UMA ADAPTAÇÃO DO ESCALONADOR ESTÁTICO DA BIBLIOTECA DATAFLOW SUCURI PARA COMPUTAÇÃO IN-SITU

Caio Bonfatti Gomes de Carvalho

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Rio de Janeiro
Março de 2018

UMA ADAPTAÇÃO DO ESCALONADOR ESTÁTICO DA BIBLIOTECA DATAFLOW SUCURI PARA COMPUTAÇÃO IN-SITU

Caio Bonfatti Gomes de Carvalho

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Carvalho, Caio Bonfatti Gomes de

Uma Adaptação do Escalonador Estático da Biblioteca Dataflow Sucuri para Computação In-Situ/Caio Bonfatti Gomes de Carvalho. – Rio de Janeiro: UFRJ/COPPE, 2018.

VIII, 56 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Leandro Augusto Justen Marzulo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 51 – 56.

1. Dataflow. 2. In-situ. 3. Edge. 4. Fog. 5. Escalonamento. 6. Smart Storage. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Para Karen.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA ADAPTAÇÃO DO ESCALONADOR ESTÁTICO DA BIBLIOTECA DATAFLOW SUCURI PARA COMPUTAÇÃO IN-SITU

Caio Bonfatti Gomes de Carvalho

Março/2018

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Programa: Engenharia de Sistemas e Computação

No modelo de computação por fluxo de dados, `dataflow`, as instruções ou tarefas são executadas de acordo com as dependências de dados, ao invés de seguir a ordem do programa, permitindo a exploração natural de paralelismo. A Sucuri é uma biblioteca dataflow em Python que permite aos usuários especificarem suas aplicações como um grafo de dependências, e as executa de maneira transparente em *clusters* de multicores, enquanto se encarrega também dos problemas de escalonamento. Avanços recentes em computação *fog* e *in-situ* assumem que dispositivos de armazenamento e de rede serão equipados com unidades de processamento, geralmente de baixo consumo de energia e baixo desempenho. Uma decisão importante em tais sistemas é a de migrar os dados para os processadores tradicionais, pagando os custos de comunicação, ou realizar a computação onde o dado está (*in-situ*), usando um processador de desempenho potencialmente menor. Este trabalho apresenta um estudo dos diferentes fatores que devem ser levados em consideração quando executando aplicações dataflow em um ambiente de computação *in-situ*. A Sucuri foi utilizada para gerenciar a execução em um pequeno sistema composto de um computador comum e uma placa Parallella simulando um disco inteligente, e uma série de experimentos foi realizada para mostrar como o tamanho dos dados a serem transferidos, a latência de rede, a perda de pacotes e a complexidade computacional da aplicação afetam o tempo de execução quando o processamento é deixado a cargo desse disco. Então, uma solução de escalonamento estático que leva em conta tais fatores é apresentada, dando a Sucuri poder de decisão sobre onde melhor realizar o processamento dos dados, evitando fazer a computação *in-situ* quando a mesma não trouxer ganhos de performance.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

AN ADAPTATION OF THE DATAFLOW LIBRARY SUCURI STATIC SCHEDULER FOR IN-SITU COMPUTING

Caio Bonfatti Gomes de Carvalho

March/2018

Advisors: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Department: Systems Engineering and Computer Science

In the dataflow computation model, instructions or tasks are executed according to data dependencies, instead of following program order, thus allowing parallelism to be exposed naturally. Sucuri is a dataflow library for Python that allows each user to specify their application as a dependency graph and execute it transparently in clusters of multicores, while taking care of scheduling issues. Recent trends in Fog and In-situ computing assume that storage and network devices will be equipped with processing elements that usually have lower power consumption and performance. An important decision for such systems is whether to move data to traditional processors (paying the communication costs), or to perform the computation where the data sits, using a potentially slower processor. Hence, runtime environments that deal with that trade-off are of extreme importance. This work presents a study on different factors that should be considered when running dataflow applications in a In-situ environment. We use Sucuri to manage the execution in a small system with a regular PC and a Parallella board, emulating a smart storage, and a set of experiments was performed to show how data transfer size, network latency, packet loss rates and computational complexity affect execution time when outsourcing computation to the smart storage. Then, a static scheduling solution is presented, allowing Sucuri to take the best decision where to execute the application, avoiding outsourcing when there would be no performance gains.

Sumário

Lista de Figuras

Capítulo 1

Introdução

A programação paralela é fundamental para que se possa extrair ao máximo o poder computacional das arquiteturas modernas, geralmente compostas por, além de processadores tradicionais de múltiplos núcleos, diferentes co-processadores e aceleradores, tais como unidades de processamento gráfico (*Graphics Processing Units* - GPUs), processadores Intel® Xeon Phi™ e circuitos integrados reprogramáveis (*Field Programmable Gate Arrays* - FPGAs) [1]. Além disso, aplicações recentes de Inteligência Artificial, Big Data e Internet das Coisas (*Internet of Things* - IoT) produzem um grande e crescente volume de dados que precisam ser eficientemente processados e armazenados, geralmente de maneira distribuída. No planejamento de tais aplicações, deve-se considerar o impacto no desempenho e consumo de energia causado pela movimentação de dados entre os dispositivos de memória e armazenamento e as unidades de processamento.

Os casos citados necessitam de um conjunto de ferramentas que ajude os programadores no desenvolvimento de aplicações paralelas e distribuídas que possam executar em uma ampla gama de dispositivos diferentes, ferramentas estas que devem liberar os desenvolvedores de terem que cuidar de aspectos tecnológicos com relação a criação de tarefas, sincronização e escalonamento ciente do ambiente utilizado, por exemplo.

Edge/Fog/In-situ Computing (computação de borda/névoa/no-local) [2–6] propõem levar o processamento para perto de onde o dado está, adicionando poder de computação a dispositivos de armazenamento [7–9], equipamentos de infraestrutura de rede [10] como *switches*, roteadores e placas de rede ou até mesmo utilizando aparelhos móveis. Esses dispositivos inteligentes seriam capazes de realizar parte da computação necessária e assim reduzir o tráfego de dados na rede ou no barramento, e ainda poderiam ser equipados com processadores customizados para a aplicação, o que poderia resultar em bom desempenho mesmo em sistemas de baixa potência.

O modelo de programação *dataflow* é um bom candidato a ser utilizado em computação In-situ, já que provê uma maneira simples e natural de se explorar

paralelismo. Um programa *dataflow* é geralmente representado como um grafo direcionado, no qual tarefas ou instruções são representadas por nós e as dependências de dados são indicadas por arestas entre esses nós. No modelo *dataflow*, instruções ou tarefas podem ser executadas tão logo seus operandos de entrada estiverem prontos, não precisando seguir a ordem do programa, e portanto tarefas independentes podem ser naturalmente identificadas e executadas em paralelo, havendo recursos suficientes. Ambientes de execução e interfaces de programação de aplicações (*Application Programming Interface* - API) *dataflow* [11–16] podem ser usados sobre a atualmente dominante arquitetura Von Neumann com desempenho equivalente ao das mais utilizadas ferramentas para programação paralela, como *OpenMP* e *Pthreads*.

Este trabalho visa a utilizar o modelo *dataflow* na arquitetura de aplicações que utilizem abordagem *In-situ*. Já que todas as dependências de dados estão descritas explicitamente no grafo, ambientes de execução *dataflow* podem se utilizar dessa informação para escalonar as tarefas de acordo com as demandas *In-situ*. É proposta a utilização da Sucuri [17–19] para orquestrar os dispositivos inteligentes presentes no sistema, e desenvolvido um escalonador estático que leva em consideração a localidade dos dados. Além disso, as semelhanças entre as necessidades dos ambientes de *Edge/Fog* e *In-situ computing*, do ponto de vista do escalonamento de tarefas ciente da localidade dos dados, tornam as soluções *In-situ* apresentadas neste trabalho promissoras à serem estudas posteriormente em ambientes específicos de *Edge/Fog Computing*.

Sucuri é uma biblioteca *dataflow* para a linguagem de programação Python que disponibiliza uma interface amigável para programação paralela, na qual os desenvolvedores podem fornecer funções personalizadas em nós e preencher as dependências entre eles apenas conectando-os com arestas em um grafo. A Sucuri também cria uma camada de abstração, utilizando a biblioteca de troca de mensagens amplamente utilizada MPI (*Message Passing Interface*), para comunicação entre máquinas remotas de maneira transparente.

Para emular um disco de armazenamento inteligente, foi utilizada uma placa Parallella [20], equipada com uma Xilinx Zynq Z7010 (processador ARM Cortex A9 *dual core* mais uma FPGA) [21] e um processador Epiphany RISC de 16 núcleos. A Parallella rodou um sistema operacional Linux em um cartão de memória SD, onde também foram armazenados os arquivos de entrada dos experimentos realizados, e um computador tradicional, de maior poder computacional, foi ligado a ela através de uma conexão ethernet. Neste trabalho, apenas o processador ARM da Parallella foi utilizado.

A solução proposta foi desenvolvida e avaliada em dois estágios.

Inicialmente, o escalonador estático da Sucuri foi modificado para dar sempre

prioridade à localidade dos dados, alocando os nós que possuem arquivos de entrada nas máquinas onde esses arquivos estão localizados. Duas aplicações artificiais de processamento de texto foram desenvolvidas, e foram executados experimentos variando-se os tamanhos dos arquivos de entrada, a latência e a perda de pacotes na rede.

Os resultados mostraram que, mesmo para um dispositivo de potência menor e com limitado poder de processamento, é possível obter-se ganhos de desempenho ao se evitar transferências de dados desnecessárias. Os ganhos são maximizados quando os custos computacionais das tarefas são pequenos e os arquivos de entrada tem tamanho grande. Além disso, cenários com maior latência de rede e perdas de pacotes tornam a solução mais atrativa. Ao mesmo tempo, aplicações de custo computacional maior mostraram-se desafiadoras para o pequeno poder de processamento da Parallella, culminando em grande perda de desempenho em casos específicos, o que deixou claro a necessidade de um escalonador que pudesse levar em conta não só a localidade dos dados, como também o poder de processamento dos diversos dispositivos do sistema.

A partir das conclusões obtidas no primeiro estágio, o escalonador foi modificado para tentar prever o tempo de transferência dos arquivos de entrada e o tempo de execução das tarefas nos diferentes dispositivos do sistema, e desta forma decidir se é melhor alocar a computação no disco inteligente, mais lento, ou mover os dados para computá-los no computador tradicional, mais poderoso. Foi criada uma aplicação externa responsável por coletar e servir ao escalonador da Sucuri os dados de que ele necessita para decidir se deve ou não realizar computação *In-situ*, tais como o tempo de transferência dos arquivos entre os *hosts* e o poder computacional deles.

O *benchmark* desta vez constitui-se de um conjunto de aplicações de busca e ordenação, de complexidades computacionais conhecidas, e novamente foram realizados experimentos variando-se os tamanhos dos arquivos de entrada e as condições de rede.

Do ponto de vista do desempenho geral das aplicações, os resultados alinharam-se com os anteriores, como era de se esperar: ganhos de desempenho para custo computacional menor, arquivos maiores e condições de rede deterioradas. No entanto, o novo mecanismo de escalonamento estático proposto nesse trabalho foi capaz de tomar boas decisões e evitar o processamento *In-situ* quando este fosse degradar o desempenho.

Desta forma, as principais contribuições deste trabalho são:

- Estudo da viabilidade e ganhos de desempenho em se realizar computação *In-situ* em um ambiente simulando um disco inteligente.

- Estudo do impacto da complexidade das aplicações e das condições da rede

(latência, perda de pacotes) no desempenho de sistemas em tal ambiente.

- Desenvolvimento de uma aplicação externa para medição e cálculo de *bandwidth*, *delay* (latência), *goodput* (*throughput* no nível da aplicação), tempo de transferência de arquivos entre *hosts* e perfil de desempenho por complexidade, e disponibilização desses dados para a Sucuri por meio de arquivos.

- Criação de um escalonador estático capaz de decidir, num ambiente de computação *Edge/Fog*, onde é melhor alocar os nós com arquivos de entrada — se em máquinas tradicionais, mais potentes, ou *In-situ*, em dispositivos inteligentes, mas possivelmente com menor poder de processamento.

- Extensão da biblioteca *dataflow* Sucuri para que a mesma possa trabalhar de maneira transparente e eficiente em ambientes de computação *In-situ*, assim como ela já o fazia em ambientes de máquinas *multicore* e *clusters*.

Os resultados da primeira fase deste trabalho foram publicados como prova de conceito no MPP 2017 (*Workshop on Parallel Programming Models*) [22], tendo recebido o prêmio de melhor artigo e sido convidado a submeter uma extensão ao periódico *International Journal of Grid and Utility Computing* (IJGUC), edição especial "Avanços em Modelos de Programação Paralela de Alto Nível para Computações Edge/Fog/In-situ". Ao IJGUC foi submetido o segundo estágio do trabalho e aguarda-se o resultado.

O restante deste trabalho está organizado da seguinte maneira: o Capítulo 2 relata os trabalhos relacionados na área de *Edge/Fog Computing*, *In-situ*, ou *dataflow*, o Capítulo 3 expõe os conceitos de *Edge/Fog/In-situ Computing*, suas semelhanças e diferenças, no Capítulo 4 é tecida uma breve discussão sobre escalonamento e no Capítulo 5 o conceito do modelo de programação *dataflow*, juntamente com a biblioteca Sucuri e detalhes sobre sua implementação e modificações para este trabalho, são apresentados. Então, no Capítulo 6 são detalhados os experimentos realizados e discutidos seus resultados, e por fim o Capítulo 7 apresenta as conclusões desta dissertação e vislumbra alguns trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo são apresentados trabalhos relacionados à esta dissertação. São mencionados trabalhos relativos à programação *dataflow*, localidade de dados e computação *fog*, por exemplo, evidenciando as diferenças e pontos comuns a este trabalho, bem como possíveis oportunidades de troca de experiência com eles. Ao final, são mencionados exemplos de sistemas de arquivos que estão sendo considerados para serem integrados à Sucuri em trabalhos futuros.

Corral [23] é um *framework* que se aproveita da previsibilidade das tarefas recorrentes em *clusters* para realizar escalonamento *offline*. O *framework* tenta colocar a computação em conjunto com os dados para aprimorar a localidade. Além disso, a fim de reduzir a interferência entre as tarefas, ele executa isolamento temporal e espacial, o que significa que o escalonador tentará colocar as tarefas em regiões diferentes do *cluster* e tentará evitar a execução concorrente de certos trabalhos. O Corral foi implementado no Apache Yarn e executado em um *cluster* de duzentas e dez máquinas, reduzindo o *makespan* (tempo para se finalizar a execução de todos os trabalhos) em até 33% em comparação com o escalonador do Yarn. Além disso, houve uma redução do tempo médio de conclusão (tempo médio entre a chegada de um trabalho e sua conclusão) de até 56%, e 20-90% de redução na transferência de dados entre *racks*. Apesar de o Corral não ser um ambiente de execução, suas políticas de escalonamento podem ser úteis como exemplo para a Sucuri.

Swift/T [14] é uma linguagem de descrição e ambiente de execução que suporta a criação dinâmica de fluxos de trabalho com diferentes granularidades de tarefas e executa em plataformas com um imenso número de elementos de processamento. A Swift/T emprega balanceamento de carga dinâmico assíncrono (*Asynchronous Dynamic Load Balancing* — ADLB) para distribuir as tarefas entre os nós computacionais. No entanto, o compartilhamento de dados entre as tarefas é feito através de um sistema de arquivos paralelo, o que pode causar degradação do desempenho devido a pobre localidade entre os dados e a interferência com outras aplicações. Em [24], os autores propõe explorar a localidade nas aplicações Swift/T através do

Hércules, um armazenador em-memória distribuído, baseado no Memcached [25]. A implementação do Swift/T foi otimizada para escalonar os trabalhos computacionais nos nós onde os dados requeridos encontram-se armazenados. A proposta foi avaliada utilizando uma aplicação sintética que acessa arquivos com diferentes padrões, mostrando resultados promissores. Nesta dissertação, no entanto, é dado um passo a frente, focando-se em escalonar tarefas em dispositivos de armazenamento.

O trabalho em [7] apresenta a BlueDBM, uma arquitetura de sistema que emprega armazenamento baseado em *flash* com capacidade de processamento *In-situ* e uma rede inter-controladora de baixa latência a alta vazão. O sistema é composto de vinte nós, cada um tendo 1TB de armazenamento *flash*. Os dispositivos de armazenamento foram construídos a partir de placas personalizadas de alta capacidade equipadas com FPGAs. Eles foram organizados em uma rede de latência quase uniforme para que pudessem oferecer um espaço de endereços global. A BlueDBM permite que os usuários implementem máquinas de processamento personalizadas e cartões *flash* são projetados para expor um conjunto de interfaces de software que habilitam otimizações específicas para as aplicações nos acessos a esses cartões: (*i*) uma interface para sistema de arquivos, (*ii*) uma interface para *drivers* de dispositivo de blocos e (*iii*) uma interface para o acelerador. Resultados de experimentos preliminares mostraram ganhos de desempenho de até uma ordem de magnitude melhores que *clusters* equipados com discos de estado sólido convencionais. A BlueDBM é um exemplo de arquitetura construída a partir de dispositivos inteligentes que poderia se beneficiar de um ambiente de execução baseado no modelo de programação *dataflow* que tornasse o uso da tecnologia transparente ao usuário, tal qual a Sucuri faz com a placa Parallella neste trabalho. Por outro lado, o ambiente de execução *dataflow* da Sucuri também sofreria melhorias se implementado sobre soluções de discos inteligentes desse porte.

Uma abordagem *dataflow* para o desenvolvimento de aplicações IoT baseadas em *Fog* é apresentada em [5]. O trabalho propõe um *framework* para facilitar a programação de aplicações IoT, utilizando uma ferramenta visual que permite ao desenvolvedor estipular as categorias dos nós e conectá-los de acordo com suas dependências, gerando um grafo *dataflow*. Nesse sentido, o trabalho dos autores assemelha-se a este, no entanto os nós são separados por funcionalidade, já que [5] foca na especialização dos nós — como sensores e atuadores —, sem o objetivo de ganhos de desempenho conseguidos por paralelismo, característica da Sucuri.

Em [26], os autores apresentam ffMDF, um suporte de execução leve para *kernels* de álgebra linear densa. Ele possui um interpretador macro-dataflow dinâmico que processa grafos acíclicos direcionados gerados em tempo de execução. Apesar de desenvolvido especificamente para máquinas de múltiplos núcleos e *cache* compartilhada, e não para cenários mais heterogêneos, seria interessante aprimorar a

Sucuri para esses casos nos quais não se tem o grafo completo no início da aplicação, prestando-se atenção ao *overhead* que isso poderia causar no escalonamento.

