Paulo Ricardo Fernandes Ferrão

Licenciado em Engenharia Informática

Processamento de fluxos de dados em arquitecturas híbridas CPU/Intel® Xeon Phi

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Paulino, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri

Presidente: Doutora Carla Maria Gonçalves Ferreira
Arguente: Doutor Vasco Fernando de Figueiredo Tavares Pedro
Vogal: Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2017

Processamento de fluxos de dados em arquitecturas híbridas CPU/Intel® Xeon Phi

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

O meu agradecimento vai primeiramente para a Universidade que me acolheu e me deu orientações e alicerces para a construção das fundações daquela que espero venho a ser a minha vida profissional, tenho as peças, as ferramentas e o sentido de orientação, a partir daqui espero aproveitar, construir e contribuir na minha carreira profissional para melhorar como pessoa e contribuir na sociedade. Pelo apoio dado ao longo da construção desta dissertação ao professor Hervé Paulino, que tantas vezes me "deu na cabeça"para não desistir do meu projeto e ir mais longe. Finalmente à minha mãe porque me fez nascer e me atura nos meu devaneios. Obrigada UNL.

Resumo

O processamento de fluxos de dados (mais conhecido pela sua designação algo-saxónica Stream processing) reflete um paradigma de computação em paralelo, cujos conceitos possibilitam a implementação de soluções que funcionam sobre grandes quantidades de dados que são produzidos em alta frequência e que têm de ser processados rapidamente. Este paradigma é essencial para implementação de sistemas de monitorização, nomeadamente na bolsa, na saúde e na prevenção de fraude. Para suportar este modelo de computação em paralelo é necessário hardware com elevada capacidade de processamento paralelo, como por exemplo, computadores com muitos núcleos, ou *clusters* de computadores ou de GPUs. No entanto, a utilização destas plataformas para processamento de fluxos de dados levanta alguns problemas, como a latência da transmissão de dados entre os nós de um *cluster*, ou ausência de sincronização global em GPUs (o que implica transferências de controlo entre o GPU e o CPU).

Uma outra plataforma hardware para computação em paralelo é o co-processador Intel Xeon Phi, desenvolvido pela Intel, e baseado na arquitetura MIC. O Intel Xeon Phi é capaz de executar código x86, o que possibilitou que alguns sistemas de processamento de fluxos de dados, como o Intel Threading Building Blocks Flow Graph, fossem portados para execução no co-processador. No entanto, não existe nenhuma proposta que tire partido das características de um nó heterogéneo CPU(s)/Xeon Phi, no seu todo.

Nesse contexto, o objetivo desta dissertação é propor um sistema com a capacidade de executar um grafo de processamento de fluxos de dados em nós heterogéneos CPU(s)/Xeon Phi. Para tal, propõe-se a extensão da biblioteca Intel TBB FG, de forma a suportar esse novo tipo de ambientes com um impacto mínimo no seu modelo de programação. A proposta foi avaliada através da comparação de processamento de fluxo utilizando só o CPU com a utilização CPU/Xeon Phi, para medir os ganhos esperados. Nessas comparações foram detectadas situações, onde a segunda alternativa tinha grandes benefícios (*speedups*) sobre a outra.

Palavras-chave: Computação em Paralelo, Intel®Xeon Phi, TBB, Processamento de Fluxo de Dados

Abstract

Stream processing, also known as data-flow processing, is a parallel computing paradigm, whose concepts enable the implementation of solutions that handle large volumes of data generated at high rates and, thus, must be swiftly processed. This paradigm is widely used in the implementation of monitoring systems, namely in the stock market, health and fraud detection. The deployment of a stream processing system requires massively parallel hardware, such as clusters or graphics processing units (GPUs). However, the use of these platforms (for processing data flows) raises several issues such as the latency of intra-node communication in clusters, to the lack of global synchronization in GPUs (which implies control transfers between the GPU and the host).

The Intel Xeon Phi is a many-core processor based on Intel's Many Integrated Core (MIC) architecture and, thus, able to run x86 binary code. This property spawned the porting of some stream processing libraries to Intel Xeon Ph, such as Intel's Threading Building Blocks Flow Graph. However, there is no solution able to harness the individual characteristics and combined computing power CPUs and Intel Xeon Phi co-processors of hybrid CPU(s)/Intel Xeon Phi nodes.

In this context, the goal of this thesis is to propose a stream processing system capable of running on hybrid CPU(s)/Intel Xeon Phi nodes. For that purpose, we extended Intel's Threading Building Blocks Flow Graph library with the ability to run on this hybrid environment, with minimal impact on the library's programming model. Our proposal was evaluated from a performance perspective by comparison with stream processing on a conventional shared-memory multicore system. The results show significant performance boosts, specially when task granularity and work load increases.

Keywords: Parallel Computing, Intel®Xeon Phi, TBB, Stream Processing

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

1.1 Motivação

O processamento de fluxos de dados (mais conhecido pela sua designação algo-saxónica *Stream processing*) tem adquirido bastante popularidade nos últimos anos. Na origem desta popularidade está o potencial do modelo para expressar, de forma comparativamente simples, computações complexas para o processamento de volumes de dados, produzidos com altas frequências, e que devem ser processados rapidamente.

Alguns contextos de aplicação conhecidos deste tipo de computação são: 1. o processamento de grandes conjunto de dados (Big Data) com mínimo de latência, como o data-mining efectuado para complementar os motores de busca de busca da Microsoft, Yahoo e Google [14] [1]; 2. a determinação em tempo real de recomendações e anúncios por parte de grandes empresas de notícias e compras online como o Mercado Livre, o Spotify e o Flipboard [34]; 3. a determinação do melhor preço de um produto entre vários fornecedores [2]; 4. a avaliação de mercado como, por exemplo, faz a empresa de procura de viagens aéreas, Wego [34]; 5. o controlo de sistemas naturais como incêndios e cheias [2]; 6. a área da saúde, onde o processamento de dados alimenta sistemas de alerta de epidemias e de monitorização de sistemas de tratamento intensivo; 7. os sistemas de prevenção da fraude [2]; 8. a segurança com aplicações de monitorização com câmaras em tempo real e segurança cibernética [2].

O modelo de programação *Stream processing* requer que se expresse a computação como um grafo, onde as arestas representam fluxos de dados e os nós representam operações a serem executadas sobre elementos de tais fluxos. O grafo pode ter múltiplos pontos de entrada e várias ramificações que levam a terminações diferentes, como ilustrado na figura 1.1.

[1] https://cloud.google.com/dataflow/what-is-google-cloud-dataflow#Use-Cases

Cada estágio consome um, ou mais, elementos dos seus fluxos de dados de entrada, executa uma operação sobre esses elementos e gera (normalmente) um único valor para ser colocado num, ou mais, fluxos de dados de saída. A execução de um determinado estágio é desencadeada sempre que os necessários dados de entrada estão disponíveis e existem recursos computacionais suficientes. As execuções dos estágios decorrem normalmente em paralelo, sendo que a execução interna de cada estágio pode, ela mesmo, utilizar múltiplos fluxos de execução.

Saliente-se que a eficiência do grafo está interligada à granularidade das transformações, aplicadas aos dados, distribuídas pelos vários estágios e à capacidade de execução dos estágios em paralelo. Se a granularidade das transformações for baixa e/ou os estágios forem raramente executados em paralelo, então os custos das transferência dos dados entre os estágios poderão colocar em questão a eficiência perante outros paradigmas de computação em paralelo.

Nesse contexto, um sistema de processamento de fluxos de dados deverá ser suportado por hardware com alta capacidade de computação paralela.

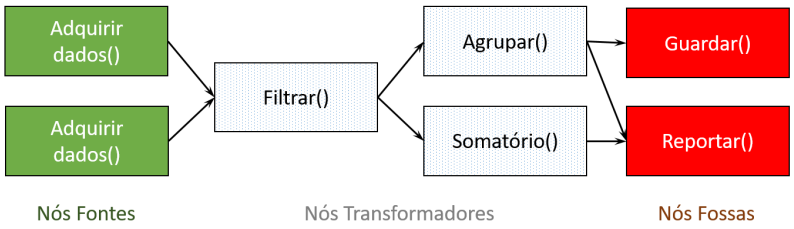

Figura 1.1: Exemplo de um grafo de processamento de fluxo de dados

Estão disponíveis várias bibliotecas para o processamento de fluxo de dados, para arquitecturas de memória partilhada. Exemplos conhecidos são o Threading Building Blocks Flow Graph (TBB FG) [36] e o RaftLib [5]. A aplicação destas ferramentas a problemas que lidam com volumes grandes de dados requerem computadores com muitos processadores.

Analisando o mercado de máquinas de memória partilhada existem máquinas com muitos processadores. No entanto, o custo de tal arquitectura é muito elevado. Por exemplo, no mercado convencional da Intel® é possível a aquisição de um processador com 24 núcleos, E7-8890 v4 [24], com *hyperthreading* e com a capacidade de partilhar a memória principal com outros 7 processadores, isto é, uma placa mãe terá a possibilidade de sustentar 8 processadores em simultâneo. A Intel recomenda a venda deste processador ao preço de $7174.00. Em comparação, é possível encontrar, na Amazon.com com o preço de $999, um processador Xeon Phi com o mesmo modelo que o que está presente num servidor da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FCT UNL), que foi utilizado para o desenvolvimento da dissertação.

Uma solução mais viável financeiramente, e mais fácil e eficientemente escalável, é a utilização de arquitecturas de memória distribuída, em que o sistema é composto por múltiplos nós com espaços de endereçamento distintos.

Estão disponíveis várias bibliotecas para o processamento de fluxo de dados, nomeadamente, para arquitecturas de memória distribuída. Exemplos conhecidos são o Apache Storm [39], o Spark Streaming [41], o Flink Streaming [7] e o Hadoop Streaming [15], que são capazes de utilizar o potencial computacional paralelo de vários computadores ligados em rede (*clusters*) para o processamento de dados. Importa realçar que uma parte desses *clusters* virtuais estão localizados na *cloud*, como são os exemplo dos fornecidos pelo Google Cloud e pela Amazon Cloud.

Apesar de vantajosa pela versatilidade e potencialidade de aumentar o escalonamento dos estágios, esta solução está circunscrita à capacidade económica de adicionar mais computadores ao *cluster* e assim sustentar um sistema de rede capaz de suportar eficientemente as transacções sobre grande volume de dados, sem se tornar um factor limitador do sistema (*bottleneck*). Normalmente, os sistemas de redes tolerantes a grandes cargas de dados são onerosos, um factor que, como já referenciado, é limitador e que pode condicionar a opção por essa solução.

Uma solução alternativa às duas anteriores é a utilização de aceleradores, quer isoladamente, quer como forma de aumentar a capacidade computacional individual dos nós de um *cluster* (sem de ter de ampliar a dimensão do *cluster* ou do peso sobre a comunicação na rede).

Nesta linha de raciocínio, refira-se que existem bibliotecas que aproveitam a capacidade computacional das unidades de processamento gráfico (GPUs) como o Brook [6], o Sponge [16] e o GPU Chariot [18]. Estas têm, no entanto, algumas limitações, das quais se salientam:

1. o código desenvolvido para executar em CPUs não é tipicamente compatível com os modelos de execução dos GPUs.

2. os GPUs não suportam sincronização global, apenas local (mais detalhes na secção 2).

No que se refere ao ponto 1, a biblioteca OpenCL é uma solução a essa limitação, pois esta biblioteca permite portabilidade do código fonte entre múltiplas arquitecturas, incluindo CPUs e GPUs. Note-se, contudo, que a portabilidade de desempenho não é garantida e que a programação em OpenCL requer conhecimentos de programação em paralelo nas arquitecturas especificas dos GPUs, pois estas diferem bastante dos CPUs.

No que se refere ao ponto 2, a única forma de efectuar uma sincronização global nos GPUs é com auxilio do sistema hospedeiro (*host*), neste caso o CPU. Mas para isso acontecer é necessário transferir o controlo para o *host* sempre que um *kernel* termine de aplicar as operações de um estágio sobre os dados. Como nota refira-se da possibilidade de reduzir a quantidade de trocas de controlo, através de técnicas que fundem múltiplos *kernels* num só [17, 31, 37]. No entanto, a sua aplicação ainda é limitada.

O Intel Xeon Phi [23] é um acelerador desenvolvido pela Intel com características que o distinguem dos GPUs. O seu processador é baseado na arquitectura múltiplos núcleos

integrados (MIC) [38], que consiste no aproveitamento da aglomeração de muitos núcleos de baixa capacidade computacional e de baixo consumo energético para computações altamente paralelizáveis. Esta arquitectura é semelhante aos GPUs, mas em contraste, os núcleos tem a capacidade de executar instruções x86, que permitem ao Xeon Phi correr qualquer código desenvolvido para o CPU e suportar um pseudo sistema operativo interno que permite sincronização global dos fios de execução.

Como nota de realce, a comunicação do Xeon Phi com o processador hospedeiro é estabelecida por uma interface `PCI express`. O Xeon Phi disponibiliza até 64 núcleos constituídos por uma unidade processamento escalar e vectorial, com capacidade de "*Simultaneous multithreading (SMT)*". Sendo assim, é possível usufruir de grande paralelização na execução dos estágios, com uma baixa latência na comunicação entre eles.

Dadas as suas características únicas, o Intel Xeon Phi apresenta-se como um acelerador que pode cooperar com o(s) CPU(s) do seu nó hospedeiro no processamento de fluxos de dados, oferecendo a sua grande capacidade de processamento sem requerer em troca o desenvolvimento de código extra por parte do utilizador. No entanto, no levantamento do estado da arte efectuado no contexto desta dissertação não se encontrou nenhuma proposta para o processamento de fluxos de dados em nós heterogéneos CPU(s) - Xeon Phi

1.2 Problema

Como referido no capítulo anterior, existem várias propostas para processamento de fluxo de dados em máquinas com memória partilhada (CPU), como o RaftLib e Intel TBB Flow Graph.

Devido à capacidade do Xeon Phi executar código x86 e linguagens como C, C++ e Fortran, a conversão de código de CPU para ser compatível com o co-processador é tipicamente simples. Além disso, a Intel oferece o TBB no seu SDK (Intel MPSS) para a programação do Xeon Phi. Portanto, o TBB Flow Graph é uma proposta oficial da Intel para processamento de fluxo de dados no Xeon Phi.

No entanto, este suporte oferecido pela Intel e os trabalhos de investigação estudados no decurso da elaboração desta tese, não permitem que parte dos estágios do grafo sejam executados pelo(s) CPU(s) e os restantes sejam executados pelo Xeon Phi. No nosso entendimento, esta abordagem simultânea oferece várias vantagens:

1. a capacidade de distribuir a computação consoante a frequência das chegadas do fluxo de dados. Por exemplo em horários de baixa frequência de fluxo de dados, é favorável a utilização apenas dos núcleos do CPU, para reduzir o consumo energético dos dois sistemas e aumentar a velocidade a que um bocado individual de fluxo de dados é processado; em horários de alta frequência, é sacrificada a velocidade a de processamento de um bocado individual de fluxo de dados pelo o aumento do processamento de fluxo de dados por um determinado momento.

2. a capacidade de utilizar tecnologias exclusivas a núcleos do CPU como por exemplo Quick Sync [22], com o co-processador;

Adicionalmente, a utilização conjunta dos núcleos do CPU e do Xeon Phi, também, permite a criação de grafos que utilizam simultaneamente os núcleos do CPU e Xeon Phi através do mecanismo *offload* [33] dos compiladores da Intel.

Esta solução requer, no entanto que:

- o CPU fique encarregue do controlo absoluto do grafo, o que pode prejudicar o desempenho, dado que se aproxima das limitações da sincronização global dos GPUs. Portanto, o grafo tem de ser inicializado e gerido no CPU;
- a utilização dos núcleos do Xeon Phi requer que um estágio no CPU envie os dados para o Xeon Phi e que aguarde pelo resultado, até que estes sejam processados pelo co-processador e enviados novamente para CPU. Tal implica, também, que trocas de dados entre estágios executados nos núcleos do Xeon Phi requerem sempre a intervenção do CPU.

Face à resenha feita e à compilação de vantagens e desvantagens das soluções anteriormente enunciadas e detalhadas, é proposto o desenho e a implementação de um sistema de processamento de fluxos de dados para estágios que:

- podem executar num CPU ou no Xeon Phi, tirando partido do poder computacional de ambos os tipos de processador;
- distribua a estrutura de controlo do grafo pelas várias plataformas de processamento;
- não requeira intervenção do CPU na comunicação entre estágios a executar no Xeon Phi;

A concepção e implementação desta proposta levanta diversos desafios, que devem ser ultrapassados por forma a garantir a viabilidade do sistema e deste modo permitir que possa competir com as outras implementações. Desses desafios salientamos:

1. **Execução paralela dos estágios** – o processamento de fluxo de dados é inerentemente paralelo, assim sendo o sistema deverá permitir que os diferentes estágios trabalhem simultaneamente sobre dados distintos do fluxo.
2. **Escalonamento dos estágios** - o sistema deve avaliar, planear e organizar vários processos de forma a escolher a distribuição que otimiza o desempenho do sistema.
3. **Comunicação entre os estágios** – a comunicação dos dados entre os estágios deverá ser o mais eficiente possível, de forma a minimizar a latência entre os vários estágios e as duas unidades de processamento, o CPU e o Xeon Phi.

4. **Paralelização dos estágios** - o sistema deverá, sempre que possível, permitir que a execução de um estágio utilize mais do que um fio de execução.

A proposta de resolução será detalhada na próxima secção, sendo esse o foco da dissertação.

1.3 Proposta

Esta dissertação endereça a concepção e implementação de um sistema de processamento de fluxo de dados, que utilize o CPU e o Xeon Phi, em que os estágios que requeiram ampla capacidade de computação paralela estarão no Xeon Phi, e os restantes ficarão no CPU, como ilustra a figura 1.2.

