-syim	Notação CAF	Cópia para buffer (uso do $transfer$)	Grupo de imagens
MPICBL-type	MPI CBL	Tipo derivado	Bloqueante
MPICUL-type-fence	MPI CUL	Tipo derivado	Barreira
MPICUL-pack-pscw	MPI CUL	Empacotamento (com MPI_Pack)	Controle refinado

Tabela 4.4: Descrição dos experimentos com o jogo da Vida.

Foi suposto que a implementação MPICBL-type é a de controle, pois possui comunicação bloqueante, uso de tipo derivado e MPI CBL. A análise dos benefícios será feita comparando o desempenho das implementações com esta.

Comportamento do desempenho da implementação de controle

A implementação de controle foi executada, e ela fornece subsídios para analisar as características do equipamento. Os gráficos na figura 4.7 mostram as curvas de *speed-up* do controle para BREEZE, SONNY1 e CROW. Pode-se observar nos gráficos que o *speed-up* é próximo do ideal, mas com algumas flutuações. A máquina CROW é a que possui a melhor curva, e chega a 128 processadores com *speed-up* sustentável, corroborando com a especificação da rede da máquina (rede proprietária de altíssima velocidade e baixa latência). Quanto as outras curvas (BREEZE e SONNY1) são similares (pelo menos até 8 processadores).

Ainda sobre a CROW o gráfico 4.8 mostra o *speed-up* da implementação de controle com todos os compiladores disponíveis. Pode-se observar que somente o GCC não acompanha os outros compiladores, em termos de *speed-up*. Os outros é praticamente impossível diferenciar

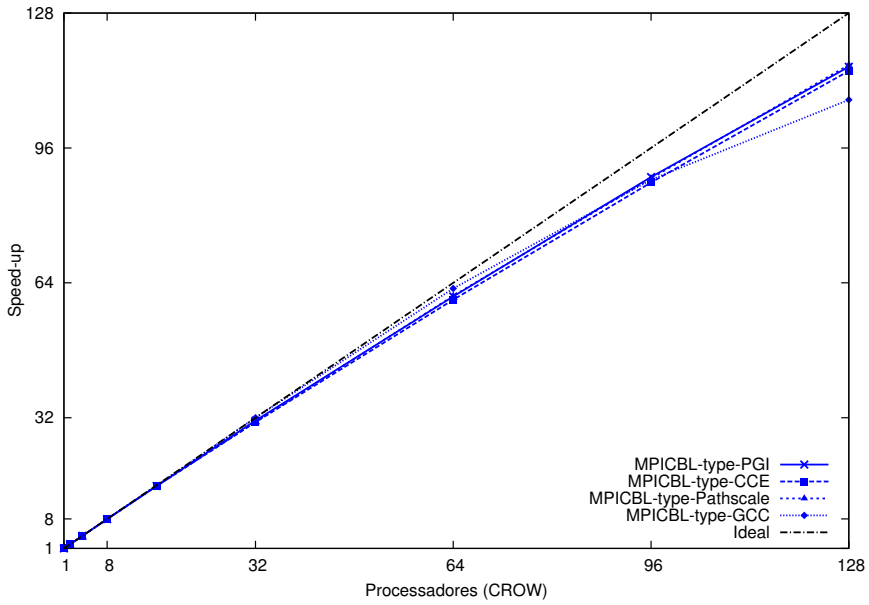

Figura 4.8: Curvas de *speed-up* da execução de controle (MPICBL-type) para o CROW com todos os quatro compiladores.

as curvas no gráfico, possuindo portanto *speed-ups* similares, quase idênticos.

Comparação entre o controle e implementação MPICUL-type-fence (MPI CUL)

As implementações MPICUL-type-fence e MPICUL-pack-pscw foram executadas nas mesmas condições da execução de controle, a MPICBL-type.

Primeiramente são apresentados os resultados relativos aos *clusters* BREEZE e SONNY1. Os gráficos da figura 4.9 ilustram a execução nas duas máquinas. É interessante notar que a implementação MPICUL-type-fence não escala tão bem como a controle, nos dois *clusters*. Na BREEZE as duas curvas começaram a se distanciar a partir de quatro processadores, e na SONNY1 a partir de seis processadores.

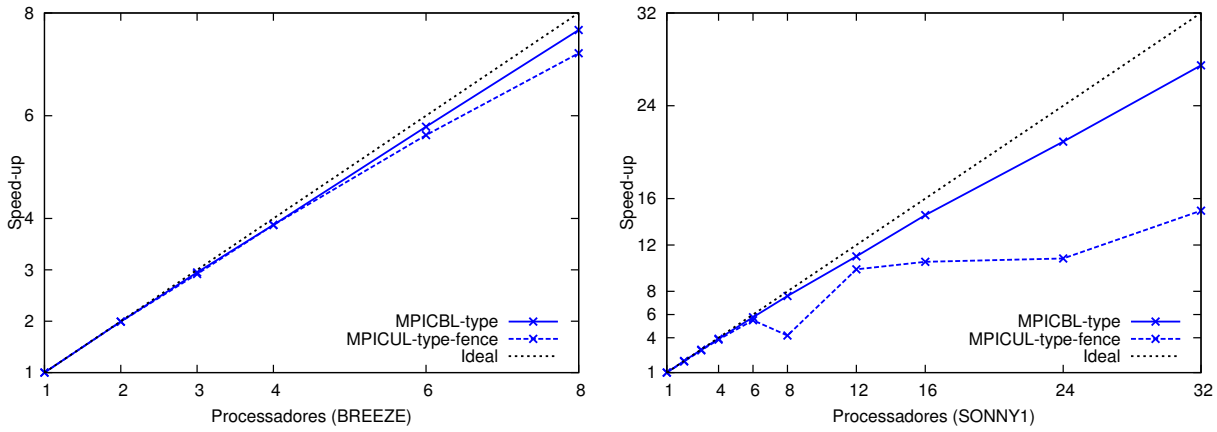

Figura 4.9: Curvas de *speed-up* da implementação MPICUL-type-fence e MPICBL-type na BREEZE e SONNY1.

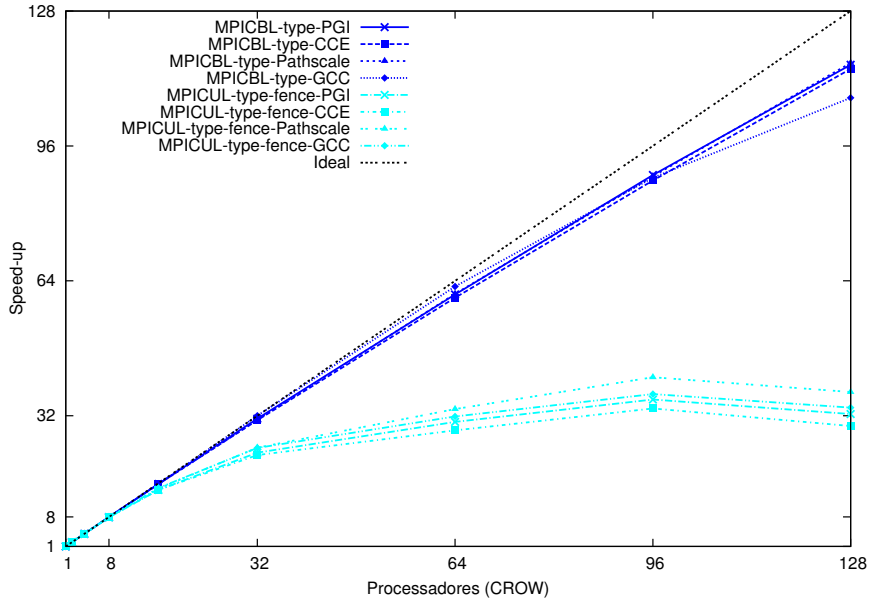

Figura 4.10: Curvas de *speed-up* da implementação MPICUL-type-fence e MPICBL-type na CROW.

De maneira semelhante o gráfico da figura 4.10 ilustra a execução no CROW. Os experimentos com MPI CUL, com linha mais clara, não escalam na mesma magnitude que o controle, independente do compilador utilizado. Vale ressaltar que é utilizado o tipo derivado, o que não acarreta em cópia de dados no nível da semântica (na implementação não se sabe nada a respeito, ela é livre para utilizar *buffers*), auxiliando em uma (possível) transferência direta de dados (RDMA).

A implementação MPI-2-pack-pscw foi executada nos *clusters*, e suas curvas de *speed-up* estão nos gráficos da figura 4.11 (confrontando-as com a implementação MPICUL-type-fence). Nota-se que, mesmo trocando o sincronismo e empacotando os dados, a implementação de controle possui melhor desempenho. Mas esta conseguiu melhor resultado que a

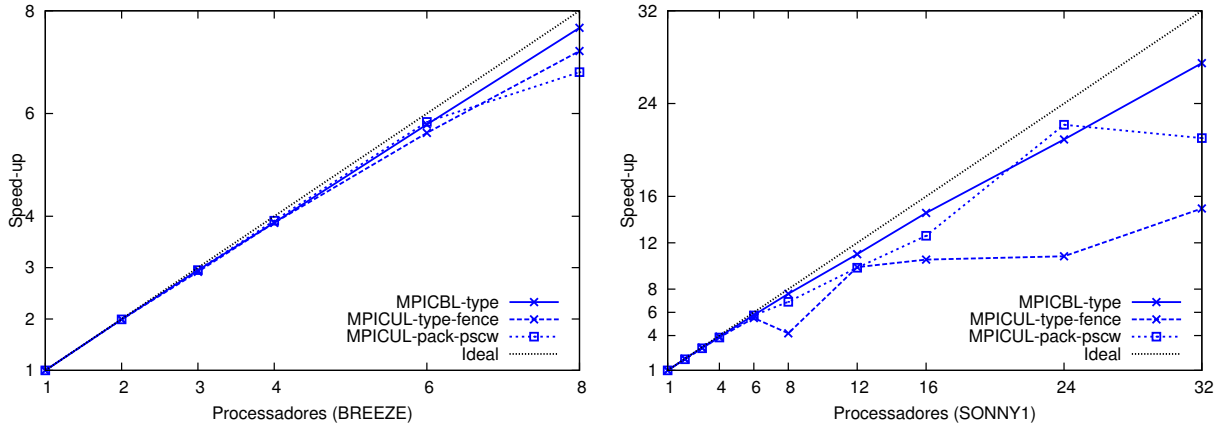

Figura 4.11: Curvas de *speed-up* das implementações MPICUL-pack-pscw, MPICUL-type-fence e MPICBL-type na BREEZE e SONNY1.

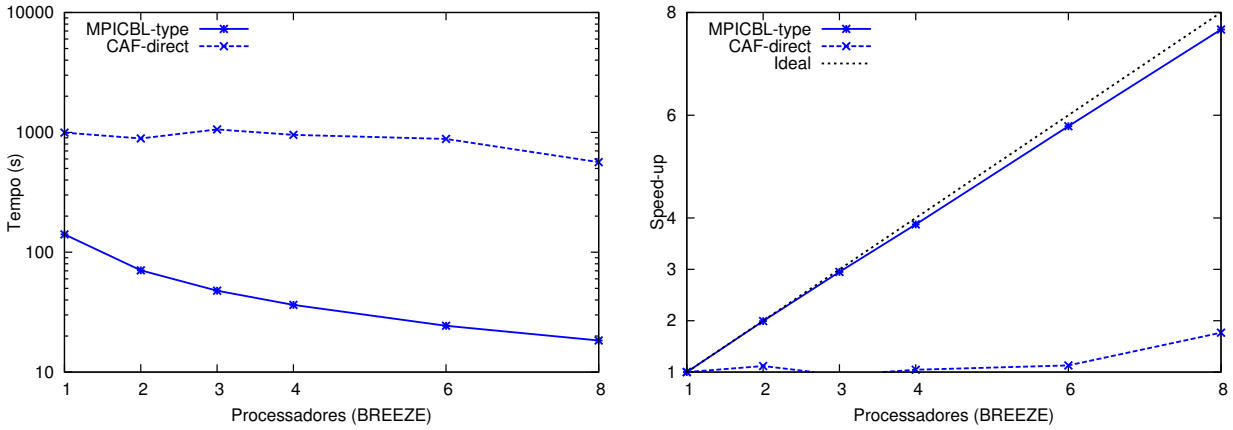

Figura 4.12: Curvas do tempo de execução e de *speed-up* para a implementação CAF-direct na BREEZE, com uso do G95 e *cocon* (notar escala logarítmica no eixo das ordenadas no gráfico de tempo de processamento).

implementação com sincronismo por barreiras. Na BREEZE, com seis processadores, a implementação com uso de exposição de janelas teve resultado melhor que a controle. Já na SONNY1 com 24 processadores o desempenho foi melhor que o controle.

Na máquina CROW não houve sucesso na execução da implementação MPICUL-pack-pscw. O programa se apresentou instável e não gerou resultados corretos com mais de 2 processadores.

Comparação entre o controle e implementações utilizando CAF

As implementações CAF-direct, CAF-pack e CAF-pack-syim foram executadas e confrontadas com o controle.

Recapitulando, os compiladores Fortran com suporte a *coarrays* são o G95 e o CCE. O G95 é utilizado em conjunto com o console de execução *cocon* (*Coarray* Console), e o CCE é um dos compiladores da CROW, máquina da Cray Inc. Os dois são resilientes com o padrão CAF de 2008.

O G95 se mostrou incapaz de escalar um código com uso de *coarrays*. O gráfico da esquerda da figura 4.12 mostram mínima diminuição do tempo de processamento na implementação CAF-direct, o que reflete uma curva de *speed-up* ruim.

De acordo com o que foi observado na BREEZE pode-se concluir que o G95 não é um compilador eficiente[3]. Ele pode ser muito interessante para levar ao ambiente de programação baseado em *softwares* livres o *coarray* FORTRAN, pode ser utilizado em várias plataformas (x86 32 e 64 bits, linux ou Mac OSX) e pode ter uma implementação bem completa do padrão CAF, mas definitivamente não é eficiente.

No CROW a experiência com CAF mostrou alguns pontos interessantes. O primeiro

[3]A BREEZE é um equipamento mais novo e mais rápido que a SONNY1. Como o desempenho foi muito ruim na BREEZE, se supõe que o desempenho na SONNY1 também será ruim, ou pior.

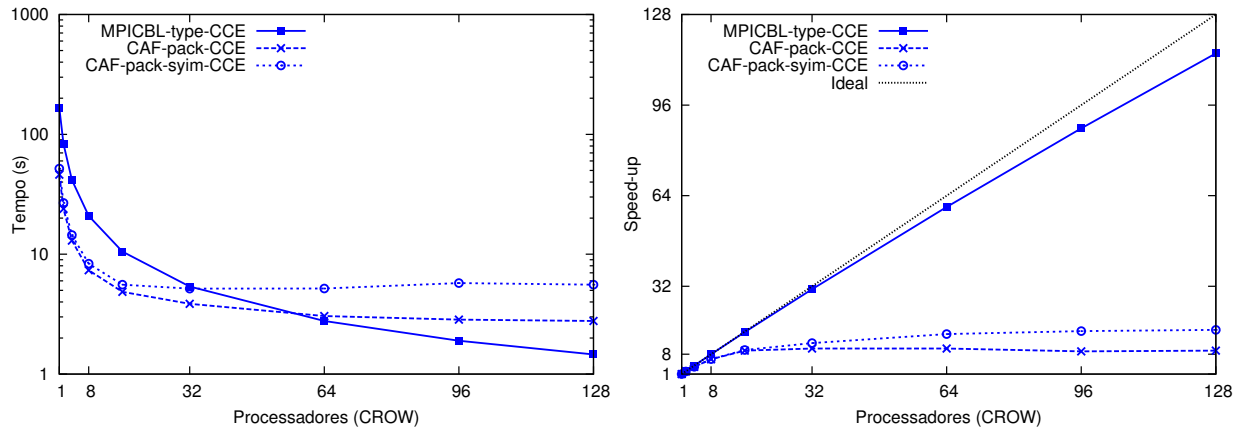

Figura 4.13: Curvas do tempo de processamento e *speed-up* para as implementações CAF-pack, CAF-pack-syim e de controle na CROW, com uso do CCE.

é o desempenho absoluto do compilador. Os gráficos da figura 4.13 mostram que o CCE, confrontando implementações do controle, CAF-pack e CAF-pack-syim do jogo da Vida, é mais rápido quando não há envolvimento de rotinas MPI (o CAF-pack e CAF-pack-syim não possuem chamadas a rotinas do MPI). Mas o *speed-up* mostrou que a implementação de controle escala melhor, principalmente depois de 8 processadores, em comparação as implementações utilizando CAF.