O Sistema de Arquivos do Google (*Google File System* - GFS) [27] é um sistema de arquivos distribuído para Linux desenvolvido pela Google para prover uma maneira eficiente e confiável para o armazenamento de arquivos em *clusters* de máquinas de prateleira. O GFS é baseado em mecanismos altamente escaláveis para o acesso a arquivos grandes. Os arquivos são divididos em pedaços espalhados por múltiplos nós, e o mapeamento dessa distribuição é mantido em estruturas indexadas no sistema de arquivos. A arquitetura é baseada no padrão mestre-escravo, com o mestre ficando a cargo de manter os metadados dos arquivos e os escravos, presentes nos diferentes nós do sistema, são responsáveis por armazenar as porções dos arquivos.

Uma evolução do GFS é o sistema de arquivos distribuído multi-plataforma *Hadoop Distributed File System* (HDFS [28, 29]), que por padrão utiliza porções maiores de arquivo do que o GFS (128MB *vs.* 64MB). Enquanto o GFS suporta um modelo baseado em múltiplas escritas e múltiplas leituras, o HDFS trabalha com múltiplas leituras e apenas uma escrita, no qual apenas um modo é suportado para a atualização dos arquivos: o modo *append*, em que um arquivo pode ser modificado apenas acrescentando-se conteúdo ao seu final, diferentemente do que ocorre no GFS, que permite que os arquivos sejam modificados em posições aleatórias. Essa aparente limitação do HDFS permite que ele escale muito bem em cenários altamente distribuídos, e se encaixe perfeitamente com o modelo de programação *MapReduce* implementado no Apache Hadoop.

Já que este trabalho é voltado para habilitar suporte para computação *In-situ* na Sucuri, incluindo-se aspectos de escalonamento, não foi tentada a integração com soluções já existentes de sistemas de arquivos distribuídos, como os já citados GFS e HDFS. Ao invés disso, a solução dispõe de um catálogo de arquivos próprio, mais simples, mas que não leva em consideração a divisão de arquivos grandes em partes ou a redundância de arquivos. Estudar e implementar tal integração é parte de trabalhos futuros.

Capítulo 3

Edge, Fog e In-situ Computing

Neste capítulo são apresentados os conceitos de *Edge, Fog*, e *In-situ Computing* (computação de borda, névoa e no local), os quais estão ligados entre si por suas denominações, relativas ao local onde o processamento dos dados é realizado. Apesar deste trabalho ter foco em *In-situ*, as definições de *Fog* e *Edge* são mostradas aqui como exemplo de ambientes onde a solução apresentada nesta dissertação também poderia ser utilizada, com adaptações e experimentos a cargo de trabalhos futuros.

O número de dispositivos eletrônicos conectados tanto à Internet quanto à redes locais vem crescendo de maneira rápida e intensa. Aplicações de IoT, Inteligência Artificial, redes celulares modernas (5G) e dispositivos móveis, por exemplo, produzem ou consomem uma quantidade cada vez maior de dados, que geralmente devem trafegar pela rede e ser processados em servidores na nuvem (*Cloud*). É esperado que a quantidade de dados comunicada pelos dispositivos desses sistemas cresça de 1,1 zetabytes (89 exabytes) por ano em 2016 para 2,3 zetabytes (194 exabytes) por ano até 2020, e a arquitetura atual baseada em *Cloud* tende ou a ficar saturada [2], ou a desperdiçar informações por falta de infraestrutura para processá-las.

3.1 A Computação em Névoa — Fog

Com o intuito de se contornar as limitações da infraestrutura de rede que vêm aparecendo nos cenários de computação *Cloud*, para poder prover melhor serviço à aplicações de missão crítica e/ou de uso intensivo de dados, a solução de computação em névoa (*Fog Computing*) vem ganhando espaço. Nas arquiteturas *Fog*, diversos componentes das aplicações, tais como processamento, armazenamento e controle, são seletivamente levados para próximo das bordas da rede, onde os dados são gerados, acarretando inúmeras vantagens sobre a computação puramente *Cloud*: diminuição de latência, alívio no congestionamento da rede, e potencial melhoria em privacidade e segurança, já que os dados trafegam menores distâncias. A computação *Fog*, portanto, é "uma arquitetura horizontal, de nível de sistema, que

Tabela 3.1: Algumas diferenças entre as computações *Cloud* e *Fog* (reproduzido de [4])

	Cloud	Fog
Latência	Alta (eventualmente consistente)	Baixa (localidade)
Acesso	Fixo e sem-fio	Principalmente sem-fio
Acesso aos Serviços	Pelo núcleo	Nas bordas/por dispositivos móveis
Disponibilidade	99,99%	Altamente volátil / redundante
# de usuários/dispositivos	Dezenas/centenas de milhões	Dezenas de bilhões
Preço por dispositivo (2014)	$1500 - 3000	$50-200
Principal gerador de dados	Humanos	Dispositivos/sensores
Geração de conteúdo	Localização central	Qualquer lugar
Consumo de conteúdo	Dispositivos-fim	Qualquer lugar

distribui funções de computação, armazenamento, controle e rede para mais próximo ao usuário, ao longo do continuum nuvem-para-coisa" [2].

A computação em névoa, no sentido apresentado, pode ser entendida como uma extensão da computação em nuvem na qual as implementações da arquitetura distribuem-se pelas várias camadas da topologia da rede. "(Computação em) Névoa, simplesmente porque a névoa é uma nuvem próxima ao chão" [3]. No entanto, a maioria dos autores defende que, além de uma extensão que por muitas vezes deve ser usada em conjunto com a computação em nuvem, a computação em névoa também é um novo paradigma capaz de trazer, por suas características, mudanças nas práticas atualmente comuns nos serviços de computação, tal como aconteceu com a *Cloud*. A arquitetura *Fog*, como mencionado, propõe o espalhamento das variadas funções de computação ao longo das várias camadas da topologia de rede, partindo de um servidor *Cloud* até os dispositivos-fim, procurando realizar sua função tão perto da borda da rede quanto seja possível e necessário. Desta forma, o que antes era feito nos servidores na nuvem pode passar a ser realizado em roteadores, *switches*, ou até nos dispositivos finais, abrindo várias oportunidades para o processamento *In-situ*.

Na tabela 3.1 podem ser encontradas, de maneira resumida, algumas das principais diferenças entre as computações *Cloud* e *Fog*.

3.2 A Computação na Borda — Edge

A Computação na Borda possui uma estreita relação com a Computação em Névoa, já que ambas foram idealizadas com o objetivo de se evitar os problemas e gargalos da Computação em Nuvem (ao mesmo tempo em que tiram vantagem da capacidade de processamento dos dispositivos modernos), e ambas apresentam como solução realizar tarefas nas bordas da rede, mais próximo ao usuário, afastando-se do núcleo *Cloud*. O próprio termo "borda"é utilizado com frequência para se designar o local onde pode ser realizada a computação em *Fog*. No entanto, pode-se perceber algumas

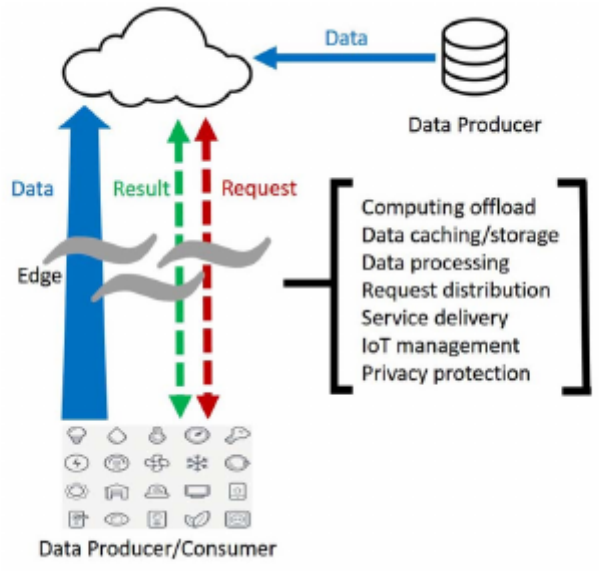
diferenças, ainda não padronizadas na literatura.

O consórcio OpenFog [2] defende que há diferenças chave entre computação *Fog* e *Edge*: *Fog* trabalha em conjunto com a nuvem, enquanto *Edge* a exclui; *Fog* é hierarquica, ao passo que *Edge* se distribui em um pequeno número de camadas; e *Fog*, ao contrário de *Edge*, também trabalha com gerenciamento de rede, armazenamento, controle e aceleração.

Já para os autores de [6], *Edge* e *Fog* são intercambiáveis, com a computação *Edge* focando mais no lado das "coisas"(dispositivos finais, tais como celulares ou sensores), enquanto a computação *Fog* teria um foco maior na infraestrutura.

A Figura 3.1 mostra uma diferença entre os paradigmas *Cloud* e *Edge*: em *Cloud*, existe uma relação em que os dispositivos finais são consumidores de conteúdos produzidos na nuvem; já em *Edge*, os dispositivos finais podem ser tanto produtores, quanto consumidores.

(a) Paradigma Cloud



(b) Paradigma Edge

Figura 3.1: Comparação entre os paradigmas de computação Cloud e Edge. (Reproduzido de [6])

3.3 Computação In-situ

O avanço recente em equipamentos como os de infraestrutura (roteadores, *switches*), armazenamento (discos, memórias), sensores e dispositivos móveis (celular, *tablets*), dotando-os de poder de processamento suficiente para que possam exercer funções além daquelas para as quais foram especializados, foi um dos principais responsáveis pelo avanço de *Fog* e *Edge Computing*.

O processamento *In-situ*, nesse contexto, significa levar o processamento aonde o dado está, diferentemente da maneira tradicional na qual o dado é transportado para o local (máquina, dispositivo) onde a computação será realizada, e o Capítulo 2 deu exemplo de vários trabalhos que utilizam a computação *In-situ*. O processamento *In-situ* permite à aplicação trabalhar a localidade dos dados, isto é, evitar pagar os custos de comunicação de dados na rede, o que pode ser vantajoso tanto em termos de desempenho, quanto melhora da latência da aplicação ou estado geral da rede. No entanto, deve-se sempre levar em conta o objetivo da aplicação e quais seus requisitos, para determinar se a computação *In-situ* será vantajosa na operação estudada.

Para fins práticos, aqui tratamos *Edge/Fog/In-situ* quase como sinônimos, pois a natureza de nossa aplicação e experimentos pôde tratar esses conceitos de maneira genérica.

Capítulo 4

Escalonamento de Tarefas

Este capítulo apresenta o conceito e alguns exemplos de escalonamento de tarefas. Ele começa discutindo alocação de recursos e os clássicos problemas de alocação de processos em um processador, para então discutir o escalonamento em ambientes paralelos de múltiplas máquinas. Por fim, o *List Scheduling* (LS), escalonamento sobre o qual foram baseadas as soluções desse trabalho, é exibido.

O escalonamento é o processo pelo qual uma determinada aplicação ou sistema decide em quais recursos cada tarefa (processo, trabalho...) irá rodar, e em que momento. O escalonamento é parte fundamental dos sistemas operacionais modernos, já que nesses sistemas um sem número de processos está competindo pelo uso do processador. O escalonamento tem impacto direto no tempo de execução do sistema, e portanto no seu desempenho. Um escalonador é um processo responsável por realizar o escalonamento em seu ambiente.

No âmbito dos sistemas operacionais de propósito geral, e considerando um ambiente multi-usuário, o escalonador possui algumas funções básicas que deve levar em consideração em seu critério de escolha, a saber [30]:

- *Utilização do processador*: deve cuidar para que os processadores não estejam ociosos.
- *Throughput*: deve tentar maximizar o throughput, ou seja, o número de processos executados em um determinado intervalo de tempo.
- *Tempo de Espera*: minimizar o tempo em que os processos permanecem ociosos na fila de prontos, aguardando serem alocados.
- *Tempo de Turnaround*: minimizar o tempo total de execução do processo.
- *Tempo de Resposta*: minimizar o tempo entre uma requisição feita ao sistema e sua resposta.

Idealmente, o escalonador deveria maximizar a utilização do processador e o throughput, enquanto minimizaria os outros critérios. Na prática, no entanto, pode-se observar que alguns critérios são conflitantes com outros, ou seja, existe uma relação de compensação na escolha desses parâmetros, e deve-se analisar com cuidado o sistema para que se possa dar prioridade ao critério correto.

Os escalonamentos podem ser divididos ainda em *preemptivos*, que podem interromper uma tarefa para executar outra, e *não-preemptivos*, que deixam o processo rodar até que este termine ou libere o recurso de maneira voluntária, e ainda em *estáticos*, que são realizados antes de se rodar a aplicação, e *dinâmicos*, que trabalham em conjunto com as aplicações, alocando tarefas em tempo de execução.

Existem diversos algoritmos de escalonamento, e a análise detalhada de cada um deles foge ao escopo desse trabalho. Como exemplo, pode-se citar dois algoritmos bastante utilizados no contexto de sistemas operacionais: o escalonamento FIFO (*First-In-First-Out* — Primeiro-a-Entrar-Primeiro-a-Sair), algoritmo simples no qual os processos vão entrando no final de uma fila de prontos e sendo retirados pelo escalonador do começo dessa fila; e o escalonamento *Round Robin*, o qual atribui uma fatia de tempo a cada processo, e caso o processo não tenha terminado quando do fim de sua fatia de tempo, o escalonador irá substitui-lo pelo próximo da lista, de maneira circular, tornando o *Round Robin* um escalonamento justo, do ponto de vista do aproveitamento de recursos do sistema.

De maneira geral, as mesmas considerações feitas para escalonamento em sistemas operacionais são válidas para as situações de *clusters*, sistemas de alto desempenho, ou ainda ambientes de *Edge/Fog/In-situ Computing*: continua sendo de grande interesse maximizar o *throughtput* e minimizar o tempo de *turnaround*, por exemplo. Além disso, os algoritmos desenvolvidos para a alocação de processos no sistema operacional podem ser aproveitados e adaptados para escalonamento em outros contextos.

Uma decisão importante a ser tomada em ambiente distribuído é se o escalonador será centralizado, distribuindo a alocação a partir de uma máquina, o que simplifica a implementação, mas leva a gargalos de desempenho por todos os nós terem que se comunicar com o escalonador mestre, ou se a alocação será distribuída, com vários ou todos os nós contando com uma implementação funcional do escalonador. A Sucuri, inicialmente, possuía um escalonador centralizado, que logo foi substituído por um distribuído com o esperado ganho de desempenho.

Encontrar um escalonador ótimo, que sempre minimize o tempo de execução da aplicação, não é uma tarefa fácil, no entanto. O problema do escalonamento em máquinas idênticas (*multiprocessor scheduling problem*) consiste em, dadas m máquinas idênticas e n tarefas com tempo de execução conhecido, encontrar uma atribuição das tarefas às máquinas que minimize o tempo máximo da operação,

formalmente definido em [31] como:

PROBLEMA: ESCALONAMENTO(m,n,t): Dados inteiros positivos m, n e um tempo t_i em $\mathbb{Q}_{\geq}$, para cada i em $\{1, ..., n\}$ encontrar uma partição $\{M_1, ..., M_m\}$ de $\{1, ..., n\}$ que minimize $\max_j t(M_j)$.

Aqui, $t(M_j) := \sum_{i \in M_j} t_i$, uma partição de $\{1, ..., n\}$ em m blocos é um escalonamento e $\max_j t(M_j)$ é o custo do escalonamento. Dessa forma, o problema é traduzido por: dados m, n e t, encontrar um escalonamento de custo mínimo. O problema de se encontrar o escalonamento de custo mínimo foi provado por Garey e Johnson em [32] como um problema NP-difícil, mesmo utilizando-se apenas duas máquinas. Graham [33] propôs uma solução aproximativa simples para o problema: alocar as tarefas uma a uma, destinando cada tarefa a máquina menos ocupada. É um algoritmo de tempo polinomial e 2-aproximativo em relação a solução ótima do ESCALONAMENTO (provas em [32]), e pela sua simplicidade vale sempre ser considerado, utilizado as vezes como complemento de algum algoritmo mais complexo.

Um dos algoritmos mais utilizados para contornar esse problema é o *List Scheduling* (Escalonamento por Lista — LS), uma heurística considerada dominante nesse caso [34, 35]. No LS, à cada nó do grafo *dataflow* é dada uma prioridade, e então eles são postos em uma lista ordenada de maneira decrescente quanto a prioridade, de onde vão sendo retirados pelo escalonador à medida em que vão sendo liberados recursos. A Sucuri, neste trabalho, utiliza uma variação do LS, a ser detalhada no Capítulo 5.

Capítulo 5

Execução Guiada por Fluxo de Dados e a Biblioteca Sucuri

Neste Capítulo é apresentado o modelo de programação *dataflow*, um modelo para execução guiada por fluxo de dados que permite a exploração de paralelismo de maneira natural. Então é apresentada a Sucuri, biblioteca em Python para *dataflow*, com sua arquitetura, método de programação e detalhes da implementação realizada nesse trabalho.

5.1 O Modelo Dataflow

As arquiteturas utilizadas atualmente pela maioria dos computadores, baseadas nos ciclos de busca/execução e modelo de Von Neumann, são essencialmente sequenciais, pois são guiadas pelo incremento de um contador de programas que controla qual parte do código deve ser executada a cada momento, ou seja, são arquiteturas orientadas ao fluxo de controle. Para contornar essa situação, diversos métodos foram criados na tentativa de se adicionar capacidade de paralelismo a essa arquitetura, com o objetivo de se obter ganhos de desempenho (trechos de códigos de uma aplicação rodando simultaneamente tem grande potencial para diminuir seu tempo de execução), como o paralelismo em nível de instrução com suas diversas técnicas (*pipelining* com predição de desvio, por exemplo), ou o paralelismo em nível de *thread*, mas tais técnicas apenas escondem a latência de um modelo que continua sequencial.

Mais tarde, processadores de múltiplos núcleos juntaram-se à equação. Apesar de poder oferecer paralelismo real às aplicações, sua utilização ainda sofre de gargalos (de comunicação no barramento e acesso à memória, por exemplo) e continua não sendo trivial programar-se eficientemente aplicações *multicore*, portanto o acréscimo de um ou mais processadores à uma máquina nem ao menos pode garantir ganho

de desempenho. Além disso, mesmo que com capacidade de se executar trechos de códigos de maneira concorrente, as aplicações continuam presas ao paradigma sequencial da busca/execução guiada pelo contador de programas.