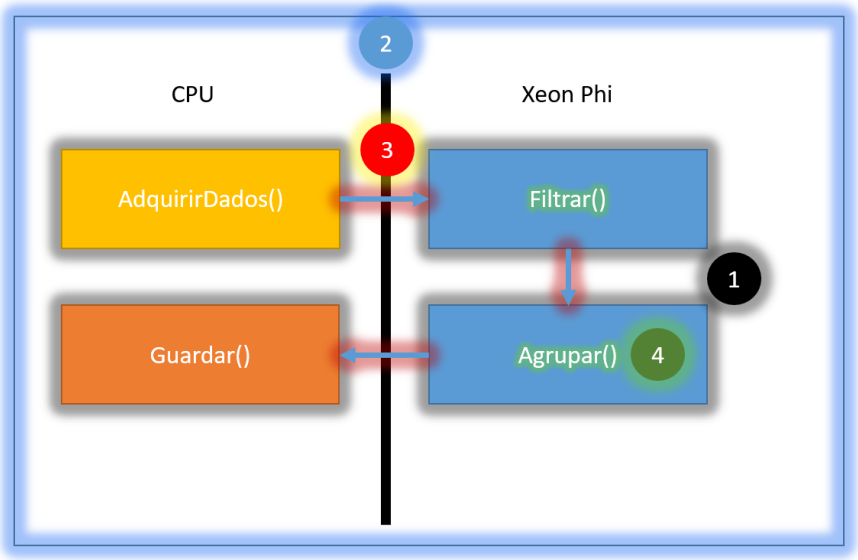

Figura 1.2: Ilustração do sistema proposto, com identificação dos vários desafios.

Iterando os desafios enumerados na secção anterior e sistematizando, no que se refere ao primeiro desafio (a execução paralela dos estágios) irá recorrer-se à biblioteca Intel TBB Flow Graph para programação paralela dos estágios. Esta biblioteca tem suporte nativo para Xeon Phi, o que permite resolver este desafio rapidamente e deste modo ser possível o enfoque nos restantes desafios.

No que se refere ao terceiro desafio (comunicação entre os estágios), o TBB não permite estabelecer comunicação entre estágios que estão a executar no Xeon Phi e outros que estão a executar no CPU. Será então necessário implementar uma nova funcionalidade ou estender as do TBB de forma a torná-lo possível. A Intel fornece uma API, SCIF [19], que permite trocar mensagens entre CPU e o Xeon Phi. Outra alternativa será o uso de um canal de comunicação TCP /IP, para estabelecer comunicação entre os dois componentes.

No que se refere ao quarto desafio (paralelização dos estágios), para se ter bons desempenhos no Xeon Phi é necessário utilizar o maior número de núcleos possível e tirar

partido da vectorização. O TBB Flow Graph permite a execução concorrente de diferentes estágios sobre diferentes elementos do fluxo de dados. Além disso, podem utilizar-se bibliotecas como OpenMP e Cilk quando o fluxo de dados não é suficiente para ocupar todos os núcleos. Por fim, o compilador da Intel inclui vectorização de código para o Xeon Phi. A nossa proposta vai no sentido de simplificar a programação de estágios que beneficiem dessa funcionalidade.

No que se refere ao segundo desafio (escalonamento dos estágios), o Xeon Phi disponibiliza parâmetros de execução, que podem ser modificados para controlar a forma como é feita a distribuição dos fios de execução pelos núcleos. No entanto, actualmente não há nenhum sistema de controlo, que faça a determinação automática da melhor forma de distribuição, dependente da computação, e que aplica essa distribuição em tempo de execução. A implementação de um sistema que seja capaz disso, aumentaria o desempenho oferecido pelo Xeon Phi, mas durante o decorrer na elaboração da dissertação não foi identificado um mecanismo que permitisse, de forma simples, o desenvolvimento de tal sistema. Por isso, o quarto desafio não foi resolvido.

1.4 Contribuições

As principais contribuições na terminação desta tese são:

- Uma biblioteca para processamento de fluxos de dados, com a capacidade de utilizar recursos de processador convencionais em simultâneo com co-processadores Xeon Phi da Intel®.
- Avaliação experimental da biblioteca, através da comparação entre grafos para processamento de fluxo de dados construídos com TBB que utilizam apenas núcleos do CPU, contra grafos construídos com a proposta extensão ao TBB, que utilizam núcleos do CPU e Xeon Phi.

1.5 Estrutura do Documento

- **Capítulo 2 – Enquadramento:** é apresentado o estado de arte. Começa por explicar o conceito de processamento de fluxo de dados e bibliotecas que suportam esse conceito.
- **Capítulo 3 – Processamento de fluxo de dados no Xeon Phi:** é explicado como a proposta resolve os desafios apresentados na secção 1.2. Depois é apresentado o modelo de programação. Por fim é detalhado aspectos importantes da implementação da proposta.
- **Capítulo 4 – Avaliação:** são descritas as métricas que se pretende avaliar na proposta. Em seguida são apresentados os casos de estudo que permitem a avaliação dessas métricas. Por fim, são feita a interpretação e conclusão dos resultados.

- **Capítulo 5 – Conclusões:** são apresentadas as conclusões desta dissertação, as suas limitações e o trabalho futuro.

Capítulo 2

Enquadramento

Neste capítulo é feita uma explanação do conceito de processamento de fluxo de dados, sendo seguida de uma descrição detalhada das características do Intel Xeon Phi. Posteriormente explica-se como as ferramentas fornecidas pelo kit de desenvolvimento de software (SDK) da Intel podem ser utilizadas na implementação da comunicação entre CPU e Xeon PHI e o processamento de fluxos de dados no Xeon Phi.

Por fim, são feitas referências a trabalhos, realizados por outros autores, para completar a dissertação. Os trabalhos referidos têm como foco o processamento de fluxo de dados em plataformas como CPUs e GPUs e estruturas de computação que utilizam o CPU e o Xeon Phi.

2.1 Processamento de fluxo de dados

Nesta secção, descreve-se de forma detalhada o paradigma de processamento de fluxo de dados. As informações apresentadas foram adquiridas nos livros [27] e [2].

2.1.1 Conceito

O processamento de fluxo de dados, também conhecido como *Data-Flow*, é um paradigma de computação, onde uma aplicação é vista como um encadeamento de estágios de computação.

Os estágios de computação trabalham em paralelo e de forma independente, mas interligados.

Cada estágio é independente, sendo possível representá-lo como um módulo que se acciona, isto é, que executa um código e consome recursos quando recebe dados de entrada suficiente para ativá-lo.

Toda a comunicação entre estes módulos ocorre através dos fluxos que entram nas portas de entrada e saem pelas portas de saída.

É possível ver um exemplo de uma aplicação de processamento de fluxo de dados na figura 2.1, que mostra vários tipos de estágios, como os fontes (sources), que introduzem fluxos de tokens de um certo tipo de dados para dentro do sistema.

Também são ilustrados os estágios de transformação que geram novos dados a partir do processamento dos fluxos de dados que recebem. Os estágios transformadores desempenham diferentes funções na aplicação [2], como adaptar, agregar, unir, particionar, reorganizar, aplicar operadores matemáticos, filtrar ou tarefas desenhadas pelo programador. E por fim, os estágios de fossa (sink), que recebem os fluxos, tipicamente no seu estado final, para armazenamento.

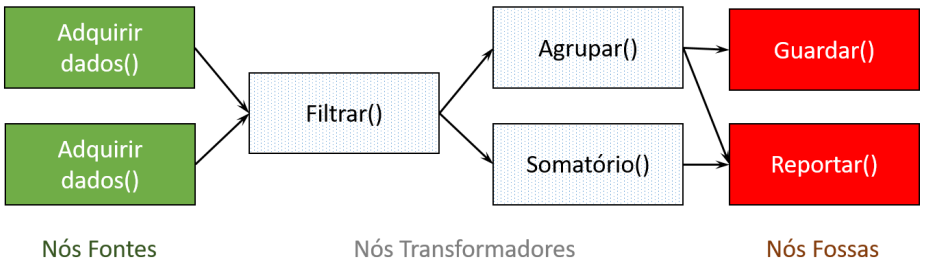

Figura 2.1: Ilustração de montagem de processamento de fluxo de dados

De imediato, as vantagens são visíveis:

- a possibilidade de modelar aplicações pelo modelo de processamento de fluxo de dados;
- o código é mais limpo, legível e tem a possibilidade de ser implementado com uso de diferentes linguagens de programação.

Outro benefício da independência dos estágios é a possibilidade de utilizar sistemas distribuídos na sua execução.

Por fim, o processamento de fluxo de dados facilita o paralelismo e tolerância de falhas relativamente a outros paradigmas.

A desvantagem de processamento de fluxo de dados são os custos associados com a transferência de dados entre os estágios, pois caso estes custos sejam muito dispendiosos, podem conduzir a penalização de desempenho.

Existem várias implementações do conceito processamento de fluxo de dados. Na próxima secção identificaremos algumas.

2.1.2 Bibliotecas

Existe um vasto leque de bibliotecas que já implementam o conceito de processamento de fluxo de dados. Estas destacam-se umas das outras pela maneira como disponibilizam

as suas interfaces aos utilizadores e pelo usufruir das plataformas (CPU, GPU, CLUSTER e Xeon Phi). Destacamos o Apache Storm [3], o Apache Spark [4], o Raftlib [5], o Brook [6] e o TBB [36].

De seguida iremos desenvolver cada uma desta bibliotecas.

Apache Storm [3] é uma biblioteca de código aberto, implementada em Clojure, que é capaz de processar fluxos de dados arbitrários, tendo sido inicialmente utilizada para processar mensagens do Twitter. O Storm executa computação em tempo real sobre fluxos de dados, da mesma maneira que o Hadoop fazia através de *batch processing*. Em Storm, os fluxos são compostos por sequências de tuplos, que contêm `integers`, `longs`, `shorts`, `bytes`, `strings`, `doubles`, `floats`, `booleans`, `byte array` ou tipos personalizados. Estes fluxos também são identificados com um identificador único.

Os estágios de source, são implementados como `spouts`. Os `spouts` geram tuplos, a partir de fontes externas, emitindo os mesmos para as topologias (explicado mais à frente).

Os `spouts` podem ser de dois tipos, confiável ou não confiável. O confiável é capaz de reintroduzir o tuplo no sistema, caso exista alguma falha no processamento. O não confiável não guarda informação gerada. Na declaração de um `spout` é necessário indicar a função geradora de tuplos e qual é o formato do seu *output* ou *outputs*. Depois disso o `spout` é capaz de emitir novo tuplo sempre que lhe é pedido e reportar o sucesso ou a falha no processamento de tuplos. Por fim, é capaz de parar enquanto o sistema permanece em funcionamento.

Os estágios transformadores e sink são representados pelos `bolts`. Nos `bolts` é necessário declarar a função que vai executar, e no caso desta emitir novos tuplos, é exigido a indicação dos tipos desses tuplos.

É possível implementar um `bolt` em linguagens que não o JAVA, para tal é necessário utilizar o método de sub-processo, que inicia outro processo dentro do `bolt`. Este outro processo pode ser escrito numa linguagem qualquer, desde que tenha um adaptador que faça a comunicação entre os processos, através de JSON.

A topologia, em Storm, consiste num grafo de computação, onde são declarados os `spouts` e os `bolts` que estarão no sistema.

Na declaração dos `spouts` é possível nomear e explicitar o número de processos que executarão o código desse `spout`. A declaração de um `bolt` é igual ao do `spout`, com a excepção que é necessário indicar os `bolts` e `spout` que fornecem tuplos de chegada para o `bolt` declarado .

Na referência das fontes dos tuplos de entrada, é crucial indicar como é feita a distribuição dos tuplos pelos diferentes processos que estão a executar o `bolt`.

Um *cluster* de Storm é composto por nós mestres e nós trabalhadores. Os mestres são responsáveis pela monitorização e distribuição de tarefas. Os nós trabalhadores executam partes da topologia. Estes últimos podem ser distribuídos por máquinas diferentes.

Por fim, o Storm tem tolerância a falhas, que garante que todos os tuplos serão executados pelo menos uma vez.

Apache Spark [4] é uma biblioteca de computação em *cluster*, cuja principal utilização é o processamento rápido de grandes volumes de dados. Disponibiliza interfaces de implementação em JAVA, Python e Scala e oferece vários componentes de computação como processamento de fluxo de dados (Spark Streaming), computação em grafo (GraphX) e computação com Sql (SparkSql).

Em contraste com o Apache Storm, que se foca na paralelização das tarefas, o Spark Streaming foca-se na paralelização dos dados. Sendo assim o conceito de estágio em Spark Streaming é representado por `DStream`.

O `DStream` é um fluxo contínuo de dados, composto por blocos de dados de intervalos de tempo diferentes. Num sistema de Spark Streaming, os dados entram através dos Receivers. Estes conseguem recolher dados por vários métodos, nomeadamente o sistema de ficheiros, as ligações por socket. Depois com os dados recolhidos são gerados `DStream`, que posteriormente poderão criar novos `DStream` através de transformações. As transformações disponíveis pelo Spark Streaming são o `map()`, `filter()`, `count()`, `reduce()`, `join()` e `transform()`.

Os valores do `DStream` podem ser guardados para a consola, para ficheiros texto, ficheiros de Hadoop, ou outro tipo de objetos de JAVA.

O Apache Spark garante que um fragmento de fluxo de dados (`DStream`) será executado apenas uma vez no sistema, ou seja, envios de fluxos de dados entre estágios, que terminam em erro, são repetidos até sucederem. Em contraste, o Apache Storm garante que todos os fragmentos de fluxo de dados (tuplos) são processados pelo sistema, mas não garante que o mesmo fragmento não seja processado duas ou mais vezes pelo mesmo estágio, ou seja, quando um fragmento é perdido durante a transmissão entre os estágios, este é reintroduzido no início do grafo.

Em ambas as bibliotecas, Apache Storm e Apache Spark, a comunicação de fluxo de dados pode ser efectuada por mensagens, que não são típicas em processamento de fluxo de dados.

Em Raftlib é uma biblioteca em que a comunicação é ponto a ponto e não tem replicação dos dados pelos vários estágios. O Raftlib [5] é uma biblioteca de processamento de fluxo de dados para C++, que tem como pontos de foco as filas de espera dinâmicas, paralelização automática, monitorização e particionamento otimizado por heurísticas.

Em [5] é mostrado o impacto de uma fila de espera que não tem o tamanho adequado para o seu estágio. Uma fila de espera muito pequena leva à paragem da aplicação, e uma fila de espera muito grande sobrecarrega a aplicação. Dado que não é possível ao operador modificar os tamanhos das filas, é necessário que a aplicação consiga adaptar-se e relatar ao utilizador por monitorização.

O Brook [6] é uma biblioteca de processamento de fluxo de dados no GPU, capaz de funcionar em hardware de AMD e Nvidia. No Brook, os estágios são denominados como kernels que executam funções sobre dados que recebem. Os `kernels` são mapeados para os `shaders` do GPU através de conversões de código. Já os fluxos de dados são mapeados para texturas de vírgula flutuante, em que os *kernels* acedem através de `streamRead` e `streamWrite`, ou seja, adquirem e submetem dados de texturas.

É possível obter melhor desempenho no GPU que o CPU, quando os estágios mapeados exigem computações complexas sobre extensos dados.

Intel TBB (Threading Building Blocks) é uma biblioteca, código aberto, para C, disponível para vasto leque de plataformas, incluído o Xeon Phi. O TBB foi desenvolvido para promover escalabilidade e paralelismo, através: de algoritmos de paralelização de ciclos (`parallel_for`); algoritmos de paralelização de grafos (`parallel_pipeline`); contentores concorrentes (`concurrent_hashmap`); exclusão mútua (`spin_mutex`); operadores atómicos; alocações de memória (`scalable_allocator, cachealigned_allocator`); gestores de tarefas.

A ferramenta do TBB que permite implementar sistemas de processamento de fluxo de dados é o Graph Flow.

Esta biblioteca permite a criação de grafos sequenciais ou com estágios paralelizáveis.

No TBB Graph Flow, um sistema de processamento de fluxo de dados é representado por um grafo que é composto por nós que desempenham diferentes funções e por arestas que ligam os vários nós, permitindo troca de mensagens de dados entre nós.

Os nós são definidos por quatro atributos:

- Pela função atribuída pelo utilizador. Esta função será utilizada para converter as mensagens que o nó recebe em mensagens que nó irá emitir para os nós sucessores.
- Pelo grafo a que o nó pertence.
- Pelo número máximo de invocações da função, em paralelo, que o nó pode usufruir para distribuir as mensagens que recebe.

Estes nós têm uma política de espera em que só emitem novas mensagens pelas portas de saída se existirem nós com portas de entrada livres para as receber.

Caso contrário, as mensagens são guardadas num `buffer`.

O TBB disponibiliza alguns nós predefinidos: `source_node`, nó que gera fluxos de dados e depois envia como mensagens através da porta de saída única; os `function_node`, `continue_node` e `multifunction_node`, que são nós que aplicam uma função a dados, que entram nas suas portas de entrada em forma de mensagem e emitem o resultante dessa função pelas portas de saída.

A diferenciação entre os três tipos de nós é que o `continue_node` é capaz de aguardar até receber o número determinado de mensagens, para depois aplicar a sua função nesse conjunto. O `multifunction_node` tem mais que uma porta de saída, onde consegue emitir

diferentes mensagens; o split_node parte as mensagens de chegada e envia os pedaços pelas suas várias portas de saída; o `join_node` combina mensagens de chegada de portas de entrada diferentes para forma uma mensagem única que emite pela porta sua única porta de saída.

As arestas são criadas pela função `make_edge` que recebe dois parâmetros, um nó precedente e um nó sucessor.

O XKaapi [29] é uma biblioteca de computação em paralelo com foco as tarefas e que inclui o suporte de fluxo de dados entre as tarefas. Foi adaptado para funcionar no Xeon Phi no modo nativo.

Em conclusão, existe muito trabalho desenvolvido sobre processamento de fluxo de dados, no entanto, bibliotecas como o Apache Spark e o Apache Storm não são compatíveis com os processadores Xeon Phi, pois este apenas executa código `C/C++` e Fortran.

Face ao exposto anteriormente, resta o recurso ao TBB FLow Graph ou o Raftlib, pois ambos estão implementados em C++.

Contudo, a Intel garante a compatibilidade do `TBB flow graph` no Xeon Phi, factor que torna o TBB numa biblioteca mais atractiva para desenvolvimento da proposta, pois remove possíveis alterações necessárias à biblioteca `Raftlib` para funcionamento no Xeon Phi.

