MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2015/04.28.19.21-TDI

PORTABILIDADE COM EFICIÊNCIA DE TRECHOS DA DINÂMICA DO MODELO BRAMS ENTRE ARQUITETURAS MULTI-CORE E MANY-CORE

Manoel Baptista da Silva Júnior

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Stephan Stephany, aprovada em 29 de maio de 2015.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3JDAC42>

INPE
São José dos Campos
2015

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2015/04.28.19.21-TDI

PORTABILIDADE COM EFICIÊNCIA DE TRECHOS DA DINÂMICA DO MODELO BRAMS ENTRE ARQUITETURAS MULTI-CORE E MANY-CORE

Manoel Baptista da Silva Júnior

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Stephan Stephany, aprovada em 29 de maio de 2015.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3JDAC42>

INPE
São José dos Campos
2015

Dados Internacionais de Catalogação na Publicação (CIP)

Silva Júnior, Manoel Baptista.

Si38p Portabilidade com eficiência de trechos da dinâmica do modelo BRAMS entre arquiteturas multi-core e many-core / Manoel Baptista da Silva Júnior. – São José dos Campos : INPE, 2015.

xxvi + 64 p. ; (sid.inpe.br/mtc-m21b/2015/04.28.19.21-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2015.

Orientador : Dr. Stephan Stephany.

1. Modelo meteorológico. 2. Processamento paralelo. 3. GPU. 4. OpenMP. 5. OpenACC. I.Título.

CDU 004.272

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

Computação Aplicada

Dr. Haroldo Fraga de Campos Velho

Presidente / INPE / São José dos Campos - SP

Dr. Stephan Stephany

Orientador(a) / INPE / SJCampos - SP

Dr. Solon Venâncio de Carvalho

Membro da Banca / INPE / SJCampos - SP

Dr. Álvaro Luiz Fazenda

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

Dr. Jairo Panetta

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

Este trabalho foi aprovado por:

( ) maioria simples

( ) unanimidade

Título: "Portabilidade com eficiência de trechos da dinâmica do modelo BRAMS entre arquiteturas Multi-Core e Many-Core"

Aluno (a): ***Manoel Baptista da Silva Júnior***

São José dos Campos, 29 de Maio de 2015

“Oh! Quão bom e suave é que os irmãos vivam em união! É como o óleo precioso sobre a cabeça, o qual desce para a barba de Aarão e desce para a gola de suas vestes. É como o orvalho do Hermon, que desce sobre os montes de Sião. Ali ordena o Senhor a sua bênção e a vida para sempre”.

Salmo 133

Este trabalho dedico primeiramente a meu Filho que é a razão e alegria de minha vida. A minha Esposa e Pais que foram a luz e o meu porto seguro em todos os momentos dessa minha jornada. A minha Família, Amigos e Colegas que me incentivaram e me ajudaram em todas as situações.

AGRADECIMENTOS

Primeiramente agradeço ao Dr. Jairo Panetta, a quem se deve a idealização e a análise deste trabalho. Agradeço novamente ao Dr. Jairo Panetta e Dr. Stephan Stephany, pela extrema paciência e dedicação, por acreditarem em mim, apoiando, ensinado e orientando, não só em minha vida acadêmica, mas também me moldando em minha vida profissional, aos Srs. estou e estarei eternamente grato. Aos meus amigos e colegas de trabalho agradeço pela ajuda e compreensão no apoio a essa jornada que fiz.

RESUMO

O aumento contínuo da resolução espacial e temporal dos modelos meteorológicos demanda cada vez mais velocidade e capacidade de processamento. Executar esses modelos em tempo hábil requer o uso de supercomputadores com centenas ou milhares de nós. Atualmente estes modelos são executados em produção no CPTEC em um supercomputador com nós compostos por CPUs com dezenas de núcleos (multi-core). Gerações mais recentes de supercomputadores apresentam nós com CPUs acopladas a aceleradores de processamento, tipicamente placas gráficas (GPGPUs), compostas de centenas de núcleos (many-core). Alterar o código do modelo de forma a usar com alguma eficiência nós com ou sem placas gráficas (código portátil) é um desafio. A interface de programação OpenMP é o padrão estabelecido há décadas para explorar eficientemente as arquiteturas multi-core. Uma nova interface de programação, o OpenACC, foi recentemente proposta para explorar as arquiteturas many-core. Ambas interfaces são semelhantes, baseadas em diretivas de paralelização para execução concorrente de threads. Este trabalho demonstra que é possível escrever um único código paralelizado com as duas interfaces que apresente eficiência aceitável, de forma a poder ser executado num nó com arquitetura multi-core ou então em um nó com arquitetura many-core. O código escolhido como estudo de caso é a advecção de escalares, um trecho da dinâmica do modelo meteorológico regional BRAMS (Brazilian Regional Atmospheric Modelling System).

PORTABILITY WITH EFFICIENCY IN A PART OF DYNAMICS OF THE MODEL BRAMS BETWEEN MULTI-CORE AND MANY-CORE ARCHITECTURES

ABSTRACT

The continuous growth of spatial and temporal resolutions in current meteorological models demands increasing processing power. The prompt execution of these models requires the use of supercomputers with hundreds or thousands of nodes. Currently, these models are executed at the operational environment of CPTEC on a supercomputer composed of nodes with CPUs with tens of cores (multi-core). Newer supercomputer generations have nodes with CPUs coupled to processing accelerators, typically graphics cards (GPGPUs), containing hundreds of cores (many-core). The rewriting of the model codes in order to use such nodes efficiently, with or without graphics cards (portable code), represents a challenge. The OpenMP programming interface proposed decades ago is a standard for decades to efficiently exploit multi-core architectures. A new programming interface, OpenACC, proposed decades ago is the many-core architectures. These two programming interfaces are similar, since they are based on parallelization directives for the concurrent execution of threads. This work shows the feasibility of writing a single code imbedding both interfaces and presenting acceptable efficiency. When executed on nodes with multi-core or many-core architecture. The code chosen as a case study is the advection of scalars, a part of the dynamics of the regional meteorological model BRAMS (Brazilian Regional Atmospheric Modeling System).

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE SIGLAS E ABREVIATURAS

AMD	*Advanced Micro Devices*
API	*Application Program Interface*
ARM	*Advanced RISC Machine*
BRAMS	*Brazilian Regional Atmospheric Modelling System*
CAM-SE	*Community Atmospheric Model - Spectral Element*
CCATT	*Coupled Chemistry Aerosol-Tracer Transport*
CCE	*Cray Compiling Environment*
CCLM	*Climate Limited-area Modelling-Community*
COSMO	*COnsortium for Small scale MOdeling*
CPTEC	Centro de Previsão do Tempo e Estudos Climáticos
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
F2C-ACC	*Fortran to C and C for CUDA*
FIFO	*First In, First Out*
FIM	*Flow-following finite-volume Icosahedral Model*
GCM	*General Circulation Model*
GEOS-5	*Goddard Earth Observing System Model, version 5*
GFDL	*Geophysical Fluid Dynamics Laboratory*
GMAO	*Global Modeling and Assimilation Office*
GPGPU	*General Purpose Graphics Processing Units*
Grads	*Grid Analysis and Display System*
HOMME	*High-Order Method Modeling Environment*
IBM	*International Business Machines*
INPE	Instituto Nacional de Pesquisas Espaciais
IPCC-AR5	*Intergovernmental Panel on Climate Change – Fifth Assessment Report*
LMDZ	*Laboratoire de Météorologie Dynamique*; Onde a letra “Z” significa “*zoom*”
MCGA	Modelo de Circulação Geral Atmosférico

MIC	*Many Integrated Core*
MPI	*Message Passing Interface*
NASA	*National Aeronautics and Space Administration*
NCAR	*National Center for Atmospheric Research*
NIM	*Non-hydrostatic Icosahedral Model*
NOAA	*National Oceanic and Atmospheric Administration*
OpenACC	*Open Accelerators*
OpenCL	*Open Computing Language*
OpenGL	*Open Graphics Library*
OpenMP	*Open Multi-Processing*
ORNL	*Oak Ridge National Laboratory*
PGI	*Portland Group, Inc.*
RAMS	*Regional Atmospheric Modeling System*
SM	*Streaming Multiprocessors*
SSAI	*Science Systems and Applications, Inc.*
WRF	*Weather Research and Forecasting*

LISTA DE SÍMBOLOS

GHz	É uma unidade de medida de frequência, equivale a um bilhão de ciclos por segundo (*Gigahertz*).
GB	É uma unidade de medida de informação (*Gigabyte*), segundo o Sistema Internacional de Unidades, que equivale a um bilhão de *bytes.*
Tflop/s	Corresponde a um trilhão de operações de ponto flutuante por segundo
MHz	É uma unidade de medida de frequência, equivale a um milhão de ciclos por segundo (*Megahertz*).
W	É uma unidade de potência do Sistema Internacional de Unidades e equivale a um joule por segundo.
km	É uma unidade medida de comprimento do Sistema Internacional de Unidades (quilómetro).

SUMÁRIO

Pág.

1 INTRODUÇÃO

Entre os avanços científicos mais importantes do século passado está a capacidade de simular sistemas físicos complexos e prever sua evolução temporal usando modelos numéricos, tendo como exemplo marcante o desenvolvimento dos modelos de circulação geral da atmosfera e dos oceanos (LYNCH, 2007). Esses modelos permitem prever o tempo com dias de antecedência com alto grau de confiabilidade (previsão de tempo) ou o clima com meses de antecedência (previsão climática), bem como acompanhar o estado da atmosfera em simulações longas que abrangem até centenas de anos.

Os modelos numéricos de previsão de tempo e clima podem ser globais ou regionais (KALNAY, 2003). Um modelo global abrange todo o globo terrestre, sendo que o CPTEC (Centro de Previsão e Estudos Climáticos) utiliza o modelo global MCGA (Modelo de Circulação Geral Atmosférico). Os modelos regionais abrangem um trecho do globo terrestre, permitindo simulações com alta resolução em tempo hábil. O CPTEC emprega os modelos regionais ETA e BRAMS (*Brazilian Regional Atmospheric Modeling System*), tendo como domínio toda a América do Sul (INPE.CPTEC, 2015).

Um modelo numérico de previsão de tempo e clima utiliza uma grade tri-dimensional, composta por uma malha horizontal e uma malha vertical, discretizando o domínio de interesse em milhares de células, onde cada célula corresponde a um pequeno volume, armazenando um único valor para cada variável atmosférica (OLSON *et al.*, 1995). A malha horizontal representa latitudes e longitudes enquanto a malha vertical representa alturas. As variáveis atmosféricas são denominadas *campos*.

Um modelo numérico constitui-se de diversos módulos. Os meteorologistas classificam os módulos principais em módulos da dinâmica e da física. A dinâmica abrange todos os módulos que governam o movimento do fluido atmosférico, resolvendo as equações de Navier-Stokes e as equações

termodinâmicas. A física representa os processos de sub-grade escalar e outros processos não incluídos na dinâmica (THUBURN, 2008) tais como trocas de calor com a superfície, microfísica de nuvens, convecção, entre outras, as quais constituem as forçantes da equação de Navier-Stokes.

O aumento contínuo da resolução espacial e temporal dos modelos meteorológicos demanda cada vez mais processamento. A velocidade de processamento necessária só é atingida pelo uso de supercomputadores com centenas de nós multiprocessados. Cada nó contém CPUs compostas de poucas dezenas de núcleos (*multi-core*). Esses modelos são tipicamente paralelizados por meio da biblioteca de comunicação por troca de mensagens MPI (*Message Passing Interface*), com divisão de domínio que explora indistintamente paralelismo inter-nó e intra-nó.

Alternativamente, pode-se substituir o paralelismo MPI intra-nó pelo uso da interface de programação OpenMP que permite a execução concorrente de múltiplos *threads* em um único nó (LEE *et al.*, 2009). A interface de programação OpenMP permite incluir diretivas de paralelização no programa sequencial, as quais são interpretadas como comentários caso o programa seja compilado para execução sequencial. Além das diretivas, o OpenMP inclui as correspondentes bibliotecas e variáveis de ambiente.

Recentemente estão disponíveis nós multiprocessados munidos de aceleradores de processamento, tipicamente placas gráficas compostas de centenas de núcleos (*many-core*), usualmente denominadas GPGPU (*General-Purpose Graphics Processing Units*). Um nó computacional que contém CPUs *multi-core* e GPGPUs *many-core* é classificado como nó de arquitetura heterogênea. A tecnologia de *many-core* apresenta milhares de núcleos simples que compartilham poucas unidades de controle. Essa nova tecnologia vem se popularizando pelo baixo custo do hardware face ao seu alto desempenho.

1.1. Uso de GPGPU na execução de modelos meteorológicos

Nota-se uma tendência crescente de uso de aceleradores de processamento como GPGPUs na execução de modelos meteorológicos paralelizados com MPI. Para programar as GPGPUS, além da linguagem CUDA (Computação de Alta Performance, 2014), há a possibilidade de utilizar o OpenACC (*Open ACCelerator*), composto por um conjunto de diretivas de paralelização para as linguagens Fortran, C e C++, bem como as bibliotecas e variáveis de ambiente associadas. Enquanto CUDA é uma linguagem proprietária da NVIDIA, o OpenACC é uma linguagem não proprietária proposta pela Cray Inc, NVIDIA, The Portland Group e a CAPS Enterprise (Open: The Origins of OpenACC, 2014). Enquanto CUDA requer o uso de equipamento NVIDIA, o OpenACC permite portabilidade entre GPGPUs de outros fabricantes e aceleradores em geral (como Intel Xeon Phi), permitindo codificar num nível de abstração mais alto, à semelhança do padrão OpenMP (OpenACC, 2015).

Outra linguagem que provê uma programação paralela heterogênea utilizando CPU e GPU é o OpenCL (Khronos Group, 2015), mas não é usada pela comunidade de modelos meteorológicos devido à dificuldade de conversão: o OpenCL utiliza como base a linguagem C, enquanto que esses modelos são escritos preponderantemente na linguagem Fortran 90.

A comunidade internacional que pesquisa o uso de novas arquiteturas de computadores em modelos meteorológicos encontra-se anualmente em eventos específicos, um deles no NCAR (*National Center for Atmospheric Research*) em Boulder, Colorado. Nesse evento, denominado *Programming Weather, Climate, and Earth-System Models on Heterogeneous Multi-Core Platforms Workshop* (Heterogeneous Multi-Core Workshop, 2014), os organizadores convidam os principais desenvolvedores a apresentarem suas pesquisas mais recentes no uso de arquiteturas heterogêneas e discuti-las entre colegas. Em consequência, o conteúdo desse evento representa

adequadamente o estado da arte do uso de arquiteturas de computadores *multi-core* e *many-core* em modelos meteorológicos.

As edições de 2012 a 2014 desse evento marcam o início do uso de GPGPU na execução de modelos meteorológicos de produção. Observaram-se esforços para adaptação dos códigos de diversos modelos meteorológicos para as linguagens CUDA e OpenACC, abrangendo componentes da física e da dinâmica de modelos meteorológicos. A Tabela 1.1 sumariza as principais apresentações nesse evento ao longo dos anos.

Na edição de 2012 do *Programming Weather, Climate, and Earth-System Models on Heterogeneous Multi-Core Platforms Workshop*, constatou-se que apenas um modelo havia sido portado para arquiteturas híbridas, o COSMO-CCLM (utilizado para gerar resultados para o IPCC AR5), ao custo da perda de portabilidade: a dinâmica desse modelo foi recodificada em CUDA. Nessa data, trechos de dois outros modelos haviam sido parcialmente portados para arquiteturas híbridas (CAM-SE da ORNL e o GEOS-5 GCM da SSAI/NASA/GMAO) ambos com perda de portabilidade pelo uso de CUDA. Portes parciais haviam sido obtidos pelo uso de um compilador de pesquisa (F2C-ACC), esforço posteriormente descontinuado.

Na edição de 2013 do *Programming Weather, Climate, and Earth-System Models on Heterogeneous Multi-Core Platforms Workshop*, utilizou-se o CUDA Fortran para as componentes da física e da dinâmica dos modelos CAM-SE e LMDZ e OpenACC para as componentes da física dos modelos GEOS-5 GCM e o GFDL. Na edição de 2014, não houve mudanças significativas, com a advecção do modelo HOMME paralelizada com CUDA e trechos da componente da física do modelo WRF portados para OpenACC, bem como do modelo CAM-SE, obtendo-se eficiência, mas sem portabilidade (Heterogeneous Multi-Core Workshop, 2014).

Entretanto, deve-se notar que existem trabalhos similares que não foram apresentados nas edições desse evento, como por exemplo, a codificação para

execução em GPU da parametrização turbulenta do modelo acoplado CCATT-BRAMS (RUIZ *et al.*, 2013).

Tabela 1.1 – Evolução do uso de linguagens paralelas para uso de GPGPUs na execução de modelos meteorológicos conforme o evento *Programming Weather, Climate, and Earth-System Models on Heterogeneous Multi-Core Platforms Workshop*

2012		
Modelos	**Física**	**Dinâmica**
COSMO – CCLM	OpenACC	CUDA
CAM-SE		CUDA
GEOS-5 GCM	CUDA Fortran	CUDA (para o futuro) Possibilidades em OpenACC
NIM – NOAA		F2C-ACC (1 rotina)
FIM – NOAA		F2C-ACC (3 rotinas)
WRF – NOAA	F2C-ACC (1 rotina)	
2013		
GFDL – NOAA	OpenACC (única coluna)	OpenACC (projeto) MPI-OpenMP/MIC (em estudo)
CAM-SE	Portando todo o código para CUDA Fortran	
GEOS-5 GCM	CUDA Fortran (quase concluído) OpenACC (turbulência)	Conversões no começo
NIM – NOAA	Portado para arquitetura MIC	
WRF – NOAA	CUDA e MIC	
LMDZ – CAS/IIT	CUDA Fortran	CUDA Fortran
2014		
HOMME		Advecção (*Traces*) em CUDA OpenACC em andamento
WRF	Trechos em OpenACC	
CAM-SE	Transportando para OpenACC	

Pode-se verificar que em nenhum dos modelos acima citados foi proposto um único código portátil para uso em arquitetura *multi-core* e *many-core*. Para obter portabilidade foram empregados artifícios como pré-processamento (que

recorre a estruturas do tipo "*IF DEF*") e a criação de bibliotecas recodificadas para cada uma das duas arquiteturas. Mas a tendência predominante é abdicar da portabilidade e utilizar dois códigos diferentes, um para cada arquitetura, principalmente na dinâmica dos modelos, justamente o trecho com paralelismo mais complexo. Consequentemente, surgiu a motivação para desenvolver o presente trabalho: será possível gerar um único código para o trecho mais difícil da paralelização (a dinâmica) e que explore eficientemente as duas arquiteturas?