No final da década de 1960 e início da década de 1970, começaram a aparecer ideias de uma arquitetura que tivesse sua execução guiada pelo fluxo de dados [36]. O modelo *dataflow* (fluxo de dados), como foi chamado, propõe que as instruções possam ser executadas tão logo seus operandos de entrada estejam disponíveis, não sendo mais necessário que as tarefas aguardem sua vez na ordem do programa. Desta forma, essa arquitetura expõe de maneira natural o paralelismo potencial das aplicações, liberando as partes que não dependem entre si para rodarem simultaneamente, limitadas agora apenas à quantidade de recursos disponíveis.

No modelo *dataflow*, uma aplicação é representada por um grafo direcionado acíclico (DAG), no qual os vértices simbolizam as instruções e as arestas, os dados que fluem entre as instruções, ou seja, suas dependências. Esse grafo é usualmente chamado de grafo de dependências, ou grafo *dataflow*.

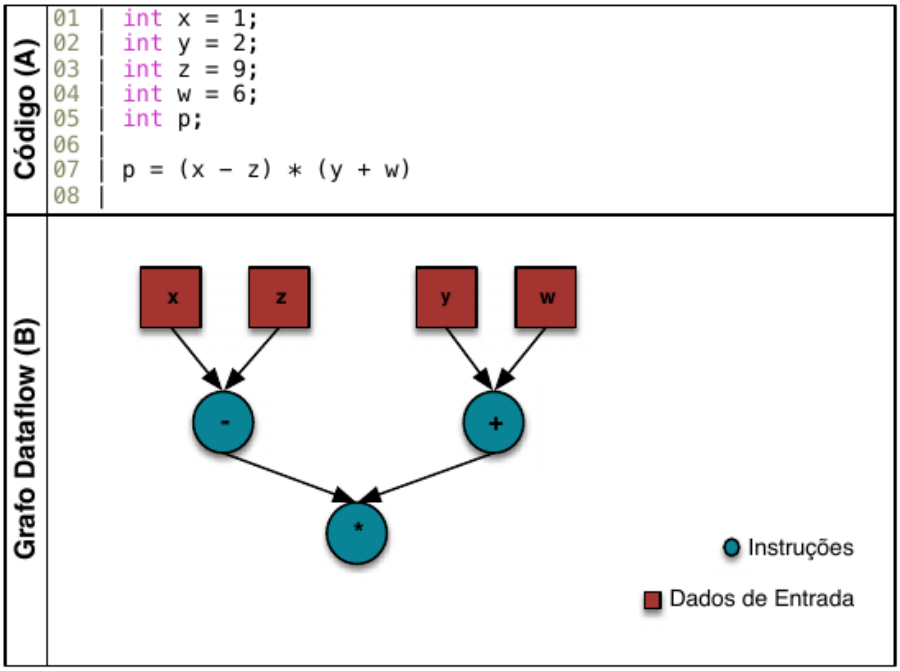


Figura 5.1: Exemplo de Grafo Dataflow. O painel (A) mostra um exemplo de código em alto nível, e o painel (B) exibe como ficaria a estrutura correspondente em um Grafo Dataflow (reproduzido de [37]).

A Figura 5.1 mostra um exemplo de grafo *dataflow*. O painel superior apresenta algumas atribuições e uma operação simples em uma linguagem de alto nível, e o painel inferior exemplifica como ficariam essas mesmas operações em um grafo *dataflow*. Os vértices redondos correspondem às instruções, e as arestas entre eles são os operandos produzidos pela origem e consumidos no destino — esses operandos representam as dependências das instruções, e as arestas nada mais são que o fluxo de

dados da aplicação. Importante notar que os vértices do topo são vértices especiais que devem ser estimulados para que a aplicação comece a rodar.

No início, vários trabalhos foram realizados com o intuito de se obter um *hardware* modificado que fosse especializado para o modelo *dataflow* [38–40]. Essas máquinas funcionariam em uma abordagem de grão fino em relação à *dataflow*, ou seja, toda instrução seria um vértice do grafo.

Um processador *dataflow* especializado contaria com alguns aspectos simplificados em relação ao original, pois não precisaria implementar toda a parte de contador de programas e fluxo de controle dos processadores tradicionais. Deveria possuir, no entanto, estruturas para o armazenamento e gerenciamento dos operandos, cujos detalhes são específicos a cada implementação.

Apesar de alguns resultados promissores, a granularidade fina das máquinas *dataflow* mostrou-se um problema para os equipamentos da época, pois nessa abordagem grafos de tamanho muito grande acabam sendo gerados e as estruturas para gerenciá-los acabam tendo custo alto. Esse fato, somado ao aumento exponencial de poder de processamento que os processadores tradicionais experimentaram por um longo período, fez com que o interesse nas máquinas *dataflow*, e no modelo de execução guiada por fluxo de dados um modo geral, diminuísse bastante.

Atualmente, com a virtual estagnação dos ganhos de desempenho por aumento de *clock* dos processadores, a computação paralela encontra grande destaque na área de arquitetura de computadores, e o modelo de execução guiada por fluxo de dados voltou a ser bastante abordado. Muitos trabalhos, no entanto, valendo-se das conclusões de [41] sobre granuralidade ótima em modelos *dataflow*, preferem uma granularidade mais intermediária do que a granularidade fina utilizada nas máquinas puramente *dataflow*, resultando em abordagens híbridas: organização da aplicação em grafo *dataflow*, no qual os vértices podem ser funções do usuário, por exemplo, e a execução de baixo nível a cargo de processadores Von Neumann tradicionais. Essa foi a abordagem escolhida por esta dissertação, e maiores detalhes sobre como ela implementa o modelo *dataflow* podem ser conhecidos na próxima seção.

5.2 Sucuri

Sucuri [17, 19] é uma biblioteca desenvolvida em Python que permite a programação utilizando modelo *dataflow* em um nível mais alto que a maioria das bibliotecas e ambientes de execução, com o intuito de facilitar o desenvolvimento para o usuário. Ela utiliza MPI de maneira transparente ao usuário para fornecer execução em *clusters* de *multicores*. Os principais componentes da Sucuri são `Nó`, `Grafo`, `Tarefa`, `Worker` e `Escalonador`, descritos a seguir:

- `Nós` são objetos associados a funções e conectados por arestas pelo programador, através do método `add_edge`. As arestas descrevem as dependências entre os dados e os nós representam o processamento a ser realizado quando as dependências são satisfeitas.

- Um objeto `Grafo` é utilizado como um contêiner, representando toda a aplicação *dataflow*.

- Uma `Tarefa` é criada pelo escalonador uma vez que todos os operandos de entrada de um certo nó tornam-se disponíveis. Cada tarefa contém a lista de operandos de entrada e o *id* dos nós relativos a eles.

- `Workers` são processos instanciados pela Sucuri para executar tarefas. Quando um *worker* está ocioso ele solicita uma tarefa a seu escalonador local. Assim que recebem uma tarefa, os *workers* consultam o nó correspondente a ela no grafo e executam a função associada. O número de *workers* por máquina é recebido por parâmetro pela Sucuri.

- Um `Escalonador` é responsável por casar os operandos de entrada e gerar tarefas, de acordo com a regra de disparo *dataflow*: cada operando enviado por um *worker* é armazenado em uma *Unidade de Correspondência* (*Matching Unit*) até que todos os operandos de entrada de um nó estejam disponíveis, resultando na instanciação de uma tarefa que é inserida em uma *Fila de Prontos* (*Ready Queue*) — de onde o escalonador irá retirá-la quando um *worker* ocioso fizer uma requisição.

A implementação original da Sucuri [17] continha um escalonador centralizado, o que significa que *workers* em máquinas remotas pediam tarefas e entregavam seus resultados a um escalonador local falso, responsável apenas por encaminhar mensagens ao escalonador principal. Essa abordagem leva a gargalos de desempenho, já que toda comunicação deve passar pelo mestre, bem como limitações no tamanho das aplicações a serem trabalhadas, pelo mesmo motivo, e por isso esta dissertação adota uma versão da Sucuri [19] que utiliza um escalonador distribuído, significando que existe uma instância do escalonador por máquina no sistema, cada uma delas responsável por gerar tarefas para um conjunto de nós. Os conjuntos de nós são fornecidos por um mecanismo estático que particiona o grafo entre os escalonadores da Sucuri.

A Figura 5.2 apresenta uma visão geral da arquitetura da Sucuri. Cada máquina possui seu próprio escalonador, com sua Fila de Prontos e sua *Matching Unit*. O grafo *dataflow* da aplicação é replicado em todas as máquinas e cada escalonador tem acesso a lista contendo os nós que foram estaticamente mapeados àquela máquina

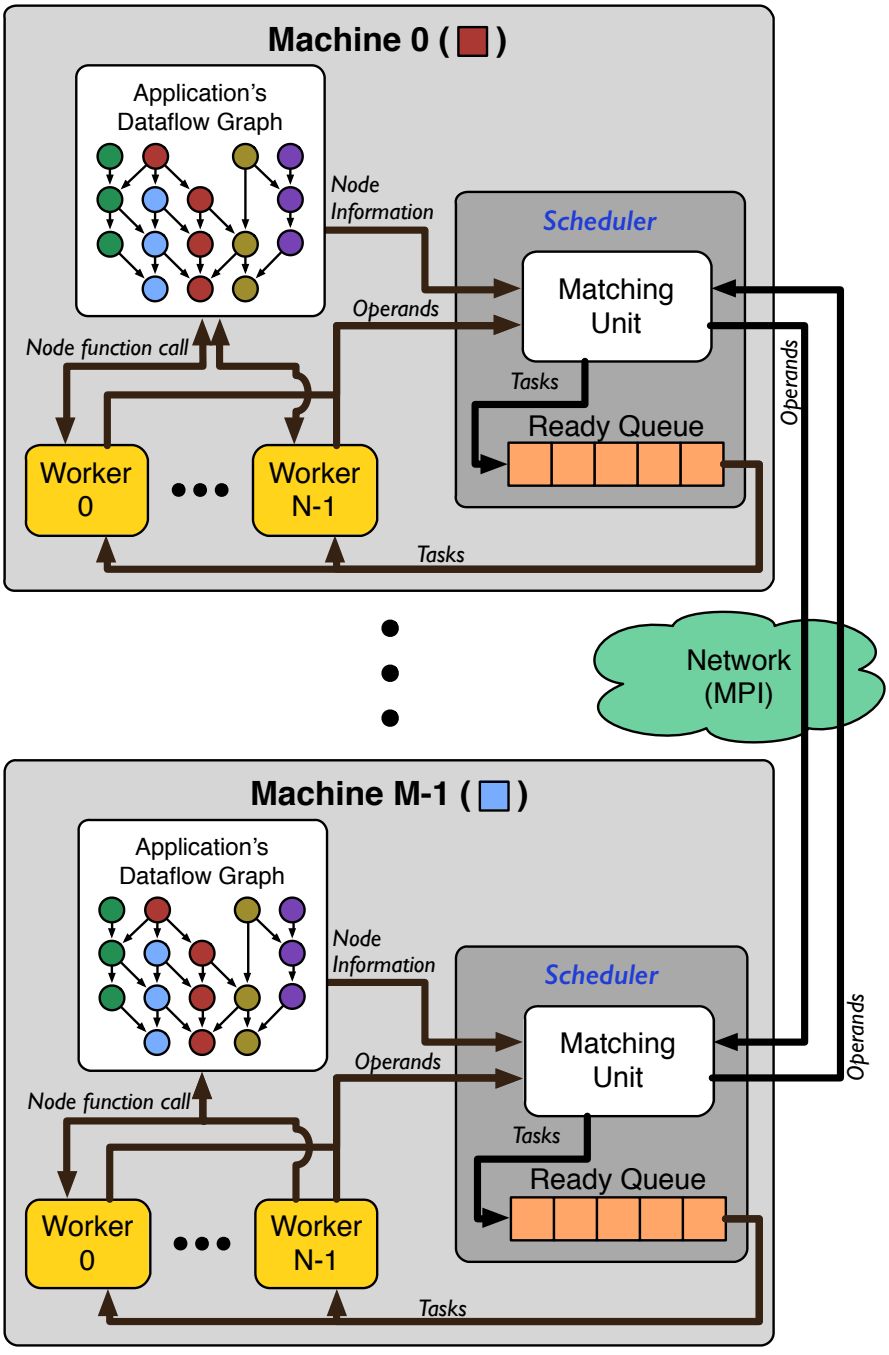
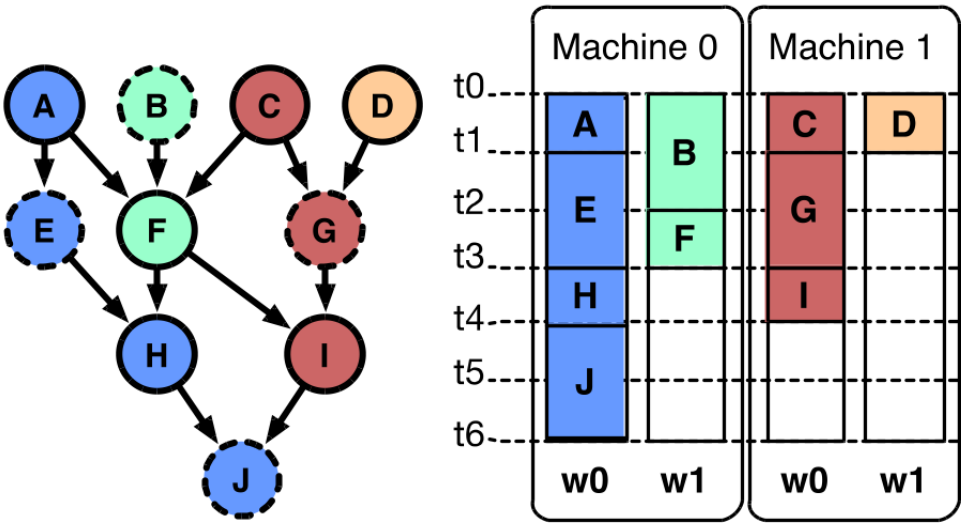


Figura 5.2: A Arquitetura Distribuída da Sucuri (reproduzido de [19]). A mesma estrutura é replicada em cada máquina, e o grafo *dataflow* da aplicação está separado por cores, indicando quais máquinas serão responsáveis por executar tarefas geradas por quais nós.

(o mapeamento é representado, na figura, por cores no grafo). À medida que vão sendo gerados pelos *workers*, os operandos vão sendo encaminhados para a *Matching Unit*, a qual contém também informações sobre os nós, para poder enviar as tarefas relativas a eles à Fila de Prontos. Os *workers*, por sua vez, quando ociosos pedirão uma tarefa ao escalonador, e ao recebê-la consultarão o grafo geral para descobrir qual o nó associado a ela, chamando sua função cadastrada.

O mecanismo de particionamento do grafo usado pela Sucuri distribuída [19] é baseado no algoritmo *List Scheduling* (LS) [34, 42–44]. A versão do LS implementada utiliza um esquema simples de prioridades de três níveis (*worker*, máquina e remoto), e cada nó do grafo tem um peso atribuído pelo programador que influencia a maneira com que o escalonador irá mapear os nós adjacentes a ele[1]. Os pesos representam o custo computacional de cada nó.

O particionamento começa com os nós raiz — aqueles que não possuem nós incidentes a eles — sendo alocados de maneira circular nos núcleos disponíveis dos processadores de cada máquina, e prossegue de acordo com as seguintes regras: (*i*) o escalonador tenta atribuir o nó ao mesmo *worker* de seu nó incidente de maior peso; (*ii*) o escalonador tenta atribuir o nó à mesma máquina de seu nó incidente de maior peso; (*iii*) caso os nós incidentes tenham o mesmo peso, ele aloca da maneira circular entre os *workers* deles; e (*iv*) se nenhum dos critérios anteriores foi atingido, aloca em uma máquina remota.



(a) Grafo (b) Alocação

Figura 5.3: Exemplo de particionamento pelo LS da Sucuri. Os nós de contorno tracejado possuem o dobro do peso dos demais, e as cores representam para qual *worker* (w) cada nó foi alocado. Reproduzido de [19].

[1]Em um grafo direcionado, dados dois nós i e j, i é incidente a j se, e somente se, há pelo menos uma aresta indo de i a j (i $\rightarrow$ j), e i é adjacente a j se há pelo menos uma aresta indo de j a i (i $\leftarrow$ j) [45].

A Figura 5.3 mostra um exemplo do resultado (Figura 5.3(b)) do particionamento de um grafo de dependências (Figura 5.3(a)) utilizando o método apresentado, em um ambiente com duas máquinas e dois *workers* em cada uma. As cores indicam os grupos de nós alocados em um mesmo *worker*, e os nós de contorno tracejado possuem o dobro do peso dos demais. Os quatro primeiros, raízes, foram alocados de maneira circular, e os demais foram distribuídos de acordo com seus nós incidentes, respeitando-se a ordem de prioridades *worker*/máquina/remoto, bem como os pesos declarados.

Importante mencionar que, como esta é uma política de escalonamento estático, a alocação dos nós pode ser efetuada apenas uma vez para várias execuções da aplicação. O mapeamento é gravado em um arquivo contendo em que máquina cada nó deve ser alocado, economizando-se o tempo de escalonamento, caso o usuário assim desejar.

5.2.1 Programação

Um dos principais objetivos da biblioteca é facilitar a utilização de paralelismo aos usuários através do modelo *dataflow*, resolvendo questões como sincronismo, concorrência e comunicação de forma transparente, e portanto a Sucuri conta com uma API simples e fácil de ser utilizada.

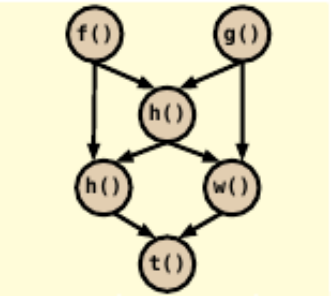
A Figura 5.4 exemplifica a criação de uma aplicação hipotética empregando a Sucuri. Na Figura 5.4(a) tem-se a representação de como seria o código sequencial, a Figura 5.4(b) representa o grafo *dataflow* resultante considerando as dependências do programa, e na Figura 5.4(c) é apresentado um código que faz uso da biblioteca. Considera-se que as funções f, g, h, w, e t são as funções especializadas da aplicação do usuário e estão implementadas em outros pontos do código.

