**Universidade de São Paulo
Instituto de Física de São Carlos**

**Um modelo de execução dirigido pelos dados em
processadores multi-core**

**São Carlos
2020**

Felipe Ferreira

Um modelo de execução dirigido pelos dados em processadores multi-core

Tese apresentada ao Programa de Pós-Graduação em Física do Instituto de Física de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Doutor em Ciências.

Área de Concentração: Física Aplicada

Opção: Física Computacional

Orientador: Prof. Dr. Gonzalo Travieso

Coorientador: Prof. Dr. Carlos Antônio Ruggiero

Versão Corrigida

(Versão original disponível na Unidade que aloja o Programa)

**São Carlos
2020**

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ferreira, Felipe

Um modelo de execução dirigido pelos dados em processadores multi-core / Felipe Ferreira; orientador Gonzalo Travieso; co-orientador Carlos Antônio Ruggiero - versão corrigida -- São Carlos, 2020.

171 p.

Tese (Doutorado - Programa de Pós-Graduação em Física Aplicada Computacional) -- Instituto de Física de São Carlos, Universidade de São Paulo, 2020.

1. Fluxo de dados. 2. Paralelismo. 3. Modelo de execução. 4. *Multi-core*. I. Travieso, Gonzalo, orient. II. Ruggiero, Carlos Antônio, co-orient. III. Título.

Em memória de minha avó materna Maria Gilda Andreotti Dalbeto.

AGRADECIMENTOS

Agradeço aos meus pais Antônio Carlos e Terezinha pela paciência e por sempre me apoiarem. Agradeço ao meu irmão Carlos Humberto por ter despertado a paixão por computação que me trouxe até aqui. Agradeço também à minha avó Gilda e ao meu tio Luiz que foram fundamentais para o meu crescimento como pessoa.

Aos meus orientadores, Gonzalo Travieso e Carlos Antonio Ruggiero, que além de me orientarem com maestria, são espelhos que levarei por toda a minha carreira profissional.

Não posso deixar de agradecer aos amigos que fiz durante a graduação e pós-graduação que me ensinaram a nunca deixar de mirar mais alto. Aos amigos que fiz durante os meus anos de ensino fundamental, médio e técnico eu agradeço pelas memórias e pela amizade que, mesmo distante, nunca deixou de existir.

Agradeço também a todo o corpo de funcionários do Instituto de Física de São Carlos, que permitiram que a pesquisa e escrita desta tese fossem realizadas da melhor maneira possível.

Também agradeço a todos os professores e demais profissionais que foram responsáveis pela minha educação, sem vocês esta pesquisa não teria acontecido.

Por fim, agradeço a você, leitor, por ler este trabalho.

*"I tell you folks
It's harder than it looks"
(It's a Long Way to the Top (If you Wanna Rock'n'Roll); AC/DC)*

*"“Fifty years,” I hackneyed, “is a long time.”
“Not when you’re looking back at them,” she said. “You wonder how they vanished so quickly.””
— Susan Calvin
(I, Robot; Isaac Asimov)*

*Este trabalho foi financiado pela
FAPESP – Fundação de Amparo à Pesquisa do Estado de São Paulo.
Processo #2015/02014-4.*

RESUMO

FERREIRA, F.. **Um modelo de execução dirigido pelos dados em processadores multi-core**. 2020. 171 p. Tese (Doutorado em Ciências) – Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2020.

A mudança conceitual no desenvolvimento dos processadores que permitiu a construção de processadores *multi-core* faz necessária a construção de novos modelos de execução para permitir a extração de toda a capacidade de processamento presente nestes novos processadores. Modelos de execução que utilizam os conceitos de fluxo de dados têm tido sucesso nesta tarefa – estes conceitos permitem que o paralelismo disponível nos códigos seja aproveitado de maneira mais eficiente. E é esta a pesquisa exposta nesta tese: a construção de um modelo de execução que seja eficiente na extração da capacidade de processamento, não só dos processadores x86_64 atuais *multi-core* como também nos futuros processadores *many-core*. A análise do modelo proposto é feita com a comparação de algoritmos implementados no modelo e em modelos de estado da arte e estas implementações são executadas em um ambiente real e em um ambiente simulado, onde o primeiro permite uma análise contemporânea e o último uma análise dos modelos com os processadores *many-core*. A execução de algoritmos foi positiva, com o modelo desenvolvido sendo competitivo e até mesmo superior aos modelos de estado da arte, principalmente com a execução no limiar *many-core*.

Palavras-chave: Fluxo de dados. Paralelismo. Modelo de execução. *Multi-core*.

ABSTRACT

FERREIRA, F.. **A datadriven execution model in multi-core processors**. 2020. 171 p. Tese (Doutorado em Ciências) – Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2020.

With the conceptual change on processor development that allowed the construction of multi-core processors came the necessity of creating new execution models able to extract all of their processing capacity. Models that use the data flow paradigm are successful in this task, as they allow the available parallelism to be exploited efficiently. The research presented in this monograph aims at the development of a new execution model using the data flow paradigm to efficiently extract the full capacity of today's multi-core processors and even of the many-core processors expected to be released soon. A prototype of the model was implemented and analyzed through a comparison with state of the art parallel models using a set of algorithms executed on real and simulated machines. The former allows for an analysis in present-day conditions, while the second allows for some prediction regarding a many-core system environment. The overall result is positive and the developed model is shown to be competitive and even better than some of the state of the art ones, mainly on the many-core region, which is our research goal.

Keywords: Dataflow. Parallelism. Execution Models. Multi-core.

LISTA DE ILUSTRAÇÕES

Figura 1 – Evolução da <i>performance</i> em GFLOPS dos supercomputadores avaliados pelo TOP500.	34
Figura 2 – Número de publicações sobre fluxo de dados nos repositórios da ACM e IEEE. Atualizado em fevereiro de 2020.	35
Figura 3 – Representação de um código na forma de dígrafo.	39
Figura 4 – Evolução temporal da frequência de <i>clock</i> . O eixo das ordenadas exibe a frequência em GHz.	46
Figura 5 – Relação entre potência dissipada e frequência de <i>clock</i> . O eixo das ordenadas exibe a potência dissipada em Watts, enquanto o eixo das abcissas exibe a frequência em GHz.	47
Figura 6 – Evolução temporal do tamanho do <i>cache</i>	48
Figura 7 – Exemplo de execução em um processador sem <i>pipeline</i>	48
Figura 8 – Exemplo de execução em um processador com <i>pipeline</i>	49
Figura 9 – Exemplo de execução em um processador com <i>pipeline</i>	50
Figura 10 – Exemplo de execução em um processador sem execução <i>out-of-order</i>	50
Figura 11 – Exemplo de execução em um processador com execução <i>out-of-order</i>	51
Figura 12 – Evolução temporal do número de <i>threads</i> por processador.	54
Figura 13 – Representação gráfica de um <i>codelet</i> . O conjunto de quadrados da parte superior representa o conjunto de entradas, onde os quadrados preenchidos representam as entradas já disponíveis. Como o <i>codelet</i> não foi executado – o conjunto de entradas não está completamente disponível – o conjunto de saídas, conjunto de quadrados representado na parte inferior do <i>codelet</i> , não está disponível.	60
Figura 14 – Exemplo de execução utilizando <i>codelets</i> na forma de um dígrafo.	60
Figura 15 – Estruturas usadas no modelo.	62
Figura 16 – Tempo de execução em número de ciclos dos algoritmos implementados variando o tipo de escalonador utilizado durante a execução. A execução foi realizada na máquina real utilizando o conjunto de entradas A . (a) Multiplicação de Matrizes; (b) Produto Interno; (c) Fatoração L.U.; (d) <i>Quick Sort</i>	67

Figura 17 – Tempo de execução dos algoritmos variando o tipo de escalonador. Execução realizada na máquina real utilizando o conjunto de entradas B . O eixo das abscissas representa o número de threads. Já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	69
Figura 18 – Grafo de dependências da implementação da fatoração L.U., com um corte sequencial em k	74
Figura 19 – Representação da execução no modelo desenvolvido com dois processadores – P1 e P2. Nesta representação os <i>codelets</i> em execução são representados pela presença do mesmo em um dos processadores e a disponibilidade de entradas e saídas é representada pelo preenchimento do seu respectivo quadrado.	76
Figura 20 – Tamanho do <i>cache</i> L3 em MB em função do número de <i>threads</i> de cada processador	84
Figura 21 – Tempo de execução dos algoritmos variando o compilador utilizando o π FlowMT e o conjunto de entradas B . (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	88
Figura 22 – Tempo de execução dos algoritmos variando o compilador utilizando OpenMP e o conjunto de entradas B . (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	89
Figura 23 – Tempo de execução dos algoritmos variando o compilador utilizando T.B.B. e o conjunto de entradas B . (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	90

Figura 24 – Tempo de execução dos algoritmos. Execução realizada na máquina real utilizando o conjunto de entradas **B**. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 91

