DF-DTM: EXPLORANDO REDUNDÂNCIA DE TAREFAS EM DATAFLOW

Leandro Rouberte de Freitas

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
Tiago Assumpção de Oliveira Alves

Rio de Janeiro
Maio de 2017

DF-DTM: EXPLORANDO REDUNDÂNCIA DE TAREFAS EM DATAFLOW

Leandro Rouberte de Freitas

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. Tiago Assumpção de Oliveira Alves, D.Sc.

Prof. Maurício Lima Pilla, D.Sc.

Prof. Claudio Luis de Amorim, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2017

Freitas, Leandro Rouberte de

DF-DTM: Explorando redundância de tarefas em Dataflow/Leandro Rouberte de Freitas. – Rio de Janeiro: UFRJ/COPPE, 2017.

XIV, 124 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Tiago Assumpção de Oliveira Alves

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2017.

Referências Bibliográficas: p. 117 – 124.

1. *Dataflow*. 2. *Memoização*. 3. Reúso. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"Existem coisas melhores adiante
do que qualquer outra que
deixamos para trás."
C.S Lewis

Agradecimentos

Agradeço primeiramente ao Senhor Jesus, meu Deus, por ter me dado capacidade, força e inteligência para chegar até aqui. Muitas vezes, pela sua misericórdia, abriu caminhos para que eu pudesse trilhar. Em todo tempo Ele esteve comigo, mesmo não merecendo, e, se o que foi realizado neste trabalho possui algum valor, o crédito é todo dEle, e não meu. A Ele seja a honra e a glória para todo sempre.

Agradeço aos professores Leandro Marzulo, Felipe França e Tiago Alves por terem participado comigo nesta empreitada. Muito obrigado professor Marzulo, mais uma vez você me demonstrou o que é ser um professor, um mestre e um amigo leal. Obrigado por toda a sua ajuda. Ao professor Felipe França, muito obrigado por ter aceitado me orientar, por se mostrar sempre solícito a responder minhas dúvidas e me ajudar. Ao professor Tiago, muito obrigado pelos conselhos, orientações acadêmicas e paciência.

Agradeço aos professores Maurício Lima Pilla, Claudio Luis de Amorim, Felipe França e Tiago Alves por terem aceitado fazer parte da banca e se colocarem à disposição para apreciação deste trabalho. Agradeço também ao professor Valmir Carneiro por ter viabilizado a minha ida ao WSCAD-2016 para apresentar este trabalho.

Agradeço à minha equipe da CETIP (atual $[B]^3$) , Antônio Caetano, Daniel Daemon, Sergio Ladany, Bruno Gomes, Claudio Rosa, Ricardo Furtado, Renato Araújo e Jorge Washington por terem garantido o melhor ambiente de trabalho que alguém poderia ter e por me apoiarem neste mestrado. Em especial, agradeço ao meu coordenador, Antônio Caetano, por muitas vezes flexibilizar meu horário de trabalho para que pudesse me dedicar aos assuntos do mestrado. Obrigado também por me incentivar em manter o foco e seguir em frente.

Agradeço aos amigos Marcos Paulo Rocha e Leandro Santiago pelo companheirismo durante o mestrado.

Agradeço aos meus primos Larissa, Augusto, Wallace, Abel e tios Karina, Adriano, Cristina, Jorge, Ana e Vavá por terem me encorajado esse tempo todo para que não desistisse e pelas suas orações.

Agradeço ao meu irmão Jean pelo seu apoio e por ouvir os meus desabafos muitas vezes em nossas conversas de madrugada. Agradeço por toda sua paciência e ajuda

nesses anos de mestrado e em todos outros.

Agradeço aos meus pais Carlos Alberto de Freitas e Lúcia Helena Rouberte de Freitas por todo carinho, amor e ensino que me deram. Por terem me apoiado até aqui, sempre me incentivando, sempre amorosos e pacientes. Obrigado por me mostrarem que com fé e dedicação é possível vencer os desafios da vida não importam as circunstâncias.

O meu agradecimento com todo meu coração e com tudo que tenho à mulher da minha vida, Gabriela da Cruz. Louvado seja Deus pela tua vida. Muito obrigado pelo seu amor, paciência, carinho, risos, piadas, momentos de felicidade, apoio em momentos de profunda tristeza e todas as outras coisas que só eu e Deus sabemos que você é capaz. Amo você com a minha própria vida e te agradeço por tudo que você me fez nesses seis anos de namoro. Esse trabalho é tão seu quanto meu, sem a sua ajuda não teria conseguido chegar tão longe. Você é minha fiel companheira, e espero tê-la assim por toda a minha vida.

Por fim, mais uma vez agradeço a Deus por todas essas pessoas que foram colocadas em minha vida e por muitas vezes, mesmo não reconhecendo, ter segurado a minha mão e me guiado até o fim. Não importa qual seja o desfecho desta história, eu Te louvo por ter estado comigo todo esse tempo.

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

DF-DTM: EXPLORANDO REDUNDÂNCIA DE TAREFAS EM DATAFLOW

Leandro Rouberte de Freitas

Maio/2017

Orientadores: Felipe Maia Galvão França
Tiago Assumpção de Oliveira Alves

Programa: Engenharia de Sistemas e Computação

Reutilização de instruções é uma técnica de otimização que pode ser utilizada em arquiteturas de Von Neumann. Esta técnica gera ganho de desempenho ao evitar que instruções redundantes executem, quando os resultados a serem produzidos por elas podem ser obtidos pesquisando uma tabela histórica de entrada/saída. A reutilização de traços pode ser aplicada aos traços de instruções de forma semelhante. No entanto, essas técnicas ainda precisam ser estudadas no contexto do modelo *Dataflow*[1–4], que vem ganhando notoriedade na comunidade de computação de alto desempenho devido ao seu paralelismo inerente. Os programas *Dataflow* são representados por grafos direcionados, onde os nós são instruções (tarefas *dataflow*) e as arestas denotam dependências de dados entre estas tarefas. Este trabalho apresenta a técnica *Dataflow Dynamic Task Memoization* (DF-DTM), a qual é inspirada na técnica de reúso de instruções em arquiteturas Von Neumann, conhecida como DTM[5] (*Dynamic Trace Memoization*). DF-DTM permite a reutilização de nós e subgrafos em modelos *Dataflow*, que são análogos às instruções e traços em modelos tradicionais, respectivamente. Modificamos a versão *Python* da biblioteca *Dataflow*, Sucuri[6], para implementar a técnica DF-DTM. O potencial da DF-DTM é avaliado por uma série de experimentos que analisam o comportamento de tarefas redundantes em cinco *benchmarks* relevantes: *Conway's Game of Life* (GoL), *longest common sub-sequence* (LCS), uma aplicação de criptografia utilizando o padrão *Triple DES* (3-DES), uma aplicação de contagem de palavras que segue um modelo *MapReduce* (MR) e o problema da mochila sem limites (KS). Nossos resultados mostram uma notável taxa de redundância potencial de aproximadamente $98,83\%$, $54,73\%$, $72,35\%$, $99,73\%$ para as aplicações *benchmarks* LCS, MR, 3-DES e GoL, respectivamente.

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

DF-DTM: EXPLORING REDUNDANCY IN DATAFLOW TASKS

Leandro Rouberte de Freitas

May/2017

Advisors: Felipe Maia Galvão França
Tiago Assumpção de Oliveira Alves

Department: Systems Engineering and Computer Science

Instruction reuse is an optimization technique that can be used in Von Neumann architectures. This technique improves performance by avoiding the execution of redundant instructions, when the result to be produced by them can be obtained by searching a historical input/output table. Trace reuse can be applied to traces of instructions in a similar fashion. However, these techniques still need to be studied in the context of the Dataflow [1–4] model, which has been gaining traction in the high-performance computing community, due to its inherent parallelism. Dataflow programs are represented by directed graphs where nodes are instructions or tasks and edges denote data dependencies between tasks. This work presents Dataflow Dynamic Task Memoization (DF-DTM) technique, which is inspired by the Von Neumann architecture reuse technique known as DTM [5] (Dynamic Trace Memoization). DF-DTM allows the reuse of nodes and subgraphs in Dataflow models, which are analogous to instructions and traces in traditional models, respectively. We modified the Python version of the Dataflow library, Sucuri[6], to implement the DF-DTM technique. The potential of the DF-DTM is evaluated by a series of experiments that analyze the behavior of redundant tasks in five relevant benchmarks: Conway's game of life (GoL), longest common sub-sequence (LCS), a cryptography application using the Triple DES standard (3-DES), a word counting application that follows a MapReduce (MR) model and the unbounded knapsack problem (KS). Our results shows a remarkable potential redundancy rate of approximate 98.83%, 54.73%, 72.35%, 99.73% for the benchmarks applications LCS, MR, 3-DES and GoL, respectively.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Extrair desempenho de processadores sempre foi um tarefa árdua para arquitetos e programadores. Estratégias como previsão de desvios, execução especulativa, uso de *pipelines* profundos, despachos múltiplos de instruções são exemplos de estratégias arquiteturais para melhorar o desempenho de processadores. O aumento da frequência do *clock* dos processadores também era uma prática muito utilizada pelas fabricantes, entretanto esta estratégia tornou-se inviável conforme os limites físicos dos *chips* foram alcançados. Isso fez com que as fabricantes optassem por outras estratégias para extrair desempenho de seus *chips*.

Atualmente, o aumento do número de *cores* por *chips* processadores tornou-se a estratégia mais bem sucedida, dando origem à processadores *multi-cores* e aceleradores paralelos, por exemplo, GPUs[7] e co-processadores XEON-PHI[8]. Entretanto, o aumento de desempenho de aplicações ao utilizar processadores paralelos não ocorre de forma direta, como era de costume com o aumento dos *clocks* de *chips single-core*. Para alcançar este objetivo é necessário que aplicações sejam re-escritas por programadores experientes que consigam analisar e decompor um programa de forma que tarefas independentes possam ser executadas paralelamente em núcleos processadores distintos. Este novo paradigma de programação, conhecido como programação paralela, insere muitos novos desafios ao desenvolvimento de aplicações, tais como, distribuição de tarefas, balanceamento de carga, comunicação, sincronização, garantia de recursos, entre outros. Interfaces como OpenMP[9] e MPI[10] são exemplos de ferramentas disponíveis à comunidade de computação para facilitar, até certo ponto, os desafios impostos pela programação paralela.

O modelo *Dataflow*[2, 3, 6, 11–13] de execução permite explorar o paralelismo natural das aplicações. Um programa *dataflow* é descrito como um grafo direcionado, também conhecido como grafo *dataflow*. Neste grafo, cada nó representa uma tarefa da aplicação, e as arestas entre os nós descrevem a dependência de dados entre estas tarefas. Desta forma, quando os dados de entrada de uma determinada tarefa estão disponíveis, ela pode ser executada. Isso permite que tarefas sejam

executadas de forma paralela, e não mais obedecendo uma ordem de programa. De fato, no modelo *Dataflow* não existe o tradicional *Program Counter* (PC), que dita a ordem de execução de um programa, encontrado em arquiteturas de Von Neumann. Pesquisas recentes têm usado o referido modelo como uma alternativa atraente para programação paralela que, além de ser mais transparente para os usuários, consegue prover o desempenho desejado [6, 14–17]. Além disso, arquiteturas *Dataflow* têm sido propostas por diferentes grupos de pesquisa [4, 18] e já são uma realidade na indústria de aceleradores para computação de alto desempenho [12].

Além do aumento de *cores* processadores, outras estratégias e soluções de nível arquitetural têm sido pesquisadas para aumentar o desempenho em um *chip*. Uma delas, conhecida como DTM (*Dynamic Trace Memoization*) [5], consiste em uma técnica de *Machine Learning* denominada *Memoization* [19]. O funcionamento da memorização em processadores é trivial. Operandos de entrada e saída de instruções são armazenadas em uma tabela histórica conforme a execução de um programa ocorre. Antes de uma nova instância de uma determinada instrução executar, esta tabela histórica é consultada. Se for encontrado um registro que possua os operandos de entrada e saída para essa instrução, ela é ignorada, e os operandos de saída são utilizados para dar continuidade à execução. Quando sequências de instruções são reutilizadas, conjuntos de instruções redundantes são formados, estes são chamados de traços. O reúso de traços e instruções redundantes pode fornecer ganhos substanciais para aplicações, visto que eles são comuns em programas de diferentes naturezas.

Neste trabalho propomos o uso do modelo de execução *Dataflow* em sinergia com a estratégia de memorização. Para isso criamos a DF-DTM (*DataFlow Dynamic Trace Memoization*. DF-DTM é uma técnica de memorização de tarefas *dataflow* inspirada na DTM. Ela permite a memorização de resultados de nós, que seriam equivalentes à instruções Von Neumann, em um grafo *dataflow*. A DF-DTM também identifica reúsos contínuos de nós no grafo *dataflow*, e utiliza esses reúsos para criar subgrafos redundantes, os quais seriam equivalentes à traços em arquiteturas de Von Neumann. A DF-DTM utiliza dois tipos de tabelas históricas para reutilizar nós e subgrafos redundantes: a NRT (*Node Reuse Table*) e SRT (*Subgraph Reuse Table*). A NRT armazena resultados de nós previamente executados no grafo, enquanto a SRT armazena resultados de subgrafos previamente executados. A DF-DTM introduz o conceito de inspeção de tarefas *dataflow*, o que permite um reúso pontual de tarefas que estão aguardando recursos do sistema para serem executadas.

O objetivo principal deste trabalho é avaliar o potencial de reúso de tarefas no paradigma *Dataflow*. Para isso, adaptamos a Sucuri[20, 21] para utilizar a DF-DTM. Sucuri é uma biblioteca *Dataflow* que permite que aplicações sejam descritas como um grafo *dataflow* utilizando a famosa linguagem *Python*. O programador que

utiliza esta biblioteca é responsável por dividir seu programa em funções *Python* que serão passadas como argumento para objetos *Nodes*. Ele também é responsável por criar arestas entre estes nós (objetos *Nodes*) especificando a dependência de dados entre eles. O escalonador central da Sucuri é responsável por identificar quais nós possuem todos os operandos de entrada disponíveis, e, a partir destes nós, instanciar tarefas, que serão colocadas em uma fila de tarefas prontas para execução. *Workers*, que são processos paralelos executando na mesma máquina do escalonador ou em máquinas distintas interligadas por rede, são responsáveis por solicitar ao escalonador uma tarefa da fila, quando estes estão ociosos. Após a execução de uma tarefa por um determinado *worker* os operandos resultantes são enviados de volta ao escalonador central, o qual os encaminha para as portas de entrada dos próximos nós obedecendo a dependência descrita pelas arestas do grafo. Quando novos operandos são disponibilizados aos nós, novas tarefas podem ser instanciadas.

Existem diferentes momentos no *pipeline* da Sucuri em que a detecção de redundância pode começar. Por conta disso, achamos relevante implementar diferentes estratégias para detecção de redundância. Estas estratégias são: *Eager Match* (EM), *Eager Match + Inspection* (EM+I), *Lazy Match* (LM), *Lazy Match + Inspection* e *Late Reuse Queue* (LRQ). A estratégia EM realiza detecção de redundância de forma imediata, assim que um novo operando chega ao escalonador. A estratégia LM processa todos os operandos recebidos pelo escalonador antes de começar a detecção de redundância. Pelo fato da DF-DTM não tornar mandatório o uso do mecanismo de inspeção, foram criadas as estratégias EM+I e LM+I, as quais funcionam de forma idêntica às estratégias EM e LM, respectivamente, porém utilizam o mecanismo de inspeção. A estratégia LRQ realiza detecção de redundância tardiamente no *pipeline* da Sucuri, isto é, somente quando um *worker* ocioso solicita uma tarefa.

Implementamos diferentes *benchmarks Dataflow* com diferentes padrões de grafos e propagações de operandos para avaliar a nossa técnica de reúso. Estas aplicações são LCS (*Longest Common Subsequence*), *Word Counting MapReduce*, GoL (*Game of Life*), o algoritmo da mochila sem limites (*UK - Unbounded Knapsack*) e uma aplicação de criptografia que utiliza o padrão 3-DES. Cada aplicação dessa produz grafos *dataflow* completamente distintos ou que pelo menos possuam um formato de propagação de operandos distintos. Por exemplo, o grafo da aplicação *MapReduce* possui um padrão *fork-join*, enquanto os grafos das aplicações LCS e UK possuem grafos utilizando o modelo *Wavefront* de paralelismo.

Nossos resultados mostram uma quantidade expressiva de redundância encontrada nestas aplicações, excetuando-se somente o algoritmo da mochila. De todas as tarefas determinísticas das aplicações LCS, *MapReduce*, Gol e 3-DES, $98,83\%$, $54,73\%$, $99,74\%$ e $72,35\%$ são redundantes, respectivamente. Experimentos vari-

ando tamanhos de *caches* NRT e SRT demonstraram que é possível utilizar poucas linhas de *caches* e ainda sim aproveitar grande parte do potencial de redundância das aplicações, o que é encorajador para utilização da DF-DTM em *hardwares Dataflow*. Experimentos também demonstraram que o mecanismo de inspeção pode prover um grande benefício à detecção de reúso, principalmente para aplicações com explosão de paralelismo.

Este trabalho realiza as seguintes contribuições:

- Estudo de reúso em nós e subgrafos em um modelo *Dataflow*.
- Estudo de tipos distintos de grafos *dataflow* e suas relações com o reúso de computação.
- Criação e implementação da DF-DTM.
- Criação e avaliação de um mecanismo de inspeção para modelos *Dataflow*.
- Avaliação da técnica de reutilização de computação sob diferentes estratégias.
- Avaliação do comportamento do reúso em *Dataflow*, quando utilizadas restrições de recursos.
- Demonstração da viabilidade da Sucuri como simulador funcional *Dataflow*.
- Criação de um conjunto de *benchmarks Dataflow* para Sucuri.

Esta dissertação está organizada da seguinte forma: Os capítulos 2 e 3 apresentam conceitos básicos sobre o modelo de execução *Dataflow* e reúso de computações, enquanto o capítulo 4 apresenta trabalhos relacionados a estes temas. O capítulo 5 apresenta com mais detalhes a arquitetura da biblioteca Sucuri, enquanto o capítulo 6 apresenta a implementação da DF-DTM nesta biblioteca. No capítulo 7, dissertamos sobre as implementações dos *benchmarks Dataflow* e discussões sobre os resultados experimentais. Por fim, o capítulo 8 apresenta conclusões e trabalhos futuros.

Capítulo 2

Modelo *Dataflow*

Neste capítulo apresentamos conceitos introdutórios sobre o modelo *dataflow* de execução. Arquiteturas e modelos de programação baseados em *dataflow* possuem a vantagem de explorar naturalmente o paralelismo inerente de aplicações, pois permite que tarefas computacionais sejam executadas conforme a disponibilidade de seus dados de entrada. As seguintes seções apresentam uma visão geral deste modelo. Apresentamos formas de que um modelo guiado por fluxo de dados possa implementar tarefas com desvios de controle, tais como, laços e cláusulas de desvios condicionais. Por fim, apresentamos alguns dos principais desafios presentes no modelo *Dataflow*.

2.1 Visão geral

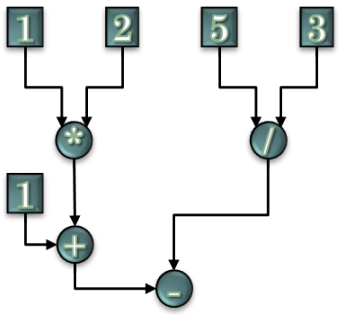
O modelo de execução *Dataflow* possui uma característica que tem atraído atenção de vários pesquisadores: sua capacidade de explorar o paralelismo inerente de aplicações. Essa é uma característica muito desejada, pois a estratégia mais utilizada e bem sucedida para ganho de desempenho é o uso de programação paralela. Este modelo consegue desempenhar essa exploração através da sua forma de descrever aplicações e escalonar tarefas. Uma aplicação escrita no paradigma *Dataflow* é representada como um grafo, chamado de grafo *dataflow*, onde os nós são as tarefas computacionais. Estas podem possuir diferentes granularidades, por exemplo, uma instrução, caso o modelo *Dataflow* esteja implementado na forma de um microprocessador, ou uma função, caso seja um modelo *Dataflow* implementado em *software* sobre uma arquitetura tradicional. No grafo *dataflow*, as arestas são utilizadas para representar as dependências de dados entre os nós (tarefas). Desta forma, quando um nó possui todos os operandos de entrada disponíveis, este pode executar a tarefa que ele representa, e enviar os resultados produzidos para os nós dependentes.

Os grafos podem ser direcionados e acíclicos (DAG), neste caso o modelo *Dataflow* em questão é denominado estático. Entretanto, modelos de *Dataflow* dinâmicos

permitem a utilização de grafos direcionados com ciclos (DDG)[13]. Grafos DDG permitem a criação de laços, mas para isso é necessário a criação de mecanismos que possam prover a distinção entre instâncias de iterações de um mesmo laço. O modelo estático pode simular um laço de forma limitada através da utilização de grafos de *stream* (*DAG-Stream*). Estas questões são discutidas com mais detalhes na seção 2.3.

```
1  void foo(){
2      int k=1;
3      int l=2;
4      int m=5;
5      int n=3;
6      int a,b,c;
7
8      a=l*k;
9      b=m/n;
10     a+=1;
11     c=a-b;
12 }
```

(a) Programa sequencial.



(b) Grafo *dataflow*.

Figura 2.1: Transformação de um programa sequencial em um grafo *dataflow*.

A figura 2.1a apresenta uma aplicação escrita em uma linguagem sequencial utilizada em máquinas tradicionais que seguem o modelo de Von Neumann. Na figura 2.1b, vemos a mesma sequência de instruções representada em um grafo *dataflow*. Note que as instruções $a = k * l$ e $b = m/n$, em um modelo de Von Neumann, seriam executadas de forma sequencial, mas por não haver dependência de dados entre elas, ambas poderiam executar paralelamente. Repare também que, no modelo *Dataflow*, este paralelismo é explorado simplesmente pela forma de representação das instruções.

2.2 Desvios em execução *dataflow*

Modelos tradicionais de computação têm suas execuções guiadas pelo fluxo de controle, isto é, a execução das instruções ocorrem na ordem em que elas aparecem no programa. Para isso é utilizado um contador de programa (PC) que avança de uma instrução corrente a uma instrução sucessora. Nos casos de desvios de controle, por exemplo, um laço ou uma cláusula *if-else*, se faz necessário modificar o próximo endereço que o PC irá utilizar. Arquiteturas *Dataflow* não possuem contador de programas, portanto é necessário converter o fluxo de controle de um programa em fluxo de dados. Existem duas formas de implementar desvios em *dataflow*: (i)

instruções seletoras condicionais[22, 23] ou (ii) instruções de desvios condicionais, também chamadas de *steer*[3, 24].

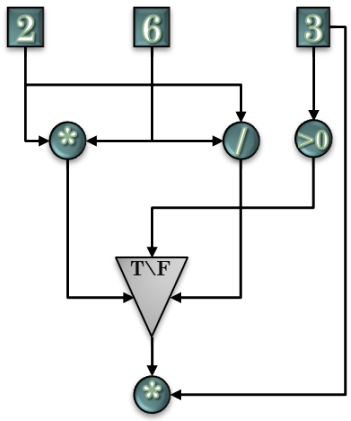
2.2.1 Instruções seletoras

Instruções seletoras recebem três operandos de entrada: dois operandos provenientes de caminhos distintos no grafo e um operando booleano que irá servir como seletor. Dependendo do valor do operando booleano, um dos dois valores de entrada são propagados no grafo. Essa estratégia possui o benefício de permitir que mais instruções possam ser executadas em paralelo, pois não há necessidade de esperar a instrução seletora executar para permitir que mais instruções sejam executadas. Entretanto, elas geram desperdício de recursos, pois uma instrução seletora apenas permitirá que um operando seja enviado adiante no grafo, logo, as instruções executadas para produção do operando não selecionado foram executadas inutilmente. Instruções seletoras só podem ser utilizadas em grafos acíclicos, isto é, utilizando apenas esta estratégia não é possível implementar laços no grafo.

A figura 2.2 apresenta um exemplo de desvios em um programa tradicional representado por um grafo *dataflow* com instruções seletoras. O programa à esquerda é convertido em um grafo *dataflow* a direita. As instruções seletoras são representadas pro triângulos invertidos. Note que todos os nós são executados, porém a instrução $a = a/b$ foi executada inutilmente.

```
1  void foo(){
2      float a=6;
3      float b=2;
4      int k=3;
5
6      if(k>0){
7          a=a*b;
8      }
9      else{
10         a=a/b;
11     }
12     a=k*a;
13 }
```

(a) Programa sequencial.



(b) Grafo *dataflow* com função seletora.

Figura 2.2: Transformação de um programa sequencial em um grafo *dataflow* utilizando função seletora.

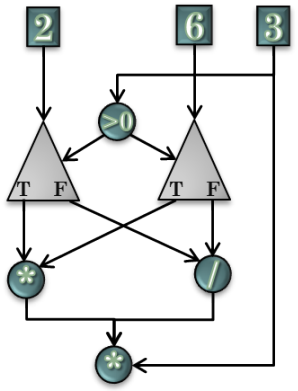
2.2.2 Instruções *steer*

Instruções *steer* recebem dois operandos: um operando normal e um operando booleano. O valor booleano é utilizado para desviar o operando para um de dois nós (instruções) disponíveis, o qual de fato deve receber o operando. É necessário que exista uma instrução de *steer* para cada operando utilizado nas diferentes ramificações oriundas do desvio de execução. *Steers* funcionam tanto para implementar desvios em grafos cíclicos como acíclicos. Elas são necessárias também para implementação de laços.

A figura 2.3 apresenta o mesmo exemplo da figura 2.2, porém desta vez em um grafo que utilize instruções *steer*. Estas são representadas na imagem como triângulos. Repare que as instruções pertencentes ao desvio foram executadas somente após a execução da instrução geradora do operador booleano e da instrução *steer*. Note também que foi necessário a criação de múltiplas instruções *steer*, uma para cada operando pertencente aos possíveis desvios.

```
1  void foo(){
2      float a=6;
3      float b=2;
4      int k=3;
5
6      if(k>0){
7          a=a*b;
8      }
9      else{
10         a=a/b;
11     }
12     a=k*a;
13 }
```

(a) Programa sequencial.



(b) Grafo *dataflow* com função *steer*.

Figura 2.3: Transformação de um programa sequencial em um grafo *dataflow* utilizando *steers*.

2.3 Laços em execução *dataflow*

Laços são fontes potenciais de paralelismo em programas. Iterações distintas de um mesmo laço podem muitas vezes executar simultaneamente. O modelo *Dataflow* permite explorar paralelismo provenientes de instruções de um mesmo laço em uma mesma iteração, assim como, paralelismo proveniente de execuções de instruções de diferentes iterações em paralelo. Entretanto, para se executar iterações distintas de forma paralela, é necessário utilizar estratégias que permitam diferenciar operan-

dos pertencentes a uma iteração em relação aos operandos de outras iterações de um mesmo laço. Existem duas estratégias de implementação de laços em modelos *Dataflow*, estas são: *Dataflow* dinâmico e *Dataflow* estático.

- *Dataflow* estático: Permite a execução de uma nova iteração de um laço somente quando a anterior finalizar. Isso reduz fortemente o paralelismo encontrado em aplicações, porém simplifica o controle de desvios no grafo e operandos.

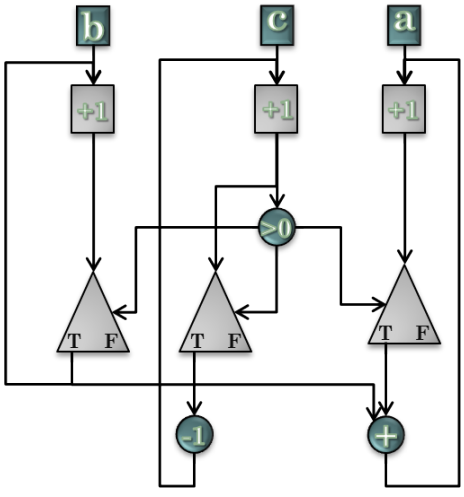
- *Dataflow* dinâmico: Permite execuções paralelas de iterações distintas de um mesmo laço, o que melhora a exploração do paralelismo em aplicações. Entretanto, se faz necessário inserir instruções de desvios condicionais e um mecanismo que permita a diferenciação de operandos entre iterações.

O mecanismo utilizado em modelos dinâmicos para diferenciar operandos entre iterações consiste em rotular cada operando com o número da instância da iteração. Este número é chamado de *tag*. Por conta disto, é necessário introduzir instruções de incremento de *tags*. Cada operando utilizado dentro do laço, antes de ser passado às instruções interiores ao laço, deve ter sua *tag* incrementada. Isso permite a identificação única de operandos.

A figura 2.4a apresenta um laço de um programa, enquanto a figura 2.4b apresenta o exemplo de um modelo *Dataflow* dinâmico para este laço. Os nós especiais que incrementam *tags* são representados como um quadrado e o valor +1. Note que cada operando utilizado pelo laço é recebido pelas instruções de incremento de *tags* após suas instruções de desvios (triângulos na figura) os repassarem à elas.

```
1  void foo(int a, int b, int c){
2      for(int i=c;i>0;i--){
3          a=a+b;
4      }
5
6  }
```

(a) Programa sequencial.



(b) Grafo *dataflow* cíclico com função *steer*.

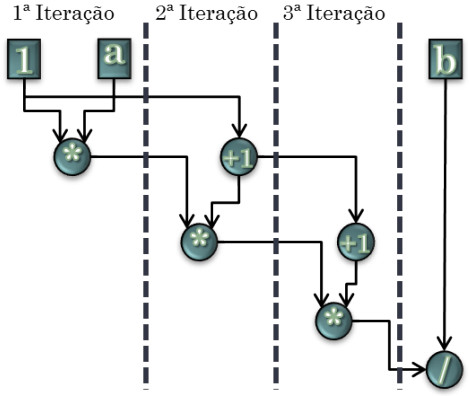
Figura 2.4: Implementação do laço em *Dataflow* dinâmico.

Existem formas mais simples de implementar repetições em *Dataflow*, entretanto estas repetições devem possuir estruturas mais limitadas. Por exemplo, se o número de iterações de um laço é conhecido, o grafo pode ser desenrolado semelhantemente a uma técnica de *loop unroling* em um compilador. Isso gera grafos acíclicos mais simples, entretanto, pode gerar grafos demasiadamente grandes. Programas cujas repetições são causadas por operações sobre elementos de um objeto iterável previamente conhecido e que não haja dependência de dados entre as iterações, também podem ser implementadas de forma mais simples em um grafo acíclico com características de *stream*[25, 26].

A figura 2.5 apresenta um exemplo de um grafo desenrolado com o código original do laço à esquerda. Note que ele não possui instruções para incremento de rótulos, entretanto, seu tamanho pode crescer exponencialmente, se muitas iterações forem utilizadas. A figura 2.6 apresenta um grafo acíclico de *stream* com o código original do programa à esquerda. Neste caso, há necessidade incrementos de *tags*, pois os diferentes elementos do objeto iterável podem estar sendo processados em paralelo. O hexágono amarelo representa o nó que lerá os elementos do objeto iterável, neste caso um *array* de 4 elementos, e para cada operando lido irá incrementar o rótulo deste.

```
1  void foo(float a){
2      float b=12;
3
4      for(int i=1;i<=3;i++){
5          a=a*i;
6      }
7
8      b=b/a;
9
10 }
```

(a) Programa sequencial.



(b) Grafo *dataflow* acíclico desenrolado.

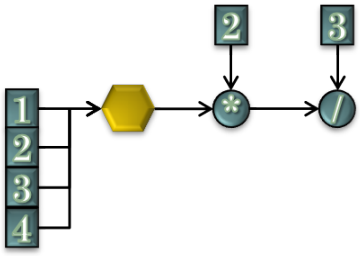
Figura 2.5: Implementação limitada do laço em *dataflow* estático.

Neste trabalho é interessante classificar os diferentes tipos de grafos *dataflow* em 3 conjuntos:

- DDG - Grafos *dataflow* dinâmicos. Estes são grafos que permitem a expressão de estruturas dinâmicas através da implementação de laços no grafo.
- DAG - Grafos *dataflow* acíclicos. Estes permitem a criação de grafos estáticos que são mais simples de implementar e não requerem estruturas de desvios complexas ou rotulação de operandos.

```
1  void foo(){
2      int V={1,2,3,4};
3      float a;
4
5      for(int i=0;i<4;i++){
6          a=V[i]*2;
7          a=a/3;
8      }
9  }
```

(a) Programa sequencial.



(b) Grafo *dataflow stream* processando objeto iterável.

Figura 2.6: Implementação do laço em *dataflow stream*.

- DAG *Stream* - Grafos *dataflow* acíclicos que operam sobre objetos iteráveis. Possuem um nó especial para alimentar o grafo com operandos novos. Também necessitam de rotulação de operandos.

2.4 Principais desafios do modelo *Dataflow*

O paradigma *Dataflow* possui muitas características vantajosas, entretanto, ele também possui muitos desafios interessantes que merecem atenção[27]. Nesta seção, dissertamos sobre alguns dos principais desafios do modelo *Dataflow*. Os desafios pertinentes à tradução de fluxos de controle em fluxos de dados foi discutido nas seções 2.2 e 2.3, portanto, nesta seção focamos em outros desafios propostos por esse novo modelo. Estes são: (*i*) mudança no paradigma de programação, (*ii*) efeitos colaterais, (*iii*) alocação de tarefas e (*iv*) recursos computacionais para implementação do modelo.

2.4.1 Mudança no paradigma de programação

O paradigma de programação tradicional consiste em explicitar um algoritmo em uma sequência de passos a serem executados de forma ordenada. No paradigma *Dataflow*, é exigido ao programador uma nova forma de pensar em algoritmos. É necessário que o programador descreva o algoritmo como tarefas, não necessariamente sequenciais, e as dependências de dados entre elas. Essa forma de construção pode não ser tão trivial quanto à forma tradicional, mas a adaptação de linguagem tradicionais à construtores *dataflow* pode fornecer um caminho a essa adaptação que permitirá a criação de aplicações paralelas mais facilmente. Trabalhos como [1, 2, 6, 28] adaptam linguagens tradicionais ao modelo *Dataflow* e permitem que o programador crie aplicações paralelas representadas como grafos.

2.4.2 Efeitos colaterais

O modelo *Dataflow* tradicional não permite a realização de operações que alterem o estado global da máquina. Operações que modifiquem o valor de uma variável global ou altere o valor de um endereço de memória são exemplos de operações com efeitos colaterais. Por conta dessa limitação, muitas estruturas de dados tradicionais, por exemplo,*arrays* e *structs*, são de difícil implementação no modelo *Dataflow*. Isso ocorre, pois seria necessário, ao modificar uma dessas estruturas, gerar uma nova cópias das mesmas e passá-las às operações (nós) dependentes.

Duas propostas de representação de estruturas de dados são apresentadas para implementar tais estruturas no modelo *Dataflow*: acesso direto e acesso indireto. Acessos diretos consiste em tratar elementos de uma estrutura de dados como um único operando *dataflow*. Esses operandos além de seus valores, possuem informações relevantes ao contexto da estrutura de dados. Por exemplo, em caso de *arrays* um operando pode consistir em uma tupla com o seu valor e sua posição em relação ao *array*. Em acessos indiretos uma área especial é criada na memória para armazenar essas estruturas de forma apropriada. Acessos são feitos à elas através de operações explícitas de leitura e escrita.

2.4.3 Alocação de tarefas

Alocação de tarefas, assim como em arquiteturas tradicionais, impõe sérios desafios às arquiteturas *Dataflow*. Diferentes estratégias de alocação de tarefas aos elementos processadores (EPs) influenciam diretamente o desempenho da arquitetura *Dataflow*. O objetivo principal de estratégias de alocação é aumentar o uso de recursos computacionais sem causar contenções, e ao mesmo tempo reduzir o custo de comunicação entre elementos processadores.

Existem duas propostas para alocação de tarefas: estática e dinâmica. A estratégia de alocação estática consiste em alocar às tarefas *dataflow* em EPs no momento de compilação do programa *dataflow* utilizando informações globais do sistema e programa. Após a definição de alocação, tarefas não poderão ser executadas em outros EPs, ainda que sejam executadas várias vezes. Este esquema possui uma desvantagem, quando estimativas de comportamentos em tempo de execução não correspondem à realidade. Por exemplo, um ciclo em um grafo *dataflow* dinâmico pode ser alocado inteiramente a um único EP por achar que este não irá gerar muitas iterações. Porém, se houver muitas iterações deste ciclo em tempo de execução, um EP ficará sobrecarregado e os demais poderão ficar ociosos. Uma política dinâmica de alocação consiste em mensurar os custos, em tempo de execução, de execuções de tarefas *dataflow* em EPs, e designar tarefas ao EP menos sobrecarregado. A maior desvantagem nesta política é o custo necessário para determinar a carga de trabalho

dos EPs e alocar as tarefas dinamicamente com esta carga.

2.4.4 Recursos computacionais

O modelo *Dataflow* permite explorar o paralelismo natural de aplicações. Entretanto, em alguns casos, isso pode gerar um consumo muito alto de recursos computacionais por parte das aplicações. Por exemplo, neste modelo, um laço pode ser desenrolado em tempo de execução e cada iteração pode ser executada em paralelo. Apesar disso ser interessante para exploração de paralelismo, se o laço expuser muitas iterações, recursos podem ficar escassos e contenções podem ocorrer. Portanto, é necessário utilizar formas de restringir o paralelismo, mas ainda sim manter o consumo dos recursos de forma equilibrada, isto é, não permitir ociosidades. Políticas para inserções de falsas dependências entre iterações distintas foram propostas para gerar limitações na instanciação de várias iterações de um mesmo laço, entretanto elas podem diminuir a exploração do paralelismo presente nas aplicações.

Capítulo 3

Reúso de computação

Reúso de computação é uma técnica de otimização baseada na estratégia de aprendizado de máquina conhecida como *memoização*[19]. Esta estratégia consiste basicamente em armazenar entradas e saídas de tarefas em uma tabela histórica. Durante a execução de um programa, caso uma tarefa tenha seus operandos de entrada armazenados nesta tabela, a tarefa pode ser ignorada, e os operandos de saída podem ser entregues às demais tarefas que necessitem deles. Quando isso ocorre, é dito que uma tarefa é redundante e foi reutilizada ou que seus resultados foram reaproveitados. Reúso de tarefas pode gerar ganhos de desempenho em arquiteturas computacionais e aplicações, pois permite que o custo de execução de uma tarefa complexa possa ser trocado por um custo, possivelmente menor, de busca em uma tabela histórica.

Na figura 3.1 exemplificamos os conceitos básicos de reúso de tarefas. Utilizamos uma função recursiva para calcular termos da série de Fibonacci. A tabela histórica utilizada por esse mecanismo de reúso exemplificado é indexada pelos operandos de entrada desta função, porém, como veremos adiante, outras formas de indexação podem existir. A figura 3.1a apresenta o código de duas funções recursivas para calcular o *n-ésimo* termo da série de Fibonacci: uma implementada de forma tradicional e outra usando uma tabela histórica de memorização. No exemplo, executamos a função de Fibonacci para $n = 5$. A figura 3.1b apresenta a árvore de recursão dessa execução. Cada nó da árvore representa uma chamada à função *fib_mem*. Note que, conforme a execução do programa avança pelas ramificações mais a esquerda da árvore, resultados são armazenados na tabela *Tab_Memo* (figura 3.1c). Por conta disso, as ramificações envoltas por retângulos cinzas na figura 3.1b não são executadas. Antes do corpo tradicional da função de Fibonacci ser invocado, a tabela de memorização é consultada. Quando a saída da chamada é conhecida, o valor é retornado da tabela, do contrário, a função é computada em sua totalidade. O reúso de computação permitiu uma poda na árvore de recursão, o que pode acarretar em ganhos de desempenho. Em sistemas computacionais o reaproveitamento

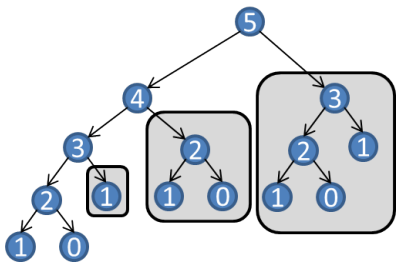
de tarefas também pode gerar economia de energia.

```
#Declaramos a tabela histórica
Tab_Memo = {}

#Função recursiva de Fibonacci
#tradicional
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

#Função recursiva de Fibonacci
#com reúso de computação
def fib_mem(n):
  if n in Tab_Memo:
    return Tab_Memo[n]
  else:
    if n == 0 or n == 1:
      Tab_Memo[n] = n
      return n
    else:
      Tab_Memo[n] = fib_mem(n-1) +
                    fib_mem(n-2)
      return Tab_Memo[n]
```

(a) Funções Fibonacci.



(b) Árvore de recursão.

Tab_Memo

Índice	Resultado
0	0
1	1
2	1
3	2
4	3
5	5

(c) Tabela histórica.

Figura 3.1: Esquema de reúso básico para uma função de Fibonacci.

Tarefas podem ser classificadas em determinísticas ou não-determinísticas. Tarefas determinísticas sempre retornam um mesmo resultado, dado que os operandos de entrada são os mesmos. Tarefas não-determinísticas possuem resultados imprevisíveis, por exemplo, tarefas que possuam aleatoriedade em sua natureza. Memorização desses tipos de tarefas são analisadas em estudos de *memoização* estocástica[29], e não fazem parte do escopo deste trabalho.

Neste capítulo apresentamos conceitos básicos sobre reúso de computação em

três níveis distintos: arquitetura de processadores (*hardwares*), compiladores e desenvolvimento de aplicações (*software*).

3.1 Reúso de computação em processadores

Técnicas de reúso de computação possuem um grande potencial em melhoria de desempenho, quando aplicadas em processadores [5, 30–32]. Neste esquema de reúso, *buffers* são adicionados ao *hardware* para armazenar valores de entrada de instruções dinâmicas e suas respectivas saídas. Esses *buffers* também chamados de *caches* servem como uma tabela de memorização.

Instruções dinâmicas são instanciações a partir de instruções estáticas que compõe um programa, por exemplo, *addi $3, $2, 1* é uma instrução estática. Quando esta é executada para um determinado valor, por exemplo, $2 = 5, dizemos que esta é uma instrução dinâmica criada a partir da instrução estática *addi $3, $2, 1*, e que o operando $2 = 5 é o operando de entrada desta instrução dinâmica. De forma semelhante, $3 = 6 é definido como operando de saída da instrução. Uma instrução estática pode gerar n instruções dinâmicas. Quando uma instrução dinâmica é reutilizada, ela é denominada instrução redundante.

Implementações de reúso de computação classificam instruções em dois grupos: instruções válidas e instruções inválidas para o reúso. Instruções que alterem o estado da máquina, como um *store*, por exemplo, podem ser consideradas como inválidas, pois, se ignoradas, deixam de gravar um valor na memória, o que, muito provavelmente, danificará a lógica do programa. Algumas estratégias também consideram *loads* como instruções inválidas, outras utilizam meios para validar se um *load* pode ser reutilizado verificando se o endereço de memória do qual a instrução irá ler o valor já foi sobrescrito por outra instrução. Normalmente é utilizado um campo de validade de memória e um campo que armazena o endereço alvo de memória na entrada da tabela de reúso. Quando uma instrução sobrescreve o local de memória, entradas com este endereço têm seus campos de validade de memória invalidados.