O código Sucuri da Figura 5.4(c) é composto da seguinte maneira:

- A linha 1 importa a biblioteca Sucuri.
- Na linha 2 é instanciado o grafo *dataflow* da aplicação, ainda vazio.
- A linha 3 instancia o escalonador, que recebe o grafo instanciado e o número de *workers* a ser utilizado pela Sucuri.
- As linhas 4 a 9 criam os nós do grafo, associando uma função a cada um deles. Além disso, o número de operandos de entrada necessários à função associada é passado para o nó.
- Nas linhas 10 a 15, os nós criados anteriormente são adicionados ao grafo.

```
01 | a = f()
02 | b = g()
03 | c = h(a,b)
04 | d = h(a,c)
05 | e = w(c,b)
06 | x = t(d,e)
```

(a) Código Sequencial



(b) Grafo *Dataflow*

```
01 | from Sucuri import *
02 | graph = DFGraph()
03 | sched = Scheduler(graph, nWorkers)
04 | a = Node(f, 0)
05 | b = Node(g, 0)
06 | c = Node(h, 2)
07 | d = Node(h, 2)
08 | e = Node(w, 2)
09 | x = Node(t, 2)
10 | graph.add(a)
11 | graph.add(b)
12 | graph.add(c)
13 | graph.add(d)
14 | graph.add(e)
15 | graph.add(x)
16 | a.add_edge(c,0)
17 | b.add_edge(c,1)
18 | a.add_edge(d,0)
19 | c.add_edge(d,1)
20 | c.add_edge(e,0)
21 | b.add_edge(e,1)
22 | d.add_edge(x,0)
23 | e.add_edge(x,1)
24 | sched.start()
```

(c) Código Sucuri

Figura 5.4: Exemplo de criação de um código Sucuri, tomando-se como base um código sequencial e seu respectivo grafo de dependências. Reproduzido de [19].

- As linhas 16 a 23 conectam os nós criados, encerrando a construção do grafo *dataflow*. O método `add_edge` conecta um nó a um outro nó que lhe for passado por parâmetro, informando também em qual porta de entrada receberá os operandos oriundos desse nó.

- Por fim, na linha 24 o escalonador é inicializado, rodando a aplicação.

Esse mesmo código pode ser executado em uma única máquina, ou em um *cluster* de máquinas, bastando para isso que o usuário passe o parâmetro `mpi_enabled` como *TRUE* para o escalonador. A Sucuri utiliza memória compartilhada para comunicação local e MPI para comunicação entre máquinas.

Por padrão o escalonador considera que todos os nós tem peso 1. Caso deseje alterá-lo, o desenvolvedor deve passar o parâmetro `weight =` w na instanciação do nó, sendo w o novo peso desejado.

A biblioteca possui ainda *templates* [18] para facilitar ainda mais a programação no caso de alguns algoritmos que são padrões muito utilizados em programação paralela, tais como *fork/join*, *pipeline* e *wavefront*, no entanto não houve a necessidade de se utilizar *templates* nesse trabalho.

5.2.2 Prova de Conceito: Sucuri com Abordagem In-situ Forçada

Os pesos de cada nó devem ser informados pelo programador, ou devem ser obtidos utilizando-se ferramentas externas de *profiling*. No contexto de computação *In-situ*, em que dispositivos de diferentes capacidades de processamento estão conectados em uma rede heterogênea, a compensação entre computação e comunicação deve representar um papel importante no escalonamento. Além disso, no caso de discos inteligentes, no qual pode-se evitar a transferência de arquivos em favor de realizar o processamento nos próprios discos, também é importante que o ambiente de execução e o escalonador saibam onde os arquivos estão armazenados, a fim de estimar os tempos de transferência de maneira mais precisa. Desta feita, era necessário que a Sucuri fosse modificada para que pudesse tratar essas questões.

Em um primeiro estágio, foi realizada uma prova de conceito na qual o escalonador estático da Sucuri levava em conta a localidade dos dados de entrada ao alocar os nós nos *hosts* do sistema. O escalonador foi modificado da seguinte maneira:

1. O programador registra o(s) arquivo(s) no nó que o irá utilizar, através do método `register_file` e informa o nome e caminho do arquivo na instanciação do nó através do parâmetro `src = "arquivo"`.

2. A Sucuri descobre em que *host* cada arquivo está fisicamente armazenado, através da consulta ao sistema de arquivos de cada máquina, e guarda essa informação em um catálogo local (um catálogo por máquina).

3. A Sucuri, utilizando MPI, combina os catálogos locais para formar um catálogo global de arquivos para utilizar quando for necessário.

4. A biblioteca sonda as distâncias relativas entre as máquinas do sistema, utilizando o aplicativo `ping`, e constrói uma matriz de adjacências.

5. A Sucuri, então, inicia um mecanismo de escalonamento estático que particionará o grafo priorizando a localidade dos dados de entrada: ela primeiro tenta alocar os nós nas máquinas onde seus arquivos de entrada estão fisicamente armazenados (com a ajuda do catálogo global de arquivos criado anteriormente). Caso não seja possível, ela aloca na máquina disponível mais próxima, que descobre consultando a matriz de adjacências.

Neste primeiro momento, a localidade dos dados foi levada em consideração de maneira ingênua, já que o escalonador forçava o escalonamento *In-situ* sem se preocupar se isto traria vantagem ou não à aplicação em termos de desempenho. Essa abordagem simples foi utilizada como prova de conceito da viabilidade e necessidade de um escalonador consciente sobre os dados de entrada.

Uma série de experimentos, detalhada no Capítulo 6, foi realizada, variando-se os tamanhos dos arquivos de entrada e as condições de rede do sistema, e chegou-se a conclusão de que a abordagem *In-situ* no escalonamento era promissora, pelos bons ganhos de desempenho obtidos em algumas situações, mas precisava ser melhorada, pois, como já era esperado, a estratégia de sempre forçar processamento *In-situ* ocasionou perda de desempenho em certos casos. Os resultados foram publicados em [22].

5.2.3 Sucuri para Computação In-Situ

As conclusões da seção anterior sugeriram que, para transformar a Sucuri em um ambiente de execução *dataflow* para computação *In-situ*, novas modificações deveriam ser feitas no escalonador para que ele fosse capaz de determinar automaticamente se deveria alocar os nós mais próximos de onde os dados estão armazenados (processamento *In-situ*), ou se deveria mover os dados para uma outra máquina de desempenho computacional maior, pagando os custos de comunicação, o que significa que existe um compromisso (comunicação *vs* computação) a ser considerado. Ademais, um catálogo distribuído de arquivos, evoluído do catálogo global da primeira solução, foi adicionado ao escalonador para que este pudesse estimar os tempos

de transferência dos arquivos no contexto de discos inteligentes (com capacidade de computação In-situ) e utilizar essa informação durante o processo de escalonamento.

Sucuri Environment Setup — SES

A fim de tomar decisões com precisão, o escalonador estático da Sucuri precisa avaliar as informações de desempenho na plataforma computacional que está em uso. Para tanto, foi desenvolvida para este trabalho uma ferramenta externa chamada *Sucuri Environment Setup* (Configuração do Ambiente Sucuri - SES), que deve ser executada antes de se rodar aplicações com a biblioteca. A SES contém uma classe `Environment` responsável por mapear, consolidar e disponibilizar por meio de arquivos os dados do ambiente no qual a Sucuri irá rodar, incluindo o desempenho de elementos da rede e de processamento. A ferramenta provê uma lista de métodos para coletar informações da rede, construir um catálogo distribuído de arquivos, e estimar os tempos de computação e transferência. Esses métodos podem ser chamados em conjunto ou de maneira independente pela linha de comando.

A SES também é implementada em Python, assim como a Sucuri, e conta com o MPI para instanciar os processos remotos utilizados para reunir métricas de rede e desempenho, bem como montar o Catálogo de Arquivos com os tempos de transferência de cada um. A seguinte linha de comando pode ser utilizada para inicializar a SES:

```
mpiexec -machinefile <hostfile> -np <n>
python ses.py <hostfile> -D -B -P -f <paths>
```

Onde:

- `hostfile` é a lista de máquinas que poderão participar da computação;
- `n` é o número de *hosts*;
- `-D` informa a SES que ela deve estimar as latências de comunicação entre os *hosts*;
- `-B` informa a SES que ela deve estimar a *bandwidth* de rede entre as máquinas;
- `-P` informa a SES que ela deve estimar o desempenho de processamento de cada host.
- `-f <paths>` informa a SES que ela deve construir o *Catálogo de Arquivos* utilizando os nomes e caminhos dos arquivos indicados no arquivo "`paths`";

Importante deixar claro que é utilizada, nesse trabalho, a definição de *bandwidth* como largura de banda de rede, ou seja, a taxa máxima (bits/s) pela qual a informação pode ser transmitida pela rede, ao invés da faixa de frequências de transmissão no meio de propagação [46].

Coleta de Informações de Rede com a SES

Latência e largura de banda tem um impacto enorme nos tempos de transferência de arquivos pela rede. No entanto, devido a fatores como os custos de transmissão e recepção nas máquinas envolvidas, ou os custos dos próprios protocolos de comunicação, como o mecanismo de controle de congestionamento do protocolo TCP, a camada de aplicação consegue utilizar apenas uma fração da largura de banda disponibilizada pela rede. Já que o escalonador da Sucuri nesta nova abordagem precisa de uma boa estimativa do tempo de transferência dos arquivos entre os dispositivos, é necessário o cálculo da quantidade de dados úteis efetivamente recebida pela aplicação em um dado período de tempo (*goodput*) [47]. Para a coleta dessas informaçãos, a SES oferece os seguintes métodos: `getLatencies`, `getBandwidth` e `getGoodput`.

O método `getLatencies` mede a latência de comunicação entre cada par de *hosts* (ou dispositivo inteligente) utilizando o aplicativo `ping`, construindo uma matriz de adjacências das latências entre as máquinas do sistema.

O método `getBandwidth` mede a largura de banda de rede entre cada par de dispositivos do sistema, utilizando o aplicativo `iperf`, e constrói uma matriz de *bandwidths*. Como o cálculo de *bandwidth* é mais complexo e custoso, e considerando que o `iperf` utiliza um padrão de comunicação cliente/servidor, o `getBandwidth` preenche apenas a diagonal estritamente superior da matriz, para evitar a criação de matrizes muitos grandes e assim escalar melhor, e realiza as medidas em várias etapas, sendo que cada etapa é composta por um grupo de medições paralelas sem repetir as máquinas de destino ou origem, já que uma máquina comunicando-se simultaneamente com outras duas ou mais causaria interferência nas medidas.

O método `getGoodput` é responsável por estimar a quantidade de informações úteis à aplicação efetivamente recebida por ela em um período de tempo. Alguns trabalhos que apresentam modelagem do desempenho do protocolo TCP e que poderiam ser aproveitados para o cálculo de goodput [47–49], em sua maioria ou são muito restritivos, ou dependem de parâmetros que não são conseguidos de maneira trivial pela camada de aplicação a menos que haja uma análise da estrutura dos pacotes utilizados na comunicação, e por esse motivo optou-se aqui por uma abordagem experimental mais simples.

Intuitivamente, o tempo de transferência (T) será diretamente proporcional ao tamanho do arquivo (s) e à uma constante relativa ao *goodput* (g). Desta forma, g

pode ser obtido transferindo-se um arquivo pela rede, medindo-se o tempo gasto no processo e utilizando-se a seguinte relação:

$$g = \frac{T}{s} \tag{5.1}$$

Importante mencionar que o valor de g deve ser calculado com base na transferência de arquivos de tamanho comparável ou maior que a *bandwidth*, pois a utilização de arquivos menores distorceria a medida, já que os custos mencionados dominariam o tempo de transferência. Para contornar esse problema, uma segunda constante, o, foi adicionada à Equação (5.1). O valor de o pode ser obtido transferindo-se um pequeno arquivo (da ordem de bytes), considerando que $o = T$, quando s é muito pequeno. Assim, obtém-se a seguinte equação para o cálculo do tempo de transferência de arquivo:

$$T = s \times g + o \tag{5.2}$$

Essa abordagem, além de simples, não necessita levar em consideração quaisquer parâmetros dependentes de implementações específicas de protocolos, tornando-a flexível o suficiente para rodar em ambientes diversos. Além disso, não é necessária precisão extrema na estimativa, já que esse é apenas um valor para auxiliar o escalonador da Sucuri na sua tomada de decisão. Os resultados apresentados no Capítulo 6 validam esse mecanismo.

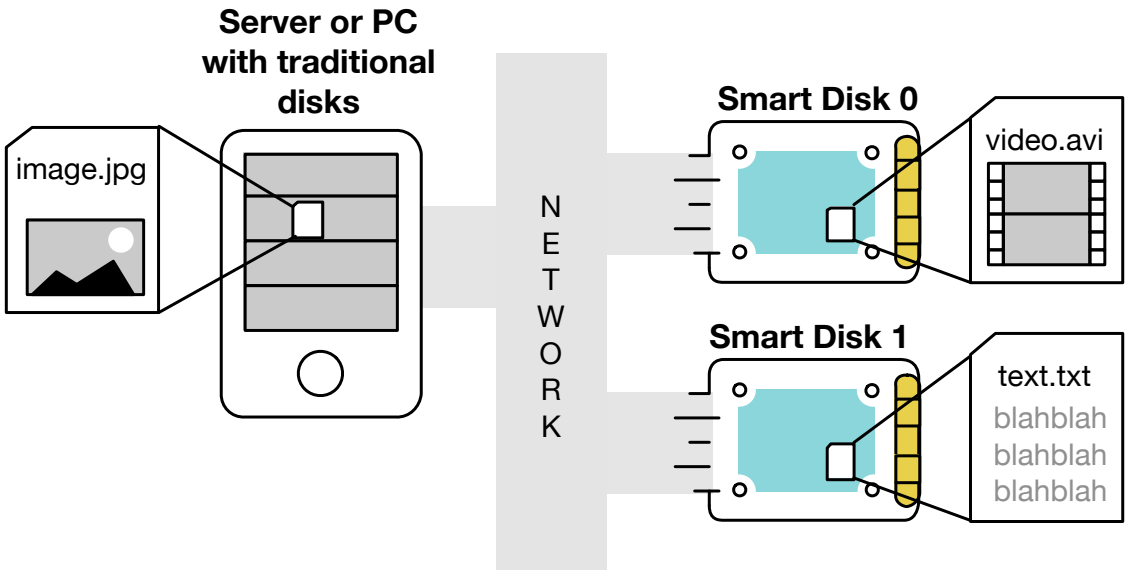
Construindo o Catálogo de Arquivos com a SES

A solução aqui implementada necessita que os usuários registrem os arquivos que serão utilizados em suas aplicações. Isto é necessário porque o escalonador da Sucuri precisa conhecer onde estão fisicamente armazenados os arquivos, para que possa estimar os tempos de transferência. O registro dos arquivos pode ser feito tanto pela chamada do método `registerFile`, da SES, quanto pela passagem de um arquivo de configuração, tal como visto anteriormente quando foram explicados os parâmetros que podem ser passados na inicialização da SES. A SES, então, descobre e registra o tamanho de cada arquivo e em que *host* estão armazenados.

Utilizando as informações coletadas, juntamente com a Equação (5.2), a ferramenta de configuração estima os tempos de transferência de cada arquivo entre todos os *hosts* do sistema, através do método `getTransferTimes`, e gera um *Catálogo de Arquivos* que será utilizado pelo escalonador estático da Sucuri, no momento de particionamento do grafo *dataflow*, para consultar o custo de transferência do arquivo desejado para a máquina especificada.

A Figura 5.5 apresenta um exemplo completo de como o usuário pode gerar um catálogo que registra três arquivos (`video.avi`, `image.jpg` and `text.txt`). A

Figura 5.5(a) mostra os arquivos distribuídos entre dois discos inteligentes e um servidor tradicional. A Figura 5.5(b) exemplifica o uso do método `registerFile` para informar quais arquivos farão parte do catálogo. Por fim, a Figura 5.5(c) exibe o Catálogo de Arquivos contendo os caminhos destes e os tempos de transferência para todos os *hosts* do ambiente.



(a) Localização dos Arquivos

```
ses.registerFile("/path1/image.jpg")
ses.registerFile("/path2/video.avi")
ses.registerFile("/path3/text.txt")
```

(b) Registro dos Arquivos

Caminho do Arquivo	Tempos de Transferência		
	Servidor	Disco 0	Disco 1
/path1/image.jpg	0	400.5	280.7
/path2/video.avi	600.3	0	200.5
/path3/text.txt	300.34	110.4	0

(c) Catálogo de Arquivos

Figura 5.5: Catálogo de Arquivos em um ambiente de armazenamento inteligente.

A implementação atual ainda não está integrada à soluções existentes para armazenamento distribuído, tais como HDFS ou GFS. No futuro, o Catálogo de Arquivos da Sucuri poderia ser substituído, e serviços como esses poderiam ser utilizados para determinar a localização dos arquivos. No entanto, nessas soluções os arquivos são quebrados em blocos, e a Sucuri teria que ser adaptada para levar a computação aonde está a maioria dos blocos, ou aonde estão os blocos com maior probabilidade de serem acessados. Finalmente, esses sistemas também implementam bons mecanismos de redundância, dos quais a Sucuri poderia aproveitar-se.

Estimando Desempenho de Processamento com a SES

O método `getPerformance` permite que a aplicação de configuração estime o desempenho computacional de cada dispositivo no sistema, executando um conjunto de *benchmarks* sintéticos de diferentes complexidades (notação $\mathcal{O}$). Em cada dispositivo são rodadas duas aplicações para cada categoria de complexidade, com dois tamanhos de entrada diferentes para cada aplicação, e para cada configuração é calculado o tempo correspondente a $\mathcal{O}(1)$ (t_1), dividindo-se o tempo de execução (t) pela complexidade conhecida da aplicação:

$$t_1 = \frac{t}{complexidade} \tag{5.3}$$

Assim, em uma aplicação de $complexidade = \mathcal{O}(n \log n)$, por exemplo, onde n é o tamanho da entrada, substituindo em (5.3) tem-se:

$$t_1 = \frac{t}{n \log n} \tag{5.4}$$

No momento, a SES possui *benchmarks* para as complexidades de $\mathcal{O}(n)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n \log n)$ e $\mathcal{O}(n^2)$. Depois das medidas, uma *Matriz de Custo Computacional* é produzida, na qual cada linha representa um dispositivo e cada coluna representa t_1 para os algoritmos de diferentes complexidades (quatro colunas, nesse caso). Essa será a matriz utilizada posteriormente pela biblioteca Sucuri para calcular o custo de computação da aplicação do usuário em cada máquina.

5.2.4 Escalonamento Estático para Computação In-situ

Na prova de conceito apresentada na Seção 5.2.2, o escalonador original da Sucuri (discutido no início desta Seção 5.2) foi modificado para levar a computação aonde o dado está armazenado, ou seja, os nós do grafo *dataflow* que manipulam arquivos seriam alocados em processadores dos discos inteligentes que armazenam esses arquivos. Se, de acordo com o escalonador estático, o processador do disco estivesse ocupado rodando um outro nó, então a decisão seria escolher o dispositivo disponível mais próximo.

Foi mencionado também (e o Capítulo 6 traz mais detalhes) que essa era uma abordagem promissora, mas que já era esperado que necessitasse de melhorias.

