UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Emmanuel Podestá Junior

PSKEL-MPPA: UMA ADAPTAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Florianópolis

2018

Emmanuel Podestá Junior

PSKEL-MPPA: UMA ADAPTAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Márcio Bastos Castro
Universidade Federal de Santa Catarina

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Podestá Junior, Emmanuel
PSkel-MPPA: : uma adaptação do framework PSkel para o processador manycore MPPA-256 / Emmanuel Podestá Junior ; orientador, Márcio Bastos Castro, 2018.
136 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2018.

Inclui referências.

1. Ciências da Computação. 2. stencil. 3. manycores. 4. PSkel. 5. MPPA-256. I. Bastos Castro, Márcio. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Emmanuel Podestá Junior

PSKEL-MPPA: UMA ADAPTAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciências da Computação.

Florianópolis, 01 de julho 2018.

Prof. Dr. Renato Cislaghi
Universidade Federal de Santa Catarina
Coordenador

Banca Examinadora:

Prof. Dr. Márcio Bastos Castro
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Laércio Lima Pilla
Universidade Grenoble Alpes

Prof. Dr. Frank Augusto Siqueira
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço a todos que contribuíram de alguma forma para a realização do presente Trabalho de Conclusão de Curso. Em especial, agradeço meu orientador, Márcio Bastos Castro, e os colegas do grupo de pesquisa que estiveram envolvidos diretamente no presente trabalho. Também agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pela concessão da bolsa de Iniciação Científica (PIBIC), cujas atividades relacionadas fomentaram o desenvolvimento deste trabalho.

"Whether you think that you can, or that you can't, you are usually right."

Henry Ford (1863-1947)

RESUMO

Aplicações paralelas podem ser classificadas de acordo com o padrão de computação e coordenação. Dentre os padrões mais conhecidos destacam-se o *map*, *reduce*, *pipeline*, *scan* e *stencil*. Este último é muito utilizado em diversas áreas, como simulação física de partículas, previsão meteorológica, termodinâmica, resolução de funções diferenciais, manipulação de imagens, entre outras. O PSkel é um *framework* de programação paralela desenvolvido para simplificar o desenvolvimento de aplicações que seguem o padrão *stencil*. Utilizando uma abstração de alto nível, programador define o *kernel* da computação, enquanto o *framework* se encarrega de executar a computação paralela em *multicores* e em *Graphics Processing Units* (GPUs) de maneira eficiente. O objetivo deste trabalho é propor uma adaptação do *framework* PSkel para o processador *manycore* emergente MPPA-256, batizada de PSkel-MPPA. A motivação para tal adaptação está relacionada à dificuldade de desenvolvimento de aplicações do padrão *stencil* para o MPPA-256, tendo em vista as suas características arquiteturais intrínsecas que tornam o desenvolvimento de aplicações paralelas onerosas e suscetíveis a erros. A adaptação do *framework* permite simplificar o desenvolvimento de aplicações *stencil* para o MPPA-256, escondendo do desenvolvedor detalhes de implementação, tais como a necessidade de comunicação explícita entre as memórias do *chip* e a distribuição de computações entre os núcleos de processamento. Diversos experimentos foram efetuados com a solução proposta para o MPPA-256, comparando-a com a solução para *multicores* já existente. Os resultados mostraram que a solução proposta para o MPPA-256 permite reduzir o consumo de energia das aplicações *stencil* em até 1.45x apesar de apresentar uma perda de desempenho de até 3.3x.

Palavras-chave: *manycores*, MPPA-256, *stencil*, PSkel.

ABSTRACT

Parallel applications can be classified according to computation and coordination patterns. Among the most well-known patterns are the map, reduce, pipeline, scan and stencil. The latter is widely used in several areas, such as physical particle simulation, weather forecasting, thermodynamics, resolution of differential functions, manipulation of images, among others. PSkel is a parallel programming framework designed to simplify the development of applications that follow the stencil pattern. Using a high level abstraction, the programmer defines the compute kernel, while the framework executes the parallel computation efficiently on multicores and on Graphic Processing Units (GPUs). The objective of this work is to propose an adaptation of the PSkel framework for the emerging manycore processor MPPA-256, called PSkel-MPPA. The motivation for such work is related to the difficulty of developing parallel stencil applications on MPPA-256. The intrinsic architectural features of MPPA-256 make the development of efficient parallel applications a costly and error-prone task. The framework adaptation allows to simplify the development of stencil applications for MPPA-256, hiding implementations details from the developer, such as the need for explicit communication between on-chip memories and task distribution between processing cores. Several experiments were carried out with the proposed solution for the MPPA-256, comparing it to the existing multicore counterpart. The results showed that the proposed solution allows to reduce energy consumption of stencil applications in up to 1.45x, despite the performance loss of up to 3.3x.

Keywords: manycores, MPPA-256, stencil, PSkel.

LISTA DE FIGURAS

SIGLAS

API – *Application Programming Interface*

CC-NUMA – *Cache-Coherent Non-Uniform Memory Access*

CMPs – *Chip Multiprocessors*

COW – *Clusters of Workstations*

CPU – *Central Processing Unit*

CPUs – *Central Processing Units*

E/S – Entrada e Saída

Flops – *Floating-point Operations per Second*

GPU – *Graphics Processing Unit*

HPC – *High Performance Computing*

IPC – *Inter-Process Communication*

LPDDR3 – *Low Power Double Data Rate 3*

MPI – *Message Passing Interface*

MPSoC – *Multiprocessor System-on-Chip*

NC-NUMA – *No Cache Non-Uniform Memory Access*

NoC – *Network-on-Chip*

NOW – *Network of Workstations*

NUMA – *Non-Uniform Memory Access*

OpenCL – *Open Computing Language*

PE – *Processing Element*

POSIX – *Portable Operating System Interface*

QPI – *Quick Path Interconnect*

RAM – *Random-Access Memory*

RAPL – *Running Average Power Limit*

RM – *Resource Manager*

SIMD – *Single Instruction, Multiple Data*

SO – Sistema Operacional

SPMD – *Single Program, Multiple Data*

UMA – *Uniform Memory Access*

SUMÁRIO

1 INTRODUÇÃO

Durante muito tempo, os avanços tecnológicos possibilitavam aumentar o desempenho de semicondutores e das arquiteturas por meio do aumento da frequência. Contudo, com o aumento da frequência, ocorre um maior consumo de energia e, consequentemente, a temperatura do *chip* aumenta. Desta forma, as indústrias começaram a investir em outros meios para aumentar o desempenho de arquiteturas, como, por exemplo, os processadores *multicore*.

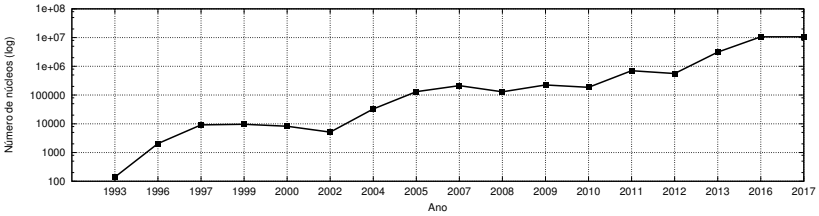
Arquiteturas utilizadas na área de *High Performance Computing* (HPC) empregam processadores *multicore* para atingir um processamento de uma imensa quantidade de dados em menos tempo. Mais especificamente, aplicações com uma grande quantidade de operações, como aplicações para previsões metereológicas ou processamento de imagens, são exemplos onde temos uma grande quantidade de dados. Operações com imagens grandes, por exemplo, podem levar muito tempo para finalizar, pois há uma grande quantidade de elementos que devem ser processados. Desta forma, supercomputadores e *clusters* de computadores são utilizados para tornar a computação de aplicações ou dados tratável do ponto de vista computacional. Essas arquiteturas são utilizadas em aplicações científicas ou, mais atualmente, no processamento de grandes volumes de dados em aplicações de *Big Data*. Normalmente, arquiteturas HPC são compostas por processadores do tipo *Central Processing Unit* (CPU) e aceleradores, tais como *Graphics Processing Unit* (GPU). Atualmente, o número de núcleos (*cores*) em arquiteturas *multicore* aumentam continuamente. A Figura 1 mostra o número total de núcleos do supercomputador em primeiro lugar no *ranking* do TOP500[1], confirmando esse comportamento.

Até a última década, o desempenho das arquiteturas utilizadas em HPC era quantificado quase que exclusivamente pelo seu poder de processamento, usualmente medido por *Floating-point Operations per Second* (Flops). Contudo, o consumo excessivo de energia é uma barreira para o aumento de desempenho de forma escalável nestas plataformas. Essa preocupação com o aumento de energia é ressaltada pelo relatório emitido pelo Departamento de Defesa do Governo dos Estados Unidos (DARPA/IPTO) (KOGGE et al., 2008). O relatório ressalta que a potência máxima aceitável para supercomputadores *Exascale* (10^{18} Flops) seria de 20 MW[2], isto é, com essas características,

[1] `http://top500.org`

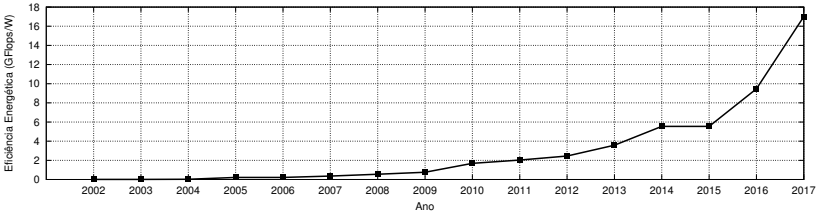
[2] Atualmente, a potência máxima está sendo considerada em até 30 MW

Figura 1 – Crescimento do número de núcleos do supercomputador número 1 do *ranking* Top500 (dados extraídos do site Top500).



Fonte: desenvolvido pelo autor.

Figura 2 – Eficiência energética do supercomputador número 1 do *ranking* Green500 (dados extraídos do site Green500).



Fonte: desenvolvido pelo autor.

um supercomputador deveria possuir uma eficiência energética de 50 GFlops. Por outro lado, atualmente, os supercomputadores mais energicamente eficientes possuem uma eficiência energética de no máximo 17 GFlops/W, como pode ser observado nos dados extraídos do *ranking* Green500[3] mostrados na Figura 2. Este valor é quase 3 vezes menor que o ressaltado pelo relatório da DARPA/IPTO, sendo assim, atualmente a eficiência energética de supercomputadores está ainda distante da ideal. Portanto, para que supercomputadores tenham potencial para atingir o *Exascale* é necessário um consumo energético viável e um alto desempenho. Contudo, o crescimento dessas duas variáveis não é proporcional.

Por essa razão, o estudo de técnicas que melhorem a eficiência energética em plataformas HPC está se tornando muito importante. Recentemente, uma nova classe de processadores *manycore* de baixa potência, tais como, o Sunway SW26010 (FU et al., 2016), Mellanox TILE-Gx (MORARI et al., 2012) e Kalray MPPA-256 (CASTRO et al.,

[3] `https://www.top500.org/green500/`

2013), foi desenvolvida. Esses processadores possuem centenas de núcleos de processamento capazes de lidar com paralelismo de dados e tarefas com baixo consumo de energia.

Apesar desses processadores *manycore* fornecerem uma melhor eficiência energética (FRANCESQUINI et al., 2014), eles possuem uma arquitetura particular que torna o desenvolvimento de aplicações paralelas uma tarefa desafiadora (VARGHESE et al., 2014; CASTRO et al., 2016, 2014). Núcleos de processamento sem coerência de *cache* são, geralmente, distribuídos em uma arquitetura organizada em *clusters*, onde cada *cluster* possui uma memória local (compartilhada somente entre os núcleos do *cluster*). Dessa forma, a comunicação entre *clusters* deve ser efetuada através de uma *Network-on-Chip* (NoC) de maneira distribuída. Por essa razão, o tempo de comunicação pode variar entre os núcleos que estão se comunicando.

O MPPA-256 possui 256 núcleos distribuídos em 16 *clusters*, denominados *Processing Element*s (PEs), e 4 subsistemas de Entrada e Saída (E/S). O ambiente do MPPA-256 é heterogêneo, sendo utilizada, entre os *clusters* e subsistemas, uma comunicação via NoC e dentro de cada *cluster* são utilizadas comunicações diretas entre PEs, por meio de memória compartilhada. As principais dificuldades do desenvolvimento de aplicações para o MPPA-256 são resumidas a seguir:

- **Modelo de programação híbrido**: problema citado anteriormente, onde *threads* em um mesmo *cluster* se comunicam através de memória compartilhada local, porém a comunicação entre *clusters* é feita explicitamente via NoC, seguindo um modelo de computação distribuída;
- **Comunicação explícita**: é necessária a utilização de uma *Application Programming Interface* (API) específica para a comunicação via NoC, similar ao *Portable Operating System Interface* (POSIX) de baixo nível para *Inter-Process Communication* (IPC);
- **Memória limitada**: cada *cluster* possui apenas 2MB de memória local de baixa latência, portanto aplicações reais precisam constantemente realizar comunicações com o subsistema de E/S;
- **Ausência de coerência de *cache***: cada PE possui uma memória *cache* privada e sem coerência com as *caches* dos demais PEs, sendo necessário atualizar a *cache* manualmente.

Uma possível solução para os problemas apresentados anteriormente é a utilização de padrões paralelos ou esqueletos (COLE, 2004).

Esqueletos são modelos de programação paralela de alto nível de abstração. Esses modelos fornecem vantagens para o desenvolvedor, escondendo a complexidade de aplicações paralelas e distribuídas. Os esqueletos especificam, mais precisamente, os padrões de acesso de dados e comunicação. Desta forma, eles possibilitam aos desenvolvedores focarem apenas nos algoritmos, ao invés da comunicação, sincronização de tarefas e escalonamento, que são gerenciados de forma transparente pelo *framework* que implementa um esqueleto. Entre os diversos esqueletos existentes (*map*, *reduce*, *pipeline* e *scan*), o padrão *stencil* é utilizado em ambientes industriais e acadêmicos, como física quântica, previsão do tempo e processamento de imagens (GONZALEZ; WOODS, 2006; HOLEWINSKI; POUCHET; SADAYAPPAN, 2012).

Frameworks são utilizados para fornecer uma abstração sobre partes de código que serão reusadas diversas vezes. Esse *framework* pode ser estendido ou adaptado para fornecer suporte para outras aplicações com diferentes características. Dessa forma, a utilização de um *framework* pode facilitar o desenvolvimento de aplicações para ambientes onerosos, como ambientes *manycore*.

Muitos *frameworks* foram propostos para o auxílio no desenvolvimento de aplicações paralelas do padrão estêncil, como o PSkel (PEREIRA; RAMOS; GÓES, 2015), SkePU (ENMYREN; KESSLER, 2010) e SkelCL (STEUWER; KEGEL; GORLATCH, 2011). Em particular, o PSkel é um *framework* que fornece uma abstração em alto nível para o desenvolvimento em ambientes heterogêneos CPU-GPU, enquanto particiona tarefas e dados de forma transparente entre esses processadores.

1.1 OBJETIVOS

Com base no exposto, são apresentados a seguir o objetivo geral e os objetivos específicos do presente projeto.

1.1.1 Objetivo Geral

O objetivo principal deste TCC é propor uma adaptação do *framework* PSkel para o processador *manycore* emergente denominado MPPA-256, facilitando assim o desenvolvimento de aplicações *stencil* neste processador. A adaptação permitirá que aplicações já existentes do PSkel possam ser executadas no MPPA-256 sem nenhuma necessidade de alteração de código.

1.1.2 Objetivos Específicos

- Definir uma estratégia de distribuição de dados entre os *clusters* do MPPA-256 a fim de lidar com a capacidade limitada de memória no *chip*;
- Propor e implementar técnicas que permitam reduzir os custos de comunicação na NoC;
- Adaptar as principais classes e abstrações existentes no PSkel para o processador MPPA-256;
- Realizar uma análise do desempenho e do consumo de energia da solução proposta para o MPPA-256 utilizando diversas aplicações estêncil já implementadas no PSkel;
- Realizar comparações de desempenho e consumo de energia com um processador *multicore* atual.

1.2 CONTRIBUIÇÕES DO TRABALHO

As contribuições principais deste trabalho foram publicadas em diferentes eventos da área de Computação Paralela e Distribuída. Abaixo são apresentados os artigos publicados:

- PODESTA JUNIOR, E. ; MARQUES, B. ; CASTRO, M. **Energy Efficient Stencil Computations on the Low-Power Many-core MPPA-256 Processor**. In: Conferência Européia Internacional de Computação Paralela e Distribuída (EURO-PAR), 2018, Turin, Itália.
- PODESTA JUNIOR, E. ; PEREIRA, A. D. ; ROCHA, R. C. O. ; CASTRO, MÁRCIO ; GOES, L. F. W. **Execução Energeticamente Eficiente de Aplicações Estêncil com o Processador Manycore MPPA-256**. In: Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD), 2017, Campinas. Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD). Porto Alegre: SBC, 2017. v. 1. p. 52-63.
- PODESTA JUNIOR, E. ; PEREIRA, A. D. ; ROCHA, R. C. O. ; CASTRO, M. ; GOES, L. F. W. **Uma Implementação**

do Framework PSkel com Suporte a Aplicações Estêncil Iterativas para o Processador MPPA-256. In: Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS), 2017, Ijuí. Anais da Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS). Porto Alegre: Sociedade Brasileira de Computação, 2017. v. 1. p. 395-398.

- PODESTA JUNIOR, E. ; PEREIRA, A. D. ; PENNA, P. H. ; ROCHA, R. C. O. ; CASTRO, M. ; GOES, L. F. W. **PSkel-MPPA: Uma Adaptação do Framework PSkel para o Processador Manycore MPPA-256**. In: Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS), 2016, São Leopoldo. Anais da Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS). Porto Alegre: Sociedade Brasileira de Computação (SBC), 2016.

1.3 ORGANIZAÇÃO DO TEXTO

O texto deste trabalho será organizado da seguinte forma. O Capítulo 2 descreve a base conceitual utilizada para realizar este trabalho. O Capítulo 3 discute os principais trabalhos relacionados. O Capítulo 4 apresenta a proposta de adaptação do PSkel para o MPPA-256. O Capítulo 5 apresenta a análise experimental dos resultados. Por fim, o Capítulo 6 conclui este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta uma fundamentação teórica básica sobre Computação Paralela sob o ponto de vista arquitetural (Seção 2.1) e de programação (Seção 2.2). Posteriormente, serão apresentados os conceitos fundamentais do padrão *stencil* (Seção 2.3) e sobre o *framework* PSkel (Seção 2.4).

2.1 ARQUITETURAS PARALELAS

De acordo com Tanenbaum *et al.* (TANENBAUM; BOS, 2015), as arquiteturas paralelas podem ser classificadas em dois grandes grupos: multiprocessadores e multicomputadores. Nas seções a seguir são apresentados os principais conceitos básicos destas duas classes de arquiteturas paralelas. Posteriormente, será discutido em mais detalhes o processador *manycore* MPPA-256, o qual será utilizado neste trabalho.

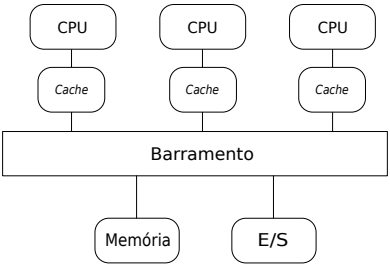
2.1.1 Multiprocessadores

Multiprocessadores são sistemas constituídos de uma ou mais *Central Processing Units* (CPUs) que compartilham totalmente a *Random-Access Memory* (RAM) do sistema. Desta forma, *threads* de um mesmo processo se comunicam através do mesmo espaço de endereçamento, por meio de escrita e leitura na memória. Essa característica do sistema pode ocasionar problemas de concorrência, onde um valor escrito por uma *thread*, localizado em uma palavra na memória, pode ser alterado por outra *thread*, trazendo inconsistência ao sistema.

Mais especificamente, multiprocessadores podem possuir propriedades adicionais, como acesso uniforme à memória. Máquinas com essa propriedade são chamadas de UMA. Por outro lado, existem multiprocessadores que não apresentam essa característica, como é o caso de multiprocessadores NUMA, que apresentam um acesso não-uniforme à memória.

A Figura 3 apresenta a arquitetura de sistemas multiprocessados UMA mais simplificados, onde existem várias CPUs que se comunicam com uma memória compartilhada por meio de um barramento. Quando uma CPU deseja efetuar a comunicação, o barramento é verificado para determinar a sua disponibilidade. Caso o barramento esteja

Figura 3 – Esquema genérico de um multiprocessador UMA.



Fonte: desenvolvido pelo autor.

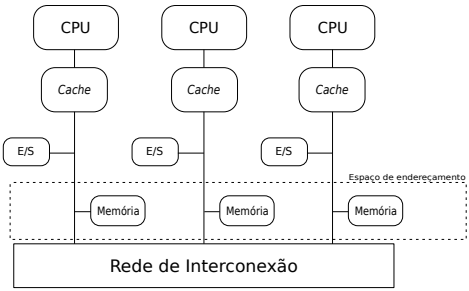
ocupado, a CPU espera até que ele fique livre. Com o barramento livre, a CPU coloca o endereço da palavra no barramento, utiliza sinais de controle e espera até que a memória coloque a palavra desejada no barramento. Esse método é gerenciável para poucas CPUs, contudo com uma maior quantidade, o número de comunicações aumenta significativamente, tornando o gerenciamento de comunicações, por meio do barramento, insuportável. Desta forma, o barramento se torna o gargalo do sistema.

A solução é utilizar *caches* nas CPUs, possibilitando que requisições de leitura sejam satisfeitas pela *cache*, diminuindo a quantidade de comunicações. Desta forma, é possível adicionar mais CPUs no barramento devido à baixa quantidade de tráfego. *Caches* possuem protocolos de coerência para manter a consistência do sistema. Quando uma CPU efetua uma escrita sobre uma palavra, todas as *caches* que possuem essa palavra serão notificadas. Uma *cache* com uma cópia modificada, isto é, diferente do dado presente na memória, deve escrever essa cópia diretamente na memória. Caso uma cópia exata do dado na memória esteja presente na *cache*, ela pode ser descartada, fazendo com que a CPU acesse diretamente a memória.

Além disso, é possível inserir mais níveis de *cache* nas CPUs, diminuindo o tráfego e retirando a necessidade de mais acessos à memória principal. Contudo, devido ao limite arquitetural, uma quantidade muito grande de *caches* é inviável. Além disso, um tamanho muito grande para *caches* torna o seu acesso muito lento, prejudicando o desempenho do sistema.

Sistemas multiprocessados NUMA são diferentes de sistemas UMA, como mencionado anteriormente, devido ao acesso à memória remota ser mais lento que à memória local. A Figura 4 apresenta a arquite-

Figura 4 – Esquema genérico de um multiprocessador NUMA.



Fonte: desenvolvido pelo autor.

tura desse sistema, onde existem nós interconectados por uma rede, e cada nó possui uma CPU e um bloco de memória. A CPU de cada nó pode possuir uma *cache*, denominada *Cache-Coherent Non-Uniform Memory Access* (CC-NUMA), que possibilita uma redução no tempo de acesso a um dado localizado em uma memória remota. Por outro lado, um sistema sem *cache* é denominado *No Cache Non-Uniform Memory Access* (NC-NUMA).

Com o desenvolvimento de novas tecnologias, o tamanho dos transistores diminuiu significativamente, se tornando possível inserir um grande número de transistores em um único *chip*. Com o aumento da quantidade de transistores, um *chip* pode ter, por exemplo, várias CPUs (núcleos), caracterizando um *chip multicore*. Geralmente chamados de *Chip Multiprocessors* (CMPs), os *multicores* são semelhantes aos multiprocessadores tradicionais, contudo, devido à proximidade de conexão entre CPUs, falhas em um componente pode ocasionar problemas em outros componentes do sistema. Esse problema pode ser ainda mais agravado em sistemas do tipo *Multiprocessor System-on-Chip* (MPSoC). Esses sistemas possuem geralmente CPUs (muitas vezes *multicore*) de propósito geral, além de processadores dedicados a atividades bem específicas, tais como decodificadores de áudio e vídeo, processadores criptográficos, entre outros. Além dos processadores, esses sistemas também incluem diferentes tipos de interface de rede e de E/S no mesmo *chip*.

Quando a quantidade de núcleos é grande, isto é, dezenas ou milhares de núcleos, o *chip* pode ser classificado como *manycore*. Contudo, o limite para classificar um *chip* em *manycore* ou *multicore* é flexível (TANENBAUM; BOS, 2015).

Com vários núcleos em um único *chip* problemas de coerência de *cache* começam a surgir. Mais especificamente, com o aumento no número de núcleos, o custo sobre o protocolo de coerência vai crescer até que aumentar o número de núcleos não auxiliará mais o desempenho, pois o sistema estará gastando muito tempo mantendo as *caches* consistentes.

Atualmente, sistemas computacionais comuns apresentam uma GPU, onde temos milhares de núcleos disponíveis. GPUs utilizam esses núcleos, essencialmente, para a solução de cálculos, focando menos em questões de *cache* e lógica de controle. Desta forma, elas são utilizadas em pequenas computações que podem ser paralelizadas. Contudo, a programação para GPUs é difícil, pois os seus núcleos fazem a execução da mesma instrução com diferentes partes do dado, isto é, são máquinas *Single Instruction, Multiple Data* (SIMD). Esse modelo de programação é interessante para abordar o paralelismo de dados, contudo, o desenvolvedor pode se deparar com dificuldades durante o desenvolvimento. Desta forma, linguagens de programação, como CUDA e *Open Computing Language* (OpenCL), abstraem o desenvolvimento de aplicações para esses processadores.

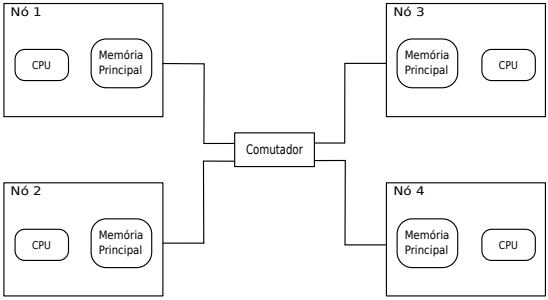
O MPPA-256 é um processador *manycore* diferente dos apresentados acima, pois os seus núcleos são conectados atraves de uma NoC. Mais detalhes sobre o MPPA-256 serão apresentados posteriormente na Seção 2.1.3, tendo em vista que ele possui também características relacionadas a multicomputadores.

2.1.2 Multicomputadores

Multiprocessadores de grande porte são difíceis de construir devido ao alto custo. Desta forma, devido à simplicidade de construção, multicomputadores começaram a surgir. A ideia principal de um multicomputador é agregar em um mesmo sistema diversos computadores, os quais muitas vezes possuem multiprocessadores (TANENBAUM; BOS, 2015). Nesse sentido, um computador com sua placa de interface de rede é considerado como um nó do sistema multicomputado, onde um gerenciamento de forma inteligente da rede auxilia o desempenho do sistema.

A Figura 5 mostra como um sistema multicomputado simples é organizado. Existem nós que possuem uma CPU, memória principal dedicada e uma conexão diretamente com outros nós ou a um comutador. A topologia apresentada na imagem é uma das topologias possíveis em

Figura 5 – Esquema simples de um sistema multicomputado.



Fonte: desenvolvido pelo autor.

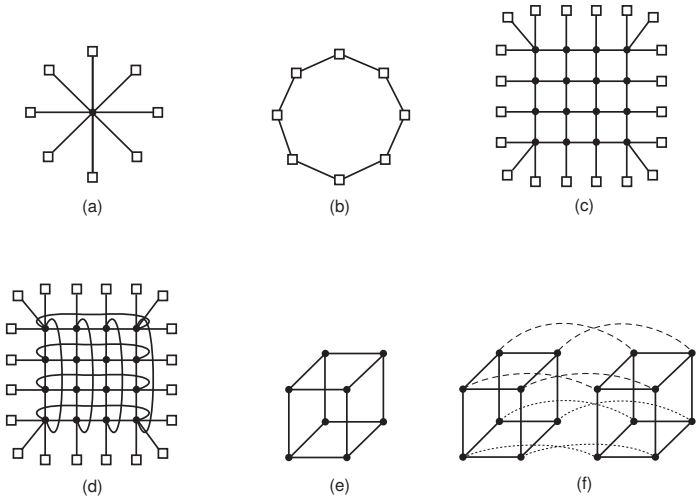
um sistema multicomputado. Outra alternativa é utilizar uma topologia em anel (Figura 6b), retirando a necessidade de um comutador, pois um nó é conectado diretamente aos outros nós. Cada nó é conectado ao nó à sua esquerda e à sua direita.

Uma outra topologia de interconexão muito utilizada em multicomputadores é a malha (*mesh*), a qual é ilustrada pela Figura 6c. Nesse tipo de topologia, os nós são interconectados em comutadores distintos pelo sistema e cada comutador é conectado a outros comutadores, formando uma espécie de malha no sistema. Uma variante dessa topologia é o toro duplo (*torus*) apresentado na Figura 6d, onde os comutadores de cantos opostos estão conectados diretamente. Desta forma, comunicações entre nós de cantos opostos não terão a necessidade de inúmeros saltos pelos comutadores para efetuar a transmissão de informações.

A Figura 6e ilustra uma topologia tridimensional, e a Figura 6f ilustra um cubo tetradimensional. Topologias n dimensionais são utilizadas para diminuir o atraso de comunicação entre nós, pois o diâmetro da rede cresce linearmente de acordo com a dimensionalidade. Mais precisamente, o diâmetro da rede é o número mínimo de nós que o pacote precisa cruzar antes de chegar ao destino mais distante possível. Desta forma, topologias n dimensionais permitem que o diâmetro da rede seja maior, utilizando mais conexões entre nós e reduzindo o atraso de comunicação. Devido a essa propriedade, topologias n dimensionais são utilizadas em vários sistemas.