Traços redundantes são sequências de instruções redundantes. Estas não precisam possuir dependência de dados entre si para fazerem parte de um mesmo traço. Reúso de traços é muito benéfico para melhoria de desempenho em processadores, pois permite que com uma única operação de busca em uma *cache* (tabela de memorização), múltiplas instruções redundantes sejam ignoradas, o que gera uma grande economia de ciclos de *clock*. A construção de um traço é iniciada, quando uma instrução redundante é detectada. Conforme instruções redundantes são detectadas em sequência, elas são adicionadas ao traço. Quando uma instrução inválida ou não redundante é encontrada a construção do traço termina. Um traço possui um contexto de entrada e um contexto de saída. Contextos de entrada são formados

pelos operandos de entrada pertencentes às instruções do traço, mas que não foram produzidos por instruções pertencentes ao traço. O contexto de saída é formado por operandos resultantes de instruções do traço que são utilizados por instruções exteriores ao traço.

A figura 3.2 exemplifica a formação de um traço. Note que anteriormente a formação do traço, as instruções *add*, *sub* e *mul* foram previamente identificadas como redundantes, e que após a formação do traço, os operandos $\{\$6, \$7\}$ e $\{\$3, \$4, \$9\}$ delas formam os contextos de entrada e saída do traço, respectivamente. Repare também que a construção do traço finaliza e os contextos são formados, quando uma instrução inválida é encontrada na sequência de instruções.

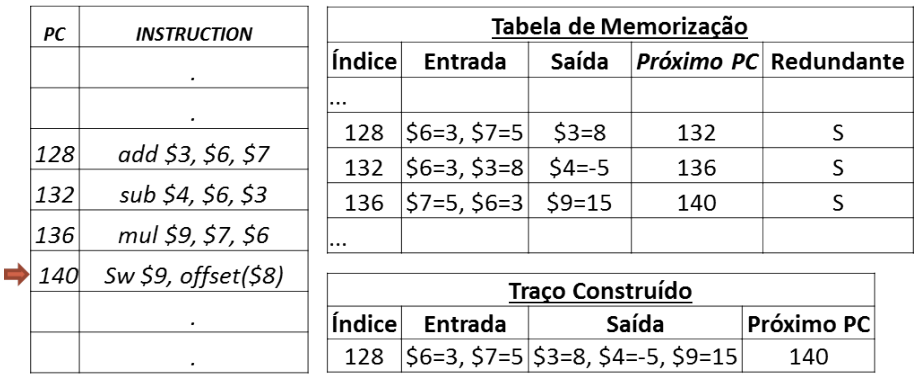


Figura 3.2: Construção de um traço.

Reúso em processadores podem ser classificados em duas categorias: (i) reúso por valor ou (ii) reúso por nome[32].

3.1.1 Reúso por valor

Esquemas de reúso por valor armazenam em tabelas de memorização os valores dos operandos de entrada e de saída. A tabela de memorização nestes esquemas é indexada pelo *Program Counter* (PC). No estado de decodificação de uma instrução, seu endereço é comparado aos endereços presentes na *cache* de reúso, se houver uma entrada na *cache* com o mesmo endereço e que possua os mesmos valores de operandos de entrada sobre os quais a instrução está executando, o valor do operando de saída é recuperado da *cache*, a instrução é ignorada, e o PC avança para próxima instrução. De forma semelhante, os valores dos operandos do contexto de entrada e saída também são armazenados em uma tabela histórica.

3.1.2 Reúso por nome

Esquemas de reúso por nome armazenam os nomes dos registradores de entrada de uma instrução dinâmica e o estado deles na tabela de reúso, ao invés do valor de

operandos. Neste esquema as tabelas também são indexadas pelo PC e instruções são buscadas na tabela no momento da decodificação. Quando uma instrução dinâmica é encontrada em uma tabela, é verificado se os seus registradores de entrada estão válidos. Essa verificação é feita através do campo de validade de registradores. Esses campos são invalidados, caso alguma outra instrução sobrescreva o registrador. No caso de traços, nesse esquema, a validade dos registradores do contexto de entrada são armazenadas e verificadas para reutilização dos mesmos.

Pode-se afirmar que para o esquema de nomes a validação de que uma instrução é redundante é mais fácil de implementar do que a verificação por valores, pois nesse caso é necessário apenas a validação de campos de 1 *bit*. Entretanto, o reúso por valor consegue expor mais instruções redundantes, pois permite que diferentes instâncias de uma mesma instrução sejam armazenadas em uma tabela.

3.2 Reúso de computação com apoio de compiladores

Técnicas de reúso de computação com apoio de compiladores[33–36] consistem em utilizar análises em tempo de compilação para identificar regiões do código, muitas vezes blocos básicos, que possuam um bom potencial de reúso. Estas análises podem ser classificadas como estáticas ou dinâmicas.

3.2.1 Análises estáticas

Análises estáticas consistem em inferir em tempo de compilação regiões que possuam uma gama de valores de entrada com pouca variação e que permitam que uma quantidade expressiva de instruções sejam ignoradas em um único *hit* na *cache* de reúso. Essas análises podem, por exemplo, deduzir um possível conjunto de valores para determinadas expressões dentro de uma região e inferir, considerando que não sejam muitos valores distintos, que essa região não possuirá muita variação, e selecioná-la para ser uma região de reúso.

3.2.2 Análises dinâmicas

Análises dinâmicas realizam simulações de valores (*value profiling*[37]) utilizando uma amostra de entradas da aplicação. Estas simulações permitem descobrir expressões invariantes e semi-invariantes que não puderam ser detectadas através de métodos estáticos. Com isso, é possível realizar especializações de regiões do código. Isto significa criar mais de uma versão de um mesmo trecho de código, porém utilizando valores fixos para expressões semi-variantes e invariantes. Desta forma, mais

regiões são formadas com menos operandos de entrada necessário para sua reutilização.

Técnicas de otimização como *loop unroling* (desenrolar de laços), junção de blocos básicos, criação de superblocos[38], *function inlining* (trocar uma chamada à função pelo seu corpo), entre outras, podem ser utilizadas para aumentar regiões de reúso e potencializar o ganho do reúso de computação por compiladores.

Em técnica de compiladores, a detecção de reúso durante a execução de uma aplicação pode ser feita com apoio de hardware ou puramente em *software*. Nos casos onde se utiliza apoio de *hardware*[33, 36], *buffers* de reúso são criados no processador. Estes são consultados em tempo de execução para verificar se todos as entradas necessárias à execução de uma determinada região estão disponíveis. Tradicionalmente, esses *buffers* possuem uma entrada por região redundante detectada, e em cada entrada n campos, cada qual com valores do conjunto de operandos de entrada da região em questão. O compilador que utiliza apoio de *hardware* gera anotações nas regiões de códigos candidatas ao reúso que são utilizadas pelo *hardware* para identificar informações relevantes ao reúso, por exemplo, quando uma região deve ser armazenada no *buffer*. Quando não há apoio de *hardware* e a detecção e reúso de regiões se dá puramente por *software*[35], regiões candidatas possuem seus códigos alterados em tempo de compilação para inserir instruções de busca em uma tabela *hash*, a qual é mantida durante a execução do programa na memória do sistema.

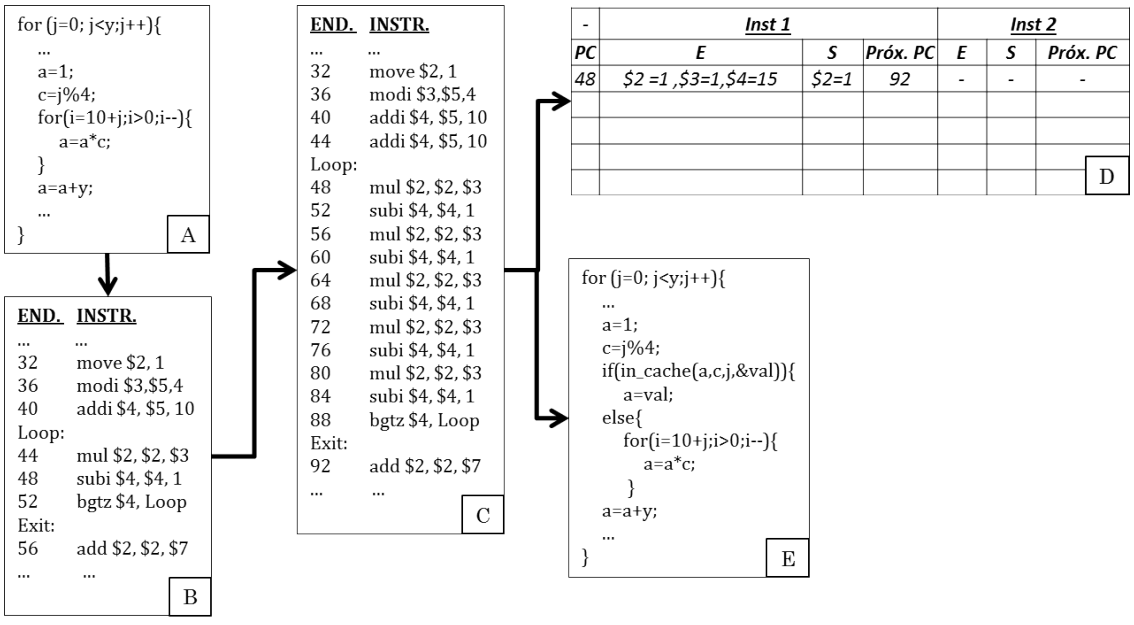


Figura 3.3: Construção de uma região de reúso.

A figura 3.3 apresenta uma forma geral de técnicas de reúso em compiladores. O quadro A apresenta o código original. O quadro B apresenta o código compilado. Neste exemplo o segmento de código selecionado para o reúso é o corpo do laço

interior. O quadro C apresenta a técnica de *loop unroling* para aumentar o tamanho do segmento candidato. O quadro D apresenta um *buffer* de reúso com capacidade de armazenar até duas instâncias da região, caso seja utilizada uma estratégia de reúso por compilador com apoio de *hardware*. Por fim, o quadro E apresenta uma solução completamente em *software*, onde o código é reescrito pelo compilador para utilizar uma função de *lookup* e recuperar o valor memorizado.

3.3 Reúso de computação em aplicações

Desenvolvimento de aplicações também podem se beneficiar de reúso de tarefas computacionais[39–41]. Conforme vimos no exemplo de Fibonacci (figura 3.1), é possível reescrever códigos utilizando técnicas de memorização. Para isso, é necessário criar um objeto que seja utilizado como uma tabela histórica, e adicionar cláusulas *if-else* para buscar o resultado de uma determinada tarefa nesta tabela.

Por conta de custos maiores na manutenção e busca em uma tabela histórica em *software*, os grãos das tarefas devem ser maiores. Logo, é muito comum que, em aplicações, funções sejam alvos de memorização. Neste caso, os argumentos das funções tornam-se os operandos de entrada necessários ao reúso. De forma semelhante, o retorno da função torna-se os operandos de saída. Da mesma forma que técnicas de reúso em *hardware* necessitam de atenção especial para operações que possuam efeito colateral, em reúso de funções também deve-se atentar para operações desta natureza. Por exemplo, funções que alterem variáveis previamente existentes à sua chamada ou escrita de arquivos que não sejam auto-contidas[39], isto é, a função escreve em um arquivo aberto previamente à sua chamada ou cria um arquivo, mas uma outra função posterior o edita.

Uma vantagem da aplicação de reúso de tarefas em *software* é a facilidade de customizar *caches*[41] de reúso para a região alvo a ser memorizada. Por exemplo, algumas regiões podem ter uma *cache* de 100 entradas e uma política LRU, enquanto outra pode ser viável ter uma política LFU com 600 entradas. Também é possível utilizar diferentes formas de indexação na tabela, por exemplo, um algoritmo de *hash md5* executado sobre as entradas da região (variáveis locais e globais à região). Essas customizações, assim como a seleção de regiões e funções candidatas ao reúso, podem ser feitas utilizando simulações da aplicação.

A linguagem de programação *Python* oferece um artifício para facilitar o uso de memorização de tarefas chamado *Decorator* [42]. *Decorators* permitem que funções sejam alteradas de forma genérica sem necessidade de uma nova codificação, basta realizar uma anotação imediatamente antes da função utilizando a sintaxe $@ < Nome\_do\_Decorator >$. A figura 3.4 apresenta um exemplo da função de Fibonacci com uso de *decorators*. Note que apesar da função ser codificada de forma