Nesta fase do trabalho, foi dado um passo a frente e o desempenho foi incluído no processo de decisão do escalonamento. Discos inteligentes provavelmente têm um processador mais lento que servidores tradicionais. Por outro lado, executar uma tarefa no processador desses discos economizaria tempo de transferência de arquivos pela rede. Informações geradas pela SES são passadas para a Sucuri na forma de arquivos de configuração, para que o escalonador estático da Sucuri possa

particionar o grafo *dataflow* de maneira a maximizar o desempenho com consciência de localidade.

O escalonador irá alocar cada nó do grafo na máquina ou dispositivo que possuir menor custo total (C), baseado no custo de processamento (C_{comp}) e no custo de comunicação (C_{comm}):

$$C = C_{comp} + C_{comm} \tag{5.5}$$

C_{comm} é basicamente o tempo de transferência do arquivo (T), disponível no *Catálogo de Arquivos*, enquanto C_{comp} é baseado no t_1 para um algoritmo de mesma complexidade (determinado pela SES e disponibilizado na *Matriz de Custo Computacional*) e o tamanho do arquivo (s):

$$C_{comp} = t_1 \times s \tag{5.6}$$

O usuário deve informar a complexidade dos algoritmos associados a cada nó *dataflow* para que o escalonador possa estimar de maneira correta o tempo de execução baseado no t_1 correto calculado pela SES. Para tanto, deve chamar o método `set_complexity` no objeto `Nó` da Sucuri. Então, o escalonador construirá um *array* para cada nó n que manipula arquivo de entrada. Cada elemento do *array* será composto por C_i, que é o custo C de se rodar n no *host* i. O escalonador selecionará o *host* m com menor custo (C_{min}) para alocar n. Não sendo possível alocar em m, n será alocado no *host* de segundo menor custo.

Capítulo 6

Experimentos e Resultados

Neste capítulo são apresentados e discutidos os experimentos e resultados obtidos a partir das duas implementações de escalonador estático desenvolvidas para este trabalho: o primeiro escalonador com abordagem In-situ forçada (Seção 6.1) e o Escalonador Estático para *In-situ Computing* (Seção 6.2).

6.1 Sucuri com Abordagem In-situ Forçada

Esta seção trata do primeiro estágio deste trabalho, apresentado na Seção 5.2.2, no qual o escalonador estático da Sucuri forçava que os nós que trabalhassem com arquivos de entrada fossem alocados nas máquinas em que esses arquivos estavam hospedados.

6.1.1 Metodologia

`Ambiente`: um pequeno sistema foi montado constituído de um computador pessoal equipado com processador Intel® Pentium™ D CPU (3.00Ghz *dual core*), com 4GB de memória rodando Linux com Kernel 3.2, e uma placa Parallella [50] equipada com uma Xilinx Zynq Z7010 (ARM Cortex A9 *Dual core* + FPGA) [21], um processador RISC Epiphany de 16 núcleos, 1GB de memória e Linux Kernel 4.6. Para este trabalho, apenas o processador ARM da Parallella foi utilizado. Os dispositivos estavam conectados por Ethernet Gigabit. Na placa Parallella, um cartão SD foi utilizado como o meio de armazenamento.

`Métrica`: foram medidos os tempos que a Sucuri gastou no escalonamento dos nós e o tempo total de execução das aplicações, com o tempo de transferência dos arquivos de entrada sendo considerado parte do trabalho. Como o escalonamento é estático, é possível usar o mesmo mapeamento várias vezes a fim de se minimizar os custos do escalonador. Isso é possível em ambientes em que a latência na rede e as perdas de pacotes não variam muito, como em um *cluster* de computação de

alto desempenho. Por outro lado, em sistemas nos quais as aplicações executam de forma concorrente (*datacenters*) ou em que a infraestrutura de rede não é robusta (IoT), pode ser necessário executar o escalonador estático com uma certa frequência. Para medir o impacto do escalonamento no desempenho, foram calculados *speedups* (ganhos de desempenho) em cenários em que o escalonador era chamado após a aplicação executar n vezes. O *speedup* é dado então por:

$$S = \frac{T_{sched} + T_{tradicional} \times n}{T_{sched} + T_{in-situ} \times n}$$

T_{sched} é o tempo gasto na execução do escalonador estático. $T_{tradicional}$ é o tempo gasto no cenário em que os dados são copiados do disco (placa Parallella) para o computador normal. $T_{in-situ}$ é o tempo de execução para os casos em que a computação é delegada para a placa Parallella (*In-situ*), e n é o número de execuções rodadas antes de se chamar o escalonador novamente. São apresentados resultados para os casos em que $n = 1$, 10, 100 e ∞. Vale ressaltar que o cenário de pior caso acontece quando $n = 1$, simulando um ambiente volátil no qual deve-se escalonar toda vez que se for rodar a aplicação, e o melhor caso acontece quando $n = \infty$, válido para ambientes estáveis, caso em que o tempo de escalonamento torna-se insignificante e pode ser retirado da equação, juntamente com n.

`Aplicações`: Já que a Parallella possuía menor poder computacional que o computador normal, a solução proposta foi avaliada conduzindo-se um conjunto de experimentos com duas aplicações de custos computacionais diferentes: (*i*) *count_character* conta o número de linhas em que um determinado caractere ocorre em um arquivo de entrada, usando a busca nativa da linguagem Python; e (*ii*) *filter* encontra números em linhas de um arquivo de entrada, utilizando expressão regular, e os adiciona.

Ambas as aplicações realizaram o trabalho em pares de arquivos de mesmo tamanho, para que as buscas pudessem ser realizadas em paralelo, tirando vantagem dos núcleos dos processadores disponíveis tanto no computador, quanto na placa Parallella. Os experimentos utilizaram arquivos de entrada de 250 bytes, 12MiB e 48MiB para as duas aplicações. A *count_character* também utilizou arquivos de entrada de 476MiB, o que não ocorreu com a *filter*, pois testes preliminares não mostraram diferenças significativas nos resultados. Todos os arquivos estavam armazenados no cartão SD da Parallella, já que ela estava fazendo o papel de um disco inteligente.

Para a aplicação *filter*, foram realizados também experimentos simulando condições adversas de rede. Foram configurados cenários de latência e perda de pacotes através do `netem`, uma funcionalidade de emulação de rede disponibilizada pelo kernel do Linux [51]. Latências de 100ms, 200ms, 300ms e 400ms, com variação de 10ms utilizando uma distribuição normal, foram simuladas, bem como perdas de

pacotes de 5%, 15% e 30%. Latências dessa ordem de magnitude podem ser encontradas quando os dados estão armazenados em um servidor distante, e as perdas de pacotes são comuns em cenários em que há sensores comunicando-se por redes sem-fio, sujeitos a sofrer interferência eletromagnética. Estas características são comuns em aplicações de Internet das Coisas nas quais sensores inteligentes utilizam conexões sem-fio para enviar dados para a nuvem.

Duas variações de escalonamento foram utilizadas, a fim de realçar a validade da solução: uma em que a Sucuri escalonou os nós de leitura de arquivos na placa Parallella, alocando as tarefas para rodarem *In-situ*, e outra na qual a Sucuri foi forçada a trazer os arquivos de entrada para o computador regular, copiando os arquivos pela rede.

6.1.2 Resultados

A Figura 6.1 apresenta os resultados obtidos para a aplicação *count_character*. O eixo x mostra os valores de n utilizados nos cálculos de *speedup*, o eixo y mostra os *speedups* obtidos, e cada linha representa o tamanho dos arquivos de entrada utilizados. Importante notar que todos os casos obtiveram ganho de desempenho, atingindo *speedups* de até $2,5$ para os arquivos de 476MiB, no cenário de pior caso ($n = 1$), e $2,8$ para os arquivos de 48MiB e $n = 100$. Arquivos maiores tiveram resultados melhores, devido aos tempos de transferência maiores.

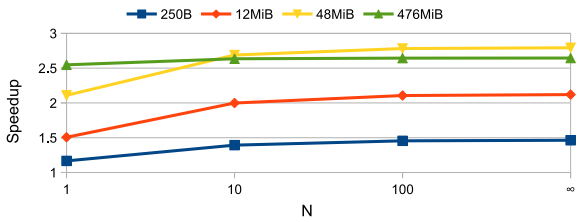


Figura 6.1: Resultados para a aplicação count_character. O eixo x mostra os valores de n utilizados, enquanto o eixo y apresenta os *speedups* obtidos. Cada linha representa os resultados para os tamanhos diferentes de arquivos de entrada utilizados (250 bytes, 12MiB, 48MiB, e 476MiB).

Considerando que os tempos de escalonamento quando as aplicações utilizam arquivos grandes de entrada já são desprezíveis comparados aos tempos de execução, foi observado praticamente nenhum ganho de desempenho quando utilizados arquivos de 476MiB e diferentes valores de n. Isso também explica porque, em $n = 100$, o *speedup* máximo de cada caso foi virtualmente alcançado.

A Tabela 6.1 mostra os tempos de execução, em segundos, para a *count_character*, usados para calcular os *speedups* apresentados na Figura 6.1. *Sched* é o tempo que a Sucuri gastou escalonando os nós, *In-situ* foi o tempo gasto realizando computação

Tabela 6.1: Tempos de execução, em segundos, para a aplicação count_character

	Sched	**250B**	**12MiB**	**48MiB**	**476MiB**
In-situ	3,0	1,7	2,5	4,9	47,8
Tradicional	3,0	2,5	5,2	13,5	126,4

In-situ e *Tradicional* foi o tempo executando a aplicação no computador normal, o que incluiu transferir os arquivos da placa Parallella. Além disso, como o processo de criação do Cátalogo de Arquivos e a construção da matriz de adjacências das distâncias relativas entre as máquinas era realizado quando a aplicação iniciava, foram considerados como parte do mecanismo de escalonamento.

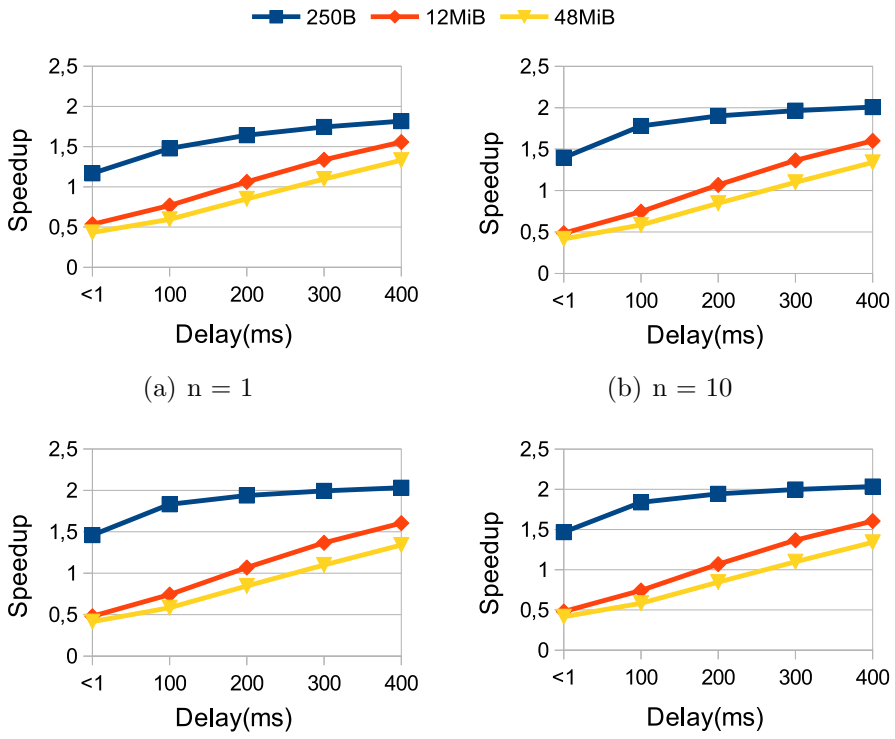
Experimentos com a *filter* obtiveram resultados diferentes. Esta aplicação tem custo computacional maior e a placa Parallella tem poder computacional menor do que o computador comum utilizado, e em condições padrão de rede apenas os cenários com os menores arquivos apresentaram ganhos de desempenho para a computação *In-situ*. Por isso, foram adicionadas artificialmente na rede, pelo *netem*, latência e perda de pacotes, para observar como a aplicação se comportaria e para simular um espectro maior de cenários existentes em ambientes reais.

A Figura 6.2 apresenta os *speedups* para as diferentes configurações de latência (100ms, 200ms, 300ms e 400ms). As curvas representam o tamanho dos arquivos de entrada, e cada gráfico mostra os resultados para um valor diferente de n (1, 10, 100 e ∞). A Tabela 6.2 exibe os tempos de execução para os experimentos com latência. T significa "Tradicional"e I "In-situ".

Diferentemente dos testes anteriores, arquivos maiores resultaram nos piores ganhos de desempenho. Isso porque a aplicação *filter* é mais pesada computacionalmente do que a *count_character*, o que significa que os custos computacionais ultrapassam os custos de comunicação para grandes arquivos. Os resultados mostraram que a abordagem *In-situ* foi vantajosa para os arquivos de 256 bytes em todos os casos. Para os arquivos maiores, entretanto, os ganhos no desempenho foram obtidos apenas para *delays* acima de 200ms. Por fim, à medida que o tamanho dos arquivos aumenta, maior é a latência necessária para se conseguir ganhos no tempo de execução para esta aplicação.

Valores maiores de n mostraram melhor desempenho, acompanhado os resultados observados para a *count_character*. Da mesma forma, $n = 100$ está próximo dos máximos *speedups*.

Testes similares foram realizados para configurações com perdas de pacotes, ao invés de *delay*. A Figura 6.3 exibe os resultados para taxas de perda de pacotes de 5%, 15% e 30%. As barras diferentes correspondem aos diferentes tamanhos de arquivos de entrada, e cada gráfico representa um valor de n. A Tabela 6.3 apresenta



(a) n = 1 (b) n = 10

(c) n = 100 (d) n = ∞

Figura 6.2: Speedups para as configurações de latência. O eixo x mostra os delays utilizados, em *ms*, enquanto o eixo y exibe os *speedups* obtidos. Cada gráfico apresenta os resultados para os diferentes valores de n usados nos cálculos.

Tabela 6.2: Tempos de execução, em segundos, para as diferentes configurações de latência (delay)

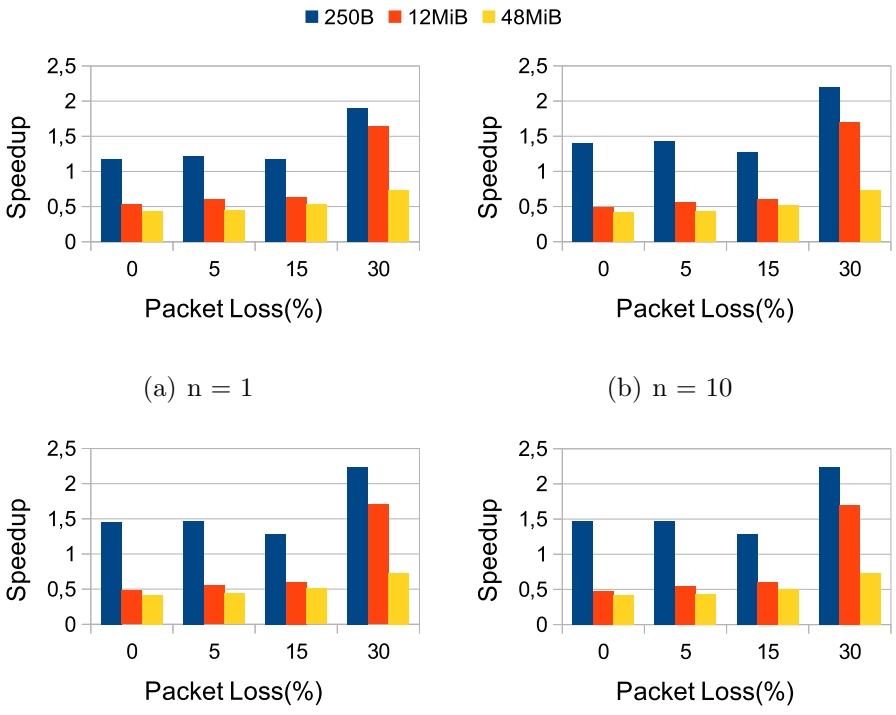
Delay(ms)	**Sched**	**250B**		**12MbiB**		**48MiB**	
		I	T	I	T	I	T
Default	3,0	1,7	2,5	25,9	12,4	97,4	40,4
100	3,0	4,0	7,3	27,9	20,7	98,7	57,6
200	3,0	6,4	12,4	30,3	32,4	101,3	85,7
300	3,0	8,8	17,6	32,7	44,7	104,0	114,3
400	3,0	11,2	22,8	35,2	56,9	106,6	142,8

Tabela 6.3: Tempos de execução, em segundos, para as diferentes configurações de perda de pacotes.

Perda(%)	**Sched**	**250B**		**12MiB**		**48MiB**	
		I	T	I	T	I	T
Default	3,0	1,7	2,5	25,9	12,4	97,4	40,4
5	3,0	2,4	3,5	26,6	14,7	97,3	42,5
15	3,0	4,9	6,2	28,1	16,9	99,7	50,0
30	3,0	7,9	17,7	32,1	54,7	104,7	76,3

os tempos de execução para os experimentos com perdas de pacotes. T representa "tradicional"e I, "In-situ".

Mais uma vez foram obtidos bons resultados para os arquivos pequenos. No entanto, para os arquivos de 12MiB, os ganhos só foram obtidos com uma taxa de perda de pacotes maior do que 20%. Para os arquivos de 48MiB, nenhum ganho foi obtido, confirmando que a perda de pacotes tem um impacto maior que a latência nas taxas de transferência dos arquivos.



(a) n = 1

(b) n = 10

(c) n = 100

(d) n = ∞

Figura 6.3: Speedups para as configurações de perda de pacotes. O eixo x mostra a taxa de perda utilizada, em %, enquanto o eixo y mostra os *speedups* obtidos. Cada gráfico mostra o resultado para os diferentes valores de n usados nos cálculos.

Além do baixo poder de processamento dos ARMs utilizados, o que removeu parte dos ganhos que poderiam ser obtidos com a localidade dos dados, é importante

notar que os experimentos cobriram apenas aplicações que leem dados de arquivos. Em um cenário de *storage*, por exemplo, no qual as aplicações também atualizam os arquivos, os tempos de transferência de volta do computador normal para a placa devem ser levados em consideração, potencialmente resultando em *speedups* maiores.