Apesar do TBB poder utilizar mecanismos como o `offload` com recurso aos núcleos do CPU e do Xeon Phi, este tem duas desvantagens comparativamente à nossa proposta:

1. o grafo só pode ser inicializado e existir no CPU, o que implica que os estágios source nunca poderão estar no Xeon Phi.

2. para utilização dos núcleos do Xeon Phi é necessário que um estágio no CPU envie os dados para o Xeon Phi e que aguarde pelo dados, até que estes sejam processados pelo co-processador e enviados novamente para CPU.

3. a passagem de dados entre dois estágios que utilizam os núcleos do Xeon Phi não é possível sem a intervenção do CPU.

2.2 Xeon Phi

O Intel Xeon Phi é o primeiro processador, desenvolvido pela Intel, baseado na arquitectura de múltiplos núcleos integrados (MIC). Foi lançado no mercado em 2013 como um co-processador para uso geral beneficiando do processamento em paralelo, em resposta a outros populares aceleradores como os produtos da série Tesla da Nvidia.

Para uma melhor compreensão dos desafios que se colocam nesta dissertação, nesta secção descrevemos as características gerais do processador. As informações apresentadas foram adquiridas nos livros [25], [35], [8], [20], e nos fóruns da Intel [1].

[1] https://software.intel.com/en-us/forums/intel-many-integrated-core

2.2.1 Características gerais

O Intel Xeon Phi apresenta uma arquitectura de múltiplos núcleos integrados (MIC), compatível com o conjunto de instruções x86-64. Assim sendo, as aplicações desenvolvidas para CPUs, que são compatíveis com o conjunto de instruções x86, podem ser facilmente portadas para executar no Xeon Phi.

Para pôr em funcionamento esse código no Xeon Phi é apenas necessário compilá-lo com os compiladores fornecidos pela Intel, com o parâmetro indicativo que o código tem de ser executável no Xeon Phi. Porém, para beneficiar da potencia computacional do Xeon Phi, o código deverá ser optimizado para a arquitectura dos seus núcleos.

Ao longo desta secção detalhamos as características essenciais do processador no que refere à comunicação com o sistema hospedeiro, à comunicação entre os múltiplos núcleos, à arquitectura interna dos núcleos e à hierarquia da memória.

Apresentamos também o que é esperado dos novos modelos do Xeon Phi.

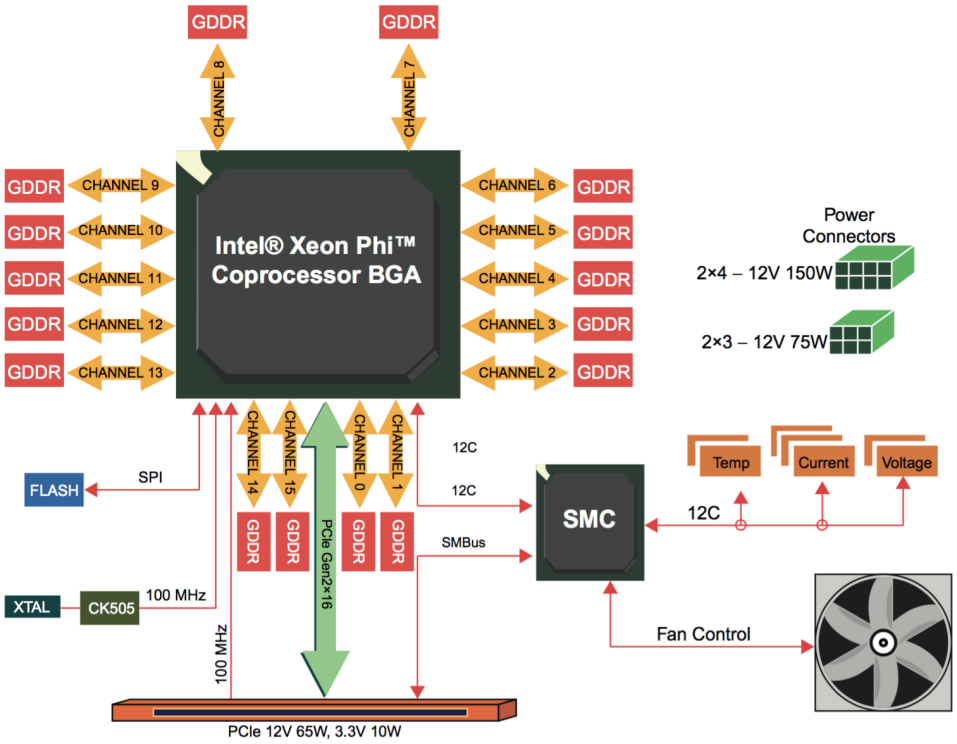

Figura 2.2: Esquema da composição do Xeon Phi [25]

2.2.2 Comunicação

O Xeon Phi comunica com o sistema hospedeiro em que se insere através da interface PCI Express. Mais precisamente, através de uma interface PCI Express Gen2 x16, com capacidade de transferência de pacotes de dados com tamanho de 256 bytes.

Casos de estudo como [13] e [35] demonstraram que a velocidade máxima de transferência de dados, entre sistema hospedeiro e o Xeon Phi, é de cerca de 6.8 GB/s, que não

se encontra muito distante da velocidade teórica máxima de 8 GB/s.

Este interface pode também ser utilizado para a comunicação entre múltiplos co-processadores Xeon Phi presentes no sistema hospedeiro, sem que seja necessária intervenção do CPU.

A comunicação interna no Xeon Phi é feita através de um anel bidirecional, chamado On-die Interconnect (ODI). O ODI figura (2.3) divide-se no total em dez anéis distintos. Em cada direção existe um barramento (bus) de dados de 64 bytes, dois barramentos de endereços e dois barramentos para controlo (*acknowledge*). Os utilizadores desta via de comunicação são os núcleos, canais de memória (*memory channels*), interface *PCIe* e as *Tag Directories*. Como é habitual, este hardware é transparente para o software que executa no processador.

A ligação em anel bidireccional foi escolhida sobre a ligação linear por esta disponibilizar múltiplos pontos de atendimento de pedidos à memória principal feita pelos núcleos, e por possibilitar circulação desses pedidos simultaneamente no anel.

Por oposição, numa ligação linear, só há um ponto de atendimento sendo somente um pedido a poder circular na ligação. A implementação de um anel bidireccional é mais complexa e onerosa do que a de um linear, mas como o Xeon Phi é composto por dezenas de núcleos, este custo é compensado em ganhos de desempenho.

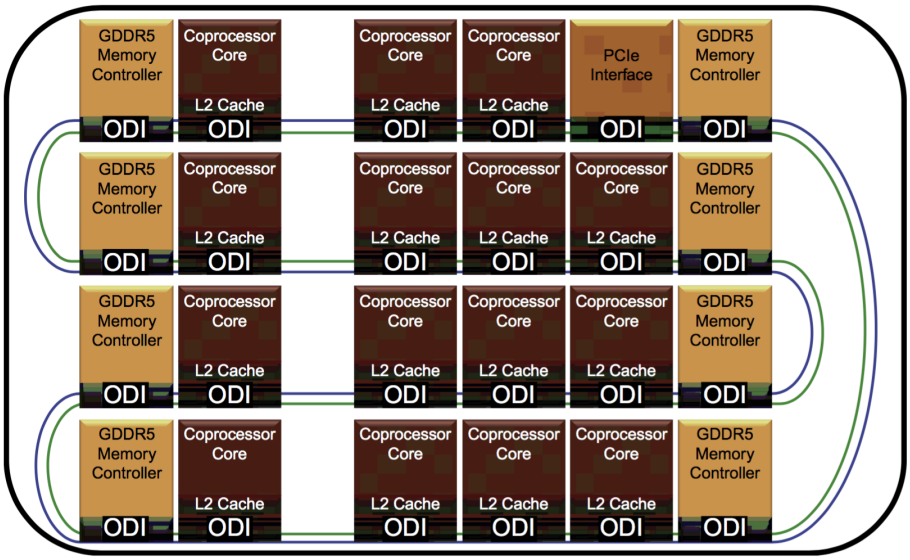

Figura 2.3: Ilustrativa do On-die Interconnect (ODI) [25] pag-247

2.2.3 Hierarquia de Memória

A hierarquia de memória do Xeon Phi pode ser representada num modelo em pirâmide ilustrada na figura 2.5, onde na base está a memória com maior capacidade, mas de menor velocidade, e ao subir encontramos memórias com maior velocidade, mas menor capacidade.

Nesse contexto, na base está a memória principal, que consiste em 6 ou 8 GB do tipo GDDR5. No nível imediatamente acima temos a memória *cache* L2 de núcleos adjacentes, depois temos memória *cache* L2 de 512KB do próprio núcleo e por fim memória *cache* L1 de 64 KB do próprio núcleo.

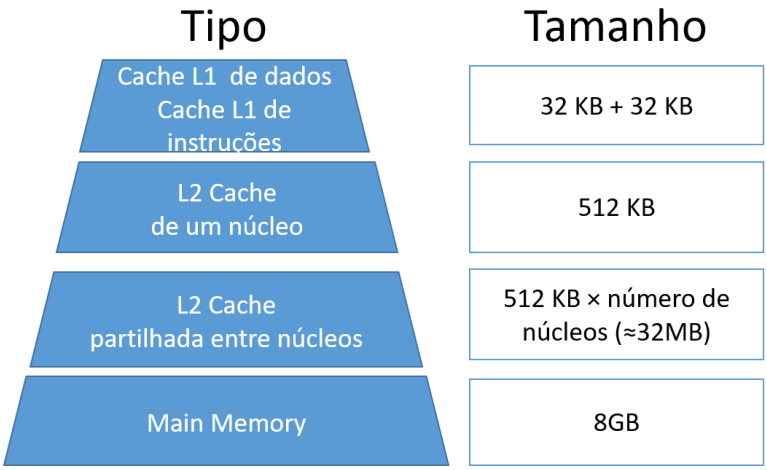

Figura 2.4: Hierarquia da memória Xeon Phi [21]

A memória *cache* L1 divide-se em 32KB para instruções e outros 32KB para dados. As caches são associativas por grupos, com grupos de 8 blocos, e têm um tempo de acesso de 3 ciclos de relógio.

A memória *cache* L2 de cada núcleo tem 512KB de capacidade, correcção de erros e, novamente, grupos de 8 blocos. O tempo de acesso é de 11 ciclos de relógio.

O seu conteúdo inclui o conteúdo da *cache* L1 [2]. Outra propriedade destas *caches* L2 é que não restringem quanto ao uso de *page size* de tamanhos diferentes. Tal permite que o programador modificar o tamanho do page size para beneficiar de acessos mais rápidos através de page size pequenos. Acresce o benefício de maior *cache hit* através de *page size* grandes.[3] A tabela 2.1 resume as várias propriedades das cache do Xeon Phi como tempo de acessos, associatividade e tamanhos das linhas.

Devido ao amplo número de núcleos, a Intel, adicionou ao Xeon Phi, componentes com função de Distributed Tag Directories (TDs), responsáveis pela coerência das *caches*.

[2] A *cache* L2 é portanto inclusiva
[3] https://software.intel.com/en-us/forums/intel-many-integrated-core/topic/586138

Parâmetro	L1	L2
Tamanho	32 KB + 32 KB	512 KB
Associatividade	8-way	8-way
Tamanho da linha	64 bytes	64 bytes
Bancos	8	8
Tempo de acesso	1 ciclo	11 ciclos
Política	pseudo LRU	pseudo LRU

Tabela 2.1: Parâmetros da cache do Xeon Phi

Ora esta funcionalidade reflecte que estamos perante um protocolo de coerência de *caches* baseado em diretórios. Estes componentes também permitem que um núcleo possa aceder à memória *cache* L2 de um outro núcleo quando ocorre uma *cache miss* na sua própria memória L2, pois uma TD reconhece que este pedido pode ser satisfeito por uma memória *cache* L2 desse mesmo núcleo. Caso não exista esse reconhecimento, o pedido é enviado para os controladores de memória que gerem a memória principal. Este tipo de acesso tem o mesmo custo de um acesso à memória principal, sendo o único benefício a redução de acesso a memória.

A memória principal é composta por Graphic Double Data Rate 5 (GDDR5). Memórias do tipo GDDR5 são especializadas para computação de alto desempenho, permitindo maiores capacidades de transferências, em troca de latências superiores e limitação do endereçamento de 32-bits.

O acesso à memória principal é feito através dos controladores de memória que são compostos por 2 canais de memória, com capacidade de transferência de 5.5 GT/s.

Tendo então em consideração que existem entre 6 a 8 controladores de memória, em teoria seria possível alcançar velocidades de 320 GB/s. No entanto, estudos feitos em [13], provam que velocidade de leitura e escrita é directamente proporcional ao número total de *threads*, que estão a fazer pedidos de dados. Mas a partir de 60 ou mais fios de execução, a velocidade de transferência atinge o seu máximo de 164 GB/s em leitura e de 120 GB/s em escrita.

2.2.4 Núcleo

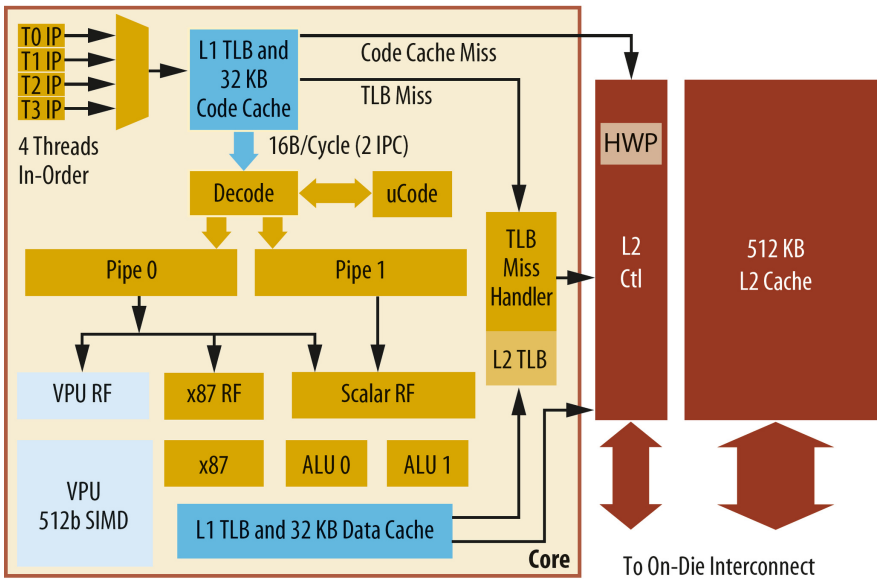

Figura 2.5: Xeon Phi Silicon Core architecture [25] pag-249

Dependendo do modelo, o Xeon Phi pode estar equipado com 57, 60 ou 61 núcleos, em que cada núcleo tem uma frequência de, respectivamente, 1.0 Ghz, 1.1 GHz, ou 1.3 GHz, com possibilidade de um aumento de 100 MHz com a tecnologia *Intel Turbo Boost*. A composição interna dos núcleos, figura 2.5, é equivalente em todos os modelos. Cada núcleo suporta de execução simultânea de 4 fios de execução, onde cada fio de execução é enviado, pelo método *round robin*, para uns dos dois pipes de execução, V-pipe and U-pipe. Isto permite a execução de duas instruções vectoriais (SIMD) em paralelo, se as instruções forem de fios de execução diferentes.

Em suma e com base nas considerações anteriores é recomendada a atribuição de 2 a 3 fios de execução para cada núcleo, para que se possa atingir a melhor performance possível. Sendo a atribuição de 4 fios de execução para cada núcleo, apenas recomendada quando não existem mais núcleos livres.

Cada núcleo tem uma unidade de execução escalar capaz de executar instruções em x86 ou x87, que são fornecidas pelo U-pipe. Dispõe ainda de uma unidade de execução vectorial capaz de executar instruções do tipo 512-bit Vector ISA (Instruction-Set Architecture), que permite o processamento de instruções de vírgula flutuante de precisão simples e dupla e inteiros de 32/64-bit. As instruções que o Vector Unit executa são fornecidas tanto pelo V-pipe como pelo U-pipe.

No Xeon Phi é possível utilizar duas metodologias para distribuição de fios de execução para cada núcleo: dispersão ou compactação.

Pelo método de dispersão, cada núcleo recebe apenas um fio de execução até tal não ser mais possível. Este método é benévolo para computações que são *communication-bound*, ou seja, requerem constantemente acessos a memória principal, pois cada núcleo terá

apenas de fazer um pedido de cada vez ao controlador de memória mais próximo, o que conduz a menor tempo de atendimento possível.

Pelo método de compactação, cada núcleo recebe dois ou mais fios de execução. Este método é cordial para computações *CPU-bound*,quando é requerida mais computação de cada núcleo e se seja possível tomar partido das partilhas de memórias nas *caches*.

2.2.5 Sistema Operativo

O sistema operativo do Xeon Phi é baseado no *kernel* padrão do Linux da kernel.org, com ligeiras alterações, para compensar a ausência de componentes de hardware que o Xeon Phi não possui, perante uma plataforma computacional convencional.

Este *kernel* é responsável pela gestão de processos, tarefas, memoria, energia, configuração e dispositivo. Acresce que este sistema operativo tem acesso a bibliotecas de padrão do Linux como `libc`, `libm`, `librt`, `glibc`, `libuitl` e `libstdc++`. Caso seja necessário, é possível adicionar novas bibliotecas ao sistema operativo, desde que estas sejam compatíveis ou adicionar novas funcionalidades através de módulos de *kernel*. Também é possível usufruir de sistemas operativos implementados por terceiros.

O sistema operativo é executado num único núcleo, igual aos outros núcleos que o Xeon Phi dispõe, no entanto não é recomendada a utilização desse núcleo.

O Xeon Phi tem a particularidade de permitir a execução de processos diferentes simultaneamente, sem a necessidade de sincronização entre os processos, contrariamente ao GPU.

No Xeon Phi é possível estabelecer canais de comunicação entre processos que estão a correr no co-processador com outros processos que estão a executar no CPU hospedeiro. Esta comunicação será mais detalhada na secção 3.1.