tradicional, ela será executada através da função *Decorator* por ter sido anotada sobre sua declaração a chamada *@memorizar*. Além de funções como *Decorator*, classes também podem ser utilizadas dessa forma deixando o comportamento de um *Decorator* mais robusto.

```
#Declaração do Decorator
def memoize(f): #Função normal fib sera passada como parâmetro
    tab_memo = {}
    def consulta_cache(val):
        #Caso ocorra um miss a função fib tradicional é invocada
        if val not in tab_memo:
            tab_memo[val] = f(val)
        #Caso ocorra um hit, o resultado é recuperado da tabela.
        return tab_memo[val]
    return consulta_cache

@memoize #Assinatura que modifica a função fib a ser memorizada
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figura 3.4: Função recursiva de Fibonacci utilizando um *Decorator*.

Capítulo 4

Trabalhos Relacionados

Neste capítulo discutimos trabalhos anteriores pertinentes ao assunto desta pesquisa. Os trabalhos relacionados são divididos em duas seções: Modelo *Dataflow* e Reuso de computação.

4.1 Modelo *Dataflow*

O modelo de execução *Dataflow*[11, 13, 43–45, 45] tem levantado fortes interesses no meio acadêmico por sua capacidade de explorar o paralelismo natural de aplicações. Algo que tornou-se extremamente necessário no nicho de computação por consequência da tendência natural de fabricantes de microprocessadores migrar de um modelo sequencial (*chips single-core*) para um design paralelo (*chips multicore*). Nesta subseção dissertamos sobre trabalhos que utilizam modelo *Dataflow* para explorar paralelismo.

4.1.1 *Manchester Machine*

Os autores propõe neste trabalho a criação de um *hardware Dataflow* conhecido como *Manchester Machine* [11]. Este *hardware* foi desenhado para permitir a execução de uma aplicação utilizando-se de um grafo *dataflow* dinâmico DDG. Neste hardware, por conta do paradigma *Dataflow*, não há necessidade de referências à uma memória global, as instruções são vistas como operações e os seus operandos são enviados e recebidos através de trocas de mensagens. Os dados são enviados como pacote de dados chamados *Tokens*. Essas mensagens são marcadas com uma *tag*, o que permite a implementação de laços no grafo *dataflow*, pois impede que operando referentes à diferentes iterações se misturem durante a execução. Os desvios são implementados através de instruções de *switch*, as quais recebem como operandos o resultado de uma instrução e um booleano, dependendo do valor do booleano, o operando é encaminhado para um dos dois caminhos no grafo.

O elemento processador da *Manchester Machine* utiliza uma estrutura de *pipeline* anelar. Esta possui os seguintes módulos: *Token Queue*, *Matching Unit*, *Overflow Unit*, *Instruction Store*, *Processing Unit* e *I/O Switch*. Os *tokens* são enviados nessa estrutura anelar a partir da *Token Queue* para a *Matching Unit*. Nesta unidade, *tokens* que são referentes à mesma instrução são colocados em pares. Caso a aplicação tenha muitos dados, se faz necessário utilizar a *Overflow Unit* junto com a *Matching Unit*. Quando uma instrução possui todos os *tokens* necessários para sua execução, ela é recuperada da *Instruction Store*, e então enviada junto com seus operandos de entrada para a *Processing Unit*, onde ela será processada. O resultado da instrução é enviada para a *Token Queue* reiniciando o processo. A unidade *I/O Switch* permite comunicação com componentes periféricos da máquina.

A linguagem utilizada para criação de programas *dataflow* é a SISAL (*Streams and Iteration in a Single Assignment Language*), uma linguagem *dataflow* semelhante ao Pascal. A linguagem *assembly* da máquina construída a partir do compilador do SISAL é chamada de TASS (*Template Assembly language*). O produto final da compilação de um programa produz um código e arquivo de dados que representam a aplicação a ser executada na máquina. Cada instrução do código possui um *opcode* e um endereço de destino para o resultado da instrução. Além disso, a instrução possui mais um campo que pode ser utilizado como um segundo endereço de destino ou um operando literal. Caso deseje-se enviar o resultado de uma instrução para mais de dois destinos deve-se utilizar a instrução especial DUP.

4.1.2 *WaveScalar*

WaveScalar[23] é uma arquitetura de conjuntos de instruções e um modelo *Dataflow* que permite a execução de programas reais escritos em linguagens imperativas. A *WaveScalar* utiliza um binário, que é basicamente um grafo *dataflow*, criado a partir de um compilador. Este compilador é responsável por realizar a divisão do grafo *dataflow* em ondas (*waves*). Ondas são subgrafos direcionados e acíclicos pertencentes ao grafo *dataflow*. Além disso, o compilador preenche instruções de memória com um sequencial, que serve para que uma determinada instrução de memória identifique no grafo qual a instrução de memória é sua predecessora e qual é sua sucessora. Isso permite que operações de memória sejam executadas em ordem, permitindo, assim, compatibilidade com linguagens imperativas tradicionais. Ambiguidades na ordenação de operações em memória são resolvidas pelo compilador através da inserção de instruções *MEMORY-NOP*. Por fim, o compilador converte desvios no fluxo de controle, instruções *if-else*, em instruções denominadas *steers*, que servem para propagar os operandos de acordo com o desvio tomado no grafo *dataflow*.

O binário da *WaveScalar* fica armazenado na memória principal. Além do grafo

de controle, ele possui um conjunto de instruções que são enviadas da memória ao processador *WaveScalar*, chamado *WaveCache*. O *WaveCache* é um *grid* de aproximadamente 2000 elementos processadores (EPs) organizados em *clusters* de 16 elementos. Cada EP consegue armazenar até 8 instruções para execução. Instruções nesses EPs são executadas seguindo o paradigma *Dataflow*, isto é, assim que seus operandos de entrada estiverem disponíveis. Os operandos resultantes são propagados às instruções dependentes. Estas podem estar em EPs remotos, em outro *cluster*, em EPs locais, no mesmo *cluster*, ou até mesmo no mesmo EP em que a instrução predecessora executou. Requisições de acesso à memória principal são enviadas pelos EPs aos *StoreBuffers* locais ou remotos. As requisições de acessos à memória são inseridas em uma lista de requisições pertencente a uma determinada onda e são executadas obedecendo o mecanismo de ordenação de memória especificado no grafo *dataflow*.

A arquitetura *WaveScalar* utiliza mecanismo de *tags* para identificar operandos de iterações distintas, no caso de execuções de laços em paralelo representados por ondas dinâmicas distintas. Isso é feito utilizando instruções *WAVE-ADVANCE*, as quais são responsáveis por receber um operando de entrada, incrementar o número da onda, e propagar o valor de entrada com o novo número da onda.

4.1.3 *Talm* e *Trebuchet*

Em [3, 46], os autores propuseram um modelo arquitetônico e uma linguagem para execução de instruções utilizando o modelo de execução guiado pelo fluxo de dados (*Dataflow*). O modelo *Talm* foi idealizado para funcionar no contexto de uma máquina tradicional de Von Neumann, ou seja, este modelo permite a criação de uma máquina virtual, simulador ou um ambiente de execução *Dataflow* em máquinas tradicionais.

A arquitetura base conceitual do modelo, a qual é agnóstica à arquitetura subextrato na qual o modelo *Talm* será implementado, possui os seguintes componentes:

- EP - EPs são elementos processadores responsáveis por executar instruções e super-instruções.
- *Buffer* de comunicação - Cada EP possui um *buffer* de comunicação. Neste *buffer* são armazenados operandos resultantes de execuções de instruções pelos demais EPs.
- Lista de instruções - Este componente está presente em cada EP, ele guarda a listagem de instruções da linguagem de montagem do *Talm* que deverão ser executadas pelo EP hospedeiro da lista de instruções.

- Fila de prontos - Cada instrução possui uma lista de operandos necessários para sua execução. Quando esses operandos são casados com uma determinada instrução, eles (operandos e instrução) são enviados para fila de prontos para execução.

A arquitetura proposta no *Talm* é composta por vários EPs idênticos, o que a torna uma arquitetura capaz de realizar execução de forma distribuída, ou seja, sem um ponto centralizador de controle. Por conta disso, a finalização da execução de uma aplicação no *Talm* é detectada por um algoritmo distribuído para detecção de terminação global.

Além da arquitetura distribuída, o *Talm* também provê uma linguagem de montagem e conjunto de instruções próprias. Estes permitem ao modelo proposto implementar grafos *dataflow* mais complexos que tenham em sua composição, por exemplo, desvios condicionais, laços de repetição dinâmicos, chamadas de funções e super-instruções.

Um grande diferencial no *Talm* é a liberdade dada ao programador para definir super-instruções e, assim, prover um ambiente de execução *Dataflow* de granularidade grossa que executa super-instruções em uma máquina Von Neumann de forma paralela. O programador especifica nas super-instruções, além da computação realizada por ela, os operandos de entrada, de saída e as interdependências entre as super-instruções criadas. As super-instruções do *Talm* seriam análogas aos blocos e *kernels* de [43] e [47] respectivamente. Porém, ao contrário destes, as super-instruções são distribuídas entre os EPs exclusivamente pelo montador do *Talm*, enquanto que nas demais existe a necessidade do programador definir onde as tarefas irão executar inicialmente.

Em [28, 46], os autores implementam o modelo *Talm* para máquinas em um paradigma de memória compartilhada. Por conta disso, neste trabalho, a comunicação entre os diferentes *cores* é realizada através de operações de leitura e escrita em áreas comuns de memória. Portanto, a passagem de operandos entre os EPs se dá por essas operações de escrita e leitura. Quando os operandos são consumidos e produzidos pelo mesmo EP, o acesso pode ser feito diretamente às estruturas armazenadoras de operandos. Desta forma, é importante que o máximo de EPs sejam utilizados para aumentar o paralelismo, e que tarefas dependentes permaneçam no mesmo EP ou em EPs próximos (que compartilhem uma *cache*) para aproveitar a localidade de valor e minimizar o custo de comunicação.

Os autores implementaram a fixação das *threads* que representavam os EPs em núcleos específicos para melhor aproveitamento da localidade das estruturas de dados nas *caches*, como também, um mecanismo simples de roubo de tarefas [48] para suavizar o problema de desbalanceamento de carga existente em aplicações paralelas.

O mecanismo funciona da seguinte forma: um EP ocioso busca aleatoriamente por tarefas na fila de tarefas prontas de EPs vizinhos.

Uma aplicação paralela para ser executada na *Trebuchet* deve ser compilada para linguagem de montagem *Dataflow* do *Talm*. As instruções geradas são distribuídas pelos EPs, e executadas em paralelo se forem independentes e houver EPs disponíveis para executá-las. Por conta da *Trebuchet*, ser uma máquina virtual, o custo de toda aplicação ser transformada em um grafo *dataflow* torna-se oneroso demais. Por conta disso, blocos de instruções Von Neumann são transformados em super-instruções. Estas são executadas pela *Trebuchet* utilizando o modelo *Dataflow*.

4.1.4 *Auto-Pipe*

Em [43], os autores propuseram um conjunto ferramental (*Auto-pipe*) para auxiliar no planejamento, avaliação e implementação de aplicações que podem ser executadas em *pipelines* computacionais (ou em outras topologias) usando um conjunto de componentes heterogêneo. Atualmente, *Auto-pipe* é compatível com componentes processadores GPUs CUDA, FPGAs e processadores *multicore*.

A linguagem de alto nível "X", é uma linguagem *dataflow* que serve para mapear tarefas de granularidade grossa em nós interligados em uma topologia. Estes nós são determinados na linguagem X como blocos. Estes podem obedecer, se especificado pelo programador, uma estrutura hierárquica. A comunicação entre os blocos, ou seja, a propagação de operandos entre blocos, é agnóstica ao componente no qual o bloco está executando. A codificação do bloco em si deve ser realizada em C, VHDL ou CUDA.

O desenvolvimento de uma aplicação em *auto-pipe* é descrita em 4 fases:

- (*i*) Implementar o algoritmo com a linguagem X sem se preocupar com problemas de desempenho. O objetivo é capturar aspectos do *pipeline* e estruturas paralelas que acreditem ser úteis, e, então, executá-lo provendo verificação funcional.

- (*ii*) Designar blocos aos dispositivos genéricos processadores, podendo estes serem processadores *multicores* ou *hardwares* reconfiguráveis. Neste passo, simulações são realizadas.

- (*iii*) Especificar os dispositivos genéricos em componentes reais e realizar algumas otimizações.

- (*iv*) Realizar a execução propriamente dita sobre os dispositivos alocados.

4.1.5 DAGuE

Em [47] os autores apresentam um *framework* para execução de tarefas em arquiteturas heterogêneas utilizando o modelo *Dataflow*. Ao contrário do *auto-pipe*, a arquitetura heterogênea não contempla aceleradores FPGAs e GPUs. Neste *framework*, tarefas são especificadas como *kernels* sequenciais. A aplicação é descrita como um grafo *dataflow* direcionado e acíclico (DAG), onde os vértices são os *kernels*, e as arestas representam a dependência de dados entre eles. O DAG é representado em um formato conhecido como JDF. Este é pré-compilado como um código C e associado à biblioteca do DAGuE.

No DAGuE o programador é responsável por especificar no formato JDF as tarefas, a distribuição delas na plataforma de execução, um *cluster*, por exemplo, e os dados iniciais para a execução da aplicação. A partir deste ponto, responsabilidades, tais como: onde as demais tarefas serão executadas e o balanceamento de carga entre as *threads*, como também, a comunicação e resolução de dependência entre tarefas ficam ao encargo do motor de execução e comunicação do DAGuE.

A comunicação neste *framework* é realizada de forma assíncrona para poder realizar sobreposição de comunicação e computação, permitindo desta forma redução destes custos. Para isso, existe uma *thread* responsável pelo escalonamento de uma tarefa, esta é fixada a um determinado *core*, e outra *thread* que irá propagar os operandos resultantes de uma determinada tarefa aos demais nós. A troca de mensagens entre os nós é realizada utilizando MPI. O processamento de mensagens é realizado com um paradigma de produtor/consumidor, onde a *thread* executora produz operandos que ficam armazenados em uma fila em forma de mensagem, e a *thread* de comunicação irá consumir dessa fila distribuindo as mensagens de operandos, quando possível.

Cada *thread* escalonadora possui duas filas para armazenar as tarefas prontas, a fila privada se chama *placeholder* e a compartilhada se chama *waiting queue*. Quando uma tarefa, após executada, permite que uma outra tarefa dependente dela execute, esta tarefa que será executada é colocada preferencialmente no *placeholder*. Como a tarefa filha, por estar nesta fila, será executada pela mesma *thread* que executou a tarefa pai, isso maximizará a localidade da memória e *cache*. Caso o *placeholder* esteja cheio, a tarefa é colocada na *waiting queue*. Uma *thread* escalonadora primeiramente irá buscar uma tarefa para execução em sua *placeholder*, se esta estiver vazia, ela buscará em sua *waiting queue*. Se, por sua vez, sua *waiting queue* estiver vazia, ela buscará tarefas nas *waiting queues* de outras *threads*, mas não de forma aleatória como em [28], a fila escolhida para o roubo de tarefas leva em consideração a distância física entre os *cores*.

4.1.6 Teraflux

O projeto Teraflux [4, 45], custeado pela união europeia, consiste em estudar o modelo *Dataflow* de forma a aplicá-lo como solução em diversos desafios presentes no paradigma de programação paralela em *teradevices*, sistemas computacionais com mais de 1000 *cores* heterogêneos. O espectro de abordagem desse projeto começa em um alto nível da computação, modelos de programação que suportam execução *Dataflow*, até o nível mais baixo de computação, uma arquitetura capaz de escalonar tarefas utilizando a disponibilidade de dados como guia de execução. O paradigma *Dataflow* é utilizado então para endereçar três desafios no uso de *teradevices*: programabilidade, confiabilidade e criar um modelo arquitetural compatível com execução em *Dataflow*.

No quesito de programabilidade, os autores apresentam soluções relacionadas ao modelo de programação *Dataflow* e compilador para programação paralela em *Dataflow*. Os autores apresentam versões modificadas de algoritmos de compilação que visam remover sincronizações implícitas por alterações em ordem no código fonte, aumentando assim o paralelismo. No nível de modelos de programação, os autores implementam o *OmpSs* que é uma extensão do *OpenMP*[9] que permite a programação baseada em tarefas. Nesta extensão, o programador especifica tarefas e as suas dependências de dados, o grafo de tarefas é criado dinamicamente e tarefas independentes são executadas de forma paralela conforme disponibilidade de dados. Outra extensão do *OpenMP* implementada é o *OpenStream*, que permite a programação de grafos *dataflow* utilizando diretrizes que utilizam *streaming* como forma de transferência de operandos entre as tarefas. Os autores também apresentam o DFScala: um módulo da linguagem funcional Scala que permite a construção de grafos *dataflow* de forma dinâmica durante a execução do programa. Por fim, são apresentados os *codelets*, que são conjuntos de instruções Von Neumann em sequência. Esses *codelets* são unidades atômicas que são entregues para unidades de computação. Eles são interligados considerando suas interdependências, provendo assim, um modelo de execução *Dataflow* de grão fino e hibrido (*Dataflow* combinado com Von Neumann).

Em termos de confiabilidade, os autores estudam a criação na arquitetura do *hardware Dataflow* Teraflux as unidades hierárquicas FDU *Fault Detection Unit*. Essas unidades são diferenciadas em D-FDU, que detectam falhas ao nível do nó, e L-FDU, que detectam falhas ao nível do *core*. Para detecção de erros na rede, são utilizadas mensagens de *heartbeats*. As informações sobre a saúde do nó são enviadas para camada de SO.

A arquitetura do Teraflux é baseada em modelos de execução *Dataflow* que executam tarefas de grão fino e grosso. A arquitetura também é estudada para

implementar suporte à transações para que mais aplicações, que não utilizem *threads Dataflow* e que alterem estruturas compartilhadas, sejam executadas nesta arquitetura. Uma modificação da ISA x86-64 foi estendida para permitir a execução de *threads Dataflow* denominada T-. Os pontos fundamentais desta extensão são: (*i*) habilitação de execução de *threads* com base no fluxo de dados da aplicação. (*ii*) A execução de uma *thread Dataflow* é escalonada por um componente externo ao núcleo do *chip* chamado DTS (*Distributed Thread Scheduler*). (*iii*) É possível utilizar diferentes tipos de memória.

4.1.7 TBB

TBB [1] é uma biblioteca C++ que permite a implementação de programas paralelos utilizando um modelo de programação baseado na decomposição da aplicação em blocos de tarefas, ao invés do modelo mais tradicional de especificação de *threads* para execução paralela. Com a TBB, o escalonamento das tarefas em *threads* se torna obrigação do *engine* da TBB, que realiza esta tarefa de forma a minimizar custos de comunicação, melhorar a utilização de *caches* e permitir um melhor balanceamento de carga entre *threads*. TBB possui compatibilidade com a maioria de processadores e compiladores C++, não necessitando de nenhum componente especial no *hardware* ou *software* para sua utilização.

Na biblioteca TBB, existe um módulo chamado *flow graph* [49] que permite a criação de programas paralelos utilizando um grafo *dataflow*. O programador cria blocos de tarefas (nós do grafo) e a comunicação entre esses blocos é representada de acordo com a dependência de dados entre eles (arestas do grafo). A comunicação é baseada na troca de *Tokens* entre as tarefas. A biblioteca da TBB escalona essas tarefas quando elas possuem todos os dados disponíveis para execução, permitindo a exploração do paralelismo implícito no grafo. Ela também permite explorar o paralelismo em dois níveis, por exemplo duas tarefas em um grafo *dataflow*, podem possuir em seu corpo interno um laço paralelizado por uma diretiva *OpenMP*[9]. O grafo *Dataflow* TBB pode ser composto por diferentes tipos de nós, semelhante à solução proposta na Sucuri[6], porém, ao contrário da Sucuri, a TBB não permite a criação de grafos no formato de *pipelines* lineares, onde um bloco de tarefa recebe operandos de mais de um bloco de tarefas precedentes no caminho do grafo.

4.1.8 *OmpSs*

OmpSs [2] é uma extensão do *OpenMP* para permitir exploração de paralelismo assíncronos nas aplicações e heterogenidade, isto é, permitir a execução da aplicação de forma paralela em diferentes tipos de *hardwares*, por exemplo, GPUs. O paralelismo assíncrono é possível graças a especificação de dependência de dados entre as

tarefas.

No *OmpSs* é possível utilizar um construtor chamado *Task*, o qual permite que blocos estruturados ou funções sejam interpretados como ponto de criação de tarefas, isto é, quando estes forem invocados, uma tarefa será instanciada. Esses construtores possuem parâmetros *in*, *out* e *inout* para especificar a dependência de dados entre as tarefas. Quando uma tarefa é instanciada, é verificado se suas dependências de entrada e saída se casam com as dependências de outras tarefas instanciadas, se uma dependência do tipo *RaW (Read after Write)*, *WaW (Write after Write)* ou *WaR* (*Write after Read*) for encontrada, a tarefa se torna uma sucessora dessas outras tarefas. Este processo cria o grafo de forma dinâmica, ao contrário da TBB que especifica o grafo de forma completa antes de executá-lo.

4.1.9 *Maxeler*

A *Maxeler*[12] é uma empresa que produz aceleradores *Dataflow* utilizando *hardwares* reprogramáveis FPGAs[50]. Estes *hardwares* são classificados em duas séries distintas de produtos: MPC-C e MPC-X. As séries MPC-C são nós computacionais que possuem um processador tradicional X86 conectado diretamente ao DFE (*Dataflow Engine*) via um barramento *PCI-EXPRESS*. As séries MPC-X são nós computacionais *Dataflow* que não possuem um processador X86, mas podem se conectar à CPUs em nós tradicionais via *Infiniband*. DFEs em um mesmo nó possuem uma rede de intercomunicação denominada *MaxRing*, a qual permite comunicação rápida entre os DFEs vizinhos.

Aplicações *Dataflow* que executam nos *hardwares* da *Maxeler* são desenvolvidas em JAVA combinada com uma extensão desenvolvida pela própria *Maxeler* que é responsável por direcionar o desenvolvimento do grafo *dataflow*. O desenvolvimento de aplicações são basicamente decompostas em duas etapas: desenvolvimento de *kernels* e um *manager* programado em JAVA. *Kernels* são caminhos de dados no *hardware* que implementam operações lógicas e aritméticas necessárias ao algoritmo. *Kernels* descrevem um grafo *dataflow* direcionado, acíclico e sem desvios através da definição de fluxos de entrada e saída, e conectando estes fluxos às operações necessárias. O compilador transforma estes *kernels* em operações de *hardware* que em conjunto formam um *hardware dataflow*. O *manager* descreve a lógica que dita a propagação de dados entre os *kernels* ou dispositivos de entrada e saída externos aos DFEs.

Nos *hardwares Maxeler*, aplicações são vistas como fluxos de dados que atravessam um *chip dataflow* que possui componentes que fazem operações necessárias ao algoritmo. O grafo *dataflow* está impresso no *hardware* e operações são realizadas conforme operandos chegam às unidades lógicas ou aritméticas presente no

chip. Isso elimina a necessidade de *caches* tradicionais para manter dados próximos aos elementos computacionais. Também elimina etapas de captura de instrução e decodificação da mesma e, por conta dos grafos não possuírem desvios, permite que as execuções de aplicações sejam feitas utilizando *pipelines* profundos. Além disso, o gerenciador de recursos *MaxelerOS* permite que mais de uma aplicação utilize o mesmo DFE, aumentando a concorrência de aplicações *dataflow* distintas no *hardware*.

4.1.10 Sucuri

Sucuri[6, 20, 51] é uma biblioteca que permite a execução de programas *Python* em um modelo *Dataflow*. Ela permite ao programador descrever suas aplicações como grafos *dataflow*, onde objetos *Nodes* recebem tarefas (funções *Python*) e são interligados a outros objetos *Nodes* via arestas. As tarefas são processadas por *workers* paralelos e a execução do grafo é orquestrada por um escalonador central. A Sucuri é descrita em detalhes no capítulo 5, pois foi a biblioteca utilizada para a implementação da técnica DF-DTM.

4.2 Reúso de computação

4.2.1 DTM - *Dynamic Trace Memoization*

DTM[5] ou *Dynamic Trace Memoization* é uma técnica de reuso que utiliza tabelas de memorização para reaproveitar resultados de instruções ou sequências de instruções redundantes (*traços)*.

Os principais aspectos para se considerar no uso de DTM são: (i) Como instruções redundantes são identificadas. (ii) Como instruções redundantes são utilizadas para construir traços redundantes. (iii) Como traços são identificados. (iv) Como instruções e traços redundantes são reutilizados.

(i) - Identificação de instruções redundantes

Este é o passo inicial. Consiste em verificar se uma instrução dinâmica instanciada utiliza os mesmos operandos das instâncias anteriores dessa instrução. Para isso, a DTM utiliza uma tabela associativa global de memorização, "*Global Memory Table*" ou "*Memo Table G*" (Tabela G). Cada entrada da tabela G serve para guardar informações associadas a uma instrução dinâmica. Ela possui os seguintes campos:

- PC: Serve para identificar o endereço da instrução estática.
- SV1, SV2: Guarda os dois operandos da instrução dinâmica.

- Res/Targ: Guarda o resultado de uma instrução aritmética ou do endereço alvo de um desvio.
- *Bit brc*: Identifica se a instrução é um desvio condicional (*branch*).
- *Bit jmp*: Identifica se a instrução é um desvio incondicional (*jump*).
- *Btaken*: Identifica se o desvio condicional foi tomado.

É importante ressaltar que a tabela G não armazena instruções de *load ou store*, pois estas não são consideradas como instruções válidas para memorização. Instruções dinâmicas válidas têm seus operandos e endereços comparados associativamente com os campos PC, SV1 e SV2 das entradas da tabela G. Se não ocorrer um *hit* nesta tabela, a instrução é identificada como não-redundante, e uma entrada na tabela G é alocada para ela com os campos PC, SV1 e SV2 preenchidos com os valores da instrução dinâmica. Se ocorrer um *hit*, a instrução é marcada como redundante, mas não é alocada uma nova entrada da tabela G.

(*ii*) - Construção de traços

A construção de um determinado traço consiste em adicionar valores aos contextos de entrada e saída deste para cada instrução redundante encontrada. Contexto de entrada é definido como o conjunto de operandos que instruções internas ao traço utilizam, mas que são produzidas por instruções externas a ele. Contexto de saída é definido como o conjunto de operandos produzidos pelo traço e utilizados por instruções externas a ele.

As estruturas de *hardware Input/Output map*, *buffers T#* e *Memo Table T* (tabela T) são utilizadas para construção de traços.

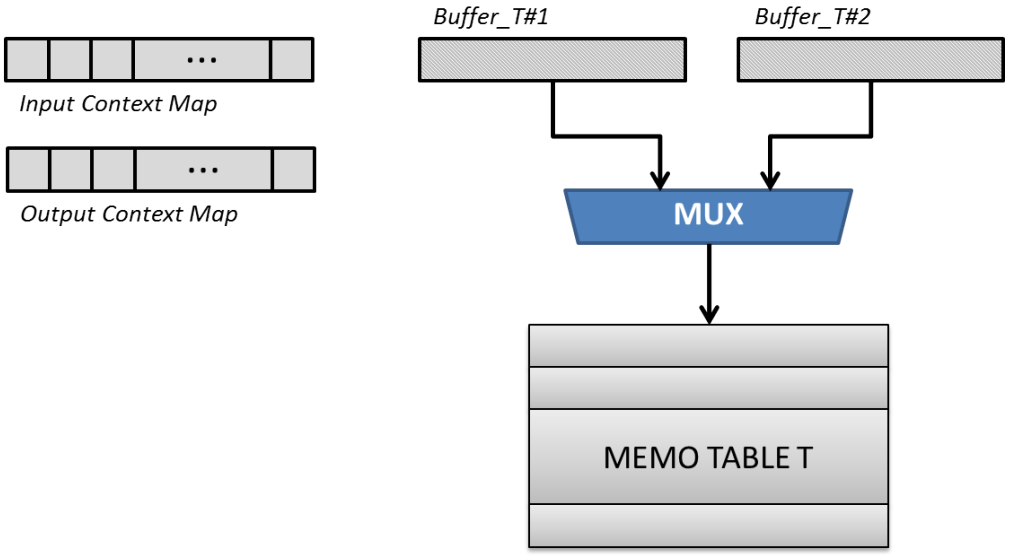
- *Input/Output map*: Um mapa que utiliza 1 bit por registrador presente na máquina para fins de identificação destes. Quando um registrador é identificado como fonte ou saída de um elemento do contexto de entrada ou saída, respectivamente, o *bit* referente àquele registrador é ativado. Isso faz com que o conteúdo deste seja armazenado em um dos dois *buffers T#*.
- *Buffer T#1 e T#2*: Estes *buffers* armazenam temporariamente informações sobre o traço sendo construído. Eles permitem que dois traços sejam construídos simultaneamente. Quando o traço estiver concluído, ele é movido do *buffer* para a tabela associativa T.
- Tabela T: Tabela que armazena os dados do traço construído.

A tabela T possui os seguintes campos:

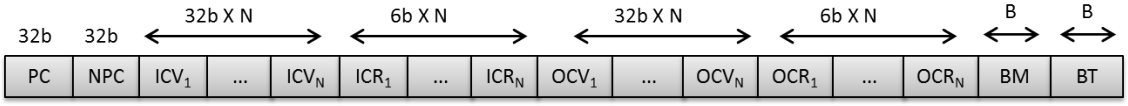
- PC: Armazena o endereço da primeira instrução do traço.
- NPC: Armazena o endereço das próximas instruções após execução do traço. Nele está contemplado todos os desvios de controle ocasionados por instruções dentro do traço.
- ICV: Composto por N campos que guardam o conteúdo dos operandos de entrada do traço.
- ICR: Composto por N campos que identificam os registradores que armazenam os valores dos campos ICV.
- OCV: Composto por N campos que guardam o conteúdo dos operandos de saída do traço.
- OCR: Composto por N campos que identificam os registradores que armazenam os valores dos campos OCV.
- *Bmask*: Os *bits* deste campo identificam uma instrução de *branch* dentro do traço.
- *Btaken*: Indica se o desvio correspondente no campo *bmask* foi tomado.

A figura 4.1a apresenta os componentes necessários para construção de um traço. A construção de um traço é iniciada, quando uma primeira instrução dinâmica é identificada como redundante. Os registradores de entrada referentes à primeira instrução e os referentes às das instruções redundantes subsequentes são identificados utilizando o mapa de *Input*. Estes registradores irão compor o contexto de entrada do traço. De forma semelhante, registradores de saída das instruções redundantes que são adicionados aos traços, os quais irão compor o contexto de saída do traço, são identificados através do mapa de *output*. Quando os *bits* nos mapas de *Input/Output* são acionados, os valores dos registradores correspondentes são adicionados ao contexto de entrada/saída do traço em construção. Informações sobre o traço em construção são armazenadas em um dos *buffers T#*. A existência de dois *buffers* permite que um novo traço possa ter sua construção inicializada, ainda que o traço anterior não tenha finalizado. A construção do traço termina quando um dos seguintes acontecimentos ocorre: (*i*) Uma instrução não-redundante é encontrada. (*ii*) O número de elementos dos contextos atingiu o limite N. (*iii*) O número de desvios dentro do traço alcançou um limite B. Quando uma dessas três condições ocorre, as informações do traço são passadas do *buffer T#* para uma entrada na tabela T.

A figura 4.1b apresenta os campos da tabela T. Como pode-se ver são necessários $64 + 76 \times N + 2 \times B$ *bits* por entrada da tabela T, onde N é o número de elementos



(a) Componentes para construção e reutilização de traços.



(b) Campos da tabela "*Memo Table T*".

Figura 4.1: *Hardware* da DTM para reúso de traços.

permitidos por contexto de entrada ou saída e B é a quantidade de desvios permitidos dentro do traço.

(iii) - Identificação de traços

A detecção de traços redundantes é feita da seguinte forma: cada instrução dinâmica tem seu valor de PC comparado ao campo PC da tabela T. Para as instruções que ocorrem *match* entre PCs da tabela e PCs das instruções, os registradores indicados pelos campos ICR válidos têm seus conteúdos comparados com os valores armazenados nos campos ICV. O traço é redundante, se os valores atuais dos registradores são iguais aos do contexto de entrada de uma instância do traço previamente armazenada na tabela T. Em [30], os autores propuseram um mecanismo de reúso especulativo de traços (DTM especulativa), o que permite que traços sejam reutilizados, ainda que não haja todos os valores presentes para o contexto de entrada do mesmo. Isso insere custos de mecanismos de *rollback*, caso a especulação seja revelada errada, porém a especulação pode potencializar os ganhos com reúso de mais traços.

(iv) - Reutilização de traços e instruções

A decisão de utilizar um traço redundante ou uma instrução redundante é baseada nos *hits* que ocorrem quando o PC de uma instrução dinâmica faz um *match* com o PC de uma entrada da tabela G ou da tabela T. Se o *hit* for na tabela G, então a instrução é reutilizada, e o resultado dela é recuperado do campo *Res/Targ*. Se o *hit* ocorrer na tabela T ou na tabela T e G simultaneamente, o traço é reutilizado. Nesse caso, o contexto de saída (valores no campo OCV) do traço redundante é escrito nos registradores indicados pelos campos OCR. Além disso, o PC da máquina é atualizado com o valor do campo NPC para executar a instrução posterior ao traço. Por fim, o estado de predição de desvios é atualizado com os valores do campo *btaken*, caso haja instruções de desvios internas ao traço.

4.2.2 GPU@DTM

GPUs CUDA são compostas por processadores de *Stream* (SMs). Cada SMP possui 32 CUDA *cores*. As CUDA *threads* são alocadas aos *cores* de um determinado SM em conjuntos de 32 *threads* chamados *warps*.

A GPU@DTM[31] aplica as técnicas de reúso de instruções e traços utilizadas pela DTM (seção 4.2.1) à GPUs. Ao contrário da DTM tradicional, que possui uma tabela G e T centralizada, a GPU@DTM utiliza uma tabela T e G por *cuda core*. Nesta implementação, as tabelas G tiveram suas entradas modificadas para serem adaptadas à arquitetura da GPU. Como em GPUs não existe mecanismo de previsão de desvios, o campo *btaken* da tabela G original foi removido. A esta tabela foram adicionados os campos: *uid*, o qual identifica a *thread* que executou a instrução, e *sv3*, que permite que instruções com 3 operandos de entrada ou 2 operandos e um predicativo sejam memorizadas. O campo *uid* é importante, pois permite que um determinado traço seja criado apenas por instruções válidas de uma mesma *thread*.

Na GPU@DTM, são classificadas como instruções inválidas: (*i*) instruções de acesso à memória, (*ii*) instruções de sincronização, (*iii*) instruções de textura e superfície e (*iv*) instruções de ponto flutuante. A tabela T não necessitou de inserções de campo, somente remoção dos campos referentes à previsão de desvios.

Por conta de GPUs serem processadores massivamente paralelos, a GPU@DTM apresenta conceitos de reutilização de traços e instruções *inter* e *intra threads*. Reutilização de instruções podem ocorrer por *threads* distintas (reúso *inter-thread*) ou por uma mesma *thread* (reúso *intra-thread*). Essa informação é especialmente relevante no processo de construção de traços, pois a construção de um traço pode ser interrompida, caso ocorra um reúso *inter-thread*. Por exemplo, suponha que a *thread a* identificou uma instrução redundante i_1 e começou a construção de um traço. Suponha ainda que esta mesma *thread* identificou as instruções i_2 e i_3 sucessoras à

i_1 como redundantes. Desta forma, temos que a *thread a* está construindo um traço t_1 composto pelas instruções i_1, i_2 e i_3. Se, por conta de uma troca de *warps*, uma outra *thread* descobrir a instrução i_4 como redundante, a construção do traço t_1 é finalizada. Isto é necessário para permitir que a consistência do estado do contexto da *thread* que utilizará um determinado traço seja preservada. Se qualquer *thread* detectar uma instrução inválida ou não-redundante, a construção do traço também é finalizada.

Traços e instruções são reutilizados de forma análoga à DTM tradicional. O campo PC é utilizado como indexador das tabelas T e G. Caso, todos os valores do contexto de entrada de um traço estiverem presentes nos registradores, no momento em que o PC de uma instrução é comparada ao PC da tabela T, o traço é reutilizado. Caso não ocorra o reúso do traço, é verificado, na tabela G, se a instrução está presente na mesma e se todos os operandos de entrada dela são iguais aos da entrada na tabela. Se isso ocorrer, a instrução é reutilizada, se não ocorrer ela é alocada em uma nova entrada da tabela G.

4.2.3 Esquemas de reúso de instruções dinâmicas

Neste trabalho os autores apresentam três esquemas distintos para o reúso de instruções por *hardware* em um processador superescalar tradicional. Estes três esquemas são: (*i*) S_v, (*ii*) S_n e (*iii*) S_{n+d}[32, 52]. Nestes três esquemas é utilizado um *buffer* chamado *Reuse Buffer* (RB). O RB permite que instruções sejam reutilizadas pontualmente. O esquema S_{n+d} permite que cadeia de instruções (traços) seja reutilizada. Para isso é adicionado outro componente de *hardware* denominado *Register Source Table* (RST). A seguir é apresentado em mais detalhes estes três esquemas.

S_v: Reúso por valor

Neste esquema é apresentada uma técnica de reúso de instruções por valores de operandos. Ela utiliza um RB que armazena os valores dos operandos de entrada de cada instrução executada, excetuando-se instruções de *store*. A etapa de cálculo de endereços em *loads* e stores é memorizada e reutilizada de forma normal. A etapa de leitura de memória (*load*) é realizada mediante a validade do campo *memvalid* do RB. Este campo é marcado como inválido, caso um o valor seja gravado no mesmo endereço de memória do qual o *load* é feito. Os demais campos presentes na RB são apresentados na listagem abaixo:

- *tag*: Armazena parte do valor do PC da instrução. Este é o indexador da tabela.
- *Op1 e Op2*: Armazena os operandos de entrada da instrução.

- *address*: Armazena endereço de memória para instruções *load*.
- *result*: Armazena o resultado das instruções.
- *memvalid*: Armazena o estado de consistência do endereço de memória.

Neste esquema, quando uma instrução é decodificada, seus operandos de entrada são comparados aos operandos de entrada presentes na tabela para esta instrução. Se ocorrer um *match*, a instrução é ignorada.

S_n: Reúso por nome

O esquema S_n utiliza os nomes de registradores físicos da máquina para realizar a detecção de redundância das instruções. A diferença entre este esquema e o S_v é que este armazena no RB os estado de consistência dos registradores utilizados pelas instruções, ao invés dos valores dos operandos. Por conta disso, os campos *Op1* e *Op2* do esquema S_v são substituídos pelos campos *Op1 reg name* e *Op2 reg name* no esquema S_n. Além disso, o campo *result valid* é adicionado ao RB. Este é marcado como falso, quando outra instrução sobrescreve algum dos registradores de entrada utilizados pela instrução armazenada no RB.

Neste esquema, as instruções são reutilizadas, quando uma instrução é decodificada, e é verificado, através do campo *result valid*, que seus registradores de entrada não foram sobrescritos por outra instrução. De forma semelhante ao esquema S_v, instruções de *load* são reutilizadas através da validação do campo *memvalid* presente no RB.

S_{n+d}: Reúso por nome e dependência

S_{n+d} apresenta um esquema de reúso de instruções e traços. Ao contrário da DTM[5, 30], a S_{n+d} somente reutiliza instruções em uma mesma janela de despacho e que possuam dependência de dados entre elas, o que limita o potencial de reúso de traços. Outra diferença entre estas técnicas se dá por conta da S_{n+d} detectar redundância pelos nomes dos registradores que compõem o contexto de entrada do traço, enquanto a DTM possui melhor oportunidade de reúso, pois ela compara os valores dos operandos presentes no contexto de entrada do traço.

O RB, neste esquema, possui mais dois campos em relação ao RB do S_n, estes são os *src-index* para cada registrador de entrada da instrução. O campo *src-index* indica o índice da RB na qual reside a instrução geradora do operando de entrada da instrução corrente, isto é, a instrução fonte. Desta forma, é possível formar um "caminho de dados"entre instruções dependentes. Além do RB, no S_{n+d}, existe uma tabela auxiliar para o reúso de traços, denominada RST (*Register Source Table*). Esta possui uma entrada para cada registrador da máquina. Em cada entrada desta

tabela é armazenado o índice da RB referente a última instrução que modificou o registrador que a entrada da RST representa.

A reutilização de traços no esquema S_{n+d} é realizada verificando se, de fato, as instruções fontes do traço foram as últimas a alterarem os registros de entrada dos traços. Essa validação pode ser facilmente realizada utilizando a RST. Basta comparar os valores dos campos *src-index* de cada registrador com os valores presentes para esses registradores representados na RST. O reúso de instruções tradicionais e *loads* independentes, neste esquema, é realizado de forma idêntica ao esquema S_n.

Uma desvantagem dos modelos de reúso baseados em nome de registradores(S_n e S_{n+d}) em relação aos de valores (S_v) é o fato destes dois primeiros necessitarem de mais mecanismos para implementação no caso de execução especulativa. No esquema de nomes, se faz necessário corrigir a validade dos operandos, caso ocorra uma predição incorreta de desvio.

4.2.4 Reúso de computação por compilador

Os autores propõem neste trabalho um esquema de reúso de computação através de reescrita de segmentos de código escolhidas através de análises de compilação e simulações de valores[35]. O esquema consiste nas seguintes etapas: (*i*) identificar segmentos de códigos candidatos ao reúso, (*ii*) análise de dependência de dados para determinar os operandos de entrada e saída dos segmentos analisados, (*iii*) estimar os custos de realizar operações de *hash* para memorizar os valores de entrada e saída dos segmentos, (*iv*) estimar a granularidade de computação dos segmentos, isto é, o número de operações que cada segmento executa, (*v*) selecionar segmentos para realizar avaliação de valores e, por fim, (*vi*) determinar os segmentos de códigos que serão reescritos para utilizar uma *cache* de operandos de entrada e saída.

Em (*i*) são selecionados segmentos de códigos que são recorrentes no código, por exemplo, segmentos internos à laços. No passo (*ii*) é utilizada análise de fluxo de dados do programa para determinara entradas e saídas do segmento de código. Entradas são todas as variáveis lidas pelo segmento que não são produzidas por ele. As saídas são variáveis computadas pelo segmento que são utilizadas após a execução deste. Na etapa (*iii*) e (*iv*) são estimados os custos de computação (C) dos segmentos selecionados e reutilização (O) destes. Segmentos que apresentam a razão $O/C > 1$ são desconsiderados para etapa (*v*). Na etapa (*v*), a taxa de reúso R do segmento é estimada através simulação de conjunto de valores (*value-set profiling*). Por fim, na etapa (*vi*), os segmentos que apresentam a relação de redundância e custos $R > O/C$ são selecionados para reescrita.

Cada segmento selecionado para reescrita possui uma tabela *hash* dedicada para o reúso de computação. Tabelas podem ser unidas, caso diferentes segmentos utili-

zem as mesmas entradas. Para isso, é adicionado um campo de *bits* à tabela para identificar qual região possui os operandos de saída disponíveis. As entradas das tabelas *hash* são geradas através da concatenação dos operandos de entrada.

4.2.5 CCR - *Compiler-Directed Computation Reuse*

CCR[33] é uma técnica de reúso de regiões de códigos por compilador, mas com auxílio de mecanismos de *hardware*. Tradicionalmente técnicas de compiladores utilizam análises estáticas e simulação de valores (*value profiling*) para identificar regiões do código que possam ser candidatas ao reúso. Entretanto, essas técnicas podem não descrever o comportamento do programa em tempo de execução com assertividade necessária para potencializar o reúso de segmentos do código. Por conta disso, os autores, neste trabalho, propõem mecanismos de *hardware* para alocar as regiões identificadas em tempo de compilação que possuam melhor potencial de reúso em tempo de execução.

O DCMS (*Dynamic Computation Management System*) é um sistema de *hardware* proposto pelos autores para realizar as seguintes tarefas: (*i*) detecção, a região deve apresentar certas estatísticas de execução para ser alocada no CRB (*Computation Reuse Buffer*). (*ii*) Após a candidatura, a região é avaliada para validar o benefício de ser reutilizada. Por fim, na etapa (*iii*), diferentes instâncias dinâmicas da região são avaliadas para determinar a melhor forma de alocação no CRB. Três componentes de *hardware* compõe o DCMS, estes são: (*i*) *Reuse Sentry* (RS), o qual armazena a quantidade de execuções de cada região e identifica regiões candidatas. (*ii*) *Evaluation Buffer*, este avalia o comportamento de execução das regiões candidatas armazenando dados referentes às execuções em um *buffer* especializado. (*iii*) o RMS *Reuse Monitoring System* é um componente de *hardware* que utiliza métricas limite armazenadas no EB para inserir novas regiões no CRB. Caso o CRB esteja cheio, o RMS analisa métricas de reúso do CRB e desaloca entradas que não estão mais gerando reúso.

No esquema CCR tradicional o apoio de *hardware* não é utilizado, portanto somente alguns segmentos do código eram anotados como candidatos ao reúso. Entretanto, com apoio de *hardware*, todos os possíveis segmentos são anotados, pois este consegue identificar em tempo de execução as melhores regiões para o reúso utilizando métricas não disponíveis em tempo de compilação. Ainda sim, análises estáticas são realizadas para criar melhores regiões de reúso. Por exemplo, reordenação de blocos básicos em tempo de compilação para formar superblocos[38] e produzir regiões com menos operandos de entrada, porém com mais operações em sequência na região. Regiões deste tipo geram maiores ganhos de desempenho.

4.2.6 *Profile Driven Computation Reuse*

Neste trabalho[41], os autores apresentam um modelo de otimização de aplicações baseando-se na redundância de computação presente nelas. Este modelo de reutilização de computação em *software* consiste em identificar uma região do código da aplicação que possua alta intensidade computacional, e criar uma *cache* de resultados específica para esta região. O reúso da região pode ser total, isto é, todas as operações contidas naquela região do código são ignoradas, ou parcial, ou seja, apenas algumas operações da região são reutilizadas e o restante da computação é realizado.

A identificação de uma região do código se dá por séries de métricas criadas a partir de dados coletados de execuções anteriores da aplicação. Primeiramente é calculada a métrica de reúso potencial de cada operação da aplicação. Operações com alto potencial de redundância são selecionadas como semente. A partir de uma semente, operações são agregadas à ela e o novo potencial de reúso, agora a nível de região, é computado. Quando a agregação de operações à operação semente prejudica o potencial de reúso, a construção da região é finalizada. O intuito desta estratégia é construir regiões que não sejam grandes o suficiente a ponto de que o reúso se torne raro, mas também não permitir que elas sejam pequenas ao ponto do reúso ter um custo maior do que a própria computação da região. Uma região é refeita, caso ela tenha operações que possuam efeitos colaterais, por exemplo, alteração de variáveis globais. Tais operações não são permitidas em regiões de reúso, pois podem causar erros na execução do programa.

A *cache* é criada de forma específica para a região selecionada e o código desta é alterado para buscar resultados na *cache*, antes de realizar a computação da região. O tamanho da *cache* é determinado com base em estatísticas das execuções anteriores. As políticas de substituições são escolhidas entre as tradicionais LRU (*Least Recently Used*), FIFO (*First In First Out*) ou LFU (*Least Frequently Used*).

Finalmente o código otimizado é executado e é avaliado o ganho de desempenho e a economia de energia. Caso os resultados não sejam satisfatórios, uma nova análise é feita e novas regiões são selecionadas.

4.2.7 IncPy

IncPy[39] é uma modificação do interpretador do *Python* para permitir que funções sejam memorizadas em tempo de execução e reutilizadas em execuções distintas ou durante a execução em que foi memorizada. O intuito dos autores com esta customização do interpretador *Python* é fornecer um ambiente de execução aprimorado para *scripts* de análise de dados, isto é, *scripts* que realizam tarefas de leitura, *parse*, transformação, processamento e retirada de conclusões de dados. Quando um *script*

é executado utilizando o *IncPy*, o mesmo automaticamente memoriza operandos de entrada e saída, assim como, as dependências de certas chamadas de funções, que sejam determinísticas e puras, e cujo os resultados não demorem mais tempo para memorizar e reutilizar do que o tempo de execução da própria função. Funções puras são funções que não modificam valores existentes antes de serem invocadas.

A *cache* de memorização do *Incpy* armazena cada resultado das funções válidas em uma entrada com os seguintes campos:

- Nome completo: Armazena o nome do arquivo gerado pela entrada da *cache* e o nome completo da função.
- Argumentos: Armazena os argumentos da função.
- Retorno: Armazena os valores de retorno da função.
- *Output* do terminal: Armazena o conteúdo das saídas *stdout* e *stderr*.
- Dependência de variáveis globais: Armazena nome e valores de todas as variáveis globais, variáveis em escopos externos alinhados à função e campos estáticos de classes lidos durante a chamada.
- Dependência de arquivos de entrada: Armazena nome e datas de última modificação de arquivos lidos durante a chamada de funções.
- Dependência de arquivos de saída: Armazena nome e datas de última modificação de arquivos escritos durante a chamada de funções.
- Dependência de código: Nome completo e *bytecodes* da função representada pela entrada da *cache* e de todas as funções que essa chama transitivamente.

Quando a chamada de uma função termina, a entrada da *cache* é serializada em um arquivo binário. Para argumentos, variáveis globais e retorno de funções, *IncPy* também serializa o objeto inteiro que cada valor se refere, isso inclui todos os objetos alcançados transitivamente a partir destes valores. Os arquivos são nomeados de acordo com o resultado de um *hash md5* realizado sobre os argumentos das funções serializados. Arquivos de entrada da *cache* referentes à mesma função são armazenados em um subdiretório, cujo o nome é um *hash md5* do nome completo da função. Quando dependências presentes nas entradas da *cache* são alteradas, o interpretador se encarrega de eliminar aquela entrada e excluir o arquivo correspondente.

Para uma função ser reutilizada, primeiramente o interpretador verifica se existe uma entrada em disco da *cache* cujos argumentos da função, nome completo da função, dependências e variáveis globais associadas à função são as mesmas presentes no arquivo. Se houver um *match*, o *IncPy* verifica se os arquivos que servem de

insumo diretamente ou indiretamente para função foram modificados, se sim, o arquivo que representa a entrada da *cache* é excluído. Caso o código da função tenha sido alterado, todos os arquivos de *cache* associados à função são excluídos. Se ocorrer um *match* e as dependências e código da função não tiverem sofrido alterações, os resultados presentes no arquivo de *cache* são enviados como retorno ao invocador da função, e as saídas *stdout* e *stderr* presentes no arquivo de *cache* são impressas.

4.2.8 *Memoizer*

Memoizer[53] é uma biblioteca *Python* que utiliza técnicas de reúso de computação para obter ganhos de desempenho em aplicações financeiras. Construção de estratégias de compra e venda para acionistas é uma tarefa computacionalmente intensa e possui muitas computações repetidas em si. Esta biblioteca utiliza memorização em arquivos, que consiste em utilizar o nome da função e seus argumentos para gerar um código *hash* que serve como o nome de um arquivo que contém o resultado da função. Argumentos destas funções possuem dados grandes, matrizes e *dataframe* com ordens de grandeza de *gigabytes*, por conta disso, é necessário um esquema de *lookup* na *cache* que não utilize algoritmos de *hash* complexos, como *md5*, e que também não seja mais custoso do que executar novamente a função. O algoritmo *hash* utilizado pelo *Memoizer* é o *xxhash 64*

Os autores utilizam uma classe *decorator*, que possui métodos para realizar a memorização de funções. Para ativar a memorização dessas funções pela classe de *decorator* é necessário realizar uma anotação do tipo *@¡Nome_Da_Classe* imediatamente antes da assinatura da função. *Memoizer* permite ao programador identificar quais funções serão memorizadas e também automaticamente desativa a memorização de funções que possuam referência à palavra *rand* em seu corpo. Isto é feito para eliminar o risco de memorizar funções não determinísticas que possam alterar a lógica da aplicação, caso seus resultados sejam reaproveitados. A *Memoizer* também permite que a memorização para diferentes funções sejam parametrizadas a fim de permitir a melhor estratégia de reúso de computação para aplicações financeiras, e, assim, obter maior desempenho.

Capítulo 5

Sucuri

Nesta seção apresentamos a biblioteca *python Dataflow* Sucuri[21]. Sucuri é uma biblioteca minimalista que permite ao programador escrever aplicações paralelas utilizando o modelo de programação *Dataflow*.

5.1 Arquitetura Sucuri

Nesta seção, descrevemos com mais detalhes o funcionamento da arquitetura da Sucuri.

A versão da arquitetura da Sucuri utilizada neste trabalho é centralizada, ou seja, possui um processo líder que administra os demais processos responsáveis pela execução de um grafo *Dataflow*. Ela é composta dos seguintes componentes básicos: unidade de casamento, fila de tarefas prontas e *workers*.

A figura 5.1 apresenta a esquematização da arquitetura da Sucuri e a interação das unidades básicas.

A comunicação entre os processos paralelos *workers* e o escalonador central (processo líder) é realizada através de troca de mensagens. A troca de mensagens pode ser feita de duas formas: através de escrita e leitura em uma área de memória comum, onde a utilização da Sucuri é feita com o paradigma de memória compartilhada; ou via MPI[54], onde a utilização da Sucuri é feita com memória distribuída executando em *clusters*.

Em uma execução do grafo *Dataflow*, o escalonador central da Sucuri inicializa os *workers* paralelos e identifica no grafo *Dataflow* quais sãos os nós que não possuem entrada, esses nós fontes iniciam a execução. O escalonador cria objetos chamados *tasks*, também chamados de tarefas, os quais são instâncias da classe *Task*, a partir destes nós fontes. As tarefas possuem as seguintes informações principais: operandos de entrada, no caso de nós fonte, estes são nulos, e o identificador do nó que gerou esta tarefa. Essas tarefas são armazenadas na fila de tarefa prontas.

Os *workers* ao serem inicializados enviam uma mensagem para o escalonador central informando que os mesmos estão ociosos e podem consumir tarefas. O escalonador, por sua vez, utilizando um canal de comunicação (memória ou MPI), envia a tarefa para o *worker* ocioso. Este *worker* irá receber a tarefa e computá-la. Cada *worker* possui uma cópia do grafo *Dataflow* e utiliza o identificador do nó contido na tarefa para descobrir qual nó deverá ser computado com os operandos de entrada também contidos no objeto *Task*. Após a computação da tarefa pelo *worker*, os operandos resultantes são enviados ao escalonador central em forma de mensagem. Além dos operandos resultantes, esta mensagem também contém os identificadores dos nós destinatários de cada operando.

O escalonador central ao receber a mensagem de um determinado *worker*, a encaminha para a unidade de casamento. Esta irá propagar os operandos contidos na mensagem recebida às portas de entrada dos nós destinatários, cujos identificadores também estão presentes na mensagem recebida. Se todos os operandos de entrada de um determinado nó estiverem disponíveis, uma tarefa é instanciada a partir do nó, e é despachada para a fila de tarefas prontas. Se o grafo em execução for um *DAG Streaming* (ver seção 5.4), a tarefa só é instanciada e despachada a partir do nó, se todos os operandos de entrada estiverem disponíveis e, além disso, estes estiverem associados à mesma *tag*.

A tarefa é criada utilizando os valores dos operandos e o identificador do nó, e é armazenada na fila de tarefas prontas. Um *worker*, quando estiver ocioso, solicita uma nova tarefa. O escalonador retira a tarefa da fila de prontos obedecendo um padrão FIFO e a entrega ao *worker*, e o processo de propagação de operandos é reiniciando.

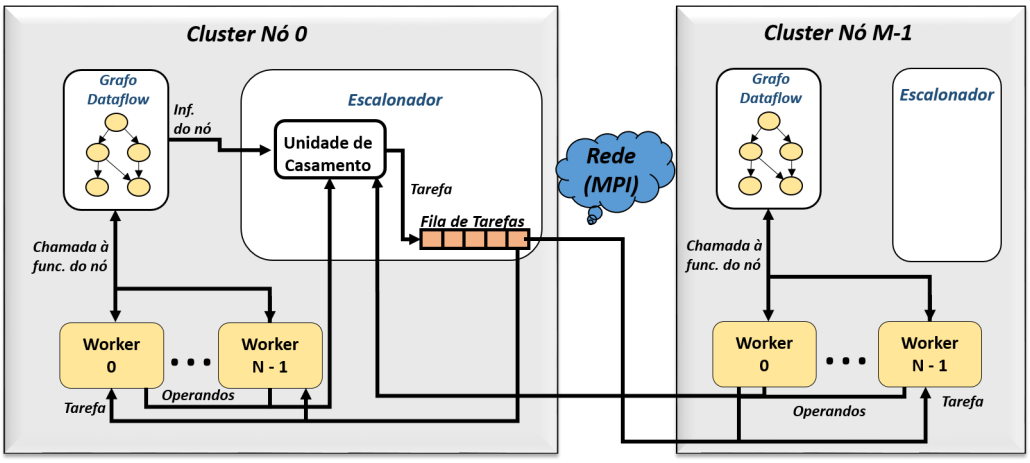


Figura 5.1: Esquematização da arquitetura da Sucuri.

A figura 5.2 apresenta o *pipeline* da Sucuri para execução de uma aplicação *Dataflow*. Os estágios representados por um retângulo executam dentro do escalonador, enquanto que os demais são executados por *workers*, podendo estes estarem na

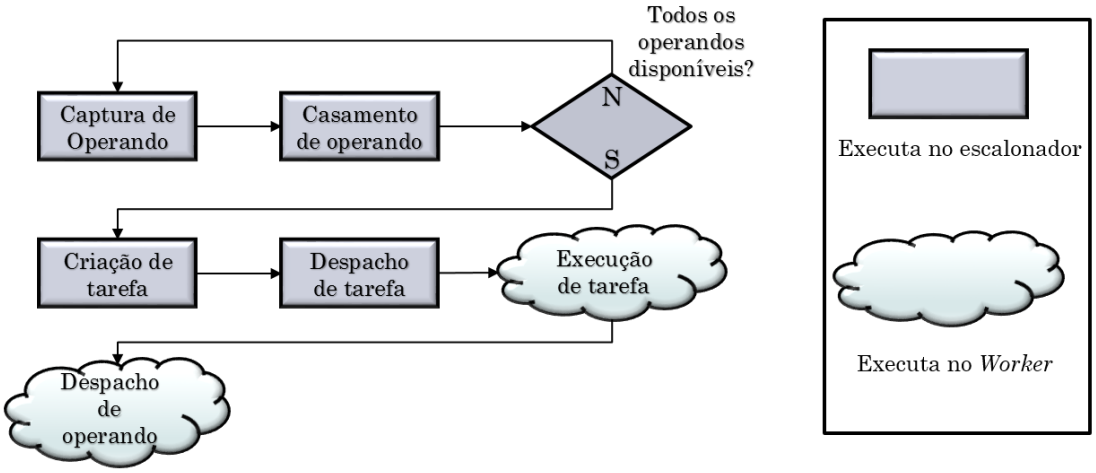


Figura 5.2: *Pipeline* da Sucuri.

mesma máquina onde o escalonador está sendo executado ou em máquinas externas interligadas por rede à maquina hospedeira do processo de escalonamento.

5.2 Modelo de programação

A Sucuri permite ao programador a escrita de aplicações em paradigmas *Dataflow*. Conforme descrito na seção 2.1, aplicações devem ser descritas na forma de grafos *Dataflow*, onde os vértices representam tarefas e as arestas representam a interdependência entre essas tarefas. Na Sucuri, os grafos *Dataflow* utilizados são DAGs [47, 55]. A implementação de grafos dinâmicos (DDG) [13] na Sucuri é um trabalho em andamento.

O programador, ao utilizar a Sucuri, deve decompor a sua aplicação em funções *python*. Estas funções são passadas como parâmetros para uma classe *python* chamada *Node*. As instâncias desta classe representam os vértices em um grafo *Dataflow*. Cada instância da classe *Node* deve ser adicionada ao grafo *Dataflow*. O grafo é um objeto instanciado a partir da classe *DFGraph*. É importante ressaltar que as funções passadas aos objetos *Node*, isto é, instâncias da classe *Node*, devem possuir dois parâmetros obrigatórios, o segundo parâmetro deve ser do tipo *list*. Este parâmetro será usado pelo *engine* da Sucuri para passar os operandos necessários à função para o escopo de visão da mesma. O primeiro parâmetro é um ponteiro que permite que a função acesse atributos do nó. Por exemplo, pode ser interessante em problemas de programação dinâmica, criar cada nó com uma coordenada (x, y). Essas coordenadas podem ser especificadas como atributos do nó e utilizadas pela função que o nó implementa para realizar alguma computação. A criação do conceito de atributos de nós e a possibilidade de acessá-los a partir da função implementada pelo nó são contribuições deste trabalho à Sucuri.

A especificação de interdependências, isto é, as arestas entre os nós dos grafos, é realizada através do método *add_edge* pertencente à classe *Node*. O método *add_edge* deve ser chamado a partir do objeto *Node* que originará os operandos que serão consumidos pelo objeto *Node* de destino. Por exemplo, dado um grafo G onde existem dois vértices $V = \{(u, v)\}$ e uma aresta direcionada $E = \{(u, v)\}$, o método *add_edge* deve ser invocado da seguinte forma $u.add\_edge(v)$.

Quando um nó é instanciado, deve-se especificar obrigatoriamente, além da função implementada por este, o número de portas de entrada dele. Na função *add_edge*, existe um parâmetro que especifica qual porta do nó destino receberá o operando produzido pelo nó fonte. Por exemplo, considerando o grafo *Dataflow* G com aresta $E = \{(u, v)\}$, onde v possui 3 portas de entrada, ao chamar o método $u.add\_edge(v, 2)$, estamos especificando que o operando produzido por u será encaminhado ao nó v em sua terceira porta de entrada. Para que a função implementada pelo nó v utilize esse operando, ela deve utilizar o parâmetro do tipo *list* com o índice 2. Por exemplo, $print\ args[2]$, imprime o operando.

Por fim, a classe *Scheduler* é instanciada tendo como parâmetro o grafo *Dataflow* criado. Esta classe possui o método *start* que dispara a execução do grafo Dataflow.

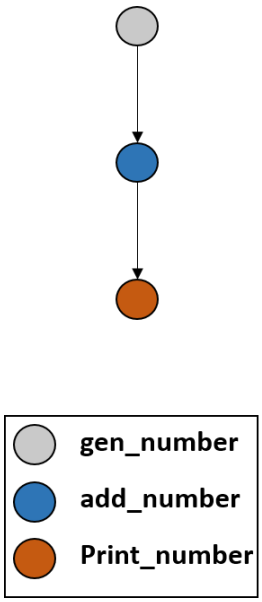
A figura 5.3b demostra a criação de uma aplicação *Dataflow* simples na Sucuri. Nesta aplicação a função *gen_number* (linha 6) gera dois números aleatórios. Esses números serão consumidos e somados na função *add_number* (linha 9). Por fim, a função *print_number* (linha 12) imprime o resultado.

O grafo para esta aplicação é construído da seguinte forma: três nós são criados para cada função descrita no parágrafo anterior (linhas 18, 21 e 24). Cada nó é adicionado ao grafo *Dataflow* (linhas 19, 22 e 25). O nó *no_gen_number* não depende de operandos, portanto o segundo parâmetro da classe *Node* recebe o valor 0. Isto significa que o nó não possui portas de entrada. O nó *no_add_number* depende dos resultados do nó *no_gen_number*, portanto é necessário que seu segundo parâmetro seja 1, isto é, ele espera um operando enviado do nó *no_gen_number* para poder ser executado. Por conta desta interdependência, existe a necessidade de explicitar uma aresta entre esses dois nós. A aresta é especificada através da chamada *no_gen_number.add_edge(no_add_number,0)* (linha 27), que significa que o operando produzido pelo nó *no_gen_number* ficará disponível na porta 0 do nó *no_add_number*. Semelhantemente, o nó *no_print_number* depende do nó *no_add_number* para executar, logo, ele também terá uma porta de entrada que receberá o valor produzido pelo o nó pai (*no_add_number*). Da mesma forma, o nó *no_print_number* também precisará criar uma aresta que indique dependência entre os dois nós (linha 28). Por fim, o grafo é passado como parâmetro para o objeto *sched*, que representa o escalonador. Ele também recebe outros parâmetros: o parâmetro *workers* indica quantos processos paralelos ele irá criar, enquanto que o parâmetro

"False" irá indicar se o módulo MPI deve ser utilizado ou não.

```
1  from pydf import *
2  import sys, random
3
4  workers = int(sys.argv[1])
5
6  def gen_number(self, args=[]):
7      return (random.random(),random.random())
8
9  def add_number(self, args=[]):
10     return args[0][0] + args[0][1]
11
12 def print_number(self, args=[]):
13     print args[0]
14     return None
15
16 graph = DFGraph()
17
18 no_gen_number = Node(gen_number,0)
19 graph.add(no_gen_number)
20
21 no_add_number = Node(add_number,1)
22 graph.add(no_add_number)
23
24 no_print_number = Node(print_number,1)
25 graph.add(no_print_number)
26
27 no_gen_number.add_edge(no_add_number,0)
28 no_add_number.add_edge(no_print_number,0)
29
30 sched = Scheduler(graph,workers,False)
31 sched.start()
32
33 graph.draw_graph()
```

(a) Código da aplicação.



(b) Grafo da aplicação.

Figura 5.3: Exemplo de uma aplicação simples utilizando Sucuri.

5.3 Aplicações DAG comum

Nesta seção apresentamos uma aplicação DAG comum paralela na Sucuri. Através da explicação detalhada de um exemplo apresentamos uma estrutura genérica para aplicações *Dataflow forkjoin*, como também, a utilização de nós especializados e a criação de arestas entre eles.

5.3.1 Exemplo de uma aplicação DAG paralela

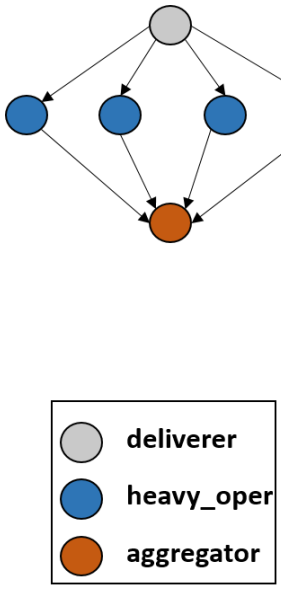
Na aplicação apresentada na figura 5.4, realizamos o processamento de um vetor de forma paralela. Utilizamos três funções, logo, três tipos de nós. O primeiro representa a função *deliverer*, esta é responsável por dividir o vetor em partes. Cada parte do vetor será entregue ao nó representante da função *heavy_oper*, o qual fará um determinado processamento complexo na parte do vetor que lhe foi entregue. Por último, o nó que representa a função *aggregator* irá receber de todos os demais nós do tipo *heavy_oper* as partes dos vetores já processados, e fará uma operação de agregação, por exemplo, somar cada elemento do vetor.

O nó *no_deliv*, do tipo *deliverer*, divide o vetor em n partes, onde n é o número de *workers*. Por conta disso ele precisa enviar para cada nó do tipo *heavy_oper* uma parte do vetor, permitindo assim o processamento paralelo deste. Para habilitar

o múltiplo direcionamento de operandos, deve-se utilizar a classe *MultiOutputNode* para criação dos nós. Esta é uma especialização da classe *Node*, e permite que um determinado nó envie para cada nó destino uma parte do operando produzido por ele. Para isso, é necessário que o retorno da função implementada pelo nó *MultiOutputNode* seja uma lista (tipo *list* do *python*) e que a relação de portas de saída do operando fonte e porta de entrada do operando destino seja especificada no comando *add_edge*.

```
1  from pydf import *
2  import sys, math, random
3
4  workers = int(sys.argv[1])
5  vect = range(1000)
6
7  def deliverer(self, args=[]):
8      cuted_vector = []
9      p_size=int(math.ceil(float(len(vect))/float(workers)))
10     for i in xrange(workers):
11        cuted_vector.append(vect[i*p_size:(i+1)*p_size])
12     return cuted_vector
13
14 def heavy_oper(self, args=[]):
15     result = 0
16     for i in args[0]:
17        result += i
18     return result
19
20 def aggregator(self, args=[]):
21     soma = 0
22     for i in args:
23         soma+=i
24     print soma
25     return soma
26
27 graph = DFGraph()
28
29 no_deliv = MultiOutputNode(deliverer,0)
30 graph.add(no_deliv)
31
32 no_aggregator = Node(aggregator,workers)
33 graph.add(no_aggregator)
34
35 no_heavy_op = []
36 for i in xrange(workers):
37     no_heavy_op.append(Node(heavy_oper,1))
38     graph.add(no_heavy_op[-1])
39     no_deliv.add_edge(no_heavy_op[-1],0,i)
40     no_heavy_op[-1].add_edge(no_aggregator,i,0)
41
42 sched = Scheduler(graph,workers,False)
43
44 sched.start()
```

(a) Código da aplicação.



(b) Grafo da aplicação.

Figura 5.4: Exemplo de uma aplicação DAG comum.

A figura 5.4a apresenta o código da aplicação, enquanto que a 5.4b apresenta o grafo resultante da aplicação. Nas linhas 29 a 30 e 32 a 33 criamos e adicionamos ao grafo os nós do tipo *deliverer* e *aggregator*, respectivamente. O laço da linha 36 cria n nós do tipo *heavy_oper*, onde n é o número de *workers*, com uma única porta de entrada. Os nós criados são colocados em uma lista (linha 37). O método *no_deliv.add_edge(no_heavy_op[-1],0,i)* na linha 39, cria uma aresta do nó *no_deliv* para o nó *no_heavy_op* recém adicionado à lista, indicando como 0 a porta de entrada do nó destino e i como porta de saída do nó fonte. Isto significa que o elemento i da lista produzida pelo nó fonte ficará disponível na porta de entrada 0 do nó destino. Portanto, se na função *deliverer*, for retornada, por exemplo, a seguinte lista de inteiros: $[[1,2,3],[4,5,6],[7,8,9]]$, o segundo nó ($i = 1$) *heavy_op* terá como operando de entrada a lista $[4,5,6]$. Semelhantemente, o nó do tipo *agregator* deve receber o resultado de todos os nós do tipo *heavy_oper*. Cada resultado deve ser

entregue a uma porta específica do nó *no_aggregator*. Por conta disso o método *no_heavy_op[-1].add_edge(no_aggregator,i,0)* (linha 40) é invocado, onde *i* corresponde ao número da porta destino no nó *no_aggregator* e 0 a porta de saída do nó *no_heavy_op*.

5.4 Aplicações DAG *stream*

No paradigma *Dataflow*, a criação de laços de repetição podem ser feitas através de ciclos no grafo *Dataflow*, entretanto a Sucuri apenas trabalha com grafos acíclicos. A criação de grafos com repetições na Sucuri deve ser realizada a partir de grafos de *streaming*[25, 26]. Grafos deste tipo são alimentados continuamente por entradas de fontes externas. Por exemplo, uma aplicação de processamento de imagem, pode ser descrita como um grafo DAG que é continuamente alimentado por *frames* de um vídeo. Neste caso, o DAG seria semelhante a um *pipeline* de transformações e o vídeo seria uma fonte de dados externa ao grafo.

A implementação de repetições de ciclos em um grafo *Dataflow* pode ser dinâmica ou estática, conforme explicado na seção 2.3. A Sucuri permite que iterações distintas de um mesmo grafo sejam executadas em paralelo. Para isso é utilizado um mecanismo de *tags* para identificar a que iteração um determinado operando pertence.

A implementação de grafos de *streaming* na sucuri é realizada utilizando nós especializados. Estes são: *Source*, *FilterTagged*, *MPFilterTagged* e *Serializer*.

5.4.1 *Source*

Este tipo de nó é responsável por continuamente enviar operandos ao nó de destino. Ele recebe como parâmetro, no momento de sua criação, um objeto iterável, por exemplo, uma lista de objetos ou de arquivos. O nó *Source* possui uma função padrão que lê elemento a elemento do objeto iterável, cria uma *tag*, isto é, um valor numérico que identifique a que iteração este operando pertence, e a associa ao operando lido. A tupla *(operando, tag)* é entregue ao nó destino.

5.4.2 *FilterTagged*

O nó *FilterTagged* é semelhante ao nó do tipo *Node*, a diferença é que o nó *FilterTagged* espera como entrada um objeto em forma de tupla do tipo *(operando, tag)*. Este nó especializado, desencapsula o operando e o passa para função implementada por ele. O resultado da função é encapsulada com o valor da *tag* que chegou ao nó, e esta nova tupla *(operando resultante, tag)* é entregue ao nó destino. Isso garante

que operandos não sejam erroneamente utilizados em outras iterações, pois o escalonador de tarefas da Sucuri utiliza a *tag* associada aos operandos para identificar se um determinado nó está pronto para execução.

5.4.3 *MPFilterTagged*

Este nó é uma especialização do *FilterTagged*, portanto, o que foi descrito sobre a classe de nós *FilterTagged* se aplica à classe de nós *MPFilterTagged*. Entretanto, esta classe de nó permite que operandos produzidos por um nó sejam enviados a diferentes destinos de forma idêntica ao nó *MultOutputNode* descrito na seção 5.3.

5.4.4 *Serializer*

O nó *Serializer* é responsável pela ordenação dos resultados. O nó *Source* alimenta o grafo *Dataflow* obedecendo uma determinada ordem, a saber, a ordem em que os elementos estão no objeto iterável. Por conta do modelo *Dataflow* realizar execução fora de ordem, pode ser necessário ordenar os resultados no fim da execução do grafo. O nó *Serializer*, ao receber os operandos necessários para execução de sua função, verifica se os mesmos estão associados à *tag* que deve ser a próxima executada. Por exemplo, se a função deste nó já foi executada para operandos da *tag* 0, então a próxima *tag* deverá ser a de número 1. Porém, se for recebido os operandos associados à *tag* de número 2, estes serão armazenados em um *buffer* do nó *Serializer*. A função somente será executada para os operandos da *tag* 2, quando os operandos associados à *tag* 1 forem recebidos e a função for executada para estes operandos.

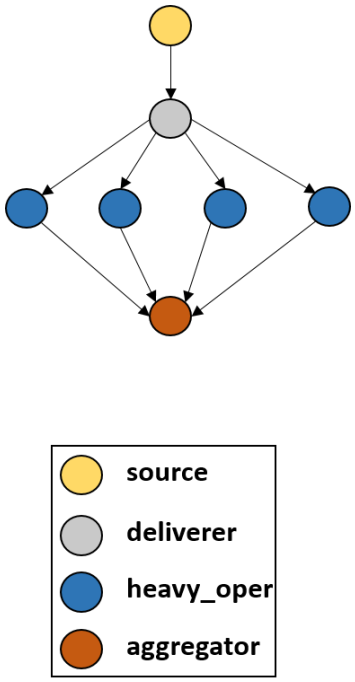
5.4.5 Exemplo de aplicação *streaming*

A figura 5.5 apresenta a aplicação da figura 5.4 modificada para ser uma aplicação de *streaming*. O objetivo da aplicação é processar uma lista de vetores, ao invés de um vetor. Primeiramente é criado um objeto iterável que contém os vetores que serão processados (linha 5). Este objeto é passado como parâmetro ao nó *Source* no momento de sua criação (linha 29). A função *deliverer* (linha 9) foi modificada, ela não mais utilizará a variável *vect*, antes ela receberá os vetores como operandos. O nó *no_src* passa a ser o inicializador da execução, pois não possui portas de entrada, e, portanto, não depende de outros nós. O nó *no_deliv* é instanciado como um nó *MPFilterTagged* (linha 32) com uma única porta de entrada, pois é necessário que ele consuma e produza operandos para diferentes destinos mantendo o mesmo número de *tag*. O nó *no_aggregator* foi criado como um *Serializer* (linha 35), pois ele fará a impressão dos resultados na ordem em que eles foram lidos pelo nó *Source*. A construção das arestas (linha $40 - 44$) entre os nós das funções *deliverer*, *heavy_oper*

e *aggregator* é realizada de forma semelhante à descrita na seção 5.3. Por fim, o escalonador é criado na linha 46 e a execução é iniciada na linha 48.