6.2 Sucuri para Computação In-situ

Os experimentos anteriores (Seção 6.1) deixaram claro como, além dos custos de comunicação, as diferenças no custo computacional tem alto impacto no desempenho das aplicações *In-situ*, podendo resultar em grande *speedup* ou *slowdown* (perda de desempenho). Esta seção trata da segunda fase deste trabalho, durante a qual o escalonador estático da Sucuri foi melhorado para levar em consideração os custos de computação e comunicação na alocação dos nós, e assim tentar mitigar os casos de *slowdown*. Portanto, nesta seção optou-se por levar em conta explicitamente as complexidades computacionais das aplicações nos experimentos.

6.2.1 Metodologia

`Ambiente`: um sistema similar ao anterior foi montado, utilizando-se novamente a placa Parallella conectada por Ethernet Gigabit a um computador comum. O computador, desta vez, possuía um processador Intel® Core™ i5-3210M CPU (2.50Ghz *quad core*), 4GB de memória e Linux com Kernel 3.10. Um cartão SD foi utilizado na Parallella, onde todos os arquivos de entrada foram armazenados, novamente simulando um ambiente de disco inteligente.

`Métrica`: foram medidos os *speedups* obtidos em várias situações. Serão apresentados, na forma de gráficos, comparações entre *speedups* forçando-se o escalonamento *In-situ* contra uma situação normal, na qual haveria sempre transferência de arquivos pela rede, tal qual foi realizado na seção anterior, bem como *speedups* considerando-se a nova solução *vs* a situação normal, a fim de se validar a tomada de decisão do escalonador relativamente a realizar ou não o processamento *In-situ*. Serão apresentados, também, gráficos das distribuições de tempos das aplicações, estendendo a análise, numa tentativa de se compreender melhor o comportamento dessas aplicações num ambiente de computação *In-situ*.

É importante realçar a interpretação dos gráficos em que são apresentados os *speedups* da solução apresentada com base na situação de escalonamento tradicional (com cópia dos arquivos dos discos inteligentes para o computador normal). A análise do gráfico deve ser interpretada da seguinte maneira: quando o *speedup* for um, significa que a Sucuri ou tomou a decisão de alocar o nó de maneira tradicional, transportando o arquivo pela rede e fazendo a computação no computador, ou

realizou a computação *In-situ*, mas a mesma resultou em tempo de execução muito próximo ao tempo gasto na maneira tradicional, e nesse caso a escolha seria indiferente com relação ao desempenho final da aplicação. Caso o *speedup* seja diferente de um, foi realizada a alocação *In-situ*. Desta forma, esses gráficos são bom indicativos da qualidade da solução apresentada e são apresentados aqui como a principal maneira de se validar o novo escalonador.

Nesses experimentos não foi utilizada a estratégia de cálculo de *speedup* de acordo com diferentes valores de n (número de execuções rodadas antes de se chamar o escalonador novamente), já que nesta fase todo o trabalho de descoberta, avaliação e consolidação dos dados do ambiente foi retirado da Sucuri, passando para um ferramenta externa (a SES), que pode rodar em *background* caso seja necessário. Como trabalho futuro está a transformação do escalonador da Sucuri em um escalonador dinâmico, que poderia receber dados atualizados da SES em tempo real para serem utilizados em suas decisões.

`Aplicações`: como a placa Parallella continuava com desempenho inferior ao computador comum (diferença ainda maior, nesse caso), a solução proposta foi avaliada através de um conjunto de experimentos contendo diferentes algoritmos de complexidades bem conhecidas:

- **Search**: executa uma busca sequencial em um arquivo de texto, contando as ocorrências de um certo caractere. A complexidade da Search é $\mathcal{O}(n)$.

- **Search 2x**: realiza a mesma operação da **Search** com dois arquivos de tamanhos iguais, com o intuito de aproveitar os núcleos disponíveis nos processadores tanto da Parallella, quanto do computador comum. A Search 2x também tem complexidade $\mathcal{O}(n)$.

- **Filter**: encontra números em linhas de um arquivo de entrada, utilizando expressão regular, e os soma. Cada linha dos arquivos de entrada possui 24 caracteres. Considerando que uma expressão regular de busca em uma *string* com m caracteres tem complexidade $\mathcal{O}(m)$, e que essa operação foi realizada em uma lista de n *strings*, a complexidade da Filter seria $\mathcal{O}(n \times m)$. No entanto, já que $n \gg m$, foi assumida a complexidade de $\mathcal{O}(n)$ para essa aplicação. Importante ressaltar que a Filter possui uma constante (tempo de $\mathcal{O}(1)$) consideravelmente maior que a Search, tornando-a um bom caso para se avaliar o mecanismo de estimativa de tempo de computação da solução.

- **HeapSort**: ordena um arquivo binário contendo inteiros de 8 bits utilizando o algoritmo Heapsort (complexidade $\mathcal{O}(n \log n)$).

- **MergeSort**: ordena um arquivo binário contendo inteiros de 8 bits utilizando o algoritmo Mergesort (complexidade $\mathcal{O}(n \log n)$).

- **SelectionSort**: ordena um arquivo binário contendo inteiros de 8 bits utilizando o algoritmo Selectionsort (complexidade $\mathcal{O}(n^2)$).
- **InsertionSort**: ordena um arquivo binário contendo inteiros de 8 bits utilizando o algoritmo Insertionsort (complexidade $\mathcal{O}(n^2)$).

Todos os *benchmarks* foram implementados em Python, sem preocupação quanto à otimizações para se evitar os custos da execução interpretada dessa linguagem, como o uso de funções em C pré-compiladas como bibliotecas dinâmicas que poderiam ser invocadas do código Python. Como o tamanho dos arquivos de entrada é característica importante neste estudo, todos os experimentos foram realizados com diferentes tamanhos de arquivos. Para Search e Search 2x, foram utilizados arquivos de 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, e 476MiB. A Filter foi executada com arquivos de 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB e 48MiB. HeapSort e MergeSort, 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB e 16MiB. Por fim, SelectionSort e InsertionSort tiveram arquivos de tamanho 256 bytes, 4KiB, 8KiB, 48KiB e 256KiB como entrada. O tamanho máximo dos arquivos escolhidos para cada aplicação foi escolhido de maneira que elas pudessem ser estressadas no contexto de computação *In-situ*.

A exemplo do estágio anterior, foram simuladas através do `netem` algumas condições adversas de rede com o intuito de observar o impacto de delay e perda de pacotes no desempenho das aplicações. Foram simulados delays de 100ms, 200ms e 300ms e perdas de pacotes de 5% e 15%.

6.2.2 Resultados

No primeiro conjunto de experimentos desta fase, a execução *In-situ* foi novamente forçada, ou seja, todos os nós da Sucuri que manipularam arquivos de entrada foram executados nos núcleos do processador ARM da placa Parallella. Então, o novo escalonador proposto neste trabalho foi ativado para se avaliar se ele foi capaz de prevenir que a Sucuri delegasse a computação à Parallella nos casos em que essa abordagem não trouxesse vantagens, evitando-se perdas no desempenho.

A Figura 6.4 mostra os resultados para as aplicações Search e Search 2x. O eixo y exibe os *speedups* baseados nos tempos de execução no computador regular, caso em que os dados são sempre copiados da placa Parallella. Nota-se que habilitar o escalonamento *In-situ* trouxe ganhos de desempenho independentemente dos tamanhos dos arquivos, portanto os resultados do escalonamento *In-situ* forçado foram os mesmos obtidos deixando-se a decisão a cargo do novo escalonador.

A Figura 6.5 apresenta os resultados para a aplicação Filter. O eixo y mostra os *speedups* com relação ao tempo de execução no computador normal, quando os dados

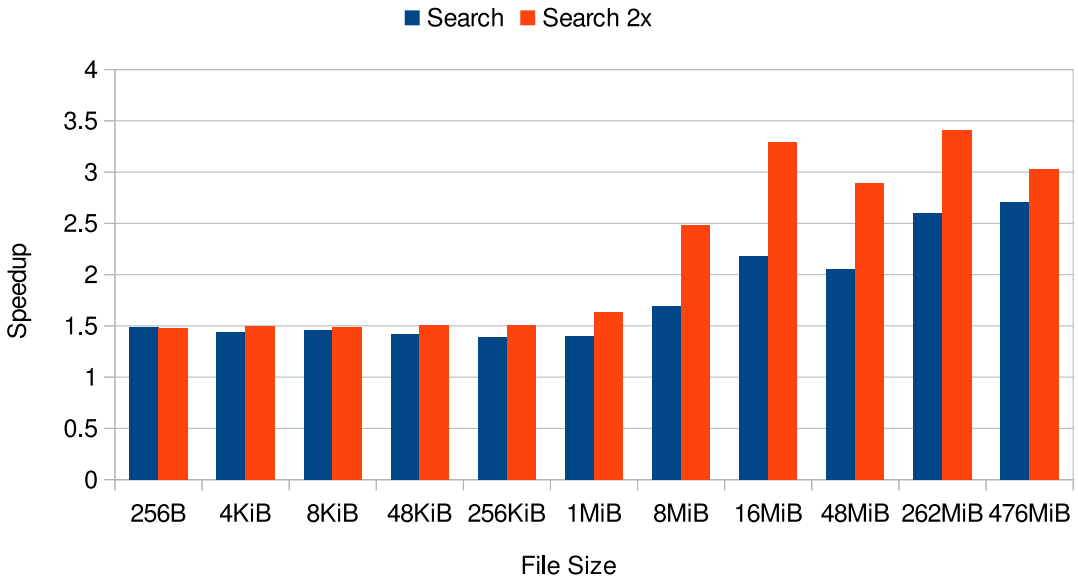


Figura 6.4: Speedups para o algoritmo Search utilizando o escalonador proposto para um e dois (Search 2x) arquivos de entrada. O eixo x mostra os tamanhos dos arquivos, enquanto eixo y apresenta o *speedup* em relação a um cenário não-in-situ (original). Cada barra provê os resultados para os arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, e 476MiB).

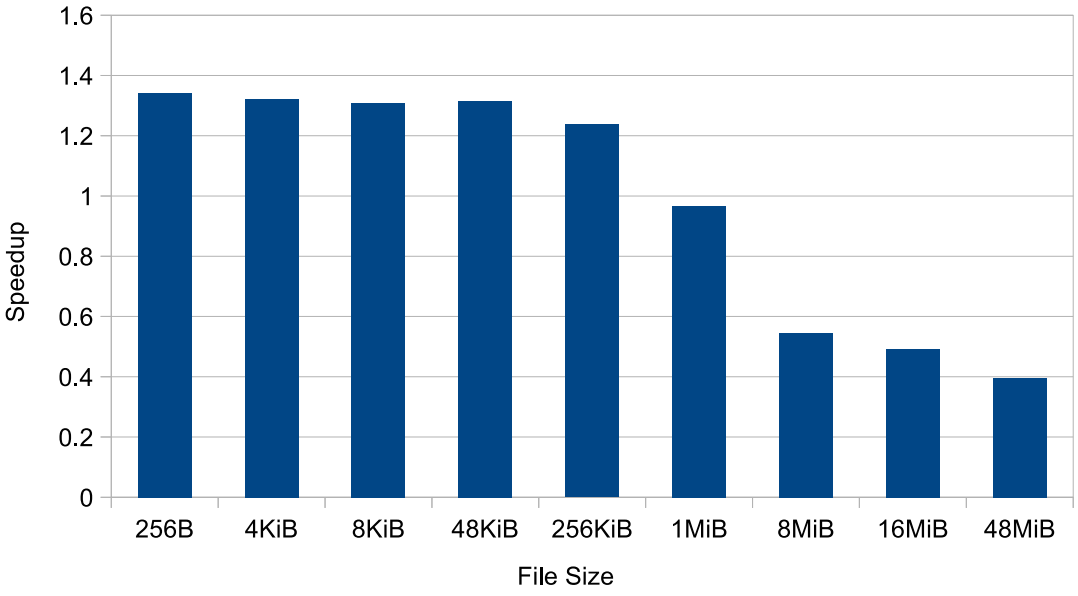
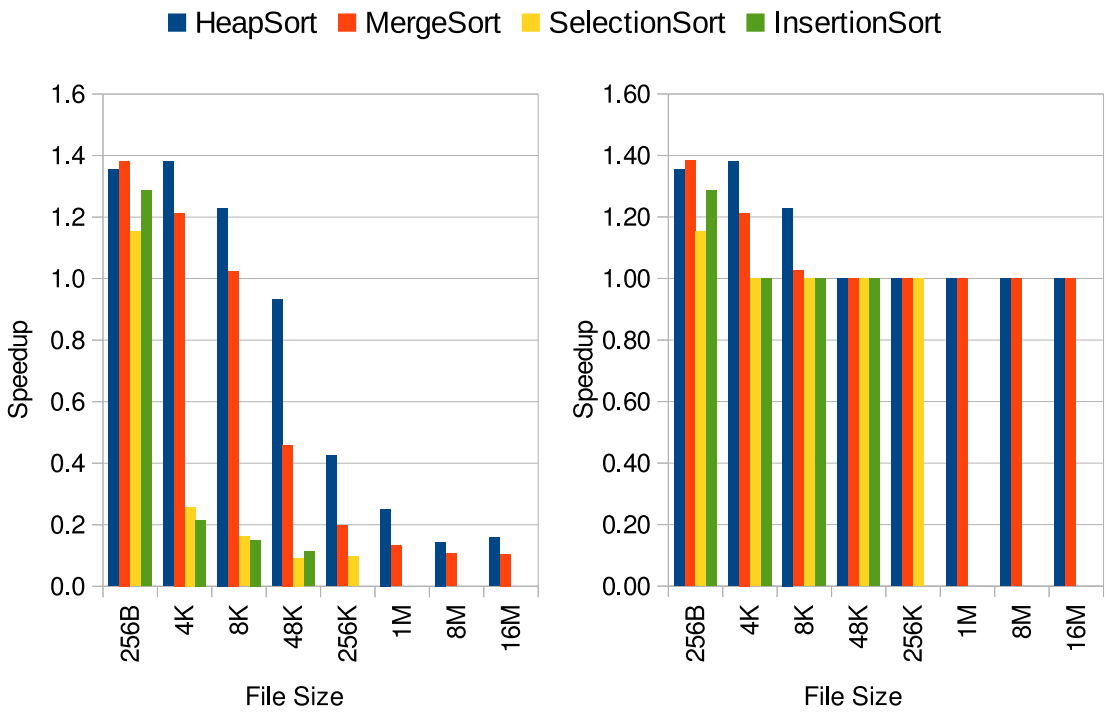


Figura 6.5: Speedups para a aplicação Filter utilizando o escalonador proposto. O eixo x mostra os tamanhos dos arquivos de entrada, enquanto o eixo y representa o *speedup* sobre o cenário original. Cada barra indica os resultados para os arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB e 48MiB).

são sempre copiados da Parallella pela rede. Nota-se que, a exemplo das aplicações anteriores, o escalonador da Sucuri decidiu por sempre realizar computação *In-situ*, o que não se mostrou a decisão correta para arquivos de entrada maiores que 1MiB. Apesar da complexidade da Filter também ser $\mathcal{O}(n)$, o tempo base (t_1), ou seja, a constante da complexidade, é bem maior que aqueles das aplicações de $\mathcal{O}(n)$ utilizadas na SES e nas aplicações Search, o que sugere que o escalonador poderia aceitar tal constante como parâmetro de entrada, a fim de melhorar suas estimativas. Essa melhoria de se incluir as constantes nas fórmulas de complexidade é objetivo de trabalho futuro.



(a) In-situ Forçado

(b) In-situ Escalonado

Figura 6.6: Speedups para os diferentes algoritmos de ordenação quando *(a)* é forçada a computação *In-situ* e *(b)* é utilizado o escalonador proposto. O eixo x mostra os tamanhos dos arquivos de entrada, enquanto o eixo y apresenta os *speedups* em relação ao cenário tradicional. Cada barra provê os resultados para os tamanhos de arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB e 16MiB).

Os resultados para as aplicações de ordenação são apresentados na Figura 6.6. O eixo y revela os ganhos de desempenho com base na execução da aplicação de maneira tradicional, na qual os dados são sempre copiados da placa Parallella para o computador. Na Figura 6.6(a), a execução *In-situ* foi forçada, e na Figura 6.6(b) essa decisão foi deixada a cargo do escalonador proposto. Nota-se que, nesse caso, o processamento *In-situ* foi vantajoso apenas para os arquivos menores, e aplicações de complexidade computacional maior mostraram os piores resultados — como a

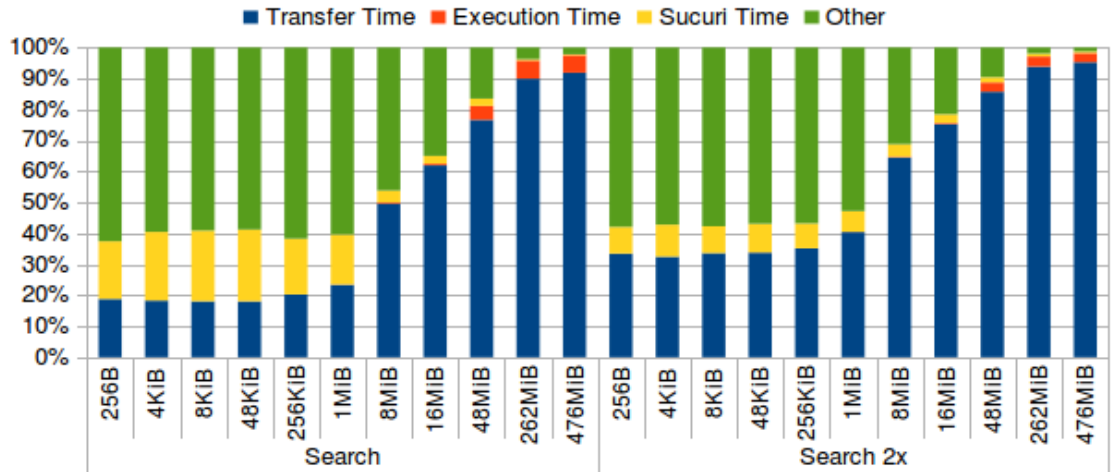
placa Parallella possuía um processador muito mais lento, em relação ao computador comum, esses resultados já eram esperados. Mesmo assim, o escalonador proposto neste trabalho foi capaz de tomar as decisões corretas e prevenir as perdas no desempenho, não permitindo a computação *In-situ* para os arquivos maiores, já que pagar o preço de transferir os arquivos pela rede nesse caso resultaria em desempenho melhor do que rodar a aplicação em um processador menos poderoso. O processamento *In-situ* de HeapSort e MergeSort produziu *speedups*, de até $1,4$, para arquivos até 8KiB. As aplicações SelectionSort e InsertionSort obtiveram ganhos de desempenho *In-situ* apenas para os arquivos de 256 bytes.