A programação em Xeon Phi é feita em C/C++ e Fortran, sendo que a Intel disponibiliza um SDK com ferramentas para programação paralela, nomeadamente, os sistemas OpenMP [10], Cilk++ [28], TBB [36] e MPI [32].

2.2.6 Modelos recentes

Em 2016, a Intel lançou uma nova geração de processadores com base na arquitectura MIC, conhecidos como Xeon Phi x200 ou pelo código The Knights Landing. Os processadores da nova coleção x200, ao contrário dos x100, não estão contidos numa placa de expansão. Em vez disso, os novos processadores estão contidos num *chip* que comunicam com a placa mãe pelo *socket*, especificamente um *socket 3647*.

Com esta passagem de papel de co-processador para processor principal, os novos processadores ganharam algumas novas características como a possibilidade de comunicar directamente com a interface de rede e a capacidade de expandir a memória.

A expansão de memória tem de ser feita através de *sticks* do tipo DDR4-2133 ou DDR4-2400 dependente do modelo do Xeon Phi. É possível expandir até 6 *sticks* de *RAM*, até ao máximo de 384 GB de *RAM*. Além da memória expansível, os novos processadores

tem uma memória interna contida no *chip*. Esta memória interna é do tipo *MCDRAM*, que é um tipo de memória onde os *chips*, em vez de serem colocados numa placa na horizontal, são empilhados. Este novo processo de construção reduz o espaço ocupado pela memória e aumenta eficiência energética e velocidade de transferência.

No contexto de computação, os transístores dos novos processadores separam-se por 14 nm, isto é, utilizam uma litografia de 14 nm, o que possibilita maiores frequências e menor necessidade de energia.

Os novos *chips* tem até 72 núcleos com frequências até 1.5 GHz, até 1.7 GHz com *turbo* e 2 unidades de processamento vetorial. A tecnologia Intel Turbo Boost, presente nos novos processadores, é a mesma usada nos processadores convencionais, o que significa que apenas é possível obter frequências de turbo quando no máximo 3 núcleos estão em uso.

Em computações domésticas, onde o número de núcleos dos processadores utilizados varia entre 4 a 10, esta tecnologia tem algum impacto. Mas a utilidade desta tecnologia em computações em paralelo com 72 núcleos é questionável.

Na tabela é possível ver as diferenças entre o co-processador usado durante a dissertação e um da nova colecção com um preço semelhante.

Xeon Phi modelo	Núcleos	Fios de exec.	Frequência	L2 cache	Memoria Interna	Expansão de Memória	Máxima computação	TDP
5110P	64	240	1.1 GHz	30 MB	8 GB de GDDR5 Trans: 320 GB/s	Não	1 Teraflop	225
7210F	64	256	1.3 GHz	32 MB	16 GB MCDRAM Trans: 400 GB/s	384 GB 6 canais DDR4-2133 Trans: 102 GB/s (6x17GB/s [4])	3.4 Teraflops	215

Tabela 2.2: Comparação entre modelo 5110P e o 7210F.

Em conclusão, os novos modelos de processadores Xeon Phi são mais eficientes e têm capacidade de trabalhar sobre maior quantidade de dados, graças a expansibilidade de memória e acesso directo a controladores de rede. Por tal, os novos modelos são ainda mais atractivos para processamento de fluxo de dados, principalmente quando versões de placas de expansão (AIB) dos novos processadores saíram, pois permite combinar o grande poder de computação em série e mecanismos exclusivos do CPU, com o enorme poder de computação em paralelo dos co-processadores Xeon Phi.

2.2.7 Modelos de execução

O SDK da Intel oferece três tipos diferentes de execução de computação no Xeon Phi, o Offload, o Simétrico e o Nativo.

O modelo de execução Offload é utilizado quando um processo principal, que está a ser executado no hospedeiro, tira proveito do co-processador para executar regiões ou funções em vez de executá-las no hospedeiro. Estas regiões ou funções tem de ser identificadas

[4]Máxima velocidade transferência de DDR4 2133 é 17 GB/s

previamente para que o compilador tome conhecimento de modo a compilar essas funções ou regiões para funcionarem no hospedeiro e no Xeon Phi.

Para fruir deste modelo, a Intel oferece pragmas, que permitem enviar funções para serem executadas no co-processador (pragma `offload`), ilustrado na figura 2.6. Também é possível a transferências de dados, isto é enviar e receber, para memória através de indicadores (`pragma offload_transfer in out`), ou alocar ou reutilizar memoria (pragma offload_transfer nocopy). As transferências podem ser assíncronas através dos pragmas (`signal`) e (`wait`).

Outro método de transferência de dados é MYO (Mine Yours Ours). Neste modelo a explicitação das variáveis a ser partilhadas entre o CPU e o co-processador é simplificada sendo possível partilhar objetos entre os dois. Para isso é utilizada a directiva `_Cilk_shared` x, em que x denota o objecto que se pretende partilhar entre CPU e o co-processador: função ou variável. A directiva `_Offload_shared` permite desencadear a executar a função partilhada no co-processador.

As desvantagens deste tipo de transferências, tanto *offload* como MYO, é que sincronização das variáveis partilhadas só ocorre durante o início e fim do *offload* da função ou região, que afecta a variável. Logo não é, por exemplo, possível fazer *offload* de uma função que vá processando um fluxo de dados proveniente do CPU.

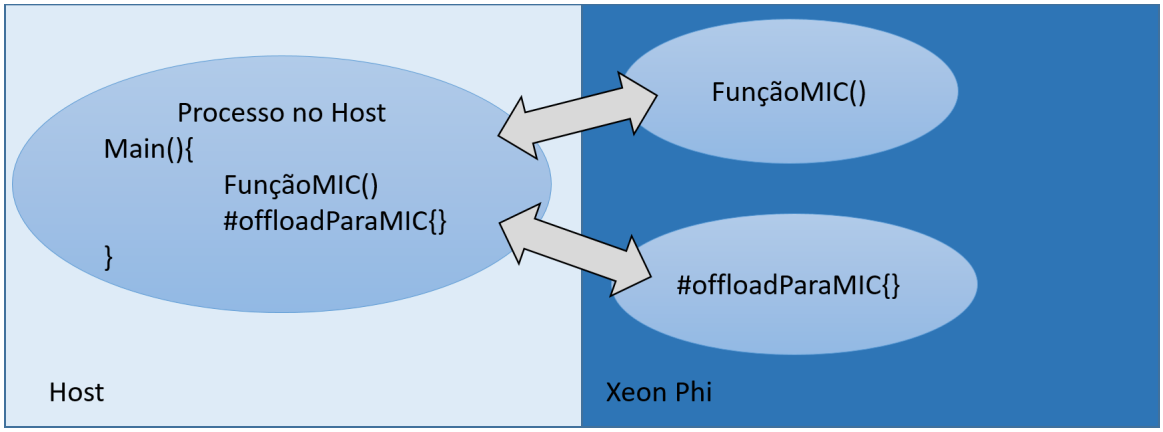

Figura 2.6: Ilustração de um modelo execução Offload

O modelo de execução nativa tem um programa principal que executa unicamente no Xeon Phi, ilustrado na figura 2.7. Existem duas maneiras de utilizar este tipo de modelo. Uma é através do comando `ssh` para o processador Xeon Phi e lançar o programa. Outra é utilizar o comando `nativeloadex`, disponível no SDK da Intel Xeon Phi, para lançar remotamente o programa no Xeon Phi. Este comando verifica se o Xeon Phi tem as bibliotecas necessárias para executar o programa.

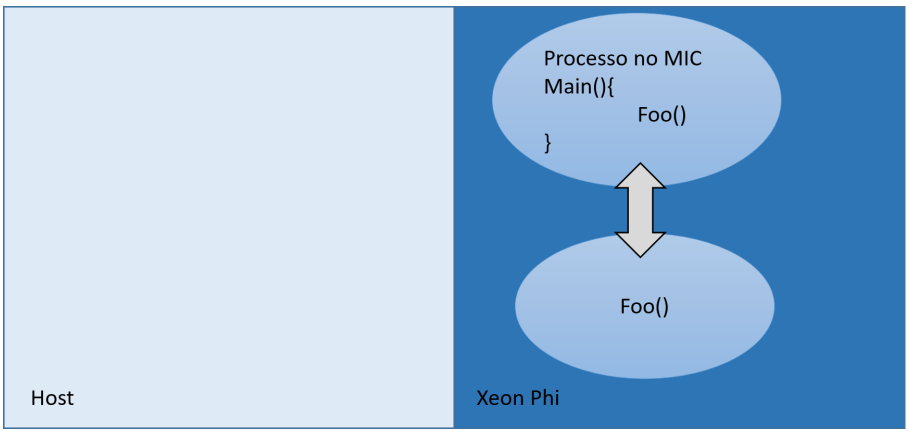

Figura 2.7: Ilustração de um modelo execução Nativa

O modelo de execução simétrico permite o lançamento de vários processos, uns no hospedeiro e outros no co-processador e estes comunicam e coordenam-se através de trocas de mensagens, utilizando bibliotecas como o SCIF e o MPI, ilustrado na figura 2.8

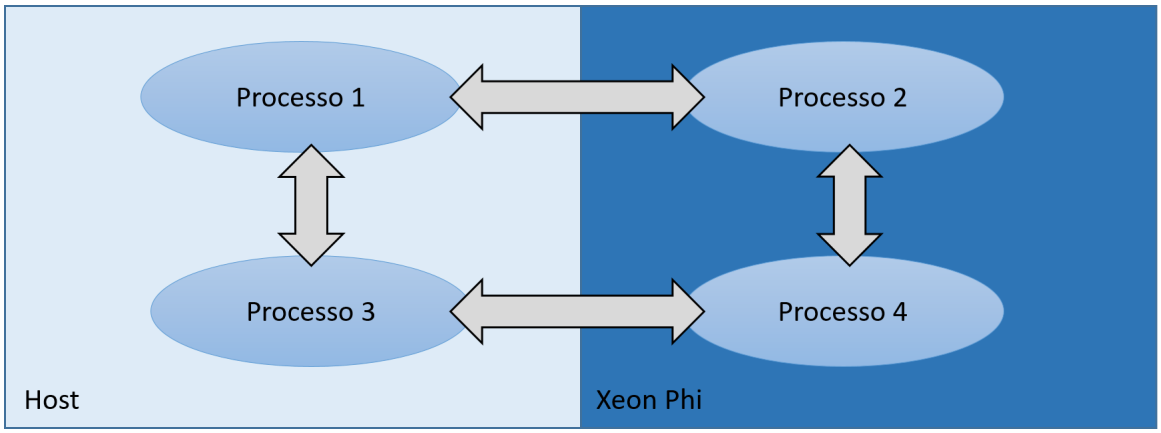

Figura 2.8: Ilustração de um modelo execução Simétrico

O MPI é uma especificação que tem várias implementações, que disponibilizam uma interface de comunicação entre processos, tipicamente, de sistemas distribuídos.

A implementação disponibilizada pela Intel é IntelMPI 5.1, que é capaz de fornecer melhor desempenho do que OpenMPI, em *clusters* sustentados por *hardware* da Intel.

Em MPI, uma computação é composta por vários processos que executam num conjunto de máquinas dado à partida. O sistema de execução encarrega-se de lançar os processos nas diversas máquinas, desde que lá esteja a correr o serviço que permita tal funcionalidade.

O `standard` oferece funções para comunicação síncrona, assíncrona, bloqueante, não bloqueante, operações colectivas e distributivas (gather, scather, reduce, broadcast). Os processos executam partes diferentes do programa que é determinado pelo MPI Rank que lhes é atribuído.

Caso comum de uso de MPI é onde um processo com *rank* 0 distribui trabalho para os restantes processos com *rank* com um valor numérico superior. Estes executam o trabalho e algumas vezes trocam mensagens entre os outros processos para completar o trabalho, e por fim devolvem a resultante ao processo de *rank* 0.

O SCIF é uma biblioteca que proporciona mecanismos de comunicação entre nós, que podem ser co-processadores ou o processador hospedeiro. SCIF abstrai os detalhes de comunicação entres as duas plataformas, feita pelo PCIe bus, numa interface equivalente para ambas plataformas. A todos os nós de SCIF é atribuído um identificador único durante o arranque do sistema, que depende da ordem de descobrimento da plataforma, excepto o nó do hospedeiro que é sempre atribuído o identificador 0.

Também existem portas de SCIF associadas a cada nó, que são identificadas pelo identificador do nó e um inteiro de 16-bit. Para estabelecer comunicação entre os nós, é necessário associar-se a uma porta através de `scif_open()`, `scif_bind()`. Depois, enquanto, um nó espera por uma ligação com `scif_listen`, o outro nó tenta estabelecer comunicação através da função `scif_connect()`. Estabelecida a comunicação, o nó que estava a espera, valida a ligação através da função `scif_accept()`. Após disso os nós podem trocar mensagens com `scif_send()`, e `scif_recv()`. Estas mensagens podem ser de $2^{31} - 1$ bytes.

Um exemplo de um projeto que utiliza a biblioteca SCIF para programar a comunicação entre processos que estão no CPU com os que estão no Xeon Phi é o HyPhi [12]. O HyPhi [12] é uma biblioteca que estende do TBB, de maneira a permitir que este utilize os núcleos do hospedeiro e do co-processador para métodos como `for_each` e `map_reduce` do TBB.

A sincronização das tarefas, distribuídas pelos métodos `for_each` e `map_reduce`, e transferência de dados são feitas através da biblioteca SCIF.

2.3 Sumário

Neste capítulo foram explicados os conceitos relativos ao processamento de fluxo de dados, como estágios sources, sinks e transformadores e em que o processamento de fluxo de dados é utilizado. Faz-se a introdução às bibliotecas já existentes para desenvolver o processamento de fluxo de dados no Xeon Phi, com foco no Threading Building Blocks (Intel® TBB), cuja compatibilidade com Xeon Phi é garantida pela Intel.

Depois, foi detalhada a tecnologia que envolve os processadores Xeon Phi, a composição dos núcleos, *caches*, interfaces de comunicação, bibliotecas de desenvolvimento de código, de modo a salientar as diferenças entre processadores convencionais e o processador Xeon Phi.

Dado o levantamento tecnológico que foi feito neste capítulo 2, no próximo capítulo 3 vamos endereçar como se podem atacar efetivamente os desafios de construir um sistema de processamento de fluxo de dados em nós CPU/Xeon Phi

Capítulo 3

Processamento de fluxo de dados Híbrido

Neste capítulo descreve-se um sistema para o processamento de fluxos de dados em sistemas híbridos (CPU/Xeon Phi) que resolve os vários desafios enunciados na secção 1.2. Posteriormente é descrito o modelo de programação que a proposta oferece ao utilizador. Finalmente, são são apresentados os detalhes de implementação mais relevantes.

3.1 Panorâmica Geral da Solução

Nesta secção damos uma panorâmica geral da solução que iremos detalhar ao longo deste capítulo, justificando as escolhas das tecnologias que serviram de base para o trabalho desenvolvido.

Execução Paralela dos Estágios no `MIC`: Depois do levantamento apresentado no Capítulo 2, uma das primeiras decisões tomadas no desenvolvimento desta dissertação foi a escolha da plataforma que serviria de base para o nosso sistema de processamento de fluxos de dados em arquitecturas híbridas `HOST/MICx` (CPU/Intel Xeon Phi).

Havia duas grandes possibilidades para a base do sistema. A primeira seria utilizar bibliotecas para sistemas distribuídos como o Apache Spark ou Apache Storm. Mas para usufruir de qualquer destas duas bibliotecas para a construção de grafos no `MIC` seria necessário desenvolver uma versão da máquina virtual do Java [9] que fosse compatível com a arquitectura do `MIC`. Para ultrapassar este problema, os grafos teriam que ser montados do lado do `Host` (CPU), através das bibliotecas Storm ou Spark, com os nós (estágios) a efectuar chamadas de `offload` aos núcleos dos `MIC`. Isto é possível, pois o Storm possibilita a programação de nós (`bolts`) com a capacidade de executar processos escritos em linguagens como C++. No caso do Spark, como este é compatível com Python, seria possível utilizar o mecanismo `offload` através do módulo pyMic [26]. No entanto,

esta solução teria alguns condicionalismos, nomeadamente nos mesmos problemas que as soluções de processamento de fluxo de dados em GPUs sofrem, isto é, a sincronização apenas seria possível ao nível do HOST.

A segunda possibilidade seria utilizar *bibliotecas* que já permitem a execução de computações no MIC, como por exemplo, Streamit, Raftlib e TBB FG. Estas perdem a integração "de graça" com as populares *bibliotecas* para sistemas distribuídos, mas podem efectivamente tirar partido das características do MIC.

Analisadas as vantagens e desvantagens de cada uma das hipóteses 1 e 2, a nossa escolha recaiu sobre a segunda possibilidade, devido às desvantagens anteriormente enunciadas na primeira solução.

Entre as bibliotecas possíveis, o Intel Threading Buildings Blocks Flow Graph (TBB FG) foi a escolhida. Esta biblioteca tem como benefício o usufruto por muitos utilizadores para além de ser suportada e mantida pela Intel®, que é a mesma companhia que desenvolveu e comercializa a arquitectura MIC. Para além disso, acresce que deste modo não é exigível que utilizador tenha de adquirir *software* adicional, excepto o *software* necessário para lidar como Xeon Phi, que é o Intel® SDK. Adicionalmente temos alguma garantia que haverá suporte e compatibilidade de todas as funcionalidades do TBB FG para as versões correntes e futuras do MIC. Facto que contribui para longevidade da proposta e para futuros trabalhos.

Paralelização Interna dos Estágios: No contexto da paralelização de computação, cada estágio do TBB FG é por um lado, executado de forma concorrente com os restantes estágios e por outro lado com ele próprio, se o utilizador pretender e se houver mais que um *Input* a chegar a esse estágio.

Ainda no contexto da paralelização, a vectorização do código executada no MIC é ainda mais importante do que nos processadores convencionais, pois as instruções vectoriais do Xeon Phi trabalham sobre 512-bit (64 bytes), em contraste com 256-bit ou 128-bit das extensões, AVX e SSE, dos processadores convencionais.

Além disso, o utilizador pode aproveitar outras bibliotecas, disponíveis com Intel® SDK, como o OpenMP ou o Cilk, que permitem paralelizar computação dentro dos estágios.