```
1  from pydf import *
2  import sys, math
3
4  workers = int(sys.argv[1])
5  vect = [range(1000), range(1000,2000), range(2000,3000)]
6
7  def deliverer(self, args=[]):
8      cuted_vector = []
9      p_size=int(math.ceil(float(len(args[0]))/float(workers)))
10     for i in xrange(workers):
11        cuted_vector.append(args[0][i*p_size:(i+1)*p_size])
12     return cuted_vector
13
14 def heavy_oper(self, args=[]):
15     result = 0
16     for i in args[0]:
17        result+=1
18     return result
19
20 def aggregator(self, args=[]):
21     soma = 0
22     for i in args:
23         soma+=i
24     print soma
25     return soma
26
27 graph = DFGraph(len(vect))
28
29 no_src = Source(vect)
30 graph.add(no_src)
31
32 no_deliv = MPFilterTagged(deliverer,1)
33 graph.add(no_deliv)
34
35 no_aggregator_ser = Serializer(aggregator,workers,len(vect))
36 graph.add(no_aggregator_ser)
37
38 no_src.add_edge(no_deliv,0)
39 no_heavy_op = []
40 for i in xrange(workers):
41     no_heavy_op.append(FilterTagged(heavy_oper,1))
42     graph.add(no_heavy_op[-1])
43     no_deliv.add_edge(no_heavy_op[-1],0,i)
44     no_heavy_op[-1].add_edge(no_aggregator_ser,i,0)
45
46 sched = Scheduler(graph,workers,False)
47
48 sched.start()
```

(a) Código da aplicação.



(b) Grafo da aplicação.

Figura 5.5: Exemplo de uma aplicação paralela de *streaming* utilizando Sucuri.

Capítulo 6

Sucuri + DF-DTM

A técnica DF-DTM estende o conceito de reúso de tarefas computacionais ao paradigma *Dataflow*. Como discutido na seção 2.1, aplicações em *dataflow* são descritas através de grafos, nos quais os nós representam as tarefas da aplicação e as arestas dirigidas suas dependências. Portanto, a técnica de reúso em *dataflow* se encarrega de memorizar o valor de entrada dos nós e reutilizá-los, caso estes sejam detectados como redundantes. Da mesma forma que, em arquiteturas tradicionais, traços redundantes podem ser formados, no paradigma *Dataflow*, subgrafos redundantes são formados. Nesta seção apresentamos com mais detalhes o reúso de tarefas em *dataflow*, como também, a implementação da DF-DTM na Sucuri.

6.1 Reuso de computação em grafos *Dataflow*

No modelo *Dataflow*, nós de um grafo são semelhantes às instruções estáticas em um modelo de Von Neumann, portanto, podemos fazer uma tradução dos termos usados neste modelo para o modelo *Dataflow*. Instruções dinâmicas instanciadas a partir de instruções estáticas, no modelo *Dataflow*, seriam nós dinâmicos instanciados a partir de um nó estático. Quando um nó possui todos os operandos de entrada necessários para a execução da tarefa por ele representada, ele instancia um nó dinâmico. Da mesma forma que instruções redundantes em sequência podem ser agrupadas formando um traço redundante, no paradigma *Dataflow*, nós redundantes podem ser agrupados dando origem a um subgrafo redundante.

Apresentamos a seguir a definição de contextos de entrada e saída e os algoritmos básicos para reúso de tarefas em *dataflow* e construção de subgrafos redundantes.

6.1.1 Contextos de entrada e saída no modelo *Dataflow*

As definições de contextos de entrada e saída também são necessárias ao reúso de computação em *dataflow*. Contextos são definidos utilizando as arestas que repre-

sentam as dependências de dados no grafo *dataflow*. Um contexto de entrada em um grafo é definido por todas as arestas provenientes de nós externos ao subgrafo que incidem sobre ele. Da mesma forma, o contexto de saída é definido pelas arestas oriundas aos nós do subgrafo redundante que saem do mesmo.

A figura 6.1 apresenta um exemplo de um subgrafo redundante e os seus contextos de entrada e saída. Os nós em amarelo são os membros do subgrafo. As entradas dos contextos estão em formas de tuplas com 3 valores: o identificador do nó, o número da aresta e o valor do operando.

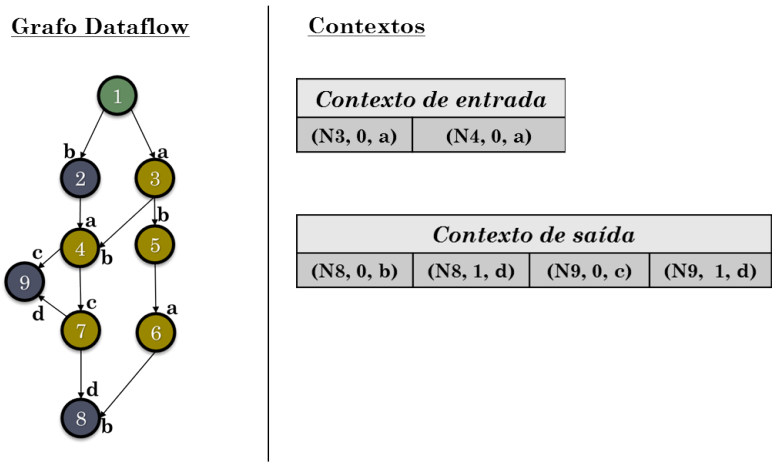


Figura 6.1: Exemplo de contextos de entrada e saída em um subgrafo redundante.

6.1.2 Algoritmos para o reúso de tarefas em *dataflow*

O reúso de nós *dataflow* ocorre através de memorizações de operandos de entrada e saída deste em uma tabela histórica. Buscas são realizadas sobre esta tabela para verificar se um nó é redundante. A construção de subgrafos utiliza os registro desta tabela para detectar e adicionar nós redundante a um subgrafo em construção. A seguir apresentamos o algoritmo de detecção de nós redundantes e construção de subgrafos.

Detecção de nó redundante

O algoritmo para detecção de nós redundantes funciona da seguinte forma: quando um nó é executado, os operandos de entrada e saída referentes à execução deste nó são armazenados em uma tabela de memorização. Cada vez que um nó é identificado como pronto para execução, ou seja, possui todos os operandos de entrada disponíveis, a tabela é consultada para verificar se a operação realizada por ele é conhecida. Se sim, os resultados da execução do nó são encaminhados aos seus vizinhos dependentes, e este é identificado como um nó redundante, caso contrário, o

nó é executado. Nós que implementem tarefas não-determinísticas são considerados como tipos inválidos de nós para o reúso.

Criação de subgrafos redundantes

A criação de subgrafos redundantes é realizada através dos seguintes passos:

1. Dado um grafo *dataflow* G, selecione um nó v marcado como pronto, isto é, possui todos os operandos de entrada disponíveis.

2. Se o nó v é redundante, adicione-o ao subgrafo em construção e propague os resultados obtidos na tabela de memorização aos seus dependentes $u_1, ..., u_n$, conforme explicitado pelas arestas do grafo.

3. Para todo nó $u_1, ..., u_n$ identificado como redundante, armazene-os em uma fila de espera F.

4. Caso haja nós em F, retire o primeiro nó da fila F, renomeie-o como v, e execute o passo 2. Caso contrário, finalize a construção do subgrafo.

O algoritmo acima utiliza um método guloso, aglutinando nós redundantes e criando o maior subgrafo redundante possível. Ele começa a criação a partir de um nó raiz e, através dele, busca caminhos redundantes no grafo, isto é, caminhos que passem por nós redundantes. Quando não há mais caminhos redundantes possíveis, a criação do subgrafo é encerrada e seus contextos de entrada e saída são memorizados.

6.2 Implementação das tabelas de memorização

A DF-DTM possui componentes semelhantes à DTM. Estes são: uma tabela para o armazenamento de nós redundantes chamada NRT (*Node Reuse Table*), análoga à *Memo Table G*, e uma tabela para o armazenamento de subgrafos redundantes chamada SRT (*Subgraph Reuse Table*), análoga à *Memo Table T*. Entretanto, por conta de algumas arquiteturas *dataflow* possuírem filas de tarefas que estão aguardando recursos, a DF-DTM insere um novo componente denominado unidade de inspeção. Nas subseções seguintes detalhamos essas tabelas e como as implementamos na arquitetura da Sucuri. A unidade de inspeção é detalhada na seção 6.3.

6.2.1 NRT - *Node Reuse Table*

A tabela ou *cache* NRT é uma tabela de memorização utilizada para armazenar valores de entrada e saída de nós válidos para o reúso no grafo *dataflow*. Para o rápido acesso dos seus elementos, utilizamos uma estrutura de particionamento baseada em *hashs* dos valores de entrada dos nós. Esta possui os seguintes campos:

- *Indx*: Número de partição da tabela de memorização.
- *Inp_Op*: Operandos de entrada utilizados na execução do nó.
- *Out_Op*: Operandos de saída produzidos pela execução do nó.
- *Node ID*: Identificador numérico único do nó.
- *Node Group ID*: Identificador numérico único do grupo ao qual o nó executado pertence.
- *Node Type*: Nome da tarefa que o nó implementa, semelhante a um mnemônico de instrução.

O campo *Indx* é construído a partir dos operandos de entrada do nó. Esses operandos são serializados em uma *string* de *bytes*. Como a Sucuri é implementada em *python*, utilizamos o método *cPickle* para realizar a serialização. A *string* é convertida em um hexadecimal através de uma função de *hash md5*, após isso, o hexadecimal é convertido em um inteiro decimal. Uma operação de módulo é realizada entre este valor e o número de partições da NRT.

Quando uma operação de busca na tabela é feita para os operandos de entrada de um nó, a conversão *hash* utilizada para calcular o campo *Indx* é realizada sobre estes operandos. O valor inteiro gerado a partir da conversão é utilizado para acessar a NRT. Isso garante um acesso mais rápido à NRT, pois apenas um subconjunto de entradas desta tabela serão verificados para validar a redundância do nó. Isto foi implementado na Sucuri para otimizar o acesso à tabela durante as simulações de execuções.

A figura 6.2 apresenta este processo de transformação para definir a partição em que a entrada da NRT será armazenada. O quadro "A"mostra o nó do grafo recebendo 3 operandos de entrada (a, b, c) e produzindo o operando d. À esquerda de "A", vemos o processo de transformação do valor (a, b, c) em um inteiro a ser utilizado como número de partição de uma NRT que possui 10 partições. O processo inicia com a criação de uma *string* de *bytes* utilizando a classe *cPickle*. A seguir, um código *hash md5* convertido para decimal é gerado, e, por fim, a operação de módulo entre este valor e a quantidade de partições na NRT, calcula o número da partição na qual esta entrada deverá ser armazenada. O quadro "C"apresenta o registro armazenado com os dados da execução do nó 2.

Os campos *Node ID*, *Node Group ID* e *Node Type* são utilizados para identificar os nós redundantes em três esquemas distintos de implementação da NRT: (i) NRT global (GNRT), (ii) NRT compartilhada (SNRT) e (iii) NRT local (LNRT).

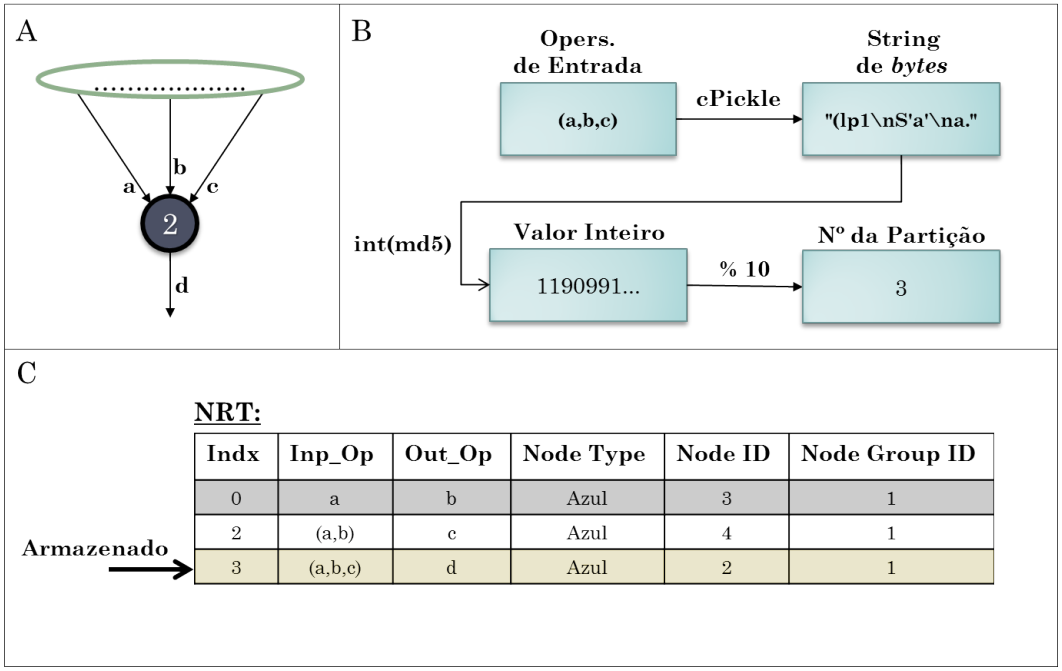


Figura 6.2: Esquema de particionamento da NRT na Sucuri.

NRT Global

A NRT global consiste em uma única tabela NRT que possui os valores de entrada e saída para todos os nós do grafo. Portanto, um nó com todos os seus operandos de entrada disponíveis, antes de executar, irá consultar a tabela NRT para verificar se o seu resultado já foi memorizado por ele mesmo ou por algum outro nó do grafo que implemente a mesma função que ele.

Note que, ao contrário das arquiteturas tradicionais, a tabela de memorização não é indexada por um campo de contador de programas, ao invés disso, é utilizado somente o tipo do nó e os valores de entrada do mesmo para encontrar o registro memorizado. Em implementações de reúso em arquiteturas tradicionais de Von Neumann, é interessante utilizar o PC como indexador da tabela por este prover um rápido acesso, como também, ser necessário na atualização de preditores de desvios e outras estruturas globais da máquina. Portanto, neste modelo, uma instrução do tipo *add*, por exemplo, não pode reutilizar outra instrução do mesmo tipo, caso possuam endereços diferentes. Por não haver PC no paradigma *dataflow*, o reúso é potencializado, pois ele ocorre para diferentes instruções do mesmo tipo, ao invés de instruções de um mesmo endereço.

A figura 6.3 apresenta o esquema de implementação global da NRT na Sucuri. Note que todos os nós possuem seus resultados escritas em uma única tabela, o que permite que nós distintos, mas que implementem a mesma função, utilizem resultados produzidos por eles entre si. Repare que a unidade de casamento provê uma entrada de um nó pronto para execução à NRT para que uma busca seja feita.

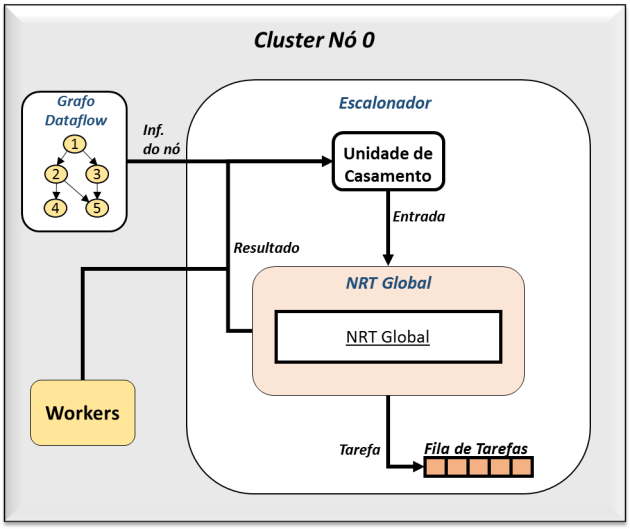


Figura 6.3: Organização do escalonador central da Sucuri com uma NRT global.

Se o resultado desta entrada for conhecido, ele é retornado à unidade de casamento para que seja propagado aos nós dependentes. Caso contrário, uma entrada é criada na NRT, e a tarefa é instanciada a paritr do nó que estava pronto para executar.

NRT Compartilhada

O esquema de implementações de NRTs compartilhadas consiste em implementar diferentes NRTs que serão utilizadas por grupos de nós distintos. Isto se assemelha à processadores *multicores* com organizações de *caches* locais L1 e L2 por núcleo.

No momento de criação de grafos na Sucuri, cada nó é associado a um identificador (*Node ID*). De forma semelhante, também podemos associar cada nó ao identificador de um grupo (*Node group ID*). Nesta primeira implementação da DF-DTM, utilizamos uma metodologia simples para associação de nós aos grupos através de operações módulo. A metodologia consiste em, dado uma quantidade n de NRTs, o grupo associado ao nó de *Node ID* id é dado por $mod(Node_{id}, n)$. Por exemplo, em uma implementação com 4 NRTs, o nó 101 seria associado ao grupo $mod(101, 4) = 1$, ou seja, este nó escreveria e leria os resultados de suas operações na NRT-1. Um esquema de NRTs compartilhadas com n NRTs é denominado de (S, n), onde S identifica um esquema de NRT compartilhada, e n a quantidade de NRTs utilizadas na implementação.

A figura 6.4 apresenta um esquema de NRT $(S, 3)$. Note que os nós são organizados em grupos distintos e a visibilidade de reúso deles está limitada à *cache* a qual o grupo está associado. Esta associação é representada na figura pela faixa azul que envolve os nós do grafo e o número n na NRT-n. Por exemplo, os nós 4 e 5 pertencem ao grupo 1 e irão ler e escrever na NRT-1. Se o nó 4 possuir operandos de entrada cujo os resultados foram produzidos anteriormente somente pelo nó 2, o

nó 4 não será detectado como redundante, pois não possui acesso à NRT-2.

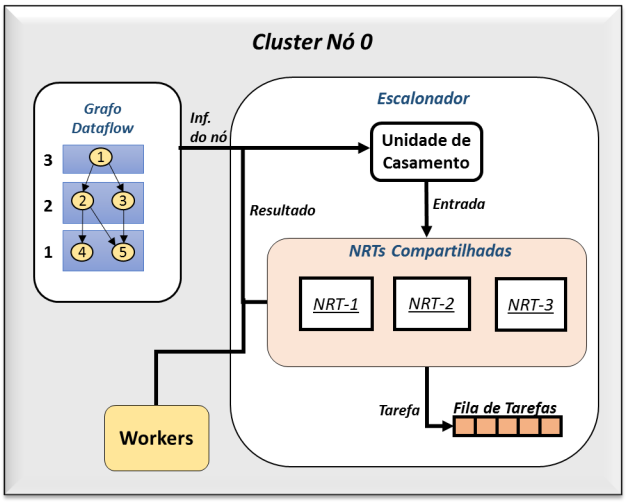


Figura 6.4: Organização do escalonador central da Sucuri com 3 NRTs compartilhadas $(S, 3)$.

NRT Local

Um esquema de implementação com NRT locais se assemelha às estratégias de reúso em arquiteturas tradicionais, onde instruções somente reutilizam resultados criados por elas. Neste cenário, são criadas n NRTs, e cada nó do grafo possui uma tabela de memorização própria. Pode-se dizer que esse esquema é uma especificação do esquema de NRTs compartilhadas, onde o número de *caches* é igual ao número de nós do grafo. Neste cenário o identificador do nó é utilizado para acessar a NRT.

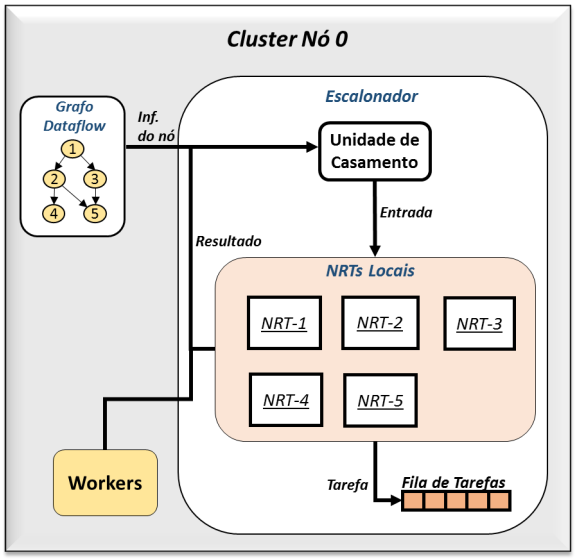


Figura 6.5: Organização do escalonador central da Sucuri com NRTs locais.

A figura 6.5 apresenta este esquema de uso das NRTs para um grafo com 5 nós. Note que cada nó possui uma *cache* dedicada a si, reduzindo possíveis contenções,

porém diminuindo o escopo de visibilidade dos mesmos. Por exemplo, o nó 1 irá ler e escrever somente na *cache* NRT-1.

6.2.2 SRT - *Subgraph Reuse Table*

A tabela ou *cache* SRT é utilizada para o armazenamento de subgrafos redundantes. Ela é uma tabela global, isto é, nela está contida subgrafos compostos de nós distintos independente dos grupos aos quais eles pertençam ou identificadores que eles possuam. Os subgrafos redundantes são representados nela através dos seus contextos de entrada e saída. A SRT possui os seguintes campos:

- *Id*: Campo contendo um identificador único para o subgrafo redundante.
- *Contexto de entrada*: Campo contendo a tupla (*Node Id do nó do contexto*, *porta de entrada*, *operando de entrada*) para cada operando que compõe o contexto de entrada.
- *Contexto de saída*: Campo contendo a tupla (*Node Id do nó destino*, *porta de entrada*, *operando de saída*) para cada operando que compõe o contexto de saída.
- *Tamanho do subgrafo*: Campo contendo a quantidade de nós no subgrafo.
- *Nós membros do subgrafo*: Campo contendo os identificadores de todos os nós que compõe o subgrafo.

Os valores do campo que armazena o contexto de entrada de um subgrafo redundante utilizam dados dos nós presentes na Sucuri chamados de número de portas. Por exemplo, um nó que possua 2 arestas incidindo sobre si, irá possuir uma aresta apontando para porta 0 e outra para porta 1. Estas informações são relevantes para formação dos contextos de entrada. Estes são formados da seguinte forma: dado um subgrafo redundante S, selecione todos os pais $p_1, p_2, ..., p_n$ exteriores ao subgrafo S, que possuam filhos $s_1, s_2, ..., s_n$ neste subgrafo. Para toda aresta (p_i, s_i), forme a tupla com os dados: $s_i.Node\_Id$, número da porta utilizada pela aresta e o valor do operando entregue pela aresta.

O contexto de saída de um subgrafo é armazenado na SRT de forma semelhante: dado um subgrafo redundante S, com nós $s_1, s_2, ..., s_n$, selecione todos os filhos (dependentes) f_i dos nós s_i que não pertençam ao subgrafo. Para cada aresta (s_i, f_i), forme a tupla com os dados: $f_i.Node\_Id$, número da porta utilizada pela aresta e o valor do operando entregue pela aresta.

Além dos campos presentes na SRT, para cada objeto *Node* da Sucuri foi adicionado o atributo *SRT_member*. Este atributo armazena o valor de cada entrada

da SRT da qual o nó faz parte. Por exemplo, se a SRT armazenou 3 subgrafos redundantes com identificadores 1, 2 e 3, respectivamente, e o nó v é membro dos subgrafos redundantes 1 e 3. Então, este nó possuirá o valor $[1, 3]$ para o atributo *SRT_member*.

Junto à SRT existe um *buffer* de construção responsável por armazenar os nós que são adicionados ao subgrafo redundante enquanto este está em processo de construção. Quando a construção do subgrafo termina, uma entrada é criada na SRT utilizando os dados presentes neste *buffer*. Em seguida, o *buffer* é esvaziado.

O tamanho do *buffer* de criação de subgrafos pode ser limitado para que subgrafos menores sejam criados. A limitação também permite a implementação deste componente em um sistema com recursos restritos. Quando existe limitação no *buffer* e um subgrafo em construção atinge esse limite, a construção é finalizada e o subgrafo é armazenado na SRT. Existem casos em que um subgrafo redundante é incorporado ao subgrafo em construção. Isso pode acarretar em um subgrafo ultrapassar o limite do *buffer* em alguns nós. Quando isso ocorre, o subgrafo redundante que estava sendo incorporado é desmembrado e os nós deste são incorporados ao subgrafo em construção até que o limite do *buffer* seja atingido. Quando o limite é finalmente atingido, o *buffer* é reiniciado e os nós restantes do subgrafo desmembrado são utilizados para dar continuidade a construção de um novo subgrafo.

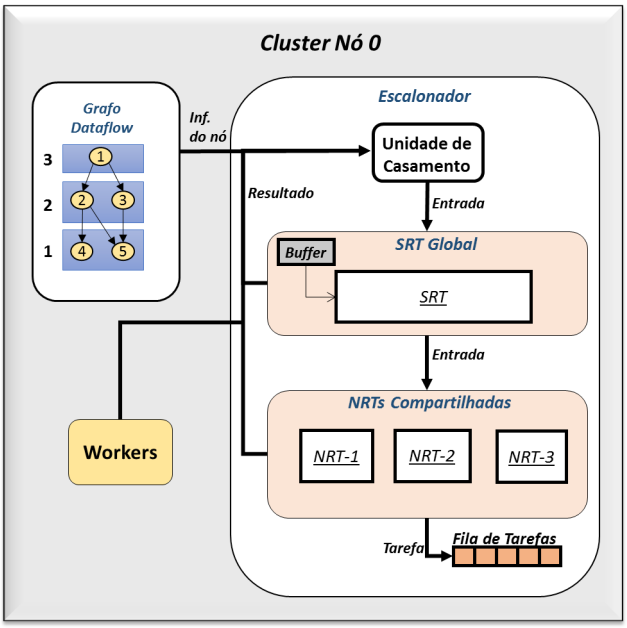


Figura 6.6: Organização do escalonador central da Sucuri com a SRT e um esquema de NRTs $(S, 3)$.

A figura 6.6 apresenta um escalonador da Sucuri utilizando uma SRT. A SRT possui precedência na busca por resultados, ela é sempre verificada antes da NRT, pois o reúso de subgrafos é preferível ao reúso de nós.

6.2.3 Funcionamento das tabelas de memorização

O reúso de computação na DF-DTM é composto pelas seguintes etapas: (*i*) armazenamento de nós na NRT, (*ii*) reutilização de nós redundantes, (*iii*) construção de subgrafos redundantes e (*iv*) reutilização de subgrafos redundantes.

(*i*) - Criação de entradas na NRT

A etapa (*i*) ocorre sempre que um *worker* retorna ao escalonador central da Sucuri o resultado de uma tarefa t. Neste momento a tabela NRT é consultada para validar se uma entrada já existe para o tipo de nó representado por t e que possua os mesmos operandos de entrada utilizados pela tarefa. Caso exista, não é criada uma nova entrada. Caso contrário, uma nova entrada na NRT é criada com os dados de entrada e saída do nó representado pela tarefa t. Isso é realizado para que execuções de tarefas iguais em paralelo não dupliquem uma entrada na NRT.

(*ii*) - Reúso de nós pela NRT

No momento em que um nó irá instanciar uma tarefa, a tabela SRT é consultada conforme descrito em (*iv*). Caso não ocorra um *hit* nesta tabela, a NRT é considerada. Neste caso, os operandos de entrada do nó são utilizados para calcular a partição em que, possivelmente, seus valores de entrada e saída estão armazenados. Caso estes valores sejam encontrados, o nó é reutilizado, a tarefa não é instanciada, e os operandos resultantes recuperados da NRT são propagados aos nós dependentes. Caso não haja um *hit* na NRT, a tarefa é instanciada normalmente e armazenada na fila de prontos, onde irá aguardar seu processamento.

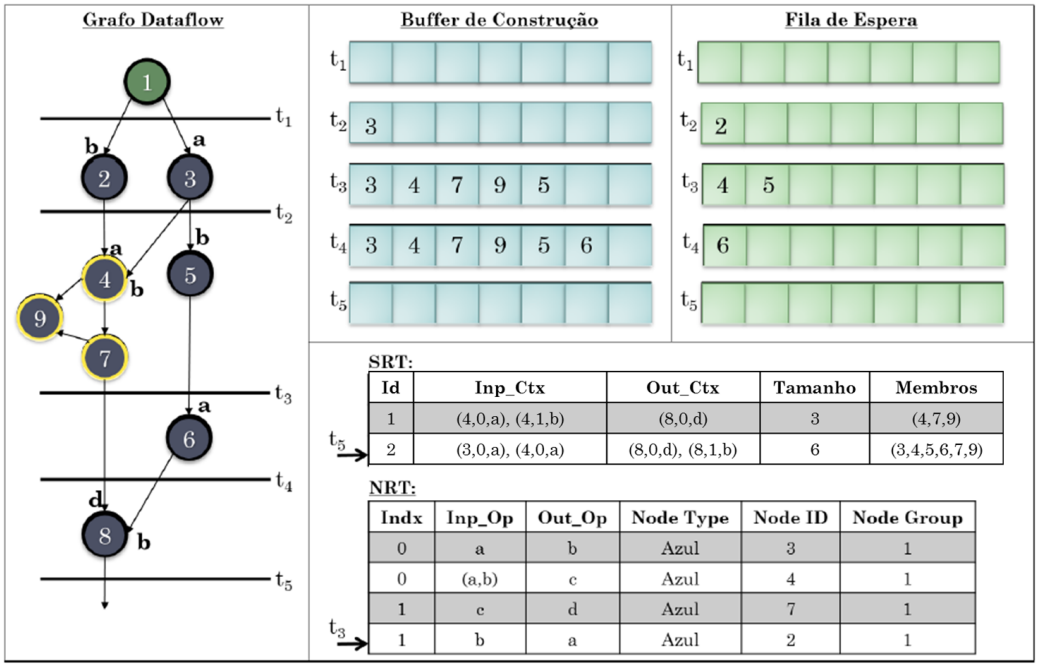
(*iii*) - Construção de Subgrafos Redundantes

A construção de subgrafos é iniciada quando um subgrafo redundante é detectado, conforme explicado em (*iv*), ou quando um nó redundante é detectado, conforme explicado em (*ii*). Neste momento, nós redundantes ou subgrafos redundantes são adicionados ao subgrafo em construção utilizando um *buffer* de construção de subgrafos. Quando não há mais nós redundantes ou subgrafos redundantes que possam ser adicionados ao subgrafo em questão, a construção deste é finalizada. Por conta disso, uma entrada na SRT é criada e o *buffer* é reiniciado, permitindo que um novo subgrafo seja construído.

(*iv*) - Reutilização de Subgrafos Redundantes

A detecção de subgrafos redundantes acontece no momento em que um nó instancia uma tarefa. Antes da tarefa ser instanciada, é verificado de quais subgrafos

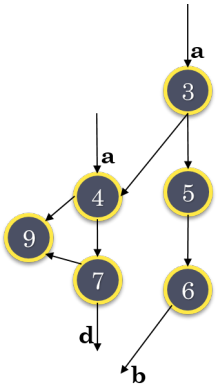
redundantes este nó é membro utilizando o atributo *SRT_member*. Em seguida, é verificado se o contexto de entrada dos subgrafos redundantes aos quais o nó pertence estão completamente preenchidos. Essa verificação ocorre para cada subgrafo, obedecendo a ordem do maior para o menor. Quando é verificado que um contexto de entrada de um subgrafo redundante está completamente preenchido, o subgrafo é reutilizado. Desta forma, é garantido que o maior subgrafo sempre será reutilizado.



Id	Inp_Ctx	Out_Ctx	Tamanho	Membros
1	(4,0,a), (4,1,b)	(8,0,d)	3	(4,7,9)
2	(3,0,a), (4,0,a)	(8,0,d), (8,1,b)	6	(3,4,5,6,7,9)

Indx	Inp_Op	Out_Op	Node Type	Node ID	Node Group
0	a	b	Azul	3	1
0	(a,b)	c	Azul	4	1
1	c	d	Azul	7	1
1	b	a	Azul	2	1

(a) Estados dos componentes da DF-DTM em cada instante t_i da execução



(b) Subgrafo redundante criado por conta da execução do grafo

Figura 6.7: Demonstração do funcionamento da DF-DTM na Sucuri

A figura 6.7a apresenta um exemplo detalhado do funcionamento das *caches* da DF-DTM. Temos, neste exemplo, um grafo *dataflow* com um subgrafo redundante já construído, conforme apresentado na tabela SRT, no quadro inferior à direita. Vemos que este possui o identificador 1. Também temos uma tabela NRT previamente preenchida. Os quadros *Buffer de Construção* e *Fila de Espera* apresentam o estado dessas estruturas em cada instante t_i de execução do grafo. A execução começa em

t_1 com o nó do tipo *verde*. Em t_2, o nó 2 e 3 recebem seus operandos de entrada. Eles são colocados em uma fila de espera para que sejam analisados. O nó 2 é removido da fila, porém, como não existe resultados para este nó na SRT ou NRT, o mesmo instancia uma tarefa que será executada posteriormente. O nó 3, em contrapartida, possui esses valores de entrada identificados na NRT. Por conta disso, ele é ignorado, e seus operandos são propagados aos dependentes. Como o nó 3 é o primeiro a ser detectado como redundante, o mesmo é adicionado ao *buffer de construção*. Repare que, neste momento, um novo subgrafo começa a ser construído tendo como raiz o nó 1 e como membro o nó 3. Vemos que em t_3, o nó 2 já foi computado por um *worker* e o seu operando de saída agora é conhecido. Antes desse operando ser propago ao seu destino, uma nova entrada é adicionada à NRT, a qual representa o valor de saída a, quando a entrada é b e o nó é do tipo *azul*. Ainda em t_3, o nó 4 e 5 são colocados na fila de espera por já estarem disponíveis para execução. O nó 4 é removido da fila, e é verificado que ele é membro do subgrafo 1, também é verificado que este subgrafo pode ser reutilizado. Logo, o subgrafo redundante 1 é integrado ao subgrafo em construção, e o operando por ele produzido é propagado ao destino, conforme especificado por seu contexto de saída. O nó 5 também é analisado no instante t_3. Este nó é detectado como redundante, pois consegue reutilizar o resultado produzido por 2. O nó 5 também é adicionado ao subgrafo em construção, e seu resultado é propagado ao nó 6. Em t_4, o nó 6 também é detectado como redundante, graças a um *hit* na NRT, e seu resultado é propagado para o nó 8. Por fim, no instante t_5, o nó 8 é analisado, e é verificado que o seu resultado não é conhecido. Por conta disso, o mesmo instancia uma tarefa. Neste momento, não há mais nenhum caminho redundante para fora do subgrafo em construção, e a construção deste termina. Note que por conta disso uma entrada nova é adicionada à SRT e o *buffer de construção* é esvaziado.

A figura 6.7b apresenta o subgrafo formado por conta da execução exemplificada na figura 6.7a

6.3 Mecanismo de inspeção

Em arquiteturas *Dataflow* é comum existir uma fila de espera que armazena todas as tarefas despachadas, enquanto não existem recursos disponíveis para execução destas, por exemplo, quando todos os EPs estão ocupados executando outras tarefas. Entretanto, enquanto estas tarefas estão aguardando sua execução, operandos resultantes são produzidos a todo momento, gerando uma oportunidade de reúso para tarefas que estão aguardando na fila.

O mecanismo de inspeção realiza uma inspeção nessas filas utilizando os operandos recém-recebidos dos EPs para validar se as tarefas ali presentes podem ser

reutilizadas, e, portanto, removidas da fila, gerando um melhor aproveitamento de recursos da máquina. Este mecanismo pode ter uma complexidade *O(n)*, caso exista somente uma unidade de comparação para todos os elementos da fila de tarefas prontas, ou *O(1)*, caso exista um comparador para cada elemento da fila realizando comparações de valores em paralelo.

A implementação deste mecanismo na Sucuri ocorre da seguinte forma: enquanto tarefas aguardam na fila de prontos, mensagens de operandos são entregues pelos *workers* ao escalonador central. Estas mensagens possuem os valores dos operandos de entrada utilizado pelo nó, tipo do nó, identificador do nó, identificador grupo do nó e operandos de saída produzidos pelo nó. Os valores dos operandos de entrada presentes na mensagem são comparados com os dos operandos de entrada presentes no objeto *task*(tarefas) da Sucuri. Esta comparação ocorre para todas as tarefas na fila de prontos. Se ocorrer um *match*, a tarefa é removida da fila e os resultados presentes na mensagem de operandos são propagados aos nós dependentes da tarefa.

Os esquemas de *caches* NRTs, isto é, local, global ou compartilhada, possuem influência sobre a forma que a inspeção ocorre. Na Sucuri combinada com DF-DTM, objetos *task* possuem informações sobre o identificador do nó, grupo do nó e tipo do nó que foi utilizado para instanciar a tarefa. Esses campos são utilizados para que, no momento da inspeção, o reúso da tarefa só ocorra se o operando for produzido pelo mesmo nó, no caso de um esquema de NRTs locais, ou por um membro do mesmo grupo de nós, caso o esquema utilizado seja de NRTs compartilhadas. Para uma NRT global, para ocorrer o reúso das tarefas em espera, basta que o tipo do nó seja o mesmo, além dos operandos de entrada.

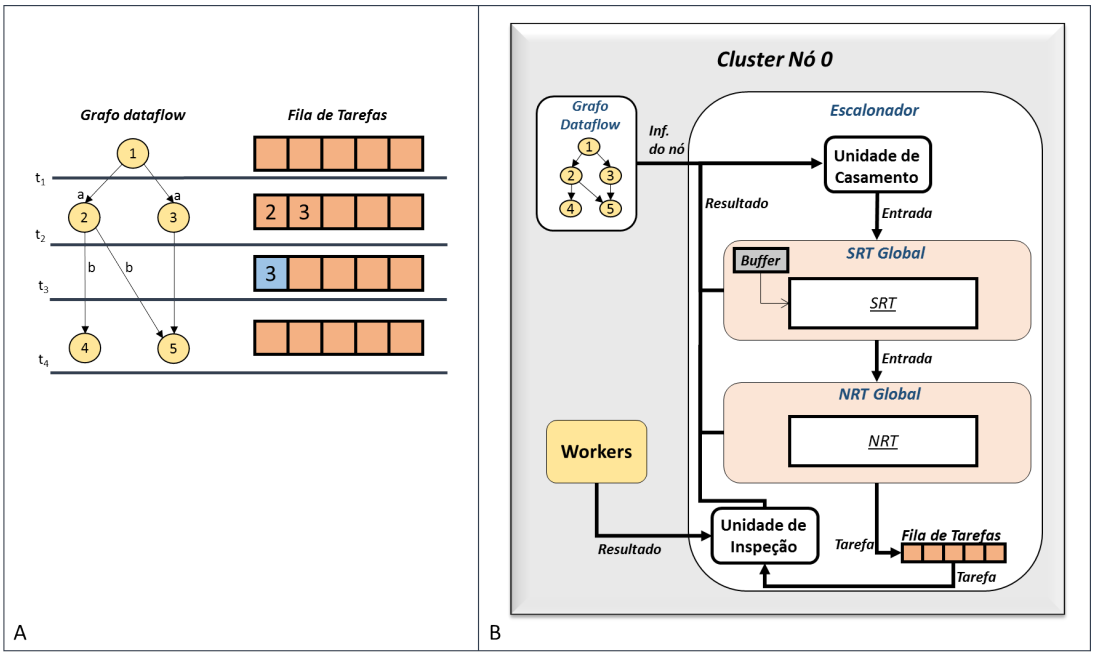


Figura 6.8: Organização do escalonador central da Sucuri com NRTs locais

A figura 6.8 apresenta à esquerda, no quadro A, um exemplo de execução com o mecanismo de inspeção ativado. Neste quadro, vemos que em t_1 a fila está vazia, pois o nó 1 está executando. Em t_2 os operandos são recebidos e propagados aos dependentes 2 e 3. Estes dão origens às tarefas 2 e 3 que ficam armazenadas na fila de tarefas prontas. Em t_3, a tarefa 2 é executada por um *worker* e o resultado retornado ao escalonador central. Neste momento, a inspeção é ativada e descobre-se que a tarefa 3 utiliza os mesmos operandos de entrada da tarefa 2, logo o resultado de 2 se aplica à 3. Em t_4 a tarefa 3 é removida da fila, mas não executada, pois ela foi reutilizada. O quadro à direita, B, apresenta a organização do escalonador com uma unidade de inspeção, NRT global e uma SRT.

6.4 Estratégias de detecção de reúso

Nesta seção dissertamos sobre as diferentes estratégias de reúso implementadas na Sucuri com a DF-DTM.

O *pipeline* da Sucuri permite que a detecção de redundância ocorra em momentos distintos. Estes momentos são os seguintes:

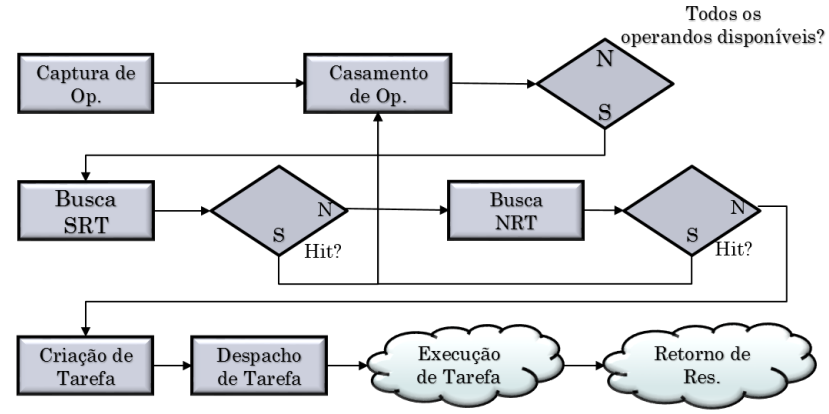
- Assim que uma mensagem de operando é recebido por algum *worker*. Esta estratégia é denominada *Eager Match* (EM).
- Após todas as mensagens de operandos serem recebidas. Neste caso a detecção de reúso só ocorre quando a fila de mensagens está vazia. Esta estratégia é denominada *Lazy Match* (LM).
- No momento em que um *worker* ocioso solicita uma tarefa ao escalonador central. Esta estratégia é denominada *Lazy Ready Queue* (LRQ).

As subseções seguintes explicam em mais detalhes cada estratégia.

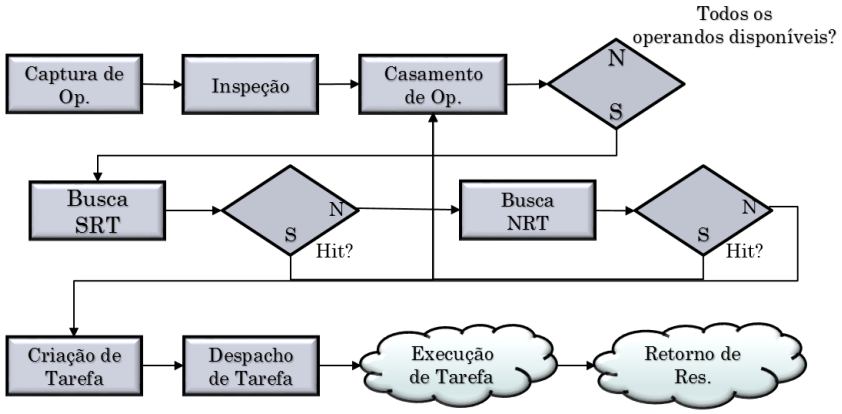
6.4.1 *Eager Match* (EM) e *Eager Match + Inspection* (EM+I)

A estratégia EM detecta o reúso no momento em que uma mensagem de operandos m_{op} referente a execução de um nó v chega ao escalonador central. Isto significa que, quando um *worker* da Sucuri envia um resultado ao escalonador central, assim que o resultado é recebido, o processo de propagação de dados é iniciado. O nó que o *worker* executou torna-se uma raiz de um possível subgrafo redundante. Os dependentes deste nó que agora possuem todos os operandos de entrada disponíveis são analisados para o reúso. Esta estratégia permite a criação de um subgrafo por vez.