Outro ponto é relacionado ao sincronismo. Quando este ocorre através de grupo de imagens (*sync images*) ele gera lentidão no código, o que é observado depois de 8 processadores.

Outro ponto intrigante foi a execução do CAF-direct. O código, conforme colocado anteriormente, teve sua rotina de transferência de bordas modificada para realizar estas comunicações através da notação CAF. As implementações CAF-pack e CAF-pack-syim usam o $transfer$ do Fortran para copiar uma linha para uma área de memória que possui atributo de *coarray*, de tamanho fixo e igual em todas as imagens. Já a implementação CAF-direct possui uma variável (*tabul*) que é um *coarray*, se utiliza de um tipo derivado (tabuleiro) e de um elemento deste tipo que possui tamanho dinâmico (*tab*). Esta construção é válida, de acordo com o padrão CAF.

O G95 produz resultados corretos com a implementação CAF-direct. Mas o CCE não realiza transferências corretas das linhas. Somente as implementações CAF-pack e CAF-pack-syim foram executadas no CROW.

4.4 Observações e impressões

O experimento, de grade estruturada, homogênea e com bordas semelhantes depois da divisão do domínio facilita uma vez que as informações sobre a memória remota são possíveis de serem inferidas sem comunicação. Auxilia também na localização dos dados pois a indexação da grade é global, portanto se uma borda superior é de índice I a inferior do processo

superior é de índice $I - 1$. Todas as bordas são de mesmo tamanho, mas por se tratar de uma linha em Fortran as mesmas não são contíguas em memória, então se fez necessário tratar esta não-continuidade.

O que foi constatado nas análises das semânticas anteriormente colocadas, a necessidade de se conhecer *a priori* todos os argumentos para a comunicação, foram percebidas nas implementações do jogo da vida. Em CAF essa dificuldade foi sanada com o acesso para coletar os atributos dos tabuleiros vizinhos utilizando-se da sintaxe, na atribuição. A não-continuidade de uma linha em Fortran foi resolvida com uso da atribuição vetorial da própria linguagem. Conclui-se que a linguagem facilitou muito nesse caso.

Já para o MPI CUL foi necessário tratar a linha, com a criação de um tipo derivado. Mas este tipo vale somente para o processo corrente, não para os vizinhos, já que a divisão do domínio pode ter designado aos vizinhos um número de linhas diferentes (o salto entre dois elementos de uma mesma linha depende do número de linhas). Duas soluções foram colocadas: o empacotamento dos dados, ficando a cargo de cada vizinho desempacotar e colocar no lugar correto do tabuleiro, com uso do seu tipo derivado; a inferência de um tipo derivado para cada vizinho e utilização do *"reshape"* que a biblioteca do MPI permite. No primeiro caso há cópia de memória, mas não é necessário saber qual é o tipo remoto, e o segundo é necessário saber o tipo remoto, sem cópia de memória.

Tanto em MPI CUL quanto em CAF foi implementada outra forma de sincronismo, por grupo de processos (ou imagens, no caso do CAF). O esforço no caso do MPI é grande, pois é necessário criar grupos de processos, enquanto CAF é tão simples quanto uma barreira, bastando identificar os processos envolvidos.

Este esforço maior em CUL não resultou em melhorias, pelo menos em termos da programação. Isso porque o desejado é a simples troca de bordas entre dois vizinhos, e as funções empregadas permitem muito mais que controlar um simples sincronismo de dois processos. A barreira é global, algo complicado quando se utilizar muitos processadores; o sincronismo por grupos de processos é complicado ao programador. A solução é buscar algo mais simples, para tarefas simples, mas o MPI CUL não parece ter, pelo menos no que diz respeito a sincronismo ativo.

No que diz respeito ao esforço geral da programação a implementação do MPI CUL foi a mais complicada, principalmente quando o sincronismo é via *post/start/complete/wait*. E também a questão do *reshape* pode ser um problema muito complicado, devido ao fato de se conhecer o mapa de memória do vizinho, duplicando (pelo menos) os esforços para a programação da divisão de domínio. No caso do jogo da Vida foram 4 linhas a mais, mas em divisões mais complicadas pode ser muitas linhas a mais. Já em CAF a redução não foi de 40% como relatado por outros autores, mas foi significativa. Os exemplos colocados anteriormente ilustram essa redução.

Analisando os resultados das execuções do jogo da Vida, com diversas implementações de transferências de linhas, é possível observar que a mudança de semântica não ajudou na

melhoria do desempenho. O que corrobora com esta afirmação são as curvas de *speed-up*, onde todas as implementações utilizando MPI CUL e CAF apresentam *speed-up* pior que a execução com MPI CBL, em todas as plataformas utilizadas (dois *clusters* e um equipamento da Cray Inc.).

É importante ressaltar que a piora é para o caso do jogo da Vida com a troca da implementação de MPI CBL para MPI CUL ou CAF, com mínima (ou nenhuma) modificação nas rotinas de aplicação das regras, divisão do domínio, etc[4]. É mais simples programar em CAF – e este mostrou ótimo desempenho absoluto, em comparação ao MPI CBL, para até 8 processadores, número de núcleos em um nó do CROW – mas este não possui um *speed-up* quase ideal, como o MPI CBL. Já em MPI CUL é mais complicado programar (variáveis expostas em janelas, grupos de processos, tipo derivado refletindo a memória remota, etc) e não sustenta um bom desempenho, mesmo em um ambiente proprietário com rede rápida como o Cray.

[4]Em outras palavras é complicado afirmar "a mudança da semântica piorou o desempenho". A culpa da piora pode ser devida a implementação utilizada neste trabalho, e pode haver outra qualquer que não apresente este resultado.

Capítulo 5

O modelo OLAM

Neste capítulo vamos descrever o modelo OLAM e a implementação de comunicação de bordas através do uso de funções do MPI CUL. Os resultados de desempenho e curvas de *speed-up* poderão ser encontrados no item 5.3.

O modelo OLAM (*Ocean Land Atmospheric Model*), desenvolvido por Walko e Avissar (Walko e Avissar [2008a], [Walko e Avissar 2008b]), possui características que o diferencia de outros modelos numéricos geofísicos. O OLAM possui como grande diferencial o fato de ser baseado no método de volumes finitos e de possuir uma grade não estruturada (formada por triângulos). Esta grade, de escala global, pode também possuir refinamentos locais, mesmo assim não caracterizando outra grade.

O OLAM possui características interessantes para os objetivos deste trabalho: grades não estruturadas e emprego de paralelismo para resolver as equações e parametrizações do modelo. O paralelismo para problemas onde se discretiza o espaço através de uma grade de pontos geralmente é através da divisão da mesma com distribuição de partições desta grade entre os processadores. A divisão, portanto, é horizontal, e as colunas verticais de cada ponto desta grade não são divididas.

O presente trabalho focalizou-se no estudo das operações que envolvessem dinâmica e processos que empregassem consulta a valores em pontos vizinhos, horizontais, uns dos outros. O particionamento da grade horizontal gera regiões de fronteira que terão tratamento essencialmente parecido daquele descrito no jogo da vida (regiões como as *ghostzones*).

O OLAM possui uma codificação em FORTRAN 90/2003 em constante mudança. Abaixo segue a versão de cada estágio do código do modelo, e aquela que foi escolhida como a versão a ser estudada e modificada neste trabalho:

- 0: versão interna ao desenvolvimento, possuía duas grades cartesianas (uma para cada hemisfério, com ponto de projeção nos polos contrários). Possuía sérios problemas na união das duas grades;
- 1: versão interna ao desenvolvimento, a união das duas grades é formada por triângulos. Também possuía problemas;

- 2.8.1: primeira versão distribuída (de forma *ad-hoc*) entre pesquisadores. Esta versão já possui uma grade formada por triângulos, baseada em um icosaedro, sendo totalmente serial, com física instável e poucas funcionalidades;

- 2.11.1: versão com física mais avançada, estável e código mais estruturado. Chegou a ser operacionalizada. Ainda é serial;

- 2.14: versão com paralelismo rudimentar, dividindo o globo em dois (no equador, provavelmente);

- 3.0: primeira versão totalmente paralela. Física e dinâmica reestruturadas, código mais limpo e comentado. Boa parte da descrição do modelo é baseada nesta versão;

- 3.2: versão com correções de *bugs* da 3.0. Esta versão é a escolhida para os trabalhos de modificação do código a serem desenvolvidos;

- 3.3: pequenas correções da versão 3.2.

A seguir é colocada uma descrição do modelo, com sua grade, a divisão da mesma para o paralelismo e a sequência de operações que o modelo executa, incluindo uma descrição do paralelismo atual, através de MPI CBL. Logo após são apresentadas as implementações com MPI CUL e dois tipos de sincronismo: via barreira e via controle refinado (*post*/*start*/*complete*/*wait*). Este capítulo é finalizado com observações relativas à mudança do paralelismo, esforços de programação, etc.

5.1 Descrição do modelo

A seguir são descritos diversos aspectos do OLAM relacionados ao âmbito do presente trabalho. Para demais informações, principalmente as relativas a parte meteorológica do mesmo, se sugere consultar a seguinte bibliografia: Walko e Avissar [2008a], [Walko e Avissar 2008b] e Silva *et al.* [2009], sendo este último uma descrição mas abrangente do modelo do ponto de vista de um usuário, e não do desenvolvedor.

5.1.1 A grade

A grade do OLAM é de escala global. Baseia-se na subdivisão de cada face de um icosaedro, ilustrado na figura 5.1. O icosaedro é um sólido com vinte faces, sendo cada face um triângulo. Partindo deste sólido a grade é construída subdividindo cada face. O número de subdivisões é um parâmetro que é fornecido ao programa e depende da configuração do modelo, e a quantidade de processamento é proporcional ao número de subdivisões.

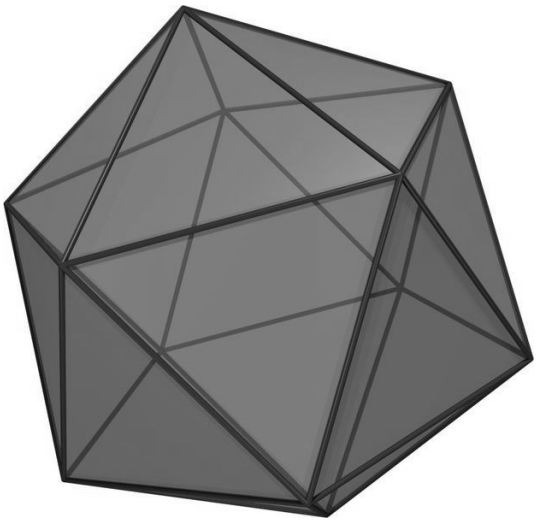

Figura 5.1: Um icosaedro.

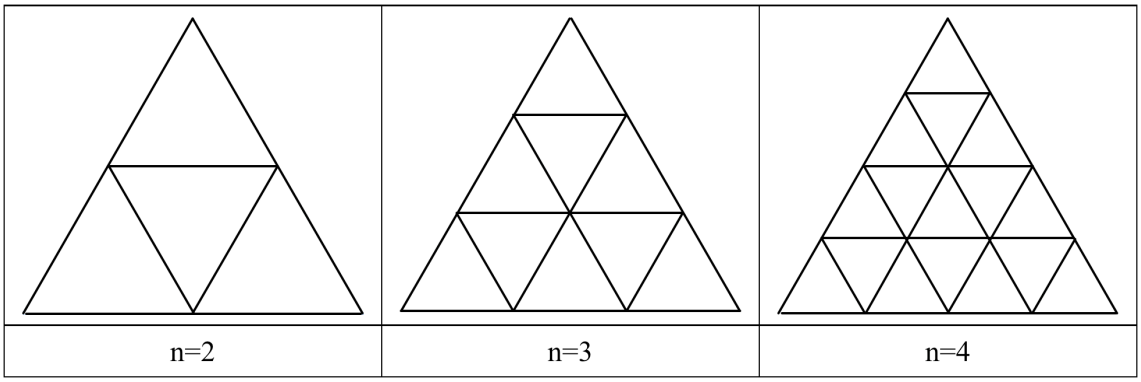

Figura 5.2: Subdivisão de uma face do icosaedro em duas, três e quatro partes.

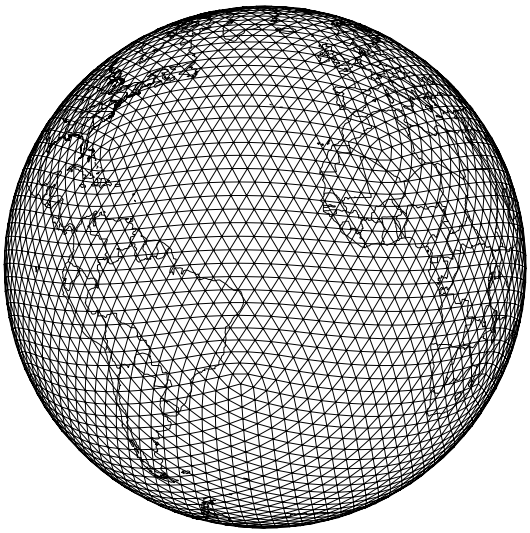

Figura 5.3: Globo terrestre com $NXP = 20$.

Uma face do icosaedro, que é um triângulo equilátero, é subdivida com o parâmetro acima descrito (denominado, no código, por NXP). A figura 5.2 mostra a face do icosaedro, o triângulo, sendo subdivido em duas, três e quatro vezes.

Após a divisão em todas as vinte faces do icosaedro, e aplicação de alguns ajustes, o globo terrestre está coberto por vários triângulos, como colocado na figura 5.3.

O OLAM emprega o conceito de grade dual (*staggered grid*). Neste arranjo as grandezas escalares são calculadas nos pontos centrais dos triângulos, denominados de ponto W. Já as grandezas vetoriais são calculadas nas faces dos triângulos, denominados de ponto U. Como uma extensão a este tipo de grade nos vértices do triângulo o OLAM armazena a topografia, e estes pontos são denominados de M. A tabela 5.1 mostra a quantidade de pontos W, U e M, derivados do parâmetro NXP.

Tipo de elemento	Número de elementos
Vértices M	$10NXP^2 + 2$
Faces U	$30NXP^2$
Triângulos W	$20NXP^2$

Tabela 5.1: Quantidade de pontos resultante de divisões de um lado de uma face do icosaedro.

Ainda sobre a divisão dos grandes triângulos (as faces do icosaedro), no momento da divisão o modelo também indexa a mesma, para assim facilitar a localização de pontos para auxílio na resolução de algoritmos principalmente relacionados a dinâmica. Algumas tabelas de apoio (uma para cada elemento da grade, ou seja, aresta, vértice e triângulo) são criadas para localizar os vizinhos. Nas figuras[1] 5.4 são ilustrados os vizinhos de cada ponto.

Como ilustração a tabela 5.2 descreve cada uma das tabelas de indexação, quantos pontos elas possuem e quais elementos estão armazenados nela (através de um tipo Fortran). É

[1]Figuras gentilmente cedidas pelo autor do modelo.

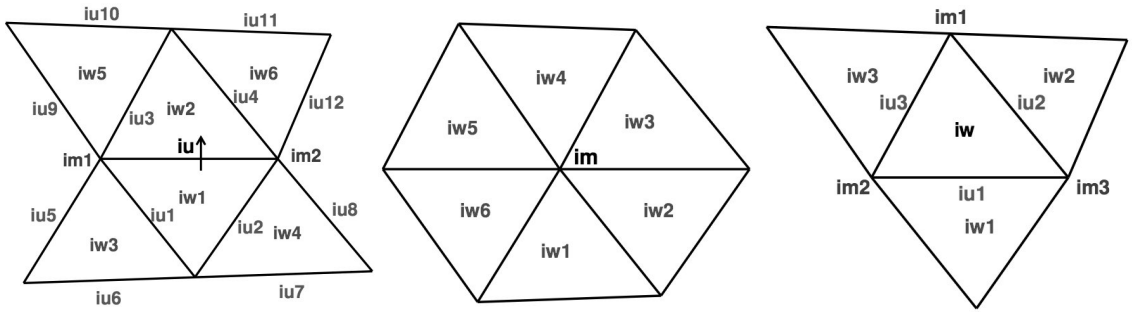

Figura 5.4: Índices dos vizinhos em U, M e W, respectivamente.

importante atentar-se a notação dos vizinhos dos pontos da grade localizados na figura 5.4 e o nome dos apontadores.