Comunicação entre Estágios no MIC e no Host: A versão corrente do TBB FG da Intel® não permite ligar estágios que estejam a ser executados no MIC com os estágios no Host, ou vice-versa. Para se implementar tal comportamento é preciso desenvolver um sistema de troca de mensagens (fluxo de dados) entre os estágios no MIC e no Host.

Há possibilidade de fazer trocas de dados entre o Host e MIC, através de mecanismos como offload ou _Cilk_shared da biblioteca CILK. No entanto, estes mecanismos são limitados na capacidade de troca de dados no sentido inverso.

Em alternativa, é possível utilizar protocolos de comunicação como TCP/IP, SCIF e Coprocessor Offload Infrastructure (COI) [11] para as trocas de dados entre os dois sistemas, sem limitações como na alternativa anterior.

Depois de uma análise detalhada, a nossa escolha recaiu sobre o SCIF, por ser uma biblioteca mais optimizada e abstracta que as outras duas alternativas, sendo mais optimizada para trocas de dados com sistemas MIC do que o TCP/IP, já que o TCP/IP é simulado através de COI. A abstracção vem do facto que SCIF abstrair uma implementação COI por trás.

O trabalho desenvolvido em HyPhi, também tem como recurso a biblioteca SCIF para implementação da comunicação entre os processos no Host e nos MICs (Xeon Phis).

3.2 Modelo de programação

O modelo de programação da proposta oferece praticamente as mesmas operações que são possíveis com a biblioteca Intel TBB FLow Graph (TBB FG), mas com uma distinção. A distinção é que nós (estágios) do grafo podem estar no HOST ou no MIC. A localização, <Location> dos nós é indicada com valores ∈ {HOST, MICx}, com x a poder tomar valores 0,1,..., de maneira a indicar qual o MIC pretendido. Os grafos são representados pela a class graph, que depois poderão suportar uma diversidade de nós, detalhados no próximo capítulo 3.2.1.

3.2.1 Nós

Todos os nós implementados na proposta são instâncias de classes *templated* que herdam da classe com o mesmo nome da biblioteca TBB FG e da classe base_node<Location>. Exemplo do template do function_node, um nó que aplica um functor a Input, é descrito na listagem 3.1. Nesse exemplo, estão duas outras classes, sender e receiver, que estendem as classes com o mesmo nome do TBB FG e base_node<Location>.

```
1  template<
2    typename Input,
3    typename Output,
4    location Location = Host,
5    typename Policy = tbb::flow::queueing,
6    typename Allocator = tbb::cache_aligned_allocator<Input> >
7  class function_node : class function_node<Input, Output, Location, Policy,
       Allocator> :
8    public tbb::flow::function_node<Input, Output, Location ,Policy, Allocator>,
9    public sender<Output, Location>,
10   public receiver<Input, Location>
```

Listagem 3.1: Template da classe function_node.

Outros nós implementados foram:

- `template< typename Output, location Location >`
  `class source_node`
- `template< typename OutputTuple, location Location >`
  `class join_node`
- `template < typename T, location Location,`
  `typename Allocator = tbb::cache_aligned_allocator<T> >`
  `class queue_node`
- `template < typename InputTuple , location Location >`
  `class split_node`

Estes nós são, respectivamente, um tipo de nó que gera fluxo de dados para grafo, um tipo de nó que recebe dois fluxos de dados e junta num tuplo, um tipo de nó que guarda uma fila de fluxo de dados, por fim, um tipo de nó que decompõe um tuplo.

Os parâmetros do `template` são descritos na tabela 3.1.

Parâmetro do template	Significado	Valores possiveis
typename Input	O tipo dos fluxo de dados recebidos pelo estágio.	int, bool, * Object, ...
typename Output	O tipo dos fluxo de dados enviados pelo estágio.	int, bool, * Object, ...
location Location	Localização do estágio	HOST, MIC0, MIC1, ...
typename Policy	Política de rejeição ou acumulação de valores.	queueing, rejecting
typename Allocator	O tipo da classe que implementa o alocador.	int, double,...
typename OutputTuple	O tipo tuplo resultante.	tuple<int,int>, ...
typename InputTuple	O tipo do tuplo que será decomposto.	tuple<Object, bool>, ...

Tabela 3.1: Descrição dos vários parâmetros dos templates.

3.2.2 Arestas

A interface da função `make_edge`, apresentada na listagem 3.2, do TBB FG é inalterada. No entanto, consoante a localização dos nós dados como argumentos, a função construí arestas locais ou baseadas em `SCIF`.

```
1 template< typename T >
2 inline void make_edge( sender<T> &p, receiver<T> &s );
```

Listagem 3.2: Template da função make_edge .

3.2.3 Comunicação Host <-> MIC

Os sistemas `HOST` e `MIC` não partilham endereçamentos de memória, por isso é necessário transferir dados entre sistemas, sempre que dados têm de fluir entre arestas que ligam nós no `Host` e `MIC`.

Os tipos básicos, como `int` e `double` não se querem implementação especial do utilizador, mas para objectos mais complexos é necessário que esses objectos implementem uma interface com uma função de envio e uma de recepção. Em secções futuras haverá exemplificação das funções enviar e recebimento da classe vector, que é uma classe que guarda vários valores com alinhamento mais correcto dependendo do sistema.

3.2.4 Construção de um Grafo

No TBB FG o grafo é a classe que guarda os vários nós dos estágios. Assim sendo, é esta classe que assume a responsabilidade do lançamento de processos dos estágios.

Existem milhares de possíveis construções de grafos. É possível construir um grafo de encadeamento de nós no `Host`, ilustrado na figura 3.1 com a listagem em 3.3. O mesmo grafo mas no `MIC`, ilustrado na figura 3.2 com a listagem em 3.4. Um grafo com alguns nós no `Host` e no `MIC`, ilustrado na figura 3.3 com a listagem em 3.5. Por fim, um grafo com um nó, separar, para o `Host` e para o `MIC`, ilustrado na figura 3.4 com a listagem em 3.6. Realce que a modificação necessária para definir o local é um argumento.

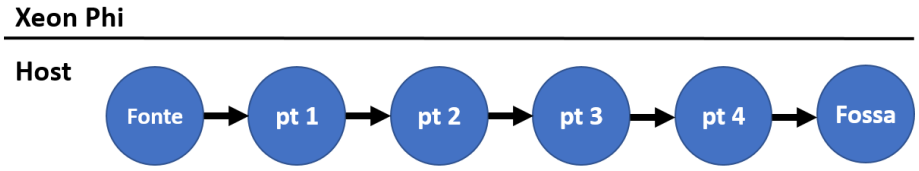

Figura 3.1: Um grafo com nós todos no CPU

```
1  using namespace marrow;
2  using namespace marrow::flow;
3
4  graph g;
5  source_node<vector<int>*> fonte (g, fonte_f())
6
7  function_node<vector<int>*,vector<int>*> pt1(g, unlimited, f1());
8  function_node<vector<int>*,vector<int>*> pt2(g, unlimited, f2());
9  function_node<vector<int>*,vector<int>*> pt3(g, unlimited, f3());
10 function_node<vector<int>*,vector<int>*> pt4(g, unlimited, f4());
11
12 function_node<vector<int>*> fossa (g, unlimited, fossa_f());
13
14 make_edge(fonte, pt1);
15 make_edge(pt1, pt2);
16 make_edge(pt2, pt3);
17 make_edge(pt3, pt4);
18 make_edge(pt4, fossa);
```

Listagem 3.3: Implementação da Figura 3.1

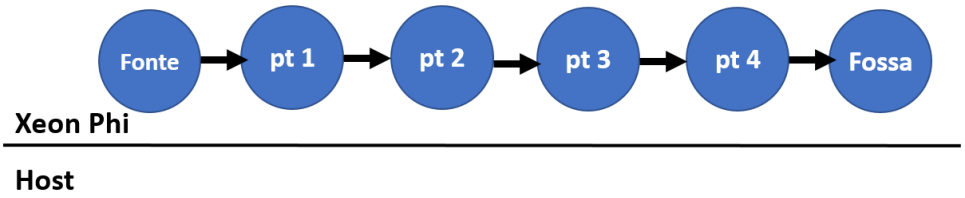

Figura 3.2: Um grafo com nós todos no MIC

```
using namespace marrow;
using namespace marrow::flow;

graph g;
source_node<vector<int>* , MIC0> fonte (g, fonte_f())

function_node<vector<int>*,vector<int>*, MIC0> pt1(g, unlimited, f1());
function_node<vector<int>*,vector<int>*, MIC0> pt2(g, unlimited, f2());
function_node<vector<int>*,vector<int>*, MIC0> pt3(g, unlimited, f3());
function_node<vector<int>*,vector<int>*, MIC0> pt4(g, unlimited, f4());

function_node<vector<int>* , MIC0> fossa (g, unlimited, fossa_f());

make_edge(fonte, pt1);
make_edge(pt1, pt2);
make_edge(pt2, pt3);
make_edge(pt3, pt4);
make_edge(pt4, fossa);
```

Listagem 3.4: Implementação da Figura 3.2

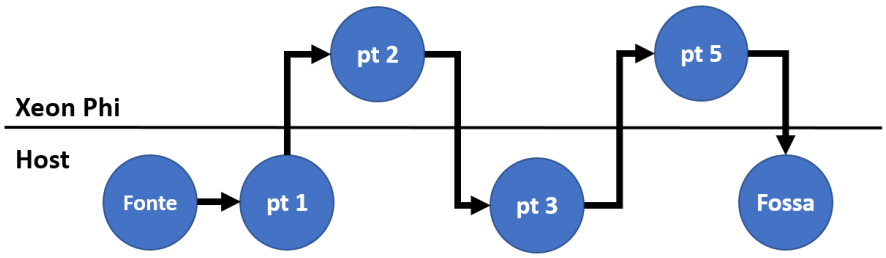

Figura 3.3: Um grafo com alguns nós no HOST e outros no MIC

```
1  using namespace marrow;
2  using namespace marrow::flow;
3
4  graph g;
5  source_node<vector<int>* , MIC0> fonte (g, fonte_f())
6
7  function_node<vector<int>*,vector<int>* > pt1(g, unlimited, f1());
8  function_node<vector<int>*,vector<int>* , MIC0> pt2(g, unlimited, f2());
9  function_node<vector<int>*,vector<int>* > pt3(g, unlimited, f3());
10 function_node<vector<int>*,vector<int>* , MIC0> pt4(g, unlimited, f4());
11
12 function_node<vector<int>* > fossa (g, unlimited, fossa_f());
13
14 make_edge(fonte, pt1);
15 make_edge(pt1, pt2);
16 make_edge(pt2, pt3);
17 make_edge(pt3, pt4);
18 make_edge(pt4, fossa);
```

Listagem 3.5: Implementação da Figura 3.3

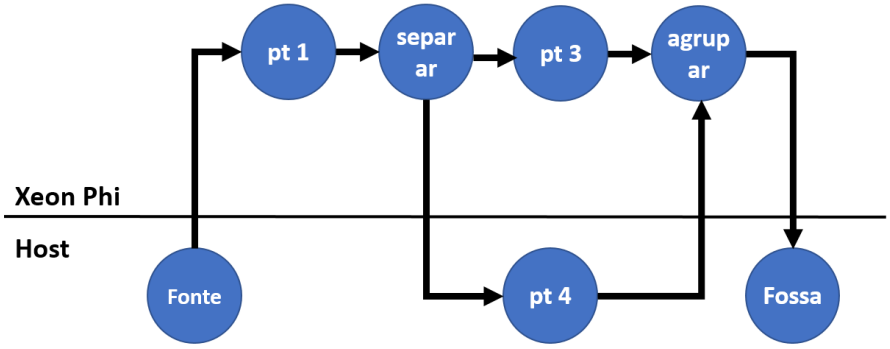

Figura 3.4: Um grafo com alguns nós de tipos diferentes no HOST e outros no MIC

```
1  using namespace marrow;
2  using namespace marrow::flow;
3  typedef std::tuple< vector<int>*, vector<int>* > out_tuple;
4
5  graph g;
6  source_node<vector<int>*> fonte (g, fonte_f())
7  function_node<vector<int>*,vector<int>*, MIC0> pt1(g, unlimited, f1());
8  split_node<out_tuple, MIC0> separar(g);
9  function_node<vector<int>*,vector<int>*, MIC0> pt3(g, unlimited, f3());
10 function_node<vector<int>*,vector<int>*> pt4(g, unlimited, f4());
11 join_node<out_tuple, MIC0> agrupar(g);
12 function_node<out_tuple> fossa(g, unlimited, fossa_f());
13
14 make_edge(fonte, pt1);
15 make_edge(pt1, separar);
16 make_edge(output_port<1>(separar), pt3);
17 make_edge(output_port<0>(separar), pt4);
18 make_edge(pt3, input_port<0>(agrupar));
19 make_edge(pt4, input_port<1>(agrupar));
20 make_edge(agrupar, fossa);
```

Listagem 3.6: Implementação da Figura 3.4

3.2.5 Programação dos Nós

A programação dos nós na proposta é equivalente a programação no TBB FG. Isto é, através de um functor C++, desde que o functor receba como argumento uma cópia do valor de um fluxo de dados do tipo `Input` e devolva um valor do tipo `Output`. Dentro da função operação, o comportamento a aplicar é livre, no entanto, para o melhor desempenho no MIC é extremamente recomendado o uso das capacidades de vectorização de cada núcleo individual. Para aproveitamento das capacidades de vectorização é necessário ter atenção a operações sobre múltiplos dados que sejam independentes umas das outras, o alinhamento dos dados e auxiliar o compilador a encontrar zonas do código mais favoráveis

com instruções vectoriais. Apresentam-se de seguida dois exemplos de ciclos, na listagem 3.7, onde num é possível a vectorização e onde noutra não é possível, com a reportagem por parte do compilador.

```
1 for (int i = 1; i < n; i++) {
2   a[i] = a[i+1];
3 }
4
5 for (int i = 1; i < n; i++) {
6   a[i] = a[i-1];
7 }
```

Listagem 3.7: Implementação de dois ciclos, um com possibilidade de vectorização e noutro que não há possibilidade.

```
1 line(2): remark: LOOP WAS VECTORIZED
2 line(2): remark : --- begin vector loop cost summary ---
3 line(2): remark : scalar loop cost: 50
4 line(2): remark : vector loop cost: 12
5 line(2): remark : estimated potential speedup: 4.2
6 line(2): remark : --- end vector loop cost summary ---
7 line(6): remark: loop was not vectorized: existence of vector dependence
8 line(6): remark: loop was not vectorized: cannot vectorize empty loop
```

Listagem 3.8: Reportagem resultante do compilador de c++ da Intel®, iccp, com os argumentos -qopt-report=5, depois de compilar código em 3.7.

3.2.6 Integração com o Marrow

O Marrow [1, 30, 40] é uma biblioteca de esqueletos de algoritmos para computações complexas, principalmente em ambientes de multi-GPU. Exemplos de alguns esqueletos são `map`, `loop` e `reduce`. Os esqueletos `map` e `reduce` foram desenhados com operações sem dependência de dados, o que leva a implementações vectorizadas. Segue-se uma listagem 3.9 de dois functor, com objectivos iguais, em que se pretende incrementar cada elemento de um vector, mas com o primeiro utiliza `loop` do Marrow.

A vantagem da utilização dos esqueletos do Marrow é que é possível ocultar detalhes ao programador como a vectorização e possivelmente também a paralelização.

3.3 Detalhes de Implementação

Nesta secção é detalhado como os conceitos de grafo, nós, arestas e vector foram implementados.

3.3.1 Grafo

O grafo do `marrow::flow` funciona como um embrulho sobre dois grafos, um no `HOST` e outro no `MIC`. Ao inicializar a classe `graph` da proposta, um `graph` é gerado no `HOST`, como

```
1  class node {
2    vector<int> * operator()(vector<int>* a){
3      marrow::map< marrow::incr<int> > (*a,1);
4      return a;
5    }
6  }
7  class node2 {
8    vector<int> * operator()(vector<int>* a){
9      for(int i = 0 ;i<a.size(); i++){
10       a[i]++;
11     }
12     return a;
13   }
14 }
```

Listagem 3.9: Implementação de dois functores, com o primeiro a utilizar a biblioteca do Marrow.

ilustrado na linha 9 da listagem 3.10, depois é feita uma chamada do mecanismo `offload` para o MIC, onde é criado um grafo no lado do MIC, linha 16. Seguidamente é criado um apontador para `void` que irá apontar para o grafo gerado no MIC e esse apontador é copiado para o HOST, linha 17 e 21.

Sempre que for necessário fazer uma chamada ao grafo no MIC, esse apontador para `void` terá de ser alterado para apontador de grafo dentro do mecanismo `offload`, linha 31. A razão da utilização de apontadores para `void` em vez de apontadores de grafo, é que durante a execução do código existe uma tabela que faz a conversão de apontadores no HOST para seus equivalentes no MIC e vice versa durante o uso do mecanismo `offload`. Ora essa tabela só funciona dentro do scope.

Por isso imaginemos que passávamos um apontador de grafo no HOST, com um equivalente no MICx para uma função. Essa função não teria acesso ao grafo no MICx, pois não tem presente a tabela de conversão. No entanto, os apontadores para `void` são excluídos da conversão, logo é possível utilizar estes apontadores fora de scope, pois o endereço de memória que o apontador aponta não é convertido.

A `class graph` contém dois atributos do tipo `bool`, que começam com valor `true` e uma variável só está presente na instância do objecto no HOST e a outra na instância no MICx, linha 3 e 4 . Os nós responsáveis pela transferência entre HOST e MICx têm acesso através de apontadores às variáveis `bool`. Os nós utilizam estas variáveis para saber se deverão continuar à espera de novos dados para transferir ou fechar as portas de comunicação que estão a usar para as transferências e terminarem. O nome atribuído a essas variáveis é `has_work` e `has_work_mic`, como é possível verificar na listagem 3.10.

