# *D-STHARk*: Avaliando Escalonadores Dinâmicos de Tarefas em Arquiteturas Híbridas Simuladas*

**Sávyo Toledo , Danilo Melo , Guilherme Andrade , Fernando Mourão , Leonardo Rocha**

DCOMP/UFSJ - São João del-Rei, MG , Brasil

{savyomachado,daniloamaral,gandrade,fhmourao,lcrocha}@ufsj.edu.br

***Abstract.*** *Aiming to exploit better new computing architectures, with several Processing Units (PUs), such as CPUs, GPUs and Intel Xeon Phi, recent works focus on proposing novel runtime environments that offer a variety of methods for scheduling tasks dynamically on different PUs. A main limitation of such proposals refers to the constrained system configurations to evaluate them. In this work we present D-STHARk (***D****ynamic* ***S****cheduling of* ***T****asks in* ***H****ybrid Simulated* ***AR****chitectures), a tool that provides a complete simulated execution environment that allows evaluating dynamic scheduling strategies on simulated applications and hybrid architectures. Adopting D-STHARk, we simulated some experiments presented in recent works, achieving the same conclusions.*

***Resumo.*** *Para melhor explorar novas arquiteturas computacionais, composta por diversas Unidades de Processamento (UPs), tais como CPUs, GPUs e Intel Xeon Phi, estudos recentes focam em propor ambientes de execução que oferecem métodos de escalonamento dinâmico para diferentes UPs. A principal limitação dessas propostas são as restrições de configuração de sistema para avaliá-las. Nesse trabalho apresentamos o D-STHARk, (***D****ynamic* ***S****cheduling of* ***T****asks in* ***H****ybrid Simulated* ***AR****chitectures), uma ferramenta que provê um ambiente completo de simulação de arquiteturas híbridas para avaliação de estratégias de escalonamento dinâmico. Simulamos com o D-STHARk experimentos apresentados em trabalhos recentes alcançando as mesmas conclusões.*

## 1. Introdução

O crescimento contínuo de dados, associados a processamentos mais sofisticados em diferentes áreas de conhecimento, tem impulsionado avanços significantes nas arquiteturas de computação, refletindo em sistemas de armazenamentos mais eficientes, como também o uso de diferentes tipos de unidades de processamento (UPs), denominadas de arquiteturas híbridas. Um exemplo dessas novas arquiteturas são computadores com múltiplos processadores (arquiteturas multicore) e diferentes coprocessadores, tais como unidades de processamento gráfico (Graphic Processing Unit - GPUs) [Fatica and Luebke 2007] e o Intel Xeon Phi (MIC) [Jeffers and Reinders 2013]. Dessa forma, tem se tornado essencial que aplicações oriundas de diferentes domínios sejam capazes de explorar, de forma coordenada e eficiente, todas as UPs disponíveis, aproveitando ao máximo as suas capacidades de processamento.

*Esse trabalho foi parcialmente financiado por CNPq, CAPES, FINEP, Fapemig, e INWEB.

Observamos na literatura várias propostas de ambientes de execução com o objetivo de tornar mais transparente o uso de diferentes UPs para os desenvolvedores [Augonnet et al. 2010]. Entre os principais métodos fornecidos por estes ambientes, destacam-se os escalonadores de tarefas, responsáveis pela distribuição adequada das várias tarefas que compõem uma aplicação entre as UPs. Escalonadores podem ser: (1) estáticos, em que as características das tarefas e as capacidades das UPS são avaliadas em um estágio de pré-processamento, usando informação globais da aplicação; e (2) dinâmicos, em que a avaliação é feita em tempo de execução, considerando uma visão limitada da execução da aplicação, sendo portanto, um cenário mais desafiador.

Embora existam diferentes propostas de escalonadores dinâmicos na literatura, a maioria deles é avaliada em configurações de sistemas reais restritas, compostos por um número reduzido de UPs (i.e, alguns núcleos de CPU, uma ou duas GPUs e / ou um ou dois MICs) devido a elevados custos associados à criação de ambientes mais completos. Por exemplo, o preço de um coprocessador Intel Xeon Phi 7120P Henhexaconta-Core é aproximadamente R$20.000,00. Como consequência, as conclusões obtidas pela avaliação da escalabilidade destas propostas podem ser limitadas. Neste contexto, um ambiente simulado em que seja possível configurar diferentes arquiteturas híbridas, composto por um número ilimitado de UPs, a fim de avaliar estratégias de escalonamento dinâmico, torna-se uma contribuição importante, sendo esse o foco do trabalho.