Geralmente, multicomputadores podem ser construídos com computadores pessoais comuns interconectados por uma interface de rede para trabalharem em conjunto. Organizações de multicomputadores

Figura 6 – Topologias de interconexão.

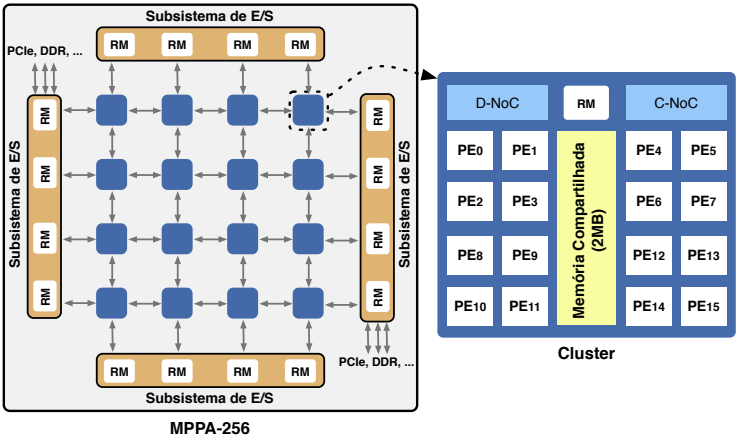


Fonte: (TANENBAUM; BOS, 2015)

deste tipo são denominadas de *Network of Workstations* (NOW). Devido à sua simplicidade, esse sistema não é focado em ganho de desempenho. Por outro lado, *Clusters of Workstations* (COW) são constituídos de computadores interconectados sem teclado, monitor e *mouse* contendo cada um diversos processadores de alto desempenho. Esses sistemas são focados em desempenho, pois possuem redes de interconexão de alto desempenho (alta largura de banda e baixa latência).

A comunicação em um multicomputador é feita por meio de troca de mensagens. Processos localizados em diferentes CPUs se comunicam por meio de mecanismos básicos disponibilizados pelo Sistema Operacional (SO). Esses mecanismos básicos do SO são utilizados como base para implementação de bibliotecas de comunicação que implementam, pelo menos, duas primitivas básicas: *send* e *receive*. A primitiva *send* envia uma mensagem para um processo, o que é determinado pelos parâmetros de entrada da mesma. A primitiva *receive* é responsável pelo recebimento de mensagens, utilizando como parâmetro o endereço que será lido para coletar a mensagem recebida. Portanto, em sistemas multicomputados a troca de mensagens entre processos é realizada de maneira explícita pelo desenvolvedor.

Figura 7 – Visão geral do MPPA-256.



Fonte: (CASTRO et al., 2013)

2.1.3 O Processador *Manycore* MPPA-256

O MPPA-256 é um processador *manycore* desenvolvido pela empresa francesa Kalray, o qual possui 256 núcleos de processamento de 400 MHz. Ele mistura características de um multiprocessador e de um multicomputador, porém em um único *chip*. Mais especificamente, o MPPA-256 utiliza um modelo multicomputado com uma comunicação via NoC em seus *clusters*, e um modelo multiprocessado dentro de cada *cluster*.

Os núcleos de processamento do MPPA-256 são denominados PEs. Além dos PEs, o processador possui 32 núcleos dedicados à gerência de recursos denominados *Resource Manager*s (RMs). PEs e RMs são distribuídos fisicamente no *chip* em 16 *clusters* e 4 subsistemas de E/S, onde cada *cluster* contém 16 PEs e 1 RM. Além dos *clusters*, o MPPA-256 possui 4 subsistemas de E/S contendo, cada um, 4 RMs. Toda a comunicação entre *clusters* e/ou subsistemas de E/S é feita através de uma NoC *torus* 2D. A arquitetura do MPPA-256 pode ser vista na Figura 7[1].

A finalidade principal dos PEs é executar *threads* de usuário de forma ininterrupta e não preemptível para realização de computação. PEs de um mesmo *cluster* compartilham uma memória de 2 MB, a

[1] A figura apresenta uma ilustração simplificada, omitindo a topologia da NoC.

qual é utilizada para armazenar os dados a serem processados pelos PEs. Cada PE possui também uma memória *cache* associativa 2-*way* de 32 KB para dados e uma para instruções. Porém, o processador não dispõe de coerência de *caches*, o que dificulta o desenvolvimento de aplicações para esse processador. Por outro lado, a finalidade dos RMs é gerenciar E/S, controlar comunicações entre *clusters* e/ou subsistemas de E/S e realizar comunicação com uma memória RAM. Na arquitetura utilizada neste trabalho, um dos subsistemas de E/S está conectado a uma memória externa *Low Power Double Data Rate 3* (LPDDR3) de 2 GB.

A comunicação dos *clusters* com o subsistema de E/S e a comunicação entre *clusters* é realizada de maneira explícita, utilizando uma API própria do MPPA-256 de baixo nível similar à POSIX IPC. Detalhes referentes à comunicação e programação nesse processador serão abordadas posteriormente na Seção 2.2.3.

2.2 PROGRAMAÇÃO PARALELA

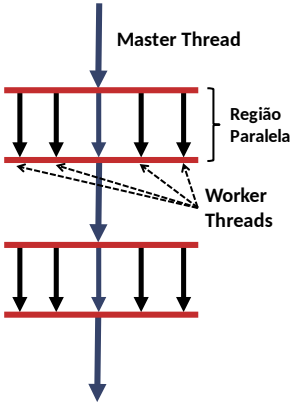
Aplicações são, geralmente, implementadas de forma sequencial, isto é, um conjunto serializado de instruções que será executado sobre uma CPU. Por outro lado, a computação paralela ou distribuída efetua o processamento de instruções sobre múltiplos elementos de processamento. A ideia principal é dividir a computação em partes menores que podem ser executadas simultaneamente entre os elementos de processamento distintos, com intuito de se realizar um processamento em menos tempo.

Diferentes APIs de programação paralela foram criadas com intuito de simplificar o desenvolvimento de aplicações em arquiteturas multiprocessadas e multicomputadas. A seguir serão apresentadas as APIs mais utilizadas no âmbito de HPC em cada tipo de arquitetura. Por fim, será apresentada a API utilizada para o desenvolvimento de aplicações no processador MPPA-256.

2.2.1 OpenMP

Para evitar uma programação de baixo nível sobre um sistema multiprocessado, são utilizadas APIs para o desenvolvimento de aplicações, como o OpenMP. O OpenMP é um modelo de programação baseado em memória compartilhada e provê uma maior facilidade no

Figura 8 – Esquemático do modelo *fork-join*.



Fonte: desenvolvido pelo autor.

desenvolvimento de aplicações para o ambiente multiprocessado, permitindo a paralelização de aplicações com variáveis de ambiente e diretivas de compilação.

O OpenMP utiliza o modelo *fork-join*, onde a execução inicia com uma única *thread*, denominada *master thread*. Como ilustrado na Figura 8, quando a *master thread* encontra uma região paralela, são criadas outras *threads* de acordo com a variável de ambiente especificada no sistema. No final da região paralela, é realizado um *join* por meio de uma barreira implícita, onde as *threads* serão sincronizadas, e a execução irá continuar apenas com a *master thread*.

Por meio de diretivas de compilação é possível definir o comportamento do OpenMP, inclusive, determinar regiões paralelas e outras funções da API. O Código 1 apresenta um exemplo de uma função que implementa o produto escalar entre dois vetores (`a` e `b`) paralelizada com uso do OpenMP. A região paralela é criada pela diretiva de compilação `#pragma omp parallel` (linha 5). Variáveis dentro de uma região paralela podem ser classificadas como `shared` (compartilhadas) ou `private` (privadas), possibilitando assim o gerenciamento da execução pela API. Uma variável marcada como `shared` é compartilhada entre as *threads* de uma região paralela. Por outro lado, variáveis marcadas como `private` são privadas para cada *thread*, isto é, cada *thread* possuirá uma cópia privada da variável. Desta forma, as modificações sobre elas serão feitas localmente em cada *thread*.

Código 1 – Produto escalar paralelo com OpenMP.

```
1 int produto_escalar(int *a, int *b, int tamanho)
2 {
3   int prod = 0;
4
5   #pragma omp parallel for private(i) shared(a, b) \
6                                       reduction(+:prod)
7   for (int i = 0; i < tamanho; i++)
8     prod += a[i] * b[i];
9
10  return(prod);
11 }
```

As diretivas do OpenMP, além de determinar regiões paralelas, possibilitam paralelizar laços de maneira automática, onde as iterações do laço são distribuídas, de forma automática e flexível, sobre as *threads* da região paralela. A paralelização das iterações do laço é mostrada na linha 6 do Código 1 com uso da cláusula `for`. Além disso, operações de redução podem ser utilizadas ao fim de uma região paralela, aplicando sobre uma variável a operação especificada. Ao final da região paralela, o resultado é atribuído a uma variável compartilhada. A redução é necessária no caso do produto escalar mostrado no Código 1, sendo necessário utilizar a cláusula `reduction` sobre a variável `prod`. A operação de redução utilizada no caso do produto escalar é a soma (`+`).

Devido à abstração que o modelo oferece, poucas modificações de código são necessárias para paralelizar uma aplicação. POSIX *threads* é outro modelo que pode ser utilizado, contudo com uma menor abstração que o modelo OpenMP.

2.2.2 MPI

A programação paralela em multicomputadores é feita através da utilização de múltiplos processos que se comunicam através de trocas de mensagens. Devido à característica de baixo nível intrínseca do modelo de troca de mensagens, utilizar *sockets* manualmente para a comunicação entre nós de uma rede de *clusters* é inadequada para o desenvolvedor. Portanto, mostra-se necessário utilizar APIs de mais alto nível, possibilitando uma maior abstração ao desenvolvedor.

O *Message Passing Interface* (MPI) é uma API utilizada amplamente em multicomputadores fornecendo uma maior abstração em

relação à programação sobre *sockets*. A API é baseada no modelo *Single Program, Multiple Data* (SPMD), onde todos os processos executam o mesmo programa, porém cada processo é responsável por realizar computações em dados distintos.

A API fornece diversas funções aos desenvolvedores. A função `MPI_Init()` permite inicializar o ambiente MPI. Após a inicialização, cada processo MPI possuirá um identificador único (de 0 até $np - 1$, onde np é o número total de processos MPI em execução). Esse identificador único, denominado *rank*, poderá ser utilizado em conjunto com instruções de seleção (`if-else`) para determinar que processos MPI distintos possam executar códigos distintos. Além disso, ele será utilizado nas primitivas de comunicação do MPI para especificar os remetentes e destinatários das mensagens. Para finalizar o ambiente MPI é feita uma chamada para a função `MPI_Finalize()`. O envio de mensagens é realizado pela função `MPI_Send()`, que construirá a mensagem, e irá adicionar o *rank* do destinatário, o *rank* do remetente, entre outras informações. A função `MPI_Recv()` será responsável por receber a mensagem enviada pelo processo, e irá armazená-la no espaço de memória do processo destinatário.

O Código 2 mostra um exemplo de um programa em MPI. Nesse exemplo, o processo com *rank* igual a zero envia uma mensagem a todos os demais processos. Ao receber a mensagem, cada um dos demais processos imprime na tela a mensagem recebida. As funções `MPI_Comm_rank()` e `MPI_Comm_size()` são utilizadas para armazenar nas variáveis `rank` e `size` o *rank* do processo e o número total de processos, respectivamente. Nas linhas 12-13, o processo com *rank* igual a zero envia para os demais processos a mensagem `“Ola mundo!”`. As linhas 16-19 são executadas somente pelos demais processos, onde cada processo realiza o recebimento da mensagem e a imprime na tela juntamente com seu *rank*.

Além de comunicações do tipo ponto-a-ponto, existem comunicações coletivas e de sincronização entre todos os processos. A função `MPI_Barrier()` é uma barreira de sincronização, responsável por bloquear a execução de um processo até que todos os outros processos cheguem na barreira. Por outro lado, a função `MPI_Bcast()` é responsável por enviar a mesma mensagem de um processo para todos os outros processos do sistema de forma otimizada.

Código 2 – Exemplo de um programa MPI.

```
 1 int main(int argc, char **argv)
 2 {
 3   int rank, size;
 4
 5   MPI_Init(argc, argv);
 6
 7   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 8   MPI_Comm_size(MPI_COMM_WORLD, &size);
 9
10   if (rank == 0) {
11     char mensagem[11] = "Ola mundo!";
12     for (int i = 1; i < size; i++)
13       MPI_Send(&mensagem, 11, MPI_CHAR, i, 0, MPI_COMM_WORLD);
14   }
15   else {
16     char mensagem[11];
17     MPI_Recv(&mensagem, 11, MPI_CHAR, 0, MPI_ANY_TAG,
18                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
19     printf("Processo %d recebeu: %s\n", rank, mensagem);
20   }
21
22   MPI_Finalize();
23   return(0);
24 }
```

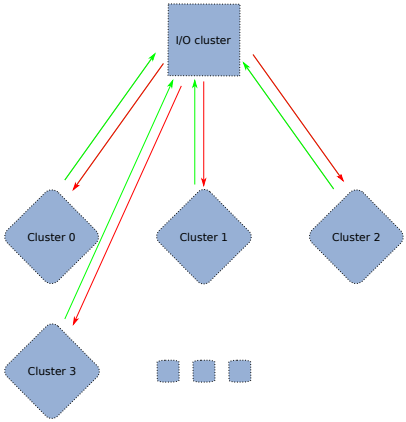
2.2.3 Modelo de Programação do MPPA-256

O MPPA-256 possui uma arquitetura interessante que permite a execução de aplicações paralelas que seguem um modelo mestre/trabalhador. Nesse modelo, o processo mestre é responsável pela coordenação e pela divisão das tarefas entre os processos trabalhadores. Os processos trabalhadores, por outro lado, são responsáveis por receber tarefas e computá-las, devolvendo os resultados para o processo mestre. No MPPA-256, o processo mestre é executado em um RM no subsistema de E/S, e é responsável por inicializar os processos trabalhadores. Cada processo trabalhador é executado em um *cluster* distinto, podendo criar até 16 *threads* POSIX, uma para cada PE.

A Figura 9 ilustra o funcionamento do modelo mestre/trabalhador no MPPA-256. O processo mestre será responsável por iniciar os *clusters* por meio da função `MPPA_Spawn()`. O MPPA-256 não possui suporte para o MPI, desta forma, os processos mestre e trabalhadores utilizam objetos de comunicação, como portais e filas, de uma API proprietária de baixo nível do MPPA-256, similar à POSIX IPC.

Cada processo trabalhador possui um espaço de endereçamento distinto, desta forma, são utilizados portais para efetuar a comunicação

Figura 9 – Esquemático do modelo mestre/trabalhador no MPPA-256.



Fonte: Manual do MPPA-256.

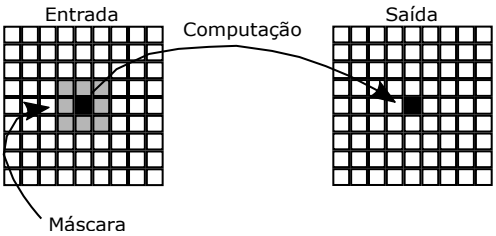
entre o mestre e os trabalhadores, e entre trabalhadores. Os portais efetuam escrita e leitura remota, onde um processo deve criar um portal com uma classificação, denominada *tag*, e relacionar o portal com um espaço de endereçamento para realizar a comunicação.

Para efetuar a escrita em um espaço de endereçamento é necessário chamar uma função denominada `mppa_pwrite()`. Essa função possui como parâmetros o portal que será utilizado na comunicação e o tamanho do dado que será enviado. A função `mppa_aio_read()` realizará a leitura dos dados recebidos pelo portal, relacionando o portal responsável pela leitura com o espaço de endereçamento em que o dado será armazenado.

Um programa para o MPPA-256 é composto por, pelo menos, dois arquivos principais: um deles conterá o código a ser executado pelo processo mestre e outro conterá o código a ser executado por cada processo trabalhador. Esses arquivos são compilados separadamente, gerando dois arquivos binários (um para o processo mestre e outro para os processos trabalhadores). O binário dos trabalhadores é utilizado como argumento de entrada da função `MPPA_Spawn()` descrita anteriormente durante a criação dos processos trabalhadores.

Trabalhos anteriores mostraram que desenvolver aplicações paralelas otimizadas para o MPPA-256 é um grande desafio (FRANCESQUINI et al., 2014) devido a alguns fatores importantes. O primeiro deles está relacionado ao **modelo de programação híbrido** exigido pelo pro-

Figura 10 – Ilustração do padrão *stencil*.



Fonte: desenvolvido pelo autor.

cessador: *threads* em um mesmo *cluster* se comunicam através de uma memória compartilhada local, porém a comunicação entre *clusters* é feita explicitamente via NoC, em um modelo de memória distribuída. Mais especificamente, aplicações desenvolvidas para o MPPA-256 precisam utilizar duas bibliotecas de programação paralela para utilizar os recursos do processador: OpenMP, baseado em um modelo de memória compartilhada, utilizada para paralelizar a computação dentro de cada *cluster* e a API proprietária do MPPA-256, que segue um modelo de memória distribuída, sendo utilizado na comunicação entre os *clusters* e o subsistema de E/S por meio da NoC. O segundo fator importante está relacionado à **capacidade limitada de memória** no *chip*: cada *cluster* possui apenas 2 MB de memória local de baixa latência. Portanto, aplicações reais precisam constantemente realizar comunicações com o subsistema de E/S conectado à memória LPDDR3. Por fim, o último fator está diretamente relacionado à **ausência de coerência de *cache***: cada PE possui uma memória *cache* privada sem coerência de *cache*, sendo necessário o uso explícito de instruções do tipo *flush* para atualizar a *cache* de um PE quando necessário.

2.3 O PADRÃO *STENCIL*

As dificuldades na computação paralela proporcionam um grande impacto no desenvolvimento de aplicações. Com o desenvolvimento de aplicações paralelas, começou-se a notar um padrão entre elas. Com isso, foram criados os padrões paralelos para simplificar o desenvolvimento de código. Para uma maior abstração e redução da complexidade dos padrões, foram propostos esqueletos paralelos. Na programação paralela com esqueletos, o esqueleto é responsável por gerenciar o controle de tarefas e dados, retirando essa responsabilidade do desenvolvedor.

Código 3 – Exemplo de código *stencil* (aplicação Jacobi).

```
1 void jacobi(int tsteps, int N, float *A, float *B){
2   int t, i, j;
3   float c1 = 0.2;
4
5   for (t = 0; t < tsteps; t++){
6     for (i = 1; i < N-1; i++)
7       for (j = 1; j < N-1; j++)
8         B[i,j] = c1 * (A[i,j] + A[i,j-1] + A[i,j+1] + A[i+1,j]
9                        + A[i-1,j]);
10
11    for (i = 1; i < N-1; i++)
12      for (j = 1; j < N-1; j++)
13        A[i,j] = B[i,j]
14  }
15 }
```

Desta forma, é possível simplificar grande parte do desenvolvimento de aplicações paralelas e auxiliar em outras funções que podem trazer uma maior dificuldade ao desenvolvedor. Mais especificamente, o desenvolvedor irá focar apenas em especificar o algoritmo, deixando o esqueleto gerenciar os detalhes de execução, diminuindo o tempo de desenvolvimento e *debug* da aplicação.

Existem diversos padrões paralelos, como o *map*, *reduce*, *scan*, *stencil*, entre outros. Dentre os padrões existentes, o padrão *stencil* é de grande importância tanto no ambiente acadêmico quanto no industrial, utilizado em diversos campos importantes, como física quântica, previsão do tempo e processamento de imagens (PEREIRA; RAMOS; GÓES, 2015).

O padrão *stencil* atualiza elementos de uma matriz de entrada (A), de acordo com um padrão especificado. Mais especificamente, em uma aplicação *stencil*, cada iteração utiliza a máscara de vizinhança responsável por determinar os vizinhos utilizados na computação. A máscara é aplicada sobre A para determinar o valor de cada célula da matriz de saída (B). No exemplo da Figura 10, o valor de cada célula da matriz de saída é determinado em função dos valores de cada uma das células vizinhas adjacentes da matriz de entrada. Esse processo é realizado para todas as células da matriz de entrada, produzindo uma matriz de saída contendo o resultado da computação *stencil*. Além disso, o padrão possibilita a computação iterativa, isto é, ao final de uma iteração, a matriz de saída será considerada como a matriz de entrada para a próxima iteração, caracterizando uma nova iteração da computação.

O Código 3 apresenta um exemplo do código de uma aplicação

baseada no padrão *stencil*, cujo o objetivo é a realização da resolução de equações matriciais pelo método iterativo de Jacobi. O número de iterações é determinado pelo parâmetro `tsteps` (linha 5). A computação *stencil* é realizada lendo-se as informações da matriz de entrada `A` e escrevendo-se os resultados em uma matriz de saída `B`, ambas de tamanho `N*N`. Nesse exemplo, foi utilizado um coeficiente `c1` e uma vizinhança de 5 elementos (linha 8). A vizinhança de 5 elementos representa a célula central e as 4 células adjacentes à célula central (cima, baixo, esquerda e direita). Devido à característica iterativa desta aplicação, existe uma troca de dados entre as matrizes `A` e `B` (linhas 10-12) para que o resultado da iteração `i` possa ser utilizado como entrada na iteração `i+1`.

2.4 PSKEL

O PSkel é um *framework* de programação em alto nível para o padrão *stencil*, que oferece suporte a execuções paralelas em arquiteturas heterogêneas incluindo CPU e GPU. Utilizando uma única interface de programação escrita em C++, o usuário é responsável por definir o *kernel* principal da computação *stencil*, enquanto o *framework* se encarrega de gerar código executável para as diferentes plataformas paralelas e realizar todo o gerenciamento de memória e transferência de dados entre dispositivos de forma transparente (PEREIRA; RAMOS; GÓES, 2015). Mais especificamente, o PSkel traduz as abstrações em código C++ de baixo nível, compatível com Intel TBB e NVIDIA CUDA.

A API do PSkel possibilita a definição de *templates* para a manipulação de estruturas *n*-dimensionais, denominadas `Array` (1 dimensão), `Array2D` (2 dimensões) e `Array3D` (3 dimensões). Além disso, o *framework* provê abstrações para a definição da vizinhança do *stencil* (máscara) e o *kernel* da computação *stencil* (`stencilKernel()`). O `stencilKernel()` é um método a ser implementado pelo usuário que descreve, especificamente, a computação que será executada para cada célula do `Array` de entrada com base nos valores de sua vizinhança (máscara).

Desta forma, o desenvolvedor deverá seguir os seguintes passos para desenvolver uma aplicação *stencil* com PSkel:

1. Identificar a dimensão do problema, construindo estruturas de acordo com os recipientes especificados pelo *framework*;

2. Definir o método `stencilKernel()` que descreve a computação

Código 4 – Exemplo do código da aplicação Jacobi no PSkel.

```
1 __parallel__ void
2 stencilKernel(Array2D<float> A, Array2D<float> B, Mask2D<int> mask,
3               struct Arguments args, int x, int y){
4   B(x,y) = args.alpha * (A(x,y) + A(x,y+1) + A(x,y-1) + A(x+1,y)
5                          + A(x-1,y));
6 }
7
8 int main(int argc, char **argv) {
9   /* declaracoes de variaveis omitidas */
10
11   Array2D<float> input(A, M, N);
12   Array2D<float> output(B, M, N);
13   int neighbors = {{0,1}, {-1,0}, {1,0}, {-1,0}};
14   Mask2D<int> mask(4, neighbors);
15   struct Arguments args(alpha);
16
17   Stencil2D<Array2D<float>, Mask2D<int>, Arguments>
18     jacobi(A, B, args);
19   jacobi.runIterative(device::GPU, tsteps, 1.0);
20
21   return(0);
22 }
```

executada sobre os elementos da máscara e do `Array` de entrada;

3. Instanciar um ou mais objetos `Stencil` para gerenciar os encapsulamentos, alocação de memória e chamadas para efetuar a computação determinada pelo método `stencilKernel()`. Mais especificamente, os recipientes são estruturas que armazenam `Arrays` para leitura/escrita de dados. Eles são responsáveis por gerenciar a alocação de memória na CPU e GPU de maneira transparente.

4. Instanciar a classe de *Runtime* que adota um padrão *Facade* que efetua a abstração dos detalhes da implementação e configurações do padrão *stencil*. Essa classe provê os métodos de execução para os padrões *stencil*, além do particionamento transparente de tarefas e dados entre CPU e GPU.

Em uma aplicação *stencil* iterativa, cada iteração utiliza a máscara de vizinhança sobre o `Array` de entrada para determinar o valor de cada célula do `Array` de saída. No exemplo da Figura 10, o valor de cada célula do `Array` de saída é determinado em função dos valores de cada uma das células vizinhas adjacentes. Esse processo é realizado para todas as células do `Array` de entrada, produzindo um `Array` de saída da computação *stencil*. Ao final de uma iteração, o `Array` de saída será considerado como `Array` de entrada para a próxima iteração no caso de uma aplicação *stencil* iterativa.

O Código 4 apresenta um exemplo da aplicação Jacobi discutida na Seção 2.3 (Código 3), porém agora implementada no *framework* PSkel. Nesse exemplo, a função *stencil* principal da aplicação está implementada no método `stencilKernel()` nas linhas 1-5. As estruturas para efetuar a computação, como o `Array` de entrada (`input`) e de saída (`output`), são mostrados nas linhas 10-11. O formato da vizinhança é especificado na linha 12, sendo guardado na variável `neighbors`. Então, a máscara é construída na linha 13 com base na vizinhança definida anteriormente. Estruturas como `Array2D`, `Mask2D`, são exemplos dos recipientes disponibilizados pelo *framework*. A classe de *runtime* é determinada pela estrutura `Stencil2D` (linha 16), onde ao efetuar a chamada para função `runIterative()` (linha 18), a execução da computação irá iniciar.

É possível notar que o *kernel* da computação *stencil* (linha 4) fica mais simplificado em relação ao código original do Jacobi mostrado anteriormente (Código 3), pois as iterações da aplicação (`tsteps`) e os laços de computação sobre as matrizes ficam implícitos, sendo gerenciados pelo *framework*. Os elementos das matrizes são determinados pelos parâmetros `x` e `y` da função `stencilKernel()`. Além disso, o coeficiente da aplicação Jacobi é passado como parâmetro por uma `struct` denominada `Arguments` (linha 14).

3 TRABALHOS RELACIONADOS

A proposta deste trabalho está diretamente relacionada a diversos outros trabalhos de pesquisa. A seguir, serão citados alguns trabalhos de pesquisa que fazem uso de esqueletos paralelos em arquiteturas *multicore* e *manycore*. Além disso, serão destacados alguns trabalhos de pesquisa sobre o MPPA-256.

3.1 ESQUELETOS PARALELOS E PADRÃO *STENCIL*

Buono et al. (BUONO et al., 2013) portaram um *framework* baseado em esqueletos paralelos, chamado de *FastFlow*, para o processador *manycore TilePro64*. Esse processador possui 64 núcleos de processamento idênticos, interconectados por uma NoC no formato de malha. O *framework FastFlow* provê padrões de *design* customizáveis, como, por exemplo, *pipelines* e *task farms*, que podem ser compostas para formar outros esqueletos, como *map* e *reduce*.

De forma similar, *Thorarensen et al.* (THORARENSEN et al., 2016) apresentaram um novo *back-end* do *framework SkePU* para o processador *manycore* de baixa potência *Myriad2*. Esse processador possui uma arquitetura heterogênea, tendo como alvo dispositivos com limites em questão de energia. O *framework SkePU* provê uma interface de programação para esqueletos paralelos como o *map*, *reduce*, e *stencil*, com suporte para diferentes *back-ends*, incluindo processadores *multicore* e GPUs.

Lutz et al. (LUTZ; FENSCH; COLE, 2013) utilizaram técnicas de *tiling* em computações *stencil* para lidar com a capacidade limitada de memória de GPUs em ambientes multi-GPU, utilizando as memórias das GPUs coletivamente. De forma similar, *Gysi et al.* (GYSI; GROSSER; HOEFLER, 2015) propuseram um *framework* para otimizações automáticas de *tiling* em computações *stencil* situadas em um ambiente híbrido CPU-GPU.

3.2 PROCESSADORES *MANYCORE* DE BAIXO CONSUMO DE ENERGIA

Alguns trabalhos surgiram recentemente com intuito de avaliar o uso de processadores *manycore* em HPC, além de discutir os desafios

do desenvolvimento de aplicações para esses processadores. Em (TOTONI et al., 2012), os autores compararam o desempenho e o consumo energético de um processador *manycore* experimental da Intel denominado *Single-Chip Cloud Computer* (SCC) com outros tipos de processadores e GPUs. Para realizar essa análise, os autores utilizaram um conjunto de aplicações paralelas implementadas em Charm++ (KALE; BHATELE, 2013). Os resultados obtidos com o Intel SCC mostraram que *manycores* são uma alternativa viável, apresentando bom desempenho e baixo consumo energético. Em (SIRDEY et al., 2013), os autores avaliaram o desempenho do processador *manycore* MPPA-256 no contexto de aplicações de decodificação de vídeo. Os resultados mostraram que o desempenho do MPPA-256 é comparável ao desempenho de processadores Intel atuais em uma decodificação de vídeo no padrão H.264, consumindo 6 vezes menos energia.