Nas listagens 3.10 a 3.12 são usadas três cláusulas diferentes. A cláusula `in` transfere as variáveis, que estão dentro dos parêntesis, para o MIC. No início do `pragma`, os valores das variáveis são transferidos do `Host` para MIC, mas no fim do *pragma*, os valores não são transferidos do MIC para `Host`. A cláusula `out` tem o mesmo comportamento que a cláusula `in`, com a diferença que no início do *pragma*, os valores das variáveis não são

```
1  tbb::flow::graph * host_graph;
2  void * mic_graph;
3  bool has_work = true;
4  void * has_work_mic;
5
6  graph::graph() : host_graph(new tbb::flow::graph()){
7    #ifdef __INTEL_COMPILER
8
9    tbb::flow::graph *  temp = (tbb::flow::graph * ) malloc(sizeof(tbb::flow::
         graph *));
10   void * ptr;
11   bool temp_b;
12   void * ptr_b;
13
14   #pragma offload target(mic) nocopy(temp: length(1) alloc_if(1) free_if(0))
         nocopy(temp_b: length(1) alloc_if(1) free_if(0)) out( ptr , ptr_b)
15   {
16     temp = new tbb::flow::graph();
17     ptr = temp;
18     temp_b = new bool(true);
19     ptr_b = temp_b;
20   }
21   mic_graph = ptr;
22   has_work_mic = ptr_b;
23   #endif // __INTEL_COMPILER
24 }
25
26 graph::#graph(){
27   host_graph->~graph();
28   #ifdef __INTEL_COMPILER
29   #pragma offload target(mic) in(ptr)
30   {
31     delete( (tbb::flow::graph *) ptr);
32     ptr=NULL;
33   }
34
35   #endif // __INTEL_COMPILER
36 }
```

Listagem 3.10: Implementação parcial do grafo no MIC

transferidos do Host para MIC, mas no fim do pragma, os valores são transferidos do MIC para Host. A cláusula nocopy que o mesmo comportamento que in, mas não transfere os valores no inicio, nem no fim do pragma. A utilidade desta cláusula é que possibilita o acesso à mesma variável no MIC em pragma diferentes. A cláusula alloc_if(x) indica que a variável deverá ser alocada ou não, no inicio do pragma, dependendo do valor x. A cláusula free_if(x) indica que a variável deverá ser libertada ou não, no fim do pragma, dependendo do valor x.

Na listagem 3.10 é possível ver o uso de transferir uma variável para o MIC, atribuir a essa variável um grafo e dar indicação que esse grafo deverá permanecer no MIC. No fim do pragma o endereço é copiado para um apontador para void e transferido para o HOST através da cláusula out. Depois é possível utilizar esse apontador para aceder ao mesmo grafo no MIC através da cláusula in noutro pragma, como exemplificado na listagem 3.10.

```
1  #ifdef __INTEL_COMPILER
2  template<typename Input, typename Output>
3  class function_node<Input, Output, MIC> :
4  public sender<Output, MIC>,
5  public receiver<Input, MIC> {
6    typedef tbb::flow::sender<Input> predecessor_type;
7    typedef tbb::flow::receiver<Output> successor_type;
8
9    public:
10   typedef tbb::flow::function_node<Input, Output> _TBB_Base;
11   typedef Input _Input_type;
12   typedef Output _Output_type;
13
14   template<typename Body, typename... Args>
15   function_node(graph &g, std::size_t con, Body body, Args ... args) :
16   sender<Output, MIC>(), receiver<Input, MIC>() {
17     tbb::flow::function_node<Input, Output> *fn;
18     const void *addr_g = g.get_mic_graph();
19     void *ptr;
20
21     #pragma offload target(mic) in(addr_g) out(ptr) nocopy(fn: length(1)
           alloc_if(1) free_if(0))
22     {
23       fn = new tbb::flow::function_node<Input, Output>((*(tbb::flow::graph *)
             addr_g), con, Body(args...));
24       ptr = fn;
25     }
26     this->addr_on_mic = ptr;
27     this->addr_of_graph = &g;
28   }
29 }
```

Listagem 3.11: Implementação parcial do nó function_node para o MIC

3.3.2 Nós no MIC

No marrow::flow qualquer variante de nós pode ser construída de duas maneiras. Se for um nó para ficar no HOST é usado o construtor normal do TBB FG. Se for um nó para ficar no MIC, então através do mecanismo offload é criado um nó equivalente no MIC, que depois é associado ao grafo do lado do MIC, posteriormente é transferido o apontador para void para o HOST, como ilustrado na listagem 3.11 nas linhas 21 a 25.

Na linha 21 da listagem 3.11 é possível verificar utilização da cláusula in com apontadores para void para aceder a variáveis fora do scope. No bloco do pragma offload, da mesma linha, há um exemplo do comportamento por norma, de quando uma variável não definida nas cláusulas, é usada. Neste caso a variável é o args e o comportamento por norma é assumir que a variável esta dentro de uma cláusula inout, cujo procedimento é igual ao in, mas é feita a transferência dos valores no início e no fim do bloco.

Em qualquer chamada a uma operação do nó no lado do HOST é utilizado o mecanismo offload para fazer essa mesma chamada ao nó no lado do MIC, como ilustrado na listagem 3.12 . De modo que o nó no MIC ou no HOST oferece o mesmo conjunto de operações, abstraindo do facto de um ser apenas um apontador.

```
1 bool try_put(const Input &v) {
2   void * temp = this->addr_on_mic;
3   #pragma offload target(mic:0) in(temp)
4   {
5     (*(tbb::flow::function_node<Input> *) temp).try_put(v);
6   }
7   return true;
8 }
```

Listagem 3.12: Implementação de uma função do nó no MIC

3.3.3 Aresta SCIF

O funcionamento do `marrow::flow::make_edge()` varia entre 4 possíveis combinações, dependo da localização dos nós (localização do primeiro nó ⇒ localização do segundo nó):

HOST ⇒ HOST: a função invoca a sua congénere no TBB FG, criando uma ligação de fluxo de dados entre os dois nós.

MICx ⇒ MICx: é feito um `offload` com os apontadores para os endereços de memória no MICx, dos dois nós e dentro desse `offload` é feita a invocação da função equivalente no TBB FG no MICx, criando uma ligação de fluxo de dados entre os dois nós.

HOST ⇒ MICx: ocorrem os procedimentos abaixo descritos. Para simplificar a descrição, designaremos por A o nó a executar no Host e por B o nó a executar no MICx.

Uma porta da biblioteca SCIF é aberta no lado do HOST e através do mecanismo de `offload síncrono` é transferido:

- o apontador para grafo no MICx
- o endereço do nó B no MICx
- o identificador único da porta SCIF aberta
- o endereço no MIC para a variável has_work_mic do grafo.

Dentro do bloco de `offload` é aberta uma outra porta SCIF e é feito handshake entre a porta no HOST e a esta nova porta criada no MICx.

Em detalhe o que ocorre é que uma porta faz `scif_listen` e espera que a outra porta faça contacto. Entretanto a outra porta faz o `scif_connect` sabendo que no HOST há uma porta com um identificador único, passado como argumento. No outro lado a porta recebe o pedido de conexão e faz `scif_accept` e estabelece um caminho onde é possível trocar mensagens entre processos no HOST com os no MICx.

Posteriormente, são criados dois nós adicionais, um nó, Fonte SCIF, do tipo `source_node` localizado no MICx, criado dentro do `offload`. O outro nó, Fossa SCIF, é do tipo `function_node`, localizado no HOST. Terminada a criação dos dois nós é estabelecida uma

ligação entre o nó A e o nó Fossa SCIF e entre o nó Fonte SCIF e o nó B, através da função `make_edge()` do TBB FG. O nó Fonte SCIF recebe um functor com os seguintes parâmetros: a porta SCIF do lado do `MICx` para possibilitar a troca de mensagens com `HOST`; um apontador para um bool que é uma variável do grafo localizado no `MICx`, `has_work`, essa variável bool informa quando é que o nó deverá interromper a escuta por novos dados. O nó Fonte SCIF executa apenas uma operação durante o tempo de execução, que é enquanto a variável `has_work` for verdadeira, então o nó irá verificar se há novas mensagens na porta SCIF. Caso haja novas mensagens, o nó reconstrui uma variável do tipo `Input` do nó B, através dessas mensagens. Quando a variável, `has_work`, passa a falso, então o nó liberta a porta SCIF e passa a modo inactivo.

Por outro lado o nó Fossa SCIF recebe um functor com a porta SCIF do lado do `HOST`. Sempre que o nó Fossa SCIF receber um elemento do tipo `Input` do nó A, então o nó Fossa SCIF decompõe o elemento em mensagens que envia para o nó Fonte SCIF pela porta SCIF. Após o sucesso de envio das mensagens, o elemento é destruído, de modo que deixa de existir o elemento no `HOST`.

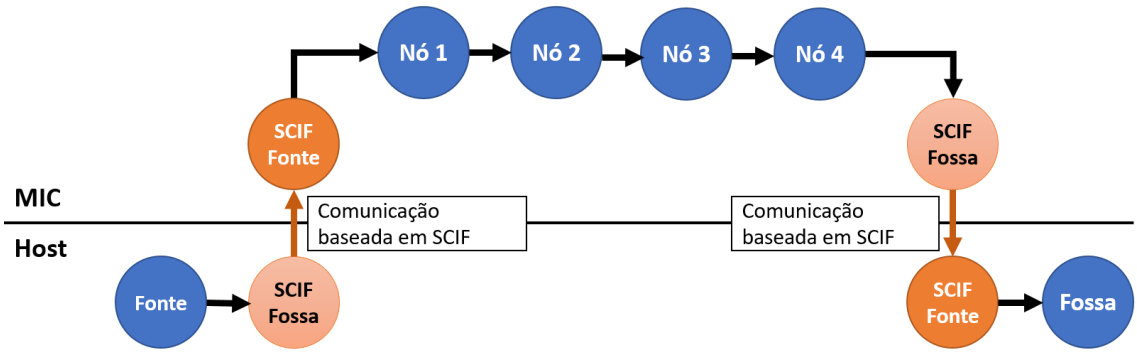

Figura 3.5: Nós intermédios criados com `make_edge()`

MICx ⇒ HOST: ocorre o mesmo que no caso anterior, com a distinção que o nó Fossa SCIF passa a estar no lado do `MICx` e o nó Fonte SCIF passa a estar no lado do `Host`.

Estruturas complexas A transferência de dados do tipo elementar como `int`, `bool`, `double` existem um functor comum que permite a transferência sem que o programador tenha de implementar novas funções. Mas para estruturas mais complexas como um vector é necessário que essas classes implementem duas funções, a função `send` e a função `receive`. A implementação da função `send` e `receive` do vector é observável na listagem 3.13 e 3.14.

```
template<typename T>
int send(scif_epd_t epd){
  unsigned int size = this->getSize();
  unsigned T * array =  this->getArray();
  bool confmensage = true;
  int block = 0; // 0 para nao bloqueante e 1 para bloqueante

  if(scif_send(epd, &confmensage, sizeof(bool), block) <0){
    return -1;// error ao enviar
  }
  block=1;
  if(scif_send(epd, &size, sizeof(unsigned int), block) <0){
  return -1;// error ao enviar
  }
  if(scif_send(epd, array, sizeof( T ) * size, block) <0){
    return -1;// error ao enviar
  }
  clear();
  return 0;
}
```

Listagem 3.13: Implementação da função send do vector

3.3.4 Vector

De maneira a optimizar a computação vectorial sobre dados que são transferidos entre `Host` e `MICx` é preciso assegurar que os dados sejam alocados e libertados quando necessário e que a alocação esteja alinhada da melhor forma para o sistema. Por isso, foi implementado `marrow::flow::vector` que garante o melhor alinhamento, caso os dados estejam no `Host` ou no `MICx`, como ilustrado na listagem 3.15;

3.3.5 Comentários Finais

Findo este capítulo temos um protótipo de um sistema de permite processamento de fluxo de dados em sistemas híbridos `CPU/Xeon Phi`, baseado no TBB GF. No capítulo seguinte iremos avaliar este mesmo protótipo com o objectivo de determinarmos em que cenários a sua utilização é vantajosa e quantificar tais ganhos.

```
template<typename T >
vector<T> receive_block(scif_epd_t epd){
  int block = 0;  // 0 para nao bloqueante e 1 para bloqueante
  unsigned int size;
  unsigned int * ToR = &size;
  bool confmensage = true;
  int no_bytes = 0 ;
  int curr_size = sizeof(unsigned int);
  int error;

  while( (no_bytes = scif_recv(epd, &confmensage, sizeof(bool), block)) > 0){
    confmensage = confmensage + no_bytes;
    curr_size = curr_size - no_bytes;
    if(curr_size == 0)
      break;
  }

  if(no_bytes==-1){
    return -1;
  }

  if(confmensage == 1) {
    block=1;
    if( (error = scif_recv(epd, &size, sizeof(unsigned int), block)) <0){
      return false;
    }

    T * array =  new T *[size];
    if( (error = scif_recv(epd, array[i], sizeof(T) * (size), block)) <0){
      return false;
    }
  }

  result = new marrow::streaming::vector<T>(size);
  result->copy(array);
  delete [] array;

  return result;

}
```

Listagem 3.14: Implementação da função receive do vector

```
void create_array(){
  #ifdef __MIC__
    array = (T **) _mm_malloc(this->size * sizeof(T *),64);
  #else
    array =  new T*[size];
  #endif
}
```

Listagem 3.15: Implementação da função criação de um vector

Capítulo 4

Avaliação

Este capítulo inicia com a descrição das questões de desempenho que se pretende responder sobre a proposta desenvolvida. Seguidamente, explicamos os casos de estudo usados para responder a essas questões. Por fim, são apresentados os resultados e respectiva interpretação.

4.1 Métricas de avaliação

Terminada a implementação da proposta colocam-se algumas questões:

1. Em que situações compensa recorrer ao Xeon Phi para executar um grafo em detrimento da execução no `HOST`?
2. Existem situações onde dividir os nós pelo `HOST` e o `Xeon Phi` tem melhor desempenho que executar o grafo todo apenas numa destas localizações?
3. Existe compensação na utilização híbrida de nós no `HOST` e no `MIC` no grafo?

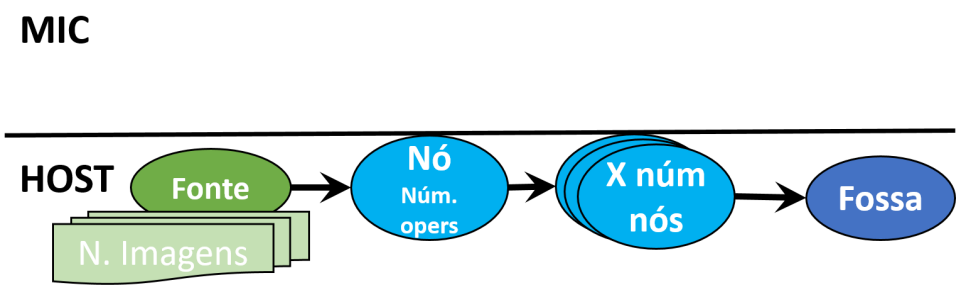

Figura 4.1: Esquema da implementação do grafo com nós só no *HOST*

Para responder a estas questões foram planeados 3 *benchmarks*.

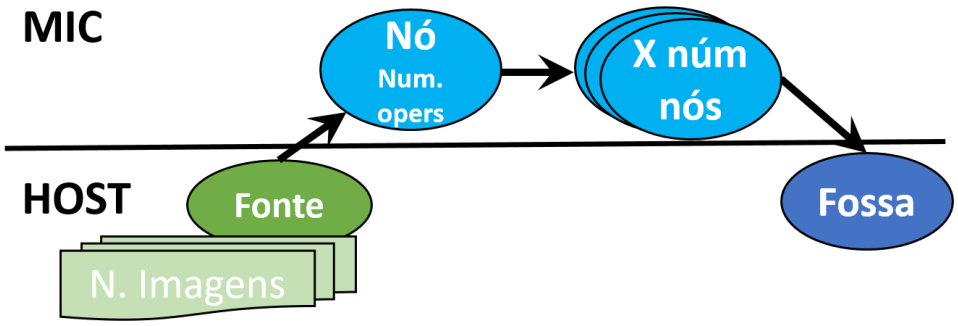

Figura 4.2: Esquema da implementação do grafo com nós no *HOST* e no *MIC*

Benchmark para primeira questão No primeiro *benchmark* temos duas variantes do mesmo *benchmark* com dois diferentes tipos de computação: sobre inteiros (`IncOp`) e sobre vírgula flutuante (`PowerOp`). Estas computações são feitas através de um grafo, esse grafo segue um dos seguintes modos *mCPU*, segundo o esquema em 4.1, ou *mHybrid*, segundo o esquema em 4.2. Em ambos esquemas o grafo, composto por uma sequência de nós, tem a seguinte decomposição: 1 `source_node`, Fonte, que corre sempre no `HOST` que efectua `ImgN` ciclos onde é feita a leitura de uma imagem no formato PGM do tipo P2 de dimensões `ImgD` no disco e é construído um vector de dimensões `ImgD` de inteiros, com os valores dos pixeis da imagem, que é enviada para os restantes nós; N nós intermédios do tipo `function_nodes`, que podem ser executados no `HOST` ou no `MIC`, dependendo do esquema que o grafo segue, que é definido pelo parâmetro de configuração do *benchmark* `EsqM`. A operação que os nós intermédios efectuam sobre os vectores que recebem como `Input`, depende do parâmetro `OpM`; 1 `function_node`, Fossa, que corre sempre no `HOST` e converte os vector de inteiros, que recebeu como `Input` dos nós intermédios, em imagens no formato PGM e escreve essas imagens no disco. Existe uma ligação do nó Fonte ao primeiro nó intermédio, do primeiro nó intermédio ao segundo, sucessivamente até N-1ésimo nó ao Nésimo nó intermédio e do Nésimo nó ao nó Fossa.

Cada imagem (vector de inteiros) passa uma vez em cada nó intermédio, onde o vector de inteiros será percorrido `WorkN` vezes, em que em cada percurso a operação `OpM` será aplicada a cada pixel da imagem (inteiro do vector).

Como mencionado na composição, a diferenciação entre os dois esquemas é a localização dos nós intermédios. O esquema da figura 4.1, `mCPU`, tem os seus nós intermédios no `HOST`, enquanto o esquema da figura 4.2, `mHybrid`, tem os seus nós no `MIC`. A operação OpM pode variar entre incrementação de um inteiro, como ilustrado na listagem 4.1, ou aplicação da potência de 2 a um inteiro, ilustrado na listagem 4.2.