Nome	Tipo de elemento	# Elementos	# sub-elementos	Apontadores
$itab\_m$	Vértices M	nma	7	$iw(maxtpn)$, $iu(maxtpn)$
$itab\_u$	Faces U	nua	64	$im1$, $im2$, $iu1$, $iu2$, $iu3$, $iu4$, $iu5$, $iu6$, $iu7$, $iu8$, $iu9$, $iu10$, $iu11$, $iu12$, $iw1$, $iw2$, $iw3$, $iw4$, $iw5$, $iw6$
$itab\_w$	Triângulos W	nwa	55	$im1$, $im2$, $im3$, $iu1$, $iu2$, $iu3$, $iu4$, $iu5$, $iu6$, $iu7$, $iu8$, $iu9$, $iw1$, $iw2$, $iu3$

Tabela 5.2: Descrição de cada tabela de indexação *itab*, onde o número de elementos é o tamanho da tabela, e sub-elementos são os elementos de cada elemento da tabela (apontadores e demais informações).

A grade, como visto, cobre a esfera terrestre com vários triângulos. Mas, ao contrário de uma grade regular, como a do jogo da vida, onde existe uma organização da matriz em memória, a do OLAM não é estruturada. A figura 5.5 ilustra uma parte da grade e sua indexação.

Isto prejudica nas operações de busca de vizinhos e se faz extremamente necessário as tabelas de indexação. Para prejudicar ainda mais a organização da grade o OLAM permite refinamentos locais, retirando a homogeneidade da grade global deixando-a com regiões com mais triângulos que outras. A figura 5.6 mostra uma grade global com 2 refinamentos "telescópicos" (refinamento inserido dentro de outro refinamento).

Esta irregularidade da grade não permite, por exemplo, o uso de uma operação como a MPI_Type_vector para construir um tipo a ser fornecido a uma operação de envio, como MPI_Isend. É necessário realizar outro tipo de construção de tipo derivado ou "estruturar" a borda a ser enviada a outro processador. A escolha do modelo foi de estruturar a grade, operação esta descrita a seguir.

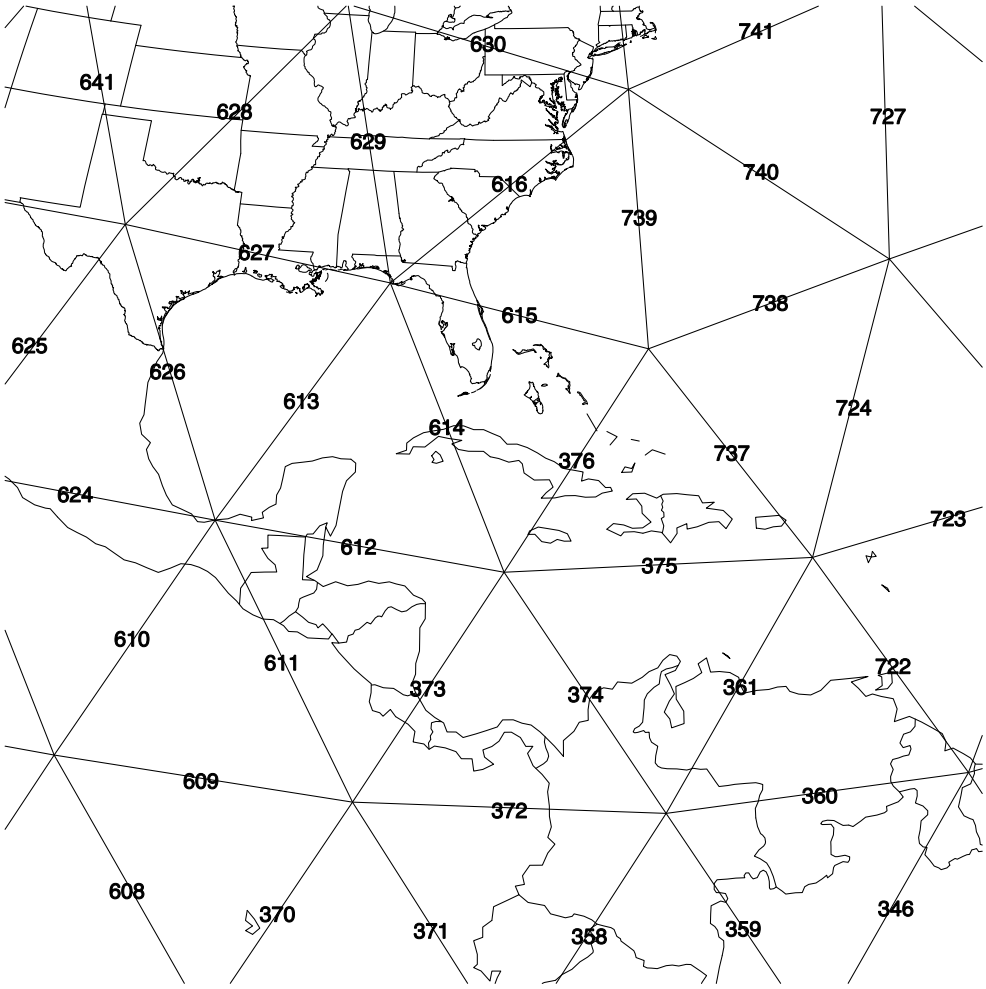

Figura 5.5: Exemplo de indexação da grade. Os valores são os índices das faces dos triângulos (armazenados na $itab\_u$).

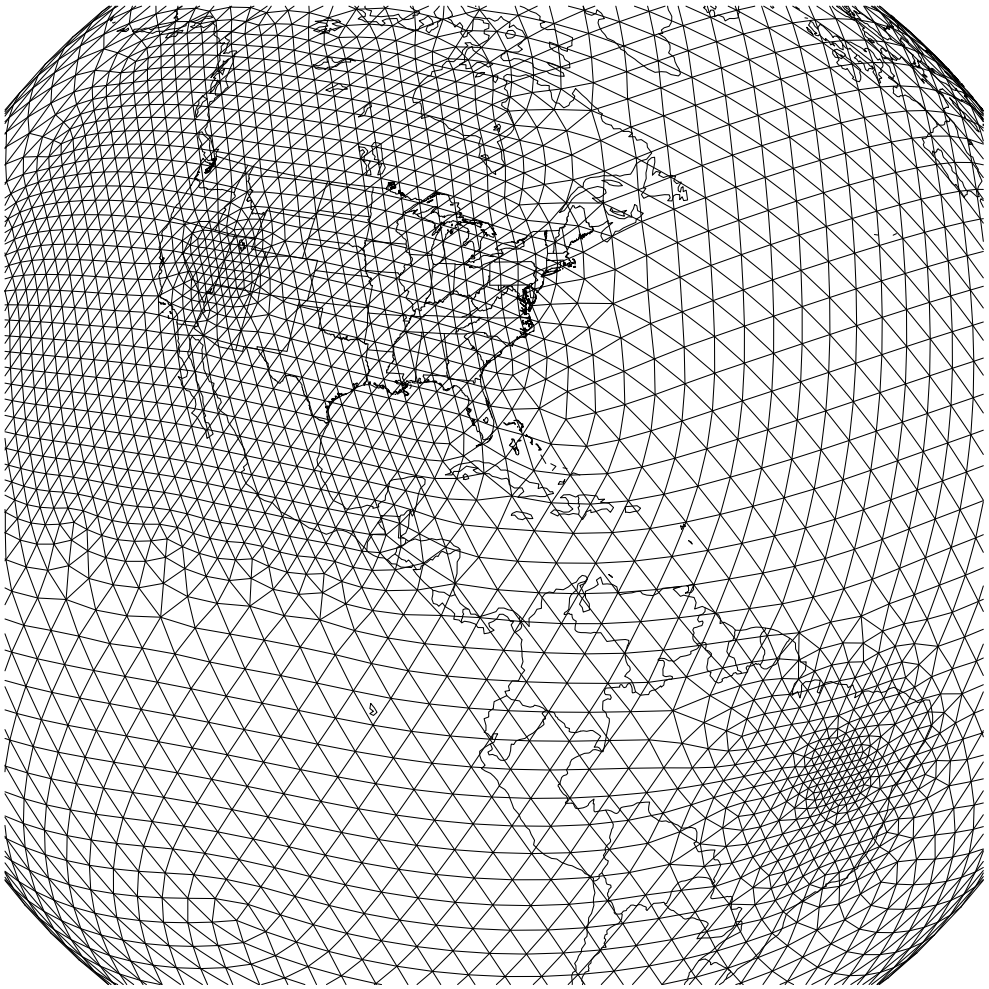

Figura 5.6: Globo terrestre com 2 refinamentos "telescópicos". Um cobrindo parte dos EUA, no lado superior esquerdo. Trata-se de dois refinamentos, o primeiro refinando a grade global (os triângulos maiores) e outro refinando o mesmo refinamento. Outro refinamento situa-se no norte do Brasil, no canto inferior direito, e são 3 refinamentos "telescópicos".

5.1.2 Divisão da grade para o paralelismo

O paralelismo no OLAM foi incluído na versão 3.0, e corrigido na 3.2. Ele possui algumas peculiaridades que o tornam diferente de outros modelos atmosféricos, como o *Fifth-Generation NCAR/Penn State Mesoscale Model* (MM5), descrito por Grell *et al.* [1994], e o *Brazilian Regional Atmospheric Model System* (BRAMS)[2] (até versão 4.2, inclusive). Ele não possui o conceito de mestre/escravo na sua fase de integração, e esta mudança de paradigma do paralelismo abre muitas possibilidades para o modelo tratar alocação de memória, E/S, sincronismo, etc.

Partindo de uma grade global a mesma precisa ser dividida entre diversos processadores. A divisão é geográfica, horizontal e baseada em elementos de área (triângulos da grade), o que é interessante pois se preservam vizinhanças em uma mesma partição (mesma memória), e isso diminui a necessidade de consulta de dados em outra partição (memória). É utilizado um algoritmo de bi-particionamento ponderado, ou seja, a divisão de uma região qualquer gera duas divisões, distintas e únicas, sem pontos duplicados, e a união das duas gera a região original. É ponderada pois a divisão de uma partição com número ímpar de processadores gera duas partições de tamanhos distintos: maior número de pontos para a partição que ficar com número maior de processadores.

A grade original é particionada através de cortes nos eixos Z, X, Y, nesta ordem, sendo a origem no centro da esfera terrestre, Z orientado ao polo norte (90^o de latitude norte), X orientado a 0^{o} de latitude 0^{o} de longitude e Y orientado ao ponto 0^{o} de latitude e 90^{o} de longitude. É tomado o cuidado de gerar novas partições com razão de aspecto pequeno (razão entre o tamanho da grade no eixo das longitudes com o eixo das latitudes).

Estas áreas são novas grades, menores que a original (pois ela foi cortada) e portanto são completas (contem todos os elementos da grade listados na tabela 5.2). O particionamento assegura que um ponto esteja na grade de somente um processador, e desta forma não poderá ocorrer 1) que um ponto não seja integrado ou 2) que um ponto seja integrado mais que uma vez por passo de tempo (*timestep*). O diagrama 5.7 mostra como é o procedimento de partir da grade global para chegar na grade local.

Entretanto, como ocorre de qualquer forma a divisão, haverá regiões onde os vizinhos de alguns pontos estarão em outra partição. Para minimizar a consulta a pontos remotos se escolheu criar regiões ao redor da borda das partições (as *ghoszones*), onde os pontos pertencentes a ela não são calculados pelo processador que a contém, na mesma idéia daquela apresentada no jogo da vida. Os cálculos do modelo após esta divisão são feitos indistintamente por todos os processadores, inclusive o procedimento da divisão.

O processo de E/S é feito globalmente, ou seja, todos os processos fazem E/S. Fora as questões de acesso por vários processadores, "quase ao mesmo tempo" (o modelo possui ordenação entre as instâncias, sem o qual é praticamente impossível resolver as equações

[2]Site: brams.cptec.inpe.br

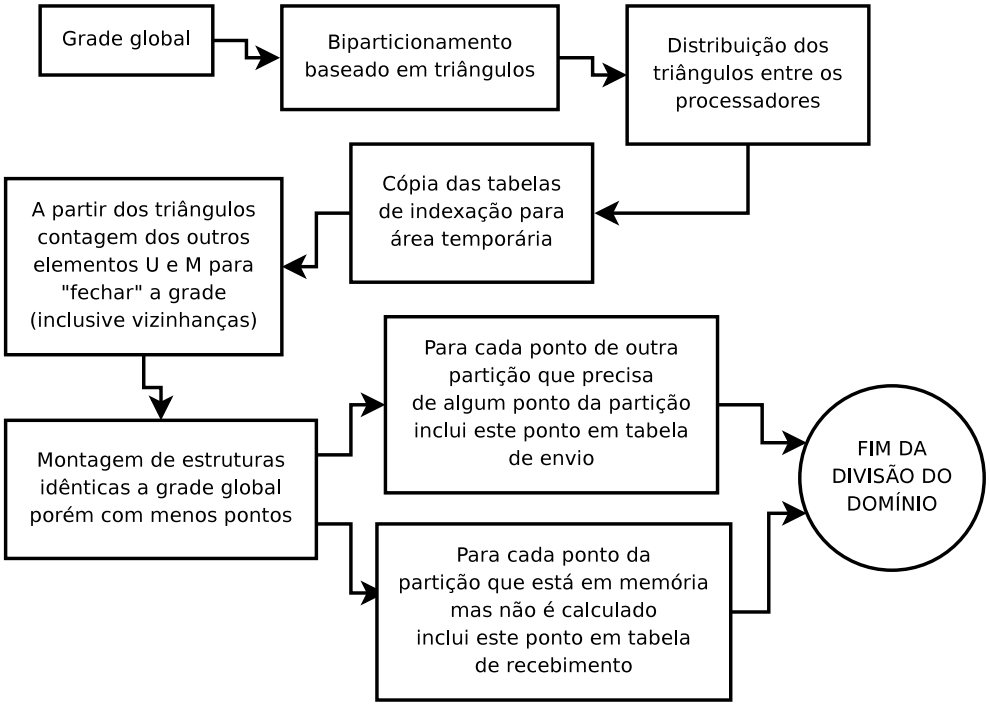

Figura 5.7: Diagrama ilustrativo simplificado das operações da divisão de domínio, a partir da grade global, para a grade local.

da dinâmica, que avançam em passos de tempo ou iterações de forma semelhante ao jogo da vida) de um mesmo arquivo para leitura na inicialização, a escrita é realizada de forma dispersa. De alguma forma no pós-processamento há uma união dos arquivos de saída para gerar um arquivo com a grade global.

A comunicação é realizada com o uso de MPI CBL, com as funções MPI_Isend e MPI_Irecv, ocorrendo ordenação, através das rotinas de *wait*, em alguns pontos do código. Na ocorrência da comunicação o modelo empacota o que for necessário para envio em um *buffer*, devido a borda ser não estruturada em memória.

O empacotamento utiliza tabelas auxiliares preenchidas pelas rotinas de divisão de domínio e escolha dos pontos que compõe as *ghostzones*. Para cada ponto da grade que é parte de uma GZ o mesmo é empacotado, com uso da rotina MPI_Pack, em um vetor de caracteres (tipo *char* do Fortran). Após o empacotamento este vetor é enviado. O processo que recebe este vetor realiza o procedimento inverso de desempacotar os dados e colocá-los na grade do modelo, com a rotina MPI_Unpack, . Este processo – empacotar, enviar, receber e desempacotar – ocorre com auxílio de diversas tabelas, conforme colocado, a fim de assegurar que os dados sejam copiados e gravados na grade em seus lugares corretos. E é justamente aqui que ocorre a troca de memória estruturada, em outras palavras, linearizada.

De forma a ilustrar o recorte da grade e a localização de cada sub-borda, a figura 5.8 foi colocada.