Figura 25 – *Speedup* obtido com a execução dos algoritmos. Execução realizada na máquina real utilizando o conjunto de entradas **B**. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 93

Figura 26 – Desvio padrão do número de *codelets* executados em função do número total de *codelets* executados. Execução realizada na máquina real utilizando o conjunto de entradas **B**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 94

Figura 27 – *Speedup* obtido com a variação do tamanho dos *codelets*. O eixo das ordenadas representa o *speedup* obtido e o eixo das abscissas representa o grau do polinômio integrado. Execução realiza com 8 *threads* utilizando o π FlowMT na máquina real. 95

Figura 28 – *Speedup* obtido com a execução dos algoritmos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*, enquanto que o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 97

Figura 29 – *Speedup* obtido com a execução dos algoritmos. Execução realizada no simulador com o *cache* projetado utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de núcleos simulados. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 98

Figura 30 – Tempo de execução dos algoritmos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 99

Figura 31 – Razão da variação entre o número de ciclos por núcleo simulado e o número total de ciclos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. 101

Figura 32 – *Overhead* obtido com a execução dos algoritmos no simulador. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*. . . 103

Figura 33 – Eficiência do *quick sort* com a execução no ambiente real (entradas **B**) e no simulado (entradas **A**) comparada com a eficiência teórica. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa a eficiência de execução. (a) Máquina Real; (b) Simulador. . . . 104

Figura 34 – Comparação do número de GFlops entre os diversos modelos de execução. Execução realizada utilizando o conjunto de entradas B . O eixo das ordenadas representa o número de GFlops, enquanto o eixo das abcissas representa o número de núcleos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	125
Figura 35 – Comparação do número de GFlops com a variação do compilador. Execução realizada utilizando o conjunto de entradas B . O eixo das ordenadas representa o número de GFlops, enquanto o eixo das abcissas representa o número de núcleos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) <i>Quick Sort</i> ; (k) <i>Quick Sort</i> - 2 Pivôs; (l) <i>Bucket Sort</i>	126
Figura 36 – Divisões geradas pelo algoritmo do <i>quick sort</i>	127

LISTA DE QUADROS

Quadro 1 – Especificações para os processadores simulados.	83
Quadro 2 – Especificações do computador usado.	84
Quadro 3 – Compiladores utilizados e suas respectivas compatibilidades.	85
Quadro 4 – Conjutos de entradas utilizados durante as várias execuções.	121

LISTA DE ALGORITMOS

Algoritmo 1	– Algoritmo para multiplicação de matrizes quadradas.	115
Algoritmo 2	– Algoritmo para a realização do produto interno entre dois vetores. . . .	115
Algoritmo 3	– Algoritmo de fatoração L.U..	116
Algoritmo 4	– Algoritmo recursivo para o cálculo da integral da função $f(x)$	116
Algoritmo 5	– Algoritmo não-recursivo para o cálculo da integral da função $f(x)$. . .	116
Algoritmo 6	– pseudocódigo do algoritmo de Cooley-Tukey para a FFT.	117
Algoritmo 7	– Algoritmo para o cálculo da função $P(l)$ utilizando o método de Monte Carlo.	118
Algoritmo 8	– Implementação com um pivô do algoritmo de ordenação <i>quick sort</i> . . .	118
Algoritmo 9	– Implementação do algoritmo <i>quick sort</i> utilizando dois pivôs.	119
Algoritmo 10	– Algoritmo de ordenação <i>Bucket Sort</i>	120

LISTA DE CÓDIGOS-FONTE

Código-fonte 1	–	Representação de um código na forma de uma sequência de instruções.	39
Código-fonte 2	–	Exemplo de código em linguagem montadora sem otimização. . . .	49
Código-fonte 3	–	Exemplo de código com <i>loop</i> em linguagem montadora.	51
Código-fonte 4	–	Exemplo de função do modelo.	70
Código-fonte 5	–	Trecho do <i>main</i> do programa de fatoração L.U. utilizando π FlowMT.	71
Código-fonte 6	–	Função <i>lufac2</i>	72
Código-fonte 7	–	Função <i>lufac1</i>	73
Código-fonte 8	–	Função <i>lufac_seq</i>	73
Código-fonte 9	–	Integração binária - Recursiva	129
Código-fonte 10	–	Integração binária - Não recursiva	134
Código-fonte 11	–	<i>Bucket sort</i>	138
Código-fonte 12	–	Produto interno	145
Código-fonte 13	–	FFT	148
Código-fonte 14	–	Fatoração L.U.	152
Código-fonte 15	–	Multiplicação de Matrizes	155
Código-fonte 16	–	Monte Carlo	158
Código-fonte 17	–	<i>Quick sort</i>	161
Código-fonte 18	–	<i>Quick sort</i> - 2 pivôs	166

LISTA DE TABELAS

Tabela 1 – Comparação entre o tempo de acesso das múltiplas estruturas para armazenamento. Cada estrutura teve seu valor modificado e acessado 10^{12} vezes.	62
Tabela 2 – Comparação entre os modos de armazenamento das funções.	63
Tabela 3 – Comparação entre a média do tempo de inserção de um elemento nas estruturas de armazenamento. Para o cálculo da média foram realizadas 2^{25} inserções.	64
Tabela 4 – Comparação entre as estruturas de pareamento para as diversas operações utilizadas, a letra O sobrescrita representa a execução do código compilado com otimizações.	70
Tabela 5 – Tempo de execução dos simuladores na realização de uma multiplicação de matrizes de tamanho 64×64 . As simulações foram realizadas com 1 <i>core</i> e 1.024 <i>cores</i>	82
Tabela 6 – Tempo de execução dos algoritmos no simulador utilizando os valores de entrada A	96

LISTA DE ABREVIATURAS E SIGLAS

ALU		<i>arithmetic logic unit</i>
AVX		<i>Advanced Vector Extensions</i>
DDM		<i>data-driven multithreading</i>
FFT		<i>Fast Fourier Transform</i>
FLOPS	...	<i>Floating point Operations Per Second</i>
IPC		<i>Instructions per cycle</i>
KILL		<i>Kill If Less than Linear</i>
MMX		<i>Matrix Math Extensions</i>
MSA		<i>MIPS SIMD Architecture</i>
PC		<i>Program Counter</i>
SIMD		<i>Single Instruction Multiple Data</i>
SMT		<i>simultaneous multithreading</i>
SSE		<i>Streaming SIMD Extensions</i>
VSI		<i>Visual Instruction Set</i>

SUMÁRIO

1	INTRODUÇÃO	33
2	MODELOS DE EXECUÇÃO E PROCESSADORES	37
2.1	Modelos de Execução	37
2.2	Processadores	45
2.3	Modelos de estado de arte	54
3	π FLOWMT	57
3.1	π FlowMT	57
3.2	Estruturas de dados	61
3.3	Métodos de sincronia	65
3.4	Códigos utilizando o modelo	70
3.5	Implementação de um programa	71
3.6	Demonstração de uma execução	75
4	METODOLOGIA DE AVALIAÇÃO	77
4.1	Algoritmos	78
4.2	Ambiente de execução	80
4.3	Métricas	85
5	RESULTADOS	87
5.1	Máquina real	87
5.2	Simulador	95
5.3	Eficiência do <i>Quick Sort</i>	102
6	ANÁLISE E PERSPECTIVAS	105
	REFERÊNCIAS	107
	APÊNDICE A ALGORITMOS	115
	APÊNDICE B ENTRADAS	121
	APÊNDICE C FLOPS	123

APÊNDICE D EFICIÊNCIA DO *QUICK SORT* 127

ANEXO A ALGORITMOS – π FLOWMT 129

INTRODUÇÃO

O desenvolvimento da computação pode ser descrito como um processo cíclico: novos algoritmos e técnicas são desenvolvidas, mas estas técnicas necessitam de um *hardware* com maior capacidade de processamento, logo um novo *hardware* é desenvolvido para executar estes novos algoritmos, mas com isso é possível desenvolver outros algoritmos que vão requerer *hardwares* ainda mais potentes, e assim o processo é repetido.

Um exemplo do crescimento da capacidade de processamento pode ser visto na figura 1.

O TOP500 (1) é uma lista de supercomputadores criada em 1993 e atualizada duas vezes por ano que tem como objetivo analisar as tendências presentes na computação de alto desempenho através da execução do *benchmark* LINPACK (2) nas máquinas que pleiteiam um lugar na lista.

Embora a lista do TOP500 considere apenas os supercomputadores, a tendência de crescimento do computadores *normais* também existe e é possível graças à combinação de uma série de fatores, que incluem desde melhorias tecnológicas, como a utilização de um processo litográfico mais otimizado, como também da utilização de novas técnicas de aprimoramento de *hardware*.

A mais recente e relevante técnica para o aumento da capacidade de processamento é a utilização de processadores que possuam mais de um núcleo de processamento, isto é, a criação do que é conhecido como processador *multi-core*.