A estratégia *Eager Match + Inspection* consiste na estratégia EM com o mecanismo de inspeção ativado. Neste caso, quando uma mensagem m_{op} é retornada ao escalonador, é verificado se as tarefas $t_1, t_2, ..., t_n$ instanciadas a partir dos nós $v_1, v_2, ..., v_n$, podem ser reutilizadas com base nas informações da mensagem m_{op}. Cada tarefa reutilizada poderá criar um subgrafo redundante que possua como raiz o nó que a tarefa representava.



(a) Estratégia EM



(b) Estratégia EM+I

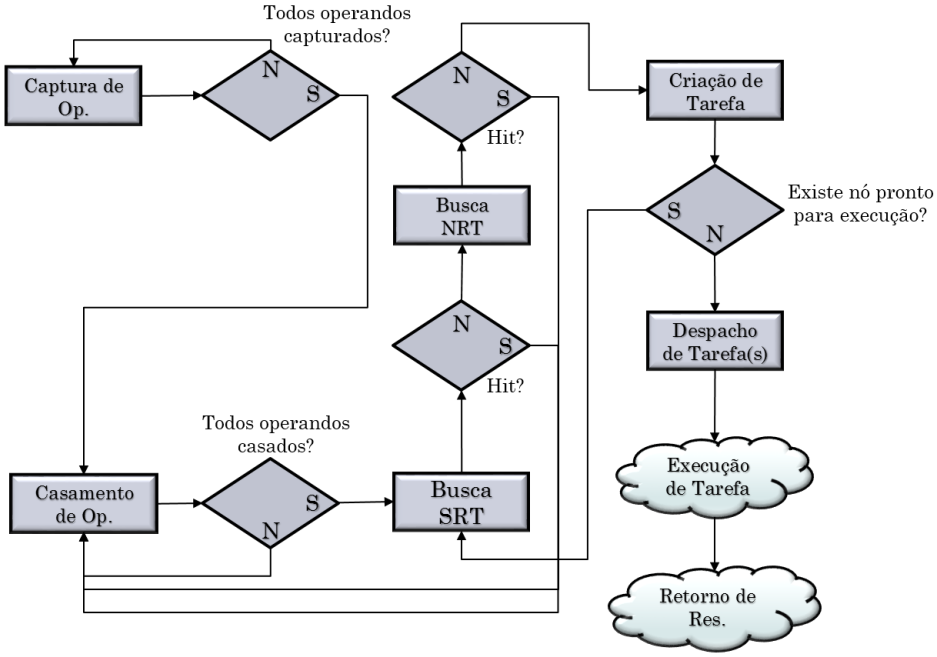
Figura 6.9: *Pipeline* da Sucuri com as estratégias EM e EM+I

A figura 6.9 mostra as etapas no processo de execução *dataflow* para a estratégia EM e EM+I. Os símbolos em forma de nuvem apresentam as etapas que são executadas por *workers*, fora do escalonador da Sucuri. As demais etapas são realizadas no escalonador central. Note que esta estratégia de detecção imediata permite que haja detecção de reúso sem gerar atrasos significativos na execução *dataflow*.

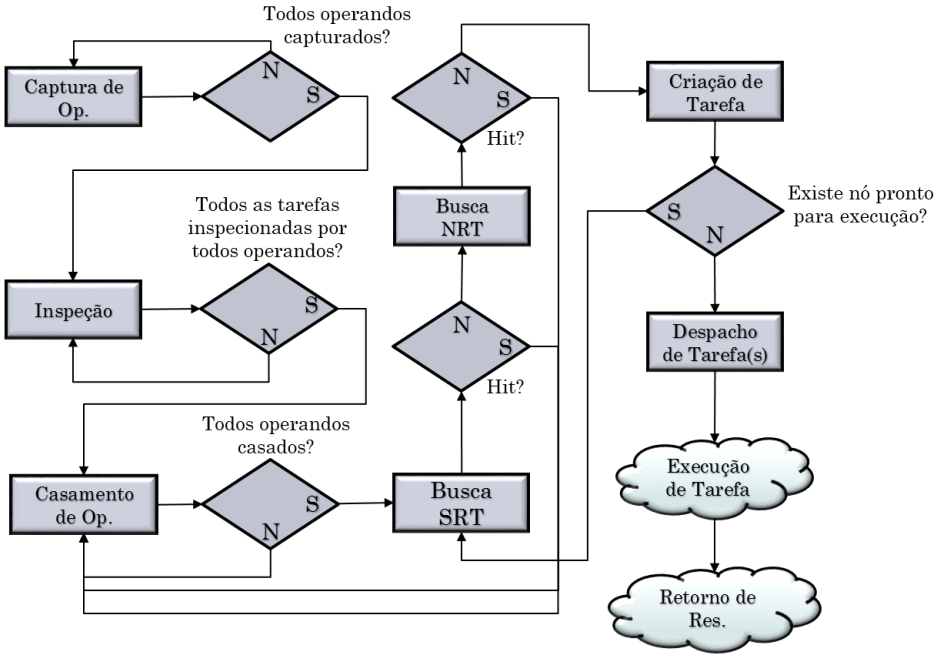
6.4.2 *Lazy Match* (LM) e *Lazy Match + Inspection* (LM+I)

A estratégia *Lazy Match* (LM) começa a propagação de operandos no grafo quando não existe mais nenhuma mensagem de operandos m_{op} aguardando seu processa-

mento, diferentemente da estratégia EM, que começa a propagação assim que uma mensagem é lida. Nesta estratégia, antes da construção de um subgrafo redundante começar, todos os operandos resultantes contidos nas mensagens lidas são disponibilizados aos nós destinatários, fazendo com que mais nós estejam disponíveis para instanciação de tarefas. Cada mensagem m_{op_i} é proveniente da execução de um nó v_i. Portanto, nesta estratégia, a construção de subgrafos é realizada para cada raiz v_i.



(a) Estratégia LM



(b) Estratégia LM+I

Figura 6.10: *Pipeline* da Sucuri com as estratégias LM e LM+I

A construção do subgrafo começa por uma raiz v_i, e, conforme nós redundantes são encontrados a partir do nó raiz, estes são adicionados ao subgrafo em construção. Quando a construção do subgrafo a partir da raiz v_i termina, a tentativa de construção de outro subgrafo se inicia para a próxima raiz v_{i+1} até não haver mais raízes disponíveis.

A estratégia *Lazy Match + Inspection* consiste na estratégia LM com o mecanismo de inspeção ativado. Ela permite que ainda mais nós sejam utilizados como raízes, pois considera os nós que instanciaram as tarefas que foram removidas das filas de prontos. Isso permite também que ainda mais operandos sejam propagados aos nós destinos, permitindo um maior preenchimento dos contextos de entradas de subgrafos redundantes existentes.

A figura 6.10 mostra as etapas no processo de execução *dataflow* para a estratégia LM e LM+I. Note que esta estratégia tenta consumir ao máximo os operandos recebidos antes de instanciar tarefas. Isso é feito com o intuito de aumentar o reúso de subgrafos.

6.4.3 *Lazy Ready Queue* (LRQ)

Lazy Ready Queue é uma estratégia de detecção tardia. Enquanto as estratégias EM, LM e suas variantes se baseiam no recebimento de mensagens para começarem a detecção de redundância, a estratégia LRQ começa a detecção no momento em que um *worker* ocioso solicita uma nova tarefa. Por conta desta detecção tardia, esta estratégia despensa o uso do mecanismo de inspeção, pois ela por si só já realiza detecção de tarefas redundantes à espera de recursos.

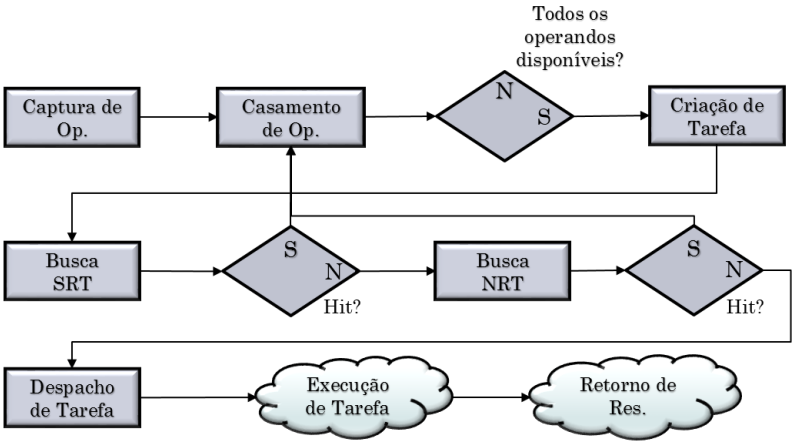


Figura 6.11: *Pipeline* da Sucuri com a estratégia LRQ

Para LRQ, quando um *worker* solicita uma tarefa, é verificado se esta é redundante. Se ela for a primeira tarefa redundante a ser detectada, se inicia a criação de um subgrafo, caso contrário, ela é adicionada ao subgrafo em formação. Quando

uma tarefa é detectada como redundante, seus operandos resultantes são propagados aos nós dependentes. Caso um nó v possua todos os operandos de entrada disponíveis por conta desta propagação, uma nova tarefa t é instanciada a partir de v. A tarefa t é adicionada no início da fila de prontas indicando que ela foi originada por consequência de uma detecção de tarefa redundante. Quando um *worker* solicitar uma nova tarefa, será verificado se t é redundante, caso sim, ela é adicionada ao subgrafo. Quando não houver mais tarefas redundantes armazenada de forma contígua na fila, a construção do subgrafo termina.

A figura 6.11 mostra as etapas no processo de execução *dataflow* para a estratégia LRQ. Esta estratégia realiza uma detecção tardia, evitando o uso de inspeção e aumentando o espectro de reúso.

Capítulo 7

Resultados Experimentais

Neste capítulo apresentamos uma série de experimentos que têm por objetivo avaliar o potencial da técnica de reúso de computação no modelo *Dataflow*. Exploramos diferentes estratégias para detectar reúso em cenários ideais e em cenários com recursos restritos. Utilizamos os 5 *benchmarks* descritos na seção 7.1 para realização dos experimentos. As demais seções deste capítulo descrevem a metodologia e os resultados alcançados.

7.1 Benchmarks

Para implementação dos *benchmarks* descritos nas subseções seguintes, utilizamos DAGs de *Stream*. Por conta disso, todas as aplicações possuem nós *Source*, os quais lêem um objeto iterável e alimentam o grafo *dataflow*. Elas também possuem nós *Serializer*, que realizam operações obedecendo a ordem em que o nó *Source* emitiu os operandos. Isso foi feito para que pudéssemos explorar o reúso em uma mesma execução e em execuções distintas de um mesmo *benchmark*.

7.1.1 LCS - Longest Common Subsequence

O objetivo desta aplicação é encontrar uma subsequência comum de maior tamanho entre duas sequências de caracteres. Dadas duas sequências A e B, uma solução com programação dinâmica preenche uma matriz M de tamanho $[(A+1) \times (B+1)]$, com base na seguinte relação de recorrência:

$$M[i,j] = \begin{cases} 0, \text{ se } i = j = 0 \\ M[i-1, j-1] + 1, \text{ se } (i,j > 0) \wedge (a_i = b_j) \\ max\{M[i-1,j], M[i,j-1]\}, \text{ caso contrário} \end{cases} \tag{7.1}$$

Ao final do algoritmo, o último elemento de M armazenará o tamanho da maior

subsequência comum. Uma operação conhecida como *traceback* pode então ser utilizada para descobrir uma subsequência comum com este tamanho. Nesta versão não é realizado o *traceback*, devido ao baixo custo computacional de tal operação.

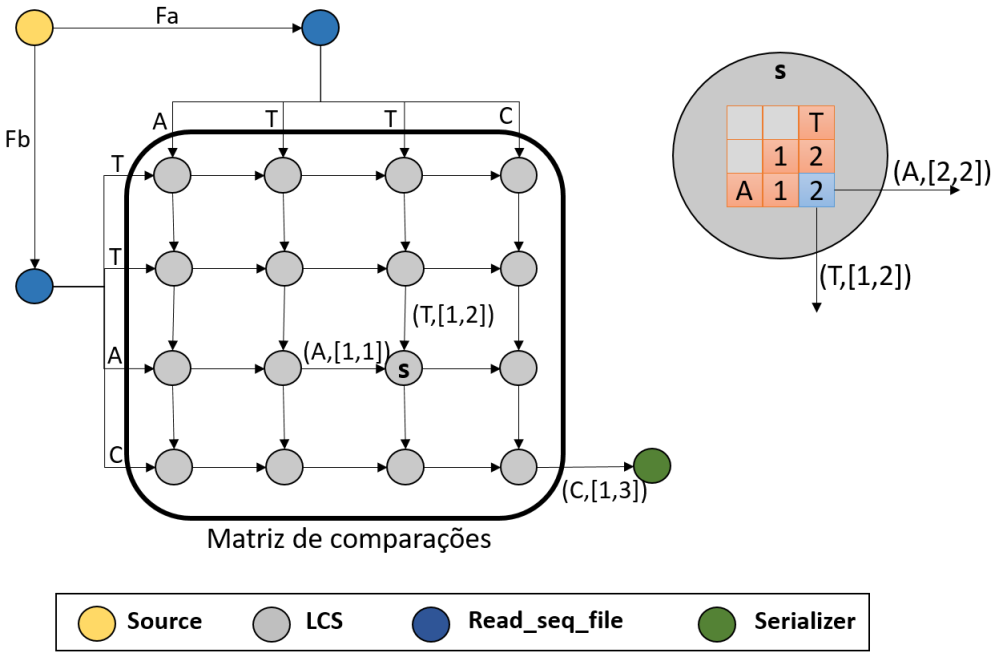


Figura 7.1: Exemplo de um grafo LCS *dataflow*.

A paralelização adotada segue o padrão *wavefront*[56], onde a matriz é dividida em blocos, e cada bloco tem dependências de dados com seus vizinhos superiores e esquerdos. O bloco pode ser um conjunto de elementos ou até mesmo um único elemento da matriz. A implementação para Sucuri deste *benchmark* é baseada em [6] e foi modificada para fazer a comparação de uma sequência biológica com uma lista de sequências biológicas em um *pipeline wavefront*. O primeiro nó do grafo é um nó do tipo *Source*, que recebe uma lista de arquivos com as sequências a serem processadas. Existem dois nós do tipo *read_seq_file*. Cada nó recebe um arquivo do *Source*, e envia os caracteres contidos nos arquivos para um subgrafo que representa a matriz de comparações. Um nó *read_seq_file* sempre receberá o mesmo arquivo do *Source*, a saber, o arquivo que contém a sequência que se deseja comparar às demais. A matriz é um subgrafo *wavefront* com nós do tipo "LCS"que computam o valor do elemento ou bloco de elementos da matriz. Finalmente, os resultados produzidos pelo último nó LCS são encaminhados para o nó *Serializer*, o qual escreve, de forma ordenada, os resultados dos alinhamentos em um arquivo. Os nós *Source* e *Serializer* não são candidatos ao reúso, pois estes não são determinísticos.

A figura 7.1 apresenta um exemplo do grafo LCS. O nó *Source* envia o arquivo "Fa"para um dos nós *read_seq_file* e o arquivo "Fb"para o outro nó do mesmo tipo. Os nós *read_seq_file* distribuem os caracteres para os nós "LCS"das extremidades da matriz. Estes nós, por sua vez, aplicam a recorrência da equação 7.1, e propagam

o resultado e os caracteres recebidos para os vizinhos abaixo e à direita. O nó marcado como "s"é exemplificado em detalhes. Note que este recebe os caracteres necessários à computação do vizinho acima e do vizinho à esquerda. Além disso, ele também recebe os valores dos elementos $[i-1, j-1]$ e $[i-1, j]$ do vizinho acima e $[i-1, j-1]$ e $[i, j-1]$ do vizinho à esquerda. Este nó recebe duas vezes o mesmo elemento $[i-1, j-1]$. Isto foi feito para simplificar a construção do grafo. O último nó "LCS", envia ao *Serializer* o valor final da matriz de comparações, a saber, o número 3, neste exemplo. Por fim, o *Serializer* apresenta este número em um dispositivo de saída.

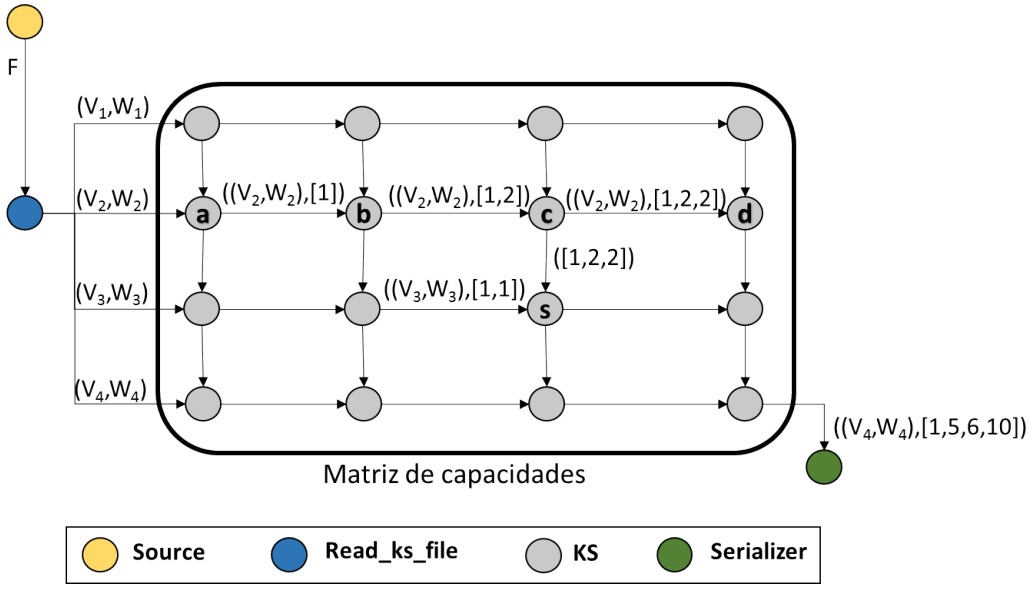
7.1.2 *Unbounded Knapsack*

Neste *benchmark*, implementamos o problema da mochila sem limitação, *Unbounded Knapsack*, que consiste em, dado um conjunto de tuplas (w_i, v_i), onde w_i é o peso de um item i e v_i é o seu valor, encontrar o subconjunto de itens que maximiza o lucro sem exceder a capacidade C da mochila [57, 58]. O algoritmo usa uma matriz M de programação dinâmica para calcular soluções do problema para mochilas de capacidades menores, e utiliza esses resultados para descobrir a solução para mochila de capacidade C. Cada elemento $M[i, j]$, exceto aqueles da primeira linha e da primeira coluna, é computado da seguinte maneira: $max(l \times v[i] + M[i-1, j-l*w[i])$ onde l varia de 0 a $j/w[i]$ (o número de itens i que a mochila comporta). $v[i]$ e $w[i]$ são os valores e os pesos de cada item i, respectivamente, e j é a capacidade da mochila que está sendo analisada.

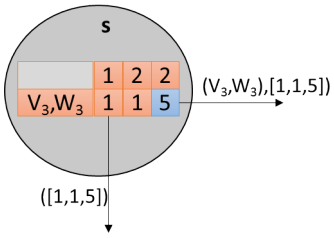
A forma trivial de se paralelizar este algoritmo é computar em paralelo os elementos de uma linha da matriz, e, depois, utilizar uma barreira para sincronizar as *threads* e iniciar o processamento da próxima linha. Como, para este problema, não é possível dividir o trabalho do processamento da matriz entre os nós do grafo *dataflow* sem executar o algoritmo inteiro, a abordagem *Dataflow* utiliza o modelo *Wavefront* de forma similar ao LCS. Ela divide cada linha da matriz em n partições, e insere dependências de forma que a partição $p[i, j]$ somente seja processada após a conclusão das partições $p[i-1, j-1]$ e $p[i, j-1]$. Apesar dessa abordagem evitar o uso de barreiras, ela insere falsas dependências.

O grafo *dataflow* deste *benchmark* computa várias mochilas. Ele é composto por um nó *Source*, um subgrafo que representa a matriz de programação dinâmica dividida em partições, e um nó *Serializer* que escreve o resultado do lucro máximo para a mochila em um dispositivo de saída. O nó *Source* utiliza um objeto iterável cujo conteúdo são n arquivos. Cada arquivo descreve o peso e valor de um conjunto de itens a ser armazenado em uma mochila de capacidade C. Este nó envia cada arquivo ao nó *read_ks_file*, que, por sua vez, propaga os dados dos itens aos nós

"KS". Estes representam a matriz de programação dinâmica dividida em partições. Cada nó "KS"pode ser responsável por computar um conjunto de elementos da matriz, em uma estratégia de granularidade mais grossa, ou apenas um elemento da matriz, para uma abordagem de granularidade fina. Por fim, o último nó "KS"envia o resultado da última partição ao nó *Serializer*, que irá escrever o lucro máximo para a mochila representada pelo arquivo transmitido pelo nó *Source*. A ordem de escrita do nó *Serializer* obedece a ordem em que os arquivos foram emitidos pelo nó *Source*.



(a) Grafo detalhado.



(b) Nó "s"detalhado.

Figura 7.2: Exemplo de um grafo para a aplicação *Unbounded knapsack*.

A figura 7.2a mostra um exemplo de grafo *dataflow* para esta aplicação. O nó *Source* envia ao nó *read_ks_file* o arquivo "F"presente em um objeto iterável, o qual contém os valores dos itens e seus respectivos pesos. O nó *read_ks_file* envia esses dados na forma da tupla (V_i, W_i) à primeira coluna de nós "KS". Os nós "KS"computam o valor de uma célula da matriz ou um conjunto de células, e propaga os resultados aos nós "KS"vizinhos. Além disso, os nós "KS"irão propagar para os seus vizinhos à direita a tupla $(w[i], v[i])$. O nó "s"é exemplificado com mais detalhes na figura 7.2b. Ele recebe uma partição do nó "KS"acima e outra do nó

"KS"à esquerda. Do nó à esquerda, ele também recebe uma tupla contendo o valor e peso do item. O nó "s"utiliza esses valores para calcular o resultado do elemento, a saber, o valor 5, no exemplo. Depois, ele propaga a tupla e a partição $[1, 1, 5]$ para o vizinho à direita, e somente a partição para o vizinho abaixo. Por fim, o último nó "KS"envia a tupla e a partição final ao nó *Serializer*. Este escreve o resultado em um arquivo.

Note que o tamanho dos operandos aumenta conforme a execução avança, conforme exemplificado na figura 7.2. Por exemplo, o nó "a"propaga o seu resultado (1) para o seu vizinho, "b", por sua vez, o nó "b"computa o resultado do elemento da matriz, concatena com o resultado de "a", e o envia ao próximo nó, "c". Este comportamento se repete para todos os nós que representam a matriz. De forma mais geral, isso ocorre pois o nó $N[i, j]$ precisa conhecer o resultado de todas as partições $p[i-1, 0]$ à $p[i-1, j]$ para computar a partição $p[i, j]$.

7.1.3 *MapReduce*

O contador de palavras *MapReduce* [58] é uma aplicação paralela do tipo *fork-join* que lê múltiplos arquivos, cada um contendo palavras separadas por espaços, e calcula a frequência de ocorrência destas palavras. O modelo de programação *MapReduce* descreve a criação de um programa na forma de funções mapeadoras e funções redutoras. A função mapeadora associa um valor a uma chave, gerando uma tupla. Neste *benchmark*, cada palavra é associada ao valor 1. A função redutora recebe as tuplas e aplica uma operação de agregação a elas.

O grafo da aplicação *MapReduce* segue o padrão *fork-join* com redução hierárquica. O nó *Source* envia os caminhos dos arquivos para o nó de mapeamento, *mapper*. O nó mapeador abre estes arquivos, e associa todas as palavras neles contidas com o valor 1, criando a tupla $(palavra, 1)$. Estas tuplas são enviadas aos nós redutores, *reducers*. Cada redutor realiza uma operação de agregação aos grupos de tuplas recebidos, e envia o resultado da agregação para um redutor de nível inferior. O redutor de nível inferior recebe dois ou três grupos de tuplas dos seus vizinhos de nível superior e executa mais uma vez a operação de agregação. O último redutor envia ao nó *Serializer* o grupo final de tuplas que representa a ocorrência total de palavras em um arquivo específico. O nó *Serializer* escreve este resultado em um dispositivo de saída. Assim como nas aplicações LCS e *Unbounded Knapsack*, os nós *Source* e *Serializer* não são candidatos para o reúso.

A figura 7.3 apresenta um exemplo do grafo *MapReduce* para 8 *reducers*. O nó *Source* envia o arquivo "F"ao nó *mapper*. Este lê o arquivo, associa as palavras ao valor 1 e distribui essas palavras entre os *reducers* de primeiro nível. Os *reducers* agregam o conjunto de tuplas recebidas, e enviam o resultado para o próximo *reducer*.

Repare que, por conta disso, os operandos tendem a crescer conforme a execução da aplicação avança. Por fim, o último *reducer* envia a frequência de todas as palavras contidas em um arquivo para o nó *Serializer*, o qual grava este resultado em um arquivo de saída.

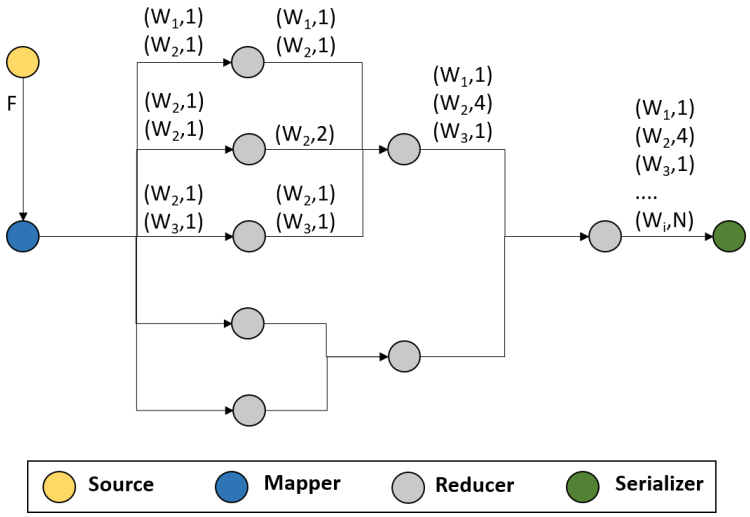


Figura 7.3: Exemplo de um grafo MapReduce *dataflow*.

7.1.4 GoL - Game of Life

O *benchmark* GoL consiste em uma implementação *Dataflow* do algoritmo *game of life* [59]. Este descreve o comportamento de populações de células através de um tabuleiro, representado por uma matriz, onde cada elemento desta representa uma célula "viva"(valor 1 para o elemento da matriz) ou "morta"(valor 0 para o elemento da matriz). Uma célula morre de superpopulação, ou seja, o valor do elemento da matriz passa de 1 para 0, quando existem em seus 8 elementos vizinhos uma quantidade de elementos com valor 1 maior que ou igual a 4. De forma semelhante, a célula também morre de "solidão"quando existem menos do que 2 elementos de sua vizinhança com valor 1. Uma célula morta renasce quando exatamente 3 elementos de sua vizinhança possuem valor 1. A célula permanece viva quando 2 ou 3 de seus vizinhos também estão vivos. O algoritmo pode realizar n iterações. A cada iteração os elementos são calculados, gerando novos padrões de células vivas e mortas. Conforme as iterações ocorrem, comportamentos das células se estabelecem e, em alguns casos, padrões recorrentes são criados, por exemplo, um inchaço populacional seguido por uma mortificação da população.

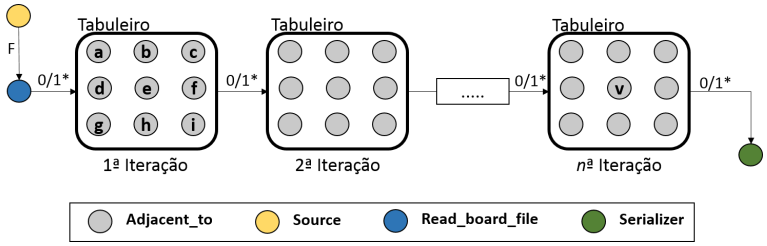
A implementação paralela trivial para este problema seria computar em paralelo cada elemento do tabuleiro, realizar uma sincronização de todas as *threads*, e depois repetir o processo n vezes, onde n é o número de iterações desejadas. Entretanto, neste tipo de paralelização, vemos que *threads* ficarão ociosas, quando determinada

região do tabuleiro for computada, pois elas aguardarão o término de processamento de outras regiões para começar o processamento da próxima iteração. O paradigma *Dtaflow* permite que regiões dos tabuleiros sejam computadas independentemente, permitindo assim melhor uso do paralelismo sem a necessidade de uma sincronização. Quando os operandos necessários para computar uma determinada região do tabuleiro na iteração $n + 1$ estiverem prontos, esta região será processada, mesmo que existam outras regiões do tabuleiro referente a iteração n ainda em processamento.

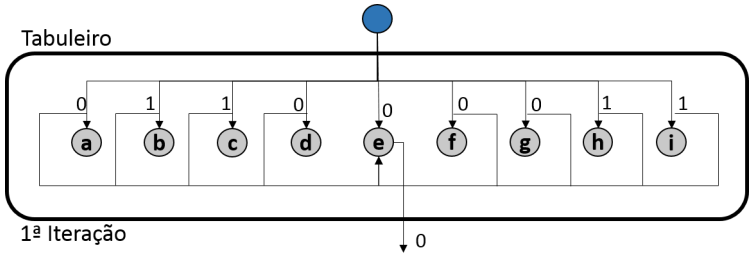
Na implementação Sucuri deste algoritmo, temos um nó *Source*, o qual possui um objeto iterável com n caminhos de arquivos, cada um deles contendo a configuração inicial de um tabuleiro. Estes caminhos são enviados ao nó *read_board_file*. O nó *read_board_file* envia aos nós *adjacent_to*, responsáveis por calcular o estado de uma célula, os operandos da vizinhança desta célula. O grafo *dataflow* possui uma representação do tabuleiro, onde cada nó do grafo é um elemento ou um bloco de elementos da matriz. Portanto, o nó *read_board_file* enviará, no máximo, 9 operandos para cada nó *adjacent_to*. Estes operandos são os valores dos vizinhos das células adjacentes e o valor corrente da célula que o nó representa. Por conta deste algoritmo permitir n iterações, é necessário que a representação em grafo *dataflow* acíclico do tabuleiro também seja repetida n vezes. Logo, temos que, os nós *adjacent_to* da iteração $n - 1$ irão enviar aos seus vizinhos da iteração n o seu estado ("célula viva"ou "célula morta"). Esse padrão de comunicação e processamento se repete até a última iteração. Nesta iteração os nós do tipo *adjacent_to* enviam os seus operandos para um nó *Serializer*, o qual organiza os operandos na forma de um tabuleiro e cria uma imagem com o seu conteúdo.

A figura 7.4a exemplifica a criação do grafo para um tabuleiro $3X3$. O nó *Source* envia o arquivo "F"para o nó *read_board_file*. Este nó, lê o arquivo, e envia todos os operandos aos nós responsáveis pelo processamento da primeira iteração do tabuleiro. Por exemplo, o nó "a"receberá, além do seu valor 1 ou 0, os valores dos nós vizinhos a ele, ou seja, os nós "b", "d"e "e". Na figura 7.4b, vemos a representação detalhada do subgrafo do primeiro tabuleiro. Para simplificar a ilustração são representadas, em totalidade, apenas as arestas do incidentes ao nó "e". Note que este nó recebe o estado de todos os elementos vizinhos, computa o estado da célula que ele representa, e envia o resultado à todos os vizinhos desta célula referentes à próxima iteração. Apesar desse comportamento estar apenas representado para o nó "e", os demais nós também realizam o mesmo procedimento. A figura 7.4c apresenta a última e penúltima iteração do algoritmo. Os nós "s"e "v"representam a mesma célula, porém em iterações distintas. Apesar de não especificado na figura, o nó "s"da iteração $n - 1$ envia o estado de sua célula para todos os outros da próxima iteração, incluindo o nó "v". Os demais nós da iteração $n - 1$ também fazem o mesmo. Todos os nós da iteração n computam o estado de suas células e enviam os

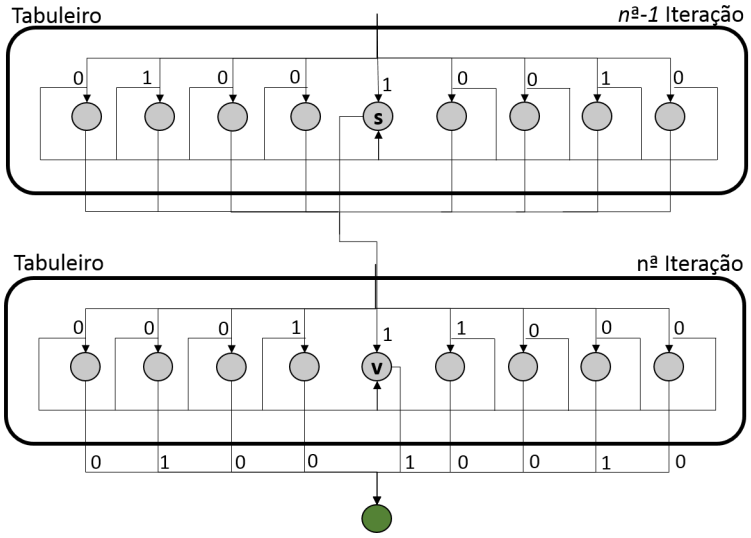
seus resultados para o nó *Serializer*. Este escreve a imagem final do tabuleiro.



(a) Grafo simplificado da aplicação GoL.



(b) Grafo detalhado do tabuleiro da primeira iteração.



(c) Grafo detalhado do tabuleiro da última e penúltima iteração.

Figura 7.4: Grafo *dataflow* para o *benchmark* GoL.

7.1.5 3-DES - *Triple Data Encryption Standard*

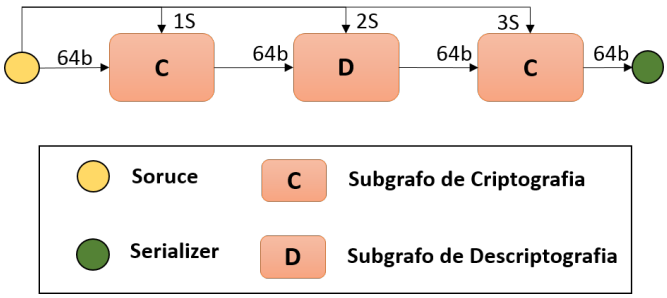
Este *benchmark* implementa a criptografia 3-DES [60] utilizando o modo de operação ECB (*Electronic CodeBook*). A aplicação de criptografia 3-DES utilizando este modo de operação consiste em um *pipeline*, onde um ou mais blocos de 8 bytes são enviados de estágio em estágio desse *pipeline* sofrendo alterações. O 3-DES utiliza uma chave principal de 64 *bits*. O processo de criptografia começa com uma

primeira permutação em um bloco de 64 *bits* (8 *bytes*). Após a primeira permutação, o bloco é dividido em duas partes de 32 *bits*, e se inicia a aplicação da função F na metade direita do bloco. Um XOR é realizado sobre a metade esquerda do bloco e o resultado dessa função. O Resultado deste XOR é enviado para uma segunda chamada da função F. Desta vez, a função F é aplicada sobre a metade esquerda do bloco, e um XOR é realizado entre o resultado desta função com o resultado do XOR anterior, referente ao resultado da primeira chamada à função F. Este processo ocorre de forma alternada 16 vezes, conforme demonstrado na figura 7.5b. Para cada uma das 16 fases é utilizada uma subchave de 48 *bits* criada a partir da chave principal de 64 *bits*. Após a aplicação da última função F é realizada uma permutação final. A descriptografia utiliza os mesmos passos, mas as funções F são aplicadas de forma inversa. Para o 3-DES, a aplicação de criptografia recebe 3 chaves de 64 *bits*, a primeira chave é utilizada para criptografar o bloco, a segunda serve para descriptografar o que foi criptografado com a primeira chave, e, por fim, a terceira chave criptografa novamente o conteúdo falsamente descriptografado pela segunda chave.

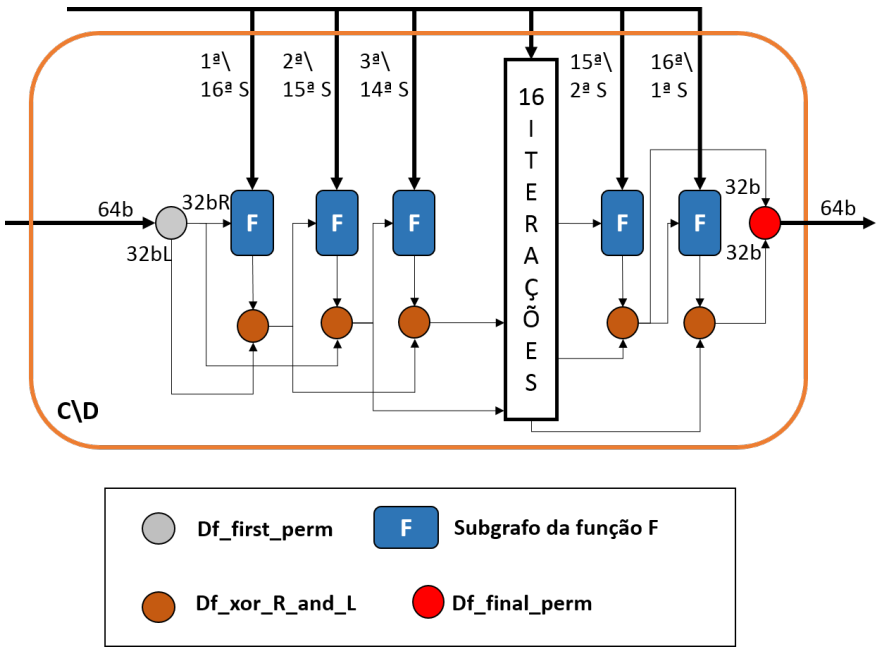
A função F é composta por uma permutação expansiva que aumenta a metade de 32 *bits* para 48 *bits*. Em seguida, é realizado um XOR entre a metade estendida e uma das subchaves criadas. Após isso, o bloco resultante de 48 *bits* é dividido em 8 partes de 6 *bits*, e cada parte é traduzida para um bloco de 4 *bits* utilizando uma tabela de substituição chamada S-Box. Todas as partes de 4 *bits* geradas a partir dessas etapas são concatenadas, e, então, é realizada uma permutação utilizando uma tabela chamada P-Box.

A figura 7.5a apresenta o grafo simplificado da aplicação 3-DES. Primeiramente, o nó *Source* recebe um objeto iterável que contém todos os blocos de 8 bytes que deverão ser criptografados. Além disso, este nó possui as subchaves de criptografia pré-processadas como um atributo. As representações $1S$, $2S$ e $3S$ representam os 3 grupos de 16 subchaves criados a partir das 3 chaves originais. O nó *Source* é responsável por enviar cada bloco de 8 bytes, representado como $64b$ na figura 7.5a, ao nó do tipo *df_first_perm*, o qual é responsável pela primeira permutação dos *bits* do bloco. Conforme vemos na figura 7.5b, o nó *df_first_perm* envia a metade direita do bloco resultante, representado como $32bR$, ao nó *df_exp_perm*, dando início à primeira chamada da função F.

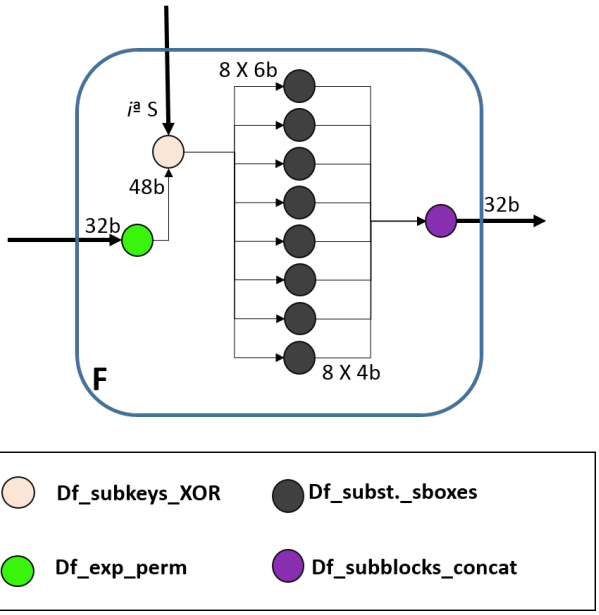
A função F, apresentada na figura 7.5c, possui o seguinte grafo: o nó *df_exp_perm* envia seu resultado, um segmento expandido de 48 *bits* ($48b$), ao nó *df_subkeys_XOR*, e o nó *Source* envia uma subchave a este nó de acordo com o número da chamada à função F. Por exemplo, no caso de criptografia, as chamadas $\{1, 2, 3, ...\}$ recebem as subchaves $\{1, 2, 3, ...\}$ respectivamente. Porém, para descriptografia, esta mesma sequência de chamadas receberiam as subchaves $\{16, 15, 14, ...\}$. Note na figura 7.5b,



(a) Grafo simplificado da aplicação 3-DES.



(b) Subgrafo do *pipeline* de criptografia\descriptografia.



(c) Subgrafo da função F.

Figura 7.5: Grafo *dataflow* para o *benchmark* 3-DES.

o subgrafo da função F mais à esquerda, que representa a primeira chamada desta função, recebe a primeira subchave ($1^{a}S$), caso seja um grafo de criptografia, ou a décima sexta subchave ($16^{a}S$), caso seja um subgrafo de descriptografia.

O resultado do nó *df_subkeys_XOR* é dividido em 8 partes de 6 *bits*, e cada uma é enviada a um nó do tipo *df_subst_sboxes*. Este realiza substituições de *bits* utilizando as *S-boxes* e gera um segmento de 4 *bits*. Cada nó *df_subst_sboxes* envia seu resultado ao nó *df_subblocks_concat*, que, por sua vez, concatena os segmentos de 4 *bits* recebidos e realiza a permutação com a *P-Box*, finalizando a função F, e produzindo um segmento de 32 *bits* (32*b*). O resultado do nó *df_subblocks_concat* é enviado para o nó do tipo *df_xor_R_and_L*, o qual realiza um XOR entre este resultado e um destes possíveis segmentos resultantes:

- Metade esquerda do segmento resultante do nó *df_first_perm*, representado por 32*bL* na figura 7.5b, quando o nó *df_xor_R_and_L* depende da primeira chamada à função F.
- Metade direita do bloco resultante do nó *df_first_perm* (32*bR*), quando nó *df_xor_R_and_L* depende da segunda chamada à função F.
- Resultado do nó *df_xor_R_and_L* dependente da $i-ésima-2$ chamada à função F.

Após 16 chamadas à função F, o nó *df_final_perm* recebe os segmentos resultantes de 32 *bits* (32*b*) dos nós *df_xor_R_and_L* posteriores às chamadas 16 e 15 da função F, conforme exemplificado na figura 7.5b. O nó *df_final_perm* realiza uma última permutação, e gera um novo bloco de 64 *bits* (64*b*), finalizando o grafo de criptografia ou descriptografia.

No padrão 3-DES, este grafo é repetido 3 vezes: A primeira vez realizando o procedimento de criptografia com a primeira chave, a segunda vez realizando a descriptografia com a segunda chave, e a última vez realizando uma nova criptografia com a terceira chave. Portanto, são necessários 2 grafos de criptografia e 1 de descriptografia, como mostra A figura 7.5a. Por fim, o nó *df_final_perm* do segundo grafo de criptografia envia o bloco criptografado para o nó *Serializer*. Este, por sua vez, ordena os blocos criptografados na mesma ordem em que saíram do nó *Source* e os escreve em um arquivo.

7.2 Ambiente de teste e metodologia

Todos os experimentos foram realizados em uma máquina virtual *Google Cloud* com 8 VCPUS, 32 GB de RAM e 10 GB de memória em disco.

As métricas utilizadas nos experimentos foram: taxa máxima de reúso, taxa real de reúso e taxa de distribuição real de reúso. A taxa máxima de reúso em um grafo *dataflow* é calculada como $1 - \frac{D(N)}{N}$, onde $D(N)$ é o número de nós dinâmicos distintos e N é o número total de nós dinâmicos criados durante a execução *dataflow*. Por outro lado, a taxa real de reúso é o número de tarefas redundantes que foram detectadas. Finalmente, a taxa de distribuição real de reúso é usada para medir por quais mecanismos as tarefas redundantes foram detectadas (*i.e* busca em NRT, SRT ou usando o mecanismo de inspeção). Como exemplo, imagine uma aplicação que possua 400 nós dinâmicos, mas apenas 100 nós distintos. A taxa máxima de reúso seria $1 - 100/400 = 0.75$ ou 75%. Agora, imagine que desses 75%, somente 50% tenham sido detectados, então 50% é a taxa real de reúso. A taxa de distribuição real de reúso poderia ser, por exemplo, 10% de acertos na SRT, 30% de acertos na NRT, e os restantes 10% através de mecanismo de inspeção.

Em todos os experimentos, as aplicações *benchmarks* foram executadas utilizando as seguintes entradas: para a aplicação LCS, nós comparamos uma sequência de 500 caracteres com outras sete sequências de mesmo tamanho. Para o *benchmark MapReduce*, nós executamos a análise de frequência de palavras de 11 arquivos, cada um deles contendo 3797 palavras. O *benchmark knapsack* (algoritmo da mochila) foi executado para 10 mochilas com capacidade 200 e considerando 100 itens. O *benchmark* de criptografia 3-DES foi executado tendo como entrada 1000 palavras. O *benchmark* GoL foi executado para 5 tabuleiros distintos de dimensões $100X100$, onde cada tabuleiro realizava 5 iterações.