Mais especificamente, neste artigo, apresentamos *D-STHARk* (**D**ynamic **S**cheduling of **T**asks in **H**ybrid Simulated **AR**chitectures), cujo objetivo é fornecer um ambiente de execução simulado completo que permite avaliar estratégias de escalonamento dinâmico de tarefas em aplicações simuladas. Além disso, o *D-STHARk* permite ao usuário simular arquiteturas híbridas, variando os tipos e número de UPs (CPUs, GPUs e MICs), bem como simular diferentes aplicações, compostas por diversas tarefas de características distintas (e.g., dependências entre tarefas, volume de dados a ser manipulado, etc.). Além de fornecer diferentes estratégias de escalonamento dinâmico, [Andrade et al. 2014b, Andrade et al. 2014a], o *D-STHARk* permite que novas estratégias sejam adicionadas por meio de uma API. Ressaltamos ainda que não encontramos na literatura propostas de simuladores com objetivos similares aos nossos.

No intuito de avaliar nossa ferramenta, simulamos a execução de uma aplicação de análise de imagens utilizada em estudos da morfologia de câncer no cérebro [Andrade et al. 2014a], considerando algumas estratégias de escalonamento dinâmico propostas em [Andrade et al. 2014b] e avaliando diferentes configurações de arquitetura. Em um primeiro momento, demonstramos que o *D-STHARk* foi capaz de apresentar os mesmos resultados encontrados nos trabalhos originais, demonstrando assim sua efetividade. Apresentamos também uma análise complementar das estratégias de escalonamento usando configurações de arquitetura não avaliadas no trabalho original, evidenciando a utilidade do *D-STHARk* em fornecer análises mais amplas.

*A implementação da ferramenta e execuções de experimentos foram realizadas pelo aluno Sávyo Toledo, sob a orientação do professor Leonardo Rocha. As análises dos resultados foram feitas com a colaboração do professor Fernando Mourão. Esse trabalho também contou com a colaboração do aluno Danilo Melo na inserção das estratégias de escalonamento, bem como do aluno de pós-graduação Guilherme Andrade na compreensão e adaptação da aplicação avaliada.*

## 2. Trabalhos Relacionados

O uso eficiente de sistemas híbridos equipados com CPUs e aceleradores é um problema desafiador que exige a implementação de aplicações otimizadas para múltiplos processadores e escalonamento de tarefas entre dispositivos heterogêneos. Recentemente, técnicas de compilação [Ravi et al. 2014], bibliotecas de domínio específico [Bradski 2000], e, sobretudo, ambientes de execução [Rossbach et al. 2011, Augonnet et al. 2010, Bueno et al. 2012, Ravi et al. 2014] têm sido propostos para reduzir o esforço de programação relacionados à portabilidade de aplicações para esses sistemas.

Plataformas de execução distribuída entre CPU, GPU e MIC têm sido alvo de vários projetos recentes [Augonnet et al. 2010, Bueno et al. 2012, Ravi et al. 2014, Lima et al. 2013]. Em [Ravi et al. 2014], é proposto um tradutor de operações genéricas de redução, baseado em técnicas de compilação, para sistemas híbridos CPU-GPU. Em [Bueno et al. 2012], os autores propõem o OmpSs, um modelo de programação paralela para aplicações de fluxo-dados que permite paralelização de códigos por meio do compilador a partir de anotações feitas no código pelo programador. Em [Augonnet et al. 2010], é apresentado o StarPU, um ambiente em que as tarefas de uma aplicação são modeladas por meio de Grafo Direcionado Acíclico. XKaapi [Lima et al. 2013] é outro ambiente de execução que possibilita a execução cooperativa em CPU-GPU-MIC, na qual as operações possuem diversas implementações focadas em diferentes dispositivos de computação.