Esta técnica começou a ser utilizada amplamente nos processadores comerciais no começo deste milênio, com o lançamento da linha de processadores *Core 2 Duo* da Intel e desde então o número de processadores *multi-core* disponíveis no mercado vem crescendo, não só em modelos diferentes como também na quantidade de núcleos presentes — já é possível adquirir um processador com 64 núcleos.

Figura 1 – Evolução da *performance* em GFLOPS dos supercomputadores avaliados pelo TOP500.

Fonte: Elaborada pelo autor.

O número de núcleos presentes nos processadores deve continuar crescendo — já existem rumores de que um processador de 128 núcleos está em desenvolvimento — possivelmente atingindo o patamar conhecido como *many-core* onde o número de núcleos é da ordem de centenas a milhares.

O aumento da capacidade de processamento que surge com o aumento do número de núcleos por processador vem acompanhado do aumento da dificuldade de criar um programa que extraia toda esta capacidade. Esses processadores necessitam que o programador sincronize os acessos à memória para garantir a coerência dos dados durante a execução e realizar esta tarefa de maneira eficiente é difícil, o que impede que o processador seja utilizado em sua plena capacidade.

Diversos modelos de execução tentam facilitar esta tarefa, entre eles um que se destaca é o modelo dirigido pelos dados.

O modelo dirigido pelos dados foi importante tópico de pesquisa durante as décadas de 70 a 90 e atualmente voltou a ser um tópico de interesse, como visto na figura 2.

Com isso as motivações para o desenvolvimento desta pesquisa ficam claras:

- Verificar a viabilidade da utilização do fluxo de dados para extrair a capacidade de processamento dos processadores com múltiplos núcleos;

Figura 2 – Número de publicações sobre fluxo de dados nos repositórios da ACM e IEEE. Atualizado em fevereiro de 2020.

Fonte: Elaborada pelo autor.

- Desenvolver e implementar um modelo de execução de propósito geral para os processadores *multi-core* atuais e futuros;
- Comparar o modelo desenvolvido com modelos de estado de arte.

Nos modelos de execução dirigidos pelos dados um programa é representado na forma de um grafo orientado e esta representação é inerentemente paralela, isto é, os elementos que são representados na mesma profundidade deste grafo podem ser executados paralelamente sem a necessidade da realização de operações de sincronização.

Estas características do fluxo de dados tornam o mesmo uma excelente escolha para ser utilizado em processadores *many-core*, já que realizar a sincronização entre todos os núcleos de um destes processador é uma tarefa árdua.

A pesquisa tem como objetivo mostrar a viabilidade da utilização do modelo de fluxo de dados neste contexto através da implementação de uma biblioteca escrita em c++.

Para a análise do modelo são implementados uma série de algoritmos já conhecidos e estudados no modelo e em modelos de estado de arte — OpenMP, *Threading Building Blocks* e OmpSs — e os mesmos são executados em um ambiente real e em um ambiente simulado, permitindo uma análise ampla do momento atual dos processadores como também do futuro com a execução simulada de um processador *many-core*.

MODELOS DE EXECUÇÃO E PROCESSADORES

O projeto desenvolvido é motivado pela união de dois tópicos de alta importância para a computação: modelos de execução e o desenvolvimento de processadores.

Para entender melhor como a união destes dois tópicos impacta a pesquisa é necessário contextualizar ambos os tópicos e exibir o cenário onde a união dos tópicos foi inevitável.

2.1 Modelos de Execução

Um modelo de execução determina os passos que levam à execução de uma operação qualquer e pode ser aplicado em diversos níveis de abstração: como uma instrução é executada em um processador, como as instruções geradas por uma linguagem de programação são organizadas e também como múltiplas instruções são divididas para a execução em um ambiente com múltiplas unidades de processamento.

Dentre os modelos de execução que atuam no nível mais baixo de abstração, destacam-se, no contexto desta pesquisa, o modelo de von Neumann e o modelo dirigido pelos dados, que serão discutidos a seguir.

2.1.1 Modelo de von Neumann

O modelo de von Neumann é facilmente identificado pela presença de duas características: a presença de uma única memória que armazena tanto os dados como também as instruções e a utilização de um registrador especial, chamado de *Program Counter* (PC), que aponta para a próxima instrução que será executada.

Note que o modelo é simples o que permite não só a fácil implementação em hardware como também a compilação de linguagens de alto nível. Esse fator permitiu a popularização do

modelo de tal maneira que o mesmo é utilizado até os dias atuais — os processadores atuais implementam o modelo de von Neumann com uma série de extensões, como será visto na seção sobre processadores (seção 2.2).

Embora a simplicidade do modelo tenha permitido a popularização do mesmo, ela também é a responsável pelas importantes limitações presentes no contexto atual dos processadores de estado de arte (*multi-cores*).

A primeira e mais conhecida — o gargalo de von Neumann ou *memory wall* — é causada por uma combinação de fatores: no modelo de von Neumann, devido à presença de uma única memória para os dados e instruções, é impossível acessar um dado e buscar uma instrução ao mesmo tempo; junte isto ao fato de que as frequências de *clock* do processador terem crescido de maneira mais intensa em relação a frequência de *clock* da memória. Esta combinação impede que o processador opere de maneira eficiente, já que a memória não consegue suprir a demanda do processador.

A segunda decorre da natureza sequencial do modelo, que existe devido à presença do registrador PC. A presença deste registrador dificulta a generalização do modelo para um ambiente com mais de um processador: como seria possível determinar quais instruções deveriam ser executadas em paralelo se o PC aponta para a próxima instrução na ordem lexicográfica em que foram escritas? Se cada uma das unidades de processamento possuir o próprio PC como determinar qual a ordem de execução das mesmas para garantir a coerência dos dados?

Observe ainda que o gargalo de von Neumann teria um impacto ainda maior: se a memória não é capaz de manter um processador alimentado, como alimentaria múltiplos?

A resposta para estas perguntas é complexa e motivo de muita pesquisa. Já que as extensões realizadas no modelo são insuficientes — vide subseção 2.2.2 — é necessário a utilização de outros modelos de execução em um nível de abstração mais alto para tentar mascarar e evitar os problemas apresentados.

2.1.2 Dataflow

O modelo dirigido pelos dados, *dataflow* em inglês, é um modelo que pode ser útil para resolver os problemas citados na subseção 2.1.1. Para entender como o modelo seria capaz disso é necessário entender a estrutura do modelo.

Nesse modelo o paradigma de execução é diferente: as instruções são executadas de acordo com a disponibilidade dos seus operandos, de modo que a execução de um conjunto de instruções não é necessariamente feita na ordem em que elas foram escritas. Essa característica faz com que o modelo não necessite de um controlado de fluxo, representado no modelo de von Neumann pelo registrador PC. Também, múltiplas instruções podem ser executadas concorrentemente.

Devido a esta ausência de controle de fluxo, a representação de um código na forma de uma sequência de instruções torna-se obsoleta. Nesses modelos, a representação natural passa a ser feita através de um dígrafo, onde os nós representam as instruções, como visto na representação do código 1 feita na figura 3.

Código-fonte 1: Representação de um código na forma de uma sequência de instruções.

```
1 x=a*b;  
2 y=4*c;  
3 r=(x+y)*(x-y)/c;
```

Figura 3 – Representação de um código na forma de dígrafo.

Fonte: Elaborada pelo autor.

A representação na forma de dígrafo também permite uma maior noção do paralelismo disponível, isto é, todos os nós do dígrafo que estão na mesma profundidade podem ser executados em paralelo, já que não existe dependência entre eles.

O modelo dirigido pelos dados foi importante tópico de pesquisas durante as décadas de 70 e 80, sendo esta a primeira onda de desenvolvimento que utilizou o modelo.

Nesta era a pesquisa foi pautada no desenvolvimento de protótipos que implementavam o modelo dirigido pelos dados a nível de *hardware*, e os protótipos desenvolvidos, em geral, eram divididos entre protótipos estáticos e protótipos dinâmicos. A diferença entre um protótipo estático e um dinâmico está em como os protótipos lidam com a execução de múltiplas instâncias de uma mesma rotina: enquanto isso não é possível nos modelos estáticos isto é implementado através de um rótulo nos modelos dinâmicos.

A segunda fase de desenvolvimento dos modelos dirigidos pelos dados é contemporânea, e assim como a pesquisa aqui desenvolvida, é voltada para a solução dos problemas apresentados na seção 2.2.

Nesta fase é possível dividir o desenvolvimento em três grupos:

- Extensões de hardware: neste caso, o fluxo de dados é implementado diretamente em uma hardware específico;
- Linguagens próprias: neste caso, o fluxo de dados ocorre durante a organização das instruções da linguagem para a execução;
- Bibliotecas: aqui o fluxo de dados é implementado em uma linguagem de propósito geral através de uma biblioteca;

A seguir serão discutidos em mais detalhes alguns dos projetos que implementam o modelo dirigido pelos dados.