Por conta da execução paralela dos *workers* da Sucuri, a ordem em que cada operando é recebido pelo escalonador pode variar a cada execução de acordo com as condições de corrida. Portanto, cada cenário em todos os experimentos foram executados 8 vezes com 8 *workers* paralelos para cada execução. Esta quantidade de *workers* foi escolhida por conta do número de VCPUs presentes na máquina virtual da *Google Cloud* disponibilizada para os nossos testes. A média aritmética dos resultados foi usada nos experimentos. É importante destacar que o desvio padrão também foi calculado e pôde ser desprezado.

A granularidade das tarefas foi a menor possível, isto é, para o LCS cada nó do tipo "LCS"era responsável por computar uma única célula da matriz de comparação. De forma semelhante, no algoritmo da mochila, cada nó do tipo *"knapsack"* também computava somente um elemento da matriz de programação dinâmica. O *benchmark MapReduce* foi executado utilizando 1 tupla por nó redutor de primeiro nível. O *benchmark* 3-DES processava um bloco de 8 *bytes*. Por fim, no *benchmark* GoL, cada nó do grafo era responsável por processar 1 elemento do tabuleiro em uma iteração.

O *benchmark unbounded knapsack* não é apresentado nos gráficos das seções

seguintes, pois ele não possui tarefas redundantes. Isso pode ser explicado pelo fato de que, nesta aplicação, os tamanhos dos operandos aumentam, conforme a execução do grafo avança, o que dificulta o reúso de tarefas, pois operandos extensos são mais difíceis de se repetirem na execução. Outro fator prejudicial para a taxa de reúso no algoritmo da mochila é que para cada nó que processa uma partição da matriz de programação dinâmica, ele produz um operando diferente, porque cada partição está contida em uma linha da matriz, e esta linha representa uma única tupla de (valor, peso). Os demais *benchmarks* são apresentados nos gráficos da seguinte forma: a legenda LCS corresponde à aplicação LCS, MR à aplicação *MapReduce*, DES ao *benchmark* 3-DES e GoL à aplicação *Game of Life*.

7.3 Tamanho do grão

O experimento apresentado nesta seção tem por objetivo avaliar a influência do tamanho do grão de tarefas sobre o potencial de redundância de computação. Para isso, começamos as execuções com uma granularidade fina e aumentamos gradualmente o tamanho do grão. Executamos cada cenário 8 vezes, com *caches* infinitas e 8 *workers*. As médias aritméticas dos resultados são apresentados a seguir. Utilizamos a estratégia de detecção LRQ, porém isso em nada influência o experimento, pois o potencial de reúso é uma característica da aplicação e é agnóstica à estratégia de detecção.

A figura 7.6 apresenta os resultados do experimento. O eixo y representa o potencial de reúso das aplicações para cada cenário de grão. No eixo x temos o tamanho do grão da tarefa. Por exemplo, para $x = 10$, temos que um nó do tipo "LCS"do *benchmark* LCS processa um bloco de 10×10 da matriz de comparações. Para o *benchmark MapReduce*, isso significa que cada nó redutor de primeiro nível recebe 10 tuplas de palavras. No caso da aplicação GoL, cada nó do grafo é responsável por processar um bloco de dimensões 10×10 do tabuleiro. Para o *benchmark* 3-DES, este valor de x significa que cada nó processa 10 blocos de 8 *bytes* por estágio do *pipeline*.

A tabela 7.1 apresenta a relação entre os tamanhos distintos de grãos de tarefa e o tamanho em *bytes* dos operandos gerados por estes grãos. Levando em consideração que para alguns *benchmarks* o valor dos operandos aumentam conforme a execução avança, utilizamos o maior valor em *bytes* que um operando poderia ter no início da execução dado o grão da tarefa. O campo "Grão"apresenta o grão da tarefa. O campo "Conf."apresenta os elementos que formam o operando principal da aplicação. Por fim, o campo "Tam."apresenta o tamanho deste campo em *bytes*. Consideramos o valor de 4 *bytes* para inteiros e 1 *byte* para caracteres. Para o *bencmark MapReduce*, o tamanho de uma *String* em *bytes* foi calculado utilizando a maior palavra que este

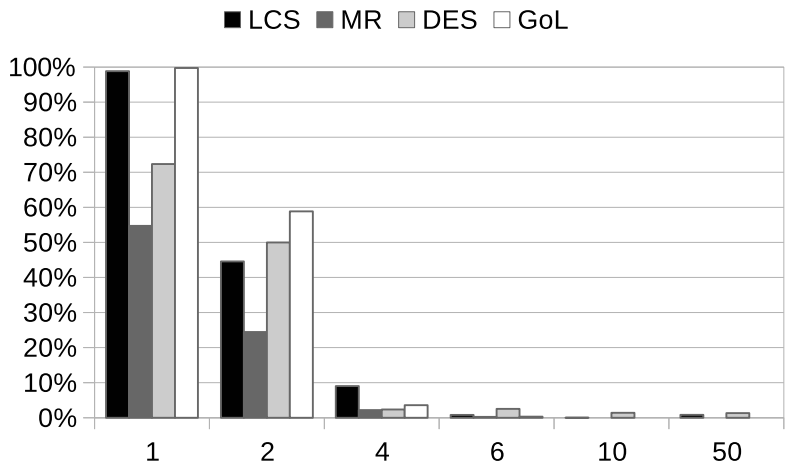


Figura 7.6: Potencial de redundância por tamanho de grão.

processava.

-	LCS		MR		DES		GoL	
Grão	**Conf.**	**Tam.**	**Conf.**	**Tam.**	**Conf.**	**Tam.**	**Conf.**	**Tam.**
1	(1 *Char*, 2 *Ints*)	9 b	(1 *String*, 1 *Int*)	6,25 b	1 X 8 *bytes*	8 b	9 *bits*	1,13 b
2	(2 *Chars*, 4 *Ints*)	18 b	(2 *Strings*, 2 *Ints*)	12,5 b	2 X 8 *bytes*	16 b	16 *bits*	2 b
4	(4 *Chars*, 16 *Ints*)	68 b	(4 *Strings*, 4 *Ints*)	25 b	4 X 8 *bytes*	32 b	36 *bits*	4,5 b
6	(6 *Chars*, 36 *Ints*)	150 b	(6 *Strings*, 6*Ints*)	37,5 b	6 X 8 *bytes*	48 b	64 *bits*	8 b
10	(10 *Chars*, 10^2 *Ints*)	410 b	(10 *Strings*, 10 *Ints*)	62,5 b	10 X 8 *bytes*	80 b	144 *bits*	18 b
50	(50 *Chars*, 50^2 *Ints*)	9,81 Kb	(50 *Strings*, 50 *Ints*)	312,5 b	50 X 8 *bytes*	400 b	2704 *bits*	338 b

Tabela 7.1: Relação de tamanhos em *bytes* dos operandos e os grãos das tarefas.

Note que em todas as aplicações o aumento do grão possui uma influência negativa no potencial de reúso. O potencial de reúso do *benchmark* LCS decai de $98,83\%$ utilizando blocos de 1×1 para $44,57\%$ e $9,03\%$ quando utilizados blocos de blocos de 2×2 e 4×4. Vemos que, para esta aplicação, a partir de um bloco de dimensões 10×10, onde o reúso é de $0,03\%$, não há mais redundância considerável a ser aproveitada. Vemos, porém, um aumento de $0,79\%$ no cenário onde o bloco possui dimensões de 50×50 em relação ao cenário dos blocos de 10×10. Isso pode ser explicado pelo fato do grafo desta aplicação ser sensível ao grão da tarefa. Grafos desse tipo diminuem em tamanho, quando o grão é aumentado e, por consequência, emitem menos nós dinâmicos. Entretanto, por conta de um dos nós *read_seq_file* ler várias vezes o mesmo arquivo, a saber, o da sequência genética que está sendo

alinhada com as demais, ele gera uma certa redundância. Por conta disso, o denominador na equação $1 - N_{Dist}/N_{Din}$ diminui, mas o numerador se mantém constante, gerando um aumento no reúso potencial.

Vemos que há uma queda considerável na aplicação 3-DES, aproximadamente $47,6\%$, no reúso potencial quando aumentamos o tamanho do grão de 2 blocos de 8 *bytes* para 4 blocos de 8 *bytes*. Note que há um aumento no reúso potencial, quando aumentamos o grão de 4 para 6. Isso ocorre, pois, apesar de no cenário onde o grão é igual a 6 haver menos nós dinâmicos instanciados, os nós distintos decaem em uma proporção que faz com que a razão entre nós distintos e dinâmicos se torne menor que esta mesma razão, quando o grão é igual a 4. Por conta disso, sendo a taxa de reúso potencial calculada como $1 - N_{Dist}/N_{Din}$, o reúso máximo torna-se maior. Este comportamento é possível ao 3-DES, pois o grafo desta aplicação possui uma propriedade interessante de não ser sensível ao tamanho da entrada ou do grão da tarefa.

Para o *benchmark* GoL, o reúso potencial decai de $99,74\%$ para $58,85\%$ e $3,58\%$, quando aumentamos o tamanho do bloco de 1×1 para 2×2 e 4×4 respectivamente. A partir do grão de tamanho 6 já não há mais redundância para aplicação.

A aplicação *MapReduce* demonstrou uma queda de $54,73\%$ para $24,45\%$ e $2,19\%$, quando aumentamos o grão da tarefa de 1 tupla por nó redutor de primeiro nível para 2 e 4 tuplas respectivamente.

O comportamento apresentado por todos os *benchmarks* indicam que a técnica de reúso de computação deve ser explorada em arquiteturas *Dataflow* que utilizem grãos finos, ou seja, *hardwares dataflow*. A exploração de modelos *Dataflow* em *software* tende a utilizar grãos mais grossos para compensar os custos de interpretação e aumentar a carga de trabalho das *threads*, porém, vemos que grãos mais grossos não produzem potenciais consideráveis de reúso. Isso desfavorece o uso dessa técnica em estratégias *dataflow* de *software*.

7.4 Tamanho da entrada

Para este experimento avaliamos o comportamento do reúso, conforme aumentamos o tamanho das entradas a serem processadas pelas aplicações *benchmarks*. Como o objetivo era validar o máximo de reúso que a aplicação poderia fornecer sem considerar distribuições (inspeção, NRT ou SRT), utilizamos somente a estratégia LRQ para avaliar este potencial. Vale lembrar que o potencial independe da estratégia utilizada. Começamos com uma entrada do tipo 0 e gradualmente a aumentamos até alcançarmos uma entrada do tipo 5. Entradas do tipo 5 foram as utilizadas para os demais experimentos apresentados neste capítulo.

Na tabela 7.2 são apresentadas as especificações dos arquivos processados pelos

benchmarks. Cada linha nesta tabela corresponde ao formato de arquivo utilizado para cada *benchmark*, porém os *benchmarks* em uma única execução processam mais arquivos. Por exemplo, o *benchmark* LCS processa sempre 8 arquivos por execução: um arquivo possui a sequência biológica que se deseja alinhar e os outros 7 arquivos são as sequências biológicas comparadas a esta. Da mesma forma, os *benchmarks* 3-DES, *MapReduce* e GoL processam 1, 11 e 5 arquivos por execução, respectivamente. A tabela 7.3 apresenta em *bytes* a mesma informação da tabela 7.2.

Tp. de Entrada	LCS	MR	GoL	3-DES
0	10 caracteres	5 palavras	tabuleiro [5X5]	100 palavras
1	100 caracteres	757 palavras	tabuleiro [10X10]	200 palavras
2	200 caracteres	1517 palavras	tabuleiro [20X20]	400 palavras
3	300 caracteres	2277 palavras	tabuleiro [50X50]	600 palavras
4	400 caracteres	3037 palavras	tabuleiro [75X75]	800 palavras
5	500 caracteres	3797 palavras	tabuleiro [100X100]	1000 palavras

Tabela 7.2: Tipos de entradas referentes ao eixo x do gráfico apresentado na figura 7.7.

Tp. de Entrada	LCS	MR	GoL	3-DES
0	80 b	267 b	500 b	492 b
1	800 b	40,6 Kb	1,95 Kb	0,99 Kb
2	1,56 Kb	81,08 Kb	7,81 Kb	1,94 Kb
3	2,34 Kb	121,62 Kb	48,83 Kb	2,94 Kb
4	3,13 Kb	162,08 Kb	109,86 Kb	3,89 Kb
5	3,91 Kb	203,01 Kb	195,31 Kb	4,83 Kb

Tabela 7.3: Tipos de entradas referentes ao eixo x do gráfico apresentado na figura 7.7 em *bytes*.

A figura 7.7 apresenta o potencial de redundância de cada aplicação por tipo de entrada. O eixo x apresenta o tipo de entrada, e o eixo y apresenta o potencial de redundância. O grão de tarefa utilizado para o experimento foi o menor possível para potencializar a redundância encontrada.

Vemos que a tendência das 4 aplicações é aumentar a taxa de reúso máximo, conforme aumentamos o tamanho da entrada. Entretanto, vemos que para as aplicações *MapReduce* e 3-DES temos uma redução no reúso máximo quando aumentamos de 3037 para 3797 palavras, no *benchmark MapReduce*, e 800 para 1000 palavras na aplicação 3-DES. A redução é de aproximadamente $4,98\%$ e $0,67\%$ para as aplicações *MapReduce* e 3-DES, respectivamente. Essa redução é possivelmente explicada pelo fato de que, ao aumentarmos o número de palavras para estas aplicações, incluímos mais palavras distintas, o que reduz o reúso.

É interessante perceber que a técnica de reúso para tarefas de grão fino apresentou grande potencial dentro dos limites da aplicação, ainda que as entradas máximas avaliadas (tipo 5) não tenham sido da ordem de grandeza de *gigabytes*. Vemos

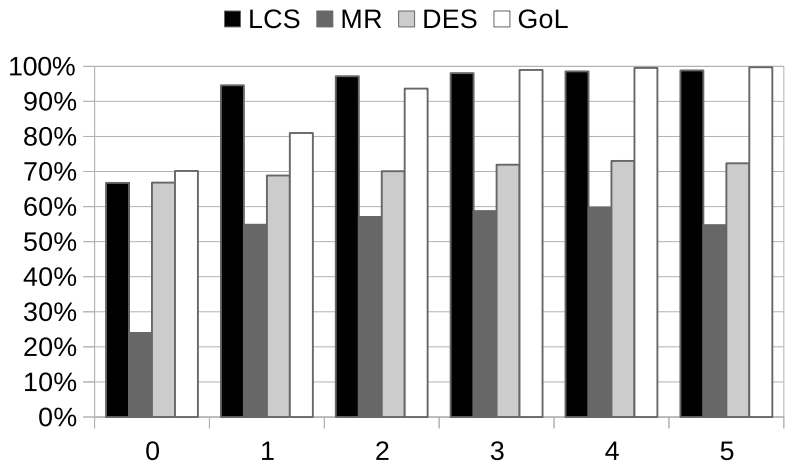


Figura 7.7: Potencial de redundância por tipo de entrada.

também que com estas entradas, as aplicações LCS e GoL, apresentaram uma incrível taxa de redundância de $98,83\%$ e $99,74\%$ respectivamente. A aplicação 3-DES não demonstrou muita variação com o aumento das entradas, mesmo assim, apresentou um reúso acima de 66% para todas os tipos de entradas. Isto é encorajador, pois nos permite inferir que esta aplicação possui uma redundância razoavelmente constante para tamanhos distintos de textos a serem criptografados. Vemos que, em suma, o aumento das entradas parece gerar um comportamento de saturação. Isto sugere que este aumento gera uma razão constante entre os nós distintos e nós dinâmicos do grafo.

7.5 Tipos de *caches*

Esse experimento analisa a influência de diferentes tipos de *cache* nas taxas de reúso em um ambiente de execução *dataflow*. Ele também analisa a distribuição de como as tarefas redundantes foram detectadas. Além disso, sob o contexto de esquemas de *caches* distintas, analisamos as diferentes estratégias de detecção de redundância de tarefas implementadas neste trabalho. Todos os cenários de testes foram executados para *caches* de tamanhos infinitos, com 8 *workers* e grão de tarefas menores possíveis.

Nós executamos o experimento com três tipos de *cache* NRT:

- NRT Local (L) - Cada nó possui sua própria *cache* e só tem visibilidade sobre seus operandos de entrada e saída.
- NRT Compartilhada (S,n) - Cada grupo de nós possui uma *cache* dedicada à eles, e os nós têm visibilidade dos operandos de entrada e saída dos demais nós do grupo.

- NRT Global (G) - Somente existe uma única *cache* centralizada, o que permite que todos os nós conheçam as entradas e saídas de todos os outros nós do grafo.

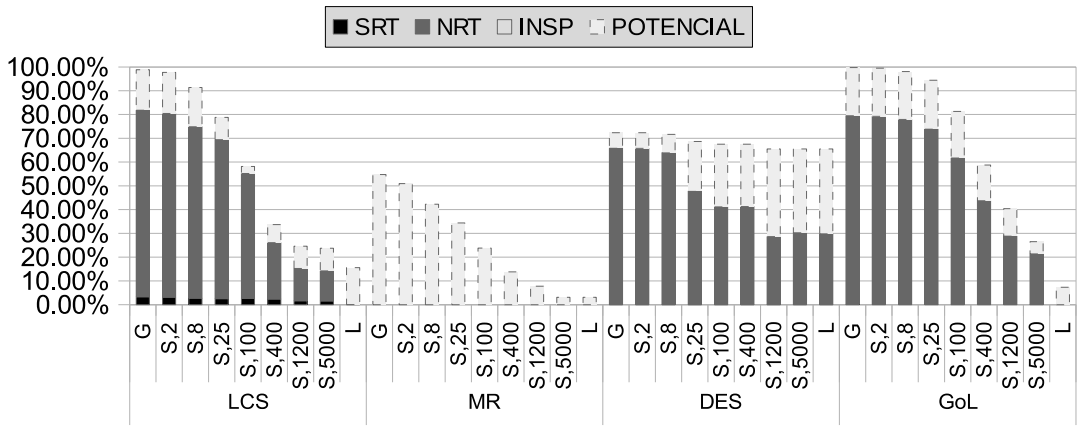
A taxa máxima de reúso, $1 - D(N)/N$, $D(N)$ e N são diretamente influenciadas pelos tipos de *caches*. Por exemplo, considere que esteja sendo utilizado um esquema de *cache* global, e nós temos os respectivos nós e operandos: $(n_1, OP(a, b))$, isto é, nó n_1 com os operandos de entrada (a, b), e $(n_2, OP(a, b))$. Nessa situação, consideramos, do ponto de vista da *cache*, dois nós dinâmicos, mas um nó distinto, e isso nos daria uma taxa máxima de reúso de 50%. Em outro cenário, se estivéssemos usando uma *cache* compartilhada ou uma *cache* local, os nós dinâmicos $(n_1, OP(a, b))$ e $(n_2, OP(a, b))$ seriam considerados dois nós distintos, pois seria impossível para o nó n_1 usar um resultado criado pelo nó n_2 e vice versa, pois eles leem e escrevem em *caches* diferentes.

As figuras 7.8, 7.9 e 7.10 apresentam os gráficos para cada estratégia de detecção de reúso implementada: EM, EM+I, LM, LM+I e LRQ. Para todas estas figuras, o eixo y mostra a taxa de reúso e o eixo x o tipo de *cache* da NRT. A barra referente à legenda "POTENCIAL"representa a taxa máxima de reúso da aplicação. As barras "SRT", "NRT"e "INSP"representam as contribuições para a taxa real de reúso dos mecanismos de *cache* NRT, *cache* SRT e mecanismo de inspeção, respectivamente.

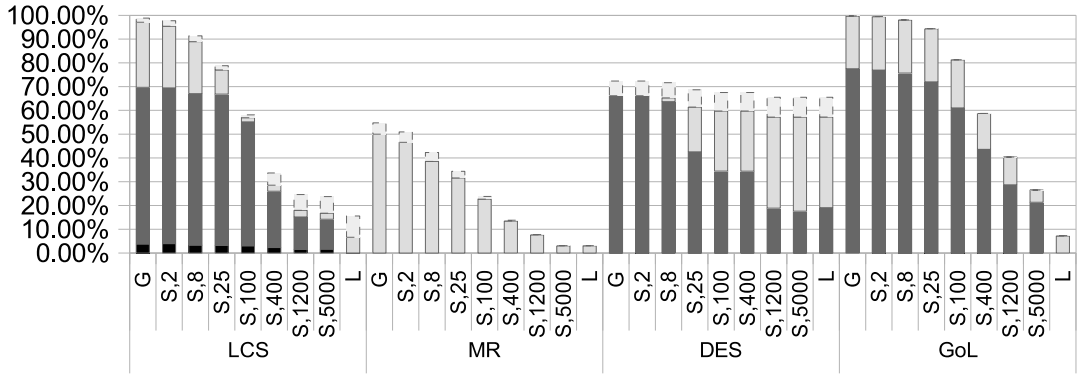
7.5.1 Estratégias EM e EM+I

Vemos que para a estratégia EM, o reúso máximo para o benchmark 3-DES decresce de $72,35\%$, utilizando uma *cache* global, para $65,50\%$, utilizando *caches* locais por nó. O mesmo decréscimo é visto para os demais *benchmarks*: o LCS decai de $98,83\%$ para $15,56\%$. O *MapReduce* decai de $54,73\%$ para $0,03\%$ e o GoL decai de $99,74\%$ para $7,31\%$. Esse decréscimo é esperado por conta da falta de visibilidade sobre os operandos das aplicações que é imposta aos nós do grafo por conta de haver subdivisão da *cache* global. Entretanto, vemos que para o *benchmark* 3-DES, a partir de 1200 *caches*, o reúso máximo se torna constante. Isso ocorre, pois o grafo de aplicação 3-DES, possui uma quantidade de nós menor que 1200, portanto, a partir deste cenário, o comportamento é de uma *cache* local. Ainda sobre a aplicação 3-DES, note que ao avançarmos os cenários "G"para "$S, 2$" e "$S, 100$" para "$S, 400$" não há diminuição no reúso máximo. Isso pode ser explicado pelo fato de que esta aplicação possui muitos tipos de nós distintos, e o reagrupamento deste nós em alguns casos específicos não interfere na visibilidade de operandos, pois eles não precisam consultar a mesma *cache* já que implementam funções distintas. Vemos que para todas as aplicações avaliadas, com exceção do algoritmo da mochila, temos uma quantidade expressiva de redundância. O *benchmark MapReduce* tem uma taxa alta de redundância, porém menor que os demais *benchmarks*. Isso pode

ser explicado pelo fato dos nós redutores estarem sempre agrupando conjunto de tuplas, portanto, o tamanho dos operandos torna-se variável, não são fixos como no LCS, GoL ou DES. Isso tem uma influência prejudicial na taxa de reúso, porque operandos maiores são difíceis de aparecer novamente durante a execução.



(a) Estratégia EM.



(b) Estratégia EM+I.

Figura 7.8: Resultados de reúso para as estratégias EM e EM+I.

Apesar da quantidade expressiva de redundância nas aplicações, ainda sim, poucos subgrafos redundantes são utilizados. Somente é visualizado um reúso de $3,33\%$ por parte da SRT no *benchmark* LCS com uma *cache* global. O reúso baixo pela SRT pode ser explicado por conta deste mecanismo utilizar os Ids dos nós para o reúso do subgrafo, o que limita grandemente o escopo do reúso. Uma alternativa para esta limitação, seria o uso de isomorfismo de subgrafos para detecção de subgrafos redundantes, porém isso geraria um custo de implementação muito grande no *hardware dataflow*. Para os demais, esse tipo de redundância foi desprezível nesta estratégia. Vemos também um decréscimo na utilização de subgrafos redundantes conforme a *cache* vai sendo subdividida em grupos de nós. Isso pode ser explicado pelo fato da falta de visibilidade entre operandos de diferentes nós prejudicarem a reutilização de subgrafos.

As detecções de redundância foram realizadas, na maioria das vezes, pela NRT, ou seja, reutilização nó a nó. Vemos que a NRT, no *benchmark* LCS, contribuiu com $78,44\%$, atingindo um total de $81,77\%$ ($78,44\%+3,33\%$) de redundância detectada. Para os demais *benchmarks* a contribuição da NRT foi de $65,76\%$, $79,32\%$, 0% para os *benchmarks* DES, GoL e *MapReduce* respectivamente. Na aplicação *MapReduce*, nenhuma tarefa redundante foi detectada pela NRT ou SRT em todos os tipos de *cache*. Isso pode ser explicado pelo fato de que esse tipo de grafo, um *fork-join* hierárquico, possui um número grande de tarefas que são assinaladas como prontas ao mesmo tempo. Por exemplo, o nó mapeador, quando executado, liberará para execução todos os redutores de primeiro nível ($red_{1,1}$, $red_{1,2}$, $red_{1,3}$, ..., $red_{1,n}$). Ainda que $red_{1,1}$ e $red_{1,2}$ sejam redundantes, $red_{1,2}$ não estará hábil a utilizar os resultados de $red_{1,1}$, pois ele já estará na fila de prontos. O reúso de computação de $red_{1,1}$ por $red_{1,2}$ somente será possível com o mecanismo de inspeção.

Analisando o cenário de *cache* local, vemos que não houve redundância detectada para os *benchmarks* LCS e GoL apesar de haver reúso a ser explorado. Isso pode ser explicado porque as tarefas instanciadas por cada nó foram executadas praticamente de forma consecutiva, sem um período considerável de tempo entre elas. Portanto, quando as *caches* eram consultadas, os *workers* não haviam enviado ainda os resultados necessários para que as tarefas fossem reutilizadas. Para outros tipos de *caches*, isso não foi um problema, pois os nós possuíam visibilidade sobre os resultados dos outros nós do grafo. Esse comportamento não ocorre para aplicação 3-DES, pois o mesmo possui mais iterações, isto é, o nó *Source* produz mais operandos. Portanto, quando um operando se repete, já ocorreram diversas iterações, permitindo que as *caches* estejam aquecidas o suficiente para que a redundância possa ser detectada.

Na figura 7.8b, analisamos os resultados para a estratégia EM+I. Vemos que o mecanismo de inspeção foi muito efetivo, substancialmente incrementando a taxa real de reúso. Para o *benchmark* LCS, o reúso alcançado foi de $97,1\%$, quase o reúso máximo de $98,83\%$ apresentado pela aplicação. Para aplicação *MapReduce*, vemos que o mecanismo de inspeção contribuiu com aproximadamente 50% do reúso detectado no melhor cenário (G) e com quase toda redundância disponível no cenário local (L). Para o *benchmark* 3-DES, vemos que a inspeção, nos melhores cenários "G"e "$S,2$", não contribuiu expressivamente com o reaproveitamento de tarefas, porém em cenários onde a visibilidade de resultados era baixa, ela proveu grande contribuição, em torno de $18,82\%$ e $38,14\%$ para os cenários "$S,25$" e "L"respectivamente. Isso pode ser explicado pelo fato de que com baixa visibilidade, menos tarefas são ignoradas, e, portanto, mais tarefas são colocadas na fila de prontos, aumentando assim, a oportunidade de reúso por inspeção. A aplicação GoL também apresentou benefícios com a implementação da inspeção. Seu reúso aproveitado de $99,67\%$ foi

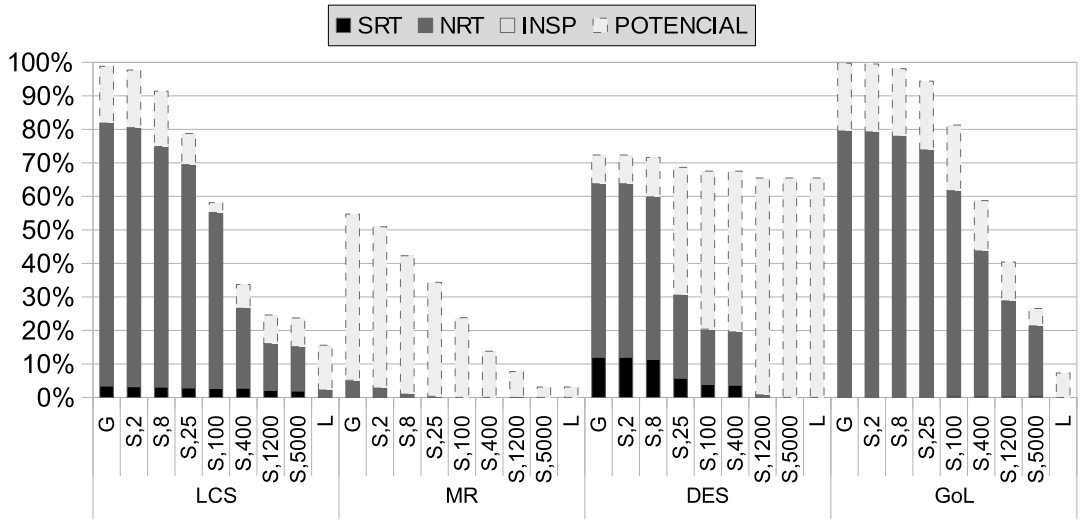
próximo ao máximo disponível pela aplicação de 99, 74%. A inspeção contribuiu com 22, 16% no cenário "G"para o GoL. No cenário "L", ela foi a única responsável por detecção de reúso com 7, 16% de contribuição.

Note que os *benchmarks* apresentados tiveram sua contribuição da NRT decrescida após a habilitação da inspeção. Esse comportamento ocorre, porque a inspeção permite que tarefas sejam removidas da fila de prontas com mais antecedência, antes da *cache* estar bem aquecida. Por consequência, os nós não são reutilizados na NRT e instanciam tarefas. Essas tarefas acabam sendo reutilizadas na inspeção, gerando, desta forma, uma regularização do reúso por parte da própria inspeção.

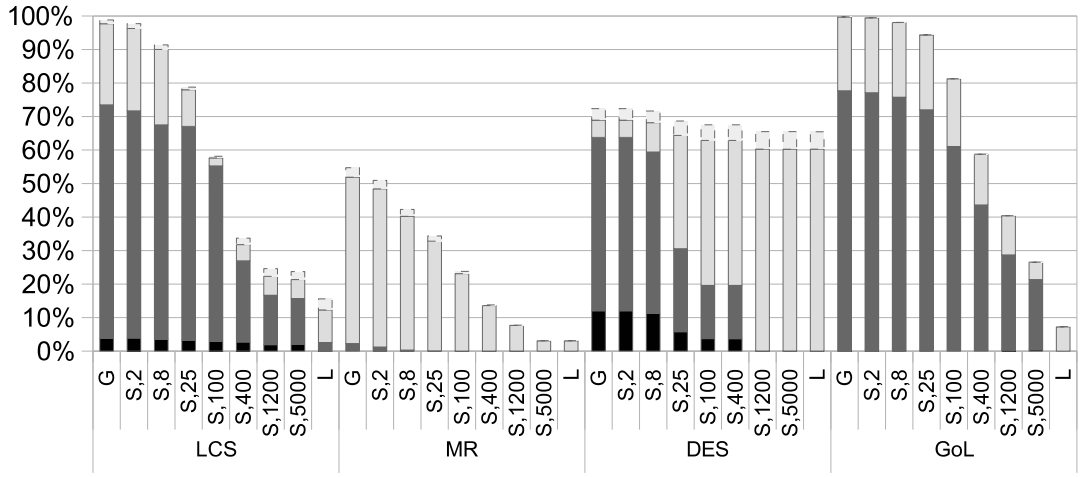
7.5.2 Estratégias LM e LM+I

Para as estratégias LM (figura 7.9a) e LM+I (figura 7.9b) vemos que houve um comportamento de reúso idêntico para os *benchmarks* GoL e LCS, entretanto vemos que o reúso da NRT para o *benchmark* 3-DES foi prejudicado para *caches* mais subdividas. Pode-se notar que, nos cenários onde a *cache* é completamente local, a redundância detectada foi desprezível. Isso pode ser explicado pelo fato de que esses dois tipos de estratégias realizam uma paralisação na execução, isto é, eles recebem operandos, processam-os todos, instancia todas as tarefas disponíveis, e as coloca na fila de prontos. Isto gerou uma ordem de execução das tarefas, onde o reúso em *caches* locais foi prejudicado. Ainda sim, por conta de muitas tarefas serem colocadas na fila de prontas, e permanecerem na fila por mais tempo, foi possível aumentar a redundância detectada pelo mecanismo de inspeção para este *benchmark*, conforme evidenciado na figura 7.9b. Analisando ainda a aplicação 3-DES, vemos que no cenário de *cache* global (G), foi possível aumentar a contribuição da SRT quando combinada com as estratégias LM e LM+I. Vemos que 12, 04% do reúso detectado é por conta de reutilização de subgrafos redundantes. Isso ocorreu, pois esses escalonadores recebem todos os operandos e os distribuem em suas respectivas portas de destino, antes de começarem a detecção de subgrafos redundantes. Desta forma, os contextos de entrada dos subgrafos gerados ficam com menos lacunas, e o reúso dos mesmos torna-se mais fácil.

Acreditamos que os *benchmarks* GoL e LCS não tenham sofrido influência negativa na detecção de redundância por parte destas estratégias, porque os mesmos possuem grafos mais paralelos e possuem poucos tipos distintos de nós. Note que com a estratégias LM e LM+I foi possível detectar em torno de 5, 15% e 2, 26% de redundância respectivamente para a aplicação *MapReduce*, o que não foi possível para as estratégias EM e EM+I. Isto ocorreu, por conta do comportamento de despacho em lotes que esta estratégia produz. No caso do *MapReduce*, ocorre uma explosão de paralelismo e muitas tarefas são colocadas na fila de prontos sem que



(a) Estratégia LM.



(b) Estratégia LM+I.

Figura 7.9: Resultados de reúso para as estratégias LM e LM+I.

haja chance de reúso. Entretanto, para essas estratégias, vemos que conforme as tarefas são despachadas e os seus operandos resultantes retornados ao escalonador, este não tenta detectar redundância imediatamente, antes lê todos os operandos resultantes disponíveis, e, então, começa a detecção de redundância. Isso permite que a *cache* esteja mais aquecida no momento de detecção, provendo, assim, uma contribuição da NRT para o *MapReduce*, ainda que baixa.

7.5.3 Estratégia LRQ

A estratégia LRQ apresentou grande potencial de detecção de redundâncias. Vemos que por conta do momento de detecção ser feito o mais tardio possível, isto é, no momento em que um *worker* ocioso solicita uma tarefa nova, a funcionalidade da

inspeção na fila de prontos pôde ser desabilitada. Note que a maior contribuinte para detecção de redundância foi a NRT. Com esta *cache* obtivemos uma taxa de aproveitamento próxima ao limite da aplicação em todos os cenários de *cache*.

A contribuição da SRT para a detecção de redundância foi prejudicada. Existem dois fatores que explicam esse comportamento: o primeiro consiste no fato de que, por conta da detecção ocorrer no momento em que a tarefa está saindo do escalonador, muitos operandos já foram retirados da porta de entradas de seus nós, o que acarreta em mais lacunas no contexto de entrada dos subgrafos. Isso dificulta o reúso dos subgrafos redundantes. O segundo fator se dá por conta da visibilidade maior sobre os resultados das tarefas que essa estratégia de detecção permite. Esta maior visibilidade gera um reúso contínuo de nós por mais tempo, produzindo subgrafos ainda maiores que são mais difíceis de serem reutilizados.

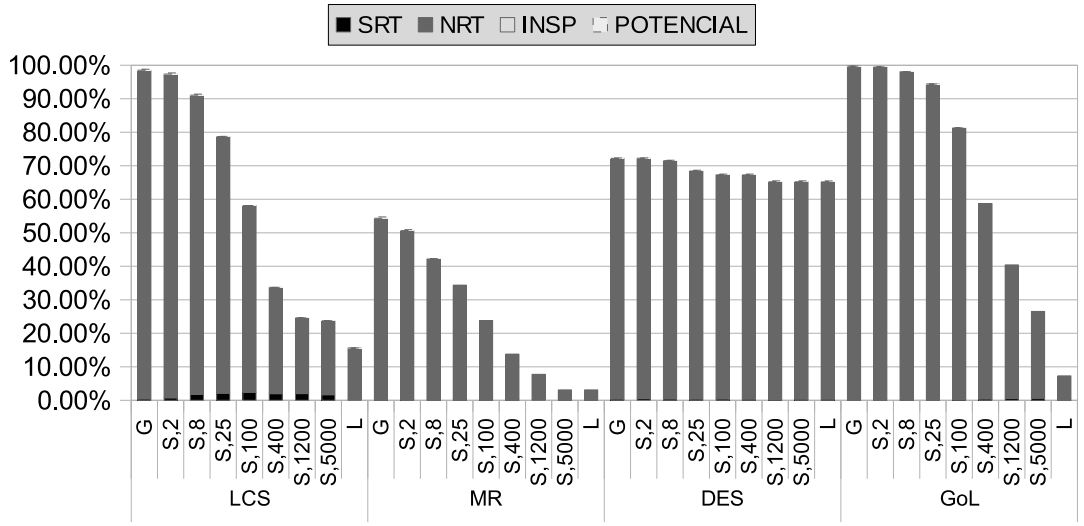


Figura 7.10: Resultados de reúso para a estratégia LRQ.

O único *benchmark* que apresentou contribuição visível da SRT foi o LCS. Vemos que para esta aplicação, conforme a visibilidade dos operandos pelos nós é prejudicada pela subdivisão da *cache*, temos um aumento na contribuição da SRT seguida de um decréscimo desta contribuição. Isso pode ser explicado pelo fato de que a menor visibilidade ajuda a produzir subgrafos redundantes menores, o que facilita o reúso destes. Entretanto, com *caches* mais locais, a visibilidade fica mais restrita, e, menos subgrafos são construídos. Logo, o reúso destes é mais prejudicado. O cenário onde há 100 tabelas de *caches*, permitiu a melhor relação "quantidade de subgrafos"X "visibilidade de operandos", e atingiu um aproveitamento de $2,38\%$.

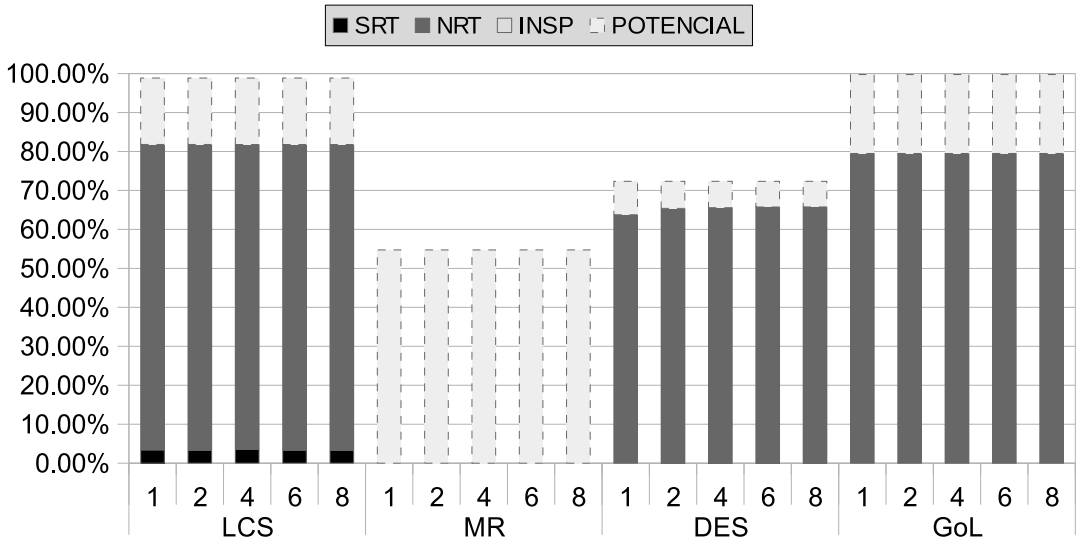
7.6 Paralelismo

O modelo *Dataflow* é um modelo de execução paralelo, por conta disso é interessante avaliar como o paralelismo influencia na detecção de redundância. Neste experimento utilizamos *caches* NRT e SRT globais e infinitas. Começamos a execução com somente 1 *worker* e gradualmente aumentamos o número de *workers* até 8. A granularidade das tarefas foram as mínimas possíveis. As figuras 7.11, 7.12 e 7.13 apresentam os resultados das taxas de reúso para as diferentes estratégias de detecção de redundância. O eixo x apresenta o número de *workers* ativos, e o eixo y apresenta as taxas de reúso. A esquematização das barras dos gráficos é idêntica a apresentada nas figuras da seção 7.5.

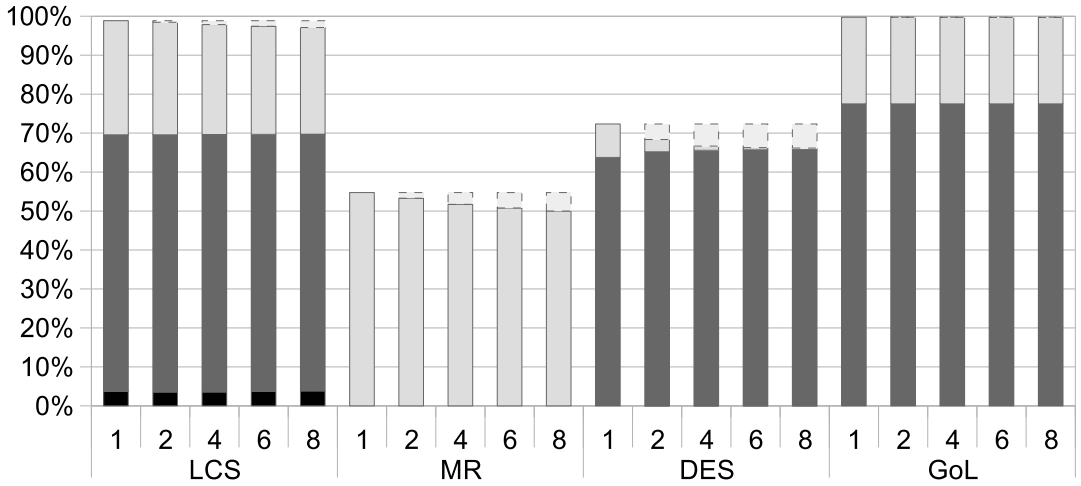
7.6.1 Estratégias EM e EM+I

A figura 7.11a apresenta os resultados do experimento para a estratégia EM. Vemos que o aumento de paralelismo não afetou a contribuição da NRT e SRT no reúso de computação para todos os *benchmarks*. Houve uma pequena melhora na detecção da NRT no *benchmark* 3-DES. Para esta aplicação, quando aumentamos de 1 *worker* para 2 *workers*, vemos um incremento no reúso de $63,76\%$ para $65,4\%$, respectivamente. Isso pode ser explicado pelo fato de que este *benchmark* possui muitos tipos de nós distintos, e quando se executa com um *worker*, a ordem de execução para este escalonador neste *benchmark* foi semelhante a uma ordem sequencial. Isto é, eram processados todos os elementos para o nó 1, após isso eram processados todos os elementos para o nó 2, e assim por diante até o último nó. Vemos que por conta desta ordem algumas poucas tarefas não puderam ser puladas, pois, possivelmente, as tarefas que eram redundantes já estavam na fila de prontos, quando seus resultados foram conhecidos. Com mais *workers*, é possível aquecer as *caches* mais rapidamente, gerando, em alguns cenários, o incremento do reúso.

Conforme demonstrado na figura 7.11b, quando ativamos a inspeção, vemos que o aumento de paralelismo degrada a contribuição desta técnica para a detecção de reúso. Isso pode ser explicado pelo fato de que a inspeção se beneficia de que tarefas passem mais tempo na fila de prontos. Com o aumento de *workers*, o consumo dessa fila aumenta, e as tarefas são removidas desta fila antes que uma mensagem de operando possa chegar com o resultado para essas tarefas. Vemos que, para o *benchmark* 3-DES, a partir de 4 *workers*, a inspeção se torna baixa. Acreditamos que isso ocorra, porque a *cache* global infinita NRT consegue resolver maior parte do reúso, o que não permite que muitas tarefas fiquem na fila de prontos. Combinando isso ao fato do maior consumo pelo aumento de *workers*, vemos uma contribuição menor para esta técnica a partir de 4 *workers*.



(a) Estratégia EM.



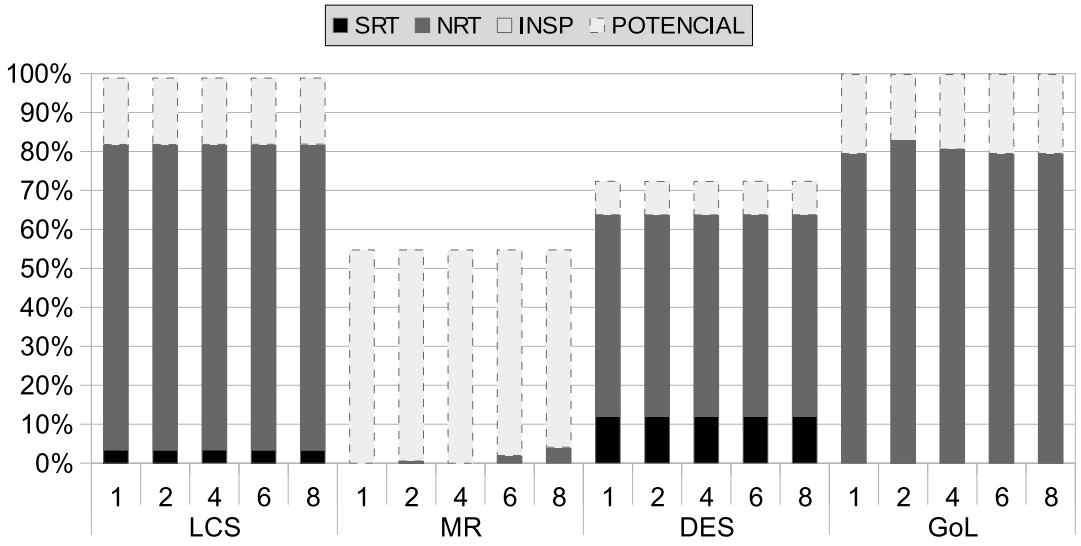
(b) Estratégia EM+I.

Figura 7.11: Resultados de reúso sob a influência de paralelismo para as estratégias EM e EM+I.

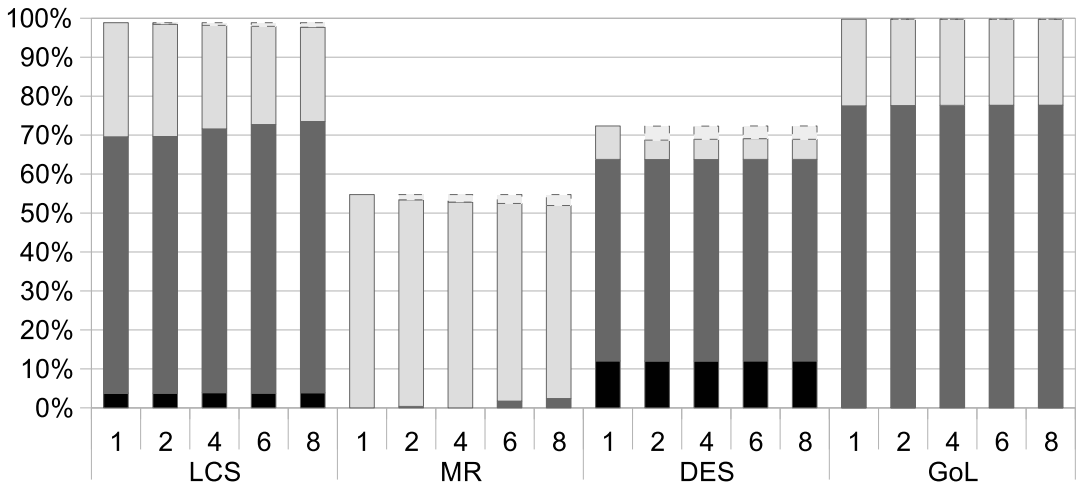
7.6.2 Estratégias LM e LM+I

Para as estratégias LM, vemos, na figura 7.12a, que os *benchmarks* LCS e 3-DES permaneceram inalterados pelo aumento de *workers*. O fato desta estratégia processar um conjunto de mensagens, a saber, todas que estão aguardando processamento na fila de espera, permite que mais resultados sejam conhecidos antes de começar a detecção de redundância. Para esses dois *benchmarks*, esse comportamento permitiu que o aumento de *workers* não degradasse as taxas de contribuição da SRT e NRT.