Esforços recentes em ambientes de execução têm dado atenção especial à exploração dos chamados escalonadores dinâmicos que distribuem, em tempo de execução, as tarefas de uma dada aplicação entre as diferentes UPs disponíveis. Basicamente, estes escalonadores realizam uma avaliação em tempo de execução das características de cada tarefa e UP. Em seguida, com base nessa avaliação, determinam a UP mais adequada para executar cada tarefa. O desafio, nesse caso, é a forma de maximizar as oportunidades de paralelização com base apenas em um conhecimento local e limitado do conjunto de tarefas que compõem a aplicação. Escalonadores devem evitar eventos que comprometam a distribuição adequada de tarefas, tais como a sobrecarga de uma UP, escolha de um UP que não seja a mais adequada para executar uma determinada tarefa, e até mesmo transferências excessivas de dados entre memórias não compartilhadas de UPs distintas. Existem várias propostas de escalonadores dinâmicos na literatura [Andrade et al. 2014b]. A principal limitação dessas propostas refere-se às restrições quanto a configurações de sistemas híbridos reais para avaliação e ajustes dessas estratégias. Em razão do custo de se criar ambientes mais completos e diversificados, grande parte destas avaliações é feita considerando-se apenas uma arquitetura em particular.

## 3. D-STHARk

Nesta seção, apresentamos *D-STHARk* dividindo a descrição em três partes. Primeiro, apresentamos seus principais componentes. Em seguida, descrevemos como esses componentes funcionam e interagem uns com outros para executar uma simulação. Por fim, apresentamos a API que permite aos usuários definir e inserir distintas estratégias de escalonamento. O *D-STHARk* está disponível em *https://github.com/SavyoToledo/D-STHARk*.

### 3.1. Componentes do *D-STHARk*

Basicamente, *D-STHARk* é composto de quatro componentes principais: (1) configuração ambiente; (2) criação de tarefas; (3) definição da estratégia de escalonamento; (4)

execução experimental e avaliação. Os usuários podem ajustar a simulação de acordo com seus objetivos, avaliando cada cenário em particular mais próximo de condições reais.

**1. Configurações do Ambiente.** *D-STHARk* permite aos usuários criarem arquiteturas híbridas usando três tipos de UPs: CPU, GPU e MIC. Cada uma destas pode ser instanciada diversas vezes. Por exemplo, podemos criar um ambiente com 6 CPUs, 2 GPUs e 2 MICs. Além disso, para cada coprocessador (i.e., GPU e MIC) é possível definir a largura da banda de transferência de dados, que corresponde à quantidade de dados que pode ser transferida entre o coprocessador e a CPU a cada unidade de tempo. Esta configuração torna as simulações ainda mais próximas de cenários reais. Além disso, os usuários podem salvar essas configurações para usar em outras simulações.

**2. Criação de tarefas.** *D-STHARk* simula cada aplicação como um conjunto de tarefas distintas e suas dependências. Por sua vez, cada tarefa é definida por três características: (1) tipo de tarefa, (2) taxa de erro (3) e tamanho da carga de trabalho. O speedup de uma UP pode variar bastante de acordo com as operações consideradas. Além disso, o desempenho relativo entre as UPs varia de acordo com a operação executadas. Consequentemente, certas UPs serão mais eficientes para determinados tipos de operações. Sendo assim, (1) *Tipo de Tarefa* representa o comportamento que um grupo de tarefas semelhantes geralmente apresenta em termos de tempo de execução para cada tipo de UP. Todas as tarefas que pertencem a um mesmo tipo terão um tempo de execução específico para cada UP, o qual é informado pelo usuário. A fim de permitir que a simulação seja o mais real possível, os usuários podem definir a *Taxa de Erro* (2), um intervalo (i.e., o valor mínimo e máximo de erro) que corresponde ao quanto se espera que o tempo de execução de uma tarefa pode variar. Depois de definir essas duas características, o próximo passo é criar as tarefas propriamente ditas. *D-STHARk* permite aos usuários inserir todas as tarefas que compõem a aplicação principal a ser simulada, onde cada tarefa deve pertencer a um dos tipos de tarefas acima referidos. De acordo com o tipo de tarefa e a taxa de erro, um tempo aleatório é definido para cada tarefa para as diferentes UPs. Finalmente, a terceira característica exigida pelo *D-STHARk* é o *Tamanho da Carga de Trabalho* (3) em MB. Essa característica é relevante, uma vez que muitas estratégias de escalonamento consideram o tamanho dos dados e o número de operações a serem executadas em cada UP para estimar o custo de comunicação. Por questões de simplicidade, consideramos que o custo de transferência varia linearmente com o tamanho dos dados. A fim de identificar cada tarefa criada, os usuários podem atribuir um identificador único (ID) para cada tarefa no *D-STHARk*. Usando esses IDs, os usuários podem designar dependências entre tarefas distintas que compõem uma aplicação, como ocorre em cenários reais. Por exemplo, a dependência de uma tarefa $t1$ e uma $t2$ significa que $t2$ deve ser executado antes de $t1$. Mais uma vez, os usuários podem salvar todas essas configurações.