Trabalhos recentes revelaram o desempenho e consumo energético do processador MPPA-256, comparando-o a outros processadores *multicore* de propósito geral e embarcados, no contexto de diferentes aplicações científicas (CASTRO et al., 2014, 2013; FRANCESQUINI et al., 2014). Os resultados mostraram que o processador *manycore* MPPA-256 apresenta em alguns casos desempenho superior a processadores *multicore* Intel Xeon 2.4 GHz com 8 núcleos, além de uma redução no consumo de energia em até 13 vezes em relação ao mesmo processador. Um outro trabalho recentemente publicado realizou uma análise comparativa de desempenho e consumo de energia entre processadores *multicore* Intel de alto desempenho e ARM (PADOIN et al., 2015). Os resultados mostraram que, apesar da potência dos processadores ARM ser pelo menos 10 vezes menor que a dos processadores Intel de alto desempenho, o consumo de energia nem sempre será melhor, sendo dependente das características da carga de trabalho a ser executada.

Morari et al. (MORARI et al., 2012) propuseram uma implementação otimizada do *radix sort* para o processador *manycore* Tilera TILEPro64. Os resultados mostraram que a solução para o TILEPro64 provê uma melhor eficiência energética em relação a um processador *multicore* de propósito geral, como o Intel Xeon W5590, e em relação a uma GPU NVIDIA Tesla C2070.

Mais especificamente, *Francesquini et al.* (FRANCESQUINI et al., 2014) analisaram três diferentes classes de aplicações (CPU-*bound*[1],

[1] O desempenho da aplicação é limitado pela velocidade da CPU.

memory-bound[2] e híbrida[3].) usando plataformas paralelas, como o MPPA-256, e um multiprocessador NUMA de 192 núcleos e 24 nós. Mostrou-se que arquiteturas *manycore* podem ser muito competitivas, mesmo se a aplicação é, naturalmente, irregular. Os resultados mostraram que o MPPA-256 pode obter um desempenho maior (e um consumo de energia menor) que um processador *multicore* de propósito geral (Intel Xeon E5-4640) em um ambiente com cargas de trabalho variadas e CPU-*bound*. Todavia, em um ambiente com cargas de trabalho *memory-bound*, o sistema NUMA obteve um melhor desempenho em relação ao MPPA-256, apesar de apresentar também um maior consumo de energia. Entre as plataformas avaliadas, o MPPA-256 apresentou a melhor eficiência energética, reduzindo a energia consumida em aplicações CPU-*bound*, híbridas e *memory-bound* em 6.9x, 6.5x e 3.8x, respectivamente.

3.3 DISCUSSÃO

Similar ao *framework SkePU*, o PSkel fornece suporte à aplicações paralelas baseadas no padrão estêncil em ambientes heterogêneos CPU-GPU. Contudo, o PSkel não oferece suporte à processadores *manycore*. Além disso, com base nos estudos sobre processadores *manycore*, pode-se perceber que o processador MPPA-256 possui um consumo energético superior em relação aos processadores *multicores*. O MPPA-256 apresenta dificuldades que podem ser solucionadas com técnicas de *tiling*, como mostrado em (LUTZ; FENSCH; COLE, 2013). Desta forma, a adaptação do *framework* PSkel para um processador de baixo consumo de energia e alto desempenho, como o MPPA-256, é interessante para caracterizar um novo estudo sobre a utilização de *frameworks* em ambientes heterogêneos. Por fim, estudos sobre técnicas de *tiling* se tornam importantes na proposta apresentada, devido à capacidade limitada de memória no MPPA-256.

[2] O desempenho da aplicação é limitado pela quantidade de memória disponível e velocidade de acesso à memória.

[3] Aplicação com características CPU-*bound* e *memory-bound*

4 PSKEL-MPPA

Esta seção apresenta as ideias fundamentais da proposta e implementação da adaptação do *framework* PSkel para o processador MPPA-256.

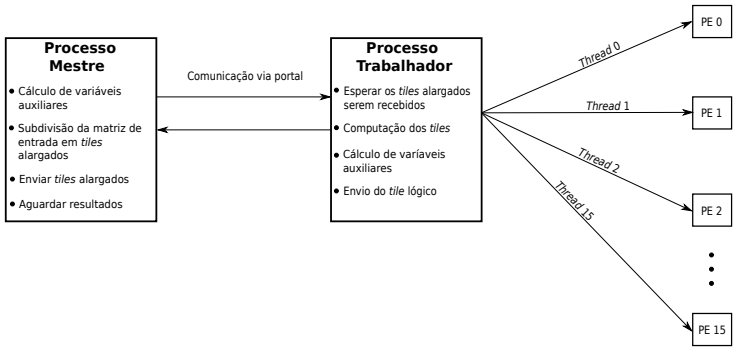
4.1 VISÃO GERAL

Como dito anteriormente, diversas dificuldades prejudicam o desenvolvimento de aplicações para processadores *manycore*, tais como o MPPA-256. Neste projeto, será dado um enfoque para uma classe de aplicações paralelas que seguem o padrão *stencil*. Nesse sentido, adaptar o *framework* PSkel para esse processador trará benefícios claros, simplificando o desenvolvimento de aplicações *stencil* para o MPPA-256. O *framework* fornecerá uma transparência no particionamento de tarefas e dados para esse ambiente, liberando o desenvolvedor da utilização de meios de comunicação via NoC. Além disso, aplicações já desenvolvidas para o *framework* poderão ser executadas no MPPA-256 sem a necessidade de nenhuma alteração em seus códigos originais.

A Figura 11 ilustra, de forma simplificada, a adaptação desenvolvida. A nova adaptação do *framework* suporta matrizes 2D, adotando o modelo mestre-trabalhador descrito na Seção 2.2.3. O processo mestre é executado no subsistema de E/S conectado à uma memória LPDDR3, alocando dados de entrada e saída. Além disso, o processo mestre irá enviar dados para os processos trabalhadores e aguardará os resultados da computação. Por outro lado, cada processo trabalhador é executado em um *cluster* com o objetivo de realizar a computação *stencil* de forma paralela. A computação será subdividida dentro dos *clusters* por meio da biblioteca OpenMP, fazendo com que cada PE realize uma parte da computação. Após a computação ser realizada, o resultado é enviado ao processo mestre. Por fim, toda a comunicação entre os processos mestre e trabalhadores é realizada por meio de portais, como especificado na Seção 2.2.3.

Devido às limitações de memória no MPPA-256, o mestre deverá enviar para o trabalhador pequenas partições da matriz de entrada, denominadas de *tiles*. Durante o processo de subdivisão, cada *tile* deverá considerar as dependências de vizinhança intrínsecas do padrão *stencil*. Mais precisamente, a computação realizada sobre um elemento de um certo *tile* poderá possuir uma relação de dependência com outros *tiles*

Figura 11 – Esquemático ilustrando a implementação mais aprofundada da proposta.



devido à máscara da computação.

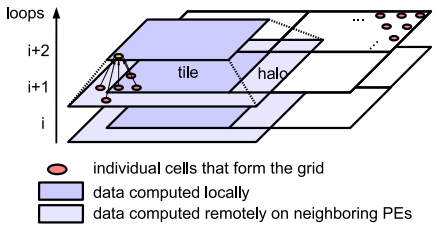
4.2 TILING TRAPEZOIDAL

Para tratar os problemas de dependência foi utilizada a técnica de *tiling* trapezoidal (MENG; SKADRON, 2011) na solução proposta, resultando em redundância de dados e computações (ROCHA et al., 2017). Mais especificamente, *tiling* é geralmente utilizado para particionar a computação de uma aplicação *stencil* em partes menores (*tiles*) entre elementos de processamento[1]. Essa subdivisão tem como objetivo possibilitar a execução paralela da aplicação.

Contudo, o processo de *tiling* introduz problemas de dependência, pois, para computar os elementos das bordas de um *tile*, é necessário obter os valores resultantes em outros elementos de processamento. A Figura 12 ilustra esse comportamento. As regiões que precisam ser resolvidas por sincronizações e comunicações entre elementos de processamento é denominada região *halo*. Dependendo do número de sincronizações e comunicações realizadas, a aplicação pode demorar um tempo considerável realizando o particionamento de dados entre elementos de processamento, prejudicando o desempenho da aplicação.

[1] Nesta seção, esse termo é utilizado para representar elementos de processamento em geral, não, necessariamente, os elementos presentes dentro dos *clusters* no MPPA-256.

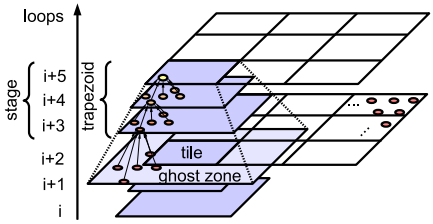
Figura 12 – Esquemático ilustrando a região *halo*.



Fonte: (MENG; SKADRON, 2011).

Para contornar essa sobrecarga após cada iteração, um *tile* pode ser alargado para incluir uma *ghost zone*. A Figura 13 ilustra essa alternativa, onde a *ghost zone* alarga o *tile*, fazendo-o sobrepor *tiles* vizinhos por meio de múltiplas regiões *halo*. Desta forma, elementos de processamento podem gerar mais sobreposições com quantidade proporcional ao número de iterações da aplicação. Além disso, a mesma figura ilustra o comportamento da *ghost zone*, onde ela agrupa as iterações em estágios. Cada estágio realiza operações sobre *tiles* sobrepostos, denominados trapezóides. Por fim, os trapezóides irão produzir um dado sem sobreposição ao final da computação de todas as iterações.

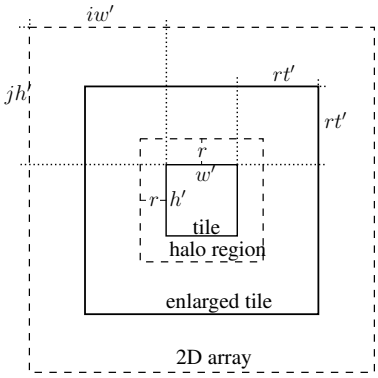
Figura 13 – Esquemático ilustrando a *ghost zone*.



Fonte: (MENG; SKADRON, 2011).

A seguir, a técnica será ilustrada mais profundamente por meio de definições formais. Seja A uma matriz de dados 2D, com dimensões

Figura 14 – *tiling* 2D.



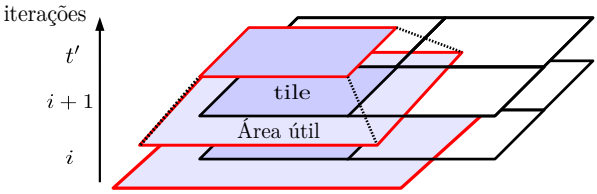
Fonte: (ROCHA et al., 2017)

$\dim(A) = (h, w)$, onde w e h são largura e altura, respectivamente. Utilizando *tiles* de dimensões (w', h'), é possível obter $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ *tiles* sobre A. Seja $A_{i,j}$ um *tile* seguindo as definições descritas, onde $0 \leq i < \lceil \frac{w}{w'} \rceil$ e $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ possui um deslocamento (iw', jh') relativo ao canto superior esquerdo de A e $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. O deslocamento é um índice necessário para acessar os elementos de um *tile*.

A Figura 14 ilustra a técnica de *tiling* trapezoidal. Um *tile* lógico (linha sólida interna) é contido em uma matriz de dados 2D (linha pontilhada externa) com deslocamentos verticais e horizontais dados por jh' e iw'. Se t iterações de uma aplicação *stencil* precisam ser realizadas, é possível computar t' iterações consecutivas sobre $A_{i,j}$ ($t' \in [1, t]$) sem a necessidade de nenhuma troca de dados entre *tiles* adjacentes, isto é, t' iterações internas. Para realizar iterações consecutivas é necessário utilizar um *tile* lógico ($A_{i,j}$) alargado com uma *ghost zone* (área entre a linha sólida interna e a linha sólida externa) que possui uma região *halo* (área entre a linha sólida interna e a linha pontilhada interna). Seja r o maior deslocamento necessário sobre a vizinhança de um elemento, determinado pela máscara *stencil*. A área com alcance r que contém a vizinhança é denominada região *halo*. O número de regiões *halo* que compõem a *ghost zone* é proporcional à t'. Desta forma, o *tile* alargado $A^*_{i,j}$ possui um deslocamento $(\max\{iw' - rt', 0\}, \max\{jh' - rt', 0\})$ relativo à A.

Após a computação das t' iterações consecutivas, sincronizações devem ser realizadas para resolver as iterações restantes (iterações ex-

Figura 15 – Propagação do erro sobre o *tile* alargado.



Fonte: Desenvolvido pelo autor.

ternas). Desta forma, $\lceil \frac{t}{t'} \rceil$ sincronizações serão responsáveis por atualizar a matriz de entrada de forma global.

Devido às dependências de vizinhança presentes na computação *stencil*, cada iteração sobre o *tile* alargado adiciona valores desatualizados na *ghost zone*. A Figura 15 ilustra como que os valores desatualizados se propagam sobre o *tile* alargado. Ao realizar as iterações, a área útil da *ghost zone* diminui proporcionalmente ao número de iterações realizadas. Após t' iterações consecutivas, o *tile* lógico será a região sem erros e sem valores redundantes. Desta forma, quanto maior for a quantidade de iterações consecutivas, a *ghost zone* será maior e, consequentemente, maior a quantidade de computações redundantes para computar corretamente o *tile* lógico. Portanto, existe uma troca entre o custo de computação redundante e o número de comunicações e sincronizações realizadas.

4.3 IMPLEMENTAÇÃO

O processo de execução de uma aplicação no PSkel-MPPA segue a descrição na Seção 2.2.3. A seguir serão apresentadas as funções de cada processo e como foi realizada a comunicação no MPPA-256. O código da implementação é aberto e pode ser acessado pelo seguinte link: `https://github.com/pskel/pskel/tree/mppaWidth`. Por fim, o anexo A ilustra como o desenvolvedor deverá utilizar as estruturas e funções do *framework* PSkel para implementar uma aplicação paralela

Código 5 – Exemplo do código da aplicação Jacobi no processo mestre.

```
1
2 int main(int argc, char **argv) {
3   /* declaracoes de variaveis omitidas */
4
5   Array2D<float> input(A, M, N);
6   Array2D<float> output(B, M, N);
7   int neighbors = {{0,1}, {-1,0}, {1,0}, {-1,0}};
8   Mask2D<int> mask(4, neighbors);
9   struct Arguments args(alpha);
10
11  Stencil2D<Array2D<float>, Mask2D<int>, Arguments>
12    jacobi(A, B, args);
13  jacobi.scheduleMPPA(slave_bin, threadsNum, clustersNum, tileDim,
14                      iterations);
15
16  return(0);
17 }
```

no MPPA-256.

4.3.1 Processo Mestre

O processo mestre, executando sobre o subsistema de E/S, será responsável por alocar na memória LPDDR3 os seguintes dados: (i) os dados de entrada e saída; (ii) a máscara do *stencil* a ser utilizada para a computação; e (iii) a estrutura *stencil*. O código do processo mestre é idêntico ao código da versão original do PSkel, contudo o processo mestre não é responsável pela descrição do *kernel* da computação. O Código 5 mostra a definição do processo mestre no MPPA-256. Na adaptação, a função `scheduleMPPA()` foi implementada para realizar o controle de dados e a computação sobre o processador. Ela é encapsulada pela classe `Stencil2D`, sendo necessário realizar a passagem de parâmetros definidos pelo usuário.

Por meio da abstração proveniente do PSkel, o processo mestre irá iniciar a execução dos processos trabalhadores nos *clusters* no MPPA-256. A quantidade de *clusters* utilizados na aplicação é determinada dinamicamente em relação à quantidade de *tiles* lógicos e o número de *clusters* definidos pelo usuário. Mais precisamente, dado uma entrada s e um tamanho de *tile* lógico s', a quantidade de *tiles* lógicos é determinada por $\lceil \frac{s}{s'} \rceil$. Caso a quantidade de *clusters* definida pelo usuário seja maior que o resultado da relação apresentada, não é necessário iniciar a execução de todos os *clusters* definidos. Desta

forma, apenas a quantidade de *clusters* necessária será inciada. Caso contrário, a quantidade determinada pelo usuário será utilizada.

Em seguida, o processo mestre irá utilizar a técnica de *tiling* trapezoidal para subdividir a matriz de entrada em *tiles* alargados, e irá enviá-los aos processos trabalhadores seguindo um escalonamento circular (*round-robin*). Devido a isso, alguns *clusters* podem receber mais *tiles* que outros, dependendo do número de *tiles* e *clusters* utilizados na computação. Para efetuar a subdivisão por meio da técnica de *tiling* trapezoidal são necessários cálculos para determinar variáveis que são modificadas dinamicamente, como, por exemplo, endereços de memória da *ghost zone* e a posição do *tile* lógico na matriz de entrada. Mais precisamente, esses cálculos serão realizados para fornecer os deslocamentos necessários para a função `tiling` da classe `StencilTiling`. Essa função é responsável por retornar os endereços de memória corretos de cada *tile* alargado. Desta forma, para determinar precisamente os endereços de memória de cada *tile* alargado são necessários alguns fatores: i) parâmetros definidos pelo usuário, como o tamanho dos dados de entrada, tamanho dos *tiles* lógicos e o número de iterações; ii) parâmetros do *kernel stencil*, como o tamanho da máscara; e iii) os deslocamentos do *tile* lógico em relação à matriz de entrada. Por fim, o processo mestre espera os processos trabalhadores finalizarem a execução para, então, agrupar os *tiles* resultantes em um único `Array2D` de saída. Caso existam mais iterações, será necessário armazenar os resultados enviados pelos processos trabalhadores em uma matriz auxiliar, permitindo a realização de sincronizações entre os processos mestre e trabalhadores.

Mais precisamente, os processos trabalhadores irão realizar iterações internas (t'), enquanto o processo mestre irá realizar iterações externas ($\lceil \frac{t}{t'} \rceil$). Após a computação de todas as iterações internas pelos processos trabalhadores, será necessário uma sincronização entre o processo mestre e os processos trabalhadores. Essa sincronização permitirá que o processo mestre atualize uma matriz auxiliar com resultados temporários, utilizando-a como base para os novos envios de *tiles* alargados em uma nova iteração externa. Com isso, é possível remover valores desatualizados calculados dentro da *ghost zone* e atualizar os dados para todos os processos trabalhadores, eliminando problemas de dependência.

As comunicações entre os processos mestre e trabalhadores são realizadas por meio de portais de comunicação inicializados em cada processo. A Seção 4.3.3 descreve mais profundamente as características da comunicação na implementação.

4.3.2 Processo Trabalhador

Os processos trabalhadores serão responsáveis pela execução do *kernel* da computação *stencil*. Em resumo, cada processo trabalhador executa os seguintes passos: i) recebe o *tile* alargado do processo mestre; ii) computa as iterações internas sobre o *tile* alargado; iii) realiza o cálculo de deslocamentos para determinar o espaço de endereçamento do *tile* lógico; e iv) envia o *tile* resultante ao processo mestre. Após cada *tile* alargado atribuído ao processo trabalhador ser computado, todos os processos trabalhadores precisam sincronizar em uma barreira. Para cumprir esse objetivo, são utilizadas funções de baixo nível do MPPA-256. A sincronização é realizada junto com o mestre, com o objetivo de atualizar a matriz auxiliar para uma nova sequência de envios de *tiles*. Esse processo é repetido até todas as sincronizações (iterações externas) serem realizadas.

A computação do *tile* em cada processo trabalhador é efetuada com o auxílio da biblioteca OpenMP, a qual instanciará 16 *threads* (uma para cada PE do *cluster*). Desta forma, é possível realizar a computação de maneira paralela em cada *cluster*. Será realizada a computação do *tile* alargado por t' iterações internas. Após todas as iterações serem realizadas, o processo trabalhador terá que efetuar cálculos, com deslocamentos enviados pelo processo mestre, para determinar o endereço de memória do *tile* lógico dentro do *tile* alargado. Essa ação deve ser realizada para filtar os valores desatualizados provenientes da computação do *tile* alargado. Por fim, o processo trabalhador irá enviar o *tile* lógico para o processo mestre. Ao completar o envio, os processos trabalhadores terão que sincronizar com o processo mestre e outros processos trabalhadores por meio de uma barreira, caracterizando uma iteração externa. Mais detalhes sobre a comunicação do processo trabalhador com o processo mestre serão abordados na Seção 4.3.3.

O Código 6 mostra a definição do processo trabalhador no MPPA-256. Este processo será responsável por descrever o *kernel* da computação e, também, as estruturas de dados de forma similar ao processo mestre. Devido aos valores de deslocamentos e variáveis importantes para a computação do *kernel* serem geradas dinamicamente, o processo trabalhador precisa receber do processo mestre, além do *tile* alargado, variáveis responsáveis por controlar a comunicação. Dentre elas, temos variáveis indicando as dimensões do *tile* a ser recebido, número de iterações totais e deslocamentos para auxiliar a comunicação. Por fim, as ações de recebimento, computação e envio de *tiles* foram encapsuladas pela função `runMPPA()`, tornando todas as fases descritas para o

Código 6 – Exemplo do código da aplicação Jacobi no processo trabalhador.

```
1 __parallel__ void
2 stencilKernel(Array2D<float> A, Array2D<float> B, Mask2D<int> mask,
3               struct Arguments args, int x, int y){
4   B(x,y) = args.alpha * (A(x,y+1) + A(x,y-1) + A(x+1,y)
5                          + A(x-1,y));
6 }
7
8 int main(int argc, char **argv) {
9   /* declaracoes de variaveis omitidas */
10
11   Array2D<float> inputTile(A, M, N);
12   Array2D<float> outputTile(B, M, N);
13   int neighbors = {{0,1}, {-1,0}, {1,0}, {-1,0}};
14   Mask2D<int> mask(4, neighbors);
15   struct Arguments args(alpha);
16
17   Stencil2D<Array2D<float>, Mask2D<int>, Arguments>
18     jacobi(A, B, mask, args);
19   jacobi.runMPPA(cluster_id, numThreads, numTiles, iterations);
20
21   return(0);
22 }
```

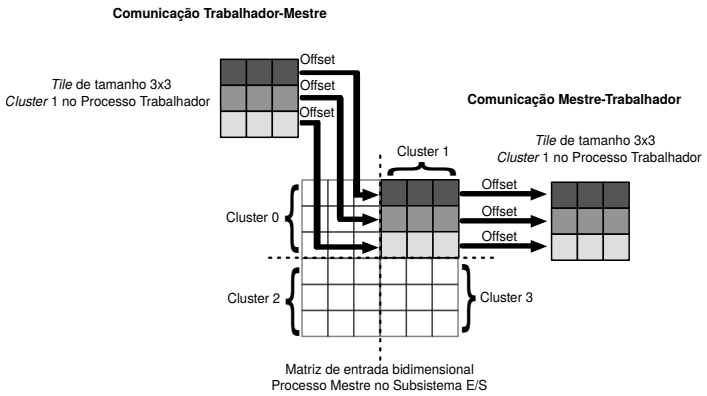
processo trabalhador transparentes ao usuário.

4.3.3 Comunicação

Para efetuar a comunicação, o processo mestre cria portais de escrita e leitura por meio de funções de baixo nível do MPPA-256. A criação dos portais são encapsulados pela estrutura `Array2D`. Desta forma, ao ser criado um `Array2D`, portais de escrita e leitura são vinculados a ele de maneira transparente. Por sua vez, o processo trabalhador irá criar estruturas `Array2D` temporárias para receber o *tile* alargado.

Devido às operações sobre *tiles* e matrizes serem realizadas sobre endereços de memória, quando o processo mestre ou trabalhador deseja enviar um dado, é necessário que ele se apresente contíguo na memória. Caso contrário, dados incorretos podem ser enviados devido à posição deles na memória. Mais precisamente, essa é uma restrição da API e da NoC no MPPA-256, onde os dados armazenados em cada *tile* precisam ser contíguos para serem transferidos pela NoC. Uma solução com cópias locais de dados é simples e contorna o problema, contudo o tempo para efetuar a cópia de um *tile* cresce proporcionalmente com o seu tamanho e desperdiça memória. Desta forma, com o objetivo

Figura 16 – Exemplo do funcionamento do método *strides* no MPPA-256.



Fonte: Desenvolvido pelo autor.

de evitar cópias locais de dados, utiliza-se o conceito de *strides*. Cada stride é uma parte contígua do `Array` original, sendo determinado por deslocamentos (*offsets*) especificados durante a execução. Os deslocamentos são dinâmicos e dependem dos *tiles* sendo computados. Essas informações são conhecidas pelo processo mestre por meio da utilização da classe `StencilTiling`. Por outro lado, o processo trabalhador recebe essas informações do processo mestre.

A Figura 16 ilustra o processo de comunicação com o método *strides* para o caso de uma matriz de entrada de tamanho 6x6, *tiles* de tamanho 3x3 e somente 4 *clusters*. Ao determinar o endereço inicial do *tile*, pode-se definir "saltos" que serão realizados sobre a memória local. Utilizando como exemplo a Figura 16, o endereço inicial do *tile* está em $init + (offset * size)$, onde $size$ é o tamanho, em bytes, do tipo armazenado pela matriz, $init$ é o endereço inicial da matriz e $offset$ é o deslocamento do *tile* relativo à matriz de entrada. O tamanho do salto realizado pelo método é determinado pela largura da matriz de entrada, enquanto a quantidade de saltos é determinada pela altura do *tile*. Além disso, a quantidade de dados após o endereço inicial do *tile* precisa ser determinada (área útil). O tamanho dos saltos é relativo ao endereço inicial do *tile*, então, esse parâmetro deve contabilizar, também, a área útil do *tile*. Ao realizar todos os saltos, o método irá enviar diretamente, via portal, os dados especificados pelo endereço

inicial do *tile* e pelos saltos na memória com a área útil especificada. Com isso, é possível enviar de forma contígua e direta os *tiles* a outro processo, sem a necessidade de cópias locais.

Por outro lado, o processo trabalhador precisa enviar apenas o *tile* lógico. Desta forma, o endereço inicial para envio não será modificado, contudo é necessário especificar a quantidade e tamanho dos saltos. A quantidade de saltos é determinada pela altura do *tile*, enquanto o tamanho dos saltos e a área útil é igual à largura do *tile*. Com o método *strides* é possível, também, determinar saltos no endereço de memória do processo destino (*target*). O processo trabalhador precisa enviar o *tile* resultante na posição de memória correta no processo mestre. Portanto, ele irá utilizar um deslocamento $(heightOffset * width) + widthOffset$ sobre o *target*, onde $heightOffset$ é o deslocamento relativo à altura, $width$ é a largura do *tile* e $widthOffset$ é o deslocamento relativo à largura. O processo mestre não utiliza essa função do método, pois não é necessário inserir o *tile* em uma posição de memória específica no processo trabalhador.

Por fim, para fornecer uma maior facilidade ao usuário, a técnica de *tiling* utilizada, as comunicações via NoC e adaptações foram realizadas dentro do *back-end* do PSkel.

5 EXPERIMENTOS

Neste capítulo é avaliado o desempenho e o consumo energético de aplicações *stencil* do PSkel quando executadas no MPPA-256 e em um sistema NUMA com dois nós. Mais especificamente, o sistema contém duas CPUs Intel Xeon E5-2640 v4 com 10 núcleos de 2.4GHz, onde cada núcleo possui 2 *threads*, e 64GB DDR4-1600 de RAM com uma largura de banda máxima de 68.3 GB/s. O processador Intel Xeon é baseado na microarquitetura Broadwell com uma interconexão *Quick Path Interconnect* (QPI) de 8 GT/s e uma potência média de 90 W. As medições de energia no MPPA-256 foram feitas considerando todos os *clusters*, a memória, os subsistemas de E/S e a NoC, onde foram coletadas por meio dos sensores de potência e energia disponíveis no MPPA-256. Como o MPPA-256 possui características intrínsecas do próprio processador que garantem uma baixa variabilidade entre as execuções, foram realizadas somente 5 repetições de cada experimento, computando-se a média aritmética dos valores. Todos os experimentos no MPPA-256 consideraram 16 PEs por *cluster*. Por outro lado, as medições de energia e de tempo de execução no sistema NUMA consideraram apenas um nó e foram feitas com auxílio do *Running Average Power Limit* (RAPL) através da biblioteca PAPI (WEAVER et al., 2012), considerando a memória e todos os núcleos. Em cada experimento foram utilizadas 10 *threads* sem uso de *hyperthreading*. Os resultados mostram a média aritmética de 30 execuções. Por fim, todos os resultados mostraram um desvio padrão menor que 1%.

Devido às limitações de memória em cada *cluster*, ao ser especificado uma quantidade muito grande de iterações, o *tile* alargado não pode ser maior do que a memória presente em cada *cluster*. Portanto, foi fixado uma quantidade de 10 iterações internas para cada *cluster*, garantindo-se assim que a quantidade de memória utilizada fosse menor do que 2 MB (tamanho da memória local do *cluster*). Além disso, para os experimentos terem uma quantidade significativa de sincronizações sobre a execução de cada aplicação, foram utilizadas 30 iterações para cada aplicação.

5.1 APLICAÇÕES *STENCIL*

Para a realização dos experimentos foram utilizadas as seguintes aplicações *stencil*:

Fur Modela a formação de padrões sobre a pele de animais[1]. Nessa aplicação, a pele do animal é modelada por uma matriz bidimensional de células de pigmento que podem estar em um dos dois estados: colorida ou não-colorida. As células coloridas secretam ativadores e inibidores. Ativadores fazem uma célula central se tornar colorida; inibidores, por outro lado, fazem uma célula central se tornar não colorida. A diferença entre as potências dos ativadores e inibidores é responsável por decidir a coloração da célula central, onde mais ativadores resulta em uma célula colorida e mais inibidores resulta em uma célula não colorida. Nos casos em que as potências dos ativadores e inibidores forem iguais, a cor da célula permanece inalterada. A máscara contém células adjacentes à célula central e seu tamanho é parametrizável. Neste trabalho foram utilizados 2 vizinhos adjacentes em cada direção.