1.2. Objetivos e contribuição

Modelos meteorológicos têm códigos com centenas de milhares de linhas paralelizados por MPI. Portar esses códigos para execução em arquiteturas *multi-core* e *many-core* por recodificação (como em CUDA) deve ser evitado, dado o volume de trabalho e a dificuldade de manutenção de duas versões do programa fonte ao longo do tempo. Já padrões como o OpenMP (*multi-core*) ou OpenACC (*many-core*) permitem a paralelização mantendo um único programa fonte, pois alteram o código sequencial apenas pela inserção de diretivas de paralelização.

Este trabalho propõe de forma inédita o porte de parte do código original da dinâmica de um modelo meteorológico para execução numa ou noutra arquitetura mantendo um único programa fonte pela inclusão de diretivas de paralelização tanto para OpenMP como para OpenACC. No caso do OpenMP, exploram-se duas técnicas de paralelização: por laços e por telhas. Em particular, o estado da arte do uso de aceleradores de processamento para a execução da dinâmica de modelos regionais não apresenta nenhum caso de uso de OpenACC e OpenMP para trechos da dinâmica, tal como aqui proposto. O código escolhido como estudo de caso é o módulo de advecção de escalares do modelo meteorológico regional de alta resolução BRAMS, que é utilizado operacionalmente no CPTEC para previsões de tempo regional. Busca-se

portabilidade sem recorrer aos artifícios descritos na seção anterior. Busca-se ainda obter portabilidade com eficiência aceitável.

A versão corrente do modelo BRAMS está paralelizada por MPI para execução em múltiplos nós multiprocessados. A paralelização MPI correntemente empregada no BRAMS divide o domínio do modelo em partições da grade de pontos entre os nós e, internamente a cada nó, cada partição é novamente subdividida para execução pelos múltiplos núcleos dos processadores do nó correspondente. Entretanto, constatou-se que isso limita a escalabilidade do modelo devido aos custos de comunicação inter-nó, que é imprescindível, e da comunicação intra-nó, sendo desejável eliminar este último custo.

Dentro da proposta apresentada, explora-se a paralelização por OpenMP ou OpenACC aplicada à partição da grade de pontos que cabe a cada nó multiprocessado no domínio de um único processo MPI. Os testes realizados referem-se à execução da advecção para essa partição da grade de pontos do domínio num único nó multiprocessado com uma CPU *multi-core* e com uma GPGPU *many-core*.

Esta dissertação está organizada nos capítulos a seguir:

- Capítulo 2: Modelos Meteorológicos, com ênfase no modelo BRAMS;
- Capítulo 3: Interfaces e linguagens de programação paralela, abrangendo o OpenMP e o OpenACC, além de descrever o ambiente computacional utilizado;
- Capítulo 4: Desempenho paralelo com OpenMP, detalhando as versões do código paralelizadas por laços ou por telhas e seu desempenho paralelo;
- Capítulo 5: Desempenho paralelo com OpenACC, detalhando as otimizações das sucessivas versões paralelas desenvolvidas e seu desempenho paralelo;

- Capítulo 6: Conclusões e trabalhos futuros.

2 MODELO METEOROLÓGICO BRAMS

O BRAMS é um modelo regional baseado no modelo RAMS (*Regional Atmospheric Modeling System*). O desenvolvimento do modelo RAMS iniciou-se em 1970 por Willian R. Cotton e Roger A. Pielke. A partir de 2003 o CPTEC desenvolveu sucessivas versões do modelo BRAMS e passou a utilizá-lo operacionalmente até os dias de hoje (PANETTA, 2012). Além do seu uso operacional, o BRAMS tem sido utilizado como veículo de pesquisas em Ciência da Computação (FAZENDA *et al.*, 2011), (RODRIGUES *et al.*, 2010) e em Meteorologia (FREITAS *et al.*, 2007), (LONGO *et al.*, 2013). Longa lista de publicações está disponível no próprio sítio do BRAMS (BRAMS, 2015).

O modelo BRAMS utiliza equações não hidrostáticas quasi-Boussinesq (TRIPOLI; COTTON, 1982). Utiliza um esquema de aninhamento múltiplo que permite resolver simultaneamente um aninhamento de grades computacionais (permite criar uma grade detalhada no interior de uma grade maior). Sua grade horizontal é Arakawa tipo C (MESINGER; ARAKAWA, 1976). Possui módulos de transferência radiativa, balanços das trocas de água, calor e momento entre a superfície e a atmosfera, transporte turbulento na camada limite planetária e microfísica das nuvens (TREMBACK *et al.*, 1987). Suas condições iniciais e de contorno são provenientes de um modelo atmosférico com grade de maior abrangência horizontal. O CPTEC utiliza, para tal fim, saídas do modelo global MCGA.

O modelo BRAMS evoluiu ao longo dos anos, sendo acoplado a um modelo de transporte e a um modelo de química para gases traço e aerossóis (FREITAS *et al.*, 2006), resultando no modelo acoplado CCATT-BRAMS (*Coupled Chemistry Aerosol-Tracer Transport - BRAMS*). O código fonte do modelo BRAMS tem aproximadamente 350.000 linhas na linguagem Fortran 2003 padrão e está paralelizado com a biblioteca de comunicação por troca de mensagens MPI (GROPP *et al.*, 1999). Atualmente, o modelo BRAMS é

executado diariamente na previsão operacional de tempo regional no CPTEC, utilizando 9.600 núcleos computacionais.

Na execução do modelo BRAMS, a parcela de tempo da dinâmica divide-se em: 25% na advecção de escalares, 5% na advecção de velocidades, 40% na difusão, 25% na acústica e 5% no restante.

2.1. Decomposição do domínio

A versão do modelo utilizada neste trabalho é a BRAMS 5, a qual possui um algoritmo que divide automaticamente o domínio entre os processos MPI. A divisão de domínio é estática, isto é, o algoritmo divide o domínio no início da execução do BRAMS e essa divisão é mantida fixa durante o restante de sua execução. Cada processo trabalha unicamente no trecho do domínio que lhe é alocado pelo algoritmo, e quando necessário, troca dados da sua fronteira com processos vizinhos. A quantidade de trabalho por processo é função do número de células do trecho do domínio que lhe foi atribuído. Consequentemente, o tempo de execução de cada processo será proporcional a esse número de células.

O arquivo de entrada ("*namelist*") que descreve o problema a ser resolvido pelo BRAMS contém as variáveis $nnxp$ e $nnyp$ que definem o número de células discretas nas direções x e y da grade de execução, as quais são numeradas de 1 a $nnxp$ e de 1 a $nnyp$. As células no extremo da grade são reservadas para condições de contorno. Consequentemente, o domínio a ser decomposto pelos processos MPI é dado por $[2: nnxp - 1] \times [2: nnyp - 1]$ não havendo divisão do domínio na direção z, ou seja, cada célula discreta no plano horizontal corresponde na verdade a uma coluna vertical de células.

O número de processos p em uma execução MPI é definido pelo valor da chave np do comando *mpirun* que dispara a computação. Sejam esses p processos numerados de 1 a p, entre os quais o domínio será particionado. Consequentemente, o algoritmo de divisão de domínio deve particionar o

domínio $[2:nnxp-1] \times [2:nnyp-1]$ em p partições com número de colunas de células aproximadamente iguais. Cada partição é especificada pelo número da primeira e da última coluna de células nos eixos x e y. O algoritmo exige que cada partição tenha no máximo uma partição vizinha em cada lado no eixo x, mas permite múltiplas partições vizinhas no eixo y.

O algoritmo inicia particionando o eixo y ($[2:nnyp-1]$) em um número de partes proporcional à raiz quadrada do número de processos. Essas partes são denominadas tiras. A proporção é a razão do número de células da grade nos eixos x e y. Em seguida, o algoritmo particiona o eixo x ($[2:nnxp-1]$) de cada tira, de tal forma que o total de partições sobre todas as tiras seja p, que é o número de processos MPI. O resultado é um conjunto de partições que tem no máximo um vizinho de cada lado na direção x (pois cada partição está em uma tira) e múltiplos vizinhos na direção y (pois a partição de uma tira é independente da partição das outras tiras). A quantidade de processos MPI por tira varia com o tamanho da tira em y, sendo menos processos atribuídos a tiras “finas” e mais processos atribuídos a tiras “grossas”.

A Figura 2.1 abaixo ilustra a decomposição de domínio quando o algoritmo é aplicado a 8 processos.

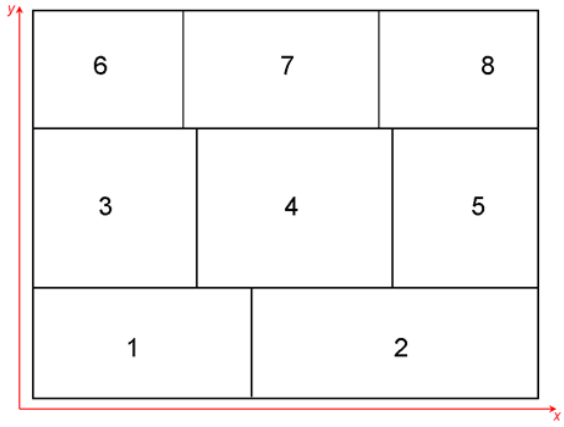


Figura 2.1 – Decomposição de domínio MPI para 8 processos feita pelo algoritmo correspondente do BRAMS 5

A partição, ou seja, o domínio atribuído a cada processo, é aumentada nas suas bordas por faixas constituídas por colunas de células, de forma a acomodar "*ghost zones*", que reproduzem colunas de células vizinhas de outras partições, além de condições de contorno. Considerando-se a Figura 2.1, nota-se que processos internos à grade, como por exemplo o processo 4, tem sua partição ampliada por duas faixas constituídas por colunas de células adicionais ao longo de cada direção (faixas ao longo das bordas esquerda e direita e faixas ao longo das bordas superior e inferior) para acomodar as "*ghost zones*" dos domínios dos processos vizinhos 3, 7, 8, 5, 2 e 1. Por outro lado, o processo 3 tem sua partição ampliada para acomodar as "*ghost zones*" dos processos 1, 6 e 7 (faixas ao longo das bordas superior e inferior), para acomodar a "*ghost zone*" do processo 4 (faixa ao longo da borda direita) e também para acomodar a condição de contorno (faixa ao longo da borda esquerda).

2.2. Etapas de execução do modelo BRAMS

A execução do modelo BRAMS é estruturada nas quatro etapas mostradas na Figura 2.2.

A primeira etapa, denominada MAKESFC, converte dados da superfície terrestre (topografia, cobertura do solo, textura do solo, índice de vegetação por diferença normalizada e temperatura de superfície do mar) para a área requerida e o formato necessário.

A segunda etapa, denominada MAKEVFILE, converte os arquivos de condições iniciais e de contorno advindos do modelo global para a resolução adequada na área geográfica desejada.

A terceira etapa, denominada INITIAL, gera o estado da atmosfera nos instantes futuros desejados e na área geográfica pré-definida, utilizando os arquivos produzidos nas duas etapas anteriores. Os arquivos resultantes desta etapa são denominados arquivos de análise.

A quarta etapa converte os arquivos de analise para o formato binário de leitura de um programa de visualização como o Grads (About GrADS Gridded Data Sets, 2015).

Toda a configuração do modelo é definida pelo arquivo RAMSIN, lido pelo BRAMS no início de cada fase da execução ("*namelist*" Fortran). Como o modelo é configurado dinamicamente, não há a necessidade de recompilar o modelo a cada nova configuração (ALMEIDA; BAUER, 2012).

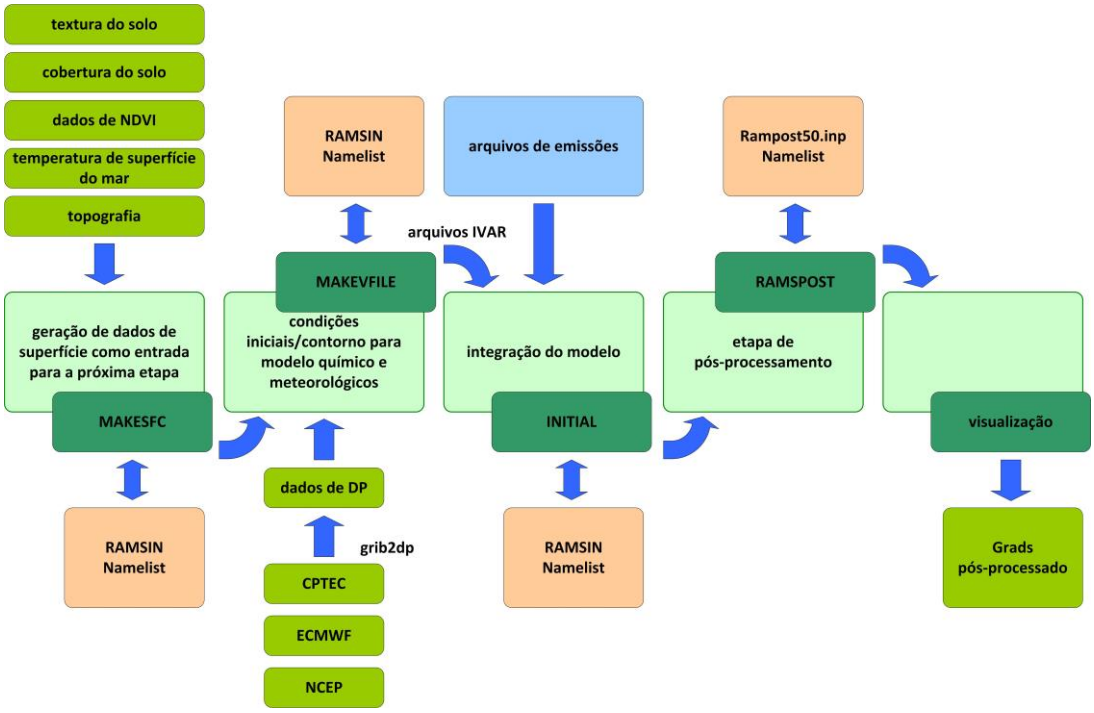


Figura 2.2 – Diagrama do modelo BRAMS.

Fonte: Adaptado de (ALMEIDA; BAUER, 2012)

2.3. Rotina *timestep* do modelo BRAMS

A fase INITIAL do BRAMS executa uma árvore de invocações a rotinas que tem em alto nível hierárquico uma rotina chamada *timestep*. Essa rotina, como o nome indica, avança o estado da atmosfera um passo de tempo.

Em execuções sequenciais cada invocação da *timestep* atua sobre todos os pontos do domínio. Em execuções paralelas o domínio horizontal é particionado entre os diversos processos MPI.

A rotina *timestep* executa uma sequência de chamadas a rotinas que compõem os módulos da física e da dinâmica do modelo, conforme ilustrado na Figura 2.3. Após a execução de determinadas rotinas há necessidade de comunicação entre os processos MPI devido às dependências de dados entre subdomínios adjacentes, conforme a mesma figura.

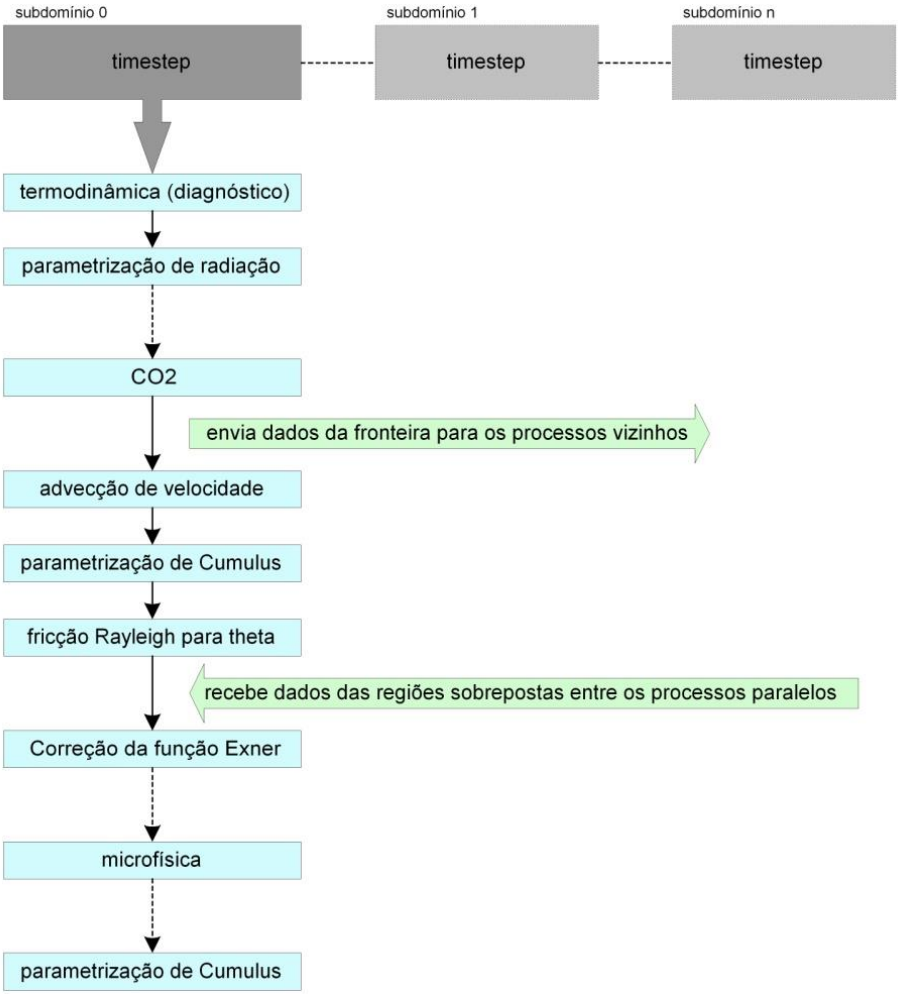


Figura 2.3 – Fluxograma parcial da rotina *timestep*.

2.4. Módulos do modelo

Um modelo de previsão de tempo constitui-se de diversos módulos. Os meteorologistas classificam os módulos principais em módulos da dinâmica e da física (Figura 2.4). A dinâmica abrange todos os módulos que governam o movimento do fluido atmosférico e as equações termodinâmicas (resolvem as equações de Navier-Stokes). A física representa os processos de sub-grade escalar e outros processos não incluídos na dinâmica (THUBURN, 2008) tais como trocas de calor com a superfície, microfísica de nuvens, convecção, etc. (as forçantes da equação de Navier-Stokes).

Cada módulo do BRAMS calcula a tendência (derivada temporal do instante presente para o instante futuro) de uma variável meteorológica devida ao fenômeno modelado. Ao término da *timestep*, o estado futuro da atmosfera é composto a partir do estado atual e da soma das tendências de todos os módulos sobre cada variável.