**3. Definição das Estratégias de Escalonamento.** Atualmente, a *D-STHARk* tem quatro estratégias implementadas: (1) FCFS (*First Come First Served*); (2) HEFT (*Heterogeneous Earliest Finish Time*); e (3) HEFT-DA (*Heterogeneous Earliest Finish Time Data-Aware*); e (4) SEQ (Sequencial), uma serialização simples de todas as tarefas a serem executadas. Enquanto a primeira estratégia é muito tradicional em trabalhos relacionados a escalonadores, a segunda e terceira estratégias foram propostas e extensivamente avaliadas em [Andrade et al. 2014a]. A última estratégia é utilizada como uma

linha de base para contrastar a diferença de desempenho das estratégias. Os usuários podem selecionar diferentes escalonadores para compor uma única simulação e, neste caso, *D-STHARk* executa individualmente cada escalonador selecionado, permitindo aos usuários comparar a performance alcançada por eles.

**4. Execução Experimental e Avaliação.** Nesta etapa, os usuários analisam e confirmam todas as configurações. Em seguida o *D-STHARk* começa a executar a simulação. A ferramenta também permite definir quantas vezes cada simulação será executada. Os resultados são relatados como a média destas execuções. O desvio padrão também é apresentado. Durante a execução de simulação, *D-STHARk* exibe uma visualização do *log* de execução detalhando a simulação e mostrando qual tarefa está sendo executada por cada UP em cada momento. Esta visualização de *log* pode ser usada para depurar os escalonadores, verificando se eles estão funcionando como esperado. Ao final da simulação, *D-STHARk* apresenta os resultados detalhados para cada escalonador simulado, o quais podem ser exportados para um arquivo de saída. Mais especificamente, nossa ferramenta mostra (1) Speedup alcançados; (2) histograma com a distribuição de tarefas entre as UPs; e (3) porcentagem de processamento realizado por cada UP.

### 3.2. Processo de Execução de Simulações

Para realizar a simulação propriamente dita, o *D-STHARk* cria uma thread[1] distinta, nomeada **Worker Thread**, para simular cada UP. Além disso, uma thread *principal* é instanciada para ler as configurações da tarefa (rotina GetTask) e enviá-la para execução (rotina SubmitTask). Por fim, a thread principal cria uma fila para cada **Worker Thread** que será gerida de acordo com a estratégia de escalonamento definida na configuração.

O gerenciamento de tarefas é executado simultaneamente com as execuções de tarefas, como em ambientes de escalonamento dinâmicos reais. Depois de receber uma tarefa, cada **Worker Thread** verifica suas dependências. Se a mesma possuir dependências, será inserida na **Stuck Task List**, que armazena todas as tarefas que ainda não tiveram suas dependências executadas. Caso contrário, será enviada para o método **PushTask** que insere a tarefa em uma das filas, de acordo com a política de escalonamento.

Quando uma **Worker Thread** específica fica ociosa, ela busca uma nova tarefa a ser executada utilizando o método *PopTask*, definida de acordo com cada estratégia de escalonamento dinâmico. Como mencionado acima, cada tarefa tem um tempo de execução específico definido para cada UP. Assim, por meio de um s̈leepä **Worker Thread** simula o tempo de execução na UP correspondente. Quando uma tarefa simula o tratamento de dados, uma segunda chamada ao método *sleep* é feita para simular o tempo de transferência de dados da memória do CPU para a memória do coprocessador. Esse tempo é calculado de acordo com a largura da banda definida, bem como o tamanho da carga de trabalho. Por exemplo, a PCI-Express 16x bus oferece uma velocidade de transferência de $4GB$ por segundo. Assim, para cada MegaByte de dados da tarefa, realizamos um *sleep* de $0.000244141s$, aproximando-se das condições reais de execução. Finalizando o processo de execução, a **Worker Thread** verifica a **Stuck Task List** e remove todas as dependências associadas à tarefa que acabou de ser executada. Em seguida, a **thread principal** começa a buscar tarefas da **Stuck Task List** com todas as

[1]Implementamos as threads usando a biblioteca pThread padrão.

suas dependências resolvidas. Desta forma, garantimos que as tarefas serão executadas na ordem correta. A Figura 1 ilustra o processo acima descrito.