2.1.2.1 MIT Static Machine Dataflow

Este protótipo estático foi desenvolvido no MIT(3) no começo da década de 80. Utilizando 8 processadores do tipo *bit slice* da AMD® conectados via uma rede composta por roteadores 2x2. Os grafos utilizados neste protótipo foram compilados na linguagem VAL.(4) A máquina foi um sucesso do ponto de vista operacional; porém do ponto de vista prático, a máquina não era capaz de executar programas complexos. Além disso os elementos processantes eram significativamente mais lentos que os roteadores.

2.1.2.2 Manchester Dataflow Machine

A máquina de Manchester(5) era dinâmica e implementada na forma de um único anel com uma fila de *tokens*, uma unidade de pareamento, uma unidade de instruções e um conjunto de *arithmetic logic unit* (ALU). As ALUs eram relativamente lentas, mas mesmo assim o protótipo demonstrou um desempenho razoável (1.2MIPS).

Este protótipo permitiu que diversos avanços em relação ao fluxo de dados fossem desenvolvidos e diversas extensões do modelo foram estudadas — por exemplo o uso de múltiplos anéis.

2.1.2.3 TRIPS

Foi desenvolvido na Universidade do Texas em parceria com a IBM, Intel e Sun Microsystems em 2005. TRIPS(6, 7) faz uso de um novo conjunto de instruções chamado *Explicit Data Graph Execution*(EDGE), que possui duas características próprias: Execução bloco-atômica e comunicação direta entre instruções de dentro do mesmo bloco.

Os blocos são constituídos de instruções *dataflow*, que nesse caso são limitados a 128 instruções por bloco, que, devido à execução do tipo bloco-atômica, são tratados como uma única entidade na busca e execução.

A comunicação direta entre instruções dentro do mesmo bloco é feita para permitir que resultados de um produtor sejam enviados para os consumidores de uma maneira *dataflow*. A comunicação entre blocos é feita utilizando os registradores e a memória.

Cada *chip* TRIPS contém dois processadores e um sistema de memória secundário e cada processador possui cinco tipos de unidades: uma unidade global de controle, 16 unidades de execução, quatro registradores, quatro *caches* de data e quatro *caches* de instrução. O protótipo desenvolvido pode executar até oito blocos de 128 instruções ao mesmo tempo, permitindo uma janela máxima de 1024 instruções.

2.1.2.4 MAXELER

MAXELER Technologies é uma empresa especializada no desenvolvimento de uma classe de computadores de propósito específico que são re-programáveis e utilizam os conceitos do fluxo de dados. O sistema MAXELER contém um processador x86 conectado, através de um *bus PCI express* de segunda geração, ao conjunto de unidades *dataflow* (MAXELER *Dataflow Engines* (DFEs)). Esse conjunto de unidades atua como um coprocessador inteiramente *dataflow* desenvolvido em uma FPGA. Como esse coprocessador é inteiramente *dataflow*, técnicas como previsão de desvio e execução fora de ordem não são necessárias. (8, 9)

Na execução, os dados são enviados da memória para os DFEs, onde as operações são realizadas. Enquanto existirem instruções para serem executadas os dados fluem, no estilo *dataflow*, de uma unidade para outra sem voltar para a memória principal até o final da execução.

Para transformar o código puramente sequencial em um código *dataflow* compatível com o sistema, duas etapas devem ser realizadas: a primeira é a escrita em MaxJ, uma extensão da linguagem Java utilizada pelo MaxCompiler, do acelerador, que define as operações em *dataflow* que serão utilizadas e mapeia as variáveis do processador no DFE. A segunda etapa

requer que a parte do código que será executada em *dataflow* seja reescrita utilizando o acelerador gerado pelo código em MaxJ.

2.1.2.5 TERAFLUX

TERAFLUX (10) é um projeto de quatro anos iniciado em 2010 financiado pela União Europeia em parceria com empresas e universidades de diversos países.

O sistema foi desenvolvido para permitir a execução tanto de *threads* no paradigma *controlflow* como de *threads* no paradigma *dataflow*. (11) Isso é feito para permitir que os atuais programas ainda sejam utilizados e para ajudar na migração de paradigmas.

Para permitir a execução das *threads dataflow* foi desenvolvida uma extensão para o conjunto de instruções x86_64. Essa extensão tem três finalidades: permitir a execução assíncrona das *threads dataflow*; permitir que a execução de uma *thread dataflow* seja decidida pelo DTS (*Distributed Thread Scheduler*), um elemento externo ao processador e desenvolvido para essa finalidade; diferenciar entre os quatro tipos de memória existentes e usadas na comunicação entre *threads*: 1-para-1 (*thread local storage*), N-para-1 (*frame memory*), 1-para-N (*owner writable memory*) e N-para-N (*transactional memory*).

2.1.2.6 WaveScalar

WaveScalar (12, 13) é um modelo de fluxo de dados dinâmico que utiliza fichas rotuladas. A grande diferença entre essa abordagem e as demais abordagens a fluxo de dados desenvolvidas é a capacidade de lidar eficientemente com a forma tradicional e ordenada, utilizada no modelo de von Neumann, de estruturação da memória; com isso é possível executar códigos escritos em linguagens como C, C++, Fortran ou Java. O rótulo, nesse caso chamado número-wave, especifica diferentes instâncias da mesma instrução.

O WaveCache (13), um sistema de *cache* de instruções distribuído, carrega as instruções da memória e as designa para os elementos de processamento, que permanecem no *cache*; isso permite uma otimização dinâmica da designação de instruções de modo que uma instrução seja designada a um elemento de processamento próxima àqueles que dependem de sua saída. Essa otimização gera um localidade do tipo *dataflow*, e por ser uma localidade seus feitos são análogos aos efeitos da localidade espacial e temporal.(14)

2.1.2.7 Data-Driven Workstation Network

D²Now (15) é uma máquina paralela baseada no modelo de execução *data-driven multithreading* (DDM). Esse modelo separa a parte de sincronização da parte de execução de um programa, permitindo que ele seja executado de maneira assíncrona. Isso aumenta a tolerância à latência, pois permite que o processador realize cálculos enquanto uma operação

que possui uma latência grande é realizada. Nesse modelo de execução o agendamento das *threads* é feito em tempo de execução de acordo com a disponibilidade dos dados.

A máquina é construída utilizando processadores Intel Pentium junto com uma unidade de sincronização de *thread* (TSU). A comunicação entre o processador e a TSU ocorre através de duas filas: a fila de prontos (RQ) e a fila de informação (AQ).⁽¹⁵⁾ Essas filas são mapeadas na memória e portanto não é necessário modificar ou adicionar novas instruções ao processador. A RQ possui os endereços das *threads* que já estão prontas para a execução, enquanto a AQ possui informações de identificação e *status* das *threads* já executadas.

2.1.2.8 Tarragon

Tarragon ⁽¹⁶⁾ é um modelo de execução, baseado nos princípios do fluxo de dados, em camadas: a primeira camada, ou camada de tarefa (*task level*), que representa as instruções dentro de uma tarefa, a segunda camada, ou camada de grafo (*graph level*), responsável por controlar a execução das tarefas de acordo com os princípios do fluxo de dados e por último a terceira camada, ou camada de controle (*control level*), que é responsável pela criação e execução dos grafos.

Diferente dos demais modelos, onde o paralelismo é obtido implicitamente, o paralelismo deve ser explicitado pelo programador, assim como a dependência de dados entre as tarefas. No entanto toda a comunicação e sincronização é executada de maneira transparente ao programador.

2.1.2.9 TIDeflow

TIDeFlow ⁽¹⁷⁾ foi desenvolvido para ser capaz de lidar eficientemente com *loops* em paralelo, dependência de dados entre os *loops* paralelos e permitir a modularidade dos programas.

Os programas em TIDeFlow são expressos utilizando grafos com pesos tanto nos nós, chamados de Atores nesse modelo, e nos arcos. Os atores representam os *loops* paralelos e possuem informações do número de iterações do *loop* e a função que é executada dentro do *loop*. Além disso possuem uma informação do estado do ator em relação à sua execução: disponíveis para a execução, não disponíveis para execução, executando e executado, e um contador que é incrementado cada vez que um ator é executado.

Quando um Ator termina sua execução ele precisa enviar para os outros Atores, que dependem de sua saída, um sinal indicando que foi executado e isso é feito através de fichas rotuladas, onde o rótulo é o contador de execução do ator, e os dados são transmitidos de um Ator para o outro através da memória.

2.1.2.10 GoDEL

GoDEL (18) é um modelo de execução a fluxo de dados baseado em redes de propagação. Redes de propagação é um modelo de computação de propósito geral para execução paralela e distribuída. Essas redes são constituídas por células, que armazenam informações sobre os dados, e propagadores, que são unidades assíncronas e autônomas que realizam as operações. Os dados que são processados por um propagador podem ser provenientes de mais de uma célula e os resultados são escritos em uma única célula. A execução de um propagador é habilitada quando as células de entrada do propagador o informa que um novo dado está disponível.