Jacobi Método iterativo para resolver sistemas de equações lineares (DEMMEL, 1997). O método converge garantidamente se a matriz de entrada é restrita ou irredutivelmente dominante diagonalmente, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, para todo i. A Equação 5.1 define a computação em cada passo do método iterativo de Jacobi para resolver a equação discreta elíptica de Poisson (DEMMEL, 1997). A solução aproximada é computada discretizando o problema na matriz em pontos espaçados de forma equivalente por $n \times n$.

$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \tag{5.1}$$

A cada passo, o novo valor de $u_{i,j}$ é obtido fazendo a média $h^2 f_{i,j}$ dos seus vizinhos, onde $h = \frac{1}{n+1}$ e $f_{i,j} = f(ih, jh)$, para uma dada função f.

GoL Autômato celular que implementa o Jogo da Vida de Conway (GARDNER, 1970). O autômato é representado por uma matriz bidimensional, onde cada elemento representa um indivíduo vivo ou um indivíduo morto. A máscara do *stencil*, a qual determina a interação entre o indivíduo e seus vizinhos, considera as 8 células vizinhas adjacentes à célula central. Dependendo dos valores dos vizinhos, o elemento pode modificar seu estado entre vivo e morto.

[1] http://ccl.northwestern.edu/netlogo/models/Fur

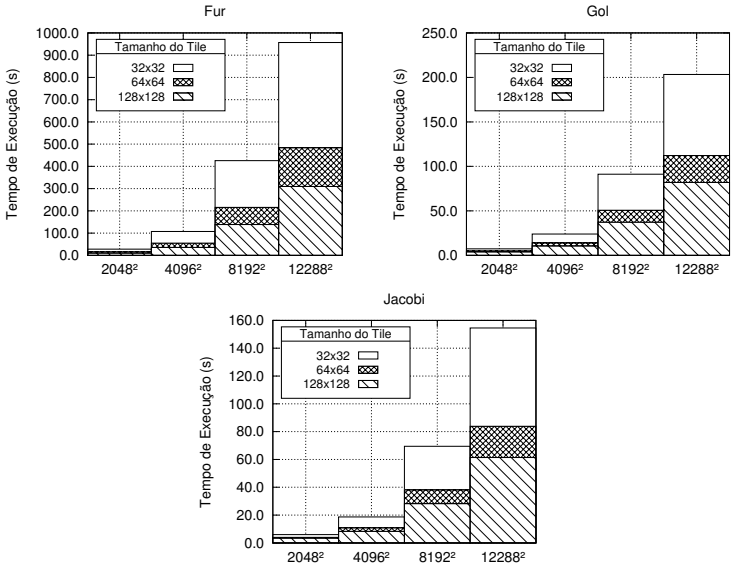


Figura 17 – Tempos de execução das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

5.1.1 Impacto do Tamanho do *Tile* no Desempenho do MPPA-256

O primeiro experimento tem por objetivo verificar o impacto do tamanho dos *tiles* no desempenho e consumo energético das aplicações. As Figuras 17 e 18 mostram, respectivamente, os tempos de execução e consumo de energia de três aplicações *stencil*, variando-se o tamanho do `Array2D` de entrada (de 2048^2 até 12288^2) e os tamanhos do *tile* (de 32^2 até 128^2). Matrizes de entrada maiores que 12288^2 e *tiles* maiores que 128^2 extrapolam às memórias LPDDR3 e dos *clusters*, respectivamente.

Pode-se perceber uma redução no tempo de execução à medida em que se aumenta o tamanho do *tile* (Figura 17), pois há menos sincronizações e comunicações de *tiles* entre os processos mestre e escravos. O comportamento das aplicações é similar, sendo diferenciado apenas pela grandeza dos tempos de execução. A Figura 18 apresenta um comportamento similar para o consumo de energia, pois o tempo de execução reduz com o aumento do tamanho do *tile*, trazendo uma redução no consumo de energia.

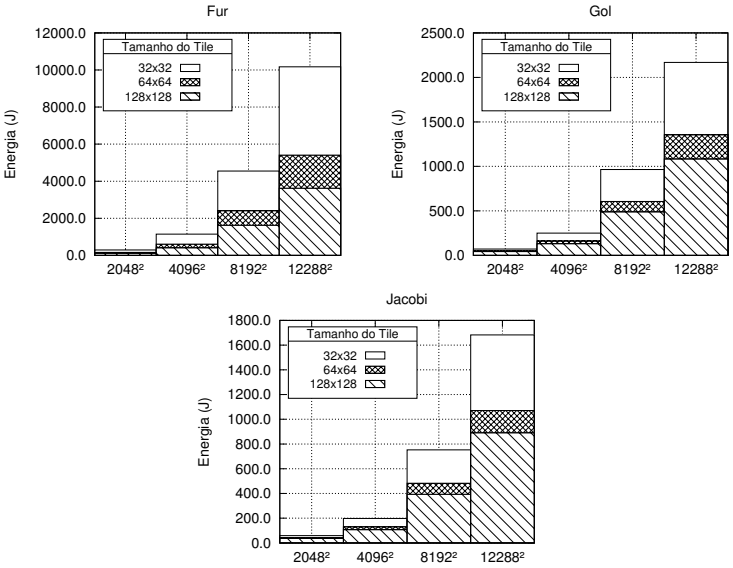
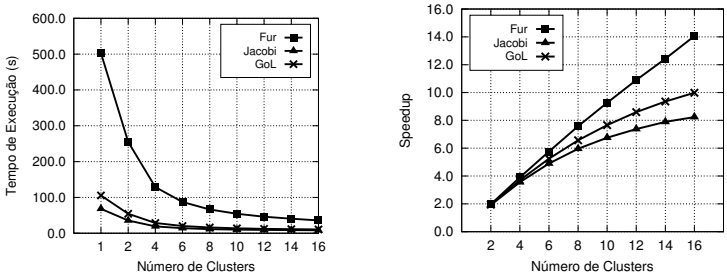


Figura 18 – Consumo de energia das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

5.1.2 Análise de Escalabilidade no MPPA-256

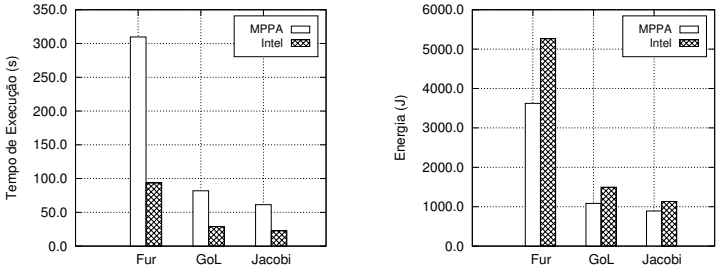
Em um segundo experimento, buscou-se verificar a escalabilidade das aplicações no MPPA-256. Para isso, variou-se o número de *clusters* em cada aplicação, com `Array2D` de entrada fixo de tamanho 4096^2 e *tiles* de tamanho 128^2. A Figura 19(a) apresenta os tempos de execução obtidos ao variar-se o número de *clusters* utilizados na computação. A Figura 19(b), por outro lado, apresenta o fator de aceleração (*speedup*) com relação ao tempo de execução com 1 *cluster*. Em outras palavras, o *speedup* com *c clusters* é computado dividindo-se o tempo de execução obtido com apenas 1 *cluster* pelo tempo de execução obtido com *c clusters*.

No geral, os resultados mostraram que a solução proposta para o MPPA-256 é escalável. Porém, pode-se notar que a aplicação *Fur* apresentou uma escalabilidade superior às demais aplicações. Esse comportamento está diretamente relacionado com a quantidade de operações realizadas pelo *kernel* da aplicação (complexidade do *kernel*). Tendo em vista a necessidade de comunicações no MPPA-256, o tempo total



(a) Tempo de execução. (b) Speedup.

Figura 19 – Resultados de tempo e *speedup* das aplicações *Fur*, *GoL* e *Jacobi*.



(a) Tempo de execução. (b) Consumo de Energia.

Figura 20 – Comparação do tempo de execução e consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* em relação a arquitetura.

de execução de uma aplicação passa a ser composto pela soma do tempo de comunicação com o tempo de computação. Para um dado *tile* t de tamanho fixo, o tempo necessário para realizar comunicações de t entre mestre e escravo será constante. Por outro lado, quanto maior o número de operações (computações) feitas em t pelo *kernel* da aplicação, maior será o paralelismo a ser explorado. Nesse caso, o tempo de computação será proporcionalmente maior que o tempo de comunicação, melhorando assim a escalabilidade obtida. Este é o caso da aplicação *Fur* cujo *speedup* se aproxima do caso ideal. Em contrapartida, aplicação *Jacobi* apresentou uma escalabilidade mais baixa que as demais, pois seu *kernel* apresenta baixa complexidade.

5.1.3 Kalray MPPA-256 vs. Intel Xeon

Por fim, foram efetuados experimentos comparativos entre o processador Intel Xeon e o MPPA-256, como pode-se ver na Figura 20. Nesses experimentos, utilizou-se um `Array2D` de entrada de tamanho 12288^2 e *tiles* de tamanho 128^2. Ao ser comparado o tempo das aplicações em cada arquitetura, nota-se que o MPPA-256 tem um desempenho pior, contudo ao ser comparado o consumo de energia percebe-se um comportamento diferente: a energia consumida pelo MPPA-256 é menor, principalmente na aplicação *Fur*. No geral, o consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* no MPPA-256 foi aproximadamente 1.45x, 1.38x e 1.27x menor que no Intel Xeon, respectivamente. Por outro lado, o tempo de execução dessas aplicações obtido no MPPA-256 foi 3.30x, 2.83x e 2.69x maior que no Intel Xeon, respectivamente. O experimento mostrou que o MPPA-256 possui um menor desempenho em relação ao processador da Intel. Esse comportamento é devido ao processo de construção de um *tile* alargado, onde é necessário realizar manualmente cálculos de deslocamentos e criação do *tile* aumentado. Mais precisamente, em situações onde o número de *tiles* é maior que a quantidade de *clusters*, o processo mestre irá enviar os *tiles* em mais de uma etapa. Em cada etapa será realizado o cálculo de deslocamentos, a criação do *tile* aumentado e, em seguida, será realizado o envio para apenas um *cluster*. Desta forma, *clusters* ficam ociosos durante o tempo em que o mestre efetua esse processo, prejudicando o desempenho. Além disso, esse comportamento é percebido, também, em execuções com apenas uma iteração, isto é, apenas uma etapa.

6 CONCLUSÃO

Este trabalho discutiu inicialmente a relação entre o aumento de núcleos e o consumo de energia nos processadores atuais. Como mostrado, a energia necessária para alcançar supercomputadores *Exascale* é muito alta, se tornando necessária a utilização de novas técnicas e processadores energeticamente eficientes. Contudo, esses processadores apresentam dificuldades de programação, tais como a existência de um modelo de programação híbrido, capacidade limitada de memória no *chip*, ausência de coerência de *cache*, entre outros. Essas dificuldades trazem problemas para o desenvolvimento de aplicações e motivam a utilização de *frameworks* sobre esses ambientes.

Neste trabalho foi proposta uma adaptação de um *framework* para desenvolvimento de aplicações *stencil* iterativas, denominado PSkel, para processador MPPA-256. A solução proposta permite esconder detalhes de baixo nível do MPPA-256, simplificando significativamente o desenvolvimento de aplicações *stencil* nesse processador. Os resultados mostraram que a solução proposta apresenta boa escalabilidade. Além disso, foi observada uma redução significativa no tempo de execução e no consumo de energia das aplicações no MPPA-256 ao se utilizar a técnica de *tiling* trapezoidal. Isso se deve, principalmente, à redução do sobrecusto de comunicações e sincronizações de *tiles*.

A aplicação *Fur* apresentou os melhores resultados de escalabilidade dentre as 3 aplicações estudas, obtendo um *speedup* de 14x em relação a apenas um *cluster*. Analisando experimentos executados sobre a adaptação pôde-se perceber uma relação entre a quantidade de computação realizada pelo *kernel* da aplicação e o *speedup* obtido. Por fim, experimentos comparativos entre o MPPA-256 e o processador Intel Broadwell mostraram que a solução proposta para o MPPA-256 apresenta uma eficiência energética superior apesar de um tempo de execução superior.

Como trabalhos futuros, pretende-se estudar formas de reduzir ainda mais os sobrecustos de comunicação através do uso de técnicas de *software prefetching*. Esta técnica possibilitará a construção de *tiles* alargados durante as computações de outros *tiles* pelos *clusters*. Portanto, ao terminar a computação, outro *tile* alargado estará esperando para ser computado. Desta forma, será possível diminuir o impacto da construção de *tiles* alargados e seu envio sobre o desempenho. Além disso, pretende-se realizar experimentos com outros *benchmarks* e aplicações que utilizam estruturas tridimensionais. Por fim, pretende-se

realizar comparações de desempenho e consumo de energia com outros processadores embarcados.

REFERÊNCIAS

BUONO, D. et al. Parallel Patterns for General Purpose Many-Core. In: *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Belfast, UK: IEEE, 2013. p. 131–139. ISSN 1066-6192.

CASTRO, M. et al. Energy Efficient Seismic Wave Propagation Simulation on a Low-power Manycore Processor. In: *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Paris, France: IEEE, 2014. p. 57–64. ISSN 1550-6533.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, Elsevier, v. 54, p. 108–120, 2016. ISSN 0167-8191.

CASTRO, M. et al. Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application. In: *Workshop on Irregular Applications: Architectures & Algorithms (IA^3)*. Denver, EUA: ACM, 2013. p. 5:1 – 5:8. ISBN 978-1-4503-2503-5.

COLE, M. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, Elsevier, Amsterdam, The Netherlands, v. 30, p. 389–406, 2004. ISSN 0167-8191.

DEMMEL, J. W. *Applied numerical linear algebra*. Berkeley, EUA: SIAM, 1997. ISBN 978-0-89871-389-3.

ENMYREN, J.; KESSLER, C. W. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In: *International Workshop on High-level Parallel Programming and Applicationsi (HLPP)*. Baltimore, USA: ACM, 2010. p. 5–14. ISBN 978-1-4503-0254-8.

FRANCESQUINI, E. et al. On the Energy Efficiency and Performance of Irregular Applications on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing*, v. 76, p. 32–48, 2014. ISSN 0743-7315.

FU, H. et al. The Sunway TaihuLight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, Science China Press, v. 59, p. 072001:1–072001:16, 2016. ISSN 1869-1919.

GARDNER, M. Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, v. 223, 1970. ISSN 0036-8733.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. New Jersey, USA: Prentice-Hall, Inc., 2006. ISBN 9780133356724.

GYSI, T.; GROSSER, T.; HOEFLER, T. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In: *ACM ICS*. California, USA: ACM, 2015. p. 177–186. ISBN 978-1-4503-3559-1.

HOLEWINSKI, J.; POUCHET, L.-N.; SADAYAPPAN, P. High-Performance Code Generation for Stencil Computations on GPU Architectures. In: *ACM ICS*. Venice, Italy: ACM, 2012. p. 311–320. ISBN 978-1-4503-1316-2.

KALE, L. V.; BHATELE, A. *Parallel Science and Engineering Applications: The Charm++ Approach*. 1st. ed. Florida, USA: CRC Press, 2013. 314 p. ISBN 978-1-4665-0412-7.

KOGGE, P. et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. [S.l.], 2008. 278 p.

LUTZ, T.; FENSCH, C.; COLE, M. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 9, p. 59:1–59:24, 2013. ISSN 1544-3566.

MENG, J.; SKADRON, K. A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, Springer US, v. 39, p. 115–142, 2011. ISSN 0885-7458.

MORARI, A. et al. Efficient Sorting on the Tilera Manycore Architecture. In: *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. New York, USA: IEEE, 2012. p. 171–178. ISSN 1550-6533.

PADOIN, E. L. et al. Performance/Energy Trade-off in Scientific Computing: The Case of ARM big.LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques*, p. 1–9, 2015. ISSN 1751-8601.

PEREIRA, A. D.; RAMOS, L.; GÓES, L. F. W. PSkel: A Stencil Programming Framework for CPU-GPU Systems. *Concurrency and Computation: Practice and Experience*, v. 27, p. 4938–4953, 2015. ISSN 1532-0634.

ROCHA, R. C. O. et al. TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience*, v. 29, p. e4053, 2017. ISSN 1532-0634.

SIRDEY, P. A. et al. Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In: *International Conference on Computational Science (ICCS)*. Barcelona, Spain: Elsevier, 2013. p. 1624–1633. ISSN 1877-0509.

STEUWER, M.; KEGEL, P.; GORLATCH, S. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPSW)*. Shangai, China: IEEE, 2011. p. 1176–1182. ISBN 978-0-7695-4577-6. ISSN 1530-2075.

TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. 4. ed. Boston, London: Pearson, 2015. 1137 p. ISBN 978-0-13-359162-0.

THORARENSEN, S. et al. Efficient Execution of SkePU Skeleton Programs on the Low-Power Multicore Processor Myriad2. In: *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Heraklion, Greece: IEEE, 2016. p. 398–402. ISSN 2377-5750.

TOTONI, E. et al. Comparing the Power and Performance of Intel's SCC to State-of-the-Art CPUs and GPUs. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. New Brunswick, Canada: IEEE, 2012. p. 78–87. ISBN 9781467311441.

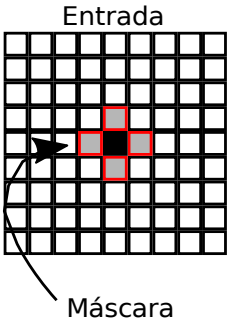
VARGHESE, A. et al. Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor. In: *International Parallel Distributed Processing Symposium Workshops (IPDPSW)*. Phoenix, USA: IEEE, 2014. p. 984–992. ISBN 978-1-4799-4116-2.

WEAVER, V. M. et al. Measuring Energy and Power with PAPI. In: *2012 41st International Conference on Parallel Processing Workshops*. Pittsburgh, USA: IEEE, 2012. p. 262–268. ISBN 978-1-4673-2509-7. ISSN 0190-3918.

ANEXO A – Implementação com o PSkel-MPPA

Neste anexo será apresentado como o desenvolvedor poderá implementar uma aplicação paralela sobre a adaptação proposta do *framework* PSkel. Na adaptação o desenvolvedor terá que implementar o código sobre o *framework* PSkel para os processos mestre e trabalhador. Como pode-se analisar no código 7, o desenvolvedor terá que implementar a função `main` com as estruturas principais. Primeiramente, o desenvolvedor irá declarar matrizes de entrada e saída com as abstrações provenientes do *framework* (linhas 4 e 5), onde $\boldsymbol{A}$ é a matriz de entrada, $\boldsymbol{M}$ e $\boldsymbol{N}$ são as dimensões de largura e altura respectivamente. Para especificar a máscara da computação, o desenvolvedor deve determinar quais vizinhos serão utilizados de forma genérica, por meio de posições relativas à célula central (linhas 6). Mais especificamente, a Figura 21 ilustra uma possível configuração de vizinhos, onde temos as posições: `pos(-1,0)` para a célula à esquerda, `pos(0,1)` para a célula superior, `pos(0,-1)` para a célula inferior e `pos(1,0)` para a célula à direita. Após a caracterização dos vizinhos, será definida a máscara da computação por meio da abstração proveniente do *framework* (linha 7). A estrutura `Arguments` (linha 8) será responsável por enviar ao *Kernel* da computação variáveis auxiliares definidas pelo desenvolvedor. Desta forma, o desenvolvedor precisa definir uma `struct` manualmente e passar para a estrutura principal do *framework*, denominada `Stencil`. A estrutura `Stencil` apresenta configurações `1D`, `2D` e `3D`. Na declaração da estrutura, o desenvolvedor irá determinar as matrizes de entrada e saída

Figura 21 – Esquemático ilustrando possíveis vizinhos em uma computação estêncil.



Fonte: Desenvolvido pelo autor.

utilizadas, a máscara da computação e a estrutura de argumentos (linhas 10 e 11). Por fim, a estrutura `Stencil` possui um método de

Código 7 – Exemplo em pseudo código de uma aplicação genérica no processo mestre.

```
 1 int main(int argc, char **argv) {
 2     /* variaveis omitidas */
 3
 4     Array2D<float> inputTile(A, M, N);
 5     Array2D<float> outputTile(B, M, N);
 6     int neighbors = {pos(x,y), pos(x,y), ...};
 7     Mask2D<int> mask(neighbors.size(), neighbors);
 8     struct Arguments args(alpha);
 9
10     Stencil2D<Array2D<float>, Mask2D<int>, Arguments>
11         application(A, B, mask, args);
12     application.scheduleMPPA(slave_bin, threadsNum, clustersNum,
13                              tileDim, iterations);
14
15     return(0);
16 }
```

Código 8 – Exemplo em pseudo código de uma aplicação genérica no processo trabalhador.

```
 1 __parallel__ void
 2 stencilKernel(Array2D<float> A, Array2D<float> B, Mask2D<int> mask,
 3 struct Arguments args, int x, int y){
 4     B(x,y) = Kernel determinado pelo desenvolvedor
 5 }
 6
 7 int main(int argc, char **argv) {
 8     /* variaveis omitidas */
 9
10     Array2D<float> inputTile(A, M, N);
11     Array2D<float> outputTile(B, M, N);
12     int neighbors = {pos(x,y), pos(x,y), ...};
13     Mask2D<int> mask(neighbors.size(), neighbors);
14     struct Arguments args(alpha);
15
16     Stencil2D<Array2D<float>, Mask2D<int>, Arguments>
17         application(A, B, mask, args);
18     application.runMPPA(cluster_id, numThreads, numTiles,
19                         iterations);
20
21     return(0);
22 }
```

gerenciamento para o MPPA-256, denominado `scheduleMPPA`, responsável por realizar o particionamento de dados e comunicação (linhas 12 e 13). Esse método será chamado passando como parâmetros: o nome do binário do processo trabalhador[1], o número de *threads* e *clusters*

[1] O nome do binário é determinado pelo desenvolvedor.

definidos pelo usuário, dimensão dos *tiles* e número de iterações sobre a aplicação.

Para o processo trabalhador, o desenvolvedor terá que implementar a função `main` de forma similar ao processo mestre. Ao final das declarações para preparar as estruturas, o desenvolvedor irá chamar o método de execução para o MPPA-256, denominado `runMPPA`, passando o identificador do *cluster*, o número de *threads*, número de *tiles* passados pelo processo mestre e o número de iterações (linhas 18 e 19). Por fim, o desenvolvedor terá que implementar a computação da aplicação, isto é, o *Kernel* estêncil da aplicação (linhas 1-5). O *Kernel*, geralmente, utiliza elementos da matriz de entrada ($\boldsymbol{A}$), realiza operações sobre esse elemento de acordo com os vizinhos caracterizados pela máscara da computação e atribui o resultado à matriz de saída ($\boldsymbol{B}$) na posição respectiva. Além disso, a computação recebe como parâmetro a estrutura de argumentos para utilizar variáveis auxiliares definidas pelo usuário. Exemplos de aplicações e da adaptação podem ser encontrados no seguinte link: `https://github.com/pskel/pskel/tree/mppaWidth`.

ANEXO B – Artigo científico

Execução Energeticamente Eficiente de Aplicações Estêncil com o Processador *Manycore* MPPA-256

Emmanuel Podestá Jr.[1], **Alyson D. Pereira**[1], **Rodrigo C. O. Rocha**[2], **Márcio Castro**[1], **Luís F. W. Góes**[2]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

[2] Grupo de Computação Criativa e Paralela (CreaPar)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – MG, Brasil

emmanuel.podesta@grad.ufsc.br, alyson.pereira@posgrad.ufsc.br,
rcor@pucminas.br, marcio.castro@ufsc.br, lfwgoes@pucminas.br

Abstract. *In this paper is proposed an adaptation for the framework PSkel to the low-power manycore processor MPPA-256. The framework simplifies iterative stencil applications development to the MPPA-256, hiding implementation details from the developer. The results on MPPA-256 showed an energy consumption reduction on iterative stencil applications up to 1.45x in comparison to the multicore processor Intel Broadwell.*

Resumo. *Neste artigo é proposta uma adaptação do framework PSkel para o processador manycore de baixa potência MPPA-256. O framework permite simplificar o desenvolvimento de aplicações estêncil iterativas para o MPPA-256, escondendo do desenvolvedor detalhes de implementação. Os resultados obtidos no MPPA-256 mostraram uma redução do consumo de energia de aplicações estêncil iterativas de até 1.45x em comparação com um processador multicore Intel Broadwell.*

1. Introdução

Plataformas de Computação de Alto Desempenho (CAD) tem sido avaliadas quase que exclusivamente pela suas capacidades de processamento. Contudo, o consumo excessivo de energia é uma barreira para o aumento de desempenho de forma escalável nestas plataformas. Por essa razão, o estudo de técnicas que melhorem a eficiência energética em plataformas de CAD está se tornando muito importante. Recentemente, uma nova classe de processadores *manycore* de baixa potência tais como o Sunway SW26010 [Fu et al. 2016] e o Kalray MPPA-256 [Castro et al. 2013] foram desenvolvidos. Esses processadores possuem centenas de núcleos de processamento capazes de lidar com paralelismo de dados e tarefas com baixo consumo de energia.

Processadores *manycore* de baixa potência (*low-power manycore processors*) apresentam uma melhor eficiência energética em comparação com processadores de propósito geral presentes atualmente [Francesquini et al. 2014], contudo as suas características arquiteturais tornam o desenvolvimento de aplicações uma tarefa desafiadora [Varghese et al. 2014, Castro et al. 2016, Castro et al. 2014].

Existem diversos padrões paralelos para simplificar o desenvolvimento de aplicações paralelas. Dentre eles (e.g., *map*, *reduce*, *pipeline* e *scan*), o padrão estêncil tem sido muito utilizado em várias áreas importantes, como física quântica, previsão do tempo e processamento de imagens [Gonzalez and Woods 2006, Holewinski et al. 2012, Lutz et al. 2013]. No padrão estêncil, para cada elemento de uma estrutura n-dimensional

de entrada é computado um novo valor para o respectivo elemento em uma estrutura n-dimensional de saída, utilizando-se como base os valores dos elementos vizinhos ao elemento de entrada. A quantidade de vizinhos e a computação a ser realizada em cada elemento é definida por uma função (ou *kernel*) estêncil. Em aplicações estêncil iterativas, os valores produzidos na estrutura n-dimensional de saída em uma iteração i são utilizados como entrada da iteração $i + 1$.

Alguns *frameworks* foram propostos para o desenvolvimento de aplicações paralelas com base no padrão estêncil, tais como SkelCL [Steuwer et al. 2011], SkePU [Enmyren and Kessler 2010] e PSkel [Pereira et al. 2015]. Em especial, o *framework* PSkel provê uma abstração de alto nível para o desenvolvimento de aplicações estêncil em ambientes heterogêneos compostos por processadores *multicore* e *Graphical Processing Units* (GPUs). Todavia, nenhum desses *frameworks* possui suporte para processadores *manycore* de baixo potência de energia emergentes tais como o MPPA-256.

Portanto, nesse artigo é proposta uma adaptação completa do *framework* PSkel para o processador MPPA-256, a qual permite simplificar significativamente o desenvolvimento de aplicações estêncil nesse processador. A adaptação permite eliminar as dificuldades de desenvolvimento intrínsecas do processador, fazendo com que as aplicações já implementadas em PSkel possam ser executadas no MPPA-256 sem a necessidade de nenhuma modificação em seus códigos. Os resultados obtidos mostram que o MPPA-256 apresenta uma melhor eficiência energética que um processador Intel Broadwell com 10 núcleos físicos ao executar três aplicações estêncil implementadas no PSkel.

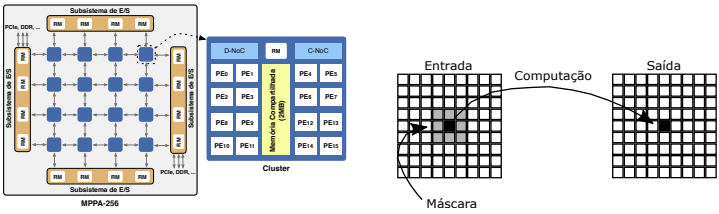
O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os principais conceitos do processador *manycore* MPPA-256 e do *framework* PSkel. A Seção 3 discute a adaptação do *framework* PSkel para fornecer suporte ao MPPA-256. Os resultados obtidos com a adaptação do *framework* PSkel para o MPPA-256 são apresentados na Seção 4. Por fim, a Seção 5 apresenta as conclusões deste trabalho.

2. Fundamentação Teórica

2.1. MPPA-256

O MPPA-256 é um processador *manycore* desenvolvido pela empresa francesa Kalray, o qual possui 256 núcleos de processamento de 400 MHz denominados *Processing Element*s (PEs). Além dos PEs, o processador possui 32 núcleos dedicados a gerência de recursos denominados *Resource Managers* (RMs). PEs e RMs são distribuídos fisicamente no *chip* em 16 *clusters* e 4 subsistemas de Entrada/Saída (E/S), contendo cada *cluster* 16 PEs e 1 RM. Além dos *clusters*, o MPPA-256 possui 4 subsistemas de E/S contendo, cada um, 4 RMs. Toda a comunicação entre *clusters* e/ou subsistemas de E/S é feita através de uma *Network-on-Chip* (NoC) *torus* 2D. A arquitetura do MPPA-256 pode ser vista na Figura 1a.