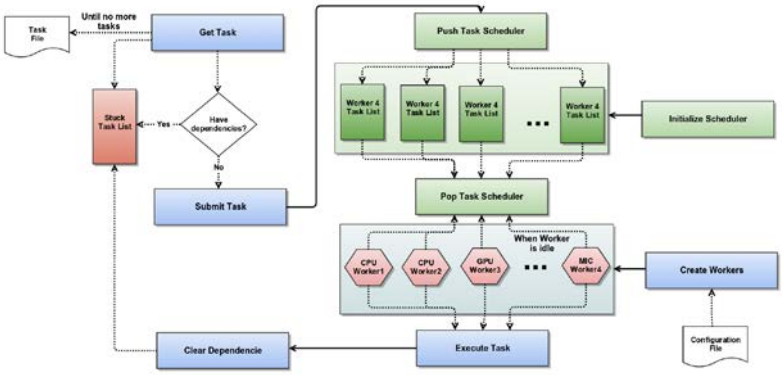



**Figura 1. Processo de Execução de Simulações**

### 3.3. API do Escalonador Dinâmico

O *D-STHARk* permite que os usuários incluam as suas próprias estratégias de escalonamento. Para este propósito, nossa ferramenta oferece uma API que especifica uma diretriz a ser seguida a fim de tornar essas novas estratégias compatíveis com o sistema. Basicamente, todos os novos escalonadores devem ser implementados na linguagem C. Essas implementações podem usar diferentes rotinas e nomes de arquivos, que devem ser carregados no *D-STHARk* através da GUI disponível. No entanto, o código-fonte deve ter um arquivo nomeado *scheduler.c*, nomeado, com quatro rotinas obrigatórias:

**1. InitializeScheduler**: responsável por iniciar componentes do escalonador. Ela também cria as filas de tarefas, associando-as com a respectiva UP.

**2. PushTask**: define qual estratégia será usada para escalonar as tarefas.

**3. PopTask**: esta rotina é chamada por cada **Worker Thread**, sempre que esta estiver ociosa, definindo a próxima tarefa a ser executada.

**4. DestroyScheduler**: responsável por finalizar o escalonador. Nesta rotina a memória usada pelo escalonador deve ser liberada.

## 4. Avaliação Experimental

Nesta seção, apresentamos os experimentos realizados para avaliar *D-STHARk*, que consistem em simular o mesmo cenário originalmente apresentado em [Andrade et al. 2014a], em que os autores avaliaram diferentes estratégias de escalonamento dinâmico para uma aplicação real [Teodoro et al. 2014]. Nossas avaliações contrastam os resultados simulados contra os originais, considerando ambos os tempos de execução e distribuição de tarefas entre UPs. Além disso, simulamos também o uso de mais coprocessadores do que inicialmente relatado para avaliar a utilidade do *D-STHARk* em fornecer análises mais amplas.

### 4.1. Configuração da simulação

#### 4.1.1. Aplicação da Simulação

Em nossos experimentos, simulamos uma aplicação relacionada a estudos de câncer no cérebro [Teodoro et al. 2014], que visa encontrar melhores classificações para os tumores

usando *Whole Tissue Slide Images (WSIs)*. Esta aplicação também particiona cada WSI em vários pedaços *tiles* de imagem que podem ser analisados de forma independente. Há muitas fases nesta análise de imagens, mas as que demandam a maior tempo de execução são a de segmentação (*segmentation*) e computação de características (*features computation*). Estas duas fases têm sido o foco de otimização de execução em máquinas híbridas em grande escala [Andrade et al. 2014a]. Cada pedaço de imagem é submetido a diferentes operações, formando um fluxo de execução como mostrado na Figura 2.