Para avaliar os potenciais ganhos de desempenho nos *benchmarks* apresentados, foram medidas também as porcentagens de tempo gasto em: transferência de arquivos; execução da aplicação (algoritmo); sobrecargas (*overheads*) da Sucuri, tais como inicialização, comunicação entre os *hosts* por MPI e encerramento; e outros *overheads*, incluindo aí *overheads* de Sistema Operacional e inicialização do Python e MPI, por exemplo. Isso foi feito tanto para o cenário tradicional (com processamento *In-situ* desabilitado), quanto para as execuções guiadas pelo escalonador proposto. As Figuras 6.7, 6.8 e 6.9 apresentam os resultados para as aplicações Search, Filter e de ordenação, respectivamente.

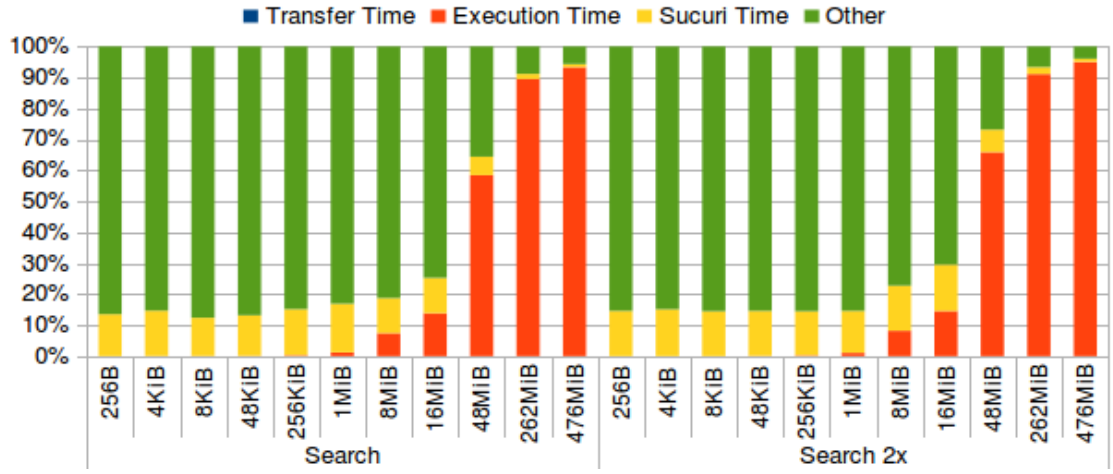
Nota-se na Figura 6.7 que os *overheads* (tanto da Sucuri, quanto outros) foram predominantes para os arquivos pequenos, já que o tempo total, nesses casos, é bem curto (da ordem de 2s). Para arquivos maiores, os tempos de transferência dos arquivos dominaram os tempos de execução para as configurações não-in-situ (Figura 6.7(a)). Por outro lado, os tempos de transferência foram completamente eliminados em todos os cenários na Figura 6.7(b), já que o escalonador optou por sempre habilitar o processamento *In-situ*. Na realidade, ter os tempos de transferência como um aspecto dominante nessa aplicação é exatamente o que a torna uma boa candidata para computação *In-situ*, já que o baixo desempenho dos processadores do disco inteligente não se torna um grande obstáculo.

Na Figura 6.8, os *overheads* também predominaram para os arquivos pequenos, já que novamente o tempo total de execução nesses casos foi baixo (da ordem de 2s). No entanto, para arquivos maiores, os tempos de transferência e execução do algoritmo mostraram-se bem balanceados, sugerindo que poderiam haver ganhos no processamento *In-situ* (Figura 6.8(a)). Os *slowdowns* aconteceram pois a placa Parallella possuía um poder de processamento bem menor do que o computador comum utilizado nos experimentos.

A Figura 6.9, relativa às distribuições de tempo para os algoritmos de ordenação, mostra um cenário diferente. Os *overheads*, tanto da Sucuri, quanto os outros, dominaram apenas para arquivos muito pequenos, especialmente para as aplicações SelectionSort e InsertionSort, de complexidade maior. Nesse caso, processadores

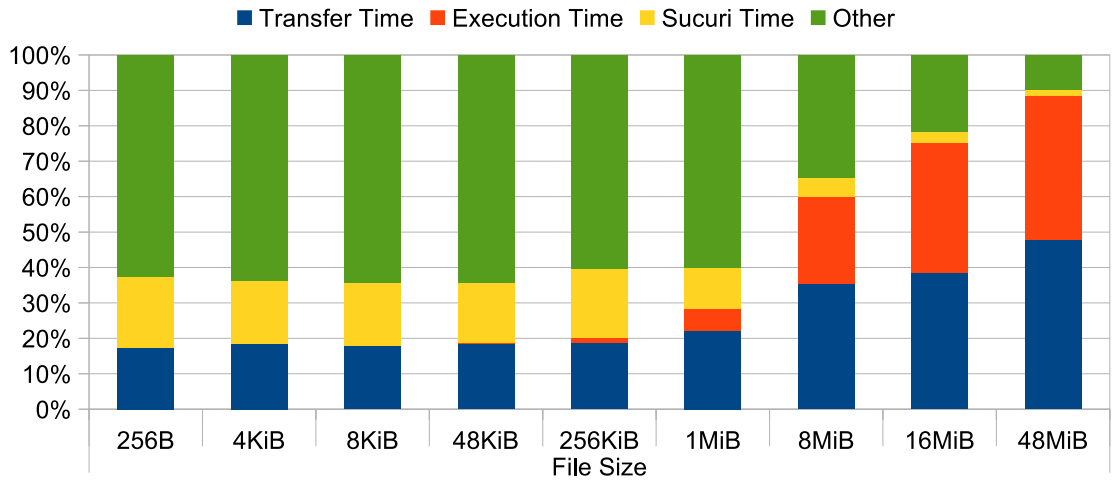


(a) Original

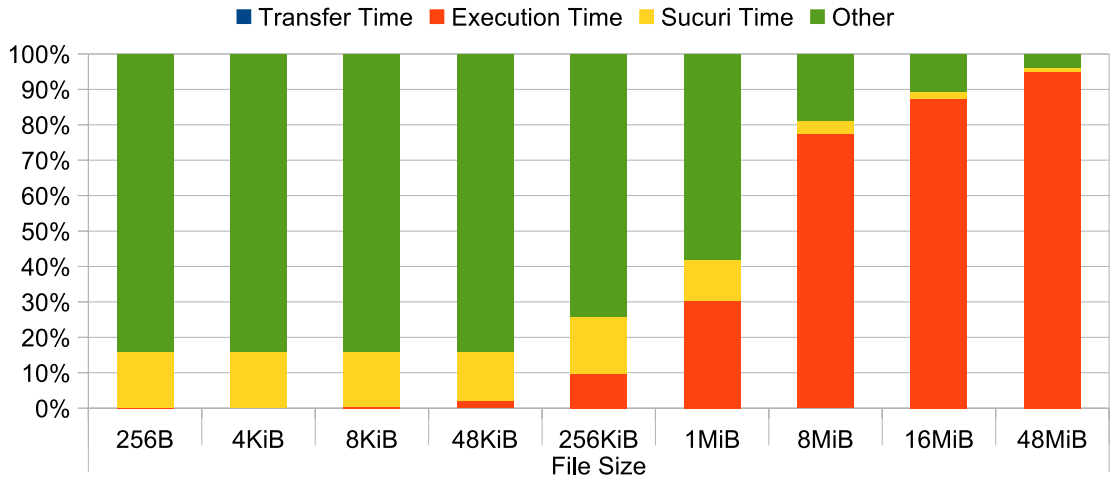


(b) In-situ (escalonado)

Figura 6.7: Distribuição de tempos para a aplicação Search com um e dois arquivos de entrada (2x). Os resultados são apresentados tanto para o cenário original, quanto para o processamento *In-situ* decidido pelo escalonador. O eixo x mostra os tamanhos dos arquivos, enquanto o eixo y apresenta a porcentagem de tempo gasta na transferência de arquivos, execução do algoritmo, Sucuri *overhead*, e outros *overheads* (tais como os custos do Python e de inicialização do MPI). Cada barra disponibiliza os resultados para os tamanhos dos arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, e 476MiB).

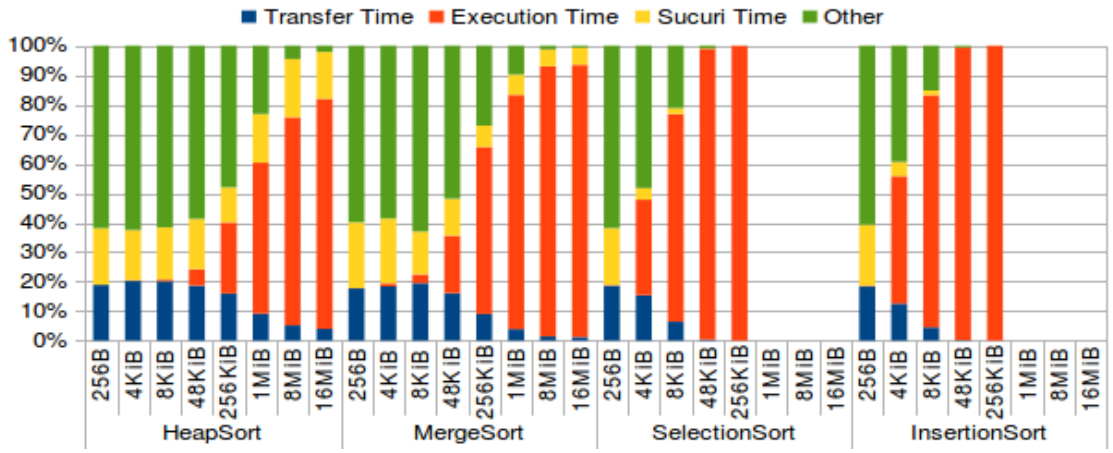


(a) Original

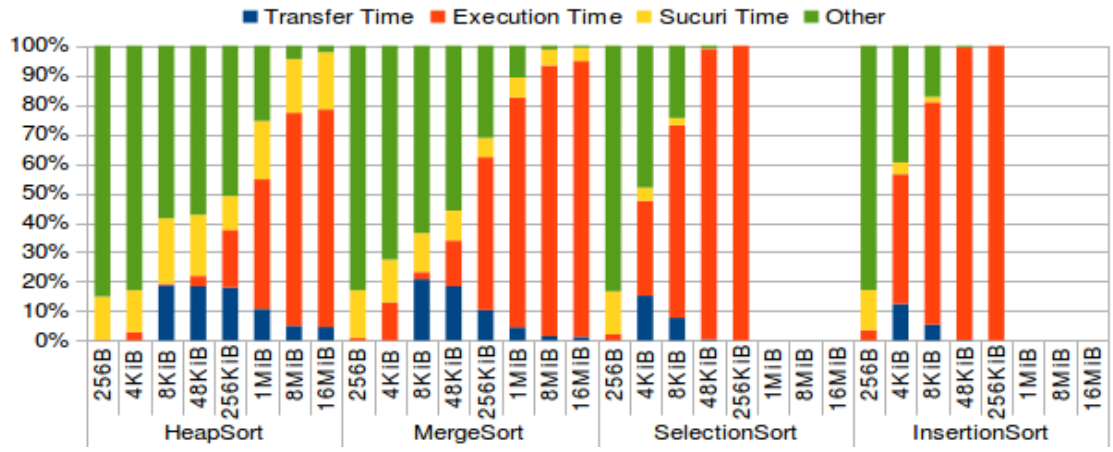


(b) In-situ (escalonado)

Figura 6.8: Distribuição de tempos para a aplicação Filter. Os resultados são apresentados tanto para o cenário original, quanto para o processamento *In-situ* decidido pelo escalonador. O eixo x mostra os tamanhos dos arquivos, enquanto o eixo y apresenta a porcentagem de tempo gasta na transferência de arquivos, execução do algoritmo, Sucuri *overhead*, e outros *overheads* (tais como os custos do Python e de inicialização do MPI). Cada barra disponibiliza os resultados para os tamanhos dos arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB and 48MiB).



(a) Original



(b) In-situ (escalonado)

Figura 6.9: Distribuição de tempos para os diferentes algoritmos de ordenação, nos cenários original e *In-situ* escalonado. O eixo x mostra os tamanhos dos arquivos, enquanto o eixo y apresenta a porcentagem de tempo gasta na transferência de arquivos, execução do algoritmo, Sucuri *overhead*, e outros *overheads* (tais como os custos do Python e de inicialização do MPI). Cada barra disponibiliza os resultados para os tamanhos dos arquivos de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB), agrupados por algoritmo.

In-situ mais poderosos seriam necessários para se atingir ganhos de desempenho ao se evitar tempos de transferência relativamente tão pequenos.

Para avaliar a solução no contexto de aplicações *Edge/Fog*, no qual a rede de comunicação pode apresentar latência e perda de pacotes maiores que em redes locais comuns, foram realizados experimentos variando-se esses parâmetros, conforme mencionado e a exemplo do que foi feito na prova de conceito. As aplicações Search e Search 2x não foram incluídas nesses testes por já terem apresentado bons resultados, mesmo em configurações normais de rede local.

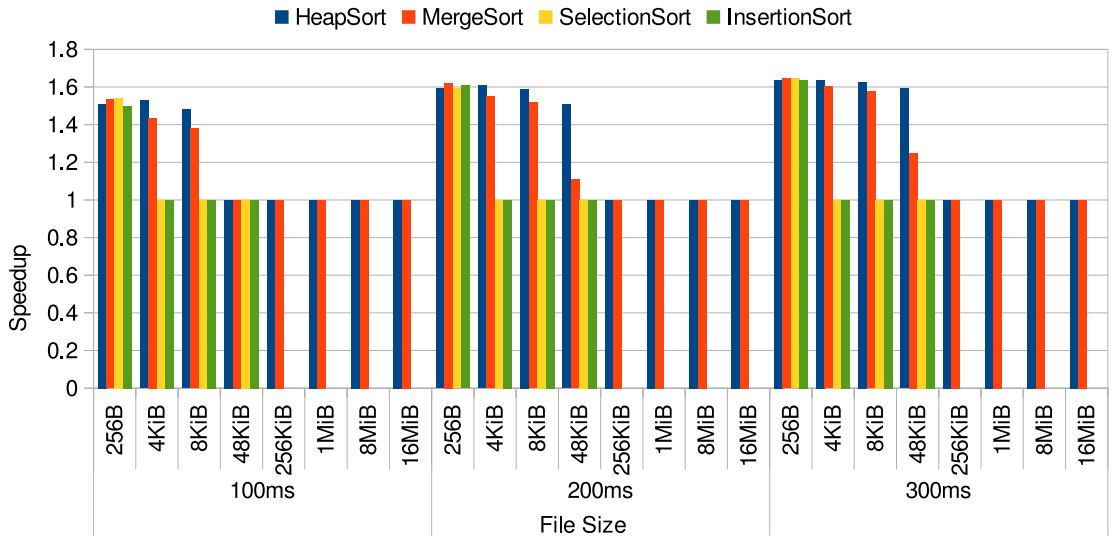


Figura 6.10: Speedups para os diferentes algoritmos de ordenação simulando-se diferentes latências de rede e utilizando-se o escalonador proposto. O eixo x mostra os tamanhos de arquivo e os valores de latência, e o eixo y apresenta os speedups com base em um cenário não-in-situ (original). Cada barra apresenta resultados para os tamanhos de arquivo de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB e 16MiB).

A Figura 6.10 apresenta os resultados para os experimentos com latência, utilizando o escalonador da Sucuri. À medida que a latência aumenta, o processamento *In-situ* torna-se mais vantajoso para arquivos maiores, especialmente para as aplicações HeapSort e MergeSort, que alcançaram ganhos de desempenho para arquivos até 48KiB. SelectionSort e InsertionSort, no entanto, não apresentaram ganhos significativos. Além disso, pode-se notar uma tendência de limite superior para o *speedup* por volta de 1,6, possivelmente porque a as aplicações foram capazes de saturar o núcleo do processador ARM da placa Parallella, mesmo para arquivos de tamanho pequeno. Mais importante, o escalonador proposto novamente foi capaz de evitar as perdas de desempenho nos casos em que o processamento *In-situ* fosse prejudicial ao tempo de execução das aplicações.

Os resultados para os experimentos com perdas de pacotes, utilizando o escalo-

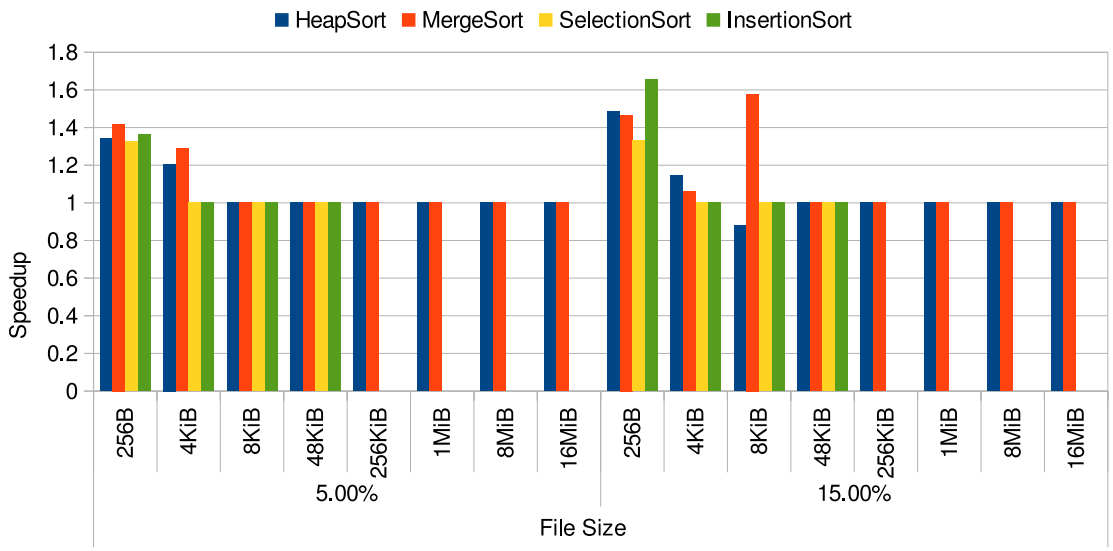


Figura 6.11: Speedups para os diferentes algoritmos de ordenação simulando-se diferentes taxas de perdas de pacote e utilizando-se o escalonador proposto. O eixo x mostra os tamanhos de arquivo e as taxas de perda de pacote, e o eixo y apresenta os speedups com base em um cenário não-in-situ (original). Cada barra apresenta resultados para os tamanhos de arquivo de entrada utilizados (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB).

nador da Sucuri, são apresentados na figura 6.11. Nota-se também aqui, como nos experimentos com latência, que a execução *In-situ* tende a ser mais vantajosa para arquivos maiores, à medida que a taxa de perda de pacotes aumenta. No entanto, como a perda de pacotes também afeta negativamente os pacotes do próprio ambiente de execução da Sucuri, e tende a deixar a comunicação pela rede mais instável, foi observado grande desvio padrão para a taxa de 15% de perda de pacotes. Mesmo assim, a solução proposta mais uma vez pôde evitar *slowdown* nos casos em que a computação *In-situ* fosse resultar em perda de desempenho.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste trabalho, a Sucuri, uma biblioteca de programação *dataflow* em Python, foi transformada de maneira a torná-la capaz de realizar processamento *In-situ* em ambientes heterogêneos. A biblioteca já era versátil o suficiente para permitir a execução transparente em *clusters* de *multicores*, e agora a Sucuri também trabalha com localidade de dados de maneira direta. O escalonador da Sucuri foi modificado para levar em consideração os tempos de transferência de arquivos pela rede e os tempos de processamento nos diferentes dispositivos do ambiente para decidir se deve delegar a execução dos nós do grafo *dataflow* da aplicação aos discos inteligentes que armazenam os dados.