GoDEL é implementado na forma de uma linguagem, que incorpora qualquer linguagem do *host*, permitindo assim o uso de códigos desenvolvidos nessas linguagens nos propagadores GoDEL.

2.1.2.11 Derflow

Derflow (19) é uma extensão que implementa a semântica *dataflow* ao modelo de execução Erlang. Erlang é um modelo de execução de passagem de mensagens que permite que processos concorrentes enviem mensagens de maneira assíncrona. O modelo sem a extensão é altamente não determinístico, pois cada processo pode receber dados de todos os outros processos de maneira assíncrona, o que gera um número gigantesco de possíveis ordens de execução, tornando a tarefa de validar a execução muito mais difícil.

Derflow é uma extensão do modelo Erlang que tem como objetivo eliminar o não-determinismo ao implementar o paradigma *dataflow*. É implementado como uma biblioteca da linguagem do modelo Erlang.

Os modelos apresentados anteriormente são destaques de duas eras que implementam o fluxo de dados de maneira distinta: enquanto os modelos desenvolvidos nas décadas de 70 a 90 fazem usos de protótipos que eram inacessíveis ao usuário *comum*, a segunda é mais acessível com a implementação de bibliotecas e linguagens próprias.

Uma segunda diferença importante entre as eras é o grau de paralelismo utilizado para o fluxo de dados, enquanto na primeira o fluxo de dados era implementado à nível de instrução, também conhecido como granularidade fina, a segunda era o implementa com granularidade grossa, isto é, com a utilização de um conjunto de instruções que é tratado como um único bloco.

Ainda é possível classificar a segunda era de modelos em duas áreas:

- **Implementação em *hardware*:** estes modelos utilizam uma extensão de *hardware* para a execução do fluxo de dados. Dos exemplos citados encontram-se nesta categoria: TRIPS, MAXELER, TERAFLUX e WaveScalar;

- **Linguagens próprias:** outros modelos fazem uso de uma linguagem própria, que implementa o fluxo de dados diretamente, caso do *D²Now*, Tarragon, TIDeFlow, Derflow e GoDEL.

Outros modelos de execução serão discutidos na seção 2.3.

2.2 Processadores

O desenvolvimento de novos processadores é claramente dividido em duas eras: a primeira buscava um crescimento vertical, isto é, um processador único e extremamente rápido, enquanto a segunda era, que é contemporânea a este trabalho, busca um desenvolvimento mais horizontal com a criação de um processador com múltiplas unidades de processamento.

Essa mudança foi necessária, pois os tradicionais esforços que eram aplicados no desenvolvimento de novos processadores atingiram um patamar onde a relação custo e ganho passou a ser muito elevado, como será visto a seguir.

2.2.1 Aumento da frequência de clock

O método mais óbvio para aumentar o desempenho de um processador é aumentar a frequência de *clock*; isto permite a realização de um maior número de operações no mesmo intervalo de tempo.

Durante a primeira era de desenvolvimento dos processadores, o aumento da frequência foi um elemento crucial, como visto na figura 4.

Observe que a partir de 2004, o valor da frequência parou de aumentar, e este é um dos fatores que definem a mudança entre as eras de desenvolvimento.

Esta limitação para o valor da frequência é facilmente explicada: o aumento da frequência é fortemente ligado ao aumento da potência dissipada, como visto na figura 5. Note que um processador que dissipa uma grande quantidade de energia é problemático, pois este não só consome mais energia, aumentando o custo por hora utilizada, como também é muito mais difícil de mantê-lo controlado do ponto de vista térmico, sendo necessário a utilização de métodos não triviais de resfriamento, mas nem mesmo esses métodos são capazes de passar o que é conhecido como *power wall*.⁽²⁰⁾

Note que o constante aprimoramento nas técnicas de litografia, isto é, a diminuição do tamanho dos transístores, foi um importante fator para a continuidade do aumento da frequência¹ — e de outras melhorias de *hardware*, como visto na subseção 2.2.2 —, mas mesmo se fosse possível criar um processador sem os problemas relacionados à dissipação de energia

¹ Transístores menores utilizam uma tensão menor, então mesmo com o aumento da densidade de transístores a potência dissipada diminui

Figura 4 – Evolução temporal da frequência de *clock*. O eixo das ordenadas exibe a frequência em GHz.

Fonte: Elaborada pelo autor.

ainda não seria possível aumentar a frequência de maneira indiscriminada, já que limites mais extremos ainda estariam presentes, como a velocidade da luz e o princípio de incerteza.

2.2.2 Otimizações de hardware

O outro método para tornar um processador mais rápido é aumentar o número de instruções executadas em um ciclo de *clock* — *Instructions per cycle* (IPC).

Os métodos que aumentam o número de IPC já foram citados anteriormente neste capítulo: eles são as extensões ao modelo de von Neumann. Existem diversas técnicas que permitem um aumento no número de IPC. As principais serão discutidas a seguir.

2.2.2.1 Cache

O impacto do gargalo de von Neumann, criado pelo aumento da diferença entre os *clocks* da memória e do processador, pode ser diminuído se a memória for mais rápida, e é essa a motivação para a criação da memória *cache*. O *cache* é uma pequena memória extremamente rápida que intermedeia os acessos do processador à memória: o valor do endereço de interesse da memória é copiado para o *cache* e mantido lá enquanto for necessário, o que permite que múltiplos acessos a estes dados sejam realizados de maneira mais rápida.

Figura 5 – Relação entre potência dissipada e frequência de *clock*. O eixo das ordenadas exibe a potência dissipada em Watts, enquanto o eixo das abscissas exibe a frequência em GHz.

Fonte: Elaborada pelo autor.

Embora pareça uma tarefa simples, ao adicionar essa memória — chamada *cache* — uma série de medidas de *segurança* devem ser feitas para garantir a coerência de dados e o ganho de desempenho (21) — caso a política de substituição dos valores armazenado pelo *cache* seja ineficiente o número de *cache miss* aumenta e o desempenho diminui, isto é, as informações presentes no *cache* não são as informações necessárias e, portanto, é necessário realizar um novo acesso à memória para buscar os dados corretos.

A evolução natural desta técnica é a utilização de *caches* maiores e mais níveis (22) — isto é, uma outra camada de *cache* intermediando o acesso entre o primeiro nível de *cache* e a memória. A figura 6 mostra a evolução do tamanho do *cache* L3 de acordo com os anos.

Os ganhos obtidos com a implementação desta memória variam de acordo com o programa executado — programas podem ser escritos de maneira *cache-friendly* onde os acessos à memória são feitos buscando minimizar o número de *cache miss* — (23) e com a razão entre o tempo de acesso do *cache* e da memória principal.

2.2.2.2 Pipeline

Esta é uma técnica que implementa o paralelismo a nível de instrução em um único processador. A técnica é implementada com o objetivo de manter todas as partes do processador

Figura 6 – Evolução temporal do tamanho do *cache*.

Fonte: Elaborada pelo autor.

ocupadas através da divisão da execução em partes independentes, como visto nas figuras 7 e 8.

Figura 7 – Exemplo de execução em um processador sem *pipeline*.

Fonte: Elaborada pelo autor.

É possível observar que a presença de um *pipeline* aumenta o IPC do processador(24), mas os ganhos nem sempre são tão efetivos, já que durante a execução de um programa, acessos à memória são realizados e devido aos acessos à memória serem notoriamente lentos, o *pipeline* é *travado* até que a unidade de acesso a memória do processador finalize a operação, diminuindo o ganho do IPC.

A dificuldade de manter o *pipeline* ocupado é a principal razão pela qual não são criados

	1	2	3	4	5	6	7	8
1	F	D	E	W				
2		F	D	E	W			
3			F	D	E	W		
4				F	D	E	W	
5					F	D	E	W

Figura 8 – Exemplo de execução em um processador com *pipeline*.

Fonte: Elaborada pelo autor.

pipelines arbitrariamente longos, já que quanto maior o *pipeline*, maior o custo envolvido e menor o ganho obtido.

Estudos demonstram que o tamanho ideal para o *pipeline* é de 15 unidades (25), já que a partir deste número o ganho de desempenho é irrisório ou até mesmo negativo.

2.2.2.3 Superscalar

Se na execução com *pipeline* a ideia é manter todas as unidades do processador ocupadas, na execução *superscalar* a ideia é duplicar certas unidades, permitindo que execuções independentes sejam executadas em paralelo, como visto na figura 9.

O exemplo apresentado é chamado de 2-way *superscalar*, já que as unidades foram apenas duplicadas. Existem exemplos onde as unidades foram quadruplicadas, tais como no MIPS R10k. (26)

A criação de um processador com um número alto de unidades replicadas só seria eficiente em casos extremos, já que a execução das instruções só acontece em paralelo caso não haja dependência entre as mesmas.

2.2.2.4 Execução Out-of-order

A motivação por trás desta otimização é diluir os custos de acesso à memória. Considere a execução do código 2 em pseudo-linguagem em um processador com um *pipeline* de 4 estágios (ver figura 10).