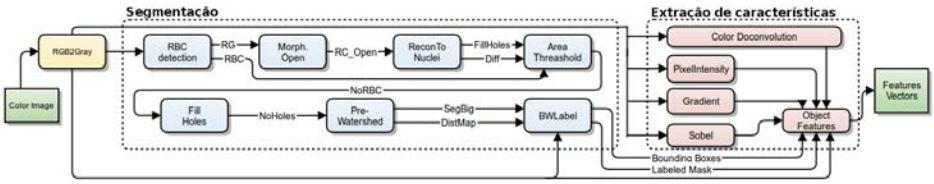



**Figura 2. Fluxo de Execução da Aplicação**

No *D-STHARk*, instanciamos um **tipo de tarefa** para cada operação, representados como retângulos coloridos na figura acima. O desempenho relativo (i.e., tempo específico) para cada UP em cada tipo de tarefa foi definido com base nos valores originalmente reportados em [Teodoro et al. 2014]. A taxa de erro foi configurada como zero, seguindo os experimentos originais [Andrade et al. 2014a]. O processamento de vários pedaços de imagem é um problema embaraçosamente paralelo, uma vez que cada um deles determina um fluxo de execução diferente. Assim, instanciamos uma tarefa individual para cada operação distinta que é aplicada em cada pedaço de imagem. Além disso, inserimos as dependências entre tarefas, como mostrado no fluxo de execução, para simular a ordem correta. Para ilustrar esse processo, considere as operações *Morph.Open* e *Recon.Nuclei* aplicadas a dois pedaços de imagem. Primeiro, criamos dois tipos de tarefa, $Tipo1$ (i.e., *Morph.Open*) e $Tipo2$ (i.e., *Recon.Nuclei*). Então, *D-STHARk* determina automaticamente os tempos de execução nas UPs para cada tipo de tarefa, com base nas configurações previamente definidas. Em seguida, são instanciadas as tarefas relacionadas a cada pedaço da imagem: (a) $t1$ do $Tipo1$ para o primeiro pedaço; (b) $t2$ do $Tipo2$ para o primeiro pedaço; (c) $t3$ do $Tipo1$ para o segundo pedaço; e (d) $t4$ do $Tipo2$ para o primeiro pedaço. Por fim, adicionamos as dependências de $t2$ para $t1$ e de $t4$ para $t3$, garantindo assim a ordem correta da execução.

Simulamos 800 pedaços imagem, o que gera 10.400 tarefas para execução. Além disso, no artigo original, os autores relataram que cada pedaço apresenta uma resolução de $4Kx4K$ pixels. Considerando uma representação de $256$ cores, cada pedaço de imagem tem um tamanho de $15,25MB$, que foi a carga de trabalho instanciada para cada tarefa.

### 4.1.2. Escalonadores Simulados

As características da aplicação discutida acima induzem comportamentos bastante semelhantes em duas estratégias (i.e., HEFT e HEFT-DA). Assim restringimos nossa análise a duas estratégias avaliadas em [Andrade et al. 2014a]:

- **FCFS:** é uma estratégia de escalonamento baseada em uma fila global. Quando uma **Worker Thread** está ociosa, ela busca a primeira tarefa na inicio da fila.
- **HEFT**: Mantém uma fila de tarefas para cada UP. A distribuição de tarefas entre as UPs é definida de acordo com as suas capacidades de processamento. Mais

especificamente, o escalonador mantém um histórico dos tempos de execução e, assim, atribui uma tarefa a uma UP específica que minimize a Equação 1.

$$min_{Pi}(Avail(Pi) + Est_{Pi}(T)) \quad (1)$$

$Pi$ é a UP a ser avaliada; $Avail(Pi)$ representa a quantidade de tempo que leva para $Pi$ processar todas as tarefas que lhe são atribuídas; e $Est_{Pi}$ denota o tempo estimado que $Pi$ leva para executar uma tarefa específica $T$.