Pode-se notar que a divisão designa para cada sub-borda um valor equivalente para o envio e para recebimento, de acordo com a descrição da figura. Mas o valor do índice da

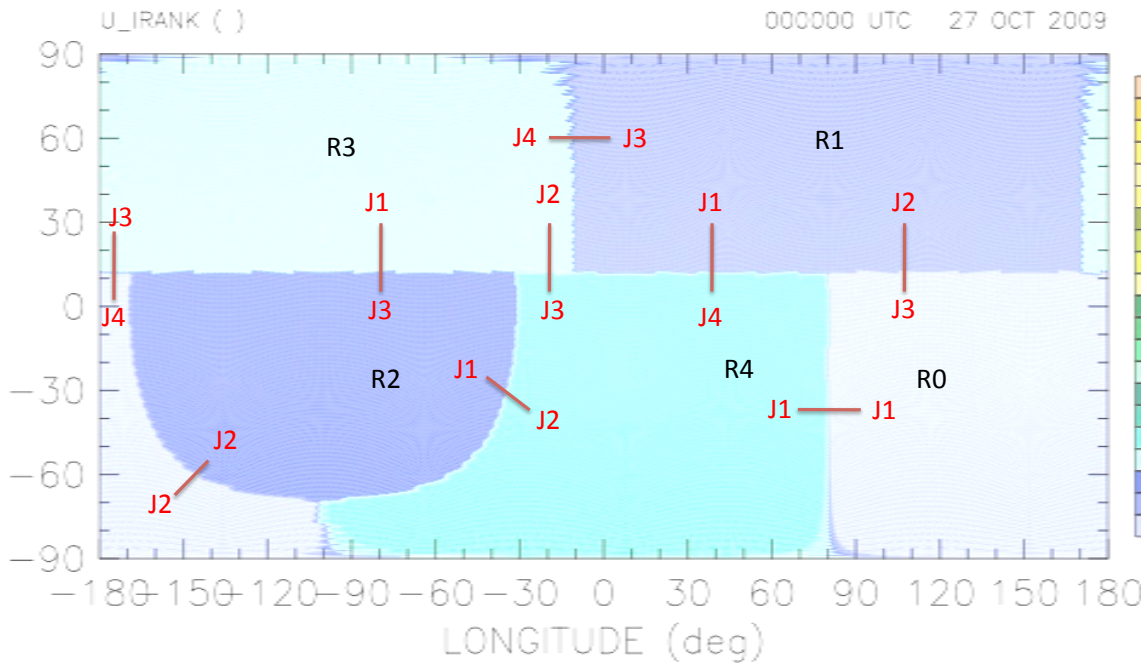

Figura 5.8: Diagrama ilustrativo da grade do OLAM dividida em 5 partições. Os valores em preto são os índices dos *ranks*. Os valores em vermelho são os índices que cada sub-borda possui para cada *rank* em suas estruturas de envio (as de recebimento são iguais). A borda de uma partição é a união de todas as sub-bordas.

sub-borda do *rank* remoto que faz fronteira com outro *rank* pode ser diferente. Por exemplo o $R3$ possui a sua borda e GZ para com $R2$ com índice $J1$, e o $R2$ possui a mesma borda e GZ com índice $J3$. Inclusive um determinado *rank* pode ter um número menor de bordas que outros *ranks*: o $R3$ possui 4 sub-bordas, e o $R2$ possui 3.

5.1.3 Código, fases e execução

O OLAM possui características que podem dar a ele o adjetivo de "código monolítico". Ele possui todas as fases do processamento no código. Seu único executável é responsável por todas as fases do processamento, e seu controle é realizado através de modificações em um arquivo de parâmetros denominado *namelist* (NML). O NML é um arquivo texto com cerca de 200 parâmetros, descritos em Walko [2008], e ele controla todo o funcionamento do modelo.

Ele possui 5 fases de execução, a saber:

1. $MAKESFC$: fase responsável por realizar o pré-processamento de dados de superfície, como topografia, vegetação, uso de solo, etc. São dados estáticos;

2. $MAKEGRID$: responsável por gerar um arquivo com a grade do modelo, com os índices e localização dos triângulos (das três grades);

3. $INITIAL$: a responsável pelo processamento, integrando a atmosfera no tempo. É a que mais demanda tempo de processamento e a única passível de ser executada em paralelo;

4. $PARCOMBINE$: como o E/S do modelo é realizado em paralelo esta fase reúne cada partição (domínio de cada processador) em uma grade global. Basicamente "monta" a grade global, preparando-a para o pós-processamento;

5. $PLOTONLY$: responsável por produzir gráficos com os campos processados pelo modelo. Utiliza, como entrada para os gráficos, os arquivos gerados pela $PARCOMBINE$. A de número 3 também pode gerar figuras com gráficos, porém só da parte da grade que o processador gerencia.

De todas as fases acima descritas somente a de número 3 é paralelizada e portanto utiliza de rotinas de gerenciamento do paralelismo via MPI. Desta forma prosseguiremos com a descrição dessa fase.

O OLAM possui duas grandes partes em sua execução na fase $INITIAL$. A primeira é a parte de inicialização do modelo, preparando os campos para serem integrados. Já a segunda fase é basicamente restrita a sub-rotina *model*, que é responsável por avançar no tempo com os campos. Ambas as fases possuem chamadas a funções de apoio ao paralelismo.

A figura 5.9 é colocada de forma a ilustrar o que ocorre na fase inicial, antes da chamada da rotina *model*.

O modelo começa suas operações inicializando o paralelismo, com contagem de número de processadores, identificação do número do processador dentro do ambiente paralelo, etc. As demais operações são listadas abaixo:

1. Inicialização 1: tarefas como leitura, checagem do NML, inicialização das rotinas da camada de *plots* via pacote *"NCAR Graphics"* (NCARG)[3];

2. Decomposição do domínio entre os processadores, conforme colocado no item 5.1.2. Deste ponto em diante toda operação é realizada na partição, e não na grade global (ela deixa de existir, para o modelo, a partir deste momento);

3. Inicialização 2: inicializa *scheduler* (conjunto de parâmetros que controlam quais laços, trechos, sub-modelos e parametrizações que o OLAM deve executar, para o *namelist* fornecido), aloca grade;

4. Aloca estrutura de troca de bordas, com envio e recebimento de tamanho das estruturas entre os processadores;

[3] *Site: www.ncl.ucar.edu*

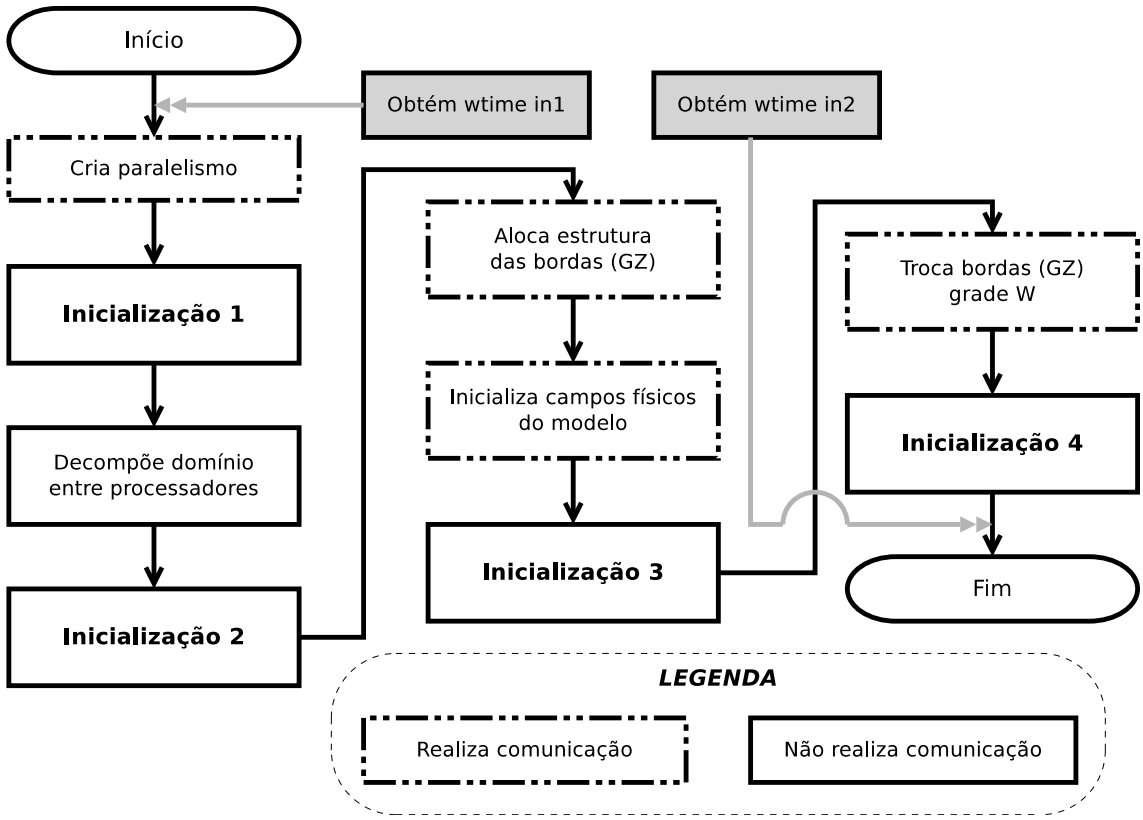

Figura 5.9: Diagrama ilustrativo com as operações realizadas pela fase $INITIAL$ na sua inicialização.

5. Inicializa campos, com leitura de dados de entrada e eventual troca de campos (especialmente as GZ) entre processadores (lembrando que a leitura é paralela, portanto as GZ não são lidas);

6. Inicialização 3: inicializa microfísica;

7. Troca de bordas da grade W (pontos nos centros dos triângulos). Envio e recebimento;

8. Inicialização 4: inicializa radiação, sub-modelos de solo e oceânico, calcula perfis de fricção, plota campos (via NCARG) e grava saída.

Após a inicialização o modelo entra na sua simulação dos campos, avançando no tempo e aplicando toda a física[4] e dinâmica necessária. Esta parte é gerenciada em sua totalidade pela sub-rotina *model*. Esta sub-rotina realiza operações de avançar o estado dos campos do modelo no tempo, executando a sub-rotina *timestep*, e através de um laço avançar o tempo do modelo até este chegar ao tempo final desejado, medindo o tempo de execução de cada iteração desse laço e o tempo total de execução do laço. Este tempo será levado em

[4]Aquilo que se convenciona chamar de "física do modelo" (exemplos são radiação, parametrização de processos de turbulência e microfísica de nuvens) são processos verticais (também denominados processos de coluna), que não precisam de tratamento após a divisão da grade pois a vizinhança necessária para os cálculos estará dentro de uma mesma partição.

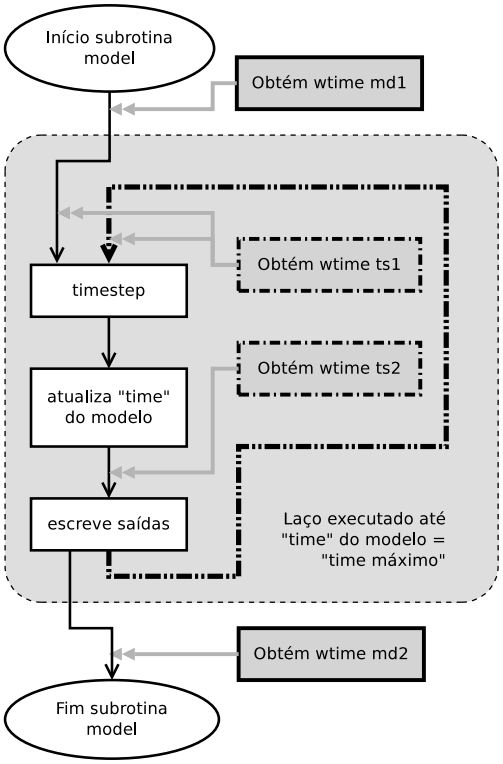

Figura 5.10: Ilustração da função *model*.

consideração como medição do tempo de processamento do modelo. Ocorre também a cada iteração a escrita dos campos do modelo em arquivos de saída, operação governada pelos parâmetros do *namelist*. A figura 5.10 mostra o funcionamento da função *model*.

A medição do tempo, aquela que será levada em consideração nos testes efetuados no item 5.3, é realizada nesta rotina. São dois tempos medidos, abaixo descritos:

- Tempo de execução de cada iteração do laço: o tempo é medido com a subtração dos tempos $ts2$ e $ts1$. Observar que o tempo de escrita das saídas não é medido;

- Tempo de execução de todo o laço: é medido na subtração de $md2$ e $md1$. Todas as iterações do laço e escrita em disco são levadas em consideração. Este tempo é o que vamos utilizar como tempo de processamento do modelo.

Na descrição da primeira parte – a inicialização do modelo – também é medido um tempo, o tempo da inicialização. Ele não é computado na rotina *model*, e não é levado em consideração nos testes de desempenho descritos mais adiante.

A *timestep* é a função que executa cada sub-modelo: transporte, radiação, de solo, oceânico, de tendências, microfísica, etc. A figura 5.11 ilustra as partes principais da função.

As operações, na figura, foram agrupadas. Abaixo temos a descrição sucinta de cada agrupamento:

1. *Timestep* 1: zera tendências;

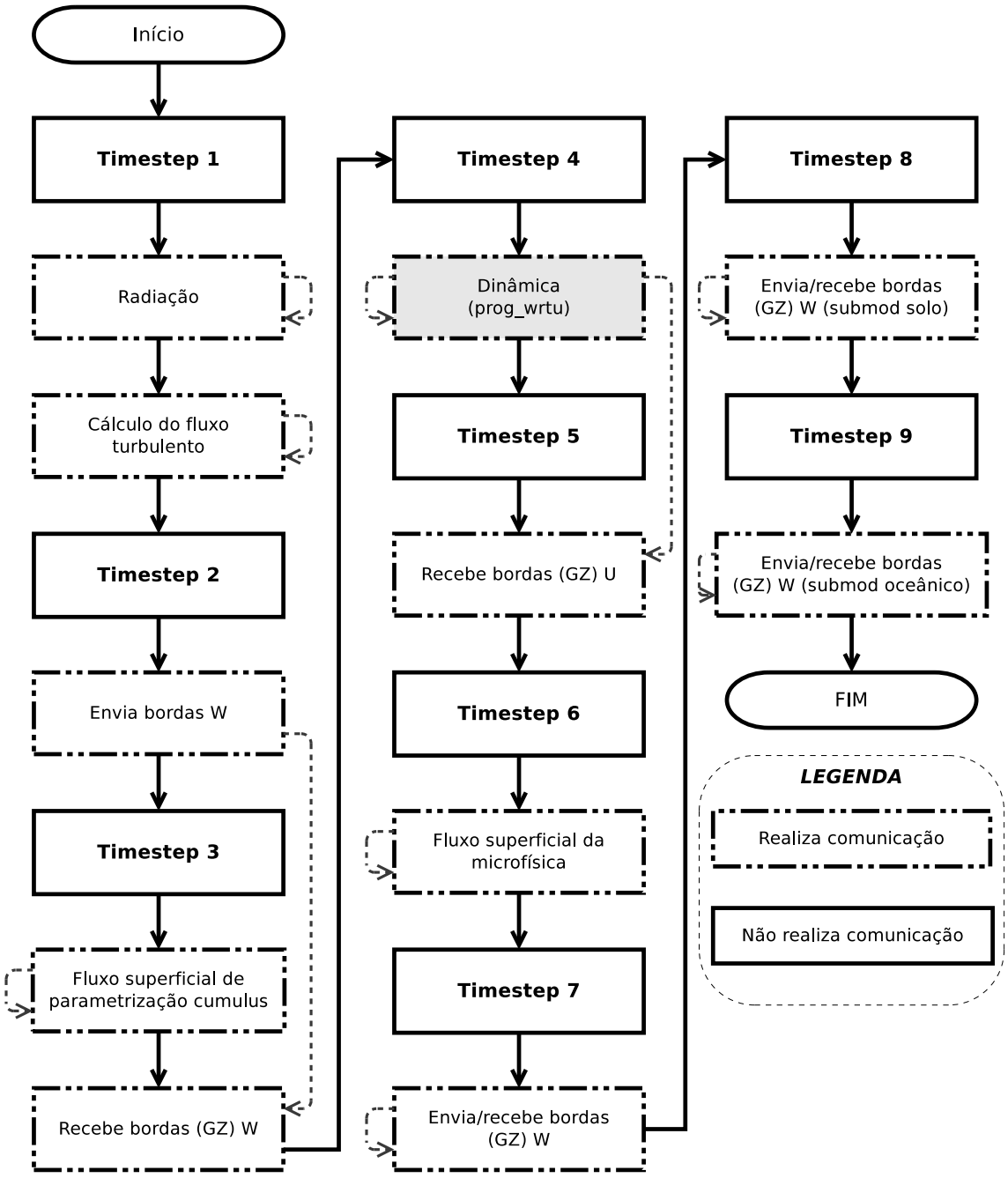

Figura 5.11: Diagrama ilustrativo com as operações realizadas pela fase *INITIAL*, na função *timestep*. As linhas tracejadas, que partem de um retângulo e se dirigem para outro (ou para o mesmo) mostram onde ocorre o envio (origem da linha) e o recebimento (fim da linha, identificado com uma "ponta-de-flecha"). Quando a linha partir e chegar ao mesmo retângulo é porque a operação realiza envio e recebimento, não havendo sobreposição de computação com comunicação.