Figura 9 – Exemplo de execução em um processador com *pipeline*.

Fonte: Elaborada pelo autor.

Código-fonte 2: Exemplo de código em linguagem montadora sem otimização.

```

1 mov R1, #02h ; R1 := 2
2 mov R2, #03h ; R2 := 3
3 mov R3, @2000h ; R3 := Conteúdo do endereço 2000h da memória
4 add R1, #01h ; R1 := R1+01h
5 add R2, R1 ; R2 := R2+R1
6 sub R2, R3 ; R2 := R2-R3
7 ...

```

Figura 10 – Exemplo de execução em um processador sem execução *out-of-order*.

Fonte: Elaborada pelo autor.

Observe que se o código fosse reescrito colocando a instrução presente na linha #3 após as instruções das linhas #4 e #5 o código seria mais eficiente, já que manteria o *pipeline* ocupado por uma parte maior do tempo, como visto na figura 11.

Figura 11 – Exemplo de execução em um processador com execução *out-of-order*.

Fonte: Elaborada pelo autor.

A otimização em *hardware* permite que o processador analise o código e verifique se é possível modificar a ordem das instruções e com isso dilua os custos de acesso à memória. (27)

2.2.2.5 Preditor de Desvios

O preditor de desvios, em inglês *branch predictor*, melhora o desempenho do processador, pois permite que o *pipeline* do processador fique ocupado por mais tempo e diminui os acessos à memória. Considere o código 3.

Código-fonte 3: Exemplo de código com *loop* em linguagem montadora.

```

1  mov R1, #100d ; R1 := 100
2  mov R2, #00h ; R2 := 00h
3  j1: sub R1, #01h ; R1 := R1-01h
4  add R2, #02h ; R2 := R2+02h
5  cnj R1, #00h, j1 ; Compara R1 com #00h e se não for
    igual volta ao j1
6  mul R2, #02h ; R2 := R2*02h
7  ...

```

Diferentemente dos códigos anteriores este possui um *loop*. Perceba que durante boa parte da execução, a instrução de desvio condicional da linha 5 é verdadeira, ou seja, se o

processador não esperar o resultado de todo o *loop* e começar a buscar as instruções internas — linhas 3 a 4 —, predizendo os resultado do desvio condicional, o processador seria capaz de manter o *pipeline* ocupado por uma quantidade maior de tempo, aumentando o *throughput* de instruções.

Quando o processador errar — isto é, na centésima iteração do *loop* — basta descartar os resultados parciais e buscar as instruções que dão continuidade ao código. (28)

2.2.2.6 Renomeio de registradores

O número de registradores presente em um processador é limitado e muitas vezes pequeno. Essa limitação muitas vezes força o uso de um mesmo registrador em lugares distintos do código, o que pode levar a utilização do mesmo registrador em partes distintas e independentes do código, que por usarem o mesmo registrador, não podem ser executadas juntas no *pipeline*. A possibilidade do processador renomear os registradores permite que o mesmo identifique e renomeie os registradores, o que permite que o processador mantenha o *pipeline* ocupado por mais tempo.

2.2.2.7 Operações vetoriais

Processadores vetoriais, que na taxonomia de Flynn são conhecidos como *Single Instruction Multiple Data* (SIMD) (29), são processadores que executam uma instrução em mais de um dado ao mesmo tempo. Foram importantes fontes de alto desempenho entre as décadas de 70 e 90, principalmente, com os supercomputadores Cray. (30)

Todavia, com o surgimento e rápido crescimento dos microprocessadores, o seu custo-benefício deixou de ser interessante e, por consequência, sua presença foi aos poucos sendo extinguida (31), mas, como suas características permitiam um desempenho bom, não foram completamente esquecidos.

Na metade da década de 90 as grandes empresas desenvolvedoras de processadores incluíram extensões aos seus conjuntos de instruções que implementavam operações vetoriais aos processadores tradicionais da época — alguns exemplos de extensões vetoriais são *Matrix Math Extensions* (MMX)(1997), *Streaming SIMD Extensions* (SSE)(1999) e *Advanced Vector Extensions* (AVX)(2011) da Intel, *3DNow!*(1998) da AMD, *Visual Instruction Set* (VSI)(1994) da Sparc, *Altivec*(1996) do PowerPC e *MIPS SIMD Architecture* (MSA) da MIPS.

2.2.3 Multithread

As otimizações exibidas nas subseções 2.2.2.1-2.2.2.7 garantem um aumento significativo no número de instruções por ciclos (32), porém elas rapidamente são saturadas e o seu ganho de desempenho, caso exista, não é o suficiente para cobrir os *custos* dos transístores utilizados (33) — no desenvolvimento de processadores há uma *regra* chamada *Kill If Less than Linear* (KILL)

que, em linhas gerais, diz que caso uma otimização de *hardware* não gere um aumento linear de desempenho em relação ao número de transístores utilizados ela não deve ser aplicada. Essa regra é parte do que limita a profundidade dos *pipelines* exibidos na subseção 2.2.2.2, quão especulativa a execução fora de ordem da subseção 2.2.2.4 pode ser e quão precisa a predição de desvio da subseção 2.2.2.5.

Logo, resta determinar o que fazer com os transístores extras que continuam a aparecer. Uma alternativa que faz bom uso desses transístores é continuar com o aumento do *cache* — como visto na subsubseção 2.2.2.1 —, mas mesmo essa alternativa é limitada, já que é difícil manter *caches* tão grandes com a mesma velocidade de acesso. (34)

A alternativa encontrada foi permitir que os processadores executem múltiplas *threads* ao mesmo tempo.

Um processador *multithreaded* é um processador que é capaz de executar instruções de diferentes *threads* em um único *pipeline* (35) — isto é, o processador deve possuir mais de um PC (*program counter*) para percorrer as instruções de cada *thread*, um mecanismo interno para distinguir as instruções de *threads* diferentes e um mecanismo que alterne entre as *threads* — o mecanismo deve ser suficientemente rápido para impedir que o *overhead* causado pela mudança de *thread* seja sentido pelo usuário.

Esta tecnologia é implementada nos processadores atuais com o nome de *hyperthreading* nos processadores Intel e *simultaneous multithreading* (SMT) nos processadores das demais empresas. Embora a maioria dos processadores *multithread* sejam do tipo 2–way SMT, isto é, podem executar duas *threads* em paralelo, existem processadores que conseguem executar até 8 *threads* em paralelo. (36)

Embora o processador seja capaz de executar mais de uma *thread*, o desempenho com a execução de múltiplas *threads* não é perfeitamente escalável: o desempenho de um processador 2–way SMT não é duas vezes o desempenho de um processador não SMT. (37)

2.2.4 Multi-core

Como visto nas seções anteriores, a criação de um único processador extremamente rápido não era sustentável e os esforços aplicados não eram capazes de manter a demanda por maiores capacidades de processamento.

Foi necessário então implementar o que é conhecido como multiprocessador, ou *multi-core*, que é um processador que possui múltiplos processadores dentro de um mesmo *chip* — em geral os múltiplos processadores de um multiprocessador dividem entre si algumas das unidades, de modo que um multiprocessador não contém todas as unidades multiplicadas.

Os processadores *multi-core* foram rapidamente difundidos no mercado, como visto na figura 12, e o número de núcleos presentes deve continuar crescendo, até atingir o patamar

chamado como *many-core*, onde o número de núcleos de processamento é da ordem de centenas a milhares. (38)

Figura 12 – Evolução temporal do número de *threads* por processador.

Fonte: Elaborada pelo autor.

Embora a capacidade de processamento bruta destes processadores seja elevada, a tarefa de extraí-la é árdua: estes processadores não dividem a execução de um programa entre os seus múltiplos núcleos de maneira automática, deixando esta divisão para o programador e requer uma alta perícia, pois, para garantir a coerência dos dados, é necessário que os múltiplos núcleos acessem a memória de maneira sincronizada. Uma implementação eficiente, para a sincronização dos dados é difícil, já que não existe intercomunicação direta entre os núcleos, forçando, novamente, o uso da memória.

2.3 Modelos de estado de arte

Para facilitar a tarefa do programador é necessário a utilização de um modelo de execução de mais alto nível. Este modelo organizará o código para a execução entre os múltiplos núcleos, evitando que o programador explicita todo sincronismo necessário, o que não só facilita o processo de escrita do código, como também, em geral, permite a criação de um código mais eficiente. A seguir serão apresentados modelos atuais.

2.3.1 OpenMP

OpenMP é um conjunto de diretivas paralelas desenvolvido para c, c++ e fortran que funciona através da utilização de diretrizes para o compilador.

As regiões do código que estão dentro de uma destas diretrizes são paralelizadas automaticamente, de modo que o programador não precisa se preocupar com a criação e sincronização das *threads*.