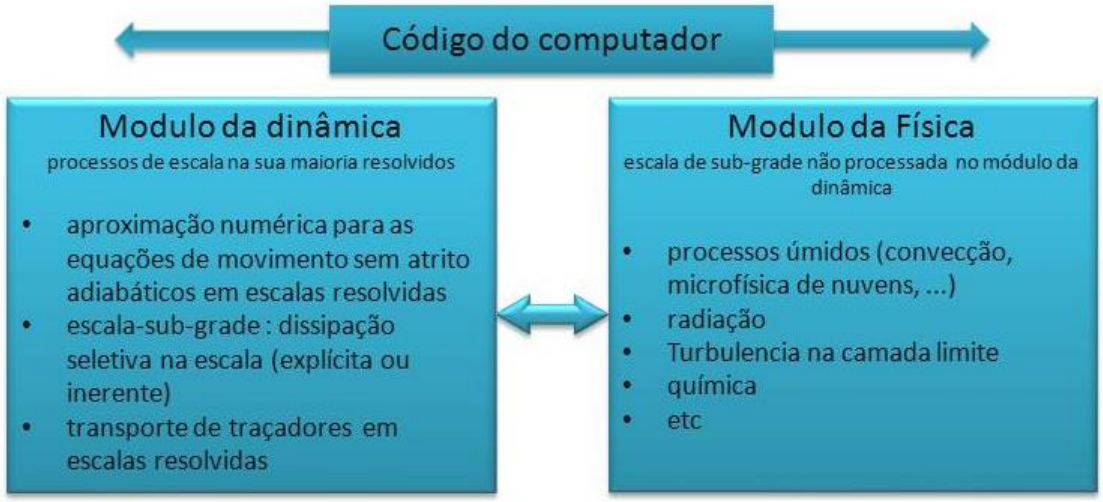


Figura 2.4 – Representação das partes numéricas de um modelo.

Fonte: Adaptado de (LAURITZEN, 2009)

2.5. Modulo de advecção

A advecção é um dos módulos que compõem a dinâmica. Como todos os outros módulos, a advecção é invocada pela *timestep.* Enquanto a maioria dos módulos é invocada uma única vez a cada execução da *timestep*, a advecção é invocada duas vezes (Figura 2.5). Uma invocação é usada para calcular uma grandeza vetorial (a velocidade do ar), que é representada por três componentes nas direções dos eixos cartesianos. A outra invocação calcula as grandezas escalares, como a temperatura e a concentração de poluentes, que são representadas por valores únicos em cada célula da grade.

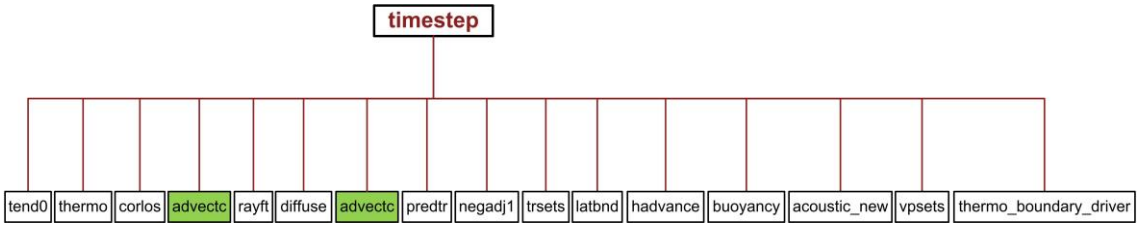


Figura 2.5 – Rotina *timestep* (componentes da dinâmica).

2.5.1. Formulação da advecção de campos escalares

A advecção tridimensional do BRAMS é obtida por três advecções unidimensionais sucessivas em x, y e z, nessa ordem. As advecções são acumulativas, ou seja, o campo resultante das advecções nas direções já contabilizadas é utilizado para a advecção na próxima direção.

A advecção em qualquer uma das três dimensões utiliza a formulação de segunda ordem unidimensional descrita em (TREMBACK *et al.*, 1987), sumarizada a seguir.

Seja $\emptyset$ um campo escalar e u a componente da velocidade na direção x. A equação da advecção unidimensional em forma de fluxo é:

$$\frac{\partial \emptyset}{\partial t} = \frac{-\partial u \emptyset}{\partial x}, \tag{2.1}$$

a qual é discretizada utilizando diferenças finitas por:

$$\emptyset_j^{n+1} = \emptyset_j^n + \frac{\Delta t}{\Delta x}\left[F_{j+\frac{1}{2}} - F_{j-\frac{1}{2}}\right], \tag{2.2}$$

onde $F_{j-\frac{1}{2}}$ e $F_{j+\frac{1}{2}}$ representam o fluxo $u\emptyset$ ao longo do tempo Δt nas bordas da célula discretizada.

O fluxo é aproximado pela integral de um polinômio de Lagrange do segundo grau que interpola $\emptyset$:

$$F_{j+\frac{1}{2}} = \frac{1}{\Delta x}\int_{x_{j+\frac{1}{2}}-u\Delta x}^{x_{j+\frac{1}{2}}} P\left[x, \emptyset_j^n\right] dx \tag{2.3}$$

A expressão resultante para o fluxo é:

$$F_{j+\frac{1}{2}\frac{\Delta t}{\Delta x}} = \frac{\propto}{2}\left(-\emptyset_j - \emptyset_{j+1}\right) + \frac{\propto^2}{2}\left(-\emptyset_j + \emptyset_{j+1}\right), \tag{2.4}$$

onde $\propto = u\frac{\Delta t}{\Delta x}$.

2.5.2. Estrutura do código da advecção de campos escalares

A rotina que realiza a advecção de campos escalares (denominada *advectc*) calcula a advecção de todos os campos escalares a cada invocação. A rotina é composta por uma inicialização comum a todos os campos escalares seguida por um laço que percorre os campos escalares, calculando a advecção de um único campo escalar a cada iteração do laço.

A inicialização (rotina *fa_preptc*) computa variáveis intermediárias independentes do campo escalar, amazenadas em *arrays* tridimensionais que dependem da geometria da grade, da projeção estereográfica, do passo no tempo e da velocidade do vento, dentre outros. A inicialização elimina cálculos

repetitivos, fatorando esses cálculos para fora do laço que percorre os campos escalares. A inicialização é custosa pois calcula 11 campos tridimensionais por meio de 13 aninhamentos que percorrem todo o domínio do processo MPI (horizontal e vertical).

Cada iteração do laço que percorre os campos escalares começa copiando os valores correntes do campo escalar para um *array* intermediário que será advectado. Segue invocando três procedimentos que advectam nas direções *x*, *y* e *z* o *array* intermediário (rotinas *fa_xc*, *fa_yc* e *fa_zc*). Essas rotinas têm a mesma estrutura de cálculos, diferenciadas apenas em suas direções. A iteração termina com a invocação de um procedimento que calcula a tendência devida à advecção (rotina *advtndc*), utilizando os valores do *array* intermediário e os valores correntes, posto que estes não foram alterados pela advecção.

Cada um dos três procedimentos que calcula a advecção unidimensional é composto por dois aninhamentos que percorrem todo o domínio do processo MPI. O primeiro aninhamento calcula os fluxos nas faces das células. O segundo procedimento utiliza os fluxos para calcular a advecção. Já o procedimento que calcula a tendência contém um único laço que percorre todo o domínio do processo MPI.

3 INTERFACES E LINGUAGENS DE PROGRAMAÇÃO PARALELA

3.1. OpenMP

O OpenMP é uma API (*Application Program Interface*) desenvolvida para execução paralela de *threads* em máquinas de memória compartilhada (COSTA; SENA, 2008), (CHAPMAN *et al.*, 2008).

O sucesso do OpenMP é devido à proliferação de arquiteturas *multi-core*, à sua robustez em termos de programação e à facilidade do seu uso. Fazendo-se uso do OpenMP, um encadeamento de instruções, que corresponde ao *master thread*, é dividido em múltiplos *threads* (*slave threads*) na execução de trechos paralelos do programa. Os *threads* são executados concorrentemente nos múltiplos núcleos da máquina de memória compartilhada, conforme ilustrado na Figura 3.1. Os trechos de execução paralela são definidos por diretivas de paralelização segundo o padrão OpenMP.

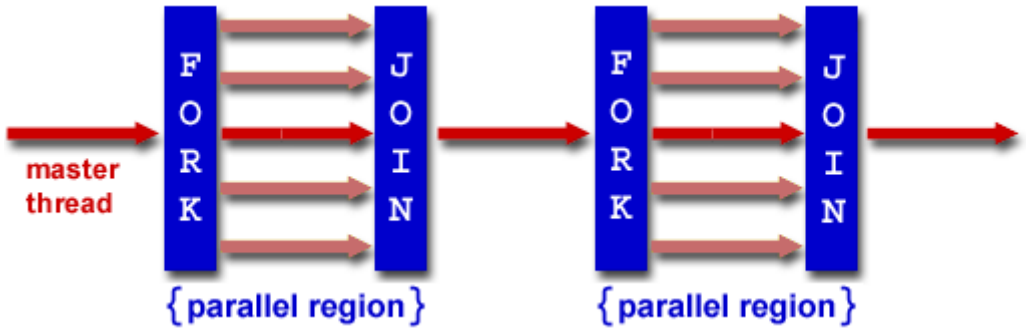


Figura 3.1 – Modelo de programação OpenMP (*fork-join*).
Fonte: http://www.ocgy.ubc.ca/~yzq/books/OpenMP.html (2007)

Nas regiões paralelas do programa, cada *thread* executa sua própria sequência de instruções, porém (de maneira geral) compartilha as mesmas variáveis das demais *threads*, não havendo portanto a necessidade de troca de mensagens como no caso do MPI.

As vantagens da programação em OpenMP são: poucas alterações no código serial existente, uma robusta estrutura para suporte à programação paralela, fácil compreensão e uso das diretivas, suporte a paralelismo aninhado e o ajuste dinâmico do número de *threads* (Figura 3.2).

```
!$omp do private(j,i,k), collapse(2)
do j = ja-1, jz+1
   do i = ia-1, iz
      do k = 1,mzp
         vt3da(k,i,j) = (up(k,i,j) + uc(k,i,j)) * dtlto2
      end do
   end do
end do
!$omp end do nowait
```

Figura 3.2 – Exemplo de laço em Fortran com diretivas OpenMP

Suas desvantagens são a escalabilidade limitada pela arquitetura de memória, requerer um compilador com suporte a OpenMP, carecer de mecanismo refinado para controlar o mapeamento *thread-processor* e não conter uma manipulação segura do erro.

3.2. Interfaces de programação para placas gráficas

Com o surgimento das placas gráficas que possuem um poder de processamento suficientemente grande para gerar gráficos de jogos, cientistas de computação começaram a usar essas placas para acelerar diversos aplicativos científicos (Computação de Alta Performance, 2014).

Em 2006 a NVIDIA, no intuito de facilitar a programação que empregava linguagens gráficas como OpenGL (*Open Graphics Library*), criou uma linguagem proprietária chamada CUDA (*Compute Unified Device Architecture*), baseada na linguagem C (Computação de Alta Performance, 2014).

O OpenCL, inicialmente desenvolvido pela Apple, foi incorporada em junho de 2008 pela Khronos Group. O OpenCL prove uma interface homogênea para a

exploração da computação paralela em CPUs (Intel, IBM, ARM, AMD) e GPGPUs da AMD e Nvidia (Khronos Group, 2015).

3.2.1. OpenACC

No dia 14 de novembro de 2011, durante o SC2011 (*Super Computing 2011*) em Seattle a Cray Inc, NVIDIA, The Portland Group e a CAPS Enterprise anunciaram um novo padrão de programação para GPGPUs batizado de OpenACC, tendo como objetivo tomar mais fácil a programação em placas gráficas.

As APIs do OpenACC descrevem um conjunto de diretivas para laços e regiões de código nos padrões Fortran, C e C++. As diretivas transferem as computações selecionadas da CPU para a GPGPU, deixando todo o trabalho de comunicação da CPU com a GPGPU por conta do compilador. A Figura 3.3 expõem um exemplo prático de programação paralela por diretivas de OpenACC para a linguagem Fortran.

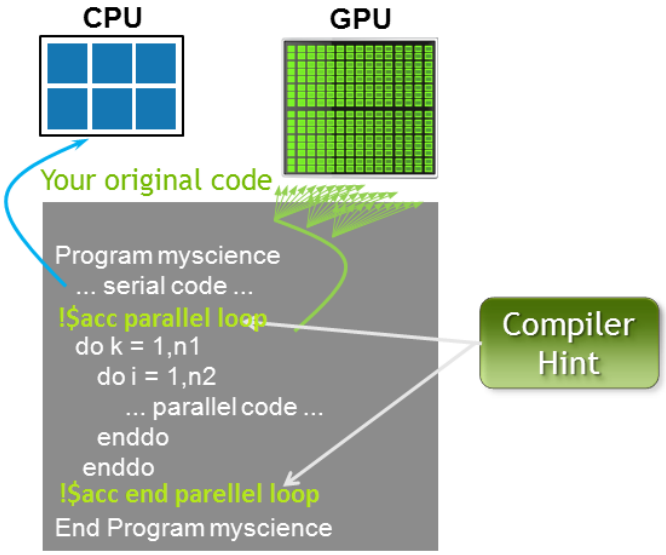


Figura 3.3 – Exemplo de laço em Fortran com diretivas OpenACC.
Fonte: https://developer.nvidia.com/openacc (2015)

Uma diferença expressiva entre a programação exclusiva na CPU e a programação mista entre CPU e GPGPU é que a memória da GPGPU é completamente separada da memória da CPU. O movimento de dados entre as memórias da CPU e da GPGPU é programado explicitamente, invocando rotinas de bibliotecas de comunicação que são executadas em tempo real.

Na programação em CUDA e em OpenCL, fica a cargo do programador gerenciar essa transferência de dados entre as memórias. No OpenACC o movimento de dados entre as memórias é implícito e gerenciado pelo compilador, com base em diretrizes do programador.

3.3. Ambiente computacional

O sistema utilizado para compilar e executar o modelo BRAMS com OpenMP e OpenACC, é um sistema Cray com nós XE6 e nós XK7. Cada nó XE6 tem dois processadores AMD Opteron (Interlagos) de 2,1 GHz com 16 núcleos cada um e 32 GB de memória compartilhada. Cada nó XK7 tem um processador AMD Opteron (Interlagos) de 2,1 GHz com 16 núcleos cada um (16 GB de memória compartilhada) e uma placa GPGPU Nvidia Tesla K20 (família Tesla, arquitetura Kepler) com 2496 núcleos com 706 MHz, 5 GB memória, desempenho nominal 3,52 Tflop/s (precisão simples) e 1,17 TFlop/s (precisão dupla), 225 W consumo, 13 SM's, cada um com 192 cores, *warp size* 32, máximo numero *threads* por SM igual a 2048, máximo numero *threads* por bloco igual a 1024.

Os compiladores disponíveis nesse sistema Cray são o PGI da Portland e o Compilador CCE da Cray. Este trabalho utilizou apenas o compilador CCE da Cray versão 8.3.7. Além disso, utilizou-se um conjunto de ferramentas desenvolvidas pela CRAY que possibilitam a instrumentação do código do modelo BRAMS chamado CrayPat. Nos nós XK7 optou-se por utilizar o profile perftools versão 6.2.0. Além disso, utilizou-se o conjunto de ferramentas *NVIDIA Visual Profiler* para gerar e analisar o desempenho da GPGPU (Performance Analysis Tools, 2015).

4 DESEMPENHO PARALELO COM OpenMP

Apesar de o modelo BRAMS ter alta escalabilidade (PANETTA, 2012) com o uso de biblioteca de comunicação por troca de mensagens MPI, sua estrutura paralela atual pode ser um fator limitante em futuras gerações de computadores. Pelo fato de sua programação estar somente implementada com as bibliotecas de comunicação por troca de mensagens MPI, impõe um gargalo em execuções com mais de algumas dezenas de milhares de núcleos computacionais. Boa parte dessa limitação seria aliviada se possuísse paralelismo híbrido, que combina um processo MPI por nó (paralelismo inter-nós) com múltiplas *threads* OpenMP em cada nó (paralelismo intra-nó).

Implementar paralelismo OpenMP em um programa com centena de milhares de linhas não é tarefa simples. A parte mais trabalhosa é tornar todos os procedimentos livres de condição de corrida (*race conditions*), permitindo o uso de paralelismo de memória compartilhada.

Para acelerar a realização de experimentos e permitir potenciais alterações substanciais nas estruturas de dados básicas do BRAMS, este trabalho gerou um código isolado contendo apenas a advecção de escalares do BRAMS e um programa principal. Entretanto há duas versões de código, uma para a técnica de paralelização por laços e outra, para a técnica por telhas.

A subrotina *advectc* foi extraída do modelo BRAMS e colocada em um módulo (*ModAdvectc*) que também contém todos os procedimentos invocados por ela. O programa principal lê os campos necessários à advecção e invoca *advectc* em um laço, acumulando as tendências resultantes. Este laço do programa principal simula os múltiplos avanços no tempo necessários para calcular um dia de previsão.

Todos os experimentos reportados neste trabalho utilizam esse código isolado. Os experimentos reportam apenas os tempos de execução do laço do programa principal que simula o avanço no tempo. Os tempos de inicialização,

leitura de dados e término do programa principal não são utilizados, por serem irrelevantes aos objetivos dessa dissertação.

Os experimentos codificados em OpenMP invocam a advecção de campos escalares 10.800 vezes, que é o número de avanços no tempo necessário para prever 24 horas em uma grade com 5 km de resolução. Os experimentos utilizam dez domínios distintos, desde o domínio com 10 pontos em *x* e 10 pontos em *y* (denominado 10x10) até o domínio com 100 pontos em *x* e 100 pontos em *y* (denominado 100x100), em passos de 10 pontos nas duas direções. Todos os domínios possuem 42 níveis verticais. Cada experimento foi repetido cinco vezes para minimizar ruídos do sistema, sendo observados coeficientes de variação inferiores a 2% para a técnica de laços e 5% para a técnica por telhas. Os tempos de execução reportados são a média das cinco execuções. Os experimentos foram executados em um nó XE6.

4.1. OpenMP por laços

A forma clássica de introduzir paralelismo OpenMP em um programa é paralelizar os laços de todos os aninhamentos passíveis de paralelização, ou seja, laços cujas iterações podem ser calculadas independentemente umas das outras. Essa foi a forma de paralelismo explorada pela codificação OpenMP por laços.

Foram inseridas diretivas de cooperação de trabalho OpenMP em todos os aninhamentos de laços no corpo da *advectc*. As diretivas foram inseridas nos laços mais externos dos aninhamentos. Imediatamente antes do laço que invoca *advectc* no programa principal, abre-se uma única região paralela, que contém todas as invocações de *advectc.* Todos os *threads* executam todas as iterações desse laço, que simula o avanço no tempo. A Figura 4.1 mostra a região paralela em linha tracejada englobando todos os procedimentos invocados por *advectc*.