#### 4.1.3. Arquiteturas Simuladas

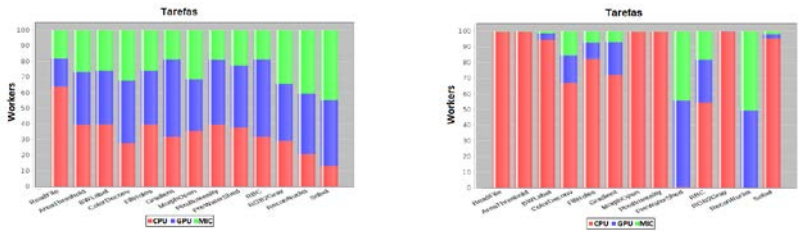
Basicamente, nossas análises consideram três arquiteturas híbridas distintas [Andrade et al. 2014a]: (1) **CPU-GPU**: 15 núcleos de CPU e 1 GPU; (2) **CPU-MIC**: 15 núcleos de CPU e 1 MIC; e (3) **CPU-GPU-MIC**: 14 núcleos de CPU, 1 GPU, e 1 MIC. Avaliamos todos os escalonadores simulados em cada uma dessas arquiteturas. Além disso, simulamos outros cenários, considerando mais coprocessadores do que relatado em [Andrade et al. 2014a], no intuito de evidenciar a utilidade do *D-STHARk* em fornecer análises mais amplas. Em todas essas análises, definimos a largura de banda como $4GB$, como uma PCI-Express 16x usada nos experimentos originais.

### 4.2. Análise dos Resultados

Primeiramente, analisamos o quão semelhante foi a distribuição de tarefas definida por cada estratégia de escalonamento na arquitetura simulada com as distribuições observadas na arquitetura real. Em seguida, investigamos o quão similar estão os tempos de execução de nossas simulações para os tempos medidos em uma arquitetura real. Para estas duas questões, consideramos a arquitetura de **CPU-GPU-MIC**, como avaliado em [Andrade et al. 2014a]. Finalmente, estendemos a avaliação original para outros tipos de arquiteturas, a fim de chegar a conclusões mais amplas. Ressaltamos também que todos os tempos discutidos abaixo representam a média de 10 execuções distintas.

#### 4.2.1. Distribuição de Tarefas

Em [Andrade et al. 2014a], os autores avaliaram o impacto do número de pedaços de imagem distintas simultaneamente processadas por cada *Worker Thread*, concluindo que o desempenho geral pode ser melhorado utilizando altos níveis de concorrência (ou seja, 55 pedaços de imagem). Em nossas análises, os escalonadores simulados funcionaram sem limitações sobre este número, apresentando os resultados encontrados pelos autores. A Figura 3 apresenta os resultados do *D-STHARk*. Em primeiro lugar, observa-se que o escalonador FCFS para as UPs é quase o mesmo para todas as operações, demonstrando que o FCFS não é capaz de tirar pleno proveito sobre a variabilidade de desempenho entre as operações. Por outro lado, podemos notar que HEFT tem priorizado o uso de núcleos de CPU para a maioria das tarefas, independentemente do desempenho das outras UPs. Conforme apresentado em [Teodoro et al. 2014], algumas operações (e.g., *PreWaterShad*, *ReconNuclei* e *RBC*), com elevados custos computacionais, podem diminuir o tempo de execução usando coprocessadores. Na verdade, HEFT foi capaz de identificar essas características, reduzindo o tempo total de execução. Uma discussão mais detalhada destes resultados pode ser encontrada no trabalho original [Andrade et al. 2014a].

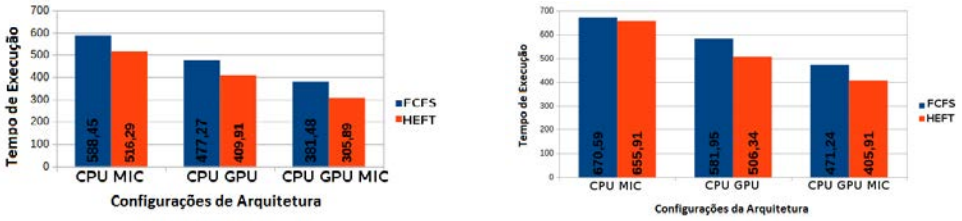



(a) FCFS (b) HEFT

**Figura 3. Distribuição de tarefas executadas na Simulação entre as UPs.**

### 4.2.2. Tempos de Execução

Os tempos de execução dos escalonadores são apresentados na Figura 4 (a) e (b), que correspondem aos resultados simulados e reais, respectivamente. Mais uma vez, os resultados obtidos utilizando *D-STHARk* são bastante semelhantes aos obtidos usando uma arquitetura real sobre a qual FCFS atinge o pior desempenho entre todas as configurações. Como relatado em [Andrade et al. 2014a], é importante salientar que a utilização da GPU produz sempre um bom desempenho. Por exemplo, o melhor tempo de execução CPU-GPU é de cerca de 1,26× mais rápido do que o melhor tempo de execução CPU-MIC. Este resultado nos motiva a realizar os experimentos descritos a seguir, a fim de demonstrar a aplicabilidade da nossa ferramenta.