2.3.2 Threading Building Blocks

Nesse modelo o paralelismo é expresso através de tarefas (39). Essas tarefas são regiões especiais do código que realizam uma ação específica quando são executadas pela *TBB runtime library*. A escolha de expressar o paralelismo através de tarefas deve-se ao fato de as tarefas poderem ser criadas e destruídas eficientemente e, como são designadas dinamicamente, permite um maior balanço de carga.

A designação de tarefas é feita de modo que cada *thread* possua uma lista de tarefas para a execução. Porém, para lidar com o problema de balanceamento de carga, uma *thread* sem tarefas disponíveis pode *roubar* tarefa das outras *threads*.

Este modelo também conta com uma ferramenta que permite a criação de grafos, através da ferramenta *Flow Graph*, neste caso as tarefas são representadas por nós de modo a permitir que tarefas independentes sejam executadas em paralelo.

2.3.3 OmpSs

Trata-se de uma das implementações para o modelo de execução desenvolvido pelo *Barcelona Supercomputing Center* chamado StarSs (40) com o foco em permitir e facilitar a computação heterogênea².

O modelo aborda o paralelismo através de *tasks*, ou tarefas, que são executadas assincronamente de acordo com a disponibilidade dos dados de entrada. Para lidar com a dependência de dados entre as diferentes tarefas o modelo adiciona à linguagem, diretivas para determinar as entradas e saídas da tarefa. Além disso, também devido à inclusão de diretivas, o modelo é nativamente compatível com a computação heterogênea.

Diversas implementações foram desenvolvidas para o modelo StarSS:

- **CellSs**, desenvolvida especificamente para o processador Cell/BE;
- **SMPSs**, a implementação principal, desenvolvida a partir da CellSs;

² Integração de elementos processantes de diferentes arquiteturas para melhorar o desempenho, onde, na maioria dos casos, o elemento adicionado é altamente eficiente em uma determinada operação.

- **GPUSs**, desenvolvida para GPGPUS da Nvidia;
- **OmpSs**, adiciona as características do StarSs – fluxo de dado e computação heterogênea – através de diretivas no OpenMP.

ANTES de explicar os detalhes do modelo desenvolvido é importante contextualizar a origem do projeto e sua ligação com o instituto de física de São Carlos.

Π Flow é conjunto de implementações de fluxo de dados em diversos níveis de abstração. A primeira implementação foi realizada em 2012 com o trabalho intitulado "PIFLOW - projeto, simulação e implementação de um protótipo *dataflow* em FPGA" (41) que implementou uma versão da máquina de fluxo de dados de Manchester utilizando uma FPGA.

Depois deste primeiro trabalho houve mais três trabalhos pertencentes ao π Flow:

- π FlowBW, sucessor direto do primeiro trabalho, esta implementação tem como objetivo utilizar dutos de largura de banda para diminuir os *overheads* de comunicação presentes na máquina de fluxo de dados de Manchester e, por consequência, em sua implementação em FPGA;
- π FlowMR (42), este trabalho é uma extensão do π FlowBW, neste caso são implementados múltiplas instâncias do π FlowBW e arranjadas topologicamente na forma de um anel. Esta extensão permite que mais unidades de processamento sejam inseridas ao sistema de maneira direta, já que basta inserir mais uma instância mantendo a topologia de anel;
- π FlowMT, esta é a implementação apresentada neste trabalho.

3.1 π FlowMT

π FlowMT é um modelo de execução dirigido pelos dados, desenvolvido com objetivo ser eficiente na execução de aplicações em um ambiente do tipo *many-core*, isto é, um ambiente onde o número de unidades de processamento é alto e a memória do sistema é compartilhada entre todas as unidades.

O foco em processadores *many-core* surge de maneira natural, com a atual tendência de crescimento do número de núcleos dos processadores é apenas uma questão de tempo para que processadores deste tipo estejam disponíveis em ambientes de pesquisas e até mesmo no mercado. E como o desempenho é fator fundamental para a escolha de qual modelo de execução será utilizado nestes processadores é importante que o desenvolvimento do mesmo seja realizado antes dos processadores estarem disponíveis, já que com isso é possível que o modelo de execução já esteja otimizado para lidar com estes processadores, permitindo uma vantagem significativa em relação aos outros modelos.

Para atingir este objetivo o modelo foi desenvolvido utilizando técnicas de fluxo de dados. Estas técnicas permitem que os problemas encontrados com a utilização de modelos de execução tradicionais sejam evitados, como, por exemplo, a utilização de técnicas de sincronização. Note que a não utilização de técnicas de sincronias pelo programador pode ser duplamente vantajosa: não só permite que o programador faça um código mais simples, como também permite que sua execução possa ser mais eficiente, já que implementar as técnicas de sincronia de maneira eficiente é uma tarefa difícil.

O modelo foi desenvolvido com foco na utilização de fluxo de dados com granularidade grossa: as instruções são agrupadas em blocos de múltiplas instruções. Na representação de um código na forma de grafo direcionado os nós do grafo são estes blocos de instruções e as arestas são as entradas e saídas dos mesmos. Como um bloco de instruções possui um número arbitrário de instruções é necessário que estes blocos possuam múltiplas entradas e saídas, portanto na representação em grafo múltiplas arestas tem como entrada ou destino um único bloco de instruções.

A utilização de técnicas de modelos dirigidos pelos dados são fundamentais para permitir que o modelo seja eficiente em processadores *many-core*, já que com elas o paralelismo presente é explorado de maneira natural, isto é, todos os blocos de instruções que tenham todas as suas entradas disponíveis podem ser executados em paralelo, o que é importante, já que com o número elevado de núcleos a exploração *manual* do paralelismo seria uma tarefa ainda mais difícil.

De um ponto de vista menos abstrato é necessário armazenar e organizar estes blocos de instruções durante a execução de um programa neste modelo: aqueles blocos de instruções que possuem todas as entradas disponíveis e aqueles que ainda esperam entradas. Os blocos que já possuem todas as entradas disponíveis já podem ser executados e, portanto, não precisam de nenhum tratamento extra, já os blocos de instruções que ainda necessitam de entradas devem ser armazenados em uma estrutura que realize o pareamento do fluxo de dados, isto é as saídas de um bloco de instruções que se tornarão entradas para outros, com os blocos de destino.

3.1.1 Implementação

O π FlowMT foi implementado em c++ na forma de uma biblioteca para facilitar a utilização do mesmo em códigos já existentes. A outra razão para utilização da linguagem é o fato da mesma ser uma linguagem de baixo nível, permitindo que os acessos aos módulos do sistema operacional, por exemplo a criação ou destruição de uma *thread*, sejam realizados da maneira mais eficiente possível.

O modelo utiliza como base, isto é, para a criação e gerenciamento das *threads*, a estrutura paralela presente na API definida pelo padrão POSIX implementado na biblioteca PThreads.

O modelo é do tipo *thread-pool*, isto é, as *threads* são criadas e esperam tarefas para serem executadas. As tarefas são os blocos de instruções, que neste caso foram implementados em uma estrutura desenvolvida chamada *codelet*.

Há pelo menos duas definições para o termo *codelet* na literatura (43, 44): “uma sequência, geralmente curta, de instruções que são executadas ininterruptamente até o final.”(43) ou “uma função pura que não contém nenhuma variável estática ou volátil e não possui nenhum valor de retorno.”(44)

O *codelet* desenvolvido em conjunto com o π FlowMT, embora possua certa semelhança com a segunda definição, é mais amplo e possui mais elementos, como visto a seguir:

1. **Um conjunto de entradas:** que quando completo, *i.e.* quando todas as entradas estão disponíveis, habilita o *codelet* para a execução;
2. **Uma função:** um trecho de código que é executado sobre as entradas;
3. **Um conjunto de saídas:** quando a execução da função é finalizada, o conjunto de saídas é completado e cada elemento desse conjunto atua como elementos de conjunto de entradas para outros *codelets*;
4. **Um identificador único:** um número que serve para identificar o *codelet* durante o processo de *match* das entradas e saídas.

Um exemplo de *codelet* pode ser visto na figura 13.

Além do *codelet* ser o responsável por *popular* as *threads* com tarefas para execução, eles também são peça fundamental para entender o fluxo de dados presente no modelo. O fluxo de dados acontece através dos conjuntos de saídas e entradas presentes nos *codelets*, isto é, as saídas de um *codelet* atuam como entradas para outros *codelets*. E, devido às características do fluxo de dados presente no modelo, a execução de um *codelet* só se torna possível quando todo o conjunto de entradas está disponível.

Figura 13 – Representação gráfica de um *codelet*. O conjunto de quadrados da parte superior representa o conjunto de entradas, onde os quadrados preenchidos representam as entradas já disponíveis. Como o *codelet* não foi executado – o conjunto de entradas não está completamente disponível – o conjunto de saídas, conjunto de quadrados representado na parte inferior do *codelet*, não está disponível.

Fonte: Elaborada pelo autor.