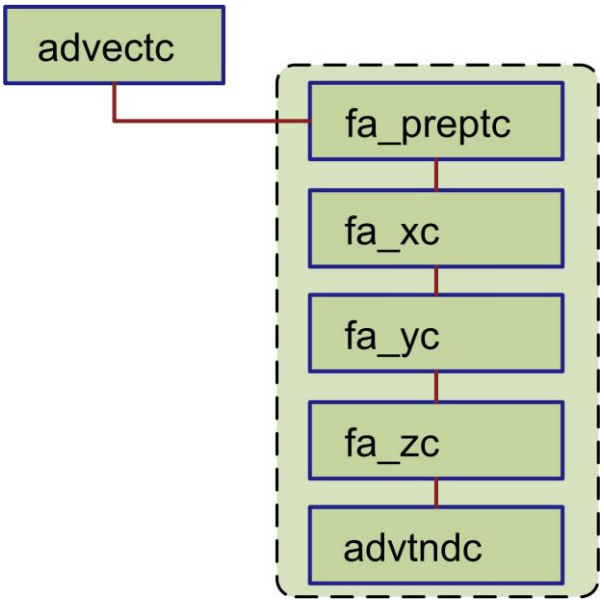


Figura 4.1 – modadvectc_advtndc.

Verificou-se na *advectc* que, em alguns aninhamentos de laços consecutivos que havia independência entre os aninhamentos, isto é, dois ou mais aninhamentos consecutivos eram independentes entre si, possibilitando assim a retiradas das barreiras entre esses aninhamentos que são impostas automaticamente ao término de cada aninhamento paralelizado pela diretiva *$omp do* do OpenMP, por meio da clausula *nowait*.

A Figura 4.2 apresenta os tempos médios de execução do módulo de Advecção (em segundos) para todas as grades, de 10x10 até 100x100 e a Figura 4.3 apresenta os respectivos ganhos (*speed up*), medido com relação ao tempo de execução de 1 *thread* da respectiva versão no domínio acima descrito.

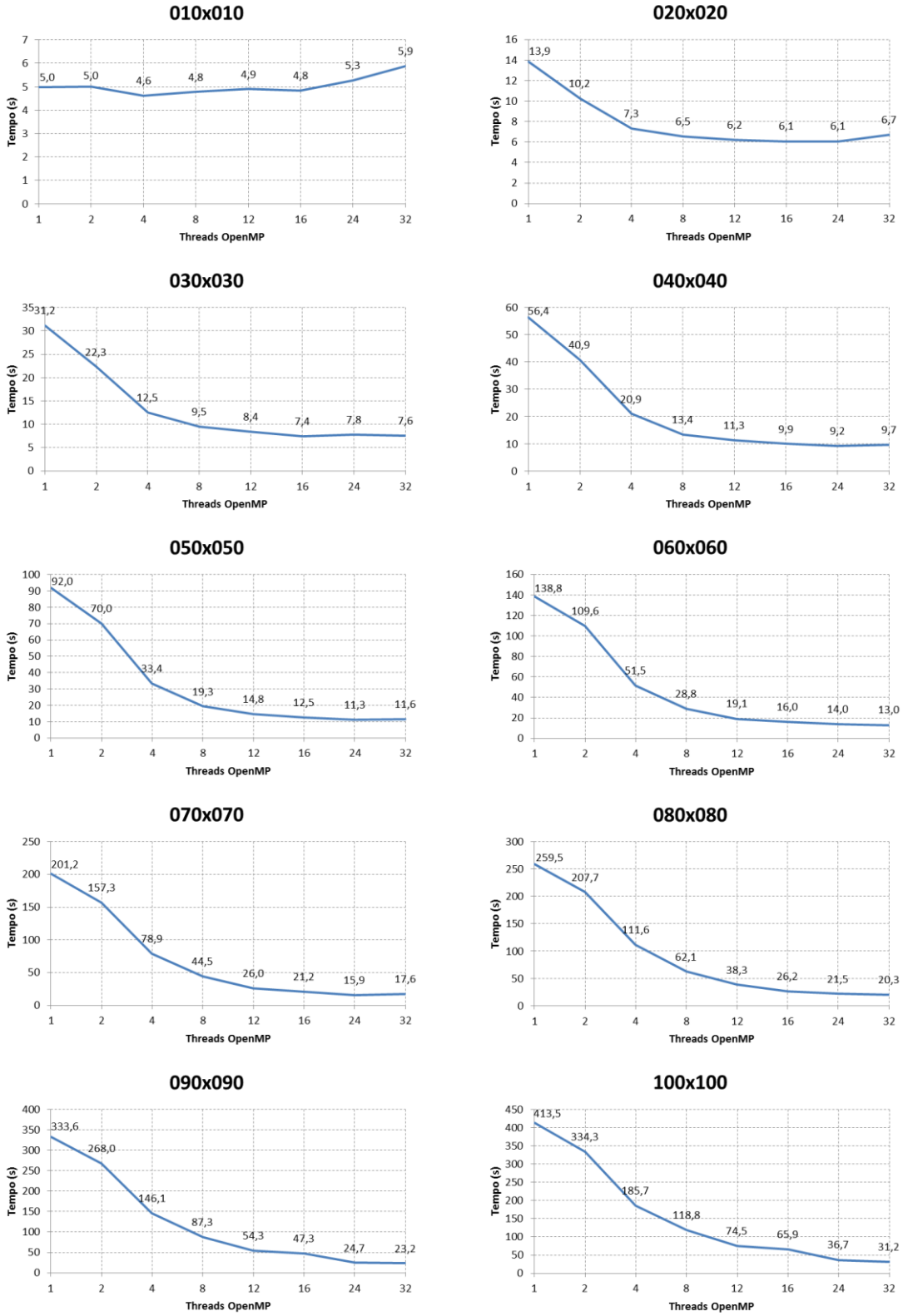


Figura 4.2 – Tempo de execução do modulo de Advecção em função do número de *threads* da versão OpenMP paralelizada por laços para diferentes tamanhos de grade.

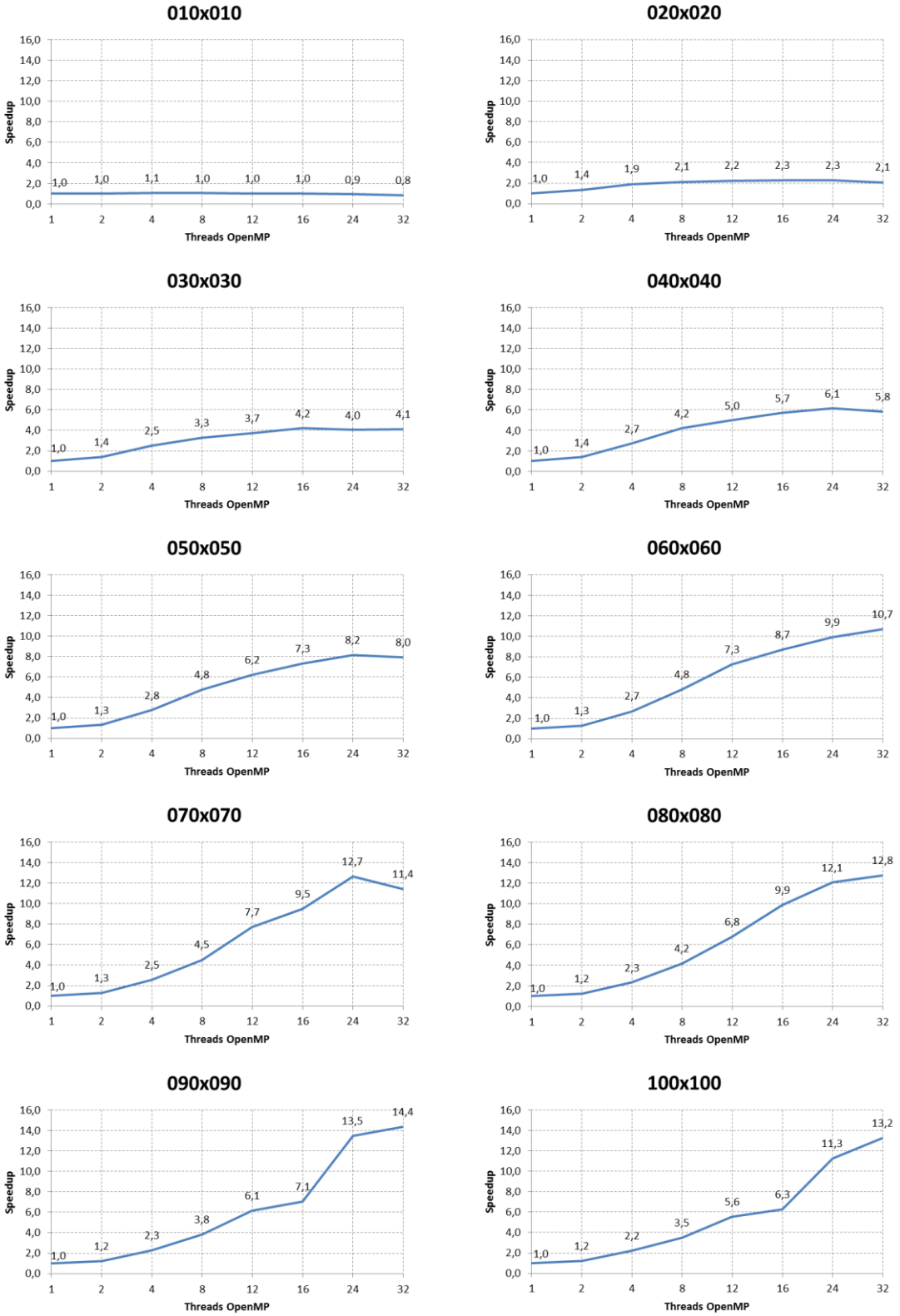


Figura 4.3 – *Speed up* do modulo de Advecção em função do numero de *threads* da versão OpenMP paralelizada por laços para diferentes tamanhos de grade.

Na Figura 4.2, nota-se que o tempo de execução com uma única *thread* tende a aumentar com o tamanho da grade, mas sempre menos que quadraticamente. Assim, aumenta de 5 segundos (grade 10x10) para 92 segundos (grade 50x50), ou seja, mais que 18 vezes (o aumento quadrático corresponderia a 25 vezes), ou então para 413,5 segundos (grade 100x100), ou seja, mais que 82 vezes (o aumento quadrático corresponderia a 100 vezes). Exceto pelas grade 10x10, o tempo de processamento diminui com o aumento do número de *threads* OpenMP, embora essa diminuição seja menor ao se passar de 24 para 32 *threads* (em alguns casos, até aumentando ligeiramente). No caso da grade 10x10, o tempo de execução praticamente não caiu com o aumento do número de *threads*, provavelmente devido à baixa granularidade. Na Figura 4.3, nota-se que os *speed up's* tendem a aumentar com o tamanho da grade, chegando a ultrapassar ligeiramente 50% para alguns casos (mas jamais para 32 *threads*, mesmo nas grades maiores).

4.2. OpenMP por telhas

O paralelismo OpenMP por telhas (*tiles*) baseia-se na técnica de blocagem, que explora a divisão do domínio horizontal em blocos para otimizar o acesso à memória. O uso de OpenMP torna-se possível pelo fato de que a advecção de qualquer trecho do domínio horizontal é independente da advecção de qualquer outro trecho desse domínio. Consequentemente, é possível dividir o domínio horizontal em blocos retangulares e executar, simultaneamente, a advecção nesses trechos. Para computar a advecção em todo o domínio horizontal basta que os blocos retangulares formem uma partição do domínio. Quando formam uma partição do domínio os blocos são denominados telhas (*tiles*).

Os procedimentos invocados por *advectc* foram modificados para atuarem sobre uma telha e não sobre todo o domínio horizontal. Para tanto, os extremos de todos os aninhamentos tridimensionais foram cautelosamente modificados. O procedimento *advectc* passou a ser composto por um novo laço que percorre

todas as telhas do domínio horizontal, invocando o código original de *advectc* modificado para atuar sobre uma telha a cada iteração do novo laço.

A única diretiva de cooperação de trabalho deste paralelismo é a que divide as iterações do novo laço pelos *threads*. Como no paralelismo por laços, o programa principal abre a única seção paralela imediatamente antes do laço que simula o avanço no tempo invocando *advectc* repetidamente e fecha a seção paralela imediatamente após esse laço. Todos os *threads* executam todas as iterações desse laço. Durante a execução de uma iteração desse laço, ao adentrar *advectc* e atingir o laço que percorre as telhas, *threads* distintos recebem telhas distintas e realizam a advecção na telha correspondente. O paralelismo ocorre na advecção simultânea das telhas, pela divisão de tarefas do laço que percorre as telhas.

O tamanho de uma telha é definido pelo número de células da grade em *x* e em *y* na telha. O tamanho da telha é dado de entrada do programa. O número de células no respectivo eixo é particionado em trechos consecutivos do tamanho desejado. O último trecho tem tamanho menor ou igual ao desejado, para acomodar divisões inexatas de inteiros. As telhas são o produto cartesiano dessa divisão nos dois eixos. Este trabalho denota uma telha pelo seu próprio tamanho n_x x n_y.

A seguir, apresentam-se os tempos obtidos na versão OpenMP por telhas de tamanho 2x2 e 4x4 para as grades de 10x10 até 100x100 (Figura 4.4), com seus respectivos *speed ups* (Figura 4.5). Nas execuções com 1 *thread*, nota-se que os tempos de execução para telhas 4x4 aumentam ligeiramente em relação à versão OpenMP por laços, mas aumentam significativamente (embora menos que 50%) para telhas 2x2. Isso se explica pelo seguinte: o tamanho de telha 2x2 corresponde na prática a 4x4 células, pois é necessário incluir o cálculo das bordas do domínio abrangido pela telha, enquanto que o tamanho 4x4, corresponde na prática a 6x6 células. Telhas pequenas como a 2x2 implicam em maior número de cálculos repetidos, pois considerando-se

cada telha, suas bordas são repetidas nas telhas vizinhas, enquanto que telhas maiores não replicam tantos cálculos, porém resultam num número total de telhas menor a ser executado pelos *threads*, o que diminui o grau de paralelismo.

Por exemplo, os fluxos da coluna central de uma telha 3x3 são calculados apenas nessa telha, pois não pertencem à borda de qualquer outra telha 3x3. O mesmo ocorre com as quatro colunas centrais de uma telha 4x4. Em geral, o número de colunas que não são repetidas aumenta quadraticamente com o aumento da telha, uma vez que é proporcional à área da telha. Como o número de colunas é fixo em um domínio de tamanho fixo, como cada coluna pode ter seus cálculos repetidos ou não e como o número de colunas com cálculos não repetidos aumenta com o aumento do tamanho da telha, conclui-se que o número de colunas repetidas diminui com o aumento da telha em um domínio de tamanho fixo.

Entretanto, aumentar o tamanho da telha diminui o grau de paralelismo da versão por telhas, por diminuir o número de telhas em um domínio fixo. Dessa forma, aumentar o tamanho da telha tem efeitos contraditórios no tempo de execução da versão por telhas. Apesar de diminuir a repetição de cálculos, limita o grau máximo de paralelismo, implicando em algo que poderia ser chamado de “granularidade excessiva”.

Similarmente aos resultados de desempenho da versão OpenMP por laços, observa-se na Figura 4.4 que o tempo de execução com uma única *thread* tende a aumentar com o tamanho da grade, mas sempre menos que quadraticamente. O tempo de processamento diminui com o aumento do número de *threads* OpenMP, embora essa diminuição seja menor ao se passar de 24 para 32 *threads* (em alguns casos, até aumentando ligeiramente). No caso da grade 10x10, o tempo de execução praticamente somente diminuiu até 4 *threads* (telhas 4x4) ou 16 *threads* (telhas 2x2). O tamanho de telha 2x2 resultou em tempos de execução menores para grades menores ou iguais que

70x70, tendência que se inverteu para grades maiores ou iguais que 80x80, em que as telhas 4x4 tiveram tempos de execução menores. Na Figura 4.5, nota-se que os *speed up's* tendem a aumentar com o tamanho da grade, chegando a ultrapassar ligeiramente 50% para alguns casos. Para grades maiores que 60x60, os *speed up's* são melhores para as telhas 4x4.

Os resultados completos para telhas 2x2 até 5x5 aparecem em (SILVA JUNIOR *et al.*, Submetido em 2015).

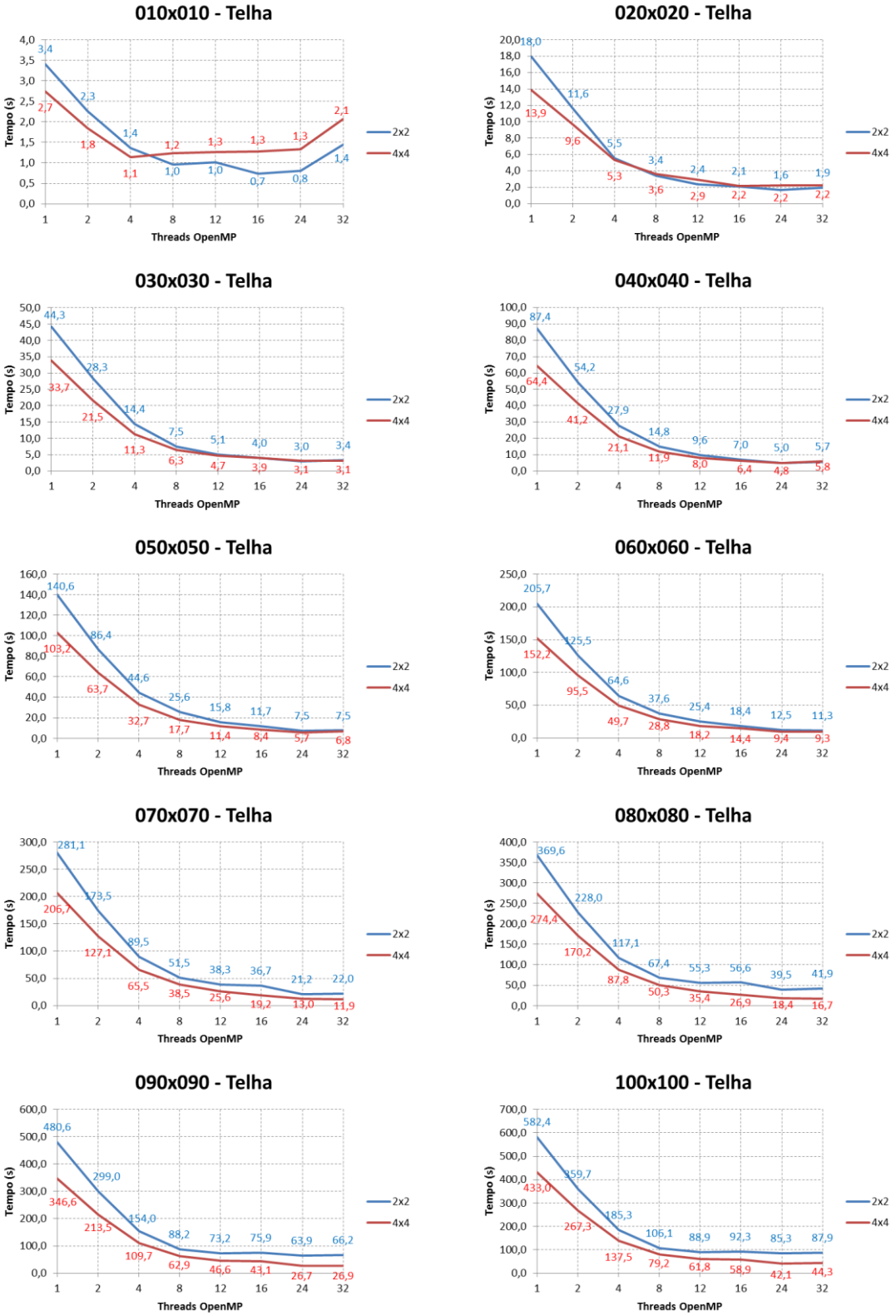


Figura 4.4 – Tempo de execução do modulo de Advecção em função do numero de *threads* da versão OpenMP paralelizada por telha para diferentes tamanhos de grade.