Avaliamos que existe algumas configurações de números de *workers* que podem ser melhores para a detecção de reúso. Vemos que na estratégia LM, o *benchmark MapReduce* teve contribuições de $0,55\%$, $1,97\%$ e $4,10\%$ pela NRT no cenário de



(a) Estratégia LM.



(b) Estratégia LM+I.

Figura 7.12: Resultados de reúso sob a influência de paralelismo para as estratégias LM e LM+I.

2, 6 e 8 *workers*. Note que os cenários de 1 e 4 *workers* não houve detecção de redundância. Acreditamos que, com 2 *workers*, foi possível detectar redundância, ainda que pequena, pois enquanto 1 *worker* processava o nó *Source*, o outro processava as tarefas que o nó *Source* liberava, o que gerava uma alimentação da *cache* melhor. Entretanto, quando avançamos para o cenário com 4 *workers*, as tarefas que haviam sido detectadas no cenário anterior foram executadas em paralelo, eliminando assim a pouca redundância que havia sido encontrada. Porém, quando aumentamos os *workers* para 6 e 8, foi possível aquecer a *cache* mais rapidamente, gerando assim um melhor aproveitamento da redundância. Vemos que para o *benchmark* GoL, houve uma melhora na contribuição da NRT no cenário de 2 *workers*. Neste cenário, a contribuição foi de $82,8\%$ aproximadamente, e para os demais

cenários a contribuição da NRT se manteve semelhante, entre 79, 48% e 80, 62%.

Quando ativamos a inspeção (estratégia LM+I) e aumentamos o número de *workers*, vemos na figura 7.12b, um comportamento de degradação da redundância detectada por esta técnica, em torno de 5, 08%, 3, 44% e 5, 18% aproximadamente, nos *benchmarks* LCS, 3-DES e *MapReduce* respectivamente, quando comparados os cenários de 1 *worker* com 8 *workers*. Assim como na estratégia EM+I, este é um comportamento esperado, pois o maior consumo na fila de prontos remove a chance de uma tarefa ser utilizada enquanto aguarda nesta fila. É interessante notar que para o *benchmark* 3-DES, depois do aumento de 1 *worker* para 2 *workers*, o decaimento da contribuição da inspeção se torna constante, ao contrário do que foi visto na estratégia EM+I. Isso pode ser explicado pela forma da estratégia LM+I propiciar que mais tarefas fiquem na fila de prontos, pois ela processa os operandos resultantes provenientes dos *workers* em lotes, gerando, consequentemente tarefas em lotes também.

7.6.3 Estratégia LRQ

Nesta estratégia, conforme mostra A figura 7.13, vemos que o aumento de paralelismo gerou baixa influência na detecção de reúso. O decaimento da contribuição da NRT devido ao aumento de *workers* foi de aproximadamente 0, 04%, 0, 15%, 0, 42% e 0, 35% para os *benchmarks* GoL, DES, *MapReduce* e LCS, respectivamente, quando comparamos os cenários de 1 *worker* com 8 *workers*. Esta baixa influência pode ser explicada por conta da detecção tardia de redundância ocorrer no último momento da tarefa no escalonador central. Quando essa detecção começa, muitos operandos já foram recebidos, e a *cache* já está bem aquecida, permitindo um maior reúso. O fato de haver um pequeno decréscimo da contribuição da NRT, se dá pelo fato de que tarefas com mesmos operandos de entrada foram executadas em paralelo, pois se fossem executadas de forma serial, pelo menos uma delas seria reutilizada.

7.7 Limitação de *cache*

Este experimento avalia o comportamento do reúso para as estratégias de detecção implementadas em um cenário onde o tamanho da *cache* NRT e SRT é limitado. Começamos com uma *cache* de tamanho 100, e aumentamos a quantidade de suas entradas até chegarmos em um cenário de *cache* infinita. A política de substituição utilizada foi a LRU *Least Recently Used*. Cada cenário de *cache* foi executado 8 vezes. Os *benchmarks* foram utilizados com o menor grão de tarefa possível para cada cenário. O número de *workers* utilizados foram 8. As figuras 7.14, 7.15 e 7.16 apresentam os resultados deste experimentos para as diferentes estratégias. O eixo

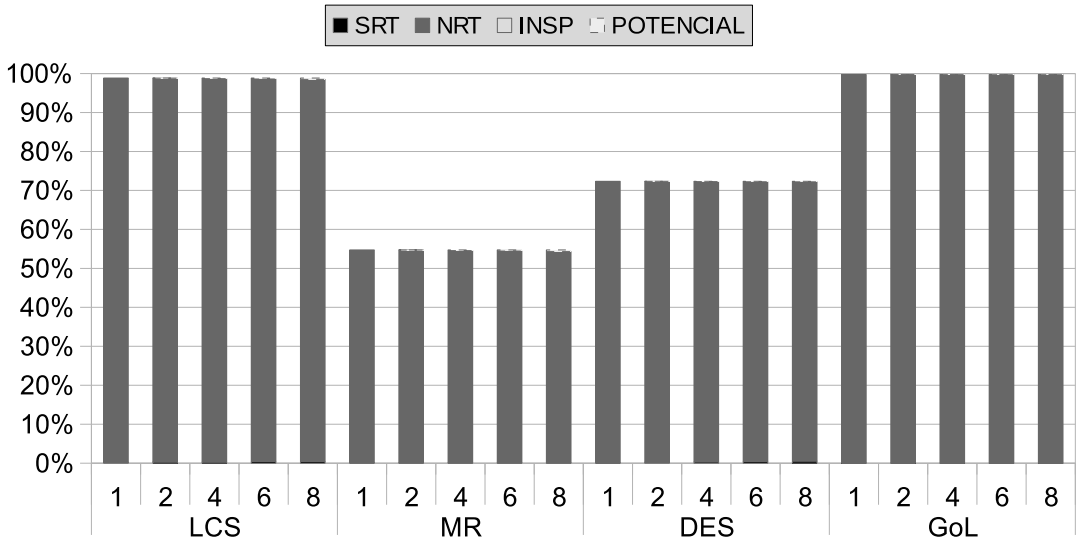


Figura 7.13: Resultados de reúso sob a influência de paralelismo para a estratégia LRQ.

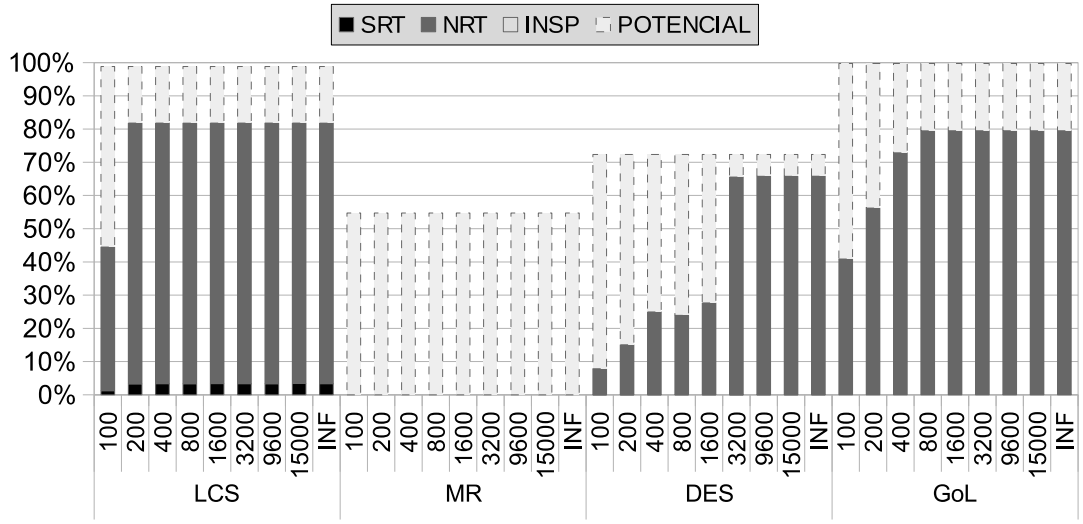
x mostra o tamanho das *caches* utilizadas e o eixo y apresenta as taxas de reúso (máxima, real e distribuição). As esquematições das barras dos gráficos seguem o mesmo modelo apresentado nas seções 7.5 e 7.6.

7.7.1 Estratégias EM e EM+I

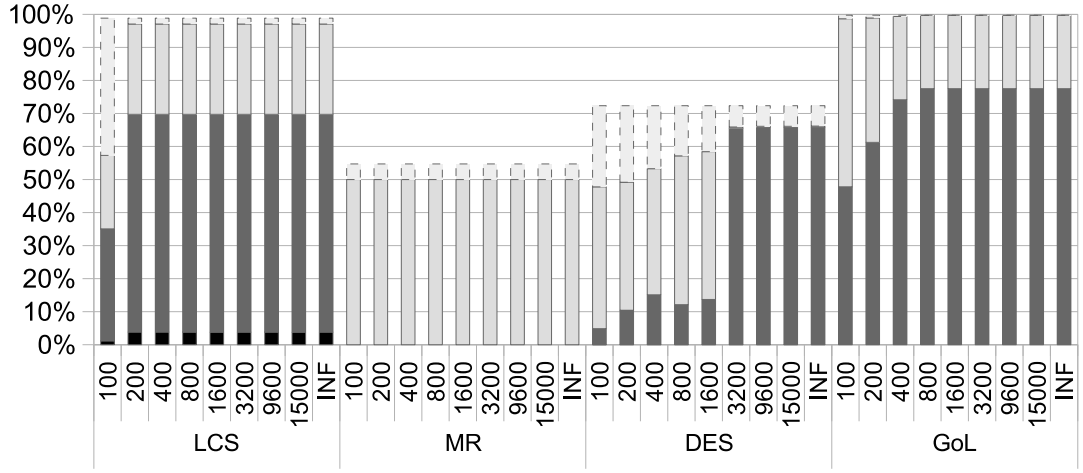
A figura 7.14 apresenta os resultados deste experimento para as estratégias EM e EM+I. Pode-se notar que, para estas estratégias, com uma *cache* de 200 entradas conseguimos atingir para o *benchmark* LCS um comportamento semelhante a de uma *cache* infinita. Isso pode ser explicado, pois o tempo necessário para que um operando seja reutilizado nesta aplicação é muito curto, e depois de um determinado período de execução, ele não é mais utilizado. Isso garante que uma *cache* razoavelmente pequena, onde a detecção de reúso é imediata, atenda bem a localidade de valores da aplicação. Analisando a SRT para a aplicação LCS, vemos que a contribuição desta para redundância detectada também a mesma de uma *cache* infinita, aproximadamente $3,33\%$.

Para o *benchmark* 3-DES, vemos que uma *cache* NRT de 3.200 entradas tem os mesmos benefícios de uma *cache* infinita. Note que para este *benchmark*, na figura 7.14a, vemos que a *cache* com 400, 800 e 1.600 entradas possuem comportamentos semelhantes. Quando a inspeção é habilitada, notamos que o aumento da *cache* para esses três cenários aumenta a contribuição da redundância detectada pelo mecanismo de inspeção, conforme apresentado na figura 7.14b. Acreditamos que, mesmo que os comportamentos destes três cenários sejam semelhantes, as *caches* maiores permitem que mais tarefas sejam liberadas para execução simultaneamente, o que gera mais tarefas nas filas de prontos, e, consequentemente, potencializa a detecção por parte

da técnica de inspeção.



(a) Estratégia EM.



(b) Estratégia EM+I.

Figura 7.14: Resultados de reúso com limitação do tamanho da *cache* para as estratégias EM e EM+I.

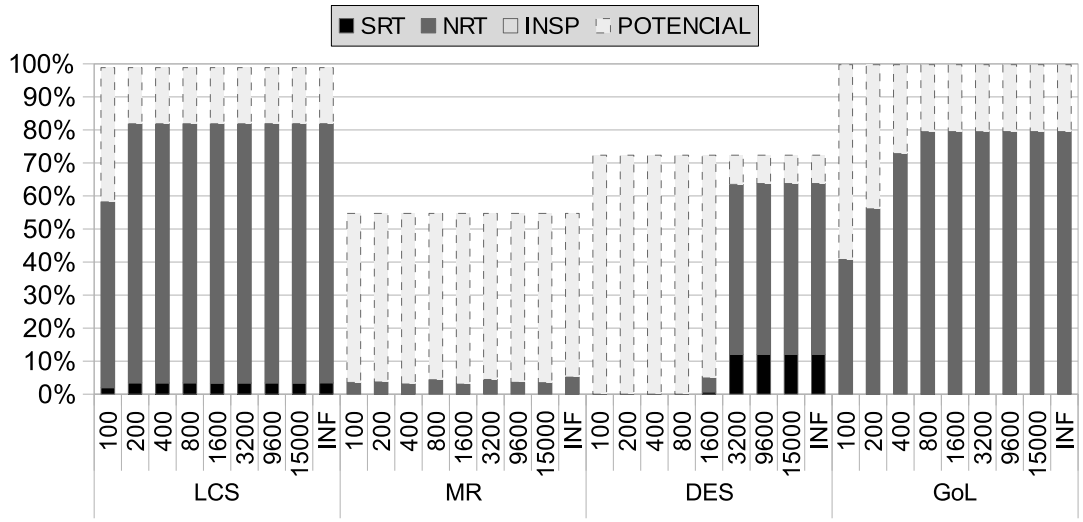
O *benchmark* GoL apresenta um comportamento regular de incremento no reúso detectado tanto por inspeção, quanto por acertos na NRT, conforme o tamanho das *caches* é incrementado. Note que com uma *cache* de 800 entradas apenas, temos reúso similar a uma *cache* infinita para ambos escalonadores. Da mesma forma o efeito da inspeção é o mesmo a partir de uma *cache* de 800 entradas.

A aplicação *MapReduce* não demonstrou uso de *caches* para a estratégias EM, pois, por conta da explosão de paralelismo desta aplicação, grande parte das tarefas são colocadas na fila de prontos simultaneamente, retirando, assim, a oportunidade para o reúso. Vemos a redundância sendo detectada neste tipo de escalonador somente quando utilizamos inspeção (estratégia EM+I). Neste caso, o tamanho das

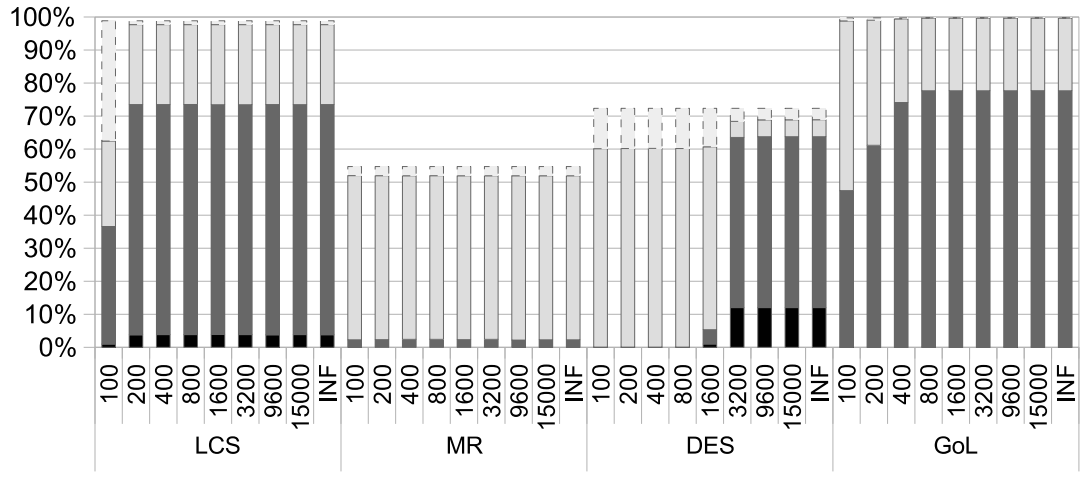
caches NRT e SRT não influenciam a contribuição da inspeção.

7.7.2 Estratégias LM e LM+I

As estratégias LM e LM+I, figura 7.15a e 7.15b respectivamente, não alteram o comportamento da detecção de reúso dos *benchmarks* GoL e LCS, se comparados às estratégias EM e EM+I.



(a) Estratégia LM.



(b) Estratégia LM+I.

Figura 7.15: Resultados de reúso com limitação do tamanho da *cache* para as estratégias LM e LM+I.

Para as estratégias LM e LM+I, o *benchmark* 3-DES apresentou uma taxa de reúso real por parte da NRT de $4,23\%$ e $4,35\%$, respectivamente, com uma *cache* de 1600 entradas. Esta foi uma contribuição muito inferior à avaliada nas estratégias EM e EM+I. Para este mesmo cenário de *cache* com 1600 entradas, mas com as

estratégias EM e EM+I, a contribuição da NRT foi de $27,62\%$ e $13,64\%$, respectivamente. Para tamanhos menores do que 1600, as estratégias LM e LM+I não alcançaram uma taxa de reúso acima de 1%, enquanto para as estratégias EM e EM+I, a NRT já possuía uma contribuição de $7,83\%$ e $4,84\%$ para o cenário com apenas 100 entradas. Esta desvantagem das estratégias LM's em relação às EM's é explicada pelo fato de que as estratégias LM's processam muitos operandos antes de começarem a detecção de redundância. Portanto, quando a *cache* é verificada para descobrir se o resultado de uma determinada tarefa já é conhecido, os operandos daquela tarefa já não estão mais presentes na *cache*, pois muitos novos operandos foram inseridos e essas entradas foram descartadas. Vemos que a partir de uma *cache* de 3.200 entradas esse comportamento é eliminado e temos resultados semelhantes a uma *cache* infinita. Note também que a contribuição da técnica de inspeção na estratégia LM+I foi maior do que a contribuição da mesma técnica na estratégia EM+I para o 3-DES. Acreditamos que, pelo fato da estratégia de detecção LM+I proporcionar que mais tarefas sejam disponibilizadas na fila de prontos, a inspeção para este escalonador foi potencializada.

O *benchmark MapReduce* apresentou comportamento similar, com poucas variações, para os cenários de *caches*. Por conta deste *benchmark* ter pouco reúso detectado por estas estratégias, ainda que com *cache* infinita, é natural que uma *cache* pequena seja o suficiente para capturar esta pequena amostra de redundância.

7.7.3 Estratégia LRQ

A figura 7.16 apresenta a influência do tamanho da *cache* e a política LRU de substituição para a estratégia LRQ.

O *benchmark* LCS, para a estratégia LRQ, somente atingiu um comportamento semelhante a uma *cache* infinita a partir de 800 entradas. As demais estratégias, apresentaram um comportamento de *cache* infinita a partir de 200 entradas apenas. Isso ocorreu, pois a estratégia LRQ realiza detecção tardia, e, por conta disso, perde a oportunidade de reutilizar tarefas que possuíam seus resultados escritos na *cache* muito cedo. Vemos que a partir de 800 entradas, temos um reúso de $98,17\%$ para a LRQ. Este reúso real detectado é superior à redundância detectada pelas outras estratégias com *caches* infinitas. Vemos o mesmo comportamento para os *benchmarks* 3-DES e GoL. No caso do 3-DES, com uma *cache* NRT de 3.200 entradas conseguimos os mesmos benefícios de uma *cache* infinita, alcançando uma detecção de redundância de $71,97\%$, muito superior à taxa alcançada pelas estratégias LM's e EM's com *caches* NRT infinitas· Da mesma forma, para o GoL, o reúso alcançado com uma NRT de apenas 800 entradas, aproximadamente $99,70\%$, foi superior ao reúso detectado pelas demais estratégias nos cenários ilimitados sem inspeção, e

semelhante às demais estratégias com inspeção.

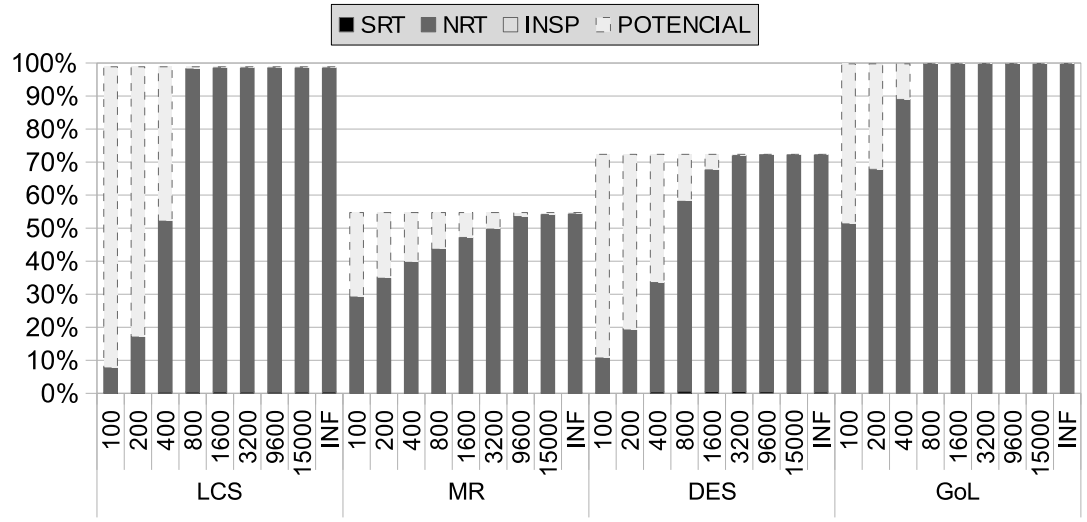


Figura 7.16: Resultados de reúso com limitação do tamanho da *cache* para a estratégia LRQ.

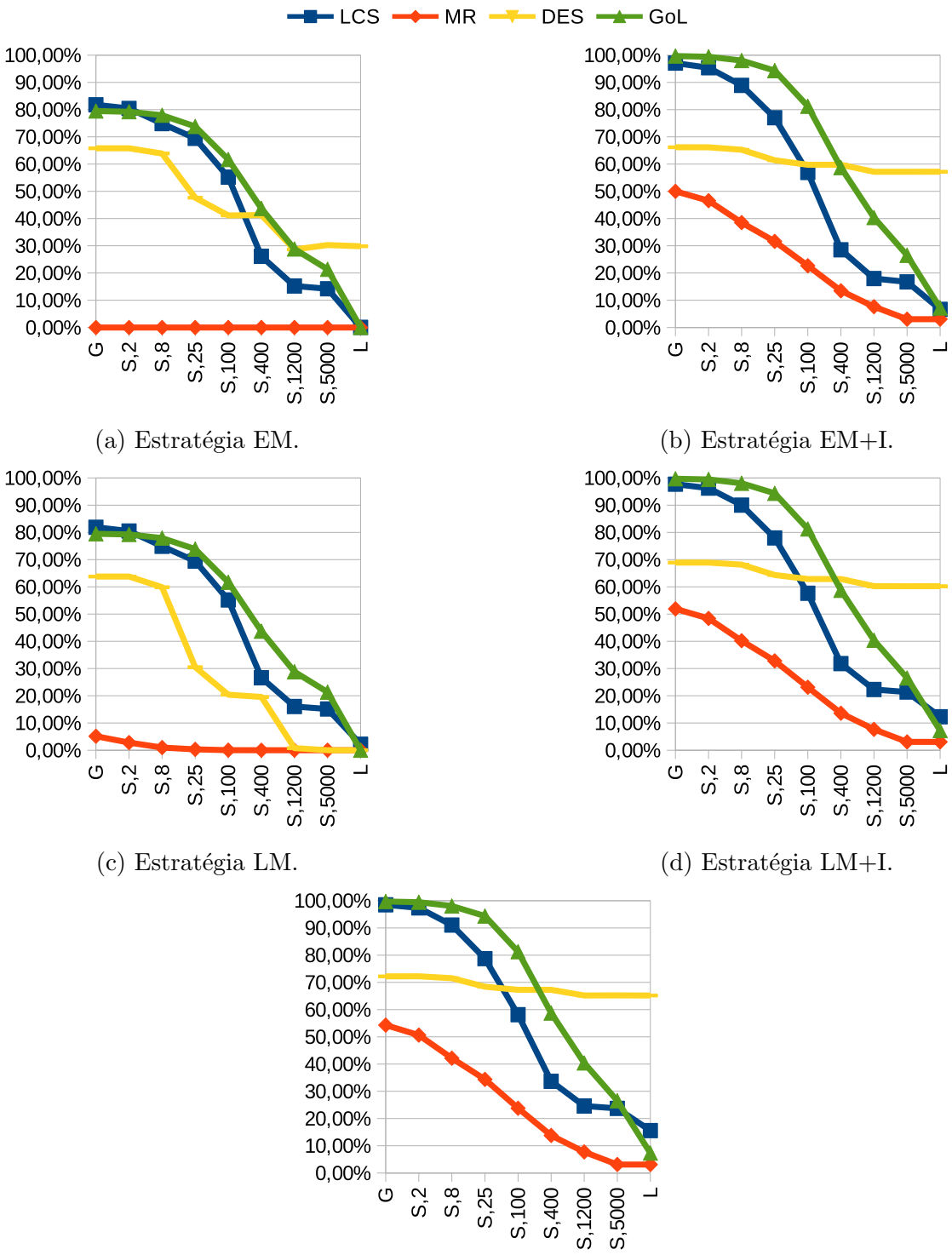
Na aplicação *MapReduce*, somente com a *cache* de tamanho máximo de 15.000 entradas, alcançamos uma taxa de reúso real de $54,13\%$, apenas $0,6\%$ abaixo do reúso máximo permitido pela entrada da aplicação. Isso ocorreu pelo fato de que a explosão de paralelismo que essa aplicação apresenta gerou muitos operandos distintos em curto espaço de tempo, sendo necessária uma *cache* maior para armazenar todos esses resultados e permitir o reúso destes por tarefas posteriores.

O fato de que para os 4 *benchmarks*, *caches* relativamente pequenas proporcionaram uma grande quantidade de reúso de nós, demonstra que essa técnica possui aspectos promissores para implementações em *hardware*.

7.8 Taxa de comunicação

Neste experimento avaliamos a relação do reúso de computação com a taxa de comunicação entre *workers* e o escalonador centra da Sucuri. Esse experimento foi executado 8 vezes e com 8 *workers*. A granularidade das tarefas para este experimento foi a menor possível. Utilizamos a média aritmética dos resultados. O desvio padrão foi desprezível. A figura 7.17 apresenta os resultados de redução na taxa de mensagens enviadas e recebidas para todas as estratégias de detecção de reúso. O eixo y apresenta o quanto, em porcentagem, foi reduzida a troca de mensagens. O eixo x apresenta as configurações de diferentes tipos de *caches*, de forma semelhante aos gráficos apresentados na sessão 7.5.

Note que para todos as estratégias, a redução de mensagens acompanhou a taxa real de reúso alcançada. As maiores reduções que alcançamos foram estas: O *bench-*



(a) Estratégia EM.

(b) Estratégia EM+I.

(c) Estratégia LM.

(d) Estratégia LM+I.

(e) Estratégia LRQ.

Figura 7.17: Redução de trocas de mensagens por estratégias.

mark GoL, obteve uma redução de 99, 7% na taxa de comunicação para a estratégia LM+I. A aplicação LCS, quando combinada com a estratégia LRQ, obteve uma redução de 98, 49% na taxa de comunicação. Com o *benchmark MapReduce*, obtivemos uma redução de 54, 31% utilizando a estratégia LRQ. Para a aplicação 3-DES,

obtemos uma redução de $72,22\%$ utilizando a estratégia LRQ. Estas reduções na taxa de comunicação que acompanham a taxa de reúso é um comportamento esperado, pois a Sucuri só possui troca de mensagens para enviar tarefas aos *workers* e receber operandos dos mesmos. Por conta disso, quando se reutiliza uma tarefa, a necessidade de troca de mensagens para determinada tarefa é eliminada, relacionando assim, a taxa de reúso com a taxa de redução de comunicação.

Esta relação entre redução de mensagens e reúso é interessante, pois como vimos na seção 7.3, o reúso é melhor aproveitado quando estamos trabalhando com tarefas de grão fino. Uma desvantagem de se explorar paralelismo com tarefas de grão fino é justamente um aumento na comunicação entre os elementos processadores. Entretanto, vemos que o que potencializa o reúso de computação pode trabalhar em sinergia com este tipo de exploração de paralelismo para reduzir gastos com comunicação.

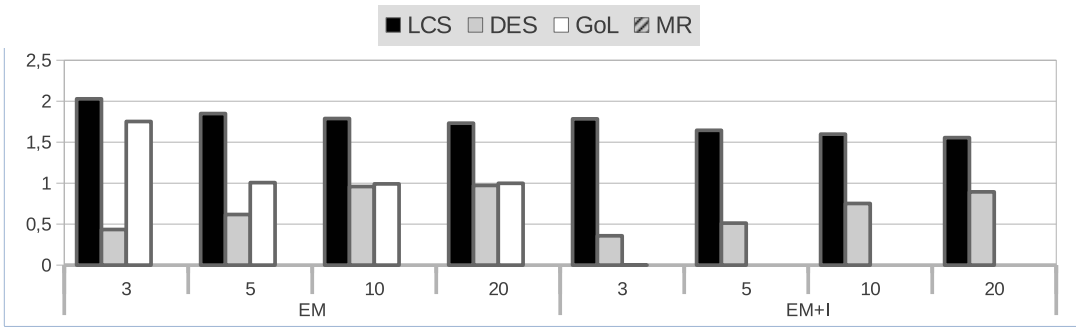
7.9 Tamanhos de subgrafos redundantes

Nos demais experimentos, a construção de subgrafos candidatos para o reúso era realizada de forma ilimitada, ou seja, enquanto nós fossem detectados como redundantes, estes eram adicionados aos subgrafos. Isto permitia a criação de subgrafos maiores que possuíam maiores contextos de entrada, dificultando o reúso dos mesmos. Nesta seção apresentamos experimentos que avaliam o potencial de reúso de subgrafos redundantes quando o tamanho destes são limitados. Começamos com um limite de 3 nós por subgrafo, ou seja, será possível existir subgrafos de 2 e 3 nós, e aumentamos gradualmente até 20 nós por subgrafos, neste cenário será possível que existam subgrafos com 2 a 20 nós.

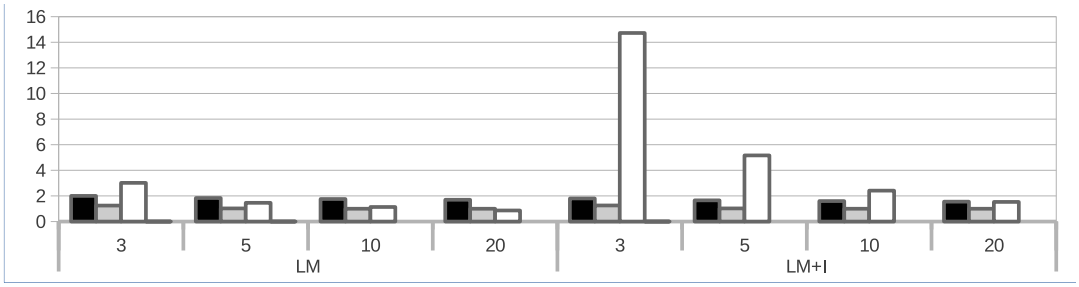
A figura 7.18 mostra os resultados desses experimentos para as diferentes estratégias de reúso. O eixo x apresenta o número limite de nós por subgrafos, juntamente com a estratégia utilizada. O eixo y apresenta a razão entre o uso da SRT, quando há limitações de tamanhos aos subgrafos, em relação ao uso da SRT, quando não há limitações. Por exemplo, um valor 2 para y e 3 para x, significa que quando limitamos o tamanho máximo dos subgrafos redundantes à 3 nós, o uso da SRT dobra em relação ao cenário onde não há limites de tamanho para os subgrafos. Os resultados são apresentados desta forma, pois as diferentes aplicações possuem diferentes padrões de uso da SRT, o que torna difícil a visualização de todos os resultados em um mesmo tipo de gráfico.

Neste experimento utilizamos um grão de tarefa mais fino possível, 8 *workers* e *caches* NRT e SRT infinitas. Os cenários dos experimentos foram executados 8 vezes cada, e as médias aritméticas dos resultados são apresentadas. O desvio padrão pôde ser desconsiderado.

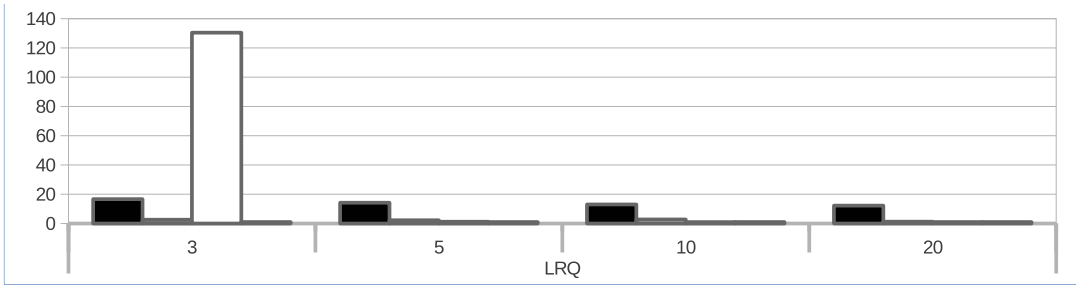
Note que para a aplicação LCS, em todas as estratégias, conforme aumentamos o tamanho limite do subgrafo, o reúso da SRT, em relação ao uso desta mesma *cache* em um cenário sem limitações, decai. Isto pode ser explicado pelo fato de que menores subgrafos podem implicar em contextos de entrada menores, o que aumenta a reutilização de subgrafos redundantes para esta aplicação.



(a) Estratégias EM e EM+I.



(b) Estratégias LM e LM+I.



(c) Estratégia LRQ.

Figura 7.18: Resultados do experimento para estratégias implementadas.

A aplicação 3-DES demonstrou um comportamento interessante e contrário ao esperado para as estratégia EM e EM+I. Vemos que neste cenário, quando aumentamos o limite de nós por subgrafos redundantes, o uso da SRT aumenta. Isso é possível, pois os subgrafos redundantes formados nesta estratégia são extensos, mas com o contexto de entrada pequeno. Quando se limita o tamanho desses subgrafos, estes são quebrados em mais subgrafos de tamanho menores. Logo, para se ter o mesmo aproveitamento da SRT, se faz necessário que mais contextos de entradas sejam preenchidos simultaneamente, o que dificulta o reúso. Para estas estratégias com

um subgrafo de tamanho 10 e 20 temos um aproveitamento similar de um cenário sem limitações de tamanhos para subgrafos. Para as estratégias LM e LM+I (figura 7.18b), temos um reúso melhor para subgrafos menores, aproximadamente $1,24$ e $1,25$ vezes, respectivamente, o que pode ser explicado por conta desta estratégia realizar uma certa paralisação na execução e processar mais nós em lotes, permitindo que mais contextos de entradas sejam preenchidos e mais subgrafos possam ser reutilizados. Para a estratégia LRQ (figura 7.18c), vemos que o aproveitamento da SRT foi maior para subgrafos de tamanho 10, aproximadamente $2,75$, por motivos semelhantes das estratégias EM e EM+I. A subdivisão de subgrafos em menores tamanhos, gerou mais subgrafos que não puderam ser reutilizados, pois tornou-se mais difícil o preenchimento dos diversos contextos de entrada. Para subgrafos de tamanho 20, a estratégia LRQ não demonstrou aumento de uso da SRT. Acreditamos que esse cenário gerou subgrafos com contextos de entrada mais difíceis de serem preenchidos simultaneamente.

O *benchmark* GoL apresentou um aproveitamento de $1,75$, $14,72$ e $130,4$ das SRTs em relação ao cenário sem limites para as estratégias EM, LM+I e LRQ. Subgrafos redundantes maiores para o GoL são difíceis de serem reutilizados, pois produzem contextos de entrada muito grandes, podendo estes serem calculados como $9 + 8 \times (n - 1)$, onde n é a quantidade de nós no subgrafo redundante. Note que para a estratégia EM+I e tamanho de subgrafo limitado a 3 nós, a inspeção reduziu a 0 o uso da SRT, enquanto na estratégia LM+I, a SRT foi utilizada $14,72$ vezes mais, quando comparada ao uso da SRT com mesma estratégia, mas sem limitar o tamanho dos subgrafos. Para a estratégia LM+I houve esse aumento, pois mais operandos de entrada eram providos aos contextos de entrada dos subgrafos graças ao processamento de mensagens de operandos em lotes e os nós que eram retirados da fila de prontos. No caso da estratégia EM+I, em nenhum caso houve uso da SRT, pois os nós que poderiam ser reutilizados como partes de subgrafos, foram reutilizados antes pela inspeção. A estratégia LRQ apresenta um aumento no uso da SRT em aproximadamente $130,4$ vezes, quando limitamos o tamanho do subgrafo a 3 nós. Acreditamos que isso ocorra por conta da alta visibilidade desta estratégia sobre os resultados das tarefas combinado com o fato de que, nesta aplicação, menores subgrafos possuirão menores contextos de entrada.

A aplicação *MapReduce* não apresentou melhora significativa da SRT para em nenhum dos casos, pois os seus poucos subgrafos reutilizados já possuíam tamanhos menores ou iguais a 3.

Vemos que em todos as estratégias existem dois grandes dificultadores, estes são: a criação de um subgrafo extenso com um contexto de entrada curto, e prover todos os operandos deste contexto ao mesmo tempo para que o subgrafo seja reutilizado. Em segundo lugar, somente permitir o reúso de subgrafos compostos pelos mesmos

nós. Possivelmente, uma abordagem especulativa poderia aumentar o reúso desses subgrafos por especular a presença de operandos no contexto de entrada. Entretanto, os custos de tal mecanismo devem ser avaliados, pois existirá a necessidade de uma etapa extra de *rollback*, caso a especulação esteja errada. Estes trabalhos ficam designados à pesquisas futuras pertinentes ao tema de reúso de computação em modelos *Dataflow*.

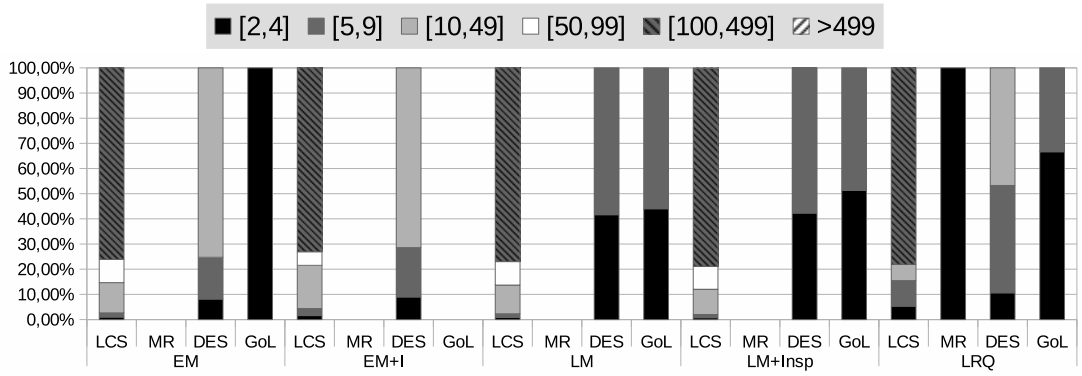
7.10 Distribuição de subgrafos redundantes

Reutilização de subgrafos possui um maior potencial de economizar ciclos de computação do que reutilização simples de nós, pois com o reúso de um subgrafo, vários nós podem ser reutilizados de uma só vez. Apesar de subgrafos maiores poderem economizar mais ciclos, eles são menos prováveis de ocorrer novamente durante a execução de uma aplicação. Considerando este fato, buscamos avaliar a distribuição dos tamanhos de subgrafos redundantes memorizados e reutilizados para cada estratégia de detecção e aplicação. Executamos o experimento 8 vezes utilizando uma granularidade menor possível para maximizar a redundância. O número de *workers* utilizado foi 8, as *caches* NRT e SRT utilizadas eram ilimitadas. A figura 7.19a apresenta a distribuição de tamanho dos subgrafos memorizados e reutilizados durante a execução das aplicações. A figura 7.19b apresenta a distribuição de tamanho de todos os subgrafos memorizados, incluindo reutilizados e não reutilizados. O eixo x apresenta os *benchmarks* e as estratégias de detecção. O eixo y apresenta as distribuições de tamanhos dos subgrafos em relação ao total de subgrafos reutilizados (figura 7.19a) ou todos os subgrafos armazenados (figura 7.19b). Cada coluna representa a porcentagem de subgrafos reutilizados (figura 7.19a) ou armazenados (figura 7.19b) com 2 a 4 nós ($[2, 4]$), 5 a 9 nós ($[5, 9]$), 10 a 49 nós ($[10, 49]$), 50 a 99 nós ($[50, 99]$), 100 a 499 nós ($[100, 499]$) e com mais de 499 nós (> 499).

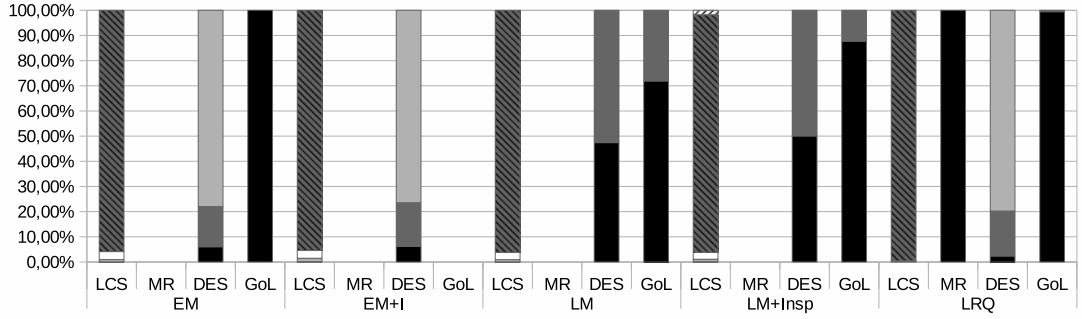
Note que, para aplicação LCS e em todas estratégias de reúso, os subgrafos reutilizados (figura 7.19a) possuem tamanhos maiores. Os subgrafos de 100 a 499 nós compõe $76,13\%$, $73,07\%$, $76,99\%$, $78,66\%$ e $78,03\%$ para as estratégias EM, EM+I, LM, LM+I e LRQ, respectivamente . Este resultado é possivelmente explicado por conta desta aplicação possuir um comportamento de reúso contínuo nos nós das extremidades do grafo. No grafo LCS, cada nó representa um elemento da matriz de comparação. Essa aplicação, para os elementos das extremidades da matriz, propaga um mesmo resultado para os seus vizinhos a direita ou para os vizinhos abaixo na maior parte das vezes. Isso favorece um reúso contínuo, produzindo maiores subgrafos redundantes. Analisando a distribuição de todos os subgrafos construídos (figura 7.19b), reutilizados ou não, para esta aplicação, também podemos verificar que a predominância de subgrafos armazenados era formada por aqueles com 100 a

499 nós. Aproximadamente, mais de 94, 4% dos subgrafos armazenados eram destas proporções para todas as estratégias.

A aplicação GoL mostrou um comportamento interessante para as estratégias EM e EM+I. Vemos, nas figuras 7.19a e 7.19b, que 100% dos subgrafos reutilizados e memorizados na estrategia EM possuíam tamanhos entre 2 e 4 nós, mas quando ativamos a inspeção (EM+I), esse reúso de subgrafos tornou-se inexistente. A figura 7.19b mostra que para a estratégia LM a distribuição de subgrafos estava em 71, 92% para subgrafos com tamanhos entre 2 e 4 nós e 28, 08% para subgrafos com tamanhos entre 5 e 9 nós. Entretanto, quando consideramos a inspeção, os subgrafos menores são favorecidos, pois os resultados mostram uma distribuição de 87, 74% e 12, 26% para subgrafos de tamanhos entre 2 a 4 nós e 5 a 9 nós, respectivamente. Acreditamos que apesar da inspeção aumentar o reúso detectado, ela favorece a formação de subgrafos menores, pois ela remove os nós da fila de prontos mais cedo, isso ocasiona um reúso com uma *cache* ainda pouco aquecida, favorecendo a formação de subgrafos menores. A estratégia LRQ produziu predominantemente subgrafos de 2 a 4 nós, conforme mostra A figura 7.19b, mas permitiu que em 33, 33% das vezes, subgrafos de 5 a 9 nós fossem reutilizados, conforme apresentado na figura 7.19b.



(a) Distribuição de subgrafos reutilizados para todas as estratégias e aplicações.



(b) Distribuição de subgrafos memorizados para todas as estratégias e aplicações.

Figura 7.19: Distribuição de subgrafos redundantes.

Nota-se que para o *benchmark* 3-DES, a distribuição de subgrafos reutilizados (Figura 7.19a) e criados (Figura 7.19b) não sofreu influências significativas, quando

comparamos as estratégias EM com EM+I e LM com LM+I. Isso ocorre pois, conforme visto na Seção 7.5, a inspeção tem pouca participação quando utilizada uma *cache* global para esta aplicação. É interessante notar que para estratégia EM e EM+I os subgrafos reutilizados possuem entre 10 e 49 nós, enquanto os subgrafos mais utilizados nas estratégias LM e LM+I são menores, possuindo entre 5 e 9 nós. Acreditamos que esse comportamento tenha ocorrido, pois as estratégias EM e EM+I fazem a detecção de redundância imediata e não paralisam a execução, isso permite que resultados de nós mais posteriores no grafo sejam conhecidos em tempo suficiente de serem utilizados na criação de subgrafos redundantes, o que propicia a existência de subgrafos mais extensos. Ao contrário das estratégias LM e LM+I, as quais paralisam a execução para processar nós em lotes, impedindo que resultados mais posteriores na execução sejam conhecidos, gerando assim subgrafos mais curtos. A estratégia LRQ favoreceu a construção de subgrafos mais extensos, aproximadamente $79,76\%$ para subgrafos com 10 a 49 nós. Isso ocorreu por motivos semelhantes ao ocorrido na estratégia EM e EM+I, mas por conta desta estratégia inserir mais lacunas aos contextos de entradas, subgrafos menores também tiveram uma quantidade mais expressiva de reúso, aproximadamente $10,63\%$ e $42,65\%$ para subgrafos com 2 a 4 nós e 5 a 9 nós respectivamente.

O *benchmark MapReduce* somente apresentou construção de subgrafos redundantes na estratégia LRQ, e estes possuíam entre 2 e 4 nós. Por conta disso, é predominante a coluna $[2, 4]$ na figura 7.19a para esta aplicação e estratégia.

7.11 Recomendações de *hardware*

Nesta seção, utilizamos os resultados produzidos por nossos experimentos para inferir possíveis configurações de *caches* NRT e SRT em uma arquitetura *Dataflow* implementada em *hardware*. O critério principal foi priorizar esquemas da DF-DTM que apresentaram mais reúso de subgrafos. Também não consideramos para memorização nós que fazem leitura de arquivos. Isso foi feito com intuito de diminuir o tamanho de entrada das *caches*.

A tabela 7.4 apresenta as recomendações de componentes da DF-DTM e estratégias de reúso para os diferentes *benchmarks*. A coluna "Estr."indica qual a estratégia de detecção de reúso acreditamos ser melhor para a aplicação. O campo "Tipo de NRT"indica qual tipo de NRT foi escolhida: global, compartilhada *(S,n)* ou local. As colunas "Número de entradas"e "Tamanho das *caches*"indicam a quantidade de entradas da *cache* e o tamanho resultante dela em *Kbytes*, respectivamente. O campo "Tamanho do *buffer*"indica quantos nós podem ser armazenados no *buffer* de construção. Este *buffer* dita o tamanho máximo que um subgrafo redundante pode ter. Por fim, o campo "Inspeção"indica se o mecanismo de inspeção deve ser

ativado (S) ou não (N).

-	-	-	**Núm. de Entradas**		**Tamanho das *caches***		-	-
App	**Estr.**	**Tipo de NRT**	**NRT**	**SRT**	**NRT**	**SRT**	**Tamanho do *buffer***	**Inspeção**
LCS	EM	S,2	200	200	7,54 Kb	39,87 Kb	5	S
3-DES	LM	S,2	3200	3200	80,86 Kb	833,59 Kb	10	S
GoL	LRQ	S,8	800	800	3,81 Kb	17,19 Kb	3	N
MR	-	-	-	-	-	-	-	S

Tabela 7.4: Recomendações para os diferentes *benchmarks*.

As tabelas 7.5 e 7.6 apresentam o tamanho dos campos em *bits* para as *caches* NRT e SRT, respectivamente. Os campos "*Indx*"na NRT e "Membros"na SRT foram removidos, pois não são essenciais para uma implementação da DF-DTM em *hardware*. Os tamanhos de cada campo foram calculados utilizando informações específicas de cada *benchmark*. Estas informações são apresentadas na tabela 7.7.

App	**Inp_Op**	**Out_Op**	**Node_ID**	**Node Group_ID**	**Node Type**	**Total**
LCS	144	144	18	1	2	38,625 *bytes*
3-DES	128	64	10	1	4	25,875 *bytes*
GoL	9	9	16	3	2	4,875 *bytes*

Tabela 7.5: Tamanho dos campos para a *cache* NRT.

App	**Inp_Ctx**	**Out_Ctx**	**Tamanho**	**Total**
LCS	815	815	3	204,125 *bytes*
3-DES	1390	740	4	266,75 *bytes*
GoL	87	87	2	22 *bytes*

Tabela 7.6: Tamanho dos campos para a *cache* SRT.

A tabela 7.7 apresenta informações utilizadas para calcular os tamanhos dos campos das *caches*. O campo "Configuração"apresenta os componentes dos operandos de entrada e saída dos nós selecionados para memorização. O campo "Tamanho"apresenta o tamanho desses operandos em *bits*. Os campos "Entrada.Arestas"e "Saída.Arestas"apresentam a quantidade máxima de operandos de entrada e de saída que um nó pode ter, respectivamente. Os campos "Nós", "Grupos"e "Tipos"da aba "Quantidades"apresentam as quantidades de nós do grafo da aplicação, grupos de nós e de tipos de nós, respectivamente.