2. Chamada da radiação: responsável pela radiação do modelo, possui diversas chamadas a envio e recebimento de GZ dos sub-modelos de solo e oceânico;

3. Calcula fluxo turbulento de superfície, com chamadas a envio e recebimento de GZ dos sub-modelos de solo e oceânico;

4. *Timestep* 2: chama a rotina de parametrização da turbulência;

5. Envio de bordas da grade W;

6. *Timestep* 3: parametrização de cumulus;

7. Calcula fluxo entre solo e atmosfera de umidade resultante da parametrização de cumulus, possui chamadas a envio e recebimento de GZ do sub-modelo de solo;

8. Recebimento das bordas (GZ) da grade W, realizada por 5;

9. *Timestep* 4: atualiza tendências, *nudging* da observação e zera componentes do fluxo de massa;

10. Rotina de cálculo da dinâmica. É a mais dependente do paralelismo, devido a grande necessidade de vizinhança horizontal. Efetua envio e recebimento de campos de vento e outras grandezas. Ao final envia borda da grade U com campos de vento;

11. *Timestep* 5: cálculo de fluxo de massa médio e transporte de escalares;

12. Recebimento das bordas (GZ) da grade U, realizada ao final de 10;

13. *Timestep* 6: termodinâmica, microfísica;

14. Fluxo de precipitação para o solo devido a microfísica, possui chamadas a envio e recebimento de GZ do sub-modelo de solo;

15. *Timestep* 7: prognostica escalares;

16. Envio e recebimento de diversos campos da grade W;

17. *Timestep* 8: execução do sub-modelo de solo;

18. Envio e recebimento de bordas relativas ao sub-modelo de solo;

19. *Timestep* 9: execução do sub-modelo oceânico;

20. Envio e recebimento de bordas relativas ao sub-modelo oceânico.

Ao final o modelo libera a memória das grades, finaliza o ambiente paralelo, etc.

5.1.4 Envio de bordas e recebimento de *ghostzones*

Foram identificadas, no código, as funções que tratavam o envio e recebimento dos dados. O código possui um total de 14 funções, sendo sete de envio e sete de recebimento, descritas na tabela 5.3.

Nome(∗)	Envio e recebimento de	Onde é chamado	Variáveis ou grandezas
mpi_???_u	Faces *U*	Dinâmica	vento e momentum
mpi_???_uf	Faces *U*	Dinâmica	número de *courant*
mpi_???_w	Triângulos *W*	Dinâmica e *timestep*	Diversas variáveis com posicionamento na grade *W*
mpi_???_wl	Triângulos *W* (mas pode estar localizado de maneira diferente, pois a grade do solo é diferente da grade atmosférica)	Radiação, fluxo turbulento na superfície, fluxo devido a parametrização de *cumulus* e no final da *timestep*	Temperatura do solo, radiação de onda longa, temperatura do dossel
mpi_???_wlf	Triângulos *W* (mas pode estar localizado de maneira diferente, pois a grade do solo é diferente da grade atmosférica)	Radiação, fluxo turbulento na superfície e fluxo devido a parametrização de *cumulus*	Inicialização, fluxos turbulentos, fluxos de parametrização de cumulus, fluxos de precipitação da microfísica, fluxos radiativos
mpi_???_ws	Triângulos *W*	Radiação e final da *timestep*	Temperatura da superfície do mar, radiação de onda longa
mpi_???_wsf	Triângulos *W*	Radiação	Fluxos turbulentos, fluxos de parametrização de cumulus, fluxos de precipitação da microfísica, fluxos radiativos

Tabela 5.3: Descrição sucinta sobre as funções responsáveis pelo envio ou recebimento de bordas ou *ghostzones* no OLAM (∗: o nome da função possui, entre os "_" e no lugar de "???", a palavra *send* ou *recv*, significando que a função realiza envio ou recebimento, respectivamente).

As funções são similares entre si, com a diferença de tratarem de grades e dados diferentes. As de envio respeitam a ordem de operações conforme ilustrado na figura 5.12.

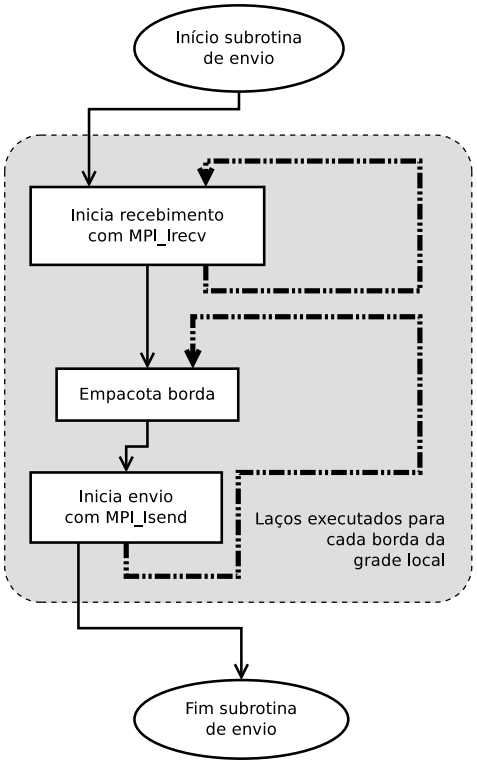

Figura 5.12: Ordem de operações para envio de dados para processador remoto no OLAM.

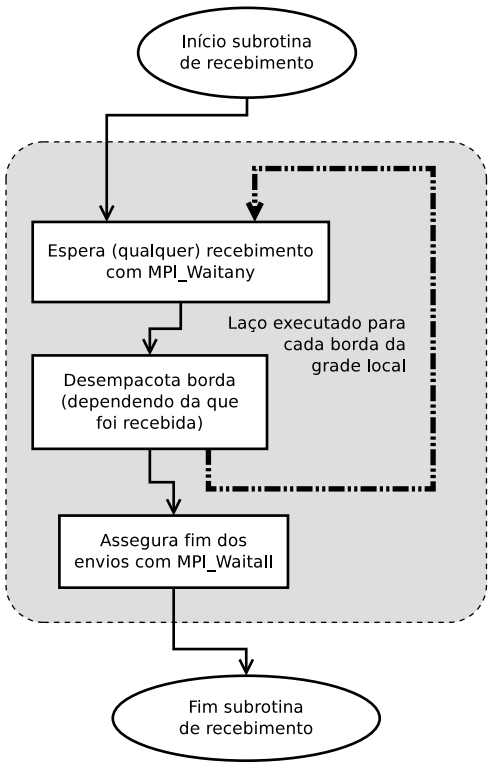

Figura 5.13: Ordem de operações para recebimento de dados para processador remoto no OLAM.

Este esquema é interessante pois ilustra dois pontos: o primeiro é ilustrativo do trata-

mento da grade e suas vizinhanças. A grade local possui diversos vizinhos (outras grades, espalhadas no paralelismo) que são tratadas como sub-bordas (ou uma parte da borda da grade local). Outro ponto é a exploração do retorno imediato das rotinas do MPI CBL, as MPI_Isend e MPI_Irecv. O dado é empacotado, com uso da função MPI_Pack e enviado. Já o recebimento está ilustrado na figura 5.13.

No recebimento é explorada ao máximo a funcionalidade da rotina $MPI\_Waitany$. A cada iteração do laço a mesma recebe o conjunto de recebimentos que existem para a partição, e ela retorna com a transferência que finalizou primeiro. O desempacotamento (via MPI_Unpack) depende da informação do pacote (o primeiro elemento do pacote identifica a qual sub-borda ele pertence). Ao fim é chamada a $MPI\_Waitall$ para assegurar que os envios terminaram.

O esquema de envio e recebimento de sub-bordas utiliza várias práticas para minimizar a latência. Primeiro é sinalizado o recebimento, um para cada sub-borda. Depois é realizado o empacotamento com sinalização do envio do mesmo. Até a chamada de rotina de recebimento (que pode ocorrer após algum processamento que não dependa do que está sendo transferido) o MPI trata de enviar a borda e recebê-la, em segundo plano, geralmente com uso de *buffers* intermediários.

Quando a rotina de recebimento é chamada a tarefa é esperar por algum recebimento que tenha terminado. O desempacotamento é realizado utilizando-se a informação do próprio pacote (o primeiro elemento do pacote descreve onde o mesmo deve ser desempacotado). Este mecanismo é feito para cada borda, mas sem uma ordem específica. Ao final é assegurado que o envio retorne, o que possui retorno quase imediato pois os respectivos recebimentos já terminaram.

5.2 Modificação do código para o MPI CUL

Da mesma forma que o jogo da Vida o código do OLAM foi modificado para possuir comunicação unilateral. Porém esta modificação teve como principal idéia a minimização do impacto no código, ou seja, realizar as modificações que forem necessárias para viabilizar a mudança da forma de envio e recebimento de dados, sem alterar outras estruturas do código.

Conforme listado nos objetivos específicos (item 1.1) o estudo visa analisar o que a comunicação unilateral, seja via MPI CUL ou via *Coarray* Fortran (em uma abstração mais ampla), agrega à programação paralela em grades não estruturadas. Portanto não se pretende modificar a forma que a grade é dividida, quando e como os dados são enviados e a ordem de execução de funções, sub-rotinas, etc. A maneira encontrada foi então de modificar as funções específicas para envio e recebimento de dados e trocar as chamadas da biblioteca MPI CBL para CUL, funções estas listadas na tabela 5.3. Foi possível portanto estudar se é plausível utilizar a comunicação unilateral com a simples troca de funções.

Em resumo o que foi feito foi a troca da rotina MPI_Isend/MPI_Irecv para a

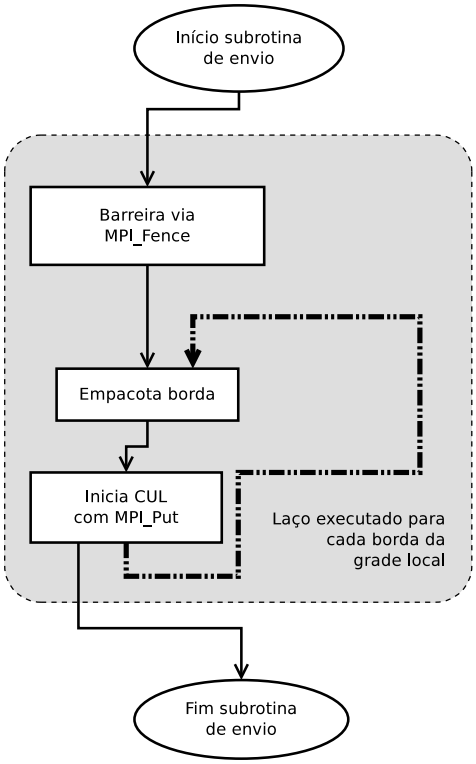

Figura 5.14: Esquema de envio de dados via MPI CUL, com uso da função MPI_Put, no OLAM.

MPI_Put, juntamente com toda a estrutura necessária para janelas, grupos de processos e sincronismos. Apesar de discreta, a mudança significou replanejar toda a sequência de operações de envio de uma sub-borda.

É importante notar que o comportamento da transferência como um todo – envio e recebimento – mudou principalmente na parte da execução para assegurar o recebimento (MPI_Wait contra MPI_Fence). Assegurá-los via MPI_Wait não impõe sincronismo entre duas instâncias. Não há conhecimento, no retorno da função, da condição do processo que envia o dado. Já no MPI_Fence ocorre sincronismo e as duas instâncias entram na barreira no mesmo instante temporal.

A modificação do código, para o MPI CUL, seguiu a sequência de operações, para envio, ilustradas na figura 5.14. Já a sequência para recebimento está na figura 5.15.

Buscou-se explorar a funcionalidade dos acessos a regiões de uma única da janela, conforme figura 2.6. Cada vizinho escreve em trecho da janela, sem ocorrência de sobreposição. Portanto foi criada somente uma janela, responsável tanto para sincronismo quanto para escrita remota (não foi utilizada uma janela *dummy*).

Durante o desenvolvimento foi pensado (e tentado) em expor cada sub-borda, identificada pelo índice J na figura 5.8, em janelas distintas. Mas não há equivalência entre processadores, os índices "não batem". Outro inconveniente é que haveria processadores com mais bordas que outros, o que não é possível devido ao padrão MPI CUL não permitir a existência de janelas que não estejam definidas em todos os processadores de um mesmo comunicador.

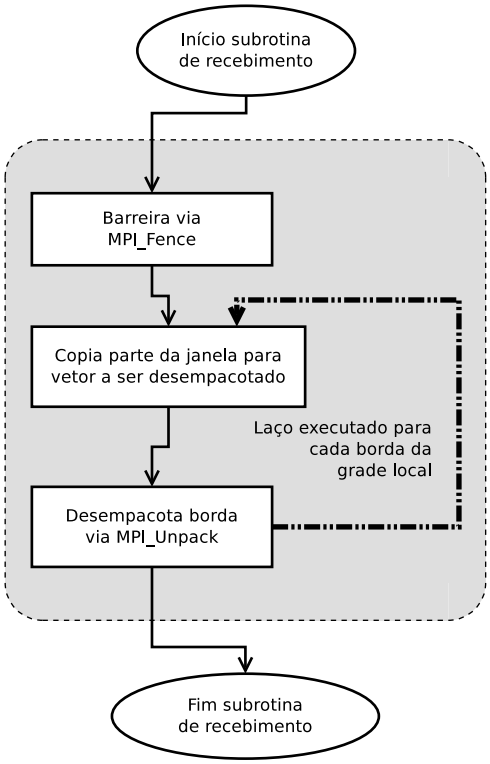

Figura 5.15: Sequência de operações para recebimento de dados via MPI CUL no OLAM.

Houve também o planejamento de uma implementação com diversos comunicadores, um para cada sub-borda. Neste contexto uma janela seria definida para cada sub-borda, e os *ranks* estariam alocados em diversos comunicadores. Ele não seguiu adiante, pois o conceito de comunicação unilateral começa a perder sentido, uma vez que somente dois processadores teriam acesso a uma sub-borda, chegando muito perto do conceito da comunicação bilateral (um envia, outro recebe).

Na implementação de uma grade descrevendo toda a borda ocorreu um inconveniente, que é a cópia de memória para colocação dos dados de cada parte do vetor exposto pela janela para as áreas das sub-bordas. Devido a limitação para modificação do código colocada anteriormente não se modificou as operações de desempacotamento. E também, devido a criação de uma única janela (aquela que expõe a área das bordas) não se criou uma janela para cada sub-borda.

De qualquer forma entende-se que a cópia de memória introduz uma penalidade de tempo linear para cada *timestep*, que depende do tamanho da borda. Quanto maior a borda maior é o tempo gasto pela cópia. Quanto menor, menor é o tempo. A borda de uma sub-grade é geralmente proporcional ao número de elementos que esta contém. Quanto menor a sub-grade menor será a borda.

O número de pontos que uma sub-grade terá é inversamente proporcional ao número de divisões da grade global. Quanto mais dividida for a grade, menor é o número de pontos da borda. As relações da equação 5.1 descrevem o que foi dito sobre a borda.