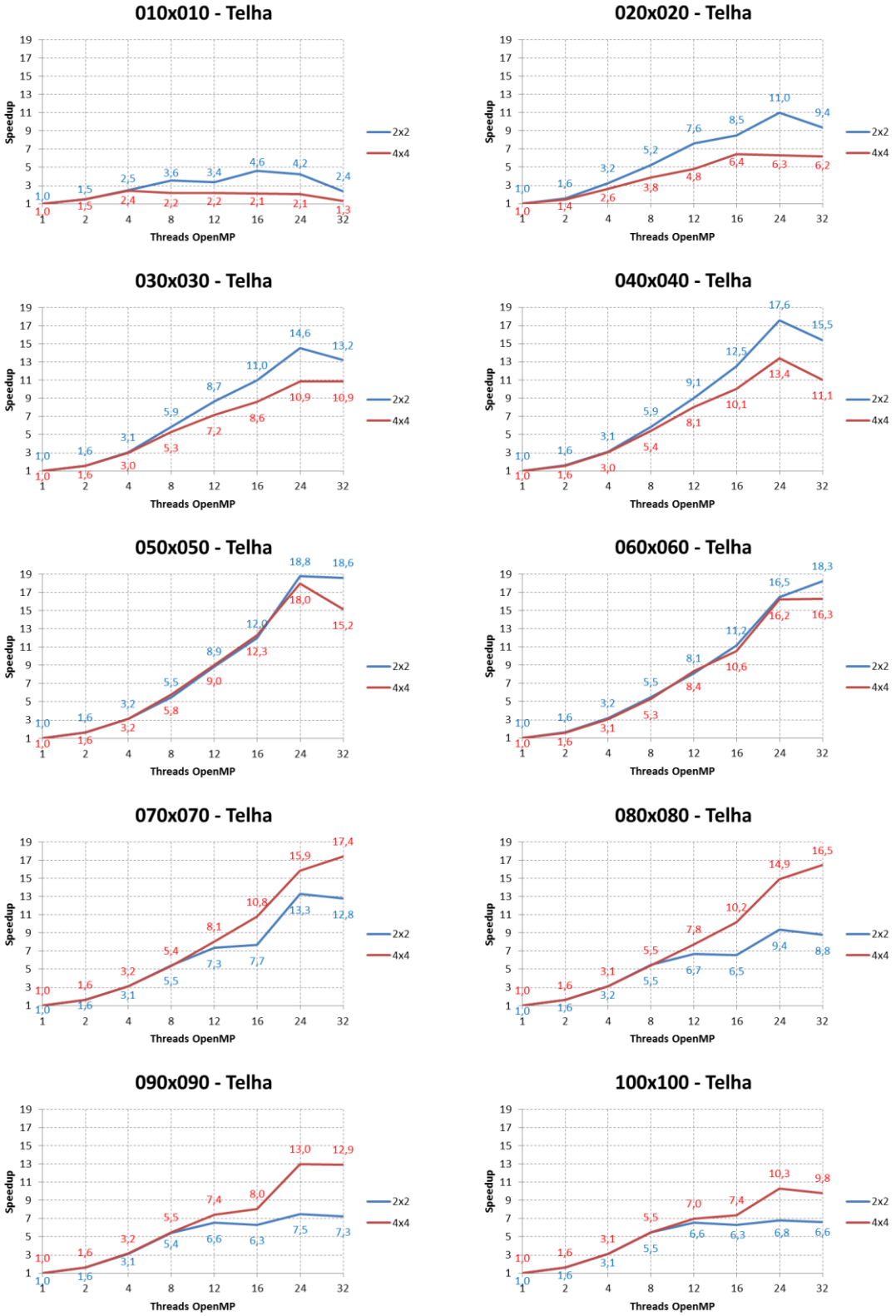


Figura 4.5 – *Speed up* do modulo de Advecção em função do número de *threads* da versão OpenMP paralelizada por telha para diferentes tamanhos de grade.

4.3. Conclusões sobre o desempenho com OpenMP

Nesse capitulo demonstrou-se duas formas de codificação em OpenMP para o módulo de advecção da dinâmica do modelo BRAMS, uma delas a forma clássica de paralelismo (por laços) e a outra mais elaborada (por telhas). A codificação por laços tem um desempenho razoável, obtendo eficiência pouco acima de 50% nos melhores casos. Nas execuções com 24 ou 32 *threads*, *speed ups* acima de 12 somente foram obtidos para as grades de maior tamanho.

De maneira geral, como pode ser observado na Figura 4.6, a codificação por telhas 4x4 apresentou um desempenho melhor que a codificação por laços, exceto por alguns poucos casos, como para as grades 90x90 e 100x100, ambas com 24 e 32 *threads*. As telhas 2x2 obtiveram melhor desempenho apenas para tamanhos de grade pequenos. Note-se que o tempo "sequencial" (execução com 1 *thread*) é maior para a versão por telhas do que para a versão por laços, embora essa diferença seja maior para as telhas 2x2 e menor para as telhas 4x4. Na maioria dos casos, a versão codificada com telhas 4x4 obteve eficiências variando entre 40% e 50%. Houve casos piores, como por exemplo, grade 100x100 e 32 *threads*, em que a eficiência foi de 30%, ou então, melhores, como para a grade 70x70 com 4 *threads*, em que foi de 80%.

Assim, pode-se concluir que, de maneira geral, a versão codificada por telhas 4x4 tem melhor desempenho que a versão codificada por laços.

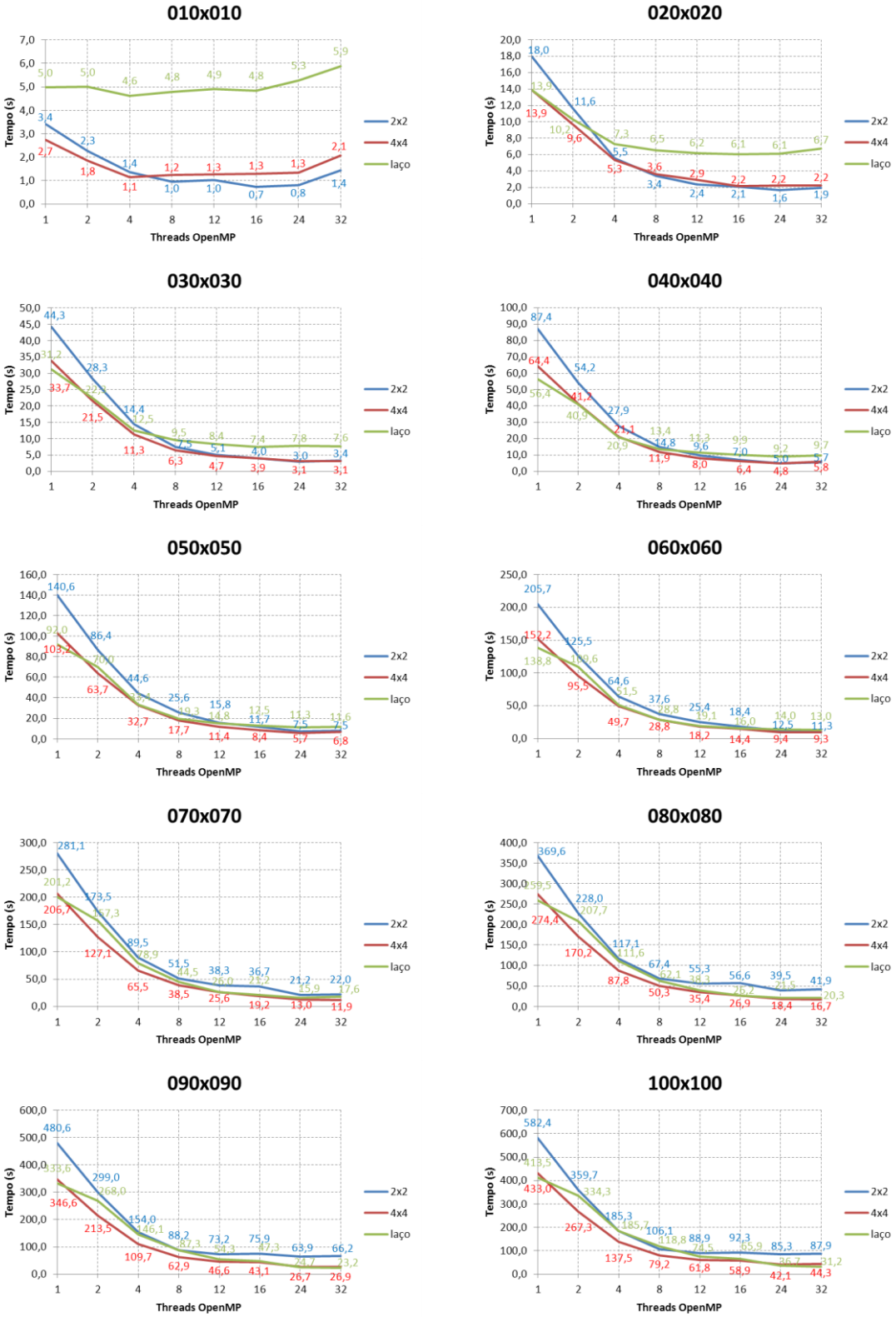


Figura 4.6 – Tempo de execução do modulo de Advecção em função do numero de *threads* da versão OpenMP paralelizada por telha e por laço para diferentes tamanhos de grade.

5 DESEMPENHO PARALELO COM OPENACC

Adotou-se para versão codificada em OpenACC a paralelização por laços exposta no capítulo anterior na versão codificada em OpenMP. Essa opção foi devido à menor granularidade resultante do uso das telhas, que as tornariam inadequadas em termos de desempenho para a execução em GPGPU.

Os experimentos codificados em OpenACC utilizaram a mesma configuração dos experimentos em OpenMP: a advecção isolada foi executada 10.800 vezes em uma grade com 5 km de resolução. Cada experimento foi repetido cinco vezes para obter as médias reportadas, sendo observados coeficientes de variação inferiores a 1%. Os experimentos foram realizados em nó XK7. Todos os testes de desempenho expostos a seguir referem-se a esta configuração de experimento.

Para melhor entendimento do progresso da codificação em OpenACC na advecção isolada, as sucessivas otimizações do código paralelo foram organizadas nas 6 etapas reportadas a seguir. Os tempos de execução de cada uma das etapas são reportados apenas na grade 40x40. O tempo de execução em todas as grades é reportado ao final deste capítulo. O ganho de desempenho nessas etapas é sumarizado na Figura 5.1 que apresenta os tempos de execução nessa grade para as diversas etapas de otimização, comparando-os ao tempo da execução sequencial, assumido como sendo a execução da versão OpenMP com um único *thread*.

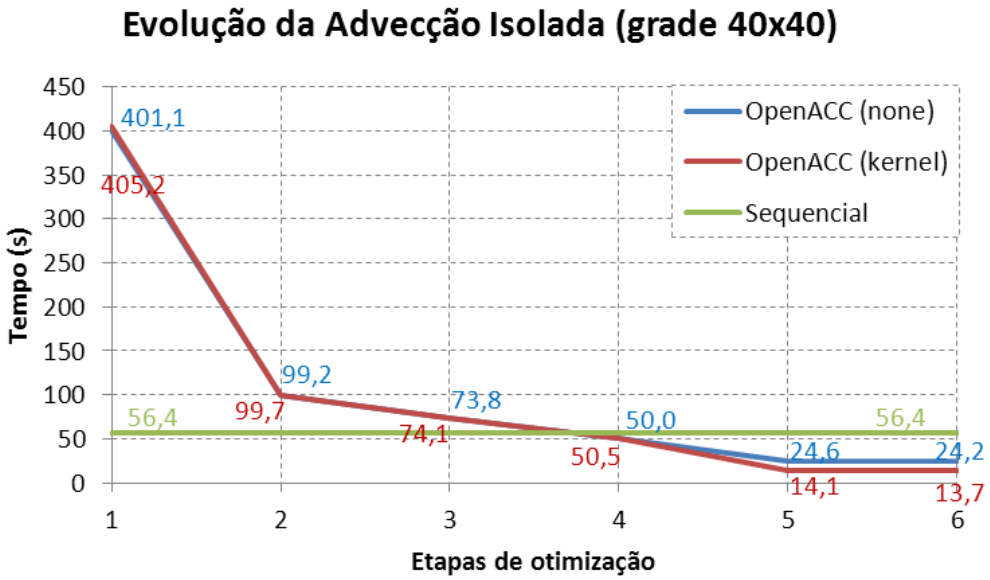


Figura 5.1 – Tempos de execução das versões OpenACC com opções de compilação *async_kernel* e *async_none* para cada etapa de otimização, comparados ao tempo de execução sequencial.

A versão do compilador OpenACC utilizada (Cray CCE) apresenta duas opções de compilação (*auto_async_none* e *auto_async_kernel*) as quais determinam, respectivamente, se a CPU aguarda ou não aguarda o término da execução das computações (*kernels*) enviadas pela CPU para a GPGPU para prosseguir na sua computação. Ou seja, essas opções definem se o fluxo de execução entre a CPU e a GPGPU é síncrono ou assíncrono, respectivamente. Essas opções comandam o sincronismo da execução dos *kernels* disparados pela CPU mas não comandam o sincronismo das transferências de dados entre as memórias da CPU e da GPGPU. As transferências de dados são síncronas nas duas opções. Uma terceira opção, *auto_async_all*, permite comportamento assíncrono entre CPU e GPGPU nas duas operações (computações e transferência de dados).

O funcionamento das opções *auto_async_none* e *auto_async_kernel* foi comprovado por um experimento realizado com a versão otimizada da 5ª e 6ª

etapa. O experimento mediu o tempo decorrido na CPU antes e depois do disparo de uma computação (*kernel*) executada na GPGPU. Esse tempo decorrido é medido pela diferença entre os tempos do *system wall clock* imediatamente antes e depois do disparo, capturados por uma rotina do sistema operacional executada na CPU. A versão compilada com a opção *auto_async_none* demandou 314 microssegundos enquanto a versão compilada com a opção *auto_async_kernel* demandou 76 microssegundos. O experimento demonstrou que utilizando a opção *auto_async_kernel* a CPU executou a segunda medida de tempo sem esperar que a computação terminasse de ser executada na GPGPU, visto que o tempo de execução da computação na GPGPU é o mesmo nas duas execuções.

O tempo de execução das quatro primeiras etapas de otimização não foi alterado pela escolha entre as opções *auto_async_none* e *auto_async_kernel.* A escolha entre essas opções só foi relevante nas duas últimas etapas da otimização. Isso ocorreu devido à frequência de transferência de dados da GPGPU para a CPU, que é alta nas quatro primeiras etapas e significativamente mais baixa nas duas últimas etapas. A alta frequência de transferência de dados nas quatro primeiras etapas reduz as oportunidades de computações assíncronas, pois o fluxo de execução da CPU só envia novas computações para a GPGPU após receber os dados solicitados.

Compiladores OpenACC utilizam as bibliotecas *runtime* da linguagem CUDA. As versões mais recentes de CUDA permitem a definição de múltiplos fluxos de execução simultâneos (*streams*), cada um comandado por uma fila de execução. Solicitações em cada fila de execução são atendidas sequencialmente, num esquema *FIFO* (Figura 5.2). Solicitações em filas distintas podem ser atendidas simultaneamente, se houver suporte de hardware. Aparentemente a versão do compilador OpenACC utilizado não permite a definição de múltiplos fluxos. Portanto, em todos os experimentos realizados não há execução concorrente de *kernels* na GPGPU.

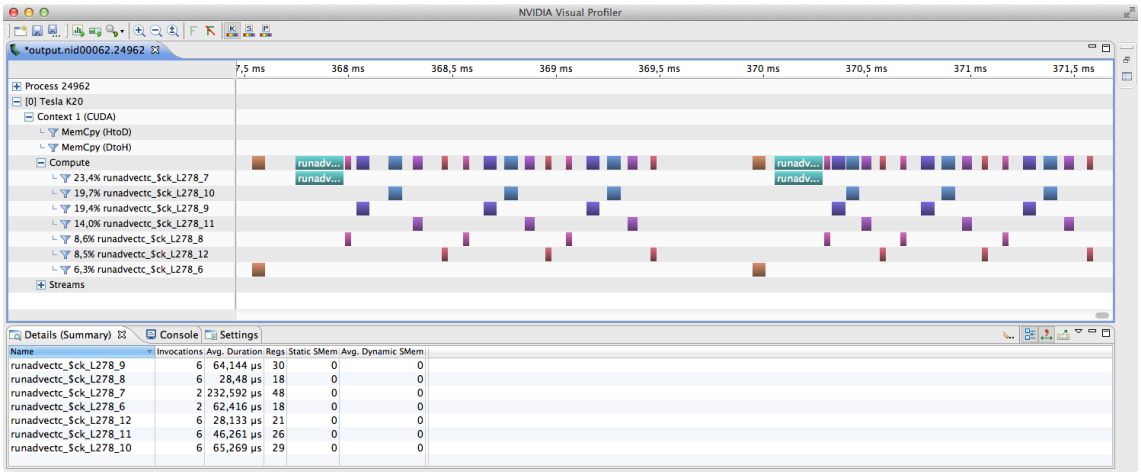


Figura 5.2 – *NVIDIA Visual Profiler* (2 *timestep*).

5.1. Primeira etapa de otimização

OpenACC permite a introdução incremental de regiões aceleradas, assim como OpenMP permite a introdução incremental de regiões paralelas. A primeira etapa de otimização utilizou esse mecanismo incremental, operando sobre um novo aninhamento de laços a cada nova versão. A versão final desta etapa envia para execução na GPUGPU cada um dos aninhamentos contidos no interior do procedimento *advectc* e nos procedimentos por ele invocados.

Como as iterações de cada aninhamento enviado para execução na GPGPU são independentes, as iterações de cada aninhamento são executadas em paralelo pela introdução da diretiva *acc loop* no laço mais externo de cada aninhamento. Uma única região acelerada foi criada para cada procedimento, abrangendo todo o texto desse procedimento. Desta forma, a CPU executa o texto da *advectc* invocando cada um dos seus procedimentos, que por sua vez disparam sucessivos *kernels* que utilizam a GPGPU.

Nesta primeira etapa de otimização todos os dados residem na memória da CPU. Em cada região acelerada (uma por procedimento) existe uma região de dados que abrange todo o texto da região acelerada (que é o texto do

procedimento). Cada região de dados declara todas as variáveis como *data copy*, ou seja, o compilador decide quais variáveis enviar da CPU para a GPGPU na entrada da região de dados e quais variáveis enviar da GPGPU para a CPU ao término da região de dados. Como o compilador desconhece a sequência de regiões de dados, nenhuma variável persiste na memória da GPGPU entre as regiões de dados. O tráfego de dados entre a CPU e a GPGPU requer sincronização na entrada e na saída de cada região de dados, aumentando substancialmente o tempo de execução.