A finalidade principal dos PEs é executar *threads* de usuário de forma ininterrupta e não preemptível para realização de computação. PEs de um mesmo *cluster* compartilham uma memória de 2 MB, a qual é utilizada para armazenar os dados a serem processados pelos PEs. Cada PE possui também uma memória *cache* associativa 2-*way* de 32KB para dados e uma para instruções. Porém, o processador não dispõe de coerência de *caches*, o que dificulta o desenvolvimento de aplicações para esse processador. Por outro lado, a finalidade dos RMs é gerenciar E/S, controlar comunicações entre *clusters* e/ou subsistemas de E/S e realizar comunicação com uma memória RAM. Na arquitetura utilizada neste artigo, um dos subsistemas de E/S está conectado a uma memória externa *Low Power Double Data Rate 3* (LPDDR3) de 2 GB.



(a) MPPA-256. (b) O padrão estêncil.

Figura 1. Visão geral do MPPA-256 (esquerda) e uma ilustração do padrão estêncil oferecido pelo PSkel (direita).

Estudos anteriores mostraram que desenvolver aplicações paralelas otimizadas para o MPPA-256 é um grande desafio [Francesquini et al. 2014] devido a alguns fatores importantes tais como: **(i) modelo de programação híbrido**: *threads* em um mesmo *cluster* se comunicam através de uma memória compartilhada local, porém a comunicação entre *clusters* é feita explicitamente via NoC, em um modelo de memória distribuída; **(ii) comunicação**: é necessário a utilização de uma *Application Programming Interface* (API) específica para a comunicação via NoC, similar ao modelo clássico POSIX de baixo nível para *Inter-Process Communication* (IPC); **(iii) memória**: cada *cluster* possui apenas 2 MB de memória local de baixa latência, portanto aplicações reais precisam constantemente realizar comunicações entre o subsistema de Entrada e Saída (E/S) (conectado à memória LPDDR3); e **(iv) coerência de *cache***: cada PE possui uma memória *cache* privada sem coerência com as *caches* dos demais PEs, sendo necessário o uso explícito de instruções do tipo *flush* para atualizar a *cache* de um PE em determinados casos.

2.2. PSkel

O PSkel é um *framework* de programação em alto nível para o padrão estêncil, baseado nos conceitos de esqueletos paralelos, que oferece suporte para execução paralela em ambientes heterogêneos incluindo CPU e GPU. Utilizando uma única interface de programação escrita em C++, o usuário é responsável apenas por definir o *kernel* principal da computação estêncil, enquanto o *framework* se encarrega de gerar código executável para as diferentes plataformas paralelas, realizando de maneira transparente todo o gerenciamento de memória e transferência de dados entre dispositivos [Pereira et al. 2015].

A Figura 1b ilustra o funcionamento da computação estêncil em aplicações iterativas. Em cada iteração, uma máscara de vizinhança é utilizada na matriz de entrada para determinar o valor de cada célula da matriz de saída. Nesse exemplo, o valor de cada célula da matriz de saída é determinado em função dos valores das células vizinhas em todas as direções. Esse processo é realizado para todos os pontos da matriz de entrada, produzindo uma matriz saída da computação estêncil. Ao final de uma iteração, a matriz de saída será considerada como sendo a matriz de entrada da próxima iteração, gerando assim uma nova matriz de saída ao final da próxima iteração.

3. Adaptação do *framework* PSkel para o MPPA-256

A adaptação do *framework* PSkel para o processador MPPA-256 proposta neste artigo segue um modelo mestre/escravo. Um processo mestre é executado no subsistema de E/S conectado à memória LPDDR3 de 2 GB, sendo responsável por alocar o `Array` de entrada e por distribuir os dados entre os processos escravos. Em cada *cluster* é instanciado

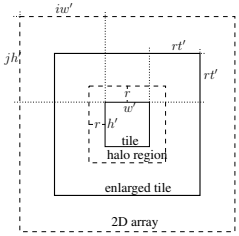


Figura 2. Diagrama do *tiling* 2D. Um *tile* lógico (linha interna sólida) é contido dentro do Array 2D (linha externa pontilhada) com *offsets* verticais e horizontais dado por jh' e iw'. Computar t' consecutivas iterações estêncil no *tile* requer um aumento no *tile* lógico com uma *ghost zone* (área entre a linha interna sólida e a linha externa sólida), que é constituída de regiões *halo* (área entre a linha interna sólida e a linha interna pontilhada).

um único processo escravo que é responsável por gerenciar a computação no seu *cluster*. Devido às limitações de memória dos *clusters* (apenas 2 MB por *cluster*), o processo mestre deve subdividir o `Array` de entrada em blocos denominados *tiles* e, então, gerenciar as comunicações dos mesmos com os processos escravos.

O processo mestre particiona o `Array` de entrada com dimensão n em b blocos, onde b é o número de *clusters* utilizados na computação. Então, cada bloco é particionado em *tiles* de tamanho fixo definidos pelo usuário. Quando são feitas computações estêncil sobre o *tile*, dependências de vizinhança, inerentes ao padrão paralelo do estêncil, precisam ser consideradas durante o particionamento dos dados. Uma das principais soluções para satisfazer essas dependências é via blocos sobrepostos, resultando em dados redundantes e computação por *tile* [Meng and Skadron 2011, Holewinski et al. 2012, Rocha et al. 2017]. Essa técnica é muito importante em *manycores* de baixa potência como o MPPA-256, onde o sobrecusto de comunicação pode ser elevado. O impacto dos custos de comunicação será analisado posteriormente na Seção 4.

Portanto, foi implementada uma técnica de *tiling* trapezoidal. Para ilustrar e detalhar como essa técnica de *tiling* pode ser aplicada na computação estêncil, utilizamos a definição a seguir. Seja A um `Array2D`, com dimensões $\dim(A) = (w, h)$, onde w e h são, respectivamente, a largura e a altura. Utilizando *tiles* de dimensões (w', h') produz $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ *tiles* possíveis de A. Seja $A_{i,j}$ um *tile*, onde $0 \leq i < \lceil \frac{w}{w'} \rceil$ e $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ possui *offset* (iw', jh') relativo ao canto superior esquerdo de A e $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. O *offset* é uma indexação de deslocamento necessário para acessar os elementos do *tile* (Figura 2). Essa técnica pode ser facilmente estendida para mais dimensões.

Aplicar um estêncil em A envolve aplicar a função de vizinhança (máscara) contendo o deslocamento de cada vizinho de um dado elemento central. Por causa da dependência entre vizinhos, para computar a função estêncil, de acordo com as limitações necessárias pelos *tiles*, se torna necessário obter valores de *tiles* adjacentes. Seja r o *range* da máscara de vizinhos, i.e., r é o deslocamento mais distante necessário para a vizinhança definida pela máscara. A área de r envolvendo a vizinhança é denominada região *halo*. Se a função estêncil é aplicada iterativamente sobre A, para t iterações, a dependência da vizinhança entre os *tiles* limita o número de iterações que podem ser computadas consecutivamente sem a necessidade de realizar comunicações entre *tiles*.

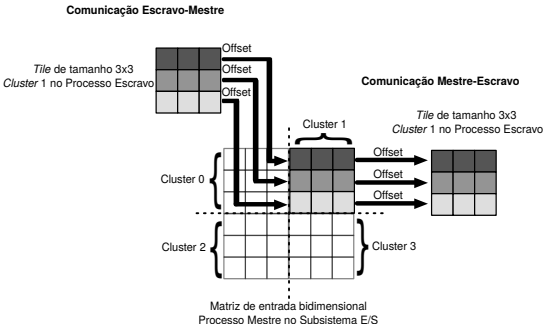


Figura 3. Exemplo do funcionamento do método *strides* do MPPA-256.

Além disso, devido à restrição da API e NoC no MPPA-256, os dados armazenados em cada *tile* precisam ser contíguos para serem transferidos pela NoC. A fim de se evitar cópias locais de dados, o que desperdiçaria memória e tempo de processamento, utiliza-se o conceito de comunicação por *strides*. Cada *stride* é uma parte contígua do `Array` original, sendo determinado por deslocamentos (*offsets*) especificados durante a execução. Então, cada *stride* é enviado para o `Array` de entrada em um processo escravo na posição determinada de acordo com o tamanho dos *tiles* e do `Array` original. O método de *strides* possibilita a definição de algumas variáveis para o gerenciamento do `Array`, sendo a mais importante os *offsets* que serão utilizados. A partir deles, o método irá efetuar a comunicação de maneira direta para o`Array` destino. Como dito anteriormente, a utilização de *tiles* aumentados permite reduzir a quantidade de comunicações necessárias entre os processos mestre e escravos. Fazendo o aumento dos *tiles* em uma dimensão temporal, os processos escravos podem executar múltiplas iterações sem a necessidade de comunicação ou sincronização com o processo mestre.

O escalonamento dos *tiles* nos *clusters* é feito de maneira circular (*round-robin*). Devido a isso, alguns *clusters* podem receber mais *tiles* que outros, dependendo do número de *tiles* e *clusters* usados na computação. Toda a comunicação entre o mestre e os escravos é feita utilizando-se a API de comunicação assíncrona oferecida pelo processador MPPA-256. A comunicação com cada processo escravo é feita de forma individual, mapeando diretamente áreas contíguas de memória de seus respectivos *tiles*. Além disso, a implementação atual permite a execução de aplicações estêncil iterativas. Nesse caso, o escalonamento dos *tiles* e a computação dos mesmos pelos *clusters* é repetida a cada iteração da aplicação.

Cada processo escravo realiza a computação do *tile* recebido no *cluster* utilizando o *kernel* de computação estêncil definido pelo usuário. A paralelização da computação dentro do *cluster* é feita com auxílio da API OpenMP. Em cada *cluster* podem ser criadas até 16 *threads* (uma para cada PE), onde cada uma é responsável por executar o *kernel* estêncil em um subconjunto de elementos dos *tiles*. Quando a computação do *kernel* estêncil é finalizada, os *tiles* resultantes são enviados pelos processos escravos para o processo mestre, onde são agrupados em um único `Array`, constituindo o resultado final da computação estêncil em uma iteração. Tendo em vista que os *tiles* são regiões contíguas na memória, processos escravos precisam gerenciar os *offsets* de dados para escrevê-los nas posições corretas no `Array` no processo mestre. Esse gerenciamento é efetuado pela

API do MPPA-256, mais especificamente, pelo método de *strides* mencionado anteriormente. A Figura 3 ilustra esse procedimento para o caso de um `Array2D`.

Para fornecer uma maior facilidade ao usuário, todas as tarefas complexas relacionadas com a técnica de *tiling*, comunicações NoC e adaptações discutidas nessa seção são abstraídas, pois elas são incluídas no *back-end* do PSkel. Isso significa que aplicações desenvolvidas com o *framework* PSkel podem executar no MPPA-256 sem a necessidade de modificações no código fonte.

4. Resultados Experimentais

Nesta seção será avaliado o desempenho e o consumo energético de aplicações estêncil do PSkel quando executadas no MPPA-256 e em um processador Intel Xeon E5-2640 v4 com 10 núcleos de 2.4GHz (Broadwell). As medições de energia no MPPA-256 foram feitas considerando todos os *clusters*, a memória, os subsistemas de E/S e a NoC, onde foram coletadas por meio dos sensores de potência e energia disponíveis no MPPA-256. Como o MPPA-256 possui características intrínsecas do próprio processador que garante baixa variabilidade entre as execuções, foram realizadas somente 5 repetições de cada experimento, computando-se a média aritmética dos valores. Todos os experimentos no MPPA-256 consideraram 16 PEs por *cluster*. Por outro lado, as medições de consumo de energia foram feitas no processador Intel com uso do *Running Average Power Limit* (RAPL) através da biblioteca PAPI [Weaver et al. 2012]. Em cada experimento foram utilizadas 10 *threads* (uma *thread* por núcleo) sem uso de *hyperthreading*. Cada experimento foi repetido 30 vezes e a média aritmética dos resultados foi calculada. Todos os resultados (MPPA-256 e Intel) apresentaram um desvio-padrão menor que 1%.

É possível reduzir a quantidade de sincronizações e comunicações realizadas na solução para o MPPA-256 de duas maneiras: aumentando o tamanho dos *tiles* ou aumentando a quantidade de iterações sobre cada *tile*. Como podemos ver na Figura 2, ao aumentarmos a quantidade de iterações sobre o *tile*, precisamos aumentar o *tile* lógico, formando um *tile* aumentado que será enviado para o *cluster*. Desta forma, devido às limitações de memória em cada *cluster*, ao ser especificado uma quantidade muito grande de iterações, o *tile* aumentado enviado para o escravo pode ser maior que o limite de memória de cada *cluster*. Portanto, foi fixado uma quantidade de 10 iterações sobre cada *tile*. Além disso, para os experimentos terem uma quantidade significativa de sincronizações sobre aplicações estêncil iterativas, foi adotado 30 iterações para cada aplicação.

4.1. Aplicações Estêncil

Para a realização dos experimentos foram utilizadas as seguintes aplicações estêncil:

Fur: modela a formação de padrões sobre a pele de animais[1]. Nessa aplicação, a pele do animal é modelada por uma *array* bidimensional de células de pigmento que podem estar em um dos dois estados: colorida ou não-colorida. As células coloridas secretam ativadores e inibidores. Ativadores fazem uma célula central se tornar colorida; inibidores, por outro lado, fazem uma célula central se tornar não colorida. A diferença entre as potências dos ativadores e inibidores é responsável por decidir a coloração da célula central, onde mais ativadores resulta em uma célula colorida e mais inibidores resulta em uma célula não colorida. Nos casos em que as potências dos ativadores e inibidores forem iguais, a cor da célula permanece inalterada. A máscara contém células adjacentes à célula central e seu tamanho é parametrizável. Neste trabalho foi utilizado 2 vizinhos adjacentes em cada direção.

[1] http://ccl.northwestern.edu/netlogo/models/Fur

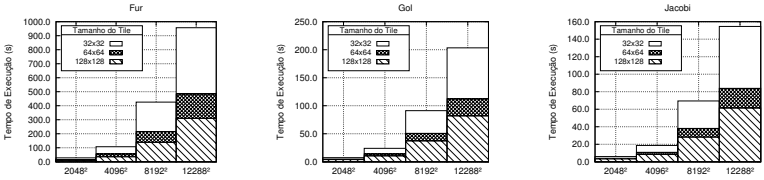


Figura 4. Tempos de execução das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

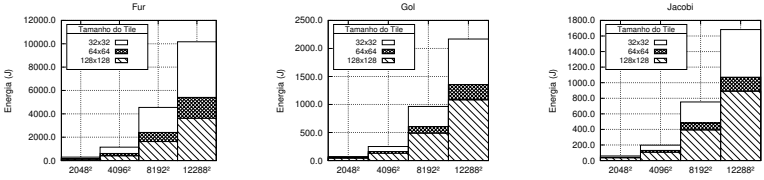


Figura 5. Consumo de energia das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

Jacobi: método iterativo para resolver equações matriciais [Demmel 1997]. O método converge garantidamente se a matriz de entrada é restrita ou irredutívelmente dominante diagonalmente, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, para todo i. A Equação 1 define a computação em cada passo do método iterativo de Jacobi para resolver a equação discreta elíptica de Poisson [Demmel 1997]. A solução aproximada é computada discretizando o problema na matriz em pontos espaçados de forma equivalente por $n \times n$.

$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \quad (1)$$

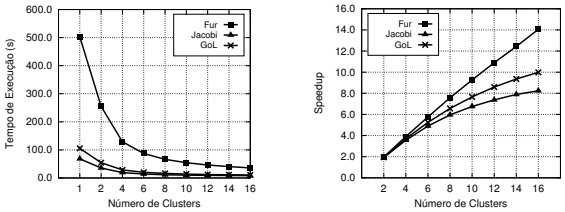
A cada passo, o novo valor de $u_{i,j}$ é obtida fazendo a média $h^2 f_{i,j}$ dos seus vizinhos, onde $h = \frac{1}{n+1}$ e $f_{i,j} = f(ih, jh)$, para uma dada função f.

GoL: autômato celular que implementa o Jogo da Vida de Conway [Gardner 1970]. O autômato é representado por um *array* bidimensional, onde cada elemento representa um indivíduo vivo ou um indivíduo morto. A máscara do estêncil, a qual determina a interação entre o indivíduo e seus vizinhos, considera as 8 células vizinhas adjacentes à célula central. Dependendo dos valores dos vizinhos, o elemento pode modificar seu estado entre vivo e morto.

4.2. Impacto do Tamanho do *Tile* no Desempenho do MPPA-256

O primeiro experimento tem por objetivo verificar o impacto do tamanho dos *tiles* no desempenho e consumo energético das aplicações. As Figuras 4 e 5 mostram, respectivamente, os tempos de execução e consumo de energia de três aplicações estêncil, variando-se o tamanho do `Array2D` de entrada (de 2048^2 até 12288^2) e os tamanhos do *tile* (de 32^2 até 128^2). *Arrays* de entrada maiores que 12288^2 e *tiles* maiores que 128^2 extrapolam às memórias LPDDR3 e dos *clusters*, respectivamente.

Pode-se perceber uma redução no tempo de execução à medida em que se aumenta o tamanho do *tile* (Figura 4), pois há menos sincronizações e comunicações de



(a) Tempo de execução. (b) Speedup.

Figura 6. Resultados de tempo e *speedup* das aplicações *Fur*, *GoL* e *Jacobi*.

tiles entre os processos mestre e escravos. O comportamento das aplicações é similar, sendo diferenciado apenas pela grandeza dos tempos de execução. A Figura 5 apresenta um comportamento similar para o consumo de energia, pois o tempo de execução reduz com o aumento do tamanho do *tile*, trazendo uma redução no consumo de energia.

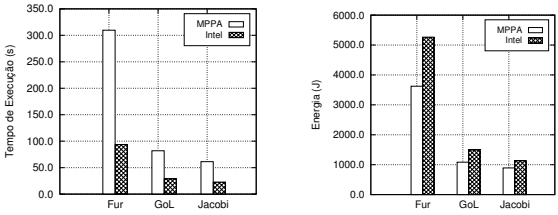
4.3. Análise de Escalabilidade no MPPA-256

Em um segundo experimento, buscou-se verificar a escalabilidade das aplicações no MPPA-256. Para isso, variou-se o número de *clusters* em cada aplicação, com `Array2D` de entrada fixo de tamanho 4096^2 e *tiles* de tamanho 128^2. A Figura 6(a) apresenta os tempos de execução obtidos ao variar-se o número de *clusters* utilizados na computação. A Figura 6(b), por outro lado, apresenta o fator de aceleração (*speedup*) com relação ao tempo de execução com 1 *cluster*. Em outras palavras, o *speedup* com c *clusters* é computado dividindo-se o tempo de execução obtido com apenas 1 *cluster* pelo tempo de execução obtido com c *clusters*.

No geral, os resultados mostraram que a solução proposta para o MPPA-256 é escalável. Porém, pode-se notar que a aplicação *Fur* apresentou uma escalabilidade superior às demais aplicações. Esse comportamento está diretamente relacionado com a quantidade de operações realizadas pelo *kernel* da aplicação (complexidade do *kernel*). Tendo em vista a necessidade de comunicações no MPPA-256, o tempo total de execução de uma aplicação passa a ser composto pela soma do tempo de comunicação com o tempo de computação. Para um dado *tile* t de tamanho fixo, o tempo necessário para realizar comunicações de t entre mestre e escravo será constante. Por outro lado, quanto maior o número de operações (computações) feitas em t pelo *kernel* da aplicação, maior será o paralelismo a ser explorado. Nesse caso, o tempo de computação será proporcionalmente maior que o tempo de comunicação, melhorando assim a escalabilidade obtida. Este é o caso da aplicação *Fur* cujo *speedup* se aproxima do caso ideal. Em contrapartida, aplicação *Jacobi* apresentou uma escalabilidade mais baixa que as demais, pois seu *kernel* apresenta baixa complexidade.

4.4. Kalray MPPA-256 vs. Intel Broadwell

Por fim, foram efetuados experimentos comparativos entre o processador Intel Xeon e o MPPA-256, como pode-se ver na Figura 7. Nesses experimentos, utilizou-se um `Array2D` de entrada de tamanho 12288^2 e *tiles* de tamanho 128^2. Ao ser comparado o tempo das aplicações em cada arquitetura, nota-se que o MPPA-256 tem um desempenho pior, contudo ao ser comparado o consumo de energia percebe-se um comportamento diferente: a energia consumida pelo MPPA-256 é menor, principalmente na aplicação *Fur*. No geral, o consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* no MPPA-256 foi



(a) Tempo de execução. (b) Consumo de Energia.

Figura 7. Comparação do tempo de execução e consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* em relação a arquitetura.

aproximadamente 1.45x, 1.38x e 1.27x menor que no Intel Xeon, respectivamente. Por outro lado, o tempo de execução dessas aplicações obtido no MPPA-256 foi 3.30x, 2.83x e 2.69x maior que no Intel Xeon, respectivamente.

5. Conclusão

Neste artigo foi proposta uma adaptação de um *framework* para desenvolvimento de aplicações estêncil iterativas, denominado PSkel, para processador MPPA-256. A solução proposta permite esconder detalhes de baixo nível do MPPA-256, simplificando significativamente o desenvolvimento de aplicações estêncil nesse processador. Os resultados mostraram que a solução proposta apresenta boa escalabilidade. Além disso, foi observado uma redução significativa no tempo de execução e no consumo de energia das aplicações no MPPA-256 ao se utilizar a técnica de *tiling* trapezoidal. Isso se deve, principalmente, à redução do sobrecusto de comunicações e sincronizações de *tiles*.

A aplicação *Fur* apresentou os melhores resultados de escalabilidade dentre as 3 aplicações estudas, obtendo um *speedup* de 14x em relação à apenas um *cluster*. Analisando experimentos executados sobre a adaptação pôde-se perceber uma relação entre a quantidade de computação realizada pelo *kernel* da aplicação e o *speedup* obtido. Por fim, experimentos comparativos entre o MPPA-256 e o processador Intel Broadwell mostraram que a solução proposta para o MPPA-256 apresenta uma eficiência energética superior apesar de um tempo de execução superior.

Como trabalhos futuros, pretende-se estudar formas de reduzir ainda mais os sobrecustos de comunicação através do uso de técnicas de *sofware prefetching*. Além disso, pretende-se realizar experimentos com outros *benchmarks* e aplicações que utilizam estruturas tridimensionais. Por fim, pretende-se realizar comparações de desempenho e consumo de energia com outros processadores embarcados.

Referências

Castro, M., Dupros, F., Francesquini, E., Méhaut, J.-F., and Navaux, P. O. A. (2014). Energy efficient seismic wave propagation simulation on a low-power manycore processor. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 57–64, Paris, France. IEEE Computer Society.

Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120.

Castro, M., Francesquini, E., Nguélé, T. M., and Méhaut, J.-F. (2013). Analysis of computing and energy performance of multicore, NUMA, and manycore platforms for an irregular application. In *Workshop on Irregular Applications: Architectures & Algorithms (IAˆ3)*, pages 5:1–5:8, Denver, EUA. ACM.

Demmel, J. W. (1997). *Applied numerical linear algebra*. SIAM.

Enmyren, J. and Kessler, C. W. (2010). SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA. ACM.

Francesquini, E., Castro, M., Penna, P. H., Dupros, F., de Freitas, H. C., Navaux, P. O. A., and Méhaut, J.-F. (2014). On the energy efficiency and performance of irregular applications on multicore, NUMA and manycore platforms. *J. Parallel Distrib. Comput.*, 76:32–48.

Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., and Yang, G. (2016). The sunway taihulight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, 59(7):072001:1–072001:16.

Gardner, M. (1970). Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, 223(3).

Gonzalez, R. C. and Woods, R. E. (2006). *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc.

Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ACM ICS*, pages 311–320.

Lutz, T., Fensch, C., and Cole, M. (2013). PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24.

Meng, J. and Skadron, K. (2011). A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, 39(1):115–142.

Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). PSkel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27(17):4938–4953.

Rocha, R. C. O., Pereira, A. D., Ramos, L., and Góes, L. F. W. (2017). TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience*, 29(8):e4053.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA. IEEE Computer Society.

Varghese, A., Edwards, B., Mitra, G., and Rendell, A. P. (2014). Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In *International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pages 984–992, Phoenix, USA. IEEE Computer Society.

Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., and Moore, S. (2012). Measuring energy and power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*, pages 262–268.