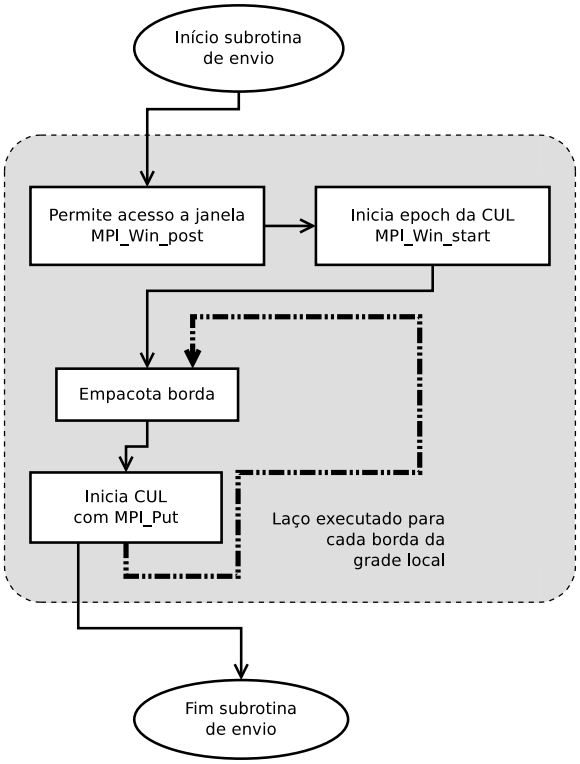

Figura 5.16: Esquema de envio de dados via MPI CUL, com uso da função MPI_Put, no OLAM, com sincronismo *post/start/complete/wait*.

$$tempo \propto borda \propto sub-grade \propto divisões^{-1} \propto instâncias^{-1} \tag{5.1}$$

Desta forma podemos entender que quanto mais instâncias estiverem envolvidas menor é o tempo gasto pela cópia da borda. É de se esperar que com poucos processadores (dois) o tempo gasto será máximo, enquanto que com vários (oito, por exemplo) o tempo será menor. E a figura 5.15 mostra que a operação de cópia da parte da memória exposta pela janela para a área a ser fornecida à operação de desempacotamento ocorre para cada sub-borda, e que a operação de cópia introduz um aumento de tempo linear para cada desempacotamento.

A implementação de CAF no OLAM infelizmente ficou comprometida. A razão é a impossibilidade de executar o código com as premissas colocadas no início deste item. O compilador CAF, o *g95* e seu executor, o *cocon*, impossibilitam o uso concomitante de rotinas do MPI e da notação *coarrays*[5].

De forma similar a sincronização com barreira se utilizou o *post/start/complete/wait* para tentar minimizar a penalidade de tempo que a barreira impõe a heterogeneidade do processamento. As figuras 5.16 e 5.17 ilustram, de forma similar as figuras relativas a barreira, a ordem de chamadas de envio e recebimento, respectivamente.

Para este tipo de sincronismo foi necessária a criação de grupos de processos, o que ocorre

[5]Outro equipamento utilizado, da Cray Inc., possui compilador com suporte a CAF. Mas este foi disponibilizado na finalização deste trabalho, não havendo tempo hábil para a codificação.

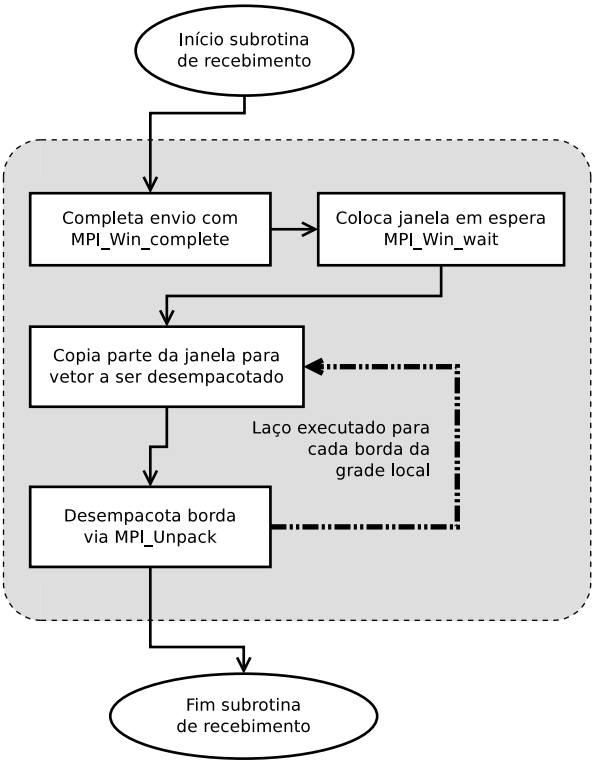

Figura 5.17: Sequência de operações para recebimento de dados via MPI CUL no OLAM, com sincronismo *post/start/complete/wait*.

antes da rotina *model*, na parte de divisão do domínio. Estes grupos são compostos somente pelos vizinhos de um processo, diminuindo então a influência de todos os componentes do paralelismo na questão do tempo entre sincronismos, conforme explicado anteriormente. Mas há dependência pelo "sincronismo", de qualquer forma, ou seja, os processos do grupo só serão liberados para continuar a execução quando todos invocarem a rotina MPI_Win_wait.

Após as implementações foram feitos testes e curvas de *speed-up*, colocadas no item 5.3.

5.3 Testes de desempenho do OLAM

Neste item são descritos os testes realizados com o OLAM tendo como auxílio para a transferência de GZ a comunicação unilateral. De acordo com as premissas que guiaram as implementações, os testes possibilitam analisar se a comunicação unilateral impacta positivamente ou negativamente no desempenho do modelo.

5.3.1 Ambientes e compiladores

Os testes de desempenho realizados com o OLAM ocorreram em três equipamentos: dois *clusters* de PCs e um equipamento da Cray Inc. A descrição destes equipamentos pode ser

encontrado no item 4.3.1, mais especificamente na tabela 4.1. A pilha de *software*[6] de cada equipamento está descrita na tabela 4.2.

Nos *clusters* foi utilizado o compilador da Intel. Já no equipamento da Cray o compilador utilizado foi o da PGI. Esta escolha foi baseada em aspectos de compatibilidade (rotinas do OLAM que não estão diretamente envolvidas com o paralelismo geraram diversos erros de compilação com o compilador CCE) e de desempenho. O item 4.3.2 possui uma descrição mais aprofundada do desempenho do PGI, que em termos de desempenho paralelo é tão eficiente quanto o CCE, o compilador proprietário da Cray.

O modelo foi configurado para ser similar ao modelo global do CPTEC/INPE (Centro de Previsão de Tempo e Estudos Climáticos do Instituto Nacional de Pesquisas Espaciais) com configuração de $100km$ no equador:

- Divisão do lado do triângulo em 50 partes (parâmetro $NXP = 50$), sem refinamento local;
- 28 níveis na vertical;
- passo de tempo foi configurado para 60 segundos;
- inicialização heterogênea (com dados meteorológicos e de umidade do solo);
- escrita em disco desabilitada.

Foram dois experimentos: um com a física do modelo ligada e outra desligada. A física utilizada, quando ligada, foi:

- radiação (tipo Chen), com chamada a cada 3 horas;
- parametrização de Cumulus (Grell), com chamada a cada 1200 segundos;
- difusão (parametrização de Taylor);
- microfísica nível 3
- modelo de solo LEAF3 com 11 níveis de solo e 1 de neve.

Cada experimento foi executado com três tipos de transferência de dados: MPI CBL, MPI CUL com barreiras e MPI CUL com controle refinado. Uma descrição da implementação de cada tipo pode ser encontrado no item 5.2. O compilador utilizado foi o PGI no CROW e o INTEL nos *clusters* de PC's.

A implementação original, com MPI CBL, foi assumida como de controle. Durante a fase de codificação e testes das novas implementações os resultados do processamento (arquivos de

[6]Pilha de *software*, termo do inglês "*software stack*", é um conjunto de programas que trabalham em conjunto para produzir um resultado. O melhor exemplo é um sistema operacional, seus programas, bibliotecas e compiladores.

saída no formato "*Hierarchical Data Format*" – HDF5) eram confrontados[7] com os resultados do controle, de forma a assegurar que a modificação não altera o resultado.

5.3.2 Análise da divisão de domínio do OLAM

O modelo OLAM possui uma grade não estruturada, e sua divisão de domínio é realizada de forma a deixar em cada partição um número aproximadamente constante, entre as instâncias, de pontos. Mas como esta divisão "recorta" a grade em diversas partições este "recorte" pode gerar partições com mais pontos W que outras[8]. As partições que possuírem mais pontos deverão processar mais operações que as que possuem menos pontos, gerando um desbalanceamento no paralelismo.

Para verificar o comportamento da divisão da grade o OLAM (com a configuração descrita no item 5.3.1) foi executado de 1 até 75 processadores, e depois com 96 e 128 processadores. O apêndice A possui uma tabela com diversas características da distribuição dos domínios entre os processadores.

De acordo com a tabela, a média da diferença entre a maior partição e a menor partição é $5,38$ pontos, e a maior diferença proporcional ao número de pontos da menor partição é $1,55\%$, com 128 processadores. De maneira geral a divisão de domínio gera partições com número homogêneo de pontos, colaborando para um *speed-up* mais linear.

5.3.3 Análise do impacto da física no modelo

As execuções do OLAM foram nas mesmas condições do jogo da Vida. Devido a complexidade em medir tempo de processamento que foi enfrentada anteriormente, no caso do OLAM foram feitas cinco execuções para cada número de processadores, para cada comunicação, para cada configuração (sem e com física ligada), e o tempo de processamento é a média destas cinco medidas.

A implementação MPI CBL (de controle) do OLAM foi executada nos três equipamentos, de forma a inferir o peso que a física exerce no *speed-up*. Vale ressaltar que a física é um processo de coluna e que não depende de vizinhanças, introduzindo no processamento uma espécie de "penalidade" no tempo de processamento de todas as instâncias. Esta "penalidade" pode trazer desbalanceamento no paralelismo (instâncias com mais operações que outras), porque a física do modelo pode gerar uma heterogeneidade nos cálculos do domínio.

[7] Este processo, coloquialmente chamado de "conferência de binário", consiste em verificar se existem diferenças entre dois arquivos binários, com algum comando do sistema ($diff$ ou, para o caso do HDF, $h5diff$). Se o comando retornar que não existem diferenças se conclui que a implementação ou modificação do código não altera o resultado. Neste procedimento o conteúdo do arquivo não é importante (no caso, os campos meteorológicos), mas sim se a modificação não altera os resultados.

[8] Escolheu-se analisar os pontos W, os triângulos, porque nesta grade é que são discretizadas quase todas as variáveis do modelo, com exceção das grandezas vetoriais. O número de pontos das outras grades (U e M) é proporcional a W.

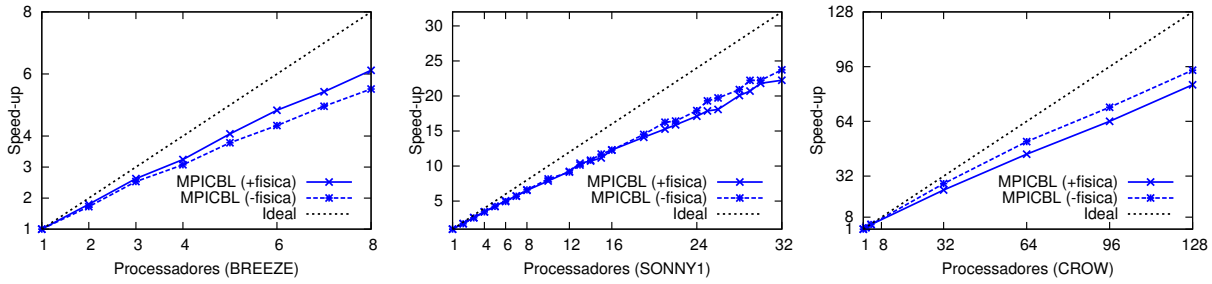

Figura 5.18: Curvas de *speed-up* do OLAM, com física ligada e desligada, nos *clusters* BREEZE, SONNY1 e CROW, com uso de MPI CBL.

Os gráficos da figura 5.18 mostram as duas curvas para cada um dos equipamentos (BREEZE, SONNY1 e CROW), como forma de comparar o desempenho do modelo com física (+fisica) e sem física (-fisica).

No *cluster* BREEZE observa-se que a física desligada possui menor *speed-up* que a ligada. A razão é que o modelo termina seus cálculos seriais mais rápido, devido ao menor número de operações no -fisica (física desligada reduz o número de operações aritméticas para serem executadas), e a comunicação (envio e recebimento das *ghostzones*) acaba tendo um peso maior no tempo de processamento. O "*overhead*" que a comunicação causa no tempo total resulta em uma curva de *speed-up* pior.

Já na SONNY1 o processador é bem mais lento que na BREEZE (a tabela 4.3 mostra o tempo de processamento para o jogo da Vida sequencial, e a SONNY1 é aproximadamente 58% mais lento). Neste caso o "*overhead*" não tem um peso significativo no tempo total, deixando as duas curvas – +fisica e -fisica – quase idênticas. No caso do -fisica, para mais de 16 processadores, este possui melhor *speed-up* que o +fisica, mas ainda com uma diferença pequena.

Na CROW o cenário diverge dos *clusters*: o teste sem física (-fisica) possui melhor curva que o teste com física. Como é uma máquina bem balanceada – rede rápida, acesso a *cache* e memória bem dimensionados – não é esperado "*overhead*" devido a comunicação. O que prejudica o balanceamento do código é a física, neste caso, pois introduz o desbalanceamento.

5.3.4 Análise de desempenho com MPI CUL

Esta análise foi realizada de forma a verificar se a comunicação unilateral (MPI CUL) traz ao tempo de processamento total algum benefício, como o uso do acesso direto a memória remota (RDMA). Mas os resultados mostraram que as funções utilizadas introduzem penalidades de tempo no código, deixando-o mais lento que a implementação de controle (com uso de MPI CBL).

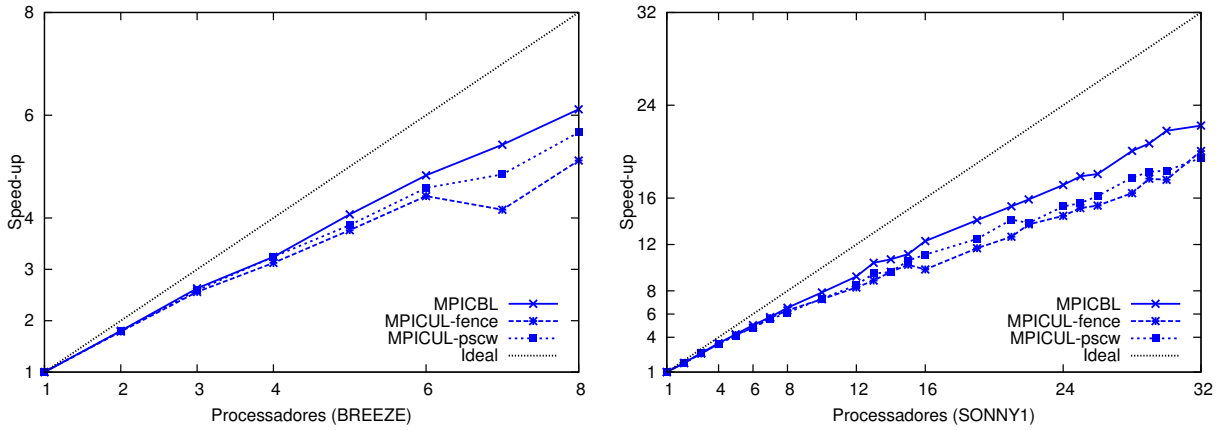

Figura 5.19: Curvas de *speed-up* do OLAM, com física ligada, nos *clusters* BREEZE e SONNY1, com uso de MPI CBL, MPI CUL com barreiras e MPI CUL com controle refinado.

Experimento com física ligada (+física)

Os gráficos da figura 5.19 mostram o comportamento do *speed-up* do OLAM com física ligada nos *clusters* BREEZE e SONNY1, para o experimento com física ligada (+fisica). São apresentadas as três implementações: MPI CBL, MPI CUL com barreiras e MPI CUL com controle refinado do sincronismo. Pode-se observar que a implementação de controle (MPICBL) é quem possui o melhor *speed-up*, seguido do MPI CUL com controle refinado (MPICUL-pscw) e o MPI CUL com barreiras (MPICUL-fence) apresenta o pior desempenho.

A diferença entre MPI CUL com barreiras e controle refinado pode ser explicada pela penalidade que uma barreira gera no tempo de processamento. Uma partição do paralelismo pode já ter terminado suas operações e entrado na barreira, mas estará ociosa esperando que outra partição termine e entre também na barreira, para assim sair e continuar seus cálculos. O controle refinado do sincronismo auxilia neste sentido, o que pode ser observado com a curva MPICUL-pscw com valores maiores que a MPICUL-fence.

A figura 5.20 mostra o comportamento do *speed-up* para o CROW. Observa-se um comportamento similar aos clusters, e a explicação pode ser relacionada com o "*overhead*" gerado pela infraestrutura do MPI CUL (janelas, sincronismos e a cópia explícita das bordas para a um *buffer*, como em um segundo empacotamento). Uma outra explicação pode ser a penalidade que a barreira gera, pois deixa o desbalanceamento entre processadores atrapalhar no tempo de execução. O experimento -fisica, colocado a seguir, mostra que isso não procede.