Diversas matrizes não são completamente referenciadas nos procedimentos internos a *advectc*. Nesse caso, constatou-se que a melhor estratégia é copiar as matrizes por inteiro da memória da CPU para a memória da GPGPU, em vez de copiar apenas as partes das matrizes referenciadas no procedimento.

Os tempos de execução desta primeira etapa de otimização foram 401,10 segundos com 85,20% de utilização da capacidade de processamento da GPU e 405,22 segundos com 85,22% de utilização da capacidade de processamento da GPU, utilizando as opções de compilação *auto_async_none* e *auto_async_kernel* respectivamente. Tais tempos são inaceitáveis, face ao tempo da execução sequencial (um *thread*) da advecção isolada codificada por laços, que foi 56,44 segundos, conforme ilustrado na Figura 5.3.

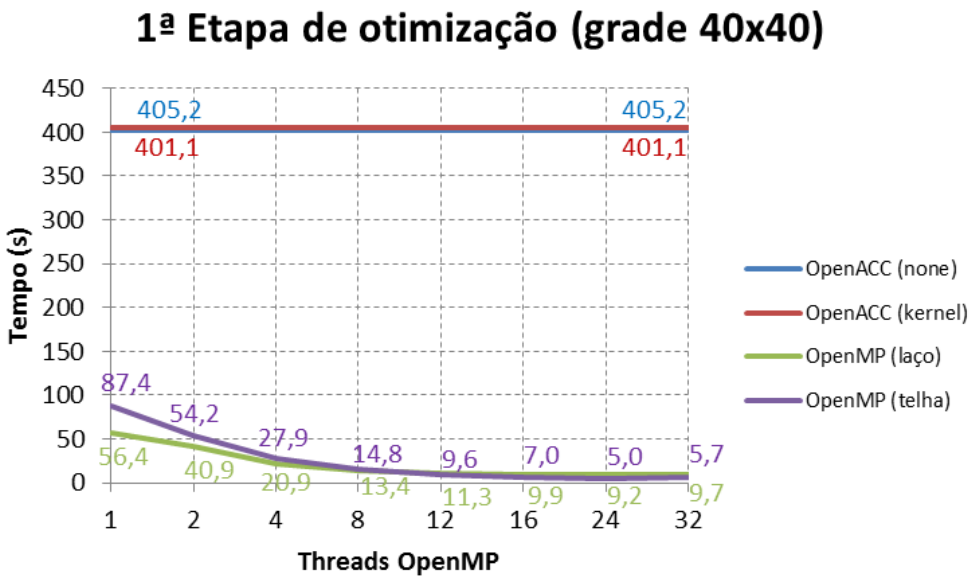


Figura 5.3 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 1ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

Este baixo desempenho paralelo decorre do tempo necessário para transferir os dados da memória da CPU para a memória da GPGPU e vice-versa cada vez que uma região paralela a ser executada pela GPGPU é aberta e fechada, conforme indicado pelo tempo de execução e número de invocações de ACC_COPY, expresso na saída do *profiler CrayPat* apresentado na Figura 5.4. A soma de todos os tempos de execução dedicados à transferência de dados nessa figura totaliza 363 segundos.

```
Table 1:  Profile by Function Group and Function

  Time% |        Time | Imb.  |  Imb.  |     Calls |Group
        |             | Time  | Time%  |           | Function

 100.0% | 399.092319 |   --  |    --  | 810002.0 |Total
|-------------------------------------------------------------------------------------------
| 100.0% | 399.092226 |   --  |    --  | 810001.0 |USER
||------------------------------------------------------------------------------------------
||  18.7% |  74.501788 |   --  |    --  |  64800.0 |advectc$modadvectc_.ACC_COPY@li.162
||  17.1% |  68.424805 |   --  |    --  |  64800.0 |advectc$modadvectc_.ACC_COPY@li.171
||  16.2% |  64.506835 |   --  |    --  |  64800.0 |advectc$modadvectc_.ACC_COPY@li.167
||   8.2% |  32.685412 |   --  |    --  |  64800.0 |advectc$modadvectc_.ACC_COPY@li.175
||   7.9% |  31.388856 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.548
||   6.5% |  26.134517 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.629
||   6.0% |  23.978266 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_COPY@li.61
||   4.7% |  18.571054 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_COPY@li.94
||   4.4% |  17.565719 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_COPY@li.121
||   2.3% |   9.195502 |   --  |    --  |      1.0 |runadvectc_
||   1.4% |   5.779287 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.162
||   1.3% |   5.121611 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.171
||   1.2% |   4.943456 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_COPY@li.136
||   1.1% |   4.256806 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.167
||   0.9% |   3.704232 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_KERNEL@li.551
||   0.6% |   2.447145 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.175
||   0.2% |   0.764784 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_KERNEL@li.61
||   0.2% |   0.732542 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_KERNEL@li.121
||   0.1% |   0.596984 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_DATA_REGION@li.171
||   0.1% |   0.587993 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_DATA_REGION@li.162
||   0.1% |   0.565670 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_DATA_REGION@li.167
||   0.1% |   0.539381 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_DATA_REGION@li.175
||   0.1% |   0.353803 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.551
||   0.1% |   0.230983 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_DATA_REGION@li.548
||   0.1% |   0.221414 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_REGION@li.61
||   0.1% |   0.215849 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_REGION@li.171
||   0.1% |   0.213854 |   --  |    --  |  10800.0 |advectc$modadvectc_.ACC_REGION@li.121
||   0.1% |   0.213077 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_REGION@li.167
||   0.1% |   0.212891 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_REGION@li.162
||   0.1% |   0.206300 |   --  |    --  |  32400.0 |advectc$modadvectc_.ACC_REGION@li.175
||   0.0% |   0.186706 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_REGION@li.551
||   0.0% |   0.044704 |   --  |    --  |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.628
||==========================================================================================
|   0.0% |   0.000093 |   --  |    --  |      1.0 |ETC
||-----------------------------------------------------------------------------------------
|   0.0% |   0.000093 |   --  |    --  |      1.0 | _END
|=========================================================================================
```

Figura 5.4 – *CrayPat Report* (1ª etapa).

5.2. Segunda etapa de otimização

O objetivo desta segunda etapa é reduzir o tráfego de dados entre a CPU e a GPGPU. Busca-se eliminar transferências repetidas durante a execução de uma invocação de *advectc*. Para tanto, as diversas regiões de dados contidas no corpo dos procedimentos invocados por *advectc* foram colimadas em uma única região de dados, declarada no início do corpo de *advectc* e que tem como escopo todo o corpo de *advectc*. As regiões de dados dos procedimentos invocados por *advectc* foram eliminadas.

A nova região de dados qualifica as variáveis utilizadas na GPGPU em variáveis de entrada, entrada e saída e variáveis locais à GPGPU. A execução dessa diretiva transfere dados da CPU para a GPGPU no início de cada execução de *advectc* e transfere dados no sentido contrário ao término de cada execução de *advectc*.

A eliminação de transferências de dados repetitivas nesta segunda etapa acarretou uma redução de quatro vezes no tempo médio total (Figura 5.5) com relação ao tempo de execução da primeira etapa. O tempo de execução desta etapa foi de 99,23 segundos para a versão *auto_async_none* com 81,44% de utilização da capacidade de processamento da GPU e 99,67 segundos para a versão *auto_async_kernel* com 80,52% de utilização da capacidade de processamento da GPU. Entretanto esse tempo ainda é substancialmente maior que o tempo sequencial.

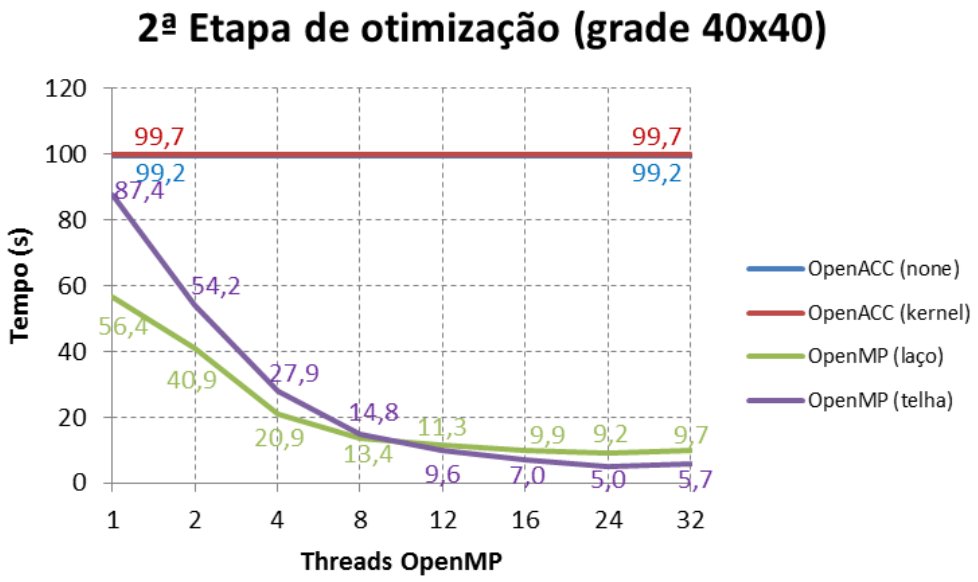


Figura 5.5 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 2ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

A análise do tempo de execução fornecida pelo *profiler CrayPat* demonstra que ainda existem muitas transferências de dados e que tais transferências demandam tempo de execução substancial. Esse comportamento é demonstrado na Figura 5.6. A soma de todos os tempos das transferências de dados reportados é de 67,01 segundos. Consequentemente, o tempo total de execução ainda é dominado por essa operação.

```
Table 1:  Profile by Function Group and Function

  Time% |      Time | Imb. |  Imb. |    Calls |Group
        |           | Time | Time% |          | Function

 100.0% | 95.861473 |   -- |    -- | 788402.0 |Total
|-------------------------------------------------------------------------------------------
| 100.0% | 95.861373 |   -- |    -- | 788401.0 |USER
||------------------------------------------------------------------------------------------
||  24.5% | 23.520563 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_COPY@li.81
||  17.6% | 16.853528 |   -- |    -- |  64800.0 |advectc$modadvectc_.ACC_COPY@li.177
||   6.4% |  6.149339 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.169
||   6.2% |  5.984878 |   -- |    -- |  64800.0 |advectc$modadvectc_.ACC_COPY@li.160
||   6.2% |  5.963012 |   -- |    -- |  64800.0 |advectc$modadvectc_.ACC_COPY@li.164
||   6.2% |  5.946483 |   -- |    -- |  64800.0 |advectc$modadvectc_.ACC_COPY@li.169
||   6.2% |  5.946034 |   -- |    -- |  64800.0 |advectc$modadvectc_.ACC_COPY@li.173
||   6.0% |  5.748001 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.164
||   5.1% |  4.913548 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.173
||   3.8% |  3.630726 |   -- |    -- |  10800.0 |fa_preptc$modadvectc_.ACC_KERNEL@li.559
||   2.6% |  2.471960 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.177
||   1.3% |  1.272859 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_KERNEL@li.160
||   1.2% |  1.123754 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_DATA_REGION@li.81
||   0.9% |  0.844704 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_COPY@li.180
||   0.7% |  0.668436 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_KERNEL@li.124
||   0.7% |  0.648121 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_KERNEL@li.94
||   0.5% |  0.511750 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_REGION@li.177
||   0.5% |  0.501561 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_REGION@li.160
||   0.5% |  0.494486 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_REGION@li.173
||   0.5% |  0.494295 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_REGION@li.164
||   0.5% |  0.492309 |   -- |    -- |  32400.0 |advectc$modadvectc_.ACC_REGION@li.169
||   0.5% |  0.450279 |   -- |    -- |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.636
||   0.4% |  0.397850 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_KERNEL@li.114
||   0.4% |  0.383230 |   -- |    -- |  10800.0 |fa_preptc$modadvectc_.ACC_COPY@li.559
||   0.2% |  0.182103 |   -- |    -- |  10800.0 |fa_preptc$modadvectc_.ACC_REGION@li.559
||   0.1% |  0.087801 |   -- |    -- |      1.0 |runadvectc_
||   0.1% |  0.065570 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_REGION@li.94
||   0.1% |  0.057719 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_REGION@li.114
||   0.1% |  0.056475 |   -- |    -- |  10800.0 |advectc$modadvectc_.ACC_REGION@li.124
||==========================================================================================
|   0.0% |  0.000100 |   -- |    -- |      1.0 |ETC
||-----------------------------------------------------------------------------------------
|   0.0% |  0.000100 |   -- |    -- |      1.0 | _END
|=========================================================================================
```

Figura 5.6 – *CrayPat Report* (2ª etapa).

5.3. Terceira etapa de otimização

O volume de transferências de dados relatado pelo *profiler CrayPat* na execução da segunda etapa demonstra que ainda há quantidade substancial de transferências nas regiões aceleradas internas à *advectc*. Esta terceira

etapa continua visando reduzir o volume e o tempo de execução dessas operações utilizando, simultaneamente, duas estratégias.

A primeira estratégia visa informar ao compilador que as variáveis referenciadas nas regiões aceleradas em uma execução de *advectc* já se encontram na memória da GPGPU. Essa estratégia foi implementada por duas alterações. A primeira alteração insere a diretiva "*inline*" no texto de *advectc*. Essa diretiva substitui as invocações existentes em *advectc* pelo texto do procedimento invocado, visando informar ao compilador que as regiões aceleradas de tais procedimentos estão no interior da região de dados que engloba *advectc*. A segunda alteração insere, em cada região acelerada de cada procedimento interno a *advectc*, a informação que todas as variáveis utilizadas na região acelerada residem na memória da GPGPU (pela clausula "*present*" da diretiva "*acc parallel*").

A segunda estratégia visa manter os dados de entrada e de saída de *advectc* na memória da GPGPU entre as invocações a *advectc*, ou seja, entre os *timesteps*. A justificativa para esta estratégia é que as variáveis de entrada de *advectc* já estariam na GPGPU caso a dinâmica fosse integralmente executada na GPGPU, carregadas por procedimentos anteriores a *advectc*, e que as variáveis de saída de *advectc* permaneceriam na GPGPU para uso dos procedimentos posteriores a *advectc*. Novamente, esta estratégia é implementada por duas alterações. A primeira alteração insere a diretiva "*inline*" no texto do programa principal englobando apenas as invocações a *advectc*. A segunda alteração move a região de dados que engloba *advectc* para o programa principal, englobando o laço que avança no tempo, e por consequência, todas as invocações a *advectc*. As variáveis necessárias à execução das regiões aceleradas de *advectc* são declaradas "*present_or_copyin*" na região acelerada.

O tempo médio de execução desta terceira etapa da advecção isolada diminuiu para 73,80 segundos com 83,50% de utilização da capacidade de

processamento da GPU sob a opção *auto_async_none* e 74,07 segundos com 83,68% de utilização da capacidade de processamento da GPU sob a opção *auto_async_kernel*. A redução do tempo de execução com relação à segunda etapa foi de 25,63% (Figura 5.7).

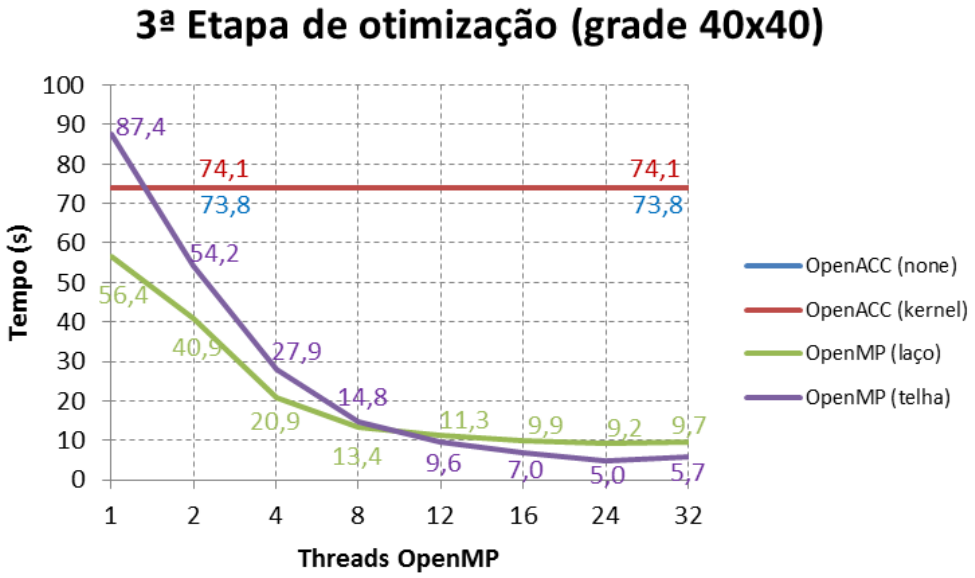


Figura 5.7 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 3ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

A Figura 5.8 mostra a saída produzida pelo *profiler CrayPat* para o código desta etapa de otimização. A análise dessa informação demonstra sucesso parcial nas duas estratégias. Por um lado, surgem transferências de dados creditadas ao programa principal (*runadvectc*), substituindo transferências de dados anteriormente creditadas a procedimentos no módulo *modadvectc*, demonstrando que os processos de "*inline*" ocorreram corretamente. Por outro lado, a soma de todos os tempos de trocas de dados reportados é de 67,01 segundos. Consequentemente, o tempo total de execução ainda é dominado pela troca de dados. Ainda mais, a tarefa com maior tempo de execução é uma transferência de dados.

```
Table 1:  Profile by Function Group and Function

  Time% |       Time | Imb. |  Imb. |     Calls |Group
        |            | Time | Time% |           | Function

 100.0% | 71.235465 |   -- |    -- | 766805.0 |Total
|-------------------------------------------------------------------------------
| 100.0% | 71.235360 |   -- |    -- | 766804.0 |USER
||------------------------------------------------------------------------------
||  50.9% | 36.248021 |   -- |    -- | 324000.0 |runadvectc_.ACC_COPY@li.255
||  33.2% | 23.641754 |   -- |    -- | 151200.0 |runadvectc_.ACC_KERNEL@li.255
||   8.3% |  5.937251 |   -- |    -- |  32400.0 |atob_long_.ACC_COPY@li.918
||   3.4% |  2.411091 |   -- |    -- | 151200.0 |runadvectc_.ACC_REGION@li.255
||   1.8% |  1.315767 |   -- |    -- |  32400.0 |atob_long_.ACC_KERNEL@li.918
||   1.4% |  0.967396 |   -- |    -- |  10800.0 |runadvectc_.ACC_DATA_REGION@li.255
||   0.7% |  0.519615 |   -- |    -- |  32400.0 |atob_long_.ACC_REGION@li.918
||   0.2% |  0.114007 |   -- |    -- |  32400.0 |atob_long_.ACC_COPY@li.926
||   0.1% |  0.054246 |   -- |    -- |      1.0 |runadvectc_.ACC_DATA_REGION@li.248
||   0.0% |  0.020435 |   -- |    -- |      1.0 |runadvectc_
||   0.0% |  0.005245 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.248
||   0.0% |  0.000533 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.264
||===============================================================================
|   0.0% |  0.000105 |   -- |    -- |      1.0 |ETC
||-------------------------------------------------------------------------------
|   0.0% |  0.000105 |   -- |    -- |      1.0 | _END
|===============================================================================
```

Figura 5.8 – *CrayPat Report* (3ª etapa).