Todos os parâmetros de configuração do `benchmark` estão presentes na tabela 4.1. Os valores do parâmetro ImgD são detalhados na tabela 4.2. Cada um destes parâmetros tem uma justificação. O N nós permite avaliar como é que a implementação do esquema `mHybrid` se comporta em relação ao esquema `mCPU` à medida que o tamanho do grafo aumenta.

```
for (int i = 0; i < WorkN; i++) {
  for (int j = 0; j < ImgD; j++) {
    pixel[j]++;
  }
}
```

Listagem 4.1: Pseudo código da operação IncOP.

```
for (int i = 0; i < WorkN; i++) {
  for (int j = 0; j < ImgD; j++) {
    pixel[j] =  std::pow((pixel[j]), 2);
  }
}
```

Listagem 4.2: Pseudo código da operação PowerOP.

Parâmetro de conf.	Significado	Valores possíveis
EsqM	Qual o esquema que o grafo segue	mCPU; mHybrid
OpM	Qual a operação que ira ser utilizada em cada pixel (inteiro)	IncOp ; PowerOp
ImgN	Número imagens que percorrem o grafo	4 ; 60 ; 120
ImgT	Dimensões da imagem.	720p ; 1080p ; 1440p
WorkN	Número de vezes que um nó intermédio percorre uma imagem	50 ; 100 ; 400
N	Número de nós intermédios do grafo	10 ; 40 ; 80
Tempos	Número de tempos cronometrados	20

Tabela 4.1: Resumo dos vários parâmetros de configuração dos *benchmarks*, com os seus significados e valores testados.

Nome da imagem	Pixels (inteiros)	Tamanho
720p	921600	3,3 MB
1080p	2073600	7,6 MB
1440p	3686400	13,2 MB

Tabela 4.2: Tamanho em MB e número pixels por cada tipo de imagem.

Benchmark para segunda questão No segundo *benchmark* em que se divide por percentagem a quantidade de núcleos no CPU e no Xeon Phi, isto é, tipicamente é 0 no CPU e 100 no Xeon Phi, quando é indicado por exemplo 30, então 30% dos nós estão no CPU e outros 70% estão no Xeon Phi. Saliente-se que mesmo no caso de 0 nós no CPU, há sempre dois nós que estão no CPU, que são o source e o sink. O nó source que faz a leitura dos ficheiros do disco rígido e passa para restantes nós e o *sink* que recebe as imagens pós as transformações e destrói-as. Este nó, também, conta o número de imagens que chegaram, até chegar a ultima para modificar a variável has_work do grafo, de maneira a indicar que terminou o processamento dos dados e que as comunicações entre CPU e Xeon Phi podem terminar.

Benchmark para terceira questão No terceiro *benchmark* foi corrido um grafo com a mesma proporção de nós no `HOST` e no `MIC`, segundo a figura 4.3. Ambos os nós no `HOST` e no `MIC` processam o mesmo número de imagens, sendo o objectivo colocar o processador e co-processador no máximo de utilização. Este *benchmark* foram utilizados os mesmos variáveis que os *benchmarks* anteriores, no entanto, apenas foi utilizado a operação de potência.

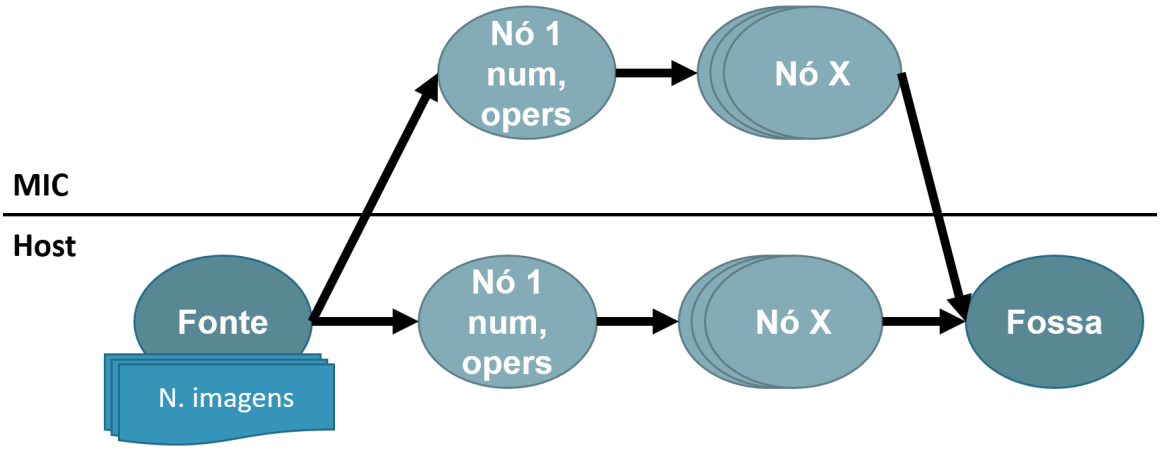

Figura 4.3: Esquema da implementação do grafo com a mesma proporção de nós no *HOST* e no *MIC*.

Cronometração dos *benchmarks* A contagem do tempo é feita depois da construção toda do grafo e enviado uma mensagem ao nó fonte que pode começar a gerar imagens. A contagem do tempo termina quando o grafo dá indicação, através da função `wait_for_all()` que não há trabalho para ser executado no grafo.

4.2 Sistema de experimentação

O servidor onde foram feito os *benchmarks* era composto pela seguintes características:

- CPU
  - Modelo - E5-2603
  - Cores - 4
  - Frequência - 1.8 GHz
  - HyperThreading - Não
  - Cache - 10 MB
  - Ram - 15GB DDR3 1066
  - Disco - Toshiba 7200 RPM
- Co-processador Xeon Phi
  - Modelo - 5110P
  - Cores - 60
  - Frequência - 1.05 GHz
  - Cache - 30 MB
  - RAM - 8 GB GDDR5 2.5GHz
- Sistema Operativo
  - Distribuição CentOS 6.7 Final
  - MPSS version - 3.6.1-1
  - Kernel - 2.6.32-573.26.1.el6.x86_64
  - Intel Compiler - 17.0.1

4.3 Resultados

Resultados para benchmark 1 Os resultados do primeiro benchmark foram divididos em duas tabelas, 4.4 e 4.6. Em ambas tabelas são calculados os valores de speedup através da formula $Speedup = \frac{tempo\_apenas\_no\_CPU}{tempo\_apenas\_no\_MIC}$. Na primeira tabela apenas estão os valores com a operação sobre inteiros, enquanto a segunda tabela estão os valores com a operação de potencia de 2. Os valores rodeados com a cor verde são valores considerados positivos, pois indicam um speedup maior que 1. Para cada tabela existe uma representação gráfica, figura 4.5 e a figura 4.7.

Speedup Incr			num. Operações		
Tipo de imagem	Num. Nós	Num. imagens	50	100	400
720p	10	4	0,19	0,27	0,58
		60	0,28	0,46	0,96
		120	0,38	0,44	1,01
	40	4	0,20	0,33	0,78
		60	0,59	0,93	2,71
		120	0,61	1,03	2,85
	80	4	0,20	0,35	0,81
		60	0,86	1,37	3,97
		120	1,02	1,77	4,84
1080p	10	4	0,22	0,29	0,60
		60	0,29	0,47	1,04
		120	0,34	0,44	1,39
	40	4	0,20	0,35	0,77
		60	0,57	0,97	3,11
		120	0,57	1,07	2,92
	80	4	0,21	0,35	0,82
		60	0,76	1,28	4,07
		120	1,04	1,71	4,83
1440p	10	4	0,20	0,28	0,58
		60	0,32	0,52	0,94
		120	0,32	0,46	1,07
	40	4	0,21	0,33	0,75
		60	0,59	0,95	2,68
		120	0,67	1,01	3,22
	80	4	0,20	0,35	0,81
		60	0,83	1,55	4,97
		120	0,97	1,76	5,16

Figura 4.4: *Benchmark* com a operação de incrementação, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem

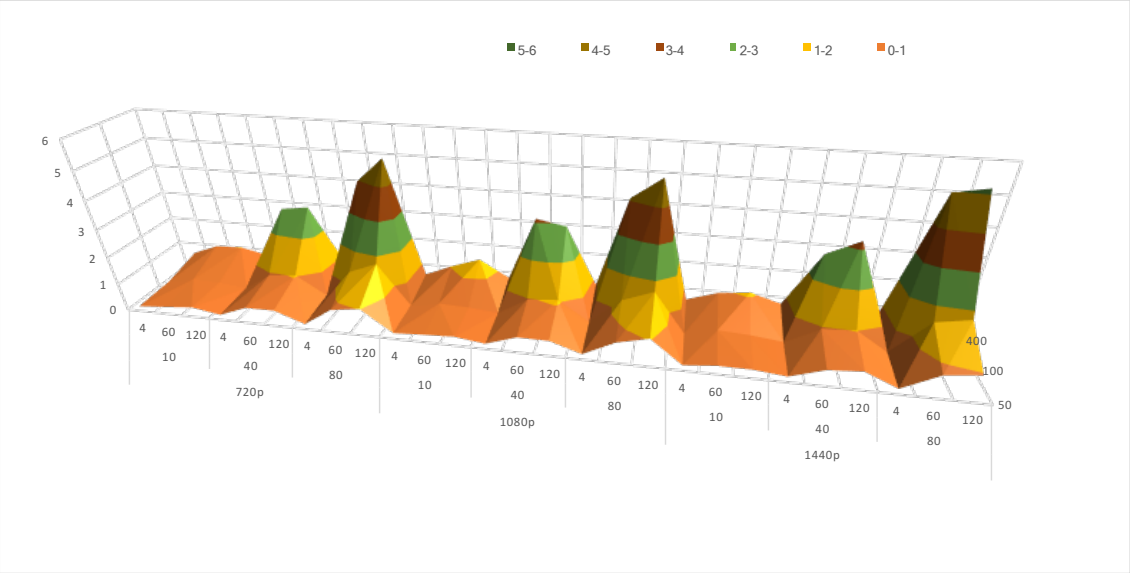

Figura 4.5: Gráfico do *Benchmark* com a operação incrementação, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem.

Speedup Pont			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	1,04	1,38	1,81
		60	1,75	2,96	7,13
		120	2,00	2,80	9,10
	40	4	1,62	1,80	1,94
		60	4,80	7,41	11,28
		120	5,91	8,81	13,00
	80	4	1,75	1,89	1,91
		60	7,33	9,08	12,11
		120	9,63	10,76	14,49
1080p	10	4	1,17	1,42	1,83
		60	1,96	3,04	6,99
		120	1,88	3,65	8,62
	40	4	1,68	1,85	1,93
		60	5,09	6,29	9,09
		120	5,15	8,94	12,80
	80	4	1,84	1,92	1,94
		60	6,66	8,21	9,49
		120	8,46	10,73	14,13
1440p	10	4	1,05	1,46	1,84
		60	1,63	3,07	5,81
		120	2,16	3,64	8,11
	40	4	1,66	1,78	1,95
		60	4,66	6,29	8,10
		120	5,77	8,06	12,61
	80	4	1,86	1,90	1,93
		60	6,47	7,57	8,49
		120	8,85	10,40	13,48

Figura 4.6: *Benchmark* com a operação potência, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem

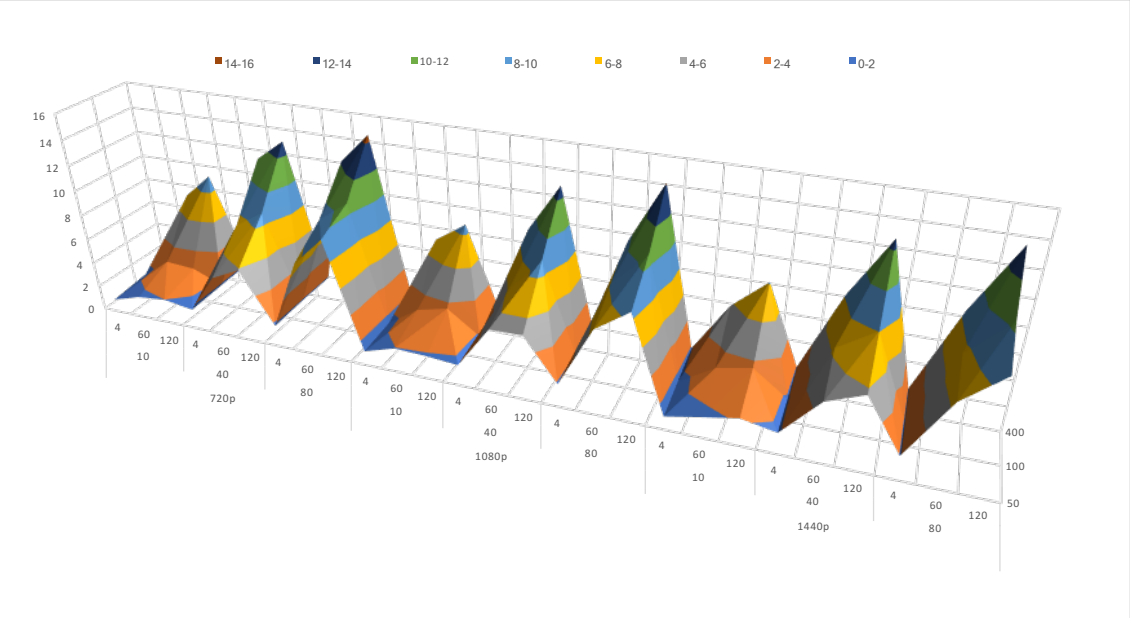

Figura 4.7: Gráfico do *Benchmark* com a operação potência de 2, variando com o número de operações, a de nós ou imagens e variando com o tipo da imagem

Resultados para *benchmark* 2 Segue alguns valores tirados para o segundo *benchmark*, onde se notou melhor desempenho em dividir o nós entre os dois ambientes.

Teste	Tipo Ima.	N. Ima.	N. Ops.	N. Nós	Função	Tempo (s)	T. com 0% (s)	Speedup
1	720p	60	100	80	Inc	31,300	22,773	137%
2	720p	120	100	40	Inc	33,678	32,543	103%
3	1440p	120	100	40	Inc	133,945	132,170	101%
4	1080p	60	400	10	Inc	35,201	33,981	104%
5	720p	4	50	10	Potência	2,588	2,481	104%
6	1080p	4	50	10	Potência	5,991	5,113	117%
7	1080p	120	50	40	Potência	459,525	89,212	515%
8	1440p	4	100	10	Potência	19,864	13,587	146%

Tabela 4.3: Resultados de alguns testes com 0% dos nós no CPU e 100% no Xeon Phi

Os valores de Melhor Speedup, com $MelhorSpeedup = \frac{tempo\_apenas\_no\_CPU}{menor\_tempo\_dentros\_das\_percentagem}$ são calculados entre o menor tempo entre as divisões e o tempo só com CPU, os valores de Extra Speedup, com $ExtraSpeedup = MelhorSpeedup - Speedup$, são a diferença entre Speedup da tabela 4.3 e Melhor Speedup

Teste	T. 10%	20%	30%	40%	50%	60%	70%	Melhor S.up	Extra S.up
1				21,82	20,82	22,54		150%	+13%
2				35,48	31,08	33,23		108%	+5%
3		139,63	101,99	111,81	120,49	135,83		131%	+30%
4			38,76	28,09	30,14	37,93		125%	+22%
5				2,55	2,45	2,34	2,50	110%	+6%
6			5,33	5,12	5,65	5,56		117%	0%
7	83,31	109,29	144,24	176,99	211,75	248,31		552%	+37%
8	14,46	14,68	14,13	15,60	13,73	15,06	14,92	145%	-2%

Tabela 4.4: Resultados dos testes da tabela4.3, mas com variação da percentagem de nós nos núcleos do CPU em vez Xeon Phi.

Resultados para *benchmark* 3 Os resultados do terceiro *benchmark* foram colocados numa única tabela, 4.8.

Speedup hibrido Pont			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	1,41	1,55	1,90
		60	4,08	4,94	10,92
		120	2,86	6,99	13,49
	40	4	1,74	1,79	1,97
		60	6,99	11,54	15,84
		120	10,71	15,93	22,30
	80	4	1,84	1,95	1,98
		60	11,86	14,36	17,23
		120	13,07	18,31	24,09
1080p	10	4	1,28	1,71	1,92
		60	4,49	6,47	10,53
		120	4,32	6,30	13,75
	40	4	1,81	1,92	1,98
		60	7,66	10,92	15,65
		120	8,46	14,26	18,84
	80	4	1,92	1,97	1,97
		60	11,52	13,97	16,73
		120	12,11	15,78	19,36
1440p	10	4	1,53	1,71	1,93
		60	4,79	7,54	11,76
		120	3,19	7,46	12,67
	40	4	1,88	1,94	1,98
		60	8,74	10,21	14,99
		120	8,51	12,71	15,87
	80	4	1,95	1,99	2,02
		60	11,35	13,08	15,57
		120	12,69	14,02	16,69

Figura 4.8: *Benchmark* com a operação potência, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem

4.4 Interpretação

Segundo os resultados da figura 4.4 e gráfico 4.5 é possível observar que não se pode enviar qualquer trabalho ao Xeon Phi e esperar que os ganhos sejam sempre melhores do que somente recorrendo ao CPU. A causalidade vem dos custos associados à transferência de dados entre CPU e Xeon Phi e à menor frequência dos núcleos.

Para chegar a resultados positivos, ou seja, *speedup* superior a 1, é necessário aproveitar os múltiplos núcleos do Xeon Phi. Desse modo com o aumento do número de imagens a serem processadas, os resultados tornam-se mais positivos. No entanto, com o aumento do número de imagens, os custos de transferência sobem de igual maneira, por isso não

se observam grandes ganhos. Razão pela qual, quando há um acréscimo do número operações ou de nós, é como se aumentássemos o número de imagens sem ampliar os custos de transferência, o que leva a melhores resultados.