Em todas as execuções com MPI CUL se observa uma curva com declividade menor que a de controle (MPI CBL), mostrando que o uso de MPI CUL gera um "*overhead*" significativo, suficiente para prejudicar o *speed-up*.

Experimento com física desligada (-fisica)

O OLAM com física desligada foi executado nos três equipamentos. Os gráficos da figura 5.21 mostram o comportamento do *speed-up* para os *clusters* BREEZE e SONNY1. O

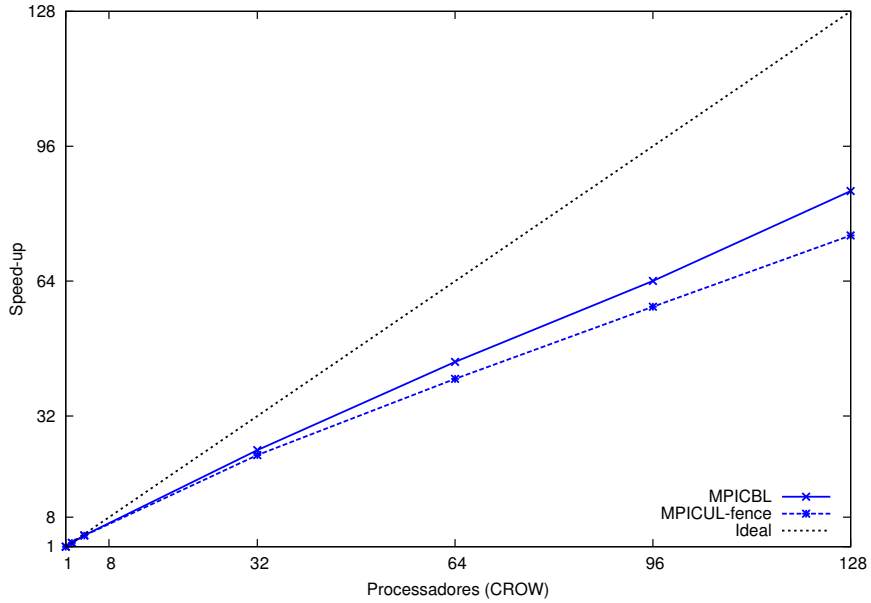

Figura 5.20: Curvas de *speed-up* do OLAM, com física ligada, no CROW, com uso de MPI CBL e MPI CUL com barreiras.

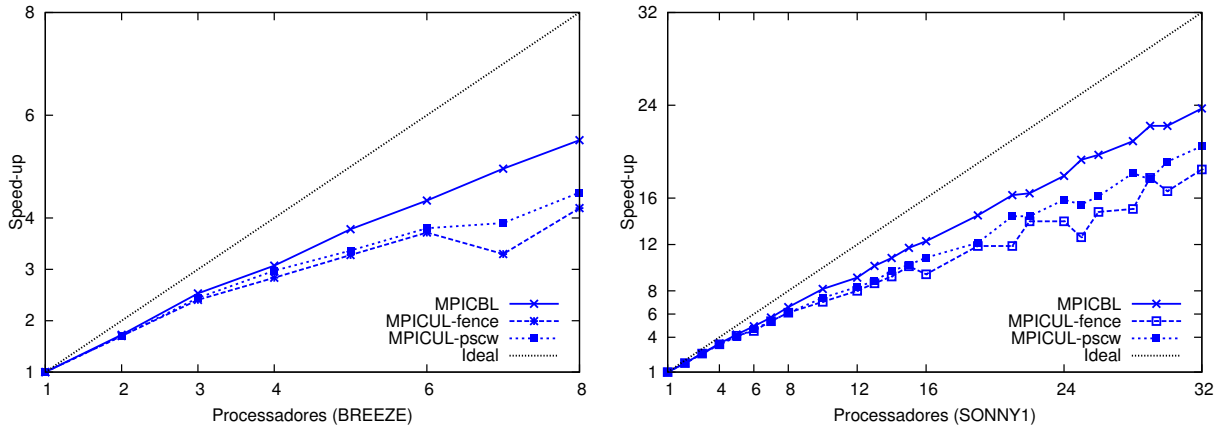

Figura 5.21: Curvas de *speed-up* do OLAM, com física desligada, nos *clusters* BREEZE e SONNY1, com uso de MPI CBL, MPI CUL com barreiras e MPI CUL com controle refinado.

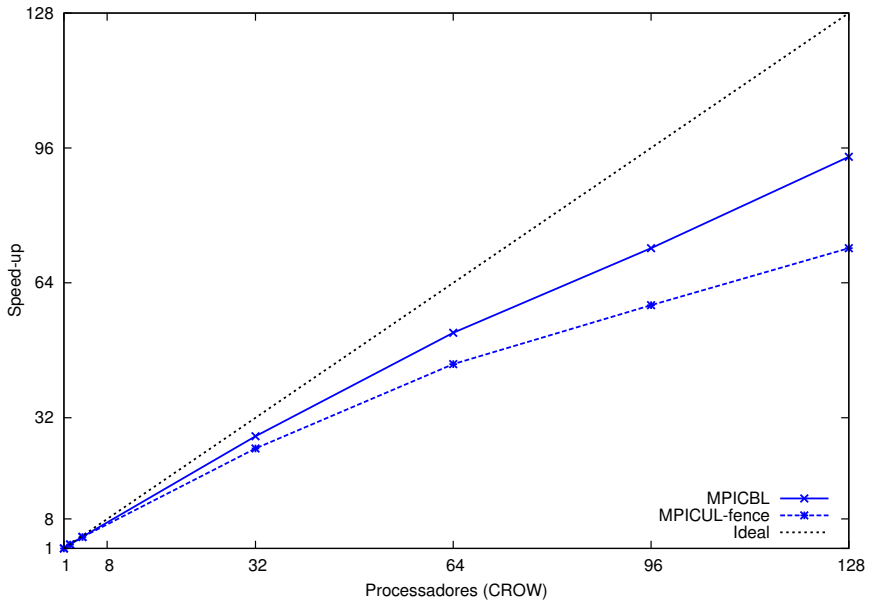

Figura 5.22: Curvas de *speed-up* do OLAM, com física desligada, no CROW, com uso de MPI CBL, MPI CUL com barreiras e MPI CUL com controle refinado.

comportamento encontrado é o mesmo para o experimento +fisica: o controle com a melhor curva, MPI CUL com barreiras a pior curva e controle refinado entre as duas anteriores. E as razões para este comportamento são as mesmas das do experimento +fisica.

No CROW as curvas apresentaram o comportamento conforme ilustrado na figura 5.22. A diferença entre as curvas é maior que no +fisica, e o comportamento é similar ao encontrado nos *clusters*. O MPI CUL com barreiras poderia estar penalizando o *speed-up*, mas duas características não corroboram com esta idéia: a primeira é que a física está desligada, e cada ponto da grade efetua os mesmos cálculos (o globo é tratado como uma "esfera perfeita e lisa"); a segunda é que com 128 processadores o "recorte" da grade distribui pontos de maneira uniforme, ou seja, a diferença entre o número de pontos da partição com a maior grade e a partição com a menor grade é muito pequena $(1,55\%)$. Não havendo agentes relacionados ao código modelo que "penalizem" o tempo de processamento total conclui-se que o MPI CUL introduz "*overhead*" no processamento de forma a prejudicar o *speed-up*.

5.4 Observações

O experimento de trocar as operações de envio e recebimento que o modelo OLAM utiliza para o paralelismo serviu para analisar alguns aspectos dos benefícios que a comunicação unilateral pode ou não trazer ao código. Ele foi limitado à mínima modificação do código, ocorrendo basicamente a troca das funções do MPI CBL para o MPI CUL.

O código do OLAM possui características que o torna interessante: grade não-estruturada; divisão da grade para o paralelismo totalmente distribuída; sobreposição de bordas com uso

de *ghostzones*; centralização da tarefa de envio e recebimento de bordas em poucas funções.

A primeira característica, a grade não-estruturada, acaba por não fazer parte do paralelismo, isso porque durante o envio e recebimento das bordas e GZ há o processo de empacotamento em vetores de caracteres. O "empacotamento dos dados" retira a desestrutura em memória da grade. O paralelismo, nesta abordagem, troca dados "estruturados" (vetores de caracteres lineares). A tarefa de "linearizar" e "deslinearizar" a GZ fica a cargo de rotinas internas do OLAM.

Pode-se dizer que neste caso o problema da não-estrutura da grade fique comprometido, eliminando qualquer abordagem para a segunda característica listada. O paralelismo, independente da sua forma (MPI CBL, CUL ou CAF, ou qualquer outro) não trabalha com a grade não-estruturada propriamente dita. Mas a divisão da mesma continua não-estruturada, pois a divisão foi realizada de forma a balancear entre os processadores algo não-estruturado. Este balanceamento é não-estruturado também. As partições possuem números similares de pontos. O padrão ou desenho das divisões não é estruturado. A física, quando utilizada, gera um desbalanceamento natural no número de operações a serem resolvidas para cada ponto da grade. E a carga de processamento também não é determinado *a priori*. Isso permite que a desestruturação não se perca, e o estudo continue tratando de algo, como dito, não-estruturado.

Em se tratando de envio e recebimento de bordas e GZ este é realizado com a boa exploração das funcionalidades das rotinas do MPI CBL. As sub-bordas – uma para cada vizinho da partição – são empacotadas e logo enviadas, em segundo plano. Assim que for oportuno o modelo consulta o controle do MPI para saber qual borda já chegou, e desempacota dependendo desta resposta. Ele "serializa" o envio, e recebe fora de ordem. Uma implementação "esperta" do MPI CBL pode utilizar memórias intermediárias para o envio e recebimento das sub-bordas. Estas memórias podem ser atualizadas em segundo plano, deixando o modelo trabalhar com suas operações, que não precisam da transferência de sub-bordas finalizada, permitindo uma boa sobreposição de comunicação e computação.

Estes aspectos levam ao entendimento que o OLAM pode usufruir da comunicação unilateral. Ela possui o envio em segundo plano, pois é assíncrona por desenho. Mas o fato de o modelo ter boa independência do envio e recebimento (estas operações ocorrem em momentos distintos, e não uma seguida da outra) pode significar que o sincronismo penalize o desempenho.

Em termos de semântica a troca de MPI CBL para MPI CUL significou realizar sincronismos antes das operações de comunicação do MPI CBL (envio e recebimento). Houve um esforço de programação maior para localização dos dados remotos, já que o OLAM deixa cada partição independente para localizar as sub-bordas, com a inclusão de um vetor a mais, que é exposto por uma única janela.

O entendimento de uma janela única para toda a borda, que é acessada de forma independente por cada vizinho, escrevendo na mesma em uma região específica, significou o

tratamento único, em termos de sincronismo, para todos eles. Ou seja, os vizinhos são livres para escrever na janela no momento que melhor lhe convier, mas o sincronismo é global (caso da barreira) ou local englobando todos eles (caso do controle refinado do sincronismo). No nosso entendimento esta abordagem de janela está no caminho contrário da independência de cada partição da grade.

Um interessante trabalho a ser desenvolvido é portar a notação CAF para o OLAM. Ela ficou comprometida devido ao fato de ser impossível executar o código com um misto de funções do MPI com uso da notação CAF[9]. Mas também para permitir explorar o poder do CAF as premissas de modificação do código precisam ser "relaxadas", para assim permitir acesso direto à sub-grade e seus índices.

Nos experimentos o OLAM mostrou-se um modelo com bom desempenho paralelo. A implementação com comunicação através de MPI CBL, denominada de "controle" possui *speed-up* considerado bom para um modelo tão complexo. Isso se dá devido a sua divisão de grade (gera partições aproximadamente homogêneas) e a inexistência de um "controlador" (como no paradigma mestre-escravo). Este bom desempenho impõe um desafio para outros tipos de comunicação. Mas quando codificado para utilizar o MPI CUL, independente do tipo de controle de sincronismo, o OLAM não apresentou um *speed-up* tão bom como o da implementação de controle. Isso mostra que a comunicação introduziu uma espécie de "penalidade" no tempo de processamento.

Nos *clusters* de PC's (BREEZE, com processador e memória mais rápidos, e SONNY1, com processador e memória mais lentos, mas com a mesma conectividade) ocorre muita flutuação no *speed-up*, o que mostra que o equipamento está interferindo no desempenho do modelo. Já no equipamento CROW (da Cray Inc) esta flutuação não é observada, lembrando que o CROW é bem dimensionado, com rede (cerca de 40 vezes mais rápida que nos PC's), acesso a memória e *cache* bem planejados.

Como foram feitas duas configurações do modelo – uma somente com seu núcleo dinâmico, e outra com física sendo utilizada – foi realizada uma comparação do desempenho do controle para cada equipamento. Nesta comparação a física pode prejudicar o *speed-up* quando esta for resolvida muito rapidamente, como é o caso da BREEZE. Mas, no CROW, o experimento sem física escala melhor que com física ligada, prejudicando o paralelismo.

Nas comparações com a implementação da comunicação com uso de funções do MPI CUL se observa que o desempenho é pior comparado a implementação com MPI CBL, mostrando a existência de uma penalidade no tempo de processamento. Esta penalidade é fruto do conjunto MPI CUL (função MPI_Put e sincronismo) e empacotamento das bordas (já previamente empacotadas) na área de memória exposta. A troca do tipo de sincronismo (barreiras para controle refinado) melhora o desempenho, mas não chega a ultrapassar o desempenho da implementação de controle.

[9]Foi-nos concedido acesso a um equipamento Cray com compilador com suporte a CAF, mas já nos momentos finais deste trabalho, não permitindo tempo hábil para portar o código do modelo.

Capítulo 6

Conclusões gerais e trabalhos futuros

O presente trabalho buscou analisar os benefícios que a comunicação unilateral (CUL), como apoio ao processamento paralelo, trouxe a programas que envolvem grades de pontos, especialmente aquelas não-estruturadas em memória. Estes benefícios podem ser no campo da semântica e no campo do desempenho. Nos programas utilizados não houve melhora no desempenho, e no campo da semântica a comunicação unilateral não deixou a codificação mais simples ou robusta, com exceção do *Coarray* Fortran.

A CUL tem como premissa que somente um dos envolvidos na comunicação deve conhecer a origem de uma informação, seu formato, o destino, seu formato no destino e o instante que a comunicação deve ocorrer. Como consequência direta é necessário saber mais informações, em comparação a forma mais clássica de comunicação, a baseada na bilateralidade (quem envia deve ter conhecimento da origem do dado e seu formato; quem recebe deve saber o destino e o seu formato). Ocorre portanto um aumento da complexidade da programação, pois quem envia (ou recebe) dados deve conhecer a outra parte. Mas este aumento proporciona maior flexibilidade, pois permite independência entre as instâncias em um grau maior que a comunicação bilateral.

A principal biblioteca de apoio ao paralelismo e que implementa a CUL (e a bilateral, a CBL) é a "*Message Passing Interface*" (MPI), e esta possui uma semântica para a CUL que a torna mais complicada que a comunicação bilateral. São necessárias janelas (onde variáveis são expostas) e o padrão dispõe de três tipos de sincronismo, o que é necessário por desenho pois as funções de comunicação não possuem sincronismo (são de retorno imediato). Estas funções, por sua vez, precisam de todos os aspectos da comunicação, conforme descrito no parágrafo anterior.

O MPI CUL foi confrontado com a extensão à linguagem Fortran, denominada *Coarray* Fortran (CAF), que incorpora a linguagem paradigmas de programação paralela, e possui unilateralismo nas operações de comunicação. O CAF reduz o esforço de programação (alguns autores encontraram redução de 40% no número de linhas de código), o que foi observado no jogo da Vida (um programa de grade estruturada em memória, utilizado como campo de prova neste trabalho).

Além de reduzir o esforço o CAF permite otimização, pois está incorporado à linguagem e não é uma biblioteca externa: o compilador desta forma tem subsídios para otimizar, a comunicação está explícita no código Fortran. Os resultados de desempenho do jogo da Vida, cristalizados nas curvas de *speed-up*, mostraram que o CAF é mais rápido que o MPI CBL e CUL, enquanto o número de instâncias for menor ou igual ao número de núcleos que um computador possui (no caso o CROW, equipamento da Cray Inc, cada nó deste cluster possui oito núcleos). Já a implementação com uso de MPI CBL é a que melhor curva de *speed-up* apresenta, mostrando que o MPI CUL penaliza o tempo de execução, mesmo utilizando-se de tipos derivados e possibilitando assim uso de tecnologia RDMA ("*Remote Direct Memory Access*").