5.4. Quarta etapa de otimização

Estudo detalhado das variáveis utilizadas na região paralela no interior de *advectc* demonstrou que algumas destas variáveis eram usadas exclusivamente na execução das regiões aceleradas na GPGPU e que não eram referenciadas na CPU. Essas variáveis eram utilizadas para cálculos intermediários ("*scratch areas*") mas estavam sendo transmitidas entre a CPU e a GPGPU a cada invocação de *advectc*, por terem sido declaradas *presente_or_copyin* na terceira etapa de otimização. Tais variáveis foram declaradas *present_or_create*, de forma a eliminar seu tráfego entre a CPU e a GPGPU.

O tempo médio resultante desta quarta versão diminui $^{2}/_{3}$ em relação ao tempo médio da 3ª etapa, demandando 50,04 segundos ao utilizar o *auto_async_none* com 80,34% de utilização da capacidade de processamento da GPU e 50,50 segundos ao utilizar o *auto_async_kernel* com 80,24% de utilização da capacidade de processamento da GPU (Figura 5.9). Pela primeira vez neste trabalho uma versão OpenACC é mais rápida que a versão sequencial (56,44 segundos).

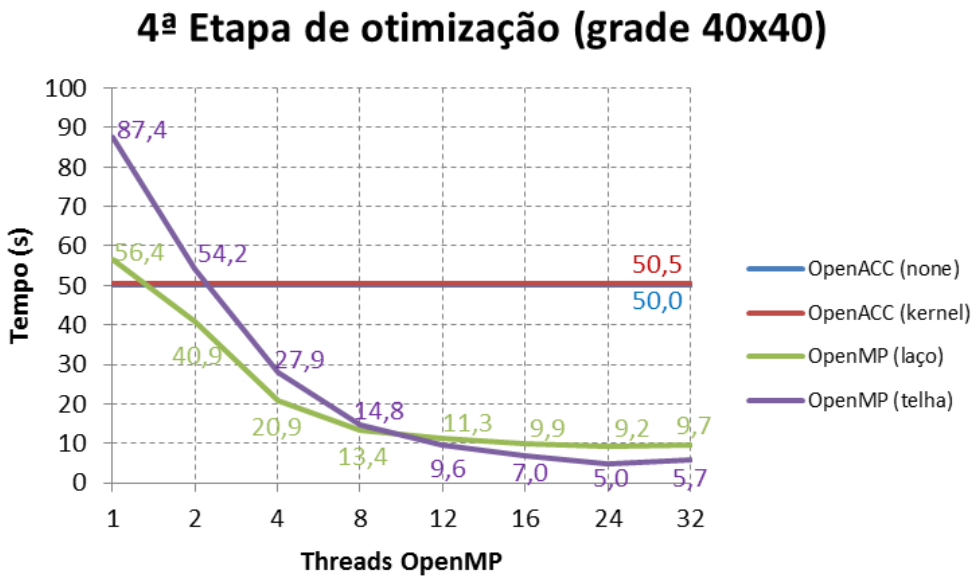


Figura 5.9 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 4ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

A saída do *profiler CrayPat* (Figura 5.10) mostra, pela primeira vez neste trabalho, que o procedimento com maior tempo de execução é um *kernel*, não uma transferência de dados. Entretanto, a soma dos tempos devido a transferência de dados ainda é alta, totalizando 19,54 segundos.

```
Table 1:  Profile by Function Group and Function

  Time% |       Time | Imb. |  Imb. |     Calls |Group
        |            | Time | Time% |           | Function

 100.0% | 47.597559 |   -- |    -- | 766806.0 |Total
|-------------------------------------------------------------------------------
| 100.0% | 47.597463 |   -- |    -- | 766805.0 |USER
||------------------------------------------------------------------------------
||  49.8% | 23.693058 |   -- |    -- | 151200.0 |runadvectc_.ACC_KERNEL@li.257
||  26.3% | 12.509716 |   -- |    -- | 324000.0 |runadvectc_.ACC_COPY@li.257
||  12.5% |  5.936550 |   -- |    -- |  32400.0 |atob_long_.ACC_COPY@li.918
||   5.1% |  2.430560 |   -- |    -- | 151200.0 |runadvectc_.ACC_REGION@li.257
||   2.8% |  1.322410 |   -- |    -- |  32400.0 |atob_long_.ACC_KERNEL@li.918
||   2.0% |  0.973508 |   -- |    -- |  10800.0 |runadvectc_.ACC_DATA_REGION@li.257
||   1.1% |  0.524232 |   -- |    -- |  32400.0 |atob_long_.ACC_REGION@li.918
||   0.3% |  0.128079 |   -- |    -- |  32400.0 |atob_long_.ACC_COPY@li.926
||   0.2% |  0.073772 |   -- |    -- |      1.0 |runadvectc_
||   0.0% |  0.005139 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.248
||   0.0% |  0.000385 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.267
||   0.0% |  0.000046 |   -- |    -- |      1.0 |runadvectc_.ACC_UPDATE@li.248
||   0.0% |  0.000008 |   -- |    -- |      1.0 |runadvectc_.ACC_UPDATE@li.267
||==============================================================================
|   0.0% |  0.000096 |   -- |    -- |      1.0 |ETC
||------------------------------------------------------------------------------
|   0.0% |  0.000096 |   -- |    -- |      1.0 | _END
|==============================================================================
```

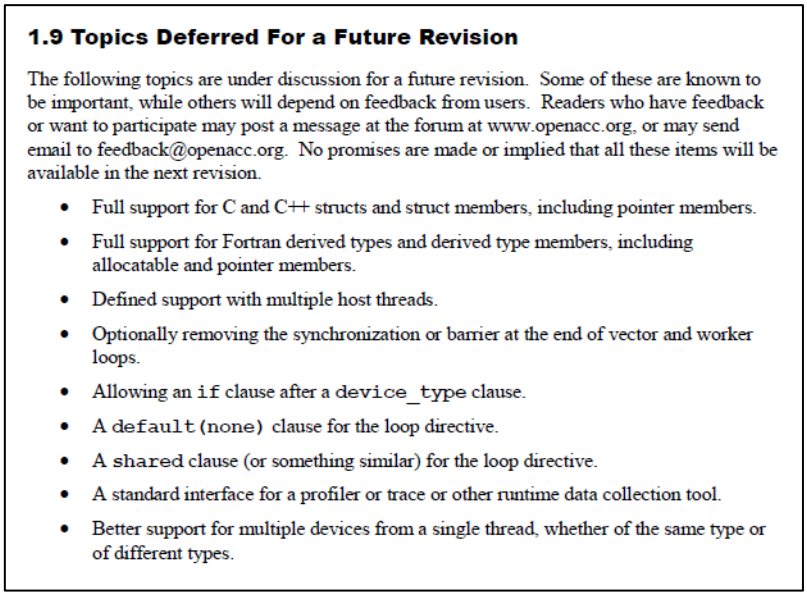
Figura 5.10 – *CrayPat Report* (4ª etapa).

5.5. Quinta etapa de otimização

Nesta etapa foi identificada e eliminada mais uma fonte de tráfego de dados entre a memória da CPU e a memória da GPGPU. O código original do modelo BRAMS utiliza uma estrutura de dados particular construída sobre um tipo derivado da linguagem Fortran denominado *scalar_table*. Este tipo derivado é composto por dois ponteiros que apontam para *arrays* tridimensionais que representam, respectivamente, os valores presentes e os valores das tendências temporais de um mesmo campo escalar. A estrutura de dados é um *array* unidimensional do tipo *scalar_table*, denominado *scalar_tab*. Esse *array* é dimensionado pelo número de campos escalares, que pode variar entre execuções do BRAMS em função das opções de execução escolhidas. Esta estrutura de dados é percorrida por *advectc* para realizar a advecção de cada campo escalar. Tal mecanismo permite que o texto de *advectc* seja independente do número selecionado de campos escalares.

Esperava-se que os dados apontados por esses ponteiros fossem transferidos da memória da CPU para a memória da GPGPU e em sentido oposto uma

única vez, mas constatou-se que a lista de campos apontados era transferida num ou noutro sentido várias vezes. Verificou-se então que o padrão OpenACC atual, versão 2.0 de junho de 2013 (OPENACC WORKING GROUP AND OTHERS, 2013), ainda não suporta tipos derivados e ponteiros, conforme documentado pelo trecho do padrão reproduzido na Figura 5.11. Alguns autores como (BEYER *et al.*, 2014) consideram que essa é a maior limitação para a larga aceitação de OpenACC. O compilador utilizado neste trabalho remove essa restrição copiando os campos apontados da memória da CPU para a memória da GPGPU na entrada de cada região acelerada que referencia a estrutura de dados e copiando no sentido inverso na saída da mesma região caso os campos sejam alterados.



1.9 Topics Deferred For a Future Revision

The following topics are under discussion for a future revision. Some of these are known to be important, while others will depend on feedback from users. Readers who have feedback or want to participate may post a message at the forum at www.openacc.org, or may send email to feedback@openacc.org. No promises are made or implied that all these items will be available in the next revision.

- Full support for C and C++ structs and struct members, including pointer members.
- Full support for Fortran derived types and derived type members, including allocatable and pointer members.
- Defined support with multiple host threads.
- Optionally removing the synchronization or barrier at the end of vector and worker loops.
- Allowing an `if` clause after a `device_type` clause.
- A `default(none)` clause for the loop directive.
- A `shared` clause (or something similar) for the loop directive.
- A standard interface for a profiler or trace or other runtime data collection tool.
- Better support for multiple devices from a single thread, whether of the same type or of different types.

Figura 5.11 – Padrão OpenACC Versão 2.0 201306.

A única solução possível para esta limitação é substituir a estrutura de dados não suportada por uma estrutura de dados suportada. Utilizou-se dois *arrays* tetradimensionais, o primeiro contendo os valores presentes e o segundo contendo as tendências temporais de todos os campos escalares. As três

primeiras dimensões de cada *array* armazenam os domínios de cada campo e a quarta dimensão a quantidade de campos escalares.

A redução do tempo de execução obtida por esta alteração foi substancial. O tempo de execução médio passou para 24,65 segundos com 69,30% de utilização da capacidade de processamento da GPU na opção *auto_async_none* e para 14,11 segundos com 100% de utilização da capacidade de processamento da GPU na opção *auto_async_kernel*, ou seja, menos da metade do tempo sequencial correspondente, conforme ilustrado na Figura 5.12.

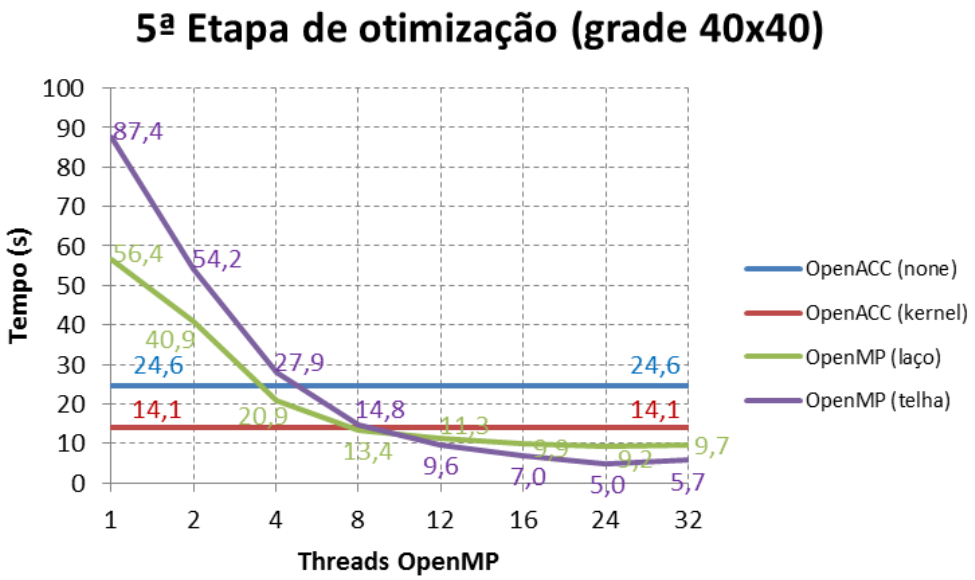


Figura 5.12 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 5ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

Como ilustrado na Figura 5.12 o tempo de execução é reduzido pelo uso de execuções assíncronas entre a CPU e a GPGPU com relação ao tempo de execução obtido em execuções síncronas. Tal ganho ocorre pela redução da frequência de troca de dados.

A saída do *profiler CrayPat* contida na Figura 5.13 permite analisar detalhadamente as atividades da GPGPU. As regiões aceleradas representadas por *ACC_REGION* e *ACC_KERNEL* são responsáveis por 89% do tempo de execução. Há uma única região de dados (*ACC_DATA_REGION*) com duas atualizações de dados (*ACC_UPDATE*), sendo uma no sentido da GPGPU antes do começo do laço (*ACC_COPY@li.273*) e a outra no sentido da CPU logo após o final do laço (*ACC_COPY@li.287*).

```
Table 1:  Profile by Function Group and Function

  Time% |      Time | Imb. |  Imb. |    Calls |Group
        |           | Time | Time% |          | Function

 100.0% | 22.063867 |   -- |    -- | 766806.0 |Total
|-------------------------------------------------------------------------------
| 100.0% | 22.063772 |   -- |    -- | 766805.0 |USER
||------------------------------------------------------------------------------
||  77.2% | 17.035208 |   -- |    -- | 183600.0 |runadvectc_.ACC_KERNEL@li.278
||  12.6% |  2.775671 |   -- |    -- | 183600.0 |runadvectc_.ACC_REGION@li.278
||   5.6% |  1.243264 |   -- |    -- | 388800.0 |runadvectc_.ACC_COPY@li.278
||   4.2% |  0.927826 |   -- |    -- |  10800.0 |runadvectc_.ACC_DATA_REGION@li.278
||   0.3% |  0.074479 |   -- |    -- |      1.0 |runadvectc_
||   0.0% |  0.006483 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.273
||   0.0% |  0.000764 |   -- |    -- |      1.0 |runadvectc_.ACC_COPY@li.287
||   0.0% |  0.000069 |   -- |    -- |      1.0 |runadvectc_.ACC_UPDATE@li.273
||   0.0% |  0.000009 |   -- |    -- |      1.0 |runadvectc_.ACC_UPDATE@li.287
||===============================================================================
|   0.0% |  0.000094 |   -- |    -- |      1.0 |ETC
||------------------------------------------------------------------------------
|   0.0% |  0.000094 |   -- |    -- |      1.0 | _END
|===============================================================================
```

Figura 5.13 – *CrayPat Report* (5ª etapa).

Nessa mesma figura o maior tempo de execução devido a transferência de dados ocorre nas execuções de *ACC_COPY@li.278.* Análise detalhada reportada a seguir demonstra que esse tempo é um *overhead* (possivelmente ampliado pelo *CrayPat*) apenas para determinar que as variáveis nas regiões aceleradas estão presentes (*present*) na memória da GPGPU.

Esse fato é verificado habilitando a variável de ambiente *CRAY_ACC_DEBUG*. Essa variável de ambiente controla o nível de coleta de informações por traçadores de eventos na GPGPU e, por consequência, das informações contidas em relatório emitido após a execução. Tal relatório é emitido apenas quando essa variável de ambiente estiver habilitada. A Figura 5.14 contém fração do relatório emitido quando foi escolhido o nível mais detalhado da

coleta de informações. Essa figura contém um trecho selecionado de uma execução de *ACC_COPY@li.278*. O relatório mostra que a transferência de 32 variáveis da memória da CPU para a memória da GPGPU demandou zero microsegundos, pois todas as variáveis já estavam presentes na memória da GPGPU.

```
ACC: Start transfer 32 items from MainIsolaAdvectc.f90:278
ACC:   flags: RETURN_ACC_TIME
ACC:
ACC:   Trans 1
ACC:       Simple transfer of 'dn0' (268800 bytes)
ACC:            host ptr 2ea0e40
ACC:            acc  ptr 700400000
ACC:            flags: REL_PRESENT REG_PRESENT
ACC:            release acc 700400000 from present table index 2 (ref_count 1)
ACC:            release present (acc 700400000)
ACC:
ACC:   Trans 2
ACC:       Simple transfer of 'dn0u' (268800 bytes)
  .                .
  .                .
  .                .
ACC:   Trans 32
ACC:       Simple transfer of 'zt' (168 bytes)
ACC:            host ptr 1207740
ACC:            acc  ptr 0
ACC:            flags: ACQ_PRESENT REG_PRESENT
ACC:            host region 1207740 to 12077e8 found in present table index 34 (ref count 2)
ACC:            memory found in present table (700600800, base 700600800)
ACC:            new acc ptr 700600800
ACC:
ACC: End transfer (to acc 0 bytes, to host 0 bytes, time 0 usec)
```

Figura 5.14 – Trecho da Saída do Código (*CRAY_ACC_DEBUG*).

5.6. Sexta etapa de otimização

As etapas anteriores objetivaram minimizar as transferências de dados entre as memórias da CPU e da GPGPU. Nesta última etapa buscou-se otimizar o tempo de execução dos *kernels* na GPGPU. Na linguagem CUDA o espaço de índices de um aninhamento de laços é mapeado para uma grade de blocos de *threads*, sendo cada bloco da grade executado por uma *gang* de *threads*. Mecanismo similar existe em OpenACC, onde o compilador pode atribuir automaticamente o número de *gangs* (parâmetro *num_gangs*) e o número de *threads* dentro de cada *gang* (parâmetro *vector_length*), mas também é possível especificar manualmente estes parâmetros. Esta etapa especifica manualmente estes parâmetros em função da arquitetura da placa.

O modelo de GPGPU utilizado nos testes tem 13 multiprocessadores, cada um com 192 núcleos, sendo que cada multiprocessador pode executar *threads* de diferentes *gangs*, mas cada *gang* somente pode ser executada por um mesmo multiprocessador. Existe ainda a limitação do *warp size*, que impõe que cada *gang* seja executada à razão de 32 *threads* de cada vez. A escolha dos parâmetros *num_gangs* igual a 3900 (múltiplo de 13) e de *vector_length* igual a 64 (múltiplo do *warp size*) permitiu uma pequena melhora de desempenho (com os tempos médios de 24,21 e 13,68 segundos, 68,70% e 100% de utilização da capacidade de processamento da GPU para as compilações com *auto_async_none* e *auto_async_kernel* respectivamente) em relação à escolha automática pelo compilador OpenACC (Figura 5.15). Na etapa anterior, o compilador escolhia esses parâmetros automaticamente, usando um número de *gangs* de 128, 1444 ou 1600 conforme o laço considerado e usando sempre 128 *threads* por *gang*.