Também é possível observar que os ganhos com o aumento do número de operações é maior que com o aumento do número nós. Isto estará relacionado que facto do aumento de operações incentivar o uso da *cache* de cada núcleo, o que não acontece com aumento dos nós. A variação do tamanho da imagem não revela diferença nos resultados.

Segundo os resultados da figura 4.6 e gráfico 4.7 os ganhos com a operação da potência de 2 são muito maiores que com a operação incrementação. Poder-se-á avançar que unidade de computação de vírgula flutuante é melhor do que a unidade de computação de inteiros no Xeon Phi.

Os ganhos são muito positivos, principalmente, no *benchmark* com a operação da potência, mas, é importante notar que o CPU de comparação é bastante antigo sendo uma má representação dos processadores modernos.

Nos resultados do *benchmark* da tabela 4.4, onde houve divisão da percentagem dos nós entre CPU e o Xeon Phi, é possível observar que nem sempre há benefícios em dividir o trabalho entre os dois sistemas comparativamente com e sem divisão, ou seja, percentagem 0.

Também é observável que quando há benefícios em distribuir os nós entre os dois sistemas a percentagem não é universal. Mas é observável que quando há paridade entre os tempos de execução do grafo apenas no CPU com os tempos no Xeon Phi, então a divisão com a percentagem de 50 tipicamente é melhor. Quando os tempos são melhores no Xeon Phi, então a percentagem tem de ser menor para encontrar valores, como é observável no teste 7.

Concluindo, os resultados são positivos, mas não suficientemente extensos para criar uma ligação dos benefícios perante outros trabalhos que já existem para computação de processamento de fluxo de dados.

Capítulo 5

Conclusão

Esta dissertação enfrentou o desafio de testar uma hipótese teórica, ainda não questionada ou testada. O trabalho realizado as soluções e caminhos encontrados ao longo do percurso de teste positivo/falência permitiram demonstrar que um desenho teórico poderia ser implementado com sucesso.

Nesta premissa residia um dos desafios. Foram as muitas as tentativas que deram em erro, nomeadamente a limitações no acesso a ferramentas que teriam auxiliado na clarificação de erros de código e de implementação. São exemplos: 1 - Acesso remoto ao servidor que limitou o acesso a algumas interfaces gráficas para otimização e apuramento de erros; 2 - Actualizações de software (instabilidade dos mecanismos de `offload`); 3 - O componentes comparativos para avaliação da implementação; 4 - A documentação de suporte difícil acesso e superficial; 5 - Acesso limitado às capacidades do Xeon Phi.

No entanto, apesar as adversidades e de muitas tentativas e erro foi possível concluir e provar a proposta inicial, ou seja, um proposta de capaz de utilizar as capacidades computacionais do CPU e Xeon Phi, de uma forma similar a já apresentada com TBB.

Assim foi possível concluir dos ganhos de desempenho em elevada capacidade de processamento paralelo.

Fica também como referência que com a utilização do mecanismo `offload` para alocar dados no Xeon Phi, quando vários processos recorrem ao mecanismo, o sistema poderá ficar instável.

No entanto, eventualmente esta instabilidade ficou a dever-se à desatualização do *software*.

E foi possível ver pelos resultados que há benefícios na utilização conjunta das duas arquiteturas, mas que tem de ser explorada para a determinação da melhor divisão do trabalho.

5.1 Trabalho Futuro

Fica em aberto para possíveis trabalhos futuros, as soluções/sistemas que não foi possível testar neste trabalho:

- A otimização das transferências entre CPU e Xeon Phi através de mapas de endereçamento.
- Os sistemas de réplicas de dados transferidos entre CPU e Xeon Phi, em foco os `vectors`.
- A melhoria do sistema de terminação dos nós de comunicação, sendo uma possível solução ser que o grafo híbrido guardasse em linked list todos os nós de comunicação e terminasse através de funções em vez de variáveis de controlo.
- A implementação de exemplos de *benchmarks* mais convencionais para determinação de desempenho, como N-Body.
- A implementação das restantes variantes de nós disponíveis pelo TBB.
- O desenvolvimento de um mecanismo para TBB distribuir trabalho pelos núcleos com objetivo de aumentar o desempenho.

5.2 Desafio Futuro

Adaptar esta solução para maior escalabilidade, ou seja, adaptar a comunicação da proposta para funcionamento com *cluster* de CPUs e *clusters* de Xeon Phis.

Bibliografia

[1] F. Alexandre, R. Marques e H. Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". Em: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. ACM, 2014, pp. 880–885. DOI: 10.1145/2554850.2555018.

[2] H. Andrade, B. Gedik e D. Turaga. *Fundamentals of Stream Processing*. 1ª ed. New York: Cambridge University Press, 2014, p. 548. ISBN: 9781139058940. DOI: 10.1017/CBO9781139058940. URL: http://ebooks.cambridge.org/ref/id/CBO9781139058940.

[3] Apache. *Apache Storm Documentation*. URL: http://storm.apache.org/documentation.html (acedido em 10/02/2016).

[4] Apache. *Spark Streaming - Spark 1.6.0 Documentation*. URL: http://spark.apache.org/docs/latest/streaming-programming-guide.html (acedido em 10/02/2016).

[5] J. C. Beard, P. Li e R. D. Chamberlain. "RaftLib : A C ++ Template Library for High Performance Stream Parallel Processing". Em: *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* February (2015), pp. 96–105. DOI: 10.1145/2712386.2712400.

[6] I Buck, T Foley, D Horn, J. Sugerman, K. Fatahalian, M. Houston e P. Hanrahan. "Brook for GPUs: stream computing on graphics hardware". Em: *Transactions on Graphics (TOG)* 23.3 (2004), pp. 777–786. ISSN: 0730-0301. DOI: 10.1145/1015706.1015800.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi e K. Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". Em: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38. URL: http://sites.computer.org/debull/A15dec/p28.pdf.

[8] Colfax International. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Ed. por Colfax International. Colfax International, 2013. ISBN: 9780988523418.

[9] *COPROCESSADOR PHI AND JAVA*. https://software.intel.com/en-us/forums/intel-moderncode-for-parallel-architectures/topic/543730. Ultimo acesso: 03/16/2017.

[10] L. Dagum e R. Menon. "OpenMP: an industry standard API for shared-memory programming". Em: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[11] *Debugging COI Applications for Intel® Xeon Phi™ Coprocessors | Intel® Software*. https://software.intel.com/en-us/blogs/2015/01/28/debugging-coi-applications-for-intel-xeon-phi-coprocessors. (Accessed on 03/20/2017).

[12] J Dokulil, E Bajrovic, S Benkner, M Sandrieser e B Bachmayer. "HyPHI - Task Based Hybrid Execution C++ Library for the Intel Xeon Phi Coprocessor". Em: *Parallel Processing (ICPP), 2013 42nd International Conference on* (2013), pp. 280–289. ISSN: 0190-3918. DOI: 10.1109/ICPP.2013.37.

[13] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che e C. Xu. "An Empirical Study of Intel Xeon Phi". Em: *arXiv preprint* Section III (2013), pp. 137–148. DOI: 10.1145/2568088.2576799. arXiv: 1310.5842.

[14] Google. *What is Google Cloud Dataflow?* URL: https://cloud.google.com/dataflow/what-is-google-cloud-dataflow#Use-Cases (acedido em 10/02/2016).

[15] *Hadoop-Streaming*. (Accessed on 03/16/2017). URL: https://hadoop.apache.org/docs/r1.2.1/streaming.html.

[16] A. H. Hormati, M. Samadi, M. Woh, T. Mudge e S. Mahlke. "Sponge: Portable stream programming on graphics engines". Em: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11* (2011), p. 381. ISSN: 03621340. DOI: 10.1145/1950365.1950409.

[17] H. P. Huynh, A. Hagiescu, W. Wong e R. S. M. Goh. "Scalable framework for mapping streaming applications onto multi-GPU systems". Em: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. ACM, 2012, pp. 1–10. DOI: 10.1145/2145816.2145818.

[18] F. Ino, S. Nakagawa e K. Hagihara. "GPU-Chariot : A Programming Framework for Stream Applications Running on Multi-GPU Systems". Em: 12 (2013), pp. 2604–2616.

[19] M. Integrated, C. Scif e U. Guide. "Intel ® Many Integrated Core Symmetric Communications Interface (SCIF) User Guide". Em: (2012).

[20] Intel. "Intel ® Xeon Phi ™ Coprocessor System Software Developers Guide". Em: (2014), p. 163.

[21] Intel. "Memory Management for Optimal Performance on Intel ® Xeon Phi ™ Coprocessor : Alignment and Prefetching". Em: (2015).

[22] *Intel Quick Sync Video*. http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html. Ultimo acesso: 03/16/2017.

[23] *Intel® Xeon Phi™ Product Family: Product Brief*. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html. Ultimo acesso: 03/16/2017.

[24] *Intel® Xeon® Processor E7-8890 v4 (60M Cache, 2.20 GHz) Product Specifications*. https://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz. Ultimo acesso: 03/16/2017.

[25] J. Jeffers e J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. 2013, p. 432. ISBN: 978-0124104143.

[26] M. Klemm e J. Enkovaara. "pyMIC: A Python offload module for the Intel Xeon Phi coprocessor". Em: *Proceedings of PyHPC* (2014). http://www.dlr.de/sc/Portaldata/15/Resources/dokumente/pyhpc2014/submissions/pyhpc2014_submission_8.pdf.

[27] M. Kleppmann. *Making Sense of Stream Processing*. Ed. por S. Cutt. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc. 183 pp. ISBN: 978-1-491-94010-5.

[28] C. E. Leiserson. "The Cilk++ Concurrency Platform". Em: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. San Francisco, California: ACM, 2009, pp. 522–527. ISBN: 978-1-60558-497-3. DOI: 10.1145/1629911.1630048. URL: http://doi.acm.org/10.1145/1629911.1630048.

[29] J. V. Lima, F. Broquedis, T. Gautier e B. Raffin. "Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor". Em: *Sbac-Pad'13* (2013), pp. 105–112. ISSN: 15506533. DOI: 10.1109/SBAC-PAD.2013.28. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6702586.

[30] R. Marques, H. Paulino, F. Alexandre e P. D. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". Em: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Vol. 8097. Lecture Notes in Computer Science. Springer, 2013, pp. 874–885. DOI: 10.1007/978-3-642-40047-6_86.

[31] J. Meng, V. a. Morozov, V. Vishwanath e K. Kumaran. "Dataflow-driven GPU performance projection for multi-kernel transformations". Em: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–11. ISSN: 21674329. DOI: 10.1109/SC.2012.42. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6468531.

[32] *MPI: A Message-Passing Interface Standard*. Rel. téc. Knoxville, TN, USA, 1994.

[33] *Native and Offload Programming Models | Intel® Software*. https://software.intel.com/en-us/articles/native-and-offload-programming-models. Ultimo acesso: 03/16/2017.

[34] *Power By Apache Storm*. http://storm.apache.org/releases/current/Powered-By.html. Ultimo acesso: 03/16/2017.

[35] R. Rahman. *Intel Xeon Phi coprocessor Architecture and Tools*. Vol. XXXIII. 2. Apress, 2013. ISBN: 9780874216561. arXiv: arXiv:1011.1669v3.

[36] J. Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[37] S. Sato e H. Iwasaki. "A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming". Em: *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. Vol. 5904. Lecture Notes in Computer Science. Springer, 2009, pp. 79–94. DOI: 10.1007/978-3-642-10672-9_8.

[38] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan e P. Hanrahan. "Larrabee: A Many-core x86 Architecture for Visual Computing". Em: *ACM Trans. Graph.* 27.3 (ago. de 2008), 18:1–18:15. ISSN: 0730-0301. DOI: 10.1145/1360612.1360617. URL: http://doi.acm.org/10.1145/1360612.1360617.

[39] A. G. Shoro e T. R. Soomro. "Big Data Analysis : Apache Spark Perspective". Em: 15.1 (2015).

[40] F. Soldado, F. Alexandre e H. Paulino. "Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments". Em: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787. DOI: 10.1002/cpe.3612.

[41] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker e I. Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

ANEXO I

Resultados do benchmark com tempos

Time CPU Inc (s)			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	0,43	0,59	1,48
		60	4,40	6,09	15,31
		120	8,75	12,17	30,42
	40	4	0,98	1,64	5,22
		60	10,12	16,86	53,30
		120	20,08	33,68	106,77
	80	4	1,73	3,05	10,19
		60	17,71	31,30	104,14
		120	35,25	62,64	207,87
1080p	10	4	0,98	1,36	3,42
		60	10,05	14,08	35,20
		120	20,03	27,87	70,15
	40	4	2,26	3,80	12,09
		60	23,08	38,95	123,27
		120	46,42	78,01	246,62
	80	4	4,00	7,06	23,66
		60	40,87	72,59	241,82
		120	81,43	144,36	482,09
1440p	10	4	1,68	2,34	5,92
		60	17,30	23,98	60,37
		120	34,45	47,91	120,82
	40	4	3,91	6,54	20,87
		60	40,02	67,24	212,75
		120	79,80	133,95	424,69
	80	4	6,92	12,19	40,88
		60	70,40	124,65	415,31
		120	140,08	248,24	831,06

Figura I.1: *Benchmark* com a operação de incrementação, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem, com os nós no Host e com os valores a representar o tempo de execução em segundos

Time CPU pont (s)			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	2,59	4,93	19,07
		60	26,73	50,70	194,85
		120	53,32	101,47	389,84
	40	4	9,65	19,07	75,75
		60	99,04	195,05	771,79
		120	197,78	389,66	1542,16
	80	4	19,06	38,00	151,20
		60	195,19	387,91	1539,47
		120	390,44	773,80	3077,72
1080p	10	4	5,99	11,48	44,38
		60	61,64	117,97	453,85
		120	124,21	235,37	906,64
	40	4	22,47	44,33	176,02
		60	230,00	453,90	1800,65
		120	459,53	906,19	3590,95
	80	4	44,46	88,31	351,96
		60	454,02	901,97	3582,83
		120	908,52	1802,29	7170,25
1440p	10	4	10,39	19,86	76,78
		60	106,48	202,93	782,86
		120	212,77	406,32	1563,30
	40	4	38,90	76,69	304,60
		60	395,98	781,53	3095,93
		120	790,74	1561,73	6187,77
	80	4	76,84	153,04	609,04
		60	782,34	1554,92	6175,79
		120	1562,14	3104,33	12352,20

Figura I.2: *Benchmark* com a operação de potencia de 2, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem, com os nós no Host e com os valores a representar o tempo de execução em segundos

Time MIC Inc (s)			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	2,23	2,20	2,54
		60	15,75	13,30	15,90
		120	22,98	27,69	30,03
	40	4	4,80	5,00	6,70
		60	17,10	18,17	19,63
		120	33,17	32,54	37,40
	80	4	8,61	8,75	12,53
		60	20,57	22,77	26,22
		120	34,66	35,29	42,93
1080p	10	4	4,35	4,62	5,71
		60	34,61	29,74	33,98
		120	58,61	63,49	50,33
	40	4	11,24	10,93	15,61
		60	40,25	40,22	39,65
		120	81,61	72,90	84,48
	80	4	19,38	20,39	28,79
		60	53,46	56,61	59,41
		120	78,32	84,54	99,83
1440p	10	4	8,39	8,45	10,17
		60	54,18	46,22	64,39
		120	108,38	103,95	113,01
	40	4	18,78	19,63	27,88
		60	67,78	70,53	79,34
		120	118,61	132,17	131,80
	80	4	34,30	35,31	50,57
		60	84,98	80,20	83,61
		120	145,16	141,07	161,12

Figura I.3: *Benchmark* com a operação de incrementação, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem, com os nós no MIC e com os valores a representar o tempo de execução em segundos

Time Mic pont (s)			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	4	60	120
720p	10	50	2,48	15,30	26,69
		100	3,56	17,14	36,29
		400	10,54	27,34	42,82
	40	50	5,95	20,63	33,46
		100	10,61	26,32	44,25
		400	39,05	68,42	118,59
	80	50	10,87	26,64	40,56
		100	20,13	42,71	71,90
		400	79,22	127,08	212,39
1080p	10	50	5,11	31,39	66,12
		100	8,07	38,86	64,48
		400	24,31	64,94	105,13
	40	50	13,35	45,23	89,21
		100	23,91	72,11	101,38
		400	91,19	198,11	280,62
	80	50	24,18	68,14	107,38
		100	46,11	109,88	167,92
		400	181,73	377,54	507,43
1440p	10	50	9,86	65,49	98,40
		100	13,59	66,00	111,56
		400	41,64	134,71	192,68
	40	50	23,37	84,95	137,08
		100	43,13	124,27	193,87
		400	156,13	382,10	490,69
	80	50	41,21	120,95	176,59
		100	80,56	205,39	298,55
		400	315,12	727,26	916,66

Figura I.4: *Benchmark* com a operação de potencia de 2, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem, com os nós no MIC e com os valores a representar o tempo de execução em segundos

Time Hibrido Pont (s)			num. Operações		
Tipo de Imagens	num. Nós	num. Imagens	50	100	400
720p	10	4	1,84	3,17	10,06
		60	6,54	10,26	17,85
		120	18,66	14,51	28,90
	40	4	5,53	10,63	38,48
		60	14,17	16,90	48,72
		120	18,47	24,45	69,15
	80	4	10,36	19,46	76,23
		60	16,46	27,02	89,32
		120	29,88	42,27	127,75
1080p	10	4	4,66	6,72	23,08
		60	13,72	18,23	43,12
		120	28,75	37,38	65,95
	40	4	12,41	23,08	89,04
		60	30,04	41,56	115,03
		120	54,29	63,56	190,64
	80	4	23,16	44,88	178,91
		60	39,40	64,59	214,11
		120	75,05	114,20	370,36
1440p	10	4	6,80	11,61	39,74
		60	22,25	26,92	66,58
		120	66,78	54,45	123,41
	40	4	20,72	39,63	153,79
		60	45,30	76,51	206,47
		120	92,88	122,89	389,91
	80	4	39,46	76,85	301,76
		60	68,95	118,88	396,65
		120	123,12	221,42	740,12

Figura I.5: *Benchmark* com a operação de potencia de 2, variando com o número de operações, de nós ou imagens e variando com o tipo da imagem, com os nós no Host e MIC e com os valores a representar o tempo de execução em segundos