Com o objetivo de trabalhar com grades não-estruturadas o modelo "Ocean Land Atmospheric Model" (OLAM) foi utilizado. Ele já é paralelizado, com uso do MPI CBL, e com algumas restrições de modificação do código trocou-se as rotinas de comunicação para utilizar o MPI CUL. Foi necessário criar uma estrutura para a comunicação unilateral (para satisfazer premissas do uso da CUL, como exposição de somente uma janela por instância paralela do modelo), que de certa forma estruturou a grade. Com estas restrições o desempenho com uso de MPI CUL, comparado com a implementação original em MPI CBL, foi pior, novamente mostrando que a CUL penaliza o tempo de execução.

Com as análises realizadas se conclui que a comunicação unilateral não traz benefícios para os programas testados. O CAF se mostrou rápido e simples mas não apresenta *speed-up* sustentável com mais instâncias que o número de núcleos que compartilham memória. A implementação da comunicação do OLAM via MPI CUL não deixou transparecer benefícios de desempenho, mas esta melhoria não foi observada no jogo da Vida, com paralelismo simples, sem uso de empacotamentos e utilizando-se de tipos derivados.

Para trabalhos futuros sugere-se investigar se o CAF no OLAM também apresenta bons resultados, principalmente quando a for utilizado otimização máxima. Outro ponto importante é relaxar as restrições de modificação do OLAM, ou premissas para o uso da CUL: isso permitirá verificar se a implementação deste trabalho é quem penaliza o tempo de execução, causando o resultado observado. Com as restrições relaxadas é possível explorar outros tipos de sincronismo que o MPI CUL fornece, como travamento de memória.

Referências Bibliográficas

Ashby e Reid(2008) J.V. Ashby e J.K. Reid. Migrating a scientific application from MPI to coarrays. *Proceedings 50th Cray User Group meeting*. Citado na pág. 8

Barrett(2006) R. Barrett. Co-array Fortran experiences with finite differencing methods. *Proceedings 48th Cray User Group meeting*. Citado na pág. 8

Bonachea(2002) D. Bonachea. GASNet Specification, v1. *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*. Citado na pág. 35

Butler e Lusk(1994) Ralph M. Butler e Ewing L. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Parallel Computing*, 20(4):547–564. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/0167-8191(94)90028-0. Citado na pág. 6

Coarfa *et al.***(2005)** C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, e D. Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. Em *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, páginas 36–47. ACM. Citado na pág. 35

Darema(2001) F. Darema. The spmd model: Past, present and future. *Lecture Notes In Computer Science*, páginas 1–1. Citado na pág. 5

Dawson(2004) Jef Dawson. Co-array Fortran for productivity and performance. *AHPCRC Bulletin*, 14(4):4–12. Citado na pág. 8

Dobbelaere e Chrisochoides(2007) Jeffrey Dobbelaere e N. Chrisochoides. One-sided communication over mpi-1. Citado na pág. 23

Dotsenko *et al.***(2004)** Yuri Dotsenko, Cristian Coarfa, e John Mellor-Crummey. A multi-platform co-array Fortran compiler. *The Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*. URL `http://www.hipersoft.rice.edu/caf/publications/caf-pact04.pdf`. Citado na pág. 35

Flynn(1972) M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960. Citado na pág. 5

Gaglianello *et al.***(1989)** RD Gaglianello, BS Robinson, TL Lindstrom, e EE Sampieri. Hpc/vorx: A local area multicomputer system. Em *Distributed Computing Systems, 1989., 9th International Conference on*, páginas 542–549. Citado na pág. 6

Gardner(1970) M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123. Citado na pág. 39

Geist *et al.***(1990)** G.A. Geist, M.T. Heath, B.W. Peyton, e P.H. Worley. A user's guide to picl: A portable instrumented communication library, 1990. Citado na pág. 6

Grell *et al.***(1994)** G.A. Grell, J. Dudhia, e D.R. Stauffer. A description of the fifth-generation penn state/ncar mesoscale model (mm5). ncar tech. *Note TN-398+ STR*. Citado na pág. 70

Gropp *et al.***(1996)** W. Gropp, E. Lusk, N. Doss, e A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828. Citado na pág. 10

Gropp *et al.***(1999a)** W. Gropp, E. Lusk, e A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. the MIT Press, segunda edição. Citado na pág. 7

Gropp(2001) W.D. Gropp. Learning from the success of MPI. *Lecture Notes in Computer Science*, páginas 79–80. Citado na pág. 6

Gropp *et al.***(1999b)** William Gropp, Weing Lusk, e Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MPI Press. Citado na pág. 7

Hempel(1991) R. Hempel. The anl/gmd macros (parmacs) in fortran for portable parallel programming using the message passing programming model–users' guide and reference manual. Citado na pág. 6

Kanellos(2005) M. Kanellos. New life for Moore's law, 2005. URL http://news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html. Citado na pág. 1

Meglicki(2004) Zdzislaw Meglicki. The history of MPI, 04 2004. URL http://beige.ucs.indiana.edu/I590/node54.html. Citado na pág. 6

Moore(1965) G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38:114–117. Citado na pág. 1

MPI-Forum(1994) Message Passing Interface MPI-Forum. MPI: A message-passing interface standard, 1994. URL http://www.mpi-forum.org/docs/mpi1-report.pdf. Citado na pág. 6

MPI-Forum(1998) Message Passing Interface MPI-Forum. MPI-2: Extensions to the message-passing interface, 1998. URL http://www.mpi-forum.org/docs/mpi2-report.pdf. Citado na pág. xiii, 6, 17

Nieplocha e Ju(1999) J. Nieplocha e J. Ju. Armci: A portable aggregate remote memory copy interface. Em *3 rd Workshop on Run-Time Systems for Parallel Programming (RTSPP'99)*, páginas 533–546. Citeseer. Citado na pág. 35

Numrich e Reid(1998) R.W. Numrich e J. Reid. Co-array Fortran for parallel programming. Em *ACM Sigplan Fortran Forum*, volume 17, páginas 1–31. ACM New York, NY, USA. Citado na pág. 7

Numrich *et al.***(1998)** R.W. Numrich, J.L. Steidel, B.H. Johnson, B.D. Dinechin, G. Elsesser, G. Fischer, e T. MacDonald. Definition of the f–extension to Fortran 90. *Lecture notes in computer science*, páginas 292–306. Citado na pág. 7

Reid(2009) John Reid. Coarrays in the next Fortran standard. Relatório técnico, ISO/IEC JTC1/SC22/WG5 N1787. Citado na pág. 7, 25, 29, 34

Silva *et al.***(2009)** R.R. Silva, P.L.S. Dias, D.S. Moreira, e E.B. Souza. Ocean-land-atmosphere model (olam): description, applications, and perspectives. *Revista Brasileira de Meteorologia*, 24:144–157. Citado na pág. 64

Sunderam *et al.***(1994)** V.S. Sunderam, GA Geist, J. Dongarra, e R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel computing*, 20 (4):531–545. Citado na pág. 6

Sutter(2005) H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210. Citado na pág. 1

Vaught(2008) Andrew Vaught. *The Complete Compendium on Cooperative Computing using Coarrays*. G95 Compiler, 10 2008. Citado na pág. 34, 36

Walko e Avissar(2008a) R.L. Walko e R. Avissar. The ocean–land–atmosphere model (OLAM). part i: Shallow-water tests. *Monthly Weather Review*, 136(11):4033–4044. Citado na pág. 63, 64

Walko e Avissar(2008b) R.L. Walko e R. Avissar. The ocean–land–atmosphere model (OLAM). part ii: Formulation and tests of the nonhydrostatic dynamic core. *Monthly Weather Review*, 136(11):4045–4062. Citado na pág. 63, 64

Walko(2008) Robert Walko. *OLAM Ocean-Land-Atmospheric Model Version 3.0: Model Input Parameters*, 11 2008. Citado na pág. 72

Apêndice A

Análise da distribuição de domínios no paralelismo do OLAM

O modelo "Ocean Land Atmospheric Model" (OLAM) é um modelo de simulação terrestre, de escala global, que discretiza o globo através de uma grade não estruturada em memória. O elemento geométrico no qual a grade é baseada é o triângulo. Portanto a grade cobre a Terra através de diversos triângulos, como colocado na figura A.1.

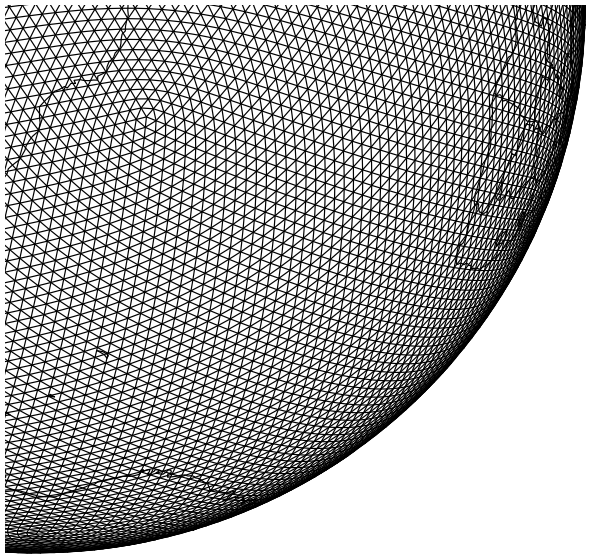

Figura A.1: Parte da grade do OLAM, formada por triângulos, cobrindo o oceano Atlântico Sul.

O OLAM é um modelo paralelizado, e sua distribuição do processamento é realizada através do recorte da grade global em diversas partições, uma para cada processador ou instância do programa. A sequência apresentada na figura A.2 mostra as partições que cada instância do programa deve processar.

Visualmente é possível observar que os domínios não se sobrepõem, mas eles possuem uma pequena sobreposição, devido ao paralelismo (pontos auxiliares para reduzir o número

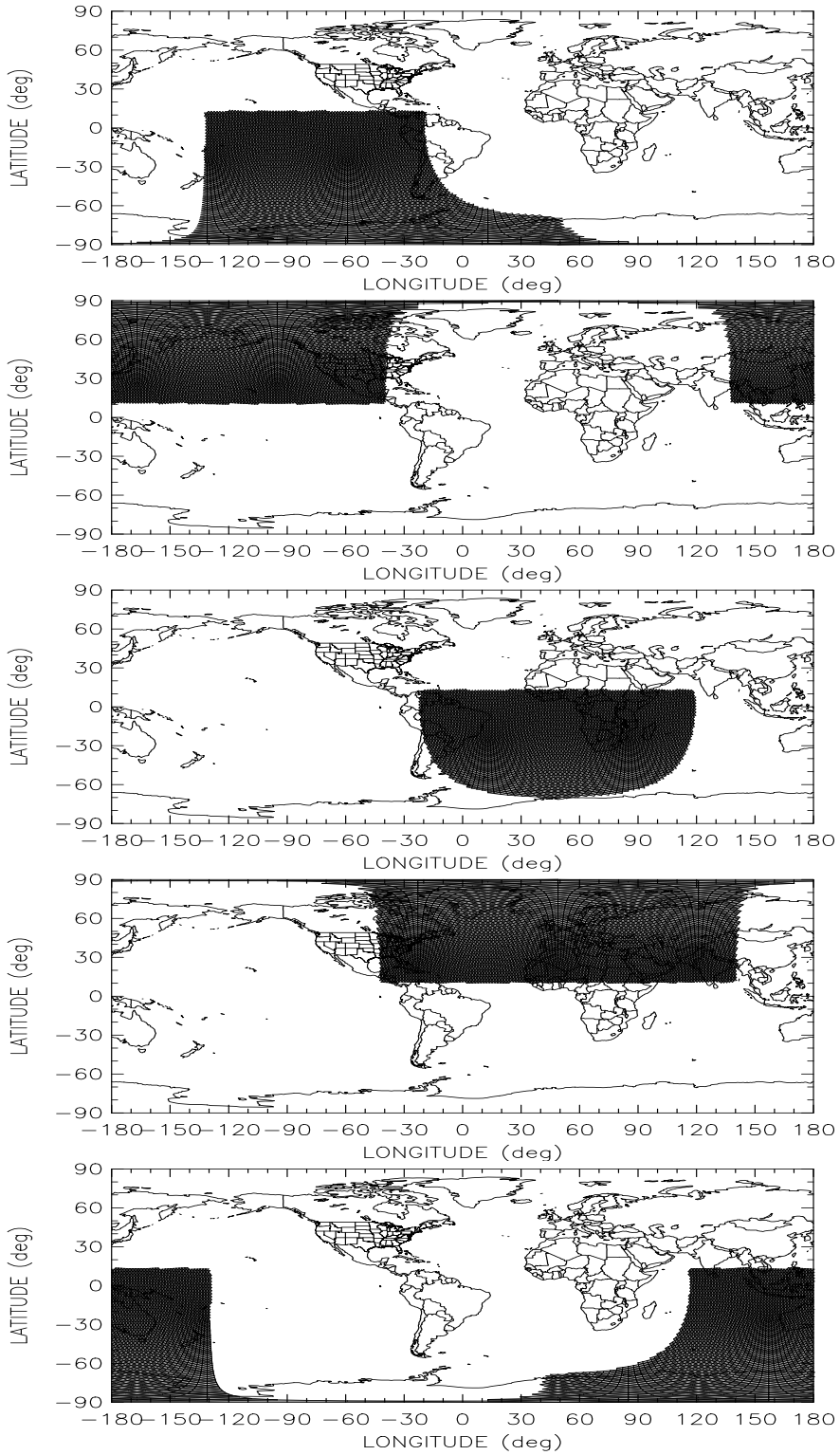

Figura A.2: Partições da grade global, distribuídas entre cinco processadores, do modelo OLAM (a figura superior é relativa ao processador de *rank* 0, a abaixo desta é relativa ao processador de *rank* 1, e assim sucetivamente).

Instâncias	Média	Maior partição	Menor partição	Diferença	Diferença (%)	Desvio Padrão
1	50001	50001	50001	0	0,00	
2	25000	25002	24998	4	0,02	2,83
3	16667	16671	16660	11	0,07	5,86
4	12500	12502	12498	4	0,03	1,63
5	10000	10002	9998	4	0,04	1,87
6	8333	8336	8332	4	0,05	1,63
7	7143	7145	7140	5	0,07	1,95
8	6250	6252	6248	4	0,06	1,51
9	5556	5558	5554	4	0,07	1,33
10	5000	5003	4997	6	0,12	1,83
16	3125	3127	3122	5	0,16	1,55
36	1389	1393	1386	7	0,51	1,51
50	1000	1003	997	6	0,60	1,59
62	806	810	803	7	0,87	1,52
75	667	670	663	7	1,06	1,47
96	521	524	518	6	1,16	1,61
128	391	394	388	6	1,55	1,46
			Média	5,3		

Tabela A.1: Características da distribuição dos pontos do OLAM em diversas partições do processamento paralelo (a coluna "Diferença (%) é a diferença entre a maior e a menor partição com relação ao número de pontos da menor partição, ou seja, quanto a maior partição é maior que a menor partição).

de eventos de comunicação entre as instâncias). Esta sobreposição, como é auxiliar, não é calculada: o modelo calcula as equações prognósticas, diagnósticas, da física e de parametrizações para todos os outros pontos da partição, com exceção da sobreposição.

Se o número destes pontos da grade local (com exceção da sobreposição) for diferente entre as instâncias então algumas terão mais pontos para calcular, levando mais tempo que outras. Haverá um desbalanceamento no processamento que impactará no desempenho paralelo do modelo.

Como forma de averiguar se ocorre este desbalanceamento foi realizado uma análise da distribuição do número de pontos entre as instâncias. Desta forma o modelo foi executado com 1, 2, 3 ou mais processadores, de forma a verificar se o divisor da grade deixa partições com mais pontos que outras[1]. A tabela A.1 mostra algumas características da divisão para diversos processadores.

A tabela mostra que a diferença é ao redor de 5 pontos, entre a menor partição e a maior partição, e mesmo com um número grande de instâncias (128) a diferença é menor que 2%.

Estes valores, para um modelo de escala global com um número considerável de equações para serem resolvidas a cada passo de tempo, são muito pequenos, e o desbalanceamento

[1]Lembrando que a grade não é estruturada, e a divisão é geográfica.

gerado pela divisão é despresível. A divisão de domínio do OLAM, portanto, é bem comportada e não tendenciosa, pelo menos para a configuração da grade utilizada neste trabalho.