A tabela 7.8 apresenta as fórmulas e valores utilizados para calcular as informações presentes nas tabelas 7.4, 7.5 e 7.6. O campo "Elemento"apresenta

-	-		Nó		Grafo		
-	Operando		Entrada	Saída	Quantidades		
App	**Configuração**	**Tamanho**	**Arestas**	**Arestas**	**Nós**	**Grupos**	**Tipos**
LCS	(1 *Char*, 2 *Ints*)	72	2	2	250004	2	4
3-DES	8 *bytes*	64	2	1	584	2	9
GoL	1 *bit*	1	9	9	50003	8	4

Tabela 7.7: Informações dos *benchmarks* para cálculo das *caches*.

o nome dos campos calculados ou o nome de constantes, e o campo "Valor/Formula"apresenta o valor das constantes ou a fórmula utilizada para o cálculo dos campos.

Elemento	**Valor/Fórmula**
1 Char	8 *bits*
1 Int	32 *bits*
$Tab_{7.5}.Inp\_Op$	$Tab_{7.7}.Operando.Tamanho \times Tab_{7.7}.Nó.Entrada.Arestas$
$Tab_{7.5}.Out\_Op$	$Tab_{7.7}.Operando.Tamanho \times Tab_{7.7}.Nó.Saída.Arestas$
$Tab_{7.5}.Node\_ID$	$\lceil log_2(Tab_{7.7}.\text{Grafo.Quantidades.Nós})\rceil$
$Tab_{7.5}.Node\_Group\_ID$	$\lceil log_2(Tab_{7.7}.\text{Grafo.Quantidades.Grupos})\rceil$
$Tab_{7.5}.Node\_Type$	$\lceil log_2(Tab_{7.7}.\text{Grafo.Quantidades.Tipos})\rceil$
$Tab_{7.6}.Inp\_Ctx$	$(\text{Tab}_{7.5}.Inp\_Op + \lceil log_2(\text{Tab}_{7.7}.\text{Nós.Entrada.Arestas})\rceil + \text{Tab}_{7.5}.Node\_ID) \times \text{Tab}_{7.4}.\text{Tamanho\_do\_}buffer$
$Tab_{7.6}.Out\_Ctx$	$(\text{Tab}_{7.5}.Out\_Op + \lceil log_2(\text{Tab}_{7.7}.\text{Nós.Saída.Arestas})\rceil + \text{Tab}_{7.5}.Node\_ID) \times \text{Tab}_{7.4}.\text{Tamanho\_do\_}buffer$
$Tab_{7.6}.Tamanho$	$\lceil log_2(\text{Tab}_{7.4}.\text{Tamanho\_do\_buffer})\rceil$
$Tab_{7.4}.Tamanho\_das\_caches.NRT$	$\text{Tab}_{7.4}.\text{Núm\_de\_entradas.NRT} \times Tab_{7.5}.Total$
$Tab_{7.4}.Tamanho\_das\_caches.SRT$	$\text{Tab}_{7.4}.\text{Núm\_de\_entradas.SRT} \times Tab_{7.6}.Total$

Tabela 7.8: Descrições de elementos e fórmulas para cálculos dos campos das tabelas 7.4, 7.5 e 7.6.

Para a aplicação LCS, escolhemos a estratégia EM com inspeção, pois ela necessitou de apenas 200 entradas para alcançar um reúso de $95,4\%$, que é semelhante a uma *cache* infinita. Quando limitamos o tamanho do *buffer* em 5 nós, foi possível ter uma contribuição da SRT de $6,3\%$, aproximadamente duas vezes mais do que quando o *buffer* é ilimitado. Escolhemos a NRT compartilhada com duas tabelas (*S,2*), pois esta apresentou resultados semelhantes a uma *cache* global. A vantagem de ter mais de uma tabela de *cache* é a diminuição de possíveis contenções.

A aplicação *MapReduce* não apresentou reúso por *caches* nas estratégias EM e LM, somente na estratégia LRQ. Nas estratégias EM e LM, todo reúso detectado foi

por inspeção, em contrapartida, a estratégia LRQ necessitou de uma *NRT* de 9600 entradas para alcançar taxas de reúso encontradas em uma *cache* infinita. Por conta disso, acreditamos que para este *benchmark* é melhor desabilitar todas as *caches* e manter somente o mecanismo de inspeção. Temos assim um *hardware* possivelmente mais simples e que aproveite bem o reúso da aplicação.

O *benchmark* 3-DES apresentou uma taxa de reúso de subgrafos de $15,11\%$ quando limitamos o *buffer* em 3 nós. Entretanto, preferimos aumentar o tamanho do *buffer* para 10, obtendo uma taxa de reúso de subgrafos de 12%. Apesar da perda de $3,11\%$ na taxa de reúso pela SRT, possibilitamos que subgrafos maiores existam e que mais nós sejam ignorados simultaneamente, o que poderia gerar mais economia de ciclos. Para este *benchmark*, um esquema de NRTs compartilhadas com 2 tabelas e 3200 entradas atingiu uma taxa de reúso semelhante a uma *cache* global e infinita.

Para a aplicação GoL, devido a sua baixa reutilização de subgrafos, optamos por utilizar a estratégia LRQ. A estratégia LRQ combinada a uma NRT compartilhada com 8 tabelas (*S,8*) e 800 entradas tornou possível a obtenção de uma taxa de reúso total de $98,9\%$, o que é próximo ao máximo de $99,7\%$ permitido pela aplicação.

Capítulo 8

Conclusão e Trabalhos Futuros

8.1 Conclusão

Neste trabalho propomos uma avaliação de potencial de redundância de computação em um modelo de execução *dataflow*. Nós criamos a DF-DTM (*dataflow dynamic trace memoization* e a implementamos na biblioteca *dataflow* para *python*, Sucuri. A DF-DTM é uma técnica de exploração de redundância de computação em um ambiente *dataflow* que permite a detecção e reutilização de nós e subgrafos *dataflow*. Isto é feito utilizando dois tipos básicos de *cache*, a NRT (*Node Reuse Table*) e a SRT (*Subgraph Reuse Table*). A NRT e SRT possuem uma política de substituição LRU quando os seus tamanhos são limitados. No caso da NRT, ela pode ser subdividida em várias tabelas para simular organização de *caches*, tais como, *caches* L1 e L2. Além das *caches* NRT e SRT, introduzimos também à DF-DTM o mecanismo de inspeção que se beneficia de tarefas prontas em espera de recursos e operandos resultantes que possam satisfazer essas tarefas.

Avaliamos o comportamento do reúso utilizando 5 estratégias distintas: EM (*Eager Match*), a qual inicia a detecção de reúso assim que um operando resultante é retornado ao escalonador. EM+I, que consiste na estratégia EM, mas com o mecanismo de inspeção ativado. A estratégia LM (*Lazy Match*) somente inicia a detecção de redundância após ler todos os operandos enviados pelos *workers*. De forma semelhante à EM+I, a estratégia LM+I, consiste na estratégia LM com o mecanismo de inspeção ativado. Por fim, implementamos a estratégia LRQ que realiza a detecção tardia de reúso, isto é, no momento em que o *worker* solicita uma nova tarefa. Desenvolvemos 5 *benchmarks dataflow* para avaliar essas estratégias e nosso mecanismo de reúso, estes são: LCS, *Game of Life*, *MapReduce*, criptografia 3-DES e *Unbounded Knapsack* (algoritmo da mochila).

Nossos experimentos revelaram uma quantidade expressiva de redundância em todos as aplicações, excetuando-se a aplicação *Knapsack*. Para os demais *bench-*

marks, notamos uma taxa de redundância máxima de $72,35\%$, $99,74\%$, $98,83\%$ e $54,73\%$ para as aplicações 3-DES, GoL, LCS e *MapReduce*, respectivamente. É impressionante notar que nas aplicações LCS e GoL apenas $0,26\%$ e $1,17\%$ são operações distintas, o restante são computações redundantes. Deste reúso máximo, nossas estratégias EM, EM+I, LM, LM +I e LRQ conseguiram reutilizar até $65,8\%$, $66,17\%$, $63,77\%$, $72,21\%$, para aplicação 3-DES, respectivamente, $79,48\%$, $99,67\%$, $79,49\%$, $99,7\%$ e $99,7\%$ para a aplicação GoL, respectivamente, $81,78\%$, $97,08\%$, $81,88\%$, $97,69\%$ e $98,49\%$ para aplicação LCS, respectivamente e, por fim, $0,0\%$, $50,01\%$, $5,15\%$, $51,91\%$ e $54,31\%$ para aplicação *MapReduce*, respectivamente.

O mecanismo de inspeção se provou muito útil para aumentar o reúso explorado nas estratégias EM+I e LM+I. A inspeção também se demonstrou essencial nestas estratégias para aplicações cujos grafos apresentem explosão de paralelismo.

O alto nível de redundância encontrado nos experimentos realizados foi atingido utilizando uma granularidade muito fina para as tarefas *dataflow*, justificando a técnica de reúso de computação como algo a ser explorado em *hardware*, por exemplo, aceleradores FPGAs *dataflow*, o qual permite a criação de *hardwares* customizados ao problema.

O reúso de subgrafos redundantes foi baixo, apesar do alto nível de redundância encontrado nas aplicações. O maior contribuição da *cache* de subgrafos que alcançamos foi de $12,04\%$ para aplicação *3-DES* com o uso da estratégia LM+I. O baixo uso de subgrafos se deve à quantidade de lacunas nos contextos de entrada destes e subgrafos com grandes contextos de entrada. Outro fator que limita o uso de subgrafos redundantes é o fato deles dependerem dos identificadores dos nós. Este fator limitante poderia ser superado completamente se realizássemos isomorfismo de subgrafos para detecção de subgrafos redundantes, porém isso geraria um aumento exponencial de complexidade no *hardware dataflow*.

Características do modelo *Dataflow* demonstraram afinidades com a técnica de reúso com granularidade fina. Em arquiteturas de Von Neumann, as tabelas de reúso são indexadas pelo *Program Counter* e necessitam de tratamentos adicionais para reúso de instruções especias, por exemplo, *stores* e *loads*. Por conta do modelo *Dataflow* visualizar as instruções como operações em um grafo, não há necessidade de uma memória global, permitindo que mais nós (instruções no modelo Von Neumann), sejam explorados. Outra vantagem que o modelo *dataflow* demonstrou foi a ausência do *Program Counter*. Isto permitiu que nós fossem reutilizados baseando-se somente no seu tipo (mnemônico da instrução no modelo Von Neumann), e, assim, o reúso encontrado foi potencializado. Vemos que quando utilizamos um modelo de reúso em *dataflow* análogo aos criados para arquiteturas Von Neumann, isto é, uma esquematização de *caches* completamente local aos nós do grafo, onde os Ids dos nós seriam semelhantes ao *Program Counter*, o reúso potencial das aplicações

decai de 81, 78% para 15, 58%, 54, 73% para 3, 12%, 72, 35% para 65, 50% e 99, 74% para 7, 31% para as aplicações LCS, *MapReduce*, 3-DES e GoL, respectivamente. São decréscimos expressivos para os *benchmarks* LCS, GoL e *MapReduce*. Para a aplicação 3-DES, o decréscimo foi menor, pois este possui mais nós distintos e um grafo mais serial.

O paralelismo do modelo *dataflow*, que é o seu principal atrativo, não apresentou influência significativa na detecção de redundância pelas *caches*. Para o mecanismo de inspeção, o aumento de paralelismo demonstrou um decréscimo no reúso detectado por esta técnica, mas o decréscimo não foi expressivo.

Por conta de utilizarmos granularidade fina para as tarefas *dataflow*, é natural que ocorra um aumento expressivo nos custos de comunicação. Um resultado encorajador foi o fato da DF-DTM implementada na Sucuri, através do reúso de computação, reduzir estes custos. Alcançamos uma redução de até 98, 49%, 54, 31%, 72, 21% e 99, 70% nos custos de comunicação para as aplicações LCS, *MapReduce*, 3-DES e GoL, respectivamente.

Neste trabalho também conseguimos demonstrar a viabilidade da biblioteca *dataflow* para *python*, Sucuri, como um simulador funcional. Com isso conseguimos avaliar potencial de novas funcionalidades para uma futura simulação e implementação arquitetural em *hardware dataflow*. A Sucuri também demonstrou uma alta programabilidade para criação de aplicações DAG *dataflow* e de *streams*.

8.2 Trabalhos Futuros

No presente trabalho apresentamos um estudo sobre a exploração de computações redundantes em um modelo de execução *dataflow* utilizando a Sucuri. Nossos resultados abrem muitos caminhos para contribuições futuras referentes a este tema, tais como, o desenvolvimento de mais aplicações *dataflow*, a criação de um simulador arquitetural para mensurar os custos e benefícios reais de ter as técnicas aqui apresentadas em um acelerador *dataflow* FPGA.

A DF-DTM apresentou baixo reúso de subgrafos redundantes. Isso é um ponto negativo de nossa técnica, pois inferimos que *speedups* significativos podem ser atingidos se conseguirmos uma reutilização razoável desses subgrafos. Estamos avaliando uma forma de construir subgrafos redundantes com contextos de entrada reduzidos, mas com tamanhos maiores, sem inserir muita complexidade arquitetural. Isso garantiria uma menor chance de lacunas no contexto de entrada e geraria mais aproveitamentos de subgrafos

Atualmente, a Sucuri possui um escalonador central que introduz uma certa limitação de escalabilidade e baixa tolerância às falhas. Trabalhos estão sendo conduzidos para distribuir esse escalonador [44]. Seria interessante estudar o contexto

de reúso de nós e subgrafos em uma Sucuri cujos escalonadores são completamente distribuídos.

Neste trabalho realizamos o agrupamento de nós para organizar os acessos às *caches* compartilhadas e simular uma esquematização de *caches* L1 e L2. A estratégia utilizada para agrupar os nós foi trivial e, possivelmente, novas estratégias de agrupamento podem ser utilizadas para aumentar o reúso detectado em cenários de *caches* compartilhadas. Uma estratégia interessante seria utilizar a distância de reúso, isto é, quantas arestas do grafo *dataflow* um determinado operando atravessa até que seja reutilizado. Uma clusterização de nós utilizando essa distância como fator agrupador pode aumentar o reúso encontrado em *caches* subdividas.

O conceito de *caches* compartilhadas também é algo interessante de se avaliar no contexto de *hardwares* reconfiguráveis, pois em uma arquitetura *Dataflow* deste tipo podemos ter *caches* distintas por *cores dataflow*. Além disso, em um hardware reconfigurável como uma FPGA, seria possível implementar uma *cache* específica para nós ou grupos de nós no grafo *dataflow*, o qual estaria especificado através de unidades lógicas e barramentos no *chip*. Também seria possível realizar operações de *profiling* em diferentes aplicações para validar se o reúso é viável. As informações resultantes do *profiling* poderiam ser utilizadas na implementação de *caches* de reúso para aplicações *Dataflow* em FPGAs. Isto seria feito com o objetivo de potencializar a exploração de redundância das aplicações.

O estudo de exploração de computação redundantes em grafos *dataflow* dinâmicos também propõe um caminho de pesquisa interessante e desafiador. Neste trabalho, devido à Sucuri utilizar DAGs, os grafos dinâmicos ficaram fora do escopo do mesmo. Apesar de utilizarmos grafos de *stream*, que utilizam conceitos de iterações dinâmicas através de operandos com *tags*, ainda sim, seria interessante um estudo mais aprofundado de grafos com ciclos. Uma proposta seria a investigação da possibilidade de subgrafos redundantes eliminarem a necessidade de computação de alguns laços no grafo.

Em nossa pesquisa focamos em estudar o potencial da DF-DTM visando uma implementação real em arquiteturas reconfiguráveis, por conta disso, nossas implementações de *benchmarks dataflow* possuíam grafos com nós que representavam instruções relativamente complexas. Seria interessante explorar o reúso de nós e subgrafos em arquiteturas e modelos *dataflow* de propósito geral, tais como, *WaveScalar*, *TALM* [46] e *hardware* semelhantes à *Manchester Machine* [11]. Arquiteturas *Dataflow* generalistas possuem instruções simples, tais como, *add*, *sub*, *stear*, *mult*, entre outras, que são utilizadas para formar o grafo de uma aplicação. Estas arquiteturas podem facilitar o reúso de subgrafos, pois, em teoria, os subgrafos formados possuiriam menores contextos de entrada.

Uma ideia interessante no reúso de subgrafos é utilizá-los como forma de guiar o

processo de criação de super-instruções que possam ser utilizadas em um processador *Dataflow*. Nosso trabalho demonstrou que a construção de subgrafos sem considerar quais nós são adicionados ao mesmo pode diminuir o paralelismo da aplicação se estes subgrafos forem transformados em super-instruções. Vemos isso ocorrer no grafo do *benchmark* LCS, por exemplo, pois este constrói subgrafos redundantes com contextos de entrada grandes. Caso esses subgrafos fossem transformados em super-instruções, eles necessitariam de muitos operandos de entrada para instanciarem uma tarefa, o que diminuiria o paralelismo da aplicação. Entretanto, é possível que ao adicionarmos nós ao subgrafo, se o fizermos de forma que os contextos de entrada desses não cresçam, poderíamos ter uma super instrução que não diminuísse o paralelismo do grafo. Isso poderia melhorar o aproveitamento dos recursos da máquina *Dataflow* e auxiliar no escalonamento de tarefas em um *hardware Dataflow* distribuído.

Demonstramos neste trabalho que certos tipos de grafos possuem topologias que podem dificultar o reúso por *caches*. Por exemplo, a aplicação *MapReduce* teve todo seu reúso detectado por inspeção quando utilizadas as estratégias LM+I e EM+I. Pode ser interessante modificar a aplicação para potencializar as taxas de reúso real da mesma. Um desenvolvimento de aplicações em grafos *dataflow* orientado ao reúso de computação pode fornecer melhores aproveitamentos dos componentes da DF-DTM.

Nosso trabalho apresentou o mecanismo de inspeção que se demonstrou útil para determinados grafos com alto grau de paralelismo, e demonstrou que, por si só, possui um certo potencial de reúso de computação. Uma investigação necessária é analisar o quanto desse potencial pode ser explorado se utilizarmos uma fila de tamanho finito. Se o potencial se mantiver relevante, ainda com esta limitação, isso seria benéfico, pois a inspeção é uma técnica que não precisa de tabelas *caches* adicionais e, possivelmente, não colocaria sobre o *hardware dataflow* uma maior complexidade.

Referências Bibliográficas

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

[2] A. DURAN, R. FERRER, E. A. R. M. B. E. J. L. "The OmpSs Programming Model". 2009. Disponível em: <`https://pm.bsc.es/ompss`>.

[3] MARZULO, L. A. J., ALVES, T., FRANÇA, F. M. G., et al. "TALM: A Hybrid Execution Model with Distributed Speculation Support". In: Bentes, C., Drummond, L. M. A., Farias, R. (Eds.), *1st Workshop on Applications for Multi and Many Core Architectures, held in conjunction with the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC PAD 2010)*, p. 31–36, Petrópolis, Rio de Janeiro, Brazil, October 2010. IEEE, IEEE. doi: 10.1109/SBAC-PADW.2010.8.

[4] SOLINAS, M., BADIA, M., BODIN, F., et al. "The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices". In: *IEEE Proceedings of the 16th EUROMICRO-DSD*, pp. 272–279, Santander, Spain, 2013. ISBN: 978-0-7695-5074-9. doi: 10.1109/DSD.2013.39. Disponível em: <`http://dx.doi.org/10.1109/DSD.2013.39`>.

[5] DA COSTA, A. T., FRANCA, F. M. G., FILHO, E. M. C. "The dynamic trace memoization reuse technique". In: *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pp. 92–99, 2000.

[6] ALVES, T. A. O., GOLDSTEIN, B. F., FRANÇA, F. M. G., et al. "A Minimalistic Dataflow Programming Library for Python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pp. 96–101, Oct 2014.

[7] KIRK, D. B., HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723, 9780123814722.

[8] RAHMAN, R. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. 1st ed. Berkely, CA, USA, Apress, 2013. ISBN: 1430259264, 9781430259268.

[9] CHAPMAN, B., JOST, G., PAS, R. V. D. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN: 0262533022, 9780262533027.

[10] GROPP, W., LUSK, E., SKJELLUM, A. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. Cambridge, MA, USA, MIT Press, 1999. ISBN: 0-262-57132-3.

[11] 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`>.

[12] PELL, O., MENCER, O., TSOI, K., et al. "Maximum performance computing with dataflow engines". In: *High-Performance Computing Using FPGAs*, pp. 747–774, 2013.

[13] DE OLIVEIRA ALVES, T. A. *DATAFLOW EXECUTION FOR RELIABILITY AND PERFORMANCE ON CURRENT HARDWARE*. Ph.d., COPPE/UFRJ, Centro de Tecnologia - Av. Horácio Macedo, 2030 - 101 - Cidade Universitária, 2014.

[14] 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.

[15] DURAN, A., AYGUADÉ, E., BADIA, R. M., et al. "OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES", *Parallel Processing Letters*, v. 21, pp. 173–193, 2011-03-01 2011.

[16] BOSILCA, G., BOUTEILLER, A., DANALIS, A., et al. "DAGuE: A generic distributed DAG engine for High Performance Computing." *Parallel Computing*, v. 38, n. 1-2, pp. 37–51, 2012.

[17] 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.

[18] SWANSON, S., MICHELSON, K., SCHWERIN, A., et al. "WaveScalar". In: *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 291–302, Dec 2003.

[19] MICHIE, D. ""Memo" Functions and Machine Learning", *Nature*, v. 218, pp. 19–22, 1968.

[20] 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.

[21] ALVES, T. A. O., GOLDSTEIN, B. F., FRANÇA, F. M. G., et al. "A Minimalistic Dataflow Programming Library for Python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pp. 96–101, Oct 2014.

[22] CYTRON, R., FERRANTE, J., ROSEN, B. K., et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Trans. Program. Lang. Syst.*, v. 13, n. 4, pp. 451–490, out. 1991. ISSN: 0164-0925. doi: 10.1145/115372.115320. Disponível em: <http://doi.acm.org/10.1145/115372.115320>.

[23] SWANSON, S., SCHWERIN, A., MERCALDI, M., et al. "The WaveScalar Architecture", *ACM Trans. Comput. Syst.*, v. 25, n. 2, pp. 4:1–4:54, maio 2007. ISSN: 0734-2071. doi: 10.1145/1233307.1233308. Disponível em: <http://doi.acm.org/10.1145/1233307.1233308>.

[24] CULLER, D. E., SAH, A., SCHAUSER, K. E., et al. "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-controlled Threaded Abstract Machine". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pp. 164–175, New York, NY, USA, 1991. ACM. ISBN: 0-89791-380-9. doi: 10.1145/106972.106990. Disponível em: <http://doi.acm.org/10.1145/106972.106990>.

[25] THIES, W., KARCZMAREK, M., AMARASINGHE, S. P. "StreamIt: A Language for Streaming Applications". In: *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 179–196, London, UK, UK, 2002. Springer-Verlag. ISBN: 3-540-43369-4. Disponível em: <http://dl.acm.org/citation.cfm?id=647478.727935>.

[26] POP, A., COHEN, A. "OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs", *ACM Trans. Archit. Code Optim.*, v. 9, n. 4, pp. 53:1–53:25, jan. 2013. ISSN: 1544-3566. doi: 10.1145/2400682.2400712. Disponível em: <http://doi.acm.org/10.1145/2400682.2400712>.

[27] LEE, B., HURSON, A. R. "Issues in dataflow computing", *ADV. IN COMPUT*, v. 37, pp. 285–333, 1993.

[28] ALVES, T. A. O., MARZULO, L. A. J., FRANCA, F. M. G., et al. "Trebuchet: Exploring TLP with Dataflow Virtualisation", *Int. J. High Perform. Syst. Archit.*, v. 3, n. 2/3, pp. 137–148, maio 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466. Disponível em: <http://dx.doi.org/10.1504/IJHPSA.2011.040466>.

[29] CASSEL, J. "Probabilistic Programming with Stochastic Memoization", *The Mathematica Journal*, v. 16, pp. 1–3, jan. 2014. ISSN: 1097-1610. doi: dx.doi.org/doi:10.3888/tmj.16-1. Disponível em: <http://www.mathematica-journal.com/2014/01/probabilistic-programmingwith-stochastic-memoization/>.

[30] PILLA, M. L., NAVAUX, P. O. A., DA COSTA, A. T., et al. "The limits of speculative trace reuse on deeply pipelined processors". In: *Proceedings. 15th Symposium on Computer Architecture and High Performance Computing*, pp. 36–44, Nov 2003. doi: 10.1109/CAHPC.2003.1250319.

[31] SAULO T. OLIVEIRA, LEANDRO SANTIAGO, B. F. G. F. M. G. F. M. C. S. D. C. A. S. N. A. C. S. I. M. C. T. A. O. A. L. A. J. M. C. B. "DTM@GPU: Explorando redundância de traços em GPU". In: *WSCAD 2016: X Simpósio em Sistemas Computacionais de Alto Desempenho*, Aracaju, SE, Brazil, October 2016.

[32] SODANI, A., SOHI, G. S. "Dynamic Instruction Reuse". In: *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pp. 194–205, June 1997.

[33] CONNORS, D. A., HUNTER, H. C., CHENG, B.-C., et al. "Hardware Support for Dynamic Activation of Compiler-directed Computation Reuse", *SIGARCH Comput. Archit. News*, v. 28, n. 5, pp. 222–233, nov. 2000. ISSN: 0163-5964. doi: 10.1145/378995.379243. Disponível em: <http://doi.acm.org/10.1145/378995.379243>.

[34] CONNORS, D. A., HWU, W.-M. W. "Compiler-directed Dynamic Computation Reuse: Rationale and Initial Results". In: *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pp. 158–169, Washington, DC, USA, 1999. IEEE Computer Society. ISBN: 0-7695-0437-X. Disponível em: <http://dl.acm.org/citation.cfm?id=320080.320104>.

[35] DING, Y., LI, Z. "A compiler scheme for reusing intermediate computation results". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 277–288, March 2004. doi: 10.1109/CGO.2004.1281681.

[36] HUANG, J., LILJA, D. "Exploiting basic block value locality with block reuse". In: *IEEE High-Performance Computer Architecture Symposium Proceedings*, pp. 106–114, IEEE Comp Soc, 1999.

[37] CALDER, B., FELLER, P., EUSTACE, A. "Value profiling". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*, pp. 259–269, Dec 1997. doi: 10.1109/MICRO.1997.645816.

[38] HWU, W. M. W., MAHLKE, S. A., CHEN, W. Y., et al. "The superblock: An effective technique for VLIW and superscalar compilation", *The Journal of Supercomputing*, v. 7, n. 1, pp. 229–248, 1993. ISSN: 1573-0484. doi: 10.1007/BF01205185. Disponível em: <http://dx.doi.org/10.1007/BF01205185>.

[39] GUO, P. J., ENGLER, D. "Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 287–297, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0562-4. doi: 10.1145/2001420.2001455. Disponível em: <http://doi.acm.org/10.1145/2001420.2001455>.

[40] VAROQUAUX, G. "Joblib: running Python functions as pipeline jobs". 2008. Disponível em: <https://pythonhosted.org/joblib/index.html>.

[41] WANG, W., RAGHUNATHAN, A., JHA, N. K. "Profiling driven computation reuse: an embedded software synthesis technique for energy and performance optimization". In: *17th International Conference on VLSI Design. Proceedings.*, pp. 267–272, 2004. doi: 10.1109/ICVD.2004.1260935.

[42] KLEIN, B. "Memoization with Decorators". 2011. Disponível em: <http://www.python-course.eu/python3_memoization.php>.

[43] FRANKLIN, M. A., TYSON, E. J., BUCKLEY, J., et al. "Auto-pipe and the X Language: A Pipeline Design Tool and Description Language". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pp. 117–117, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 1-4244-0054-6. Disponível em: <http://dl.acm.org/citation.cfm?id=1898953.1899049>.

[44] SILVA, R. J. N., 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)*, pp. 37–42, Oct 2016. doi: 10.1109/SBAC-PADW.2016.15.

[45] GIORGI, R. "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices". In: *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pp. 303–304, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1215-8. doi: 10.1145/2212908.2212959. Disponível em: <http://doi.acm.org/10.1145/2212908.2212959>.

[46] ALVES, T., MARZULO, L. A. J., FRANÇA, F. M. G., et al. "Trebuchet: Explorando TLP com Virtualização DataFlow". In: *WSCAD-SSC 2009: X Simpósio em Sistemas Computacionais*, São Paulo, SP, Brazil, October 2009.

[47] BOSILCA, G., BOUTEILLER, A., DANALIS, A., et al. "DAGuE: A Generic Distributed DAG Engine for High Performance Computing", *Parallel Comput.*, v. 38, n. 1-2, pp. 37–51, jan. 2012. ISSN: 0167-8191. doi: 10.1016/j.parco.2011.10.003. Disponível em: <http://dx.doi.org/10.1016/j.parco.2011.10.003>.

[48] ARORA, N. S., BLUMOFE, R. D., PLAXTON, C. G. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pp. 119–129, New York, NY, USA, 1998. ACM. ISBN: 0-89791-989-0. doi: 10.1145/277651.277678. Disponível em: <http://doi.acm.org/10.1145/277651.277678>.

[49] CORP., I. "Flow Graph". 2015. Disponível em: <https://www.threadingbuildingblocks.org/tutorial-intel-tbb-flow-graph>.

[50] HAUCK, S., DEHON, A. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2007. ISBN: 9780080556017, 9780123705228.

[51] GOLDSTEIN, B. F., FRANÇA, F. M. G., MARZULO, L. A. J., et al. "Exploiting Parallelism in Linear Algebra Kernels through Dataflow Execution". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pp. 103–108, Oct 2015.

[52] SODANI, A., SOHI, G. S. "Understanding the Differences Between Value Prediction and Instruction Reuse". In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pp. 205–215, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[53] MORENO, A., BALCH, T. "Speeding up Large-Scale Financial Recomputation with Memoization". In: *2014 Seventh Workshop on High Performance Computational Finance*, pp. 17–22, Nov 2014. doi: 10.1109/WHPCF.2014.9.

[54] DALCIN, L. "MPI for Python". 2014. Disponível em: <`http://pythonhosted.org/mpi4py/`>.

[55] 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.

[56] ALVES, C. E. R., CÁCERES, E. N., DEHNE, F., et al. "A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison". In: *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartII*, ICCSA'03, pp. 249–258, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN: 3-540-40161-X.

[57] P. C. GILMORE, R. E. G. "A Linear Programming Approach to the Cutting-Stock Problem", *Operations Research*, v. 9, n. 6, pp. 849–859, 1961. ISSN: 0030364X, 15265463.

[58] GAJINOV, V., STIPIĆ, S., ERIĆ, I., et al. "DaSH: A Benchmark Suite for Hybrid Dataflow and Shared Memory Programming Models: with Comparative Evaluation of Three Hybrid Dataflow Models". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pp. 4:1–4:11, New York, NY, USA, 2014. ACM. ISBN: 978-1-4503-2870-8.

[59] GARDNER, M. "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"", *Scientific American*, v. 223,

pp. 120–123, 1970. Disponível em: <http://www.worldcat.org/isbn/0894540017>.

[60] PAAR, C., PELZL, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 2009. ISBN: 3642041000, 9783642041006.