ANEXO C – Código Fonte

Código 9 – PSkelArray.h

```
1  #ifndef PSKEL_ARRAY_H
2  #define PSKEL_ARRAY_H
3
4  #ifdef PSKEL_CUDA
5  #include <cuda.h>
6  #endif
7
8  #ifdef PSKEL_MPPA
9  #include "common.h"
10 //#include "interface_mppa.h"
11 #endif
12
13 #include "PSkelDefs.h"
14
15 namespace PSkel{
16
17 /**
18  * Class that implements the basic data structure used by the
        parallel
19  * skeletons (such as stencil and map.) PSkel::ArrayBase is a 3D
        array that is also
20  * interfaced via 1D and 2D arrays.  The PSkel::ArrayBase data
        structure that
21  * is extended by PSkel::Array, PSkel::Array2D, and PSkel::
        Array3D.
22  **/
23 template<typename T>
24 class ArrayBase{
25 private:
26     //variables that hold the real boundaries (total allocated
          data.)
27     size_t realWidth, realHeight, realDepth;
28     //variables that hold the "virtual" array boundaries (
          important for sliced arrays.)
29     size_t width, height,depth;
30     //offsets for the sliced array.
31     size_t widthOffset, heightOffset, depthOffset;
32     //host and device (GPU memory) pointers
33     T *hostArray;
34     #ifdef PSKEL_CUDA
35     T *deviceArray;
36     #endif
37     #ifdef PSKEL_MPPA
38     T *mppaArray;
39     portal_t *write_portal;
40     portal_t *read_portal;
41     portal_t *aux_write_portal;
42     portal_t *aux_read_portal;
43     int* aux;
44     //T *comm_buffer;
45     #endif
46
47 protected:
48     #ifdef PSKEL_CUDA
49     /**
50          * Access a specific element of the array allocated in
               the device memory.
51          * This function is accessible only during the execution
                in the device environment.
```

```
52          * \param[in] h height offset for the element being
                accessed.
53          * \param[in] w width offset for the element being
                accessed.
54          * \param[in] d depth offset for the element being
                accessed.
55          * \return the reference of the element specified via
                parameters.
56          **/
57      __device__ __forceinline__ T & deviceGet(size_t h,size_t w,
            size_t d) const ;
58      #endif
59
60      /**
61          * Access a specific element of the array allocated in
                the host memory.
62          * This function is accessible only during the execution
                 in the host environment.
63          * \param[in] h height offset for the element being
                accessed.
64          * \param[in] w width offset for the element being
                accessed.
65          * \param[in] d depth offset for the element being
                accessed.
66          * \return the reference of the element specified via
                parameters.
67          **/
68      __host__ __forceinline__ T & hostGet(size_t h,size_t w,
            size_t d) const ;
69
70      /**
71          * The ArrayBase constructor creates and allocates the
                specified array
72          * in the host memory.
73          * \param[in] width Width for the 3D array being created
                .
74          * \param[in] height Height for the 3D array being
                created.
75          * \param[in] depth Depth for the 3D array being created
                .
76          **/
77      ArrayBase(size_t width, size_t height, size_t depth);
78  public:
79      #ifdef PSKEL_CUDA
80      /**
81       * Allocates the "virtual" array in device memory.
82       **/
83      void deviceAlloc();
84      #endif
85
86      #ifdef PSKEL_CUDA
87      /**
88       * Frees the allocated device memory.
89       **/
90      void deviceFree();
91      #endif
92
93
94      #ifdef PSKEL_MPPA
95      void portalReadAlloc(int trigger, int nb_cluster);
96      #endif
```

```
97
98      #ifdef PSKEL_MPPA
99      void portalWriteAlloc(int nb_cluster);
100     #endif
101
102     #ifdef PSKEL_MPPA
103     void portalAuxWriteAlloc(int nb_cluster);
104     #endif
105
106     #ifdef PSKEL_MPPA
107     void portalAuxReadAlloc(int trigger, int nb_cluster);
108     #endif
109
110     #ifdef PSKEL_MPPA
111     void mppaAlloc(size_t width, size_t height, size_t depth);
112     #endif
113
114     void hostAlloc(size_t width, size_t height, size_t depth);
115
116     /**
117      * Allocates the array in host (main) memory.
118      **/
119     void hostAlloc();
120
121     /**
122      * Frees the allocated host (main) memory.
123      **/
124     void hostFree();
125
126     /**
127      * Get the width size of the "virtual" array.
128      * \return the "virtual" width of the array data structure.
129      **/
130     __device__ __host__ size_t getWidth() const;
131
132     /**
133      * Get the height size of the "virtual" array.
134      * \return the "virtual" height of the array data structure.
135      **/
136     __device__ __host__ size_t getHeight() const;
137
138     /**
139      * Get the depth size of the "virtual" array.
140      * \return the "virtual" depth of the array data structure.
141      **/
142     __device__ __host__ size_t getDepth() const;
143
144     /**
145      * Get the size, in bytes, of the allocated memory for the "
           virtual" array.
146      * \return the total of bytes allocated in memory for the "
           virtual" array.
147      **/
148     __device__ __host__ size_t memSize() const;
149
150     /**
151      * Get the size of the "virtual" array, i.e. the number of
           elements
152      * \return the size of the "virtual" array.
153      **/
154     __device__ __host__ size_t size() const;
```

```
155
156     /**
157      * Get the size of the real allocated array, i.e. the number
             of elements
158      * \return the size of the real allocated array.
159      **/
160     __device__ __host__ size_t realSize() const;
161
162     /**
163      * Creates a sliced reference, in the host (main) memory, of
             the array given as argument.
164      * The slice points to the same memory space as the sliced
             array.
165      * \param[in] array original array that will be sliced.
166      * \param[in] widthOffset the width offset for the sliced
             region, relative to the array given as argument.
167      * \param[in] heightOffset the height offset for the sliced
             region, relative to the array given as argument.
168      * \param[in] depthOffset the depth offset for the sliced
             region, relative to the array given as argument.
169      * \param[in] width the width of the slice.
170      * \param[in] height the height of the slice.
171      * \param[in] depth the depth of the slice.
172      **/
173     template<typename Arrays>
174     void hostSlice(Arrays array, size_t widthOffset, size_t
             heightOffset, size_t depthOffset, size_t width, size_t
             height, size_t depth);
175
176     /**
177      * Creates a clone, in the host (main) memory, of the array
             given as argument.
178      * The clone is a copy of the array in a different memory
             space.
179      * \param[in] array original array that will be cloned.
180      **/
181     template<typename Arrays>
182     void hostClone(Arrays array);
183
184     template<typename Arrays>
185     void mppaClone(Arrays array);
186
187
188     template<typename Arrays>
189     void mppaMasterClone(Arrays array);
190
191     //template<typename Arrays>
192     //void offsetMppaMasterCopy(Arrays array, int heightOffset,
             int widthOffset, int tilingHeight, int tilingWidth);
193
194     /**
195      * Copies the data, in the host (main) memory, from the
             array given as argument.
196      * \param[in] array original array that will be copied.
197      **/
198     template<typename Arrays>
199     void hostMemCopy(Arrays array);
200
201     template<typename Arrays>
202     void mppaMasterCopy(Arrays array);
203
```

```
204     template<typename Arrays>
205     void mppaMemCopy(Arrays array);
206
207     #ifdef PSKEL_CUDA
208     /**
209      * The array is copied from the host allocated memory to the
             device allocated memory.
210      * The data is efficiently transferred from host to device.
211      * Both the host and device memory must be allocated before
             the data is transferred.
212      **/
213     void copyToDevice();
214     #endif
215
216     #ifdef PSKEL_CUDA
217     /**
218      * The array given as argument is copied from the device
             allocated memory to the host allocated memory of this
             array.
219      * The data is efficiently transferred from device to host.
220      * \param[in] array the source array that holds the data
             that will be copied from device to the host memory of
             this array.
221      **/
222     template<typename Arrays>
223     void copyFromDevice(Arrays array);
224     #endif
225
226     #ifdef PSKEL_MPPA
227     void mppaAlloc();
228     #endif
229
230     #ifdef PSKEL_MPPA
231     void mppaFree();
232     #endif
233
234     #ifdef PSKEL_MPPA
235     void auxFree();
236     #endif
237
238     #ifdef PSKEL_MPPA
239     void setTrigger(int trigger);
240     #endif
241
242     #ifdef PSKEL_MPPA
243     int* getAux();
244     #endif
245
246     #ifdef PSKEL_MPPA
247     void auxAlloc();
248     #endif
249
250     #ifdef PSKEL_MPPA
251     void setAux(int heightOffset, int widthOffset, int it, int
             subIterations, size_t coreWidthOffset, size_t
             coreHeightOffset, size_t coreDepthOffset, size_t
             coreWidth, size_t coreHeight, size_t coreDepth, int
             outterIterations, size_t height, size_t width, size_t
             depth, int baseWidth, int baseHeight);
252     #endif
253
```

```
254     #ifdef PSKEL_MPPA
255     void copyToAux();
256     #endif
257
258     #ifdef PSKEL_MPPA
259     void copyFromAux();
260     #endif
261
262     #ifdef PSKEL_MPPA
263     void copyToo(size_t offsetSlave, size_t offsetMaster, int
            tam);
264     #endif
265
266     #ifdef PSKEL_MPPA
267     void copyTo(int coreHeight,int coreWidth, size_t sJump,
            size_t tJump, size_t sOffset, size_t tOffset);
268     #endif
269
270     #ifdef PSKEL_MPPA
271     void copyTo();
272     #endif
273
274
275     #ifdef PSKEL_MPPA
276     void copyFrom();
277     #endif
278
279     #ifdef PSKEL_MPPA
280     void waitRead();
281     #endif
282
283     #ifdef PSKEL_MPPA
284     void waitWrite();
285     #endif
286
287     #ifdef PSKEL_MPPA
288     void waitAuxWrite();
289     #endif
290
291     #ifdef PSKEL_MPPA
292     void closeReadPortal();
293     #endif
294
295     #ifdef PSKEL_MPPA
296     void closeAuxReadPortal();
297     #endif
298
299     #ifdef PSKEL_MPPA
300     void closeAuxWritePortal();
301     #endif
302
303     #ifdef PSKEL_MPPA
304     void closeWritePortal();
305     #endif
306
307     #ifdef PSKEL_MPPA
308     __host__ __forceinline__ T & mppaGet(size_t h,size_t w,
            size_t d) const ;
309     #endif
310
311     #ifdef PSKEL_CUDA
```

```
312     /**
313      * The array is copied from the device allocated memory to
             the host allocated memory.
314      * The data is efficiently transferred from device to host.
315      * Both the host and device memory must be allocated before
             the data is transferred.
316      **/
317     void copyToHost();
318     #endif
319
320     /**
321      * Verifies if there is memory allocated for the array data
             structure.
322      * This function can be called both from device and host
             environment,
323      * and the respective memory space is verified.
324      * \return true if there is a valid memory spaced allocated
             for the array; false otherwise.
325      **/
326     __device__ __host__ operator bool() const ;
327 };
328
329 //*****************************************************
330 // Array 3D
331 //*****************************************************
332
333 template <typename T>
334 class Array3D: public ArrayBase<T>{
335 public:
336     /**
337          * The Array3D default constructor creates an empty
                 array withour allocating memory space.
338          **/
339     Array3D();
340
341     /*
342     ~Array3D(){
343         free(hostArray);
344         cudaFree(deviceArray);
345     }*/
346
347     /**
348          * The Array3D constructor creates and allocates the
                 specified 3-dimensional array
349          * in the host memory.
350          * \param[in] width width for the 3D array being created
                 .
351          * \param[in] height height for the 3D array being
                 created.
352          * \param[in] depth depth for the 3D array being created
                 .
353          **/
354     Array3D(size_t width, size_t height, size_t depth);
355
356     /**
357          * Access a specific element of the array allocated in
                 the memory space
358          * relative to the execution environment, i.e. either in
                  the host or device memory.
359          * \param[in] h height offset for the element being
                 accessed.
```

```
360           * \param[in] w width offset for the element being
                  accessed.
361           * \param[in] d depth offset for the element being
                  accessed.
362           * \return the reference of the element specified via
                  parameters.
363           **/
364       __attribute__((always_inline)) __forceinline__ __device__
              __host__ T & operator()(size_t h,size_t w,size_t d)
              const ;
365
366   };
367
368   //*************************************************
369   // Array 2D
370   //*************************************************
371
372   template<typename T>
373   class Array2D: public ArrayBase<T>{
374   public:
375       /**
376           * The Array2D default constructor creates an empty
                  array withour allocating memory space.
377           **/
378       Array2D();
379
380
381       //~Array2D();
382       /**
383           * The Array2D constructor creates and allocates the
                  specified 2-dimensional array
384           * in the host memory.
385           * \param[in] width width for the 2D array being created
                  .
386           * \param[in] height height for the 2D array being
                  created.
387           **/
388       Array2D(size_t width, size_t height);
389
390       /**
391           * Access a specific element of the array allocated in
                  the memory space
392           * relative to the execution environment, i.e. either in
                   the host or device memory.
393           * \param[in] h height offset for the element being
                  accessed.
394           * \param[in] w width offset for the element being
                  accessed.
395           * \return the reference of the element specified via
                  parameters.
396           **/
397       __attribute__((always_inline)) __forceinline__ __device__
              __host__ T & operator()(size_t h, size_t w) const ;
398
399   };
400
401   //*************************************************
402   // Array 1D
403   //*************************************************
404
405   template<typename T>
```

```
406 class Array: public ArrayBase<T>{
407 public:
408     /**
409          * The Array default constructor creates an empty array
                withour allocating memory space.
410          **/
411     Array();
412
413     /**
414          * The Array constructor creates and allocates the
                specified 1-dimensional array
415          * in the host memory.
416          * \param[in] size size for the 1D array being created.
417          **/
418     Array(size_t size);
419
420     /**
421          * Access a specific element of the array allocated in
                the memory space
422          * relative to the execution environment, i.e. either in
                the host or device memory.
423          * \param[in] w offset for the element being accessed.
424          * \return the reference of the element specified via
                parameters.
425          **/
426     __attribute__((always_inline)) __forceinline__ __device__
          __host__ T & operator()(size_t w) const;
427 };
428
429 }//end namespace
430
431 #include "PSkelArray.hpp"
432
433 #endif
```

Código 10 – PSkelArray.hpp

```
1  #ifndef PSKEL_ARRAY_HPP
2  #define PSKEL_ARRAY_HPP
3  #include <cstring>
4  #include <iostream>
5
6  #ifndef MPPA_MASTER
7  #include <omp.h>
8  #endif
9
10 // Maximum size of the cluster
11 #define KB 1024
12 #define MB 1024 * KB
13 // maximum size of a message
14 #define MAX_CLUSTER_SIZE 1 * MB + MB / 2
15
16 //Change how to read and write in the array for later use for a
       bunch of clusters
17 namespace PSkel{
18
19 template<typename T>
20 ArrayBase<T>::ArrayBase(size_t width, size_t height, size_t
       depth){
21     this->width = width;
22     this->height = height;
```

```
23      this->depth = depth;
24      this->realWidth = width;
25      this->realHeight = height;
26      this->realDepth = depth;
27      this->widthOffset = 0;
28      this->heightOffset = 0;
29      this->depthOffset = 0;
30      this->hostArray = 0;
31      #ifdef PSKEL_CUDA
32          this->deviceArray = 0;
33      #endif
34      #ifdef PSKEL_MPPA
35      this->mppaArray = 0;
36      this->write_portal = 0;
37      this->read_portal = 0;
38      this->aux_write_portal = 0;
39      this->aux_read_portal = 0;
40      this->aux = (int *) malloc(sizeof(size_t*) * 16);
41      if(size()>0) this->mppaAlloc();
42      #else
43          if(size()>0) this->hostAlloc();
44      #endif
45  }
46
47  template<typename T>
48  void ArrayBase<T>::hostAlloc(size_t width, size_t height, size_t
        depth){
49      this->width = width;
50      this->height = height;
51      this->depth = depth;
52      this->realWidth = width;
53      this->realHeight = height;
54      this->realDepth = depth;
55      this->widthOffset = 0;
56      this->heightOffset = 0;
57      this->depthOffset = 0;
58      this->hostArray = NULL;
59      #ifdef PSKEL_CUDA
60      this->deviceArray = NULL;
61      #endif
62
63      this->hostAlloc();
64  }
65
66  #ifdef PSKEL_MPPA
67  template<typename T>
68  void ArrayBase<T>::mppaAlloc(){
69      if(this->mppaArray==NULL){
70          this->mppaArray = (T*) calloc(this->size(), sizeof(T));
71          assert(this->mppaArray!= NULL);
72          #if defined (DEBUG) || defined (BUG_TEST)
73              #ifdef MPPA_MASTER
74                  std::cout<<"MASTER: Allocating "<<this->size()*
                        sizeof(T)<<" bytes starting on address "<<&(
                        this->mppaArray)<<std::endl;
75              #else
76                  std::cout<<"SLAVE: Allocating "<<this->size()*
                        sizeof(T)<<" bytes starting on address "<<&(
                        this->mppaArray)<<std::endl;
77              #endif
78          #endif
```

```
79      }
80  }
81  #endif
82
83  #ifdef PSKEL_MPPA
84  template<typename T>
85  void ArrayBase<T>::mppaAlloc(size_t width, size_t height, size_t
        depth){
86      this->width = width;
87      this->height = height;
88      this->depth = depth;
89      this->realWidth = width;
90      this->realHeight = height;
91      this->realDepth = depth;
92      this->widthOffset = 0;
93      this->heightOffset = 0;
94      this->depthOffset = 0;
95
96      this->mppaArray = NULL;
97      this->mppaAlloc();
98  }
99  #endif
100
101 #ifdef PSKEL_MPPA
102 template<typename T>
103 void ArrayBase<T>::mppaFree(){
104     //if(this->mppaArray!=NULL){
105     #if defined (DEBUG) || defined (BUG_TEST)
106         #ifdef MPPA_MASTER
107             std::cout<<"MASTER: Deallocating "<<this->size()*
                    sizeof(T)<<" bytes of address "<<&(this->
                    mppaArray)<<std::endl;
108         #else
109             std::cout<<"SLAVE: Deallocating "<<this->size()*
                    sizeof(T)<<" bytes of address "<<&(this->
                    mppaArray)<<std::endl;
110         #endif
111     #endif
112     free(this->mppaArray);
113     this->mppaArray = NULL;
114     //}
115 }
116 #endif
117
118 #ifdef PSKEL_MPPA
119 template<typename T>
120 void ArrayBase<T>::auxFree(){
121
122     free(this->aux);
123     this->aux = NULL;
124 }
125 #endif
126
127 template<typename T>
128 void ArrayBase<T>::hostAlloc(){
129     if(this->hostArray==NULL){
130         this->hostArray = (T*) calloc(size(), sizeof(T));
131         assert(this->hostArray != NULL);
132         //gpuErrchk( cudaMallocHost((void**)&hostArray, size()*
                sizeof(T)) );
133         //memset(this->hostArray, 0, size()*sizeof(T));
```

```
134     }
135 }
136
137 template<typename T>
138 void ArrayBase<T>::hostFree(){
139
140     free(this->hostArray);
141
142     this->hostArray = NULL;
143
144 }
145
146 template<typename T>
147 size_t ArrayBase<T>::getWidth() const{
148     return width;
149 }
150
151 template<typename T>
152 size_t ArrayBase<T>::getHeight() const{
153     return height;
154 }
155
156 template<typename T>
157 size_t ArrayBase<T>::getDepth() const{
158     return depth;
159 }
160
161 template<typename T>
162 size_t ArrayBase<T>::memSize() const{
163     return size()*sizeof(T);
164 }
165
166 template<typename T>
167 size_t ArrayBase<T>::size() const{
168     return height*width*depth;
169 }
170
171 template<typename T>
172 size_t ArrayBase<T>::realSize() const{
173     return realHeight*realWidth*realDepth;
174 }
175
176 template<typename T>
177 T & ArrayBase<T>::hostGet(size_t h, size_t w, size_t d) const {
178     return this->hostArray[ ((h+heightOffset)*realWidth + (w+
            widthOffset))*realDepth + (d+depthOffset) ];
179 }
180
181 #ifdef PSKEL_MPPA
182 template<typename T>
183 T& ArrayBase<T>::mppaGet(size_t h, size_t w, size_t d) const {
184     return this->mppaArray[ ((h+heightOffset)*realWidth + (w+
            widthOffset))*realDepth + (d+depthOffset) ];
185 }
186 #endif
187
188 template<typename T> template<typename Arrays>
189 void ArrayBase<T>::hostSlice(Arrays array, size_t widthOffset,
        size_t heightOffset, size_t depthOffset, size_t width,
        size_t height, size_t depth){
190     //maintain previous allocated area
```

```
191     #ifdef PSKEL_CUDA
192     if(this->deviceArray!=NULL){
193         if(this->size()!=(width*height*depth)){
194             this->deviceFree();
195             this->deviceArray = NULL;
196         }
197     }
198     #endif
199
200     this->width = width;
201     this->height = height;
202     this->depth = depth;
203     this->widthOffset = array.widthOffset+widthOffset;
204     this->heightOffset = array.heightOffset+heightOffset;
205     this->depthOffset = array.depthOffset+depthOffset;
206     this->realWidth = array.realWidth;
207     this->realHeight = array.realHeight;
208     this->realDepth = array.realDepth;
209     std::cout << "Print barrier for management: " << std::endl;
210     // std::cout << "ArrayWidthOffset: " << array.widthOffset <<
            "ArrayHeightOffset: " << array.heightOffset <<std::endl
            ;
211     #ifdef MPPA_MASTER
212
213     #endif
214     #ifndef PSKEL_MPPA
215     this->hostArray = array.hostArray;
216     #else
217     this->mppaArray = array.mppaArray;
218     #endif
219 }
220
221 template<typename T> template<typename Arrays>
222 void ArrayBase<T>::hostClone(Arrays array){
223     //Copy dimensions
224     this->width = array.width;
225     this->height = array.height;
226     this->depth = array.depth;
227     this->widthOffset = 0;
228     this->heightOffset = 0;
229     this->depthOffset = 0;
230     this->realWidth = array.width;
231     this->realHeight = array.height;
232     this->realDepth = array.depth;
233     //Alloc clone memory
234     this->hostArray = NULL;
235     this->hostAlloc();
236     //Copy clone memory
237     this->hostMemCopy(array);
238 }
239 #ifndef MPPA_MASTER
240 template<typename T> template<typename Arrays>
241 void ArrayBase<T>::mppaClone(Arrays array){
242     //Copy dimensions
243     this->width = array.width;
244     this->height = array.height;
245     this->depth = array.depth;
246     this->widthOffset = 0;
247     this->heightOffset = 0;
248     this->depthOffset = 0;
249     this->realWidth = array.width;
```

```
250     this->realHeight = array.height;
251     this->realDepth = array.depth;
252     //Alloc clone memory
253     this->mppaArray = NULL;
254     this->mppaAlloc();
255     //Copy clone memory
256     //#ifdef MPPA_SLAVE
257     this->mppaMemCopy(array);
258     //#endif
259 }
260 #endif
261
262 #ifdef PSKEL_MPPA
263 template<typename T> template<typename Arrays>
264 void ArrayBase<T>::mppaMasterClone(Arrays array){
265     //Copy dimensions
266     this->width = array.width;
267     this->height = array.height;
268     this->depth = array.depth;
269     this->widthOffset = 0;
270     this->heightOffset = 0;
271     this->depthOffset = 0;
272     this->realWidth = array.width;
273     this->realHeight = array.height;
274     this->realDepth = array.depth;
275     //Alloc clone memory
276     this->mppaArray = NULL;
277     this->mppaAlloc();
278     //Copy clone memory
279     //#ifdef MPPA_SLAVE
280     this->mppaMasterCopy(array);
281     //#endif
282 }
283 #endif
284
285
286 #ifdef PSKEL_MPPA
287 template<typename T>
288 void ArrayBase<T>::setTrigger(int trigger){
289     mppa_aiocb_set_trigger(&read_portal->aiocb, trigger);
290 }
291 #endif
292
293 #ifdef PSKEL_MPPA
294 template<typename T>
295 int* ArrayBase<T>::getAux(){
296     return this->aux;
297 }
298 #endif
299
300 // #ifdef PSKEL_MPPA
301 // template<typename T>
302 // void ArrayBase<T>::auxAlloc(){
303 //   this->aux = (int *) malloc(sizeof(size_t*) * 13);
304 //   assert(this->aux != NULL);
305 // }
306 // #endif
307
308 #ifdef PSKEL_MPPA
309 template<typename T>
310 void ArrayBase<T>::setAux(int heightOffset, int widthOffset, int
```

```
        it, int subIterations, size_t coreWidthOffset, size_t
        coreHeightOffset, size_t coreDepthOffset, size_t coreWidth,
        size_t coreHeight, size_t coreDepth, int outterIterations,
        size_t height, size_t width, size_t depth, int baseWidth,
        int baseHeight){
311
312     this->aux[0] = heightOffset;
313     this->aux[1] = it;
314     this->aux[2] = subIterations;
315     this->aux[3] = coreWidthOffset;
316     this->aux[4] = coreHeightOffset;
317     this->aux[5] = coreDepthOffset;
318     this->aux[6] = coreWidth;
319     this->aux[7] = coreHeight;
320     this->aux[8] = coreDepth;
321     this->aux[9] = outterIterations;
322     this->aux[10] = height;
323     this->aux[11] = width;
324     this->aux[12] = depth;
325     this->aux[13] = widthOffset;
326     this->aux[14] = baseWidth;
327     this->aux[15] = baseHeight;
328
329
330 }
331 #endif
332
333 #ifdef PSKEL_MPPA
334 template<typename T>
335 void ArrayBase<T>::copyToAux(){
336     mppa_async_write_portal(this->aux_write_portal, this->aux, (
            sizeof(int) * 16), 0);
337 }
338 #endif
339
340 #ifdef PSKEL_MPPA
341 template<typename T>
342 void ArrayBase<T>::copyFromAux(){
343     mppa_async_read_wait_portal(this->aux_read_portal);
344 }
345 #endif
346
347 #ifdef PSKEL_MPPA
348 template<typename T>
349 void ArrayBase<T>::copyToo(size_t offsetSlave, size_t
        offsetMaster, int tam){
350     tam = tam*sizeof(T);
351     T *mppaSlicePtr = (T*)(this->mppaArray) + size_t(offsetSlave
            *realDepth);
352     mppa_async_write_portal(this->write_portal, mppaSlicePtr,
            tam, sizeof(T)*offsetMaster);
353 }
354 #endif
355
356 #ifdef PSKEL_MPPA
357 template<typename T>
358 void ArrayBase<T>::copyTo(int coreHeight, int coreWidth, size_t
        sJump, size_t tJump, size_t sOffset, size_t tOffset){
359     // T *mppaSlicePtr = &((this->mppaArray) + size_t(sOffset*
            realDepth));
360     T *mppaSlicePtr = (T*)(this->mppaArray) + size_t(sOffset*
```

```
            realDepth);
361
362
363         int sstride = sJump*sizeof(T);
364         int tstride = tJump*sizeof(T);
365         int ecount = coreHeight;
366         int size = coreWidth*sizeof(T);
367
368         mppa_async_write_stride_portal(this->write_portal,
                mppaSlicePtr, size, ecount, sstride, tstride, sizeof(T)*
                tOffset);
369
370 }
371 #endif
372
373 #ifdef PSKEL_MPPA
374 template<typename T>
375 void ArrayBase<T>::copyTo(){
376     T *mppaSlicePtr = (T*)(this->mppaArray);
377     mppa_async_write_portal(this->write_portal, mppaSlicePtr,
            this->memSize(), 0);
378
379
380 }
381 #endif
382
383 #ifdef PSKEL_MPPA
384 template<typename T>
385 void ArrayBase<T>::copyFrom(){
386     mppa_async_read_wait_portal(this->read_portal);
387
388 }
389 #endif
390
391 #ifdef PSKEL_MPPA
392 template<typename T>
393 void ArrayBase<T>::portalReadAlloc(int trigger, int nb_cluster){
394     #ifdef MPPA_MASTER
395         char pathMaster[256];
396         sprintf(pathMaster, "/mppa/portal/%d:3", 128);
397         this->read_portal = mppa_create_read_portal(pathMaster,
                this->mppaArray, this->memSize(), trigger, NULL);
398     #endif
399     #ifdef MPPA_SLAVE
400     char pathSlave[25];
401     char path[25];
402     sprintf(pathSlave, "/mppa/portal/%d:%d", nb_cluster, 4 +
            nb_cluster);
403
404     this->read_portal = mppa_create_read_portal(pathSlave, this
            ->mppaArray, this->memSize(), trigger, NULL);
405     #endif
406 }
407 #endif
408
409
410 #ifdef PSKEL_MPPA
411 template<typename T>
412 void ArrayBase<T>::portalAuxWriteAlloc(int nb_cluster){
413     char path[256];
414     #ifdef MPPA_MASTER
```

```
415         sprintf(path, "/mppa/portal/%d:%d", nb_cluster, (21 +
                nb_cluster));
416         this->aux_write_portal = mppa_create_write_portal(path,
                NULL, 0, nb_cluster);
417     #endif
418 }
419 #endif
420
421 #ifdef PSKEL_MPPA
422 template<typename T>
423 void ArrayBase<T>::portalAuxReadAlloc(int trigger, int
        nb_cluster){
424     char path[25];
425     #ifdef MPPA_SLAVE
426         sprintf(path, "/mppa/portal/%d:%d", nb_cluster, (21 +
                nb_cluster));
427         this->aux_read_portal = mppa_create_read_portal(path,
                this->aux, (sizeof(int) * 16), trigger, NULL);
428     #endif
429 }
430 #endif
431
432
433 #ifdef PSKEL_MPPA
434 template<typename T>
435 void ArrayBase<T>::portalWriteAlloc(int nb_cluster){
436     char path[256];
437     #ifdef MPPA_MASTER
438         sprintf(path, "/mppa/portal/%d:%d", nb_cluster, 4 +
                nb_cluster);
439         this->write_portal = mppa_create_write_portal(path, NULL
                , 0, nb_cluster);
440     #endif
441     #ifdef MPPA_SLAVE
442         sprintf(path, "/mppa/portal/%d:3", 128);
443         this->write_portal = mppa_create_write_portal(path, NULL
                , 0, 128);
444     #endif
445 }
446 #endif
447
448 #ifdef PSKEL_MPPA
449 template<typename T>
450 void ArrayBase<T>::waitRead(){
451     mppa_async_read_wait_portal(this->read_portal);
452 }
453 #endif
454
455 #ifdef PSKEL_MPPA
456 template<typename T>
457 void ArrayBase<T>::waitWrite(){
458     mppa_async_write_wait_portal(this->write_portal);
459 }
460 #endif
461
462 #ifdef PSKEL_MPPA
463 template<typename T>
464 void ArrayBase<T>::waitAuxWrite(){
465     mppa_async_write_wait_portal(this->aux_write_portal);
466 }
467 #endif
```

```
468
469 #ifdef PSKEL_MPPA
470 template<typename T>
471 void ArrayBase<T>::closeReadPortal(){
472     mppa_close_portal(this->read_portal);
473 }
474 #endif
475
476 #ifdef PSKEL_MPPA
477 template<typename T>
478 void ArrayBase<T>::closeAuxReadPortal(){
479     mppa_close_portal(this->aux_read_portal);
480 }
481 #endif
482
483 #ifdef PSKEL_MPPA
484 template<typename T>
485 void ArrayBase<T>::closeAuxWritePortal(){
486     mppa_close_portal(this->aux_write_portal);
487 }
488 #endif
489
490 #ifdef PSKEL_MPPA
491 template<typename T>
492 void ArrayBase<T>::closeWritePortal(){
493     mppa_close_portal(this->write_portal);
494 }
495 #endif
496
497 #ifndef MPPA_MASTER
498 template<typename T> template<typename Arrays>
499 void ArrayBase<T>::hostMemCopy(Arrays array){
500     if(array.size()==array.realSize() && this->size()==this->
          realSize()){
501         memcpy(this->hostArray, array.hostArray, size()*sizeof(T
              ));
502     }else{
503         #pragma omp parallel for
504         for(size_t i = 0; i<height; ++i){
505         for(size_t j = 0; j<width; ++j){
506         for(size_t k = 0; k<depth; ++k){
507                         this->hostGet(i,j,k)=array.hostGet(i,j,k
                              );
508         }}}
509     }
510 }
511 #endif
512
513 template<typename T> template<typename Arrays>
514 void ArrayBase<T>::mppaMasterCopy(Arrays array){
515     if(array.size()==array.realSize() && this->size()==this->
          realSize()){
516         #ifdef DEBUG
517             std::cout<<"MASTER: Copying memory from address "
                  <<&(array.mppaArray)<<" to address "<<&(this->
                  mppaArray)<<std::endl;
518         #endif
519         memcpy(this->mppaArray, array.mppaArray, size()*sizeof(T
              ));
520     }
521     else{
```

```
522         #ifdef DEBUG
523             std::cout<<"MASTER: Copying element-by-element from
                    address "<<&(array.mppaArray)<<" to address "
                    <<&(this->mppaArray)<<std::endl;
524         #endif
525         for(size_t i = 0; i<height; ++i){
526             for(size_t j = 0; j<width; ++j){
527                 for(size_t k = 0; k<depth; ++k){
528                     this->mppaGet(i,j,k) = array.mppaGet(i,j,k);
529         }}}
530     }
531 }
532
533
534 #ifndef MPPA_MASTER
535 template<typename T> template<typename Arrays>
536 void ArrayBase<T>::mppaMemCopy(Arrays array){
537     if(array.size()==array.realSize() && this->size()==this->
            realSize()){
538         #ifdef DEBUG
539             std::cout<<"SLAVE: Copying memory from address "<<&(
                    array.mppaArray)<<" to address "<<&(this->
                    mppaArray)<<std::endl;
540         #endif
541         memcpy(this->mppaArray, array.mppaArray, size()*sizeof(T
                ));
542     }else{
543         #ifdef DEBUG
544             std::cout<<"SLAVE: Copying element-by-element from
                    address "<<&(array.mppaArray)<<" to address "
                    <<&(this->mppaArray)<<std::endl;
545         #endif
546         #pragma omp parallel for
547         for(size_t i = 0; i<height; ++i){
548             for(size_t j = 0; j<width; ++j){
549                 for(size_t k = 0; k<depth; ++k){
550                     this->mppaGet(i,j,k) = array.mppaGet(i,j,k);
551         }}}
552     }
553 }
554 #endif
555
556 template<typename T>
557 ArrayBase<T>::operator bool() const {
558     return(this->hostArray!=NULL);
559 }
560
561 //*****************************************************
562 // Array 3D
563 //*****************************************************
564
565 template<typename T>
566 Array3D<T>::Array3D() : ArrayBase<T>(0,0,0) {}
567
568 /*
569 ~Array3D(){
570 free(hostArray);
571 cudaFree(deviceArray);
572 }*/
573
574 template<typename T>
```

```
575 Array3D<T>::Array3D(size_t width, size_t height, size_t depth) :
        ArrayBase<T>(width,height,depth){}
576
577 template<typename T>
578 T & Array3D<T>::operator()(size_t h,size_t w,size_t d) const {
579     #ifdef PSKEL_MPPA
580         return this->mppaGet(h,w,d);
581     #else
582         return this->hostGet(h,w,d);
583     #endif
584 }
585
586 //*****************************************************
587 // Array 2D
588 //*****************************************************
589
590 template<typename T>
591 Array2D<T>::Array2D() : ArrayBase<T>(0,0,0) {}
592
593 //template<typename T>
594 // Array2D<T>::~Array2D(){
595 //   this->mppaFree();
596 // }
597
598 template<typename T>
599 Array2D<T>::Array2D(size_t width, size_t height) : ArrayBase<T>(
        width,height,1){}
600
601 template<typename T>
602 T & Array2D<T>::operator()(size_t h, size_t w) const {
603     #ifdef PSKEL_MPPA
604         return this->mppaGet(h,w,0);
605     #else
606         return this->hostGet(h,w,0);
607     #endif
608 }
609
610 //*****************************************************
611 // Array 1D
612 //*****************************************************
613
614 template<typename T>
615 Array<T>::Array() : ArrayBase<T>(0,0,0){}
616
617 template<typename T>
618 Array<T>::Array(size_t size) : ArrayBase<T>(size,1,1){}
619
620 template<typename T>
621 T & Array<T>::operator()(size_t w) const {
622     #ifdef PSKEL_MPPA
623         return this->mppaGet(0,w,0);
624     #else
625         return this->hostGet(0,w,0);
626     #endif
627 }
628
629 }//end namespace
630 #endif
```