O trabalho foi realizado em dois estágios. Em um primeiro momento, foi criada uma prova de conceito na qual o escalonador estático da Sucuri foi modificado para sempre priorizar a localidade dos dados, e assim, em um ambiente de disco inteligente, sempre optar por processamento *In-situ*. Essa prova de conceito teve o intuito de demonstrar a viabilidade e os possíveis ganhos que uma solução baseada em localidade de dados traria para a Sucuri. Mais tarde, uma solução completa foi apresentada, na qual a Sucuri, além da localidade, levava em consideração os custos de comunicação e processamento de cada dispositivo do ambiente, para então decidir se deveria ou não realizar computação *In-situ*.

Para a prova de conceito, foram realizados experimentos forçando-se a computação *In-situ* e calculando o *speedup* desses resultados, tomando como base o caso tradicional, em que o processamento é realizado em servidores comuns, depois de transferir o arquivo pela rede. Duas aplicações artificiais de processamento de texto serviram de *benchmark* para esse caso, e foram executadas em diferentes configurações de tamanho de arquivo de entrada, latência de rede e perdas de pacotes.

Os experimentos mostraram ganhos de desempenho para a aplicação de custo computacional menor. Na de custo computacional maior, os ganhos de desempenho foram limitados aos arquivos de tamanho menor, sugerindo que o custo computacional de se processar *In-situ*, em um processador de capacidade bem menor, não

compensou os custos de comunicação para esses casos. Além disso, melhores ganhos foram obtidos com maiores taxas de perda de pacotes e latência. Por fim, os resultados expuseram as limitações do equipamento utilizado como disco inteligente.

Para a solução final, o escalonador foi novamente modificado para que pudesse levar os custos de comunicação e processamento em consideração durante a alocação dos nós. Uma ferramenta para estimar o poder de processamento, *bandwidth*, latência e tempos de transferência de arquivos em diferentes máquinas, a SES, foi desenvolvida para auxiliar a Sucuri em sua decisões relativas à computação *In-situ*. A SES é responsável por construir o Catálogo de Arquivos e a Matriz de Custo Computacional, disponibilizando-os para a Sucuri, para serem utilizados pelo novo escalonador proposto.

Experimentos realizados com um conjunto de *benchmarks* de diferentes complexidades mostraram resultados semelhantes aos anteriores, do ponto de vista do tamanho dos arquivos de entrada, latência de rede e perdas de pacotes. Verificou-se também que as aplicações de complexidade menor mostraram maiores *speedups*, no geral, e aplicações de complexidade maior mostraram ganhos limitados de desempenho, sendo necessários processadores mais poderosos nos dispositivos inteligentes para que o processamento *In-situ* seja vantajoso.

O sucesso da solução foi verificado nos experimentos, pois, na grande maioria dos testes, o novo escalonador foi capaz de tomar a melhor decisão com relação ao desempenho da aplicação avaliada: alocou os nós *In-situ*, quando tal alocação fosse resultar em ganhos no tempo de execução, mas optou por transferir os dados para o computador comum, quando a computação *In-situ* fosse resultar em perda de *performance*. Isso demonstra a capacidade da Sucuri em orquestrar paralelismo através de *dataflow* em *multicores*, *clusters* e agora ambientes *In-situ* de maneira transparente.

Esta obra abre um conjunto de possíveis trabalhos futuros, alguns já mencionados durante o texto:

- Realizar experimentos com *benchmarks* diferentes, utilizando-se, por exemplo, aplicações reais.

- Empregar equipamentos diferentes, utilizando dispositivos inteligentes como discos, roteadores ou *switches*, avaliando o trabalho nos contextos de *Edge* e *Fog Computing*. Explorar na totalidade os recursos da Parallella, incluindo a FPGA e os núcleos Epiphany, a fim de aumentar seu desempenho.

- Realizar experimentos para medição do consumo de energia dos dispositivos, já que essa é uma área na qual espera-se um bom resultado em favor dos equipamentos de *Edge/Fog/In-situ*.

- Analisar a viabilidade de se incluir a descoberta da complexidade dos algoritmos utilizados pelas aplicações diretamente pela Sucuri, sem que o usuário precise informá-las.

- Incluir na SES e no escalonamento estimativas de tempo de computação que levem em conta também as constantes nas equações de complexidade, pois, como vimos, aplicações em que as constantes se afastam do caso base nem sempre apresentaram o melhor resultado.

- Realizar experimentos com conjuntos de dados bem maiores, bem como aplicações que gerem arquivos de saída, com o intuito de se realizar testes de estresse na biblioteca para que se possa encontrar eventuais gargalos a serem corrigidos.

Já está sendo trabalhada a implementação de um mecanismo de escalonamento dinâmico para a Sucuri no contexto de computação *In-situ*. O escalonador poderia invocar a SES para dinamicamente atualizar os dados da configuração do ambiente enquanto toma suas decisões de alocação, o que trará robustez a solução para o caso de ambientes dinâmicos em que as condições de rede tendem a variar muito.

Por fim, intenciona-se integrar a Sucuri com algum sistema de arquivos distribuído inspirado no GFS, como o Hadoop Distributed File System (HDFS), de código aberto, já que esses são sistemas de arquivos maduros e especializados para se explorar a localidade dos dados em arquivos muito grandes, além de oferecerem redundância. A Sucuri poderia se beneficiar dessas características e seus desenvolvedores poderiam concentrar seus esforços nos aspectos da biblioteca mais diretamente ligados a programação *dataflow*.

Referências Bibliográficas

[1] CAULFIELD, A., CHUNG, E., PUTNAM, A., et al. "A Cloud-Scale Acceleration Architecture". In: *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016. Disponível em: <https://www.microsoft.com/en-us/research/publication/configurable-cloud-acceleration/>.

[2] OPENFOG CONSORTIUM ARCHITECTURE WORKING GROUP. "OpenFog Architecture Overview". Feb 2016. Disponível em: <http://www.openfogconsortium.org/wp-content/uploads/OpenFog-Architecture-Overview-WP-2-2016.pdf>.

[3] BONOMI, F., MILITO, R., ZHU, J., et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*, pp. 13–16, New York, USA, 2012. ACM Press. ISBN: 9781450315197. doi: 10.1145/2342509.2342513. Disponível em: <http://delivery-acm-org.ez29.capes.proxy.ufrj.br/10.1145/2350000/2342513/p13-bonomi.pdf>.

[4] VAQUERO, L. M., RODERO-MERINO, L. "Finding your Way in the Fog", *ACM SIGCOMM Computer Communication Review*, v. 44, n. 5, pp. 27–32, oct 2014. ISSN: 01464833. doi: 10.1145/2677046.2677052. Disponível em: <http://delivery-acm-org.ez29.capes.proxy.ufrj.br/10.1145/2680000/2677052/p27-vaquero.pdf>.

[5] GIANG, N. K., BLACKSTOCK, M., LEA, R., et al. "Developing IoT applications in the Fog: A Distributed Dataflow approach". In: *2015 5th International Conference on the Internet of Things (IOT)*, pp. 155–162. IEEE, oct 2015. ISBN: 978-1-4673-8056-0. doi: 10.1109/IOT.2015.7356560. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7356560>.

[6] SHI, W., CAO, J., ZHANG, Q., et al. "Edge Computing: Vision and Challenges", *IEEE Internet of Things Journal*, v. 3, n. 5, pp. 637–646, Oct 2016. ISSN: 2327-4662. doi: 10.1109/JIOT.2016.2579198.

[7] JUN, S. W., LIU, M., LEE, S., et al. "BlueDBM: An appliance for Big Data analytics". In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, June 2015. doi: 10.1145/2749469.2750412.

[8] KIM, J., ABBASI, H., CHACÓN, L., et al. "Parallel in situ indexing for data-intensive computing". In: *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 65–72, Oct 2011. doi: 10.1109/LDAV.2011.6092319.

[9] "NGD Systems announces availability of industry's first Computational Storage". Accessed: Sep 7, 2017. Disponível em: <http://www.prnewswire.com/news-releases/ngd-systems-announces-availability-of-industrys-first-computational-storage-300493319.html>.

[10] "Juniper Networks Advances Application Performance and Acceleration with New Compact Compute-Integrated Network Switch". Disponível em: <https://www.maxeler.com/juniper-switch/>. last accessed on September 1, 2017.

[11] ALVES, T. A., MARZULO, L. A., FRANCA, F. M., et al. "Trebuchet: exploring TLP with dataflow virtualisation", *International Journal of High Performance Systems Architecture*, v. 3, n. 2/3, pp. 137, 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466.

[12] MARZULO, L. A., ALVES, T. A., FRANÇA, F. M., et al. "Couillard: Parallel programming via coarse-grained Data-flow Compilation", *Parallel Computing*, v. 40, n. 10, pp. 661 – 680, 2014. ISSN: 0167-8191.

[13] TBB FlowGraph. "TBB FlowGraph". accessed on August 8, 2014. Disponível em: <http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm>.

[14] WOZNIAK, J., ARMSTRONG, T., WILDE, M., et al. "Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing". In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 95–102, May 2013.

[15] WILDE, M., HATEGAN, M., WOZNIAK, J. M., et al. "Swift: A language for distributed parallel scripting", *Parallel Computing*, v. 37, n. 9, pp. 633 – 652, 2011. ISSN: 0167-8191.

[16] MATHEOU, G., EVRIPIDOU, P. "FREDDO: an efficient framework for runtime execution of data-driven objects". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 265–273, Las Vegas, 2016. ISBN: 1601324448.

[17] ALVES, T., GOLDSTEIN, B. F., FRANÇA, F. M., et al. "A Minimalistic Dataflow Programming Library for Python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*. IEEE, 2014.

[18] SENA, A. C., VAZ, E. S., FRANÇA, F. M. G., et al. "Graph Templates for Dataflow Programming". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pp. 91–96, Oct 2015. doi: 10.1109/SBAC-PADW.2015.20.

[19] SILVA, R. J., GOLDSTEIN, B., SANTIAGO, L., et al. "Task Scheduling in Sucuri Dataflow Library". In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, v. 1, pp. 37–42. IEEE, oct 2016. ISBN: 978-1-5090-4844-1. doi: 10.1109/SBAC-PADW.2016.15. Disponível em: <http://ieeexplore.ieee.org/document/7803693/>.

[20] PARALLELLA. *Parallella-1.x Reference Manual*. Parallella, 2014. Disponível em: <http://www.parallella.org/docs/parallella_manual.pdf>. last accessed on February 1, 2018.

[21] "Zynq-7000 All-Programmable Technical Reference Manual". 2017. Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf>. Accessed: Sep 7, 2017.

[22] CARVALHO, C. B., FERREIRA, V. C., FRANCA, F. M., et al. "Towards a Dataflow Runtime Environment for Edge, Fog and In-Situ Computing". In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 115–120. IEEE, oct 2017. ISBN: 978-1-5386-4819-3. doi: 10.1109/SBAC-PADW.2017.28. Disponível em: <http://ieeexplore.ieee.org/document/8109016/>.

[23] JALAPARTI, V., BODIK, P., MENACHE, I., et al. "Network-Aware Scheduling for Data-Parallel Jobs", *ACM SIGCOMM Computer Communication Review*, v. 45, n. 5, pp. 407–420, aug 2015. ISSN: 01464833. doi: 10.1145/2829988.2787488. Disponível em: <http://dl.acm.org/citation.cfm?doid=2829988.2787488>.

[24] DURO, F. R., BLAS, J. G., ISAILA, F., et al. "Exploiting data locality in Swift / T workflows using Hercules", *Nesus Workshop*, v. I, n. 1, pp. 71–76, 2014. Disponível em: <http://www.mcs.anl.gov/~wozniak/papers/Hercules_2014.pdf>.

[25] FITZPATRICK, B. "Distributed Caching with Memcached", *Linux J.*, v. 2004, n. 124, pp. 5–, ago. 2004. ISSN: 1075-3583. Disponível em: <http://dl.acm.org/citation.cfm?id=1012889.1012894>.

[26] BUONO, D., DANELUTTO, M., DE MATTEIS, T., et al. "A lightweight run-time support for fast dense linear algebra on multi-core". In: *Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014). IASTED, ACTA press*, 2014.

[27] GHEMAWAT, S., GOBIOFF, H., LEUNG, S.-T. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, p. 29, New York, New York, USA, 2003. ACM Press. ISBN: 1581137575. doi: 10.1145/945445.945450. Disponível em: <http://portal.acm.org/citation.cfm?doid=945445.945450>.

[28] SHVACHKO, K., KUANG, H., RADIA, S., et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pp. 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. Disponível em: <http://dx.doi.org/10.1109/MSST.2010.5496972>.

[29] THE APACHE SOFTWARE FOUNDATION. "HDFS Users Guide". Accessed: 29-Aug-2017. Disponível em: <http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>.

[30] MACHADO, F. B., MAIA, L. P. *Arquitetura de Sistemas Operacionais*. 4th ed. Rio de Janeiro, RJ, Livros Técnicos e Científicos Editora S.A., 2007.

[31] FERREIRA, C. E., FERNANDES, C., MIYAZAWA, F., et al. "Uma introdução sucinta a algoritmos de aproximação", *Colóquio Brasileiro de Matemática-IMPA, Rio de Janeiro-RJ*, 2001.

[32] GAREY, M. R., JOHNSON, D. S. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories, 1979.

[33] GRAHAM, R. L. "Bounds for certain multiprocessing anomalies", *Bell Labs Technical Journal*, v. 45, n. 9, pp. 1563–1581, 1966.

[34] ADAM, T. L., CHANDY, K. M., DICKSON, J. R. "A comparison of list schedules for parallel processing systems", *Communications of the ACM*, v. 17, n. 12, pp. 685–690, dec 1974. ISSN: 00010782. doi: 10.1145/361604.361619. Disponível em: <http://portal.acm.org/citation.cfm?doid=361604.361619>.

[35] HAGRAS, T., JANEČEK, J. "Static vs . Dynamic List-Scheduling Performance Comparison", *Acta Polytechnica*, v. 43, n. 6, pp. 16–21, 2003.

[36] DENNIS, J. B., FOSSEN, J. B. "Introduction to Data Flow Schemas", *Computation Strutuctures Group Memo, Massachutes Institute of Technology - Project MAC*, September 1973.

[37] GOLDSTEIN, B. F. *Construção de Núcleos Paralelos de Álgebra Linear Computacional com Execução Guiada por Fluxo de Dados*. Tese de Mestrado, COPPE - UFRJ, set. 2015.

[38] VEEN, A. H. "Dataflow Machine Architecture", *ACM Comput. Surv.*, v. 18, n. 4, pp. 365–396, dez. 1986. ISSN: 0360-0300. doi: 10.1145/27633.28055. Disponível em: <http://doi.acm.org/10.1145/27633.28055>.

[39] GURD, J. R., KIRKHAM, C. C., WATSON, I. "The Manchester Prototype Dataflow Computer", *Commun. ACM*, v. 28, n. 1, pp. 34–52, jan. 1985. ISSN: 0001-0782. doi: 10.1145/2465.2468. Disponível em: <http://doi.acm.org/10.1145/2465.2468>.

[40] GRAFE, V. G., DAVIDSON, G. S., HOCH, J. E., et al. "The Epsilon Dataflow Processor", *SIGARCH Comput. Archit. News*, v. 17, n. 3, pp. 36–45, abr. 1989. ISSN: 0163-5964. doi: 10.1145/74926.74930. Disponível em: <http://doi.acm.org/10.1145/74926.74930>.

[41] STERLING, T. "Studies on optimal task granularity and random mapping", *Advanced Topics in Dataflow Computing and Multithreading, 1995*, pp. 349–365, 1995.

[42] SINNEN, O. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007. ISBN: 0471735760.

[43] H. TOPCULOGLU, S. HARIRI, M. W. "Performance-effective and low- complexity task scheduling for heterogeneous computing". In: *IEEE Trans. Parallel Distrib. Systems 13 (3)*, pp. 260—274, March 2002.

[44] LOMBARDI, M., MILANO, M., RUGGIERO, M., et al. “Stochastic allocation and scheduling for conditional task graphs in multi-processor systems-on-chip”. In: *Journal of Scheduling doi: 10.1007/s10951-010-0184-y.*, pp. 315–345, 2010.

[45] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., et al. *Introduction to Algorithms*. 3 ed. Cambridge, Massachusetts, MIT Press, 2009.

[46] HENNESSY, J. L., PATTERSON, D. A. “Computer Architecture: A Quantitative Approach”. 5th ed., cap. Appendix F: Interconnection Networks, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[47] FORTIN-PARISI, S., SERICOLA, B. “A Markov model of TCP throughput, goodput and slow start”, *Performance Evaluation*, v. 58, n. 2-3, pp. 89–108, nov 2004. ISSN: 01665316. doi: 10.1016/j.peva.2004.07.016.

[48] MATHIS, M., SEMKE, J., MAHDAVI, J., et al. “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm”, *ACM SIGCOMM Computer Communication Review*, v. 27, n. 3, pp. 67–82, jul 1997. doi: 10.1145/263932.264023. Disponível em: <http://portal.acm.org/citation.cfm?doid=263932.264023>.

[49] ALTMAN, E., AVRACHENKOV, K., BARAKAT, C. “A Stochastic Model of TCP/IP With Stationary Random Losses”, *IEEE/ACM Transactions on Networking*, v. 13, n. 2, pp. 356–369, apr 2005. ISSN: 1063-6692. doi: 10.1109/TNET.2005.845536. Disponível em: <http://ieeexplore.ieee.org/document/1424044/>.

[50] Parallella Introduction. “The Parallella Board Overview”. Disponível em: <https://www.parallella.org/board/>. Visited in the 28th of August in 2017.

[51] LINUX FOUNDATION WIKI. “netem”. Accessed: 28-Aug-2017. Disponível em: <https://wiki.linuxfoundation.org/networking/netem>.