(a) Execução Simulada (b) Execução Real

**Figura 4. Desempenho dos Escalonadores para Diferentes Arquiteturas Híbridas.**

### 4.2.3. Aumento do número de coprocessadores

Apesar de nas conclusões do trabalho [Andrade et al. 2014a] os autores mencionarem que aumentar o número de coprocessadores pode melhorar o desempenho da aplicação, este fato não foi verificado. Nesta seção, fornecemos esta análise, configurando diferentes arquiteturas no *D-STHARk*. Usando apenas o escalonador HEFT, diminuímos o número de núcleos de CPU e aumentamos o número de coprocessadores. A Tabela 1 apresenta os resultados dessa análise. Podemos notar que, aumentando apenas um coprocessador MIC é possível remover 2 núcleos de CPU. De modo semelhante, aumentando uma GPU é possível remover 4 núcleos de CPU, ambos sem degradar o desempenho do sistema. Estes resultados demonstram que podemos reduzir o consumo de energia (reduzindo núcleos de CPU e aumentando os coprocessadores), com o mesmo desempenho.

| Configuração dos Processadores | Tempo(s) do escalonador HEFT |
|---|---|
| 14 CPUs + 1 GPU +1 MIC | 305.892 |
| 11 CPUs + 1 GPU + 2 MIC | 309.537 |
| 9 CPUs + 2 GPU + 1 MIC | 301.079 |
| 4 CPUs + 2 GPU + 2 MIC | 302.646 |

**Tabela 1. Simulando Arquiteturas variando o número de coprocessadores.**

## 5. Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos *D-STHARk* (**D**ynamic **S**cheduling of **T**asks in **H**ybrid Simulated **AR**chitectures), cujo objetivo é fornecer um ambiente simulado completo de arquiteturas híbridas (CPUs, GPUs e MICs) que permite avaliar estratégias de escalonamento dinâmico de aplicações. Avaliamos nossa ferramenta simulando uma aplicação real [Teodoro et al. 2014], bem como a arquitetura e métodos de escalonamento apresentados em [Andrade et al. 2014a]. Os resultados e conclusões obtidos com *D-STHARk* foram os mesmos originalmente relatados, mostrando a efetividade da nossa proposta. Além disso, realizamos um experimento simulando configurações de arquiteturas compostas de mais coprocessadores, as quais não foram avaliadas no trabalho original devido à falta de uma arquitetura real. Nossos resultados alcançaram as mesmas conclusões levantadas no trabalho original de que é possível reduzir o consumo de energia, mantendo a desempenho, utilizando mais coprocessadores. Como trabalho futuro, visamos eliminar algumas simplificações adotadas, incluir outras estratégias de escalonamento, bem como avaliar tarefas dinâmicas e aplicações não embaraçosamente paralelas.

## Referências

Andrade, G., Ferreira, R., Teodoro, G., da Rocha, L. C., Saltz, J. H., and Kurç, T. M. (2014a). Efficient execution of microscopy image analysis on cpu, gpu, and MIC equipped cluster systems. In *26th IEEE SBAC-PAD Paris, France*.

Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Clua, E., Ferreira, R., and Rocha, L. (2014b). Efficient dynamic scheduling of heterogeneous applications in hybrid architectures. In *Proceedings of ACM SAC*, pages 866–871.

Augonnet, C., Thibault, S., and Namyst, R. (2010). StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E., and Labarta, J. (2012). Productive Programming of GPU Clusters with OmpSs. In *26th IEEE IPDPS*.

Fatica, M. and Luebke, D. (2007). High performance computing with CUDA. Supercomputing 2007 tutorial. In Supercomputing 2007 tutorial notes.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Lima, J. V. F., Broquedis, F., Gautier, T., and Raffin, B. (2013). Preliminary experiments with xkaapi on intel xeon phi coprocessor. In *25th IEEE SBAC-PAD*, pages 105–112.

Ravi, V., Ma, W., Chiu, D., and Agrawal, G. (2014). Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ACM Supercomputing*.

Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. (2011). PTask: operating system abstractions to manage GPUs as compute devices. In *ACM SOSP*.

Teodoro, G., Kurc, T., Kong, J., Cooper, L., and Saltz, J. (2014). Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU: A Case Study from Microscopy Image Analysis. In *IPDPS*.