Código 11 – PSkelStencil.h

```
#ifndef PSKEL_STENCIL_H
#define PSKEL_STENCIL_H

#include "PSkelDefs.h"
#include "PSkelArray.h"
#include "PSkelMask.h"
//#ifdef PSKEL_CUDA
  #include "PSkelStencilTiling.h"
//#endif

namespace PSkel{

//*************************************************
// Stencil Kernels that must be implemented by the users.
//*************************************************

/**
 * Function signature of the stencil kernel for processing 1-dimensional arrays.
 * This function must be implemented by the user.
 * \param[in] input 1-dimensional Array with the input data.
 * \param[in] output 1-dimensional Array that will hold the output data.
 * \param[in] mask 1-dimensional Mask used to scan through the input data,
 * accessing the neighbourhood for each element.
 * \param[in] args extra arguments that may be used by the stencil computations.
 * \param[in] i index for the current element to be processed.
 **/
template<typename T1, typename T2, class Args>
__parallel__ void stencilKernel(Array<T1> input, Array<T1> output, Mask<T2> mask, Args args, size_t i);

/**
 * Function signature of the stencil kernel for processing 2-dimensional arrays.
 * This function must be implemented by the user.
 * \param[in] input 1-dimensional Array with the input data.
 * \param[in] output 1-dimensional Array that will hold the output data.
 * \param[in] mask 1-dimensional Mask used to scan through the input data,
 * accessing the neighbourhood for each element.
 * \param[in] args extra arguments that may be used by the stencil computations.
 * \param[in] h height index for the current element to be processed.
 * \param[in] w width index for the current element to be processed.
 **/
template<typename T1, typename T2, class Args>
__parallel__ void stencilKernel(Array2D<T1> input, Array2D<T1> output, Mask2D<T2> mask, Args args, size_t h, size_t w);

/**
 * Function signature of the stencil kernel for processing 3-dimensional arrays.
 * This function must be implemented by the user.
 * \param[in] input 1-dimensional Array with the input data.
 * \param[in] output 1-dimensional Array that will hold the
```

```
     output data.
49   * \param[in] mask 1-dimensional Mask used to scan through the
       input data,
50   * accessing the neighbourhood for each element.
51   * \param[in] args extra arguments that may be used by the
       stencil computations.
52   * \param[in] h height index for the current element to be
       processed.
53   * \param[in] w width index for the current element to be
       processed.
54   * \param[in] d depth index for the current element to be
       processed.
55   **/
56  template<typename T1, typename T2, class Args>
57  __parallel__ void stencilKernel(Array3D<T1> input, Array3D<T1>
       output, Mask3D<T2> mask, Args args, size_t h, size_t w,
       size_t d);
58
59  //**************************************************
60  // Stencil Base
61  //**************************************************
62
63  /**
64   * Class that implements the basic functionalities supported by
       the stencil skeletons.
65   **/
66  template<class Array, class Mask, class Args=int>
67  class StencilBase{
68  private:
69  protected:
70      Array input;
71      Array output;
72      Args args;
73      Mask mask;
74
75      #ifndef MPPA_MASTER
76      virtual void runSeq(Array in, Array out) = 0;
77      #endif
78
79      #ifndef MPPA_MASTER
80      // virtual void runOpenMP(Array in, Array out, size_t
           numThreads) = 0;
81    virtual inline __attribute__((always_inline)) void runOpenMP(
         Array in, Array out, size_t width, size_t height, size_t
         depth, size_t maskRange, size_t numThreads) = 0;
82      #endif
83
84  public:
85      /**
86       * Executes sequentially in CPU a single iteration of the
           stencil computation.
87       **/
88      #ifndef MPPA_MASTER
89      void runSequential();
90      #endif
91
92      /**
93       * Executes in CPU, using multithreads, a single iteration
           of the stencil computation.
94       * \param[in] numThreads the number of threads used for
           processing the stencil kernel.
```

```
95        * if numThreads is 0, the number of threads is
              automatically chosen.
96        **/
97       #ifndef MPPA_MASTER
98       void runCPU(size_t numThreads=0);
99       #endif
100
101      /**
102       * Executes sequentially in CPU multiple iterations of the
              stencil computation.
103       * At each given iteration, except the first, the previous
              output is used as input.
104       * \param[in] iterations the number of iterations to be
              computed.
105       **/
106      #ifndef MPPA_MASTER
107      void runIterativeSequential(size_t iterations);
108      #endif
109      /**
110       * Executes in CPU, using multithreads, multiple iterations
              of the stencil computation.
111       * At each given iteration, except the first, the previous
              output is used as input.
112       * \param[in] iterations the number of iterations to be
              computed.
113       * \param[in] numThreads the number of threads used for
              processing the stencil kernel.
114       * if numThreads is 0, the number of threads is
              automatically chosen.
115       **/
116      #ifndef MPPA_MASTER
117      void runIterativeCPU(size_t iterations, size_t numThreads=0)
             ;
118      #endif
119
120      #ifdef PSKEL_MPPA
121      /**
122       * Spawn the slaves in MPPA.
123       * \param[in] slave_bin_name the name of the slave bynary
              code.
124       * \param[in] tilingHeight the height for each tile.
125       * \param[in] nb_clusters the number of clusters to be spawn
              .
126       * \param[in] nb_threads the number of threads per cluster.
127       **/
128      void spawn_slaves(const char slave_bin_name[], size_t
             tilingHeight, size_t tilingWidth, int nb_clusters, int
             nb_threads, int iterations, int innerIterations);
129      #endif
130
131      #ifdef PSKEL_MPPA
132      /**
133      * Create the slices for MPPA.
134      * \param[in] tilingHeight the height for each tile.
135      * \param[in] nb_clusters the number of clusters to divide
              the tiles.
136      **/
137      void mppaSlice(size_t tilingHeight, size_t tilingWidth, int
             nb_clusters, int iterations, int innerIterations);
138      #endif
139
```

```
140     #ifdef PSKEL_MPPA
141     /**
142     * wait for the slaves to complete.
143     * \param[in] nb_clusters the number of clusters to wait.
144     **/
145     void waitSlaves(int nb_clusters, int tilingHeight, int
            tilingWidth);
146     #endif
147
148     #ifdef PSKEL_MPPA
149     /**
150     * Configure the slave execution and wait for them to finish.
151     * \param[in] slave_bin_name the name of the slave bynary
            code.
152     * \param[in] nb_clusters the number of clusters to be spawn.
153     * \param[in] nb_threads the number of threads per cluster.
154     * \param[in] tilingHeight the height for each tile.
155     * \param[in] iterations the number of iterations for the
            execution.
156     **/
157     void scheduleMPPA(const char slave_bin_name[], int
            nb_clusters, int nb_threads, size_t tilingHeight, size_t
             tilingWidth, int iterations, int innerIterations);
158     #endif
159
160     #ifdef PSKEL_MPPA
161     /**
162     * Configure the portals for the slave and execute the kernel
            .
163     * \param[in] cluster_id the id of the executing cluster.
164     * \param[in] nb_threads the number of threads for the kernel
             execution.
165     * \param[in] nb_tiles the number of tiles for the cluster to
             execute.
166     **/
167     void runMPPA(int cluster_id, int nb_threads, int nb_tiles,
            int outterIterations, int itMod);
168     #endif
169
170     #ifdef PSKEL_MPPA
171     /**
172     *
173     * \param[in] cluster_id the id of the executing cluster.
174     * \param[in] nb_threads the number of threads for the kernel
             execution.
175     * \param[in] nb_tiles the number of tiles for the cluster to
             execute.
176     * \param[in] iterations the number of iterations for the
            execution.
177     **/
178     void runIterativeMPPA(Array in, Array out, int iterations,
            size_t numThreads);
179     #endif
180 };
181
182 //*************************************************
183 // Stencil 3D
184 //*************************************************
185
186 template<class Array, class Mask, class Args>
187 class Stencil3D : public StencilBase<Array, Mask, Args>{
```

```
188 protected:
189     #ifndef MPPA_MASTER
190     void runSeq(Array in, Array out);
191     #endif
192 
193     #ifndef MPPA_MASTER
194     void runOpenMP(Array in, Array out, size_t numThreads);
195     #endif
196 
197 public:
198     Stencil3D();
199     Stencil3D(Array _input, Array _output, Mask _mask, Args
            _args);
200 };
201 
202 //*************************************************
203 // Stencil 2D
204 //*************************************************
205 
206 template<class Array, class Mask, class Args>
207 class Stencil2D : public StencilBase<Array, Mask, Args>{
208 protected:
209 
210     #ifndef MPPA_MASTER
211     void runSeq(Array in, Array out);
212     #endif
213 
214     #ifndef MPPA_MASTER
215     // void runOpenMP(Array in, Array out, size_t numThreads);
216     inline __attribute__((always_inline)) void runOpenMP(Array
            in, Array out, size_t width, size_t height, size_t depth
            , size_t maskRange, size_t numThreads);
217     #endif
218 
219 public:
220     Stencil2D();
221     Stencil2D(Array _input, Array _output, Mask _mask, Args
            _args);
222     Stencil2D(Array _input, Array _output, Mask _mask);
223     ~Stencil2D();
224 };
225 
226 //*************************************************
227 // Stencil 1D
228 //*************************************************
229 
230 
231 template<class Array, class Mask, class Args>
232 class Stencil: public StencilBase<Array, Mask, Args>{
233 protected:
234 
235     #ifndef MPPA_MASTER
236     void runSeq(Array in, Array out);
237     #endif
238 
239     #ifndef MPPA_MASTER
240     void runOpenMP(Array in, Array out, size_t numThreads);
241     #endif
242 
243 public:
244     Stencil();
```

```
245     Stencil(Array _input, Array _output, Mask _mask, Args _args)
            ;
246 };
247
248 }
249
250 #include "PSkelStencil.hpp"
251 //#include "PSkelStencilMPPA.hpp"
252
253 #endif
```

Código 12 – PSkelStencil.hpp

```
1  #ifndef PSKEL_STENCIL_HPP
2  #define PSKEL_STENCIL_HPP
3  //#define BARRIER_SYNC_MASTER "/mppa/sync/128:1"
4  //#define BARRIER_SYNC_SLAVE "/mppa/sync/[0..15]:2"
5  #include <cmath>
6  #include <algorithm>
7  #include <iostream>
8  #include "hr_time.h"
9
10 #include <iostream>
11 #include <unistd.h>
12
13 using namespace std;
14 #ifdef PSKEL_CUDA
15 #include <ga/ga.h>
16 #include <ga/std_stream.h>
17 #endif
18
19 #define ARGC_SLAVE 11
20
21 namespace PSkel{
22
23     //*********************************************
24     // Stencil Base
25     //*********************************************
26 #ifndef MPPA_MASTER
27     template<class Array, class Mask, class Args>
28         void StencilBase<Array, Mask, Args>::runSequential(){
29             this->runSeq(this->input, this->output);
30         }
31 #endif
32
33     //*********************************************
34     // MPPA
35     //*********************************************
36
37 #ifdef PSKEL_MPPA
38     template<class Array, class Mask, class Args>
39         void StencilBase<Array, Mask, Args>::spawn_slaves(const
              char slave_bin_name[], size_t tilingHeight, size_t
              tilingWidth, int nb_clusters, int nb_threads, int
              iterations, int innerIterations){
40             // Prepare arguments to send to slaves
41             int i;
42             int cluster_id;
43             int pid;
44
```

```
45              size_t wTiling = ceil(float(this->input.getWidth())/
                    float(tilingWidth));
46              size_t hTiling = ceil(float(this->input.getHeight())
                    /float(tilingHeight));
47              size_t totalSize = float(hTiling*wTiling);
48
49              int tiles = totalSize/nb_clusters;
50
51              int itMod = totalSize % nb_clusters;
52
53              int tilesSlave;
54              int r;
55              int outterIterations = ceil(float(iterations)/
                    innerIterations);
56
57
58  #ifdef DEBUG
59              cout<<"MASTER: width="<<this->input.getWidth()<<"
                    height="<<this->input.getWidth();
60              cout<<" tilingHeight="<<tilingHeight <<" iterations=
                    "<<iterations;
61              cout<<" inner_iterations="<<innerIterations<<"
                    nbclusters="<<nb_clusters<<" nbthreads="<<
                    nb_threads<<endl;
62              cout<<"MASTER: tiles="<<tiles<<" itMod="<<itMod<<"
                    outterIterations="<<outterIterations<<endl;
63  #endif
64
65              char **argv_slave = (char**) malloc(sizeof (char*) *
                     ARGC_SLAVE);
66              for (i = 0; i < ARGC_SLAVE - 1; i++)
67                  argv_slave[i] = (char*) malloc (sizeof (char) *
                        11);
68
69              sprintf(argv_slave[1], "%d", tilingWidth);
70              sprintf(argv_slave[2], "%d", tilingHeight);
71              sprintf(argv_slave[4], "%d", nb_threads);
72              sprintf(argv_slave[5], "%d", iterations);
73              sprintf(argv_slave[6], "%d", outterIterations);
74              sprintf(argv_slave[7], "%d", itMod);
75              sprintf(argv_slave[8], "%d", this->input.getHeight()
                    );
76              sprintf(argv_slave[9], "%d", this->input.getHeight()
                    );
77
78              argv_slave[10] = NULL;
79
80
81              // Spawn slave processes
82              for (cluster_id = 0; cluster_id < nb_clusters &&
                    cluster_id < totalSize; cluster_id++) {
83                  r = (cluster_id < itMod)?1:0;
84                  tilesSlave = tiles + r;
85
86                  sprintf(argv_slave[0], "%d", tilesSlave);
87                  sprintf(argv_slave[3], "%d", cluster_id);
88                  pid = mppa_spawn(cluster_id, NULL,
                        slave_bin_name, (const char **)argv_slave,
                        NULL);
89                  assert(pid >= 0);
90              }
```

```
91              for (i = 0; i < ARGC_SLAVE; i++)
92                  free(argv_slave[i]);
93              free(argv_slave);
94          }
95  #endif
96
97
98  #ifdef PSKEL_MPPA
99      template<class Array, class Mask, class Args>
100         void StencilBase<Array, Mask,Args>::mppaSlice(size_t
                tilingHeight, size_t tilingWidth, int nb_clusters,
                int iterations, int innerIterations) {
101
102             size_t wTiling = ceil(float(this->input.getWidth())/
                    float(tilingWidth));
103             size_t hTiling = ceil(float(this->input.getHeight())
                    /float(tilingHeight));
104             size_t totalSize = float(hTiling*wTiling);
105             int tiles = totalSize/nb_clusters;
106             int itMod = totalSize % nb_clusters;
107
108
109
110             size_t outterIterations;
111             size_t heightOffset;
112             size_t widthOffset;
113
114             StencilTiling<Array, Mask>tiling(this->input, this->
                    output, this->mask);
115
116             barrier_t *barrierNbClusters;
117             outterIterations = ceil(float(iterations)/
                    innerIterations);
118             Array inputCopy(this->input.getWidth(),this->input.
                    getHeight());
119             this->output.portalReadAlloc(nb_clusters, 0);
120
121
122
123             for(size_t it = 0; it<outterIterations; it++){
124                 size_t subIterations = innerIterations;
125                 if(((it+1)*innerIterations)>iterations){
126                     subIterations = iterations-(it*
                            innerIterations);
127                 }
128
129                 if(tiles == 0) {
130                     Array slice[totalSize];
131                     Array outputNumberHt[totalSize];
132                     Array tmp;
133                     ////////////////////////////////Number of
                            clusters are higher//////////////
134                     barrierNbClusters =
                            mppa_create_master_barrier(
                            BARRIER_SYNC_MASTER, BARRIER_SYNC_SLAVE,
                             totalSize);
135
136                     mppa_barrier_wait(barrierNbClusters);
137
138                     for (int i = 0; i < totalSize; i++) {
139                         outputNumberHt[i].portalAuxWriteAlloc(i)
```

```
                    ;
140                 }
141
142                 mppa_barrier_wait(barrierNbClusters);
143
144                 int tS = 0;
145                 for (int ht = 0; ht < hTiling; ht++) {
146                     for (int wt = 0; wt < wTiling; wt++) {
147                         heightOffset = ht*tilingHeight;
148                         widthOffset = wt*tilingWidth;
149                         tiling.tile(subIterations,
                                widthOffset, heightOffset, 0,
                                tilingWidth, tilingHeight, 1);
150                         outputNumberHt[tS].hostSlice(tiling.
                                input, tiling.widthOffset,
                                tiling.heightOffset, tiling.
                                depthOffset, tiling.width,
                                tiling.height, tiling.depth);
151                         outputNumberHt[tS].setAux(
                                heightOffset, widthOffset, it,
                                subIterations, tiling.
                                coreWidthOffset, tiling.
                                coreHeightOffset, tiling.
                                coreDepthOffset, tiling.
                                coreWidth, tiling.coreHeight,
                                tiling.coreDepth,
                                outterIterations, tiling.height,
                                 tiling.width, tiling.depth,
                                this->input.getWidth(), this->
                                input.getHeight());
152                         // cout << "Tiling Height Offset: "
                                << tiling.heightOffset << "
                                Tiling Width Offset: " << tiling
                                .widthOffset << "Tiling Core
                                Height Offset: " << tiling.
                                coreHeightOffset << "Tiling Core
                                 Width Offset: " << tiling.
                                coreHeightOffset << endl;
153                         tS++;
154                     }
155                 }
156
157                 for (int i = 0; i < totalSize; i++) {
158                     outputNumberHt[i].copyToAux();
159                 }
160                 for (int i = 0; i < totalSize; i++) {
161                     outputNumberHt[i].waitAuxWrite();
162                 }
163
164                 for (int i = 0; i < totalSize; i++) {
165                     outputNumberHt[i].closeAuxWritePortal();
166                 }
167
168                 for (int i = 0; i < totalSize; i++) {
169                     slice[i].portalWriteAlloc(i);
170                 }
171                 mppa_barrier_wait(barrierNbClusters);
172                 tS = 0;
173
174
175                 for(int ht = 0; ht < hTiling; ht++) {
```

```
176                         for(int wt = 0; wt < wTiling; wt++){
177                             heightOffset = ht*tilingHeight;
178                             widthOffset = wt*tilingWidth;
179                             tiling.tile(subIterations,
                                    widthOffset, heightOffset, 0,
                                    tilingWidth, tilingHeight, 1);
180                             slice[tS].hostSlice(tiling.input,
                                    tiling.widthOffset, tiling.
                                    heightOffset, tiling.depthOffset
                                    , tiling.width, tiling.height,
                                    tiling.depth);
181                             slice[tS].copyTo(tiling.height,
                                    tiling.width, this->input.
                                    getWidth(), tiling.width, (
                                    tiling.heightOffset*this->input.
                                    getWidth())+tiling.widthOffset,
                                    0);
182                             for(size_t h=0;h<slice[tS].getHeight
                                    ();h++) {
183                                 for(size_t w=0;w<slice[tS].
                                        getWidth();w++) {
184                                     printf("ClusterSlice(%d,%d)
                                            :%d\n",h,w, slice[tS](h,
                                            w));
185                                 }
186                             }
187
188                             tS++;
189                         }
190                     }
191                     for (int i = 0; i < totalSize; i++) {
192                         slice[i].waitWrite();
193                     }
194
195                     this->output.setTrigger(totalSize);
196                     this->output.copyFrom();
197                     printf("Output: ");
198                     for(size_t h=0;h<output.getHeight();h++) {
199                         printf("\n");
200                         for(size_t w=0;w<output.getWidth();w++)
                                {
201                             printf(" %f ", output(h,w));
202                         }
203                         printf("\n");
204                     }
205
206                     for (int i = 0; i < totalSize; i++) {
207                         slice[i].closeWritePortal();
208                     }
209                     mppa_close_barrier(barrierNbClusters);
210                 } else {
211                     ////////////////////////////hTiling is higher
                            ////////////////////////////////
212                     int counter = 0;
213
214                     Array cluster[nb_clusters];
215                     Array tmp;
216                     barrierNbClusters =
                            mppa_create_master_barrier(
                            BARRIER_SYNC_MASTER, BARRIER_SYNC_SLAVE,
                             nb_clusters);
```

```
217
218                         Array outputNumberNb[nb_clusters];
219                         Array outputNumberMod[itMod];
220
221                         int baseItHt = 0;
222                         int baseItWt = 0;
223                         int auxHt = 0;
224                         int auxWt = 0;
225                         int ht = 0;
226                         int wt = 0;
227                         for(int i = 0; i < tiles; i++) {
228
229                             mppa_barrier_wait(barrierNbClusters);
230
231                             for (int i = 0; i < nb_clusters; i++) {
232                                 outputNumberNb[i].
                                        portalAuxWriteAlloc(i);
233                             }
234
235                             mppa_barrier_wait(barrierNbClusters);
236                             int tS = 0;
237
238                             auxHt = baseItHt;
239                             auxWt = baseItWt;
240                             if (auxWt == wTiling) {
241                                 auxHt++;
242                                 auxWt = 0;
243                             }
244
245                             for (auxHt; auxHt < hTiling; auxHt++) {
246                                 for (auxWt; auxWt < wTiling; auxWt
                                        ++) {
247                                     heightOffset = auxHt*
                                            tilingHeight;
248                                     widthOffset = auxWt*tilingWidth;
249                                     tiling.tile(subIterations,
                                            widthOffset, heightOffset,
                                            0, tilingWidth, tilingHeight
                                            , 1);
250                                     outputNumberNb[tS].hostSlice(
                                            tiling.input, tiling.
                                            widthOffset, tiling.
                                            heightOffset, tiling.
                                            depthOffset, tiling.width,
                                            tiling.height, tiling.depth)
                                            ;
251                                     outputNumberNb[tS].setAux(
                                            heightOffset, widthOffset,
                                            it, subIterations, tiling.
                                            coreWidthOffset, tiling.
                                            coreHeightOffset, tiling.
                                            coreDepthOffset, tiling.
                                            coreWidth, tiling.coreHeight
                                            , tiling.coreDepth,
                                            outterIterations, tiling.
                                            height, tiling.width, tiling
                                            .depth, this->input.getWidth
                                            (), this->input.getHeight())
                                            ;
252                                     tS++;
253                                     if (tS >= nb_clusters) {
```

```
254                                 auxHt = hTiling;
255                                 auxWt = wTiling;
256                             }
257                         }
258                         if (auxWt == wTiling) {
259                             auxWt = 0;
260                         }
261
262                     }
263
264                     for (int i = 0; i < nb_clusters; i++) {
265                         outputNumberNb[i].copyToAux();
266                     }
267                     for (int i = 0; i < nb_clusters; i++) {
268                         outputNumberNb[i].waitAuxWrite();
269                     }
270                     for (int i = 0; i < nb_clusters; i++) {
271                         outputNumberNb[i].
                                closeAuxWritePortal();
272                     }
273
274
275                     for (int i = 0; i < nb_clusters; i++) {
276                         cluster[i].portalWriteAlloc(i);
277                     }
278                     mppa_barrier_wait(barrierNbClusters);
279                     tS = 0;
280                     ht = baseItHt;
281                     wt = baseItWt;
282                     if (wt == wTiling) {
283                         ht++;
284                         wt = 0;
285                     }
286
287                     for (ht; ht < hTiling; ht++) {
288                         for (wt; wt < wTiling; wt++) {
289                             // masterCopyItStart =
                                    mppa_master_get_time();
290                             heightOffset = ht*tilingHeight;
291                             widthOffset = wt*tilingWidth;
292                             tiling.tile(subIterations,
                                    widthOffset, heightOffset,
                                    0, tilingWidth, tilingHeight
                                    , 1);
293                             cluster[tS].hostSlice(tiling.
                                    input, tiling.widthOffset,
                                    tiling.heightOffset, tiling.
                                    depthOffset, tiling.width,
                                    tiling.height, tiling.depth)
                                    ;
294                             cluster[tS].copyTo(tiling.height
                                    , tiling.width, this->input.
                                    getWidth(), tiling.width, (
                                    tiling.heightOffset*this->
                                    input.getWidth())+tiling.
                                    widthOffset, 0);
295                             // for (int i = 0; i <
                                    nb_clusters; i++) {
296                                 // for(size_t h=0;h<cluster[
                                        tS].getHeight();h++) {
297                                 //     for(size_t w=0;w<
```

```
                                    cluster[tS].getWidth();w
                                    ++) {
298                             //          printf("
                                    ClusterSlice(%d,%d):%d\n
                                    ",h,w, cluster[tS](h,w))
                                    ;
299                             //      }
300                             // }
301                         // }
302                         // cout<<"Tile size(" << ht
                                <<","<< wt << ") is: H(" <<
                                tmp.getHeight() << ") ~ W("
                                << tmp.getWidth() << ")" <<
                                endl;
303                         // sleep(1);
304                         // cluster[tS].mppaMasterClone(
                                tmp);
305                         // masterCopyItEnd =
                                mppa_master_get_time();
306                         // cluster[tS].copyTo();
307                         tS++;
308                         if (tS >= nb_clusters) {
309                             baseItHt = ht;
310                             ht = hTiling;
311                             baseItWt = wt + 1;
312                             wt = wTiling;
313                         }
314                         // masterSendItEnd =
                                mppa_master_get_time();
315
316                     }
317                     if (wt == wTiling) {
318                         wt = 0;
319                     }
320                 }
321
322
323                 for (int i = 0; i < nb_clusters; i++) {
324                     cluster[i].waitWrite();
325                 }
326
327
328
329                 this->output.setTrigger(nb_clusters);
330
331                 this->output.copyFrom();
332
333                 printf("Output: ");
334                 for(size_t h=0;h<output.getHeight();h++)
                        {
335                     printf("\n");
336                     for(size_t w=0;w<output.getWidth();w
                            ++) {
337                         printf(" %f ", output(h,w));
338                     }
339                 }
340
341                 for (int i = 0; i < nb_clusters; i++) {
342                     cluster[i].closeWritePortal();
343                 }
344
```

```
345
346                     }
347
348                     int hTilingMod = hTiling;
349                     int wTilingMod = wTiling;
350                     if (itMod == 0) {
351                         hTilingMod = itMod;
352                         wTilingMod = itMod;
353                     }
354                     mppa_barrier_wait(barrierNbClusters);
355
356                     for(int j = 0; j < itMod; j++) {
357                         outputNumberMod[j].portalAuxWriteAlloc(j
                                );
358                     }
359
360                     mppa_barrier_wait(barrierNbClusters);
361                     int tS = 0;
362                     auxHt = baseItHt;
363                     auxWt = baseItWt;
364                     if (auxWt == wTilingMod) {
365                         auxHt++;
366                         auxWt = 0;
367                     }
368                     for (auxHt; auxHt < hTilingMod; auxHt++) {
369                         for (auxWt; auxWt < wTilingMod; auxWt++)
                                {
370                             heightOffset = auxHt*tilingHeight;
371                             widthOffset = auxWt*tilingWidth;
372                             tiling.tile(subIterations,
                                    widthOffset, heightOffset, 0,
                                    tilingWidth, tilingHeight, 1);
373                             outputNumberMod[tS].hostSlice(tiling
                                    .input, tiling.widthOffset,
                                    tiling.heightOffset, tiling.
                                    depthOffset, tiling.width,
                                    tiling.height, tiling.depth);
374                             outputNumberMod[tS].setAux(
                                    heightOffset, widthOffset, it,
                                    subIterations, tiling.
                                    coreWidthOffset, tiling.
                                    coreHeightOffset, tiling.
                                    coreDepthOffset, tiling.
                                    coreWidth, tiling.coreHeight,
                                    tiling.coreDepth,
                                    outterIterations, tiling.height,
                                     tiling.width, tiling.depth,
                                    this->input.getWidth(), this->
                                    input.getHeight());
375                             tS++;
376                         }
377                         auxWt = 0;
378                     }
379
380                     for (int i = 0; i < itMod; i++) {
381                         outputNumberMod[i].copyToAux();
382                     }
383                     for (int i = 0; i < itMod; i++) {
384                         outputNumberMod[i].waitAuxWrite();
385                     }
386                     for(int i = 0; i < itMod; i++) {
```

```
387                         outputNumberMod[i].closeAuxWritePortal()
                                ;
388                     }
389
390
391                     for(int j = 0; j < itMod; j++) {
392                         cluster[j].portalWriteAlloc(j);
393                     }
394                     mppa_barrier_wait(barrierNbClusters);
395
396
397                     tS = 0;
398                     ht = baseItHt;
399                     wt = baseItWt;
400                     if (wt == wTiling) {
401                         ht++;
402                         wt = 0;
403                     }
404
405                     for (ht; ht < hTilingMod; ht++) {
406                         for (wt; wt < wTilingMod; wt++) {
407                             heightOffset = ht*tilingHeight;
408                             widthOffset = wt*tilingWidth;
409                             tiling.tile(subIterations,
                                    widthOffset, heightOffset, 0,
                                    tilingWidth, tilingHeight, 1);
410                             cluster[tS].hostSlice(tiling.input,
                                    tiling.widthOffset, tiling.
                                    heightOffset, tiling.depthOffset
                                    , tiling.width, tiling.height,
                                    tiling.depth);
411                             cluster[tS].copyTo(tiling.height,
                                    tiling.width, this->input.
                                    getWidth(), tiling.width, (
                                    tiling.heightOffset*this->input.
                                    getWidth())+tiling.widthOffset,
                                    0);
412                             tS++;
413                         }
414                         wt = 0;
415                     }
416
417                     for (int i = 0; i < itMod; i++) {
418                         cluster[i].waitWrite();
419                     }
420                     this->output.setTrigger(itMod);
421
422
423                     this->output.copyFrom();
424
425
426                     for (int i = 0; i < itMod; i++) {
427                         cluster[i].closeWritePortal();
428                     }
429                     for(int i = 0; i < itMod; i++) {
430                         outputNumberMod[i].auxFree();
431                     }
432                     mppa_close_barrier(barrierNbClusters);
433
434                 }
435
```

```
436                     inputCopy.mppaMasterCopy(this->input);
437                     this->input.mppaMasterCopy(this->output);
438                     this->output.mppaMasterCopy(inputCopy);
439
440                 }
441
442
443
444                 inputCopy.mppaFree();
445                 this->output.closeReadPortal();
446                 this->output.mppaMasterCopy(this->input);
447
448
449             }
450 #endif
451
452
453
454
455 #ifdef PSKEL_MPPA
456     template<class Array, class Mask, class Args>
457         void StencilBase<Array, Mask,Args>::waitSlaves(int
                nb_clusters, int tilingHeight, int tilingWidth) {
458             size_t hTiling = ceil(float(this->input.getHeight())
                    /float(tilingHeight));
459             size_t wTiling = ceil(float(this->input.getHeight())
                    /float(tilingWidth));
460             size_t totalSize = float(hTiling*wTiling);
461             int pid;
462             int wait;
463             if(totalSize < nb_clusters)
464                 nb_clusters = totalSize;
465             for (pid = 0; pid < nb_clusters; pid++) {
466                 wait = mppa_waitpid(pid, NULL, 0);
467                 assert(wait != -1);
468             }
469         }
470 #endif
471
472
473 #ifdef PSKEL_MPPA
474     template<class Array, class Mask, class Args>
475         void StencilBase<Array, Mask,Args>::scheduleMPPA(const
                char slave_bin_name[], int nb_clusters, int
                nb_threads, size_t tilingHeight, size_t tilingWidth,
                 int iterations, int innerIterations){
476
477             this->spawn_slaves(slave_bin_name, tilingHeight,
                    tilingWidth, nb_clusters, nb_threads, iterations
                    , innerIterations);
478
479             this->mppaSlice(tilingHeight, tilingWidth,
                    nb_clusters, iterations, innerIterations);
480
481             this->waitSlaves(nb_clusters, tilingHeight,
                    tilingWidth);
482
483         }
484 #endif
485
486
```

```
487 #ifdef PSKEL_MPPA
488     template<class Array, class Mask, class Args>
489         void StencilBase<Array, Mask,Args>::runMPPA(int
                cluster_id, int nb_threads, int nb_tiles, int
                outterIterations, int itMod){
490             Array finalArr;
491             Array coreTmp;
492             Array tmp;
493             Array inputTmp;
494             Array outputTmp;
495             Array input;
496             Array auxPortal;
497             int *aux;
498             for(int j = 0; j < outterIterations; j++) {
499                 barrier_t *global_barrier =
                        mppa_create_slave_barrier(
                        BARRIER_SYNC_MASTER, BARRIER_SYNC_SLAVE);
500                 for(int i = 0; i < nb_tiles; i++) {
501                     mppa_barrier_wait(global_barrier);
502
503                     if(i == 0) {
504                         auxPortal.portalAuxReadAlloc(1,
                                cluster_id);
505                         finalArr.portalWriteAlloc(0);
506                     }
507
508                     mppa_barrier_wait(global_barrier);
509                     auxPortal.copyFromAux();
510
511                     aux = auxPortal.getAux();
512
513                     int heightOffset = aux[0];
514                     int it = aux[1];
515                     int subIterations = aux[2];
516                     int coreWidthOffset = aux[3];
517                     int coreHeightOffset = aux[4];
518                     int coreDepthOffset = aux[5];
519                     int coreWidth = aux[6];
520                     int coreHeight = aux[7];
521                     int coreDepth = aux[8];
522                     int h = aux[10];
523                     int w = aux[11];
524                     int d = aux[12];
525                     int widthOffset = aux[13];
526                     int baseWidth = aux[14];
527
528                     finalArr.mppaAlloc(w,h,d);
529                     inputTmp.mppaAlloc(w,h,d);
530                     outputTmp.mppaAlloc(w,h,d);
531                     inputTmp.portalReadAlloc(1, cluster_id);
532                     mppa_barrier_wait(global_barrier);
533
534                     inputTmp.copyFrom();
535
536                     for(size_t h=0;h<inputTmp.getHeight();h++) {
537                         for(size_t w=0;w<inputTmp.getWidth();w
                                ++) {
538                             printf("Arrived(%d,%d):%d\n",h,w,
                                    inputTmp(h,w));
539                       }
540                     }
```

```
541
542                     this->runIterativeMPPA(inputTmp, outputTmp,
                            subIterations, nb_threads);
543                     for(size_t h=0;h<outputTmp.getHeight();h++)
                            {
544                         for(size_t w=0;w<outputTmp.getWidth();w
                                ++) {
545                           printf("Computated(%d,%d):%d\n",h,w,
                                outputTmp(h,w));
546                       }
547                     }
548
549                     if (subIterations%2==0) {
550                         finalArr.mppaMemCopy(inputTmp);
551                     } else {
552                         finalArr.mppaMemCopy(outputTmp);
553                     }
554
555
556                     coreTmp.hostSlice(finalArr, coreWidthOffset,
                            coreHeightOffset, coreDepthOffset,
                            coreWidth, coreHeight, coreDepth);
557                     for(size_t h=0;h<coreTmp.getHeight();h++) {
558                         for(size_t w=0;w<coreTmp.getWidth();w++)
                                {
559                             printf("finalArr(%d,%d):%d, cluster
                                    [%d]\n",h,w, coreTmp(h,w),
                                    cluster_id);
560                         }
561                     }
562
563                     int masterBaseOffset = ((heightOffset*
                            baseWidth) + widthOffset);
564                     finalArr.copyTo(coreHeight, coreWidth, w,
                            baseWidth, (inputTmp.getWidth()*
                            coreHeightOffset)+coreWidthOffset,
                            masterBaseOffset);
565                     finalArr.waitWrite();
566                     finalArr.mppaFree();
567                     finalArr.auxFree();
568                     inputTmp.mppaFree();
569                     inputTmp.auxFree();
570
571
572                     outputTmp.mppaFree();
573                     outputTmp.auxFree();
574
575                     inputTmp.closeReadPortal();
576
577                     if (i == (nb_tiles-1)) {
578                         auxPortal.closeAuxReadPortal();
579                         finalArr.closeWritePortal();
580                     }
581                 }
582                 if(cluster_id >= itMod) {
583                     mppa_barrier_wait(global_barrier);
584                     mppa_barrier_wait(global_barrier);
585                     mppa_barrier_wait(global_barrier);
586                 }
587                 mppa_close_barrier(global_barrier);
588             }
```

```
589
590         }
591 #endif
592 #ifdef PSKEL_MPPA
593 #ifndef MPPA_MASTER
594     template<class Array, class Mask, class Args>
595         void StencilBase<Array, Mask,Args>::runIterativeMPPA(
                Array in, Array out, int iterations, size_t
                numThreads){
596             size_t width = this->input.getWidth();
597             size_t height = this->input.getHeight();
598             size_t depth = this->input.getDepth();
599             size_t maskRange = this->mask.getRange();
600             for(int i = 0; i < iterations; i++) {
601                 if(i%2==0) {
602                     this->runOpenMP(in, out, width, height,
                            depth, maskRange, numThreads);
603                 } else {
604                     this->runOpenMP(out, in, width, height,
                            depth, maskRange, numThreads);
605
606                 }
607             }
608
609         }
610 #endif
611 #endif
612     //***********************************************
613     //***********************************************
614 #ifndef MPPA_MASTER
615     template<class Array, class Mask, class Args>
616         void StencilBase<Array, Mask,Args>::runCPU(size_t
                numThreads){
617             numThreads = (numThreads==0)?omp_get_num_procs():
                    numThreads;
618             this->runOpenMP(this->input, this->output, this->
                    input.getWidth(), this->input.getHeight(), this
                    ->input.getDepth(), this->mask.getRange(),
                    numThreads);
619         }
620 #endif
621
622 #ifndef MPPA_MASTER
623     template<class Array, class Mask, class Args>
624         void StencilBase<Array, Mask,Args>::
                runIterativeSequential(size_t iterations){
625             Array inputCopy;
626             inputCopy.hostClone(input);
627             for(size_t it = 0; it<iterations; it++){
628                 if(it%2==0) this->runSeq(inputCopy, this->output
                        );
629                 else this->runSeq(this->output, inputCopy);
630             }
631             if((iterations%2)==0) output.hostMemCopy(inputCopy);
632             inputCopy.hostFree();
633         }
634 #endif
635
636 #ifndef MPPA_MASTER
637     template<class Array, class Mask, class Args>
638         void StencilBase<Array, Mask,Args>::runIterativeCPU(
```

```
            size_t iterations, size_t numThreads){
639             numThreads = (numThreads==0)?omp_get_num_procs():
                    numThreads;
640             size_t width = this->input.getWidth();
641             size_t height = this->input.getHeight();
642             size_t depth = this->input.getDepth();
643             size_t maskRange = this->mask.getRange();
644             //cout << "numThreads: " << numThreads << endl;
645             Array inputCopy;
646             inputCopy.hostClone(input);
647             for(size_t it = 0; it<iterations; it++){
648                 if(it%2==0){
649                     // this->runOpenMP(inputCopy, this->output,
                            numThreads);
650                     this->runOpenMP(input, this->output, width,
                            height, depth, maskRange, numThreads);
651                 }else {
652                     // this->runOpenMP(this->output, inputCopy,
                            numThreads);
653                     this->runOpenMP(this->output, input, width,
                            height, depth, maskRange, numThreads);
654                 }
655             }
656             if((iterations%2)==0) output.hostMemCopy(inputCopy);
657             inputCopy.hostFree();
658         }
659 #endif
660
661 //***********************************************
662 // Stencil 3D
663 //***********************************************
664
665
666 template<class Array, class Mask, class Args>
667     Stencil3D<Array,Mask,Args>::Stencil3D(){}
668
669 template<class Array, class Mask, class Args>
670     Stencil3D<Array,Mask,Args>::Stencil3D(Array _input, Array
            _output, Mask _mask, Args _args){
671         this->input = _input;
672         this->output = _output;
673         this->args = _args;
674         this->mask = _mask;
675     }
676
677 #ifndef MPPA_MASTER
678     template<class Array, class Mask, class Args>
679         void Stencil3D<Array,Mask,Args>::runSeq(Array in, Array
                out){
680             for (int h = 0; h < in.getHeight(); h++){
681                 for (int w = 0; w < in.getWidth(); w++){
682                     for (int d = 0; d < in.getDepth(); d++){
683                         stencilKernel(in,out,this->mask,this->
                                args,h,w,d);
684                     }}}
685         }
686 #endif
687
688 #ifndef MPPA_MASTER
689     template<class Array, class Mask, class Args>
690         void Stencil3D<Array,Mask,Args>::runOpenMP(Array in,
```

```
                Array out, size_t numThreads){
691             omp_set_num_threads(numThreads);
692 #pragma omp parallel for
693             for (int h = 0; h < in.getHeight(); h++){
694                 for (int w = 0; w < in.getWidth(); w++){
695                     for (int d = 0; d < in.getDepth(); d++){
696                         stencilKernel(in,out,this->mask,this->
                                args,h,w,d);
697                     }}}
698         }
699 #endif
700
701 //*************************************************
702 // Stencil 2D
703 //*************************************************
704
705 template<class Array, class Mask, class Args>
706     Stencil2D<Array,Mask,Args>::Stencil2D(){}
707
708 template<class Array, class Mask, class Args>
709     Stencil2D<Array,Mask,Args>::Stencil2D(Array _input, Array
            _output, Mask _mask, Args _args){
710         this->input = _input;
711         this->output = _output;
712         this->args = _args;
713         this->mask = _mask;
714     }
715
716 template<class Array, class Mask, class Args>
717     Stencil2D<Array,Mask,Args>::Stencil2D(Array _input, Array
            _output, Mask _mask){
718         this->input = _input;
719         this->output = _output;
720         this->mask = _mask;
721     }
722
723
724 template<class Array, class Mask, class Args>
725     Stencil2D<Array,Mask,Args>::~Stencil2D(){
726 #ifdef PSKEL_MPPA
727             this->input.mppaFree();
728             this->output.mppaFree();
729 #endif
730 #ifdef PSKEL_CPU
731             this->input.hostFree();
732             this->output.hostFree();
733 #endif
734         }
735
736 #ifndef MPPA_MASTER
737     template<class Array, class Mask, class Args>
738         void Stencil2D<Array,Mask,Args>::runSeq(Array in, Array
                out){
739             for (int h = 0; h < in.getHeight(); h++){
740                 for (int w = 0; w < in.getWidth(); w++){
741                     stencilKernel(in,out,this->mask, this->args,
                            h,w);
742             }}
743         }
744 #endif
745
```

```
746 #ifndef MPPA_MASTER
747     template<class Array, class Mask, class Args>
748         inline __attribute__((always_inline)) void Stencil2D<
              Array,Mask,Args>::runOpenMP(Array in, Array out,
              size_t width, size_t height, size_t depth, size_t
              maskRange, size_t numThreads){
749 //              size_t hrange = height-maskRange;
750 //              size_t wrange = width-maskRange;
751 //              int count = 0;
752 // #pragma omp parallel num_threads(numThreads)
753 //              {
754 // #pragma omp for
755 //                  for (size_t h = maskRange; h < hrange; h++){
756 //                      for (size_t w = maskRange; w < wrange; w
      ++){
757 //                          stencilKernel(in,out,this->mask,this
      ->args,h,w);
758 //                      }}
759 //              }
760             omp_set_num_threads(numThreads);
761 #pragma omp parallel for
762             for (int h = 0; h < in.getHeight(); h++){
763                 for (int w = 0; w < in.getWidth(); w++){
764                     stencilKernel(in,out,this->mask,this->args,h
                          ,w);
765                     }}
766         }
767 #endif
768
769 //*************************************************
770 // Stencil 1D
771 //*************************************************
772
773
774 template<class Array, class Mask, class Args>
775     Stencil<Array,Mask,Args>::Stencil(){}
776
777 template<class Array, class Mask, class Args>
778     Stencil<Array,Mask,Args>::Stencil(Array _input, Array
          _output, Mask _mask, Args _args){
779         this->input = _input;
780         this->output = _output;
781         this->args = _args;
782         this->mask = _mask;
783     }
784
785 #ifndef MPPA_MASTER
786     template<class Array, class Mask, class Args>
787         void Stencil<Array,Mask,Args>::runSeq(Array in, Array
              out){
788             for (int i = 0; i < in.getWidth(); i++){
789                 stencilKernel(in,out,this->mask, this->args,i);
790             }
791         }
792 #endif
793
794 #ifndef MPPA_MASTER
795     template<class Array, class Mask, class Args>
796         void Stencil<Array,Mask,Args>::runOpenMP(Array in, Array
               out, size_t numThreads){
797             omp_set_num_threads(numThreads);
```

```
798 #pragma omp parallel for
799             for (int i = 0; i < in.getWidth(); i++){
800                 stencilKernel(in,out,this->mask, this->args,i);
801             }
802         }
803 #endif
804
805 }//end namespace
806
807
808 #endif
```