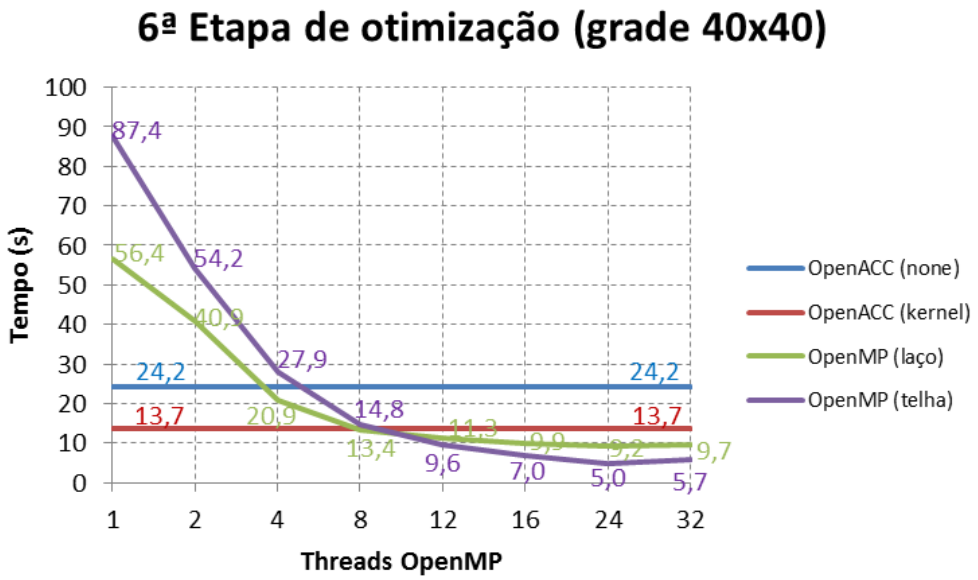


Figura 5.15 – Comparação dos tempos de execução das versões OpenMP por laços e por telhas 2x2 em função do número de *threads* com o tempo de execução das versões da 6ª etapa do OpenACC com opções de compilação *async_kernel* e *async_none.*

5.7. Desempenho com OpenACC nas demais grades

Para avaliar o desempenho de OpenACC nas demais grades o código da sexta etapa de otimização foi executado em cada grade. Os resultados são reportados na Figura 5.16 a seguir, a qual compara os tempos da execução com OpenACC na opção *auto_async_kernel* com os tempos da execução da versão OpenMP por laços em função do número de *threads* para todas as grades.

Notamos que o tempo de execução da versão OpenACC é consistentemente menor que o tempo de execução sequencial (uma *thread*) para todas as grades de tamanho superior a 10x10. Notamos ainda que o tempo de execução da versão OpenACC é cada vez mais competitivo com o tempo de execução da versão OpenMP com o aumento do tamanho da grade, pois o número de *threads* necessário para que a versão OpenMP seja mais rápida que a versão OpenACC aumenta com o tamanho da grade. São necessárias acima de quatro *threads* na grade 30x30 ou então, acima de dezesseis *threads* na grade 100x100 para que a versão OpenMP seja mais rápida.

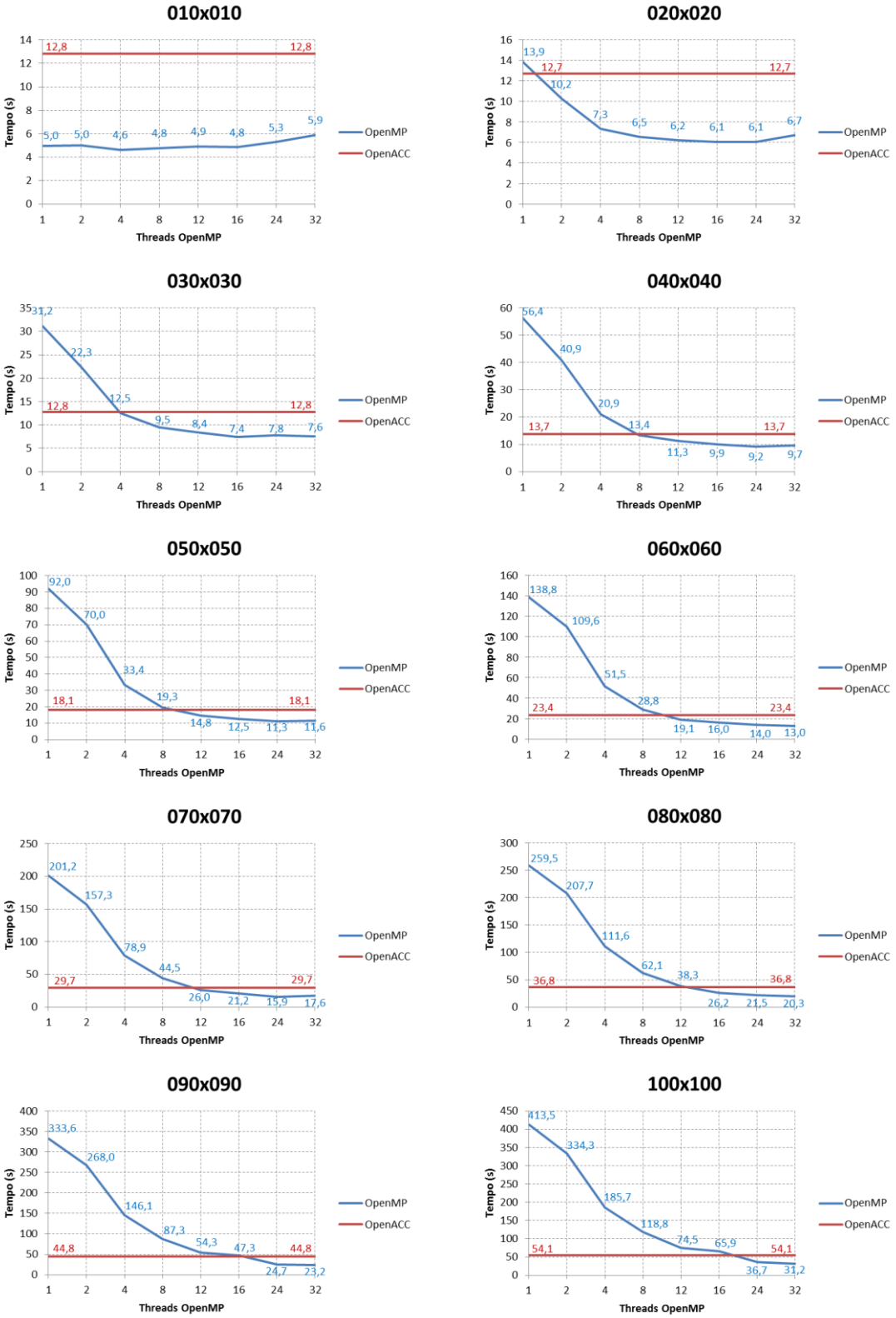


Figura 5.16 – Comparação dos tempos de processamento da melhor versão OpenACC com a versão OpenMP codificada por laços para diferentes números de *threads.*

5.8. Conclusões sobre o desempenho com OpenACC

Neste capítulo, constatou-se que a melhor versão OpenACC, correspondente à 6ª etapa de otimização com uso da opção *auto_async_kernel*, é competitiva com a versão OpenMP paralelizada por laços apresentada no capítulo 4 para tamanhos de grade maiores, conforme a análise de desempenho apresentada na seção anterior. Isso demonstra ser possível o uso de uma única codificação com diretivas de paralelização OpenMP e OpenACC para execução, respectivamente, em arquiteturas *multi-core* ou *many-core* com eficiência razoável em ambos os casos.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho explorou a proposta de se portar um código existente para execução paralela em arquiteturas *multi-core* ou *many-core*, por meio da inserção de diretivas de paralelização OpenMP e OpenACC, respectivamente. O código escolhido como estudo de caso é a advecção de escalares, um trecho da dinâmica do modelo meteorológico regional BRAMS.

Os testes de desempenho realizados para diferentes tamanhos de grade do modelo demonstraram que é possível obter-se eficiência em termos de desempenho paralelo em ambas as arquiteturas com um único código. Naturalmente, é possível obter-se códigos mais eficientes específicos para uma ou outra arquitetura, mas essa abordagem seria menos conveniente devido ao esforço para se gerar códigos distintos, especialmente no caso de códigos extensos como modelos meteorológicos.

Além disso, o presente trabalho representa uma primeira codificação em OpenACC de um trecho da dinâmica de um modelo meteorológico regional, no caso, da parte correspondente à advecção do BRAMS, à luz do estado da arte recente. Há casos em que a linguagem CUDA foi utilizada para a dinâmica, mas a programação em CUDA requer um esforço bem maior, dado seu baixo nível de abstração.

Outra contribuição é uma versão OpenMP do mesmo código paralelizada por telhas, em vez da versão usual paralelizada por laços utilizada no código único OpenMP–OpenACC. Embora essa versão por telhas tenha desempenho melhor em OpenMP, seu desempenho em OpenACC é questionável, por conter regiões paralelas menores que as da versão por laços. É importante frisar que portar modelos existentes paralelizados com MPI para codificações híbridas com OpenMP ou OpenACC representa um esforço de programação substancial e muito provavelmente esse esforço se tornaria inviável no caso de desenvolver versões independentes para OpenMP e OpenACC.

Lembramos que no capitulo anterior houve a necessidade de substituir a estrutura de dados do tipo ponteiro (*scalar_tab*) para *array* tetradimensionais (referente a 5ª etapa de otimização), com isso, foram realizados novamente os experimento para verificar se houve algum impacto aos tempos médios obtidos. Os resultados demonstram que seus tempos médios de execuções foram similares para essas duas codificações (estrutura de ponteiros e *arrays*) em OpenMP por laços com uma diferença media de 0,019%, relembrando que sua modificação foi necessária pelo fato do OpenACC não suportar estruturas de ponteiros ainda.

A otimizações de código realizadas em OpenMP e OpenACC só foram possíveis graças à utilização do conjunto de ferramentas da CRAY denominado *CrayPat*, por meio do modulo *perftools*.

Em termos de trabalhos futuros, vislumbra-se portar o restante da dinâmica do modelo BRAMS com essa abordagem de código único OpenMP–OpenACC e, eventualmente estendê-la para os demais módulos do modelo. Outro trabalho futuro seria desenvolver uma versão única OpenMP–OpenACC paralelizada por telhas, o que exigirá investigar se é possível obter bom desempenho do código também em OpenACC. Eventualmente, a baixa granularidade implicada pelas telhas pode inviabilizar um bom desempenho na arquitetura *many-core*. Em termos de otimização da memória em OpenMP, pode-se investigar a influência dos diferentes tamanhos de página de memória, que podem ser selecionados na compilação. Finalmente, o desempenho paralelo em ambas arquiteturas poderia ser avaliado na execução do modelo BRAMS completo, simulando seu uso operacional.

REFERÊNCIAS BIBLIOGRÁFICAS

ABOUT GrADS Gridded data sets, 2015. Disponível em: <http://www.iges.org/grads/gadoc/aboutgriddeddata.html>. Acesso em: 11 março 2015.

ALMEIDA, E. S.; BAUER, M. Reducing time delays in computing numerical weather models it regional and local levels: agrid-based approach. **International Journal of Grid Computing & Applications**, v. 3, n. 4, p. 1-17, 2012.

BEYER, J.; OEHMKE, D.; SANDOVAL, J. **Transferring user-defined types in OpenACC**. CUG - Cray User Group, 2014. 15 p.

BRAMS. **Model Description**, 2015. Disponível em: <http://brams.cptec.inpe.br/>. Acesso em: 11 março 2015.

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. **Using OpenMP:** portable shared memory parallel programming. [S.l.]: MIT press, v. 10, 2008. 384 p.

NVIDIA CORPOTARION. **Computação de alta performance**. 2014. Disponível em: <http://www.nvidia.com.br/object/tesla-supercomputing-solutions-br.html>. Acesso em: 05 jan. 2014.

COSTA, J. A. D. C.; SENA, M. C. R. **Tutorial OpenMP C/C++**. Maceio, AL:Sun/LCCV, mar. 2008. Disponível em: <http://grenoble.ime.usp.br/~gold/cursos/2014/labprogII/aulas/tutorialOpenMP.pdf>. Acesso em: 05 jan. 2015.

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS. CENTRO DE PREVISÃO DE TEMPO E ESTUDOS CLIMÁTICOS (INPE.CPTEC). **Produtos operacionais.** 2015. Disponivel em: <http://www.cptec.inpe.br/>. Acesso em: 01 fev. 2015.

FAZENDA, A. L.; PANETTA, J.; KATSURAYAMA, D. M.; RODRIGUES, L. F.; MOTTA, L.; NAVAUX, P. Challenges and solutions to improve the scalability of an operational regional meteorological forecasting model. **International Journal of High Performance Systems Architecture**, v. 3, n. 2, p. 87-97, 2011.

FREITAS, S. R.; LONGO, K. M.; DIAS, M. S.; CHATFIELD, R.; DIAS, P. S.; ARTAXO, P.; ANDREAE, M.; GRELL, G.; RODRIGUES, L. F.; FAZENDA, A.; OTHERS. The coupled aerosol and tracer transport model to the Brazilian

developments on the Regional Atmospheric Modeling System (CATT-BRAMS) Part 1: Model description and evaluation. **Atmospheric Chemistry and Physics Discussions**, v. 7, n. 3, p. 8525-8569, 2007.

FREITAS, S. R.; LONGO, K.; DIAS, M. S.; DIAS, P. S.; CHATFIELD, R.; FAZENDA, A.; RODRIGUES, L. F. The coupled aerosol and tracer transport model to the Brazilian developments on the Regional Atmospheric Modeling System: Validation using direct and remote sensing observations. In: INTERNATIONAL CONFERENCE ON SOUTHERN HEMISPHERE METEOROLOGY AND OCEANOGRAPHY (ICSHMO), 8, 2006. Foz do Iguaçu. **Proceedings…** São José dos Campos:INPE, 2006, p. 101-107.

GROPP, W.; LUSK, E.; SKJELLUM, A.; THAHUR, R. **Using MPI**: portable parallel programming with the message passing interface (Scientific and Engineering Computation). 2. ed. Cambridge:The MIT Press, 1999.

HETEROGENEOUS Multi-Core Workshop. In: PROGRAMMING WEATHER, CLIMATE, AND EARTH-SYSTEM MODELS ON HETEROGENEOUS MULTI-CORE PLATFORMS WORKSHOP, 2014. Colorado. Disponível em: <http://data1.gfdl.noaa.gov/multi-core>. Acesso em: 06 abr. 2015.

KALNAY, E. **Atmospheric modeling, data assimilation, and predictability**. 1. ed. Cambridge: Cambridge University Press, 2003. 364 p.

KHRONOS Group, 2015. Disponível em: <https://www.khronos.org/opencl/>. Acesso em: 18 mar. 2015.

LAURITZEN, P. H. **Atmospheric dynamics** - the CAM-FV dynamical core - CGD, 27 jul. 2009. Disponível em: <http://www.cgd.ucar.edu/cms/pel/papers/L2009CAM.pdf>. Acesso em: 11 mar. 2015.

LEE, S.; MIN, S.-J.; EIGENMANN, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. **ACM Sigplan Notices**, v. 44, n. 4, p. 101-110, 2009.

LONGO, K. M.; FREITAS, S. R.; PIRRE, M.; MARÉCAL, V.; RODRIGUES, L. F.; PANETTA, J.; ALONSO, M.; ROSÁRIO, N.; MOREIRA, D.; GÁRCITA, M.; OTHERS. The chemistry CATT-BRAMS model (CCATT-BRAMS 4.5): a regional atmospheric model system for integrated air quality and weather forecasting and research. **Model Dev. Discuss**, v. 6, p. 1173-1222, 2013.

LYNCH, P. The origins of computer weather prediction and climate modeling. **Journal of Computational Physics**, v. 227, n. 7, p. 3431-3444, 2007.

MESINGER, F.; ARAKAWA, A. **Numerical methods used in atmospheric models**. Geneva: Global Atmospheric Research Program World Meteorological Organization, v. 1, 1976.

OLSON, D. A.; JUNKER, N. W.; KORTY, B. Evaluation of 33 years of quantitative precipitation forecasting at The NMC. **Weather and Forecasting**, v. 10, n. 3, p. 498-511, 1995.

NVIDIA CUDA ZONE. **OPEN**: The Origins of OpenACC. , 2014. Disponível em: <http://devblogs.nvidia.com/parallelforall/openacc-directives-gpus/>. Acesso em: 05 jan. 2014.

OPENACC. **OpenACC Directives for Accelerators**, 2015. Disponível em: <http://www.openacc-standard.org/>. Acesso em: 17 mar. 2015.

OPENACC WORKING GROUP AND OTHERS. T**he OpenACC application programming interface**. OpenACC Home. 2013. Disponível em: <http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf>. Acesso em: 05 jan. 2014.

PANETTA, J. **Histórico de desenvolvimento do CCATT-BRAMS**. 2012. Disponível em: <https://projetos.cptec.inpe.br/attachments/237/JairoPanetta-I.pdf>. Acesso em: 01 fev. 2015. Workshop CCATT-BRAMS.

PERFORMANCE Analysis Tools. **NVIDIA**. 2015. Disponível em: <https://developer.nvidia.com/performance-analysis-tools>. Acesso em: 30 mar. 2015.

RODRIGUES, E. R.; NAVAUX, P. O.; PANETTA, J.; FAZENDA, A. L.; MENDES, C.; KALE, L. V. A comparative analysis of load balancing algorithms applied to a weather forecast model. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 22., 2010, Petrópolis. **Proceedings…** IEEE, 2010. p. 71-78.

RUIZ, R.; VELHO, H. D. C.; LESSA, L.; CHARÃO, A. Turbulent parameterization for CCATT-BRAMS by GPU. **Ciência e Natura**, p. 196-198, 2013.

SILVA JUNIOR, M. B. D.; PANETTA, J.; STEPHANY, S. Escalabilidade de diferentes formas de paralelismo na advecção de campos escalares do BRAMS. **Tema**, Submetido em 2015.

THUBURN, J. Some conservation issues for the dynamical cores of NWP and climate models. **Journal of Computational Physics**, v. 227, n. 7, p. 3715-3730, 2008.

TREMBACK, C. J.; POWELL, J.; COTTON, W. R.; PIELKE, R. A. The forward-in-time upstream advection scheme: extension to higher orders. **Monthly Weather Review**, v. 115, n. 2, p. 540-555, 1987.

TRIPOLI, G. J.; COTTON, W. R. The colorado state university three-dimensional cloud-mesoscale model. Part I: general theoretical framework and sensitivity experiments. **Journal de recherches atmosphériques**, v. 16, p. 185-220, 1982.