Código 13 – interface_mppa.h

```
1  //--------------------------------------------------
2  // Copyright (c) 2015, Marcio B. Castro <marcio.castro@ufsc.br>
3  //
4  // All rights reserved.
5  //
6  // Redistribution and use in source and binary forms, with or
       without
7  // modification, are permitted provided that the following
       conditions are met:
8  //
9  // 1. Redistributions of source code must retain the above
       copyright notice, this
10 // list of conditions and the following disclaimer.
11 //
12 // 2. Redistributions in binary form must reproduce the above
       copyright notice,
13 // this list of conditions and the following disclaimer in the
       documentation
14 // and/or other materials provided with the distribution.
15 //
16 // 3. Neither the name of the copyright holder nor the names of
       its contributors
17 // may be used to endorse or promote products derived from this
       software without
18 // specific prior written permission.
19 //
20 // THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
21 // AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
22 // IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE ARE
23 // DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
       CONTRIBUTORS BE LIABLE
24 // FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL
25 // DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
       SUBSTITUTE GOODS OR
26 // SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
       INTERRUPTION) HOWEVER
27 // CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
       STRICT LIABILITY,
28 // OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
       WAY OUT OF THE USE
29 // OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
       DAMAGE.
30 //--------------------------------------------------
```

```
31
32 #ifndef __INTERFACE_MPPA_H
33 #define __INTERFACE_MPPA_H
34
35 #include <mppaipc.h>
36 #include <inttypes.h>
37
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <string.h>
41 #include <assert.h>
42 #include <sys/time.h>
43 #include <time.h>
44 #include <iostream>
45
46 /*
47  * GLOBAL CONSTANTS
48  */
49
50 #define BARRIER_SYNC_MASTER "/mppa/sync/128:1"
51 #define BARRIER_SYNC_SLAVE "/mppa/sync/[0..15]:2"
52
53 #define TRUE 1
54 #define FALSE 0
55
56 #define IO_NODE_RANK 128
57 #define MAX_CLUSTERS 16
58 #define MAX_THREADS_PER_CLUSTER 16
59 #define MPPA_FREQUENCY 400
60 #define BILLION 1E9
61 /*
62  * INTERNAL STRUCTURES
63  */
64
65 typedef enum {
66   BARRIER_MASTER,
67   BARRIER_SLAVE
68 } barrier_mode_t;
69
70 typedef struct {
71   int file_descriptor;
72   mppa_aiocb_t aiocb;
73 } portal_t;
74
75 typedef struct {
76   int sync_fd_master;
77   int sync_fd_slave;
78   barrier_mode_t mode;
79   int nb_clusters;
80 } barrier_t;
81
82 /*
83  * FUNCTIONS
84  */
85
86 void set_path_name(char *path, char *template_path, int rx, int
       tag);
87
88 portal_t *mppa_create_read_portal (char *path, void* buffer,
       unsigned long buffer_size, int trigger, void (*function)(
       mppa_sigval_t));
```

```
89  portal_t *mppa_create_write_portal (char *path, void* buffer,
        unsigned long buffer_size, int receiver_rank);
90  void mppa_write_portal (portal_t *portal, void *buffer, int
        buffer_size, int offset);
91  void mppa_async_write_portal (portal_t *portal, void *buffer,
        int buffer_size, int offset);
92  void mppa_async_write_stride_portal (portal_t *portal, void *
        buffer, int buffer_size, int ecount, int sstride, int
        tstride, int offset);
93  void mppa_async_write_wait_portal(portal_t *portal);
94  void mppa_async_read_wait_portal(portal_t *portal);
95  void mppa_close_portal (portal_t *portal);
96
97  barrier_t *mppa_create_master_barrier (char *path_master, char *
        path_slave, int clusters);
98  barrier_t *mppa_create_slave_barrier (char *path_master, char *
        path_slave);
99  void mppa_barrier_wait (barrier_t *barrier);
100 void mppa_close_barrier (barrier_t *barrier);
101
102 struct timeval mppa_master_get_time(void);
103 struct timespec mppa_slave_get_time(void);
104 double mppa_master_diff_time(struct timeval begin, struct
        timeval end);
105 double mppa_slave_diff_time(struct timespec begin, struct
        timespec end);
106
107 #endif // __INTERFACE_MPPA_H
```

Código 14 – interface_mppa.c

```
1  //-------------------------------------------------
2  // Copyright (c) 2015, Marcio B. Castro <marcio.castro@ufsc.br>
3  //
4  // All rights reserved.
5  //
6  // Redistribution and use in source and binary forms, with or
       without
7  // modification, are permitted provided that the following
       conditions are met:
8  //
9  // 1. Redistributions of source code must retain the above
       copyright notice, this
10 // list of conditions and the following disclaimer.
11 //
12 // 2. Redistributions in binary form must reproduce the above
       copyright notice,
13 // this list of conditions and the following disclaimer in the
       documentation
14 // and/or other materials provided with the distribution.
15 //
16 // 3. Neither the name of the copyright holder nor the names of
       its contributors
17 // may be used to endorse or promote products derived from this
       software without
18 // specific prior written permission.
19 //
20 // THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
21 // AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
```

```
22 // IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
23 // DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
24 // FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
25 // DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
26 // SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
27 // CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
28 // OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
29 // OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
30 //------------------------------------------------------
31
32 #include <mppa/osconfig.h>
33 #include "interface_mppa.h"
34
35 void set_path_name(char *path, char *template_path, int rx, int tag) {
36   sprintf(path, template_path, rx, tag);
37 }
38
39 /*************************************
40  * PORTAL COMMUNICATION
41  *************************************/
42
43 portal_t *mppa_create_read_portal (char *path, void* buffer, unsigned long buffer_size, int trigger, void (*function)(mppa_sigval_t)) {
44   portal_t *ret = (portal_t*) malloc (sizeof(portal_t));
45   int status;
46   ret->file_descriptor = mppa_open(path, O_RDONLY);
47   assert(ret->file_descriptor != -1);
48
49   mppa_aiocb_ctor(&ret->aiocb, ret->file_descriptor, buffer, buffer_size);
50
51   if (trigger > -1) {
52     mppa_aiocb_set_trigger(&ret->aiocb, trigger);
53   }
54
55   // Attention: we can't use callbacks with trigger/ mppa_aio_wait (bug?)
56   if (function)
57     mppa_aiocb_set_callback(&ret->aiocb, function);
58
59   status = mppa_aio_read(&ret->aiocb);
60   assert(status == 0);
61
62   return ret;
63 }
64
65 portal_t *mppa_create_write_portal (char *path, void* buffer, unsigned long buffer_size, int receiver_rank) {
66   portal_t *ret = (portal_t*) malloc (sizeof(portal_t));
67   ret->file_descriptor = mppa_open(path, O_WRONLY);
68   assert(ret->file_descriptor != -1);
```

```
69
70    // Tell mppa_io_write to wait for resources when sending a
          asynchronous message
71    assert(mppa_ioctl(ret->file_descriptor,
          MPPA_TX_WAIT_RESOURCE_ON) == 0);
72
73    // Select the DMA interface according to the receiver's rank.
74    // This is only possible on the IO-node!
75    if (__k1_get_cluster_id() == 128)
76      assert(mppa_ioctl(ret->file_descriptor, MPPA_TX_SET_IFACE,
              receiver_rank % 4) == 0);
77
78    // We need to initialize an aiocb for asynchronous writes.
79    // It seems that the buffer and buffer size parameters are not
          important here,
80    // because we're going to specify them with
          mppa_aiocb_set_pwrite()
81    // before calling mppa_aio_write()
82    assert(mppa_aiocb_ctor(&ret->aiocb, ret->file_descriptor,
          buffer, buffer_size) == &ret->aiocb);
83
84    return ret;
85  }
86
87  void mppa_async_read_wait_portal(portal_t *portal) {
88    int status;
89    status = mppa_aio_rearm(&portal->aiocb);
90    assert(status != -1);
91  }
92
93  void mppa_async_write_wait_portal(portal_t *portal) {
94    int status;
95    while(mppa_aio_error(&portal->aiocb) == EINPROGRESS);
96    status = mppa_aio_return(&portal->aiocb);
97    assert(status != -1);
98  }
99
100 void mppa_close_portal (portal_t *portal) {
101   assert(mppa_close(portal->file_descriptor) != -1);
102   free (portal);
103 }
104
105 void mppa_write_portal (portal_t *portal, void *buffer, int
          buffer_size, int offset) {
106   int status;
107   status = mppa_pwrite(portal->file_descriptor, buffer,
          buffer_size, offset);
108   assert(status == buffer_size);
109 }
110
111 void mppa_async_write_portal (portal_t *portal, void *buffer,
          int buffer_size, int offset) {
112   int status;
113   mppa_aiocb_set_pwrite(&portal->aiocb, buffer, buffer_size,
          offset);
114   status = mppa_aio_write(&portal->aiocb);
115   assert(status == 0);
116 }
117
118 /**************************************
119  * BARRIER
```

```
120  ************************************/
121
122 barrier_t *mppa_create_master_barrier (char *path_master, char *
        path_slave, int clusters) {
123   int status, i;
124   int ranks[clusters];
125   long long match;
126
127   barrier_t *ret = (barrier_t*) malloc (sizeof (barrier_t));
128
129   ret->sync_fd_master = mppa_open(path_master, O_RDONLY);
130   assert(ret->sync_fd_master != -1);
131
132   ret->sync_fd_slave = mppa_open(path_slave, O_WRONLY);
133   assert(ret->sync_fd_slave != -1);
134
135   // set all bits to 1 except the less significative "cluster"
          bits (those ones are set to 0).
136   // when the IO receives messagens from the clusters, they will
          set their correspoding bit to 1.
137   // the mppa_read() on the IO will return when match =
          11111...1111
138   match = (long long) -(1 << clusters);
139   status = mppa_ioctl(ret->sync_fd_master, MPPA_RX_SET_MATCH,
          match);
140   assert(status == 0);
141
142   for (i = 0; i < clusters; i++)
143     ranks[i] = i;
144
145   // configure the sync connector to receive message from "ranks
          "
146   status = mppa_ioctl(ret->sync_fd_slave, MPPA_TX_SET_RX_RANKS,
          clusters, ranks);
147   assert(status == 0);
148
149   ret->mode = BARRIER_MASTER;
150
151   return ret;
152 }
153
154 barrier_t *mppa_create_slave_barrier (char *path_master, char *
        path_slave) {
155   int status;
156
157   barrier_t *ret = (barrier_t*) malloc (sizeof (barrier_t));
158
159   ret->sync_fd_master = mppa_open(path_master, O_WRONLY);
160   assert(ret->sync_fd_master != -1);
161
162   ret->sync_fd_slave = mppa_open(path_slave, O_RDONLY);
163   assert(ret->sync_fd_slave != -1);
164
165   // set match to 0000...000.
166   // the IO will send a massage containing 1111...11111, so it
          will allow mppa_read() to return
167   status = mppa_ioctl(ret->sync_fd_slave, MPPA_RX_SET_MATCH, (
          long long) 0);
168   assert(status == 0);
169
170   ret->mode = BARRIER_SLAVE;
```

```
171
172   return ret;
173 }
174
175 void mppa_barrier_wait(barrier_t *barrier) {
176   int status;
177   long long dummy;
178
179   if(barrier->mode == BARRIER_MASTER) {
180     dummy = -1;
181     long long match;
182
183     // the IO waits for a message from each of the clusters
          involved in the barrier
184     // each cluster will set its correspoding bit on the IO (
          variable match) to 1
185     // when match = 11111...1111 the following mppa_read()
          returns
186     status = mppa_read(barrier->sync_fd_master, &match, sizeof(
          match));
187     assert(status == sizeof(match));
188
189     // the IO sends a message (dummy) containing 1111...1111 to
          all slaves involved in the barrier
190     // this will unblock their mppa_read()
191     status = mppa_write(barrier->sync_fd_slave, &dummy, sizeof(
          long long));
192     assert(status == sizeof(long long));
193   }
194   else {
195     dummy = 0;
196     long long mask;
197
198     // the cluster sets its corresponding bit to 1
199     mask = 0;
200     mask |= 1 << __k1_get_cluster_id();
201
202     // the cluster sends the mask to the IO
203     status = mppa_write(barrier->sync_fd_master, &mask, sizeof(
          mask));
204     assert(status == sizeof(mask));
205     printf("Barrier\n");
206     // the cluster waits for a message containing 1111...111
          from the IO to unblock
207     status = mppa_read(barrier->sync_fd_slave, &dummy, sizeof(
          long long));
208     assert(status == sizeof(long long));
209   }
210 }
211
212 void mppa_close_barrier (barrier_t *barrier) {
213   assert(mppa_close(barrier->sync_fd_master) != -1);
214   assert(mppa_close(barrier->sync_fd_slave) != -1);
215   free(barrier);
216 }
217
218 /*************************************
219  * TIME
220  *************************************/
221
222 static uint64_t residual_error = 0;
```

```
223
224 void mppa_init_time(void) {
225   uint64_t t1, t2;
226   t1 = mppa_get_time();
227   t2 = mppa_get_time();
228   residual_error = t2 - t1;
229 }
230
231 inline uint64_t mppa_get_time(void) {
232   return __k1_io_read64((void *)0x70084040) / MPPA_FREQUENCY;
233 }
234
235 inline uint64_t mppa_diff_time(uint64_t t1, uint64_t t2) {
236   return t2 - t1 - residual_error;
237 }
```