A função é executada de acordo com os dados do conjunto de entrada, não existindo restrição teórica em relação à complexidade da função escolhida. Todavia, na prática é necessário que a função executada possua um grau de granularidade elevado – o grau deve ser alto, porém não exageradamente alto para garantir que haja paralelismo suficiente. Isto ocorre porque, caso a função fosse de granularidade fina, os tempos necessários para o fluxo de dados, isto é, de uma saída virar uma entrada para outro *codelet*, seriam da mesma ordem dos tempos de execução da própria função, o que ocasionaria uma baixa eficiência na execução.

Os *codelets* são divididos de acordo com a disponibilidade dos seus conjuntos de entradas: os *codelets* que possuem todas as suas entradas disponíveis e aqueles que ainda esperam alguma entrada.

A execução de um programa escrito com π FlowMT é representada através de um dígrafo e, portanto, para entender melhor os detalhes de uma execução, é necessário a utilização de um diagrama, como visto na figura 14.

Figura 14 – Exemplo de execução utilizando *codelets* na forma de um dígrafo.

Fonte: Elaborada pelo autor.

Note que a execução dos *codelet* 1 e *codelet* 2 permite que os *codelets* 3 e 4 sejam executados, assim como a execução desses *codelets* permite que o último *codelet* (5) seja

executado.

Embora os passos para a execução de um programa sejam simples, a implementação do π FlowMT requer ajustes finos em várias etapas. Durante o desenvolvimento do π FlowMT várias escolhas para as estruturas de dados e métodos de sincronia foram realizadas.

As razões para cada uma das escolhas estão representadas nas seções 3.2 e 3.3.

3.2 Estruturas de dados

Cada estrutura de dados utilizada no modelo foi escolhida considerando o seu desempenho, ou seja, em casos onde a estrutura de dados necessária para aquele trecho da implementação é suficientemente simples — *i.e.* não necessita de nenhuma especialização para o modelo — foram utilizadas implementações existentes que já foram otimizadas e validadas sempre que possível.

Antes de entrar nos detalhes de cada estrutura de dados utilizada, é necessário apresentar o esquema geral das estruturas utilizadas — ver figura 15. Este diagrama mostra o conjunto de estruturas que constituem o π FlowMT. Cada caixa representa uma estrutura implementada e as ligações entre as mesmas são representadas pelas flechas, onde a linha contínua representa um elemento do tipo apontado, enquanto a linha tracejada representa uma estrutura que possui um *template* que utiliza a estrutura apontada.

3.2.1 Token

Estrutura que contém um dado e um identificador. A presença do identificador é necessária já que, diferentemente de outros modelos de execução dirigidos pelos dados onde o número máximo de dados por instrução é 2, no π FlowMT o número de dados é ilimitado, e portanto, é necessário saber a qual entrada este dado está relacionado.

Embora a implementação seja razoavelmente simples, há um fator, que a princípio, pode passar despercebido: como armazenar *qualquer* tipo de dado em uma estrutura de dados genérica?

Duas técnicas foram testadas:

- Realizar um *cast* dos dados, que nesse caso devem ser alocados de maneira dinâmica, para o tipo `void*`, isto é, o campo `data` da estrutura é um campo do tipo `void*`;
- Utilizar um *container* que encapsule os dados em um tipo único, por exemplo `std::any`, `boost::any` ou `boost::spirit::hold_any`, o que permite que os dados não sejam alocados dinamicamente. Note que mesmo não havendo alocação dinâmica explícita, os métodos desses *containers* fazem uso de alocação dinâmica.

Figura 15 – Estruturas usadas no modelo.

Fonte: Elaborada pelo autor.

Testes de desempenho comparando as duas técnicas foram realizados e podem ser vistos na tabela 1.

Tabela 1 – Comparação entre o tempo de acesso das múltiplas estruturas para armazenamento. Cada estrutura teve seu valor modificado e acessado 10^{12} vezes.

	Otimizado	Não otimizado
void*	3,3s	54,8s
std::any	20,0s	376,6s
boost::any	156,8s	828,3s
boost::spirit::hold_any	50,1s	321,7s

Fonte: Elaborada pelo autor.

Os resultados demonstram que a utilização dos *containers* é altamente custosa. Quando se considera que o principal objetivo é manter o modelo eficiente, a utilização dos mesmos deve ser descartada.

3.2.2 Core

Estrutura que representa o núcleo do *codelet*, já que em partes da execução o modelo não necessita de todo o *codelet*. Esse núcleo contém um identificador do *codelet*, a função que é executada e a quantidade de elementos do conjunto de entradas.

Esta estrutura também possui um elemento que pode impactar diretamente o desempenho: a função que é executada. Dois métodos para armazenar a função existem:

- Ponteiro de função;
- Utilização de um *container* para a função — *i.e.* `std::function` ou `boost::function`;

Os dois métodos são compatíveis, isto é, não é necessário realizar modificação no código para mudar de um para o outro. Novamente a escolha é feita considerando o desempenho, a diferença de *performance* a favor do uso de ponteiros é substancial, como visto na tabela 2.

Tabela 2 – Comparação entre os modos de armazenamento das funções.

	Com otimização	Sem otimização
Ponteiro	3,3s	28,9s
<code>std::function</code>	26,7s	203,9s
<code>boost::function</code>	23,3s	153,8s

Fonte: Elaborada pelo autor.

3.2.3 codelet

A estrutura básica do *codelet* já foi discutida anteriormente. A sua implementação é direta, já que os elementos desta estrutura já foram otimizados anteriormente.

Contém um vetor de entradas e outro de saídas, além de um elemento do tipo *core*. O outro elemento presente indica o número de elementos do conjunto de entrada que já estão disponíveis para a execução. Este é um importante elemento, pois é a partir dele que se determina se o *codelet* está apto para a execução.

3.2.4 Input

Classe que encapsula uma *token* para ser utilizado como uma entrada de um *codelet*.

Os elementos do tipo *input* são utilizados, no momento, exatamente como os elementos da classe *token*. Todavia, se em algum futuro desenvolvimento e otimização do modelo for necessário incluir um elemento ao tipo *input*, a classe já está desenvolvida.

3.2.5 Output

Diferentemente da classe `input`, desta vez os elementos não são simplesmente membros da classe `token`. Para o funcionamento da classe `output`, é necessário definir se esta saída funcionará como entrada para outro `codelet` ou se é um valor *final*, i.e., que não será usado por outro `codelet`. A distinção entre os dois tipos de saída é feito através da classe `type`.

Quando a saída serve como entrada para um `codelet`, esta classe possui um elemento da classe `core` que é o núcleo do `codelet` do qual essa saída será a entrada. Esse processo, embora pareça redundante, será melhor explicado durante o processo de pareamento na subseção 3.3.2.

3.2.6 Vetor<T>

Em todos os casos onde há o uso de um elemento do tipo vetor, ele é feito através dessa classe.

A classe é simples e serve apenas para encapsular os métodos básicos de outras classes de vetores. Isso é feito para facilitar a busca pelo maior desempenho, já que com isso, pode-se modificar a classe base sem ser necessário alterar todo o código.

A comparação é feita considerando os *containers*: `vector`, `queue` da biblioteca padrão e `stable_vector`, `circular_buffer`, `queue` da biblioteca `boost`.

Tabela 3 – Comparação entre a média do tempo de inserção de um elemento nas estruturas de armazenamento. Para o cálculo da média foram realizadas 2^{25} inserções.

	Sem otimização	Com otimização
<code>std::vector</code>	42ns	31ns
<code>boost::container::stable_vector</code>	83ns	48ns
<code>boost::circular_buffer</code>	55ns	32ns
<code>std::queue</code>	50ns	32ns
<code>boost::queue</code>	62ns	32ns

Fonte: Elaborada pelo autor.

O *container* escolhido foi o `std::vector` que foi o mais rápido em todos os testes, como visto na tabela 3.

3.2.7 Thread_args

Estrutura que serve para o controle interno das múltiplas *threads*. Possui uma referência à instância da classe `piflow` que a criou, já que é possível haver mais de uma instância do modelo em execução. Ao mesmo tempo, é necessário que a classe saiba a qual ela está relacionada.

Além disso ela possui outros três elementos de controle: um identificador do *pthread*s, um contador de *codelets* executados — usado para depuração e avaliação do balanceamento de carga —, e uma variável que indica se a *thread* está executando algum *codelet*.

3.2.8 Piflow

Classe principal do modelo, seus elementos são descritos a seguir:

- *vetor<token>*: é o vetor que contém todas as saídas que não são utilizadas como entradas para outros *codelets*;
- *vetor<tid>*: vetor com os identificadores das *threads* criadas usando *pthread*s;
- *vetor<thread_args>*: é o vetor que contém as variáveis de controle das *threads*;
- *sync*: classe que contém as estruturas de sincronização das *threads*;
- *hash_map*: classe que realiza o pareamento das saídas e *codelets*;
- *queue*: classe que mantém os *codelets* que estão prontos para a execução;
- $N_{threads}$: número de *threads* geradas.

As classes *sync*, *hash_map* e *queue* serão explicados de maneira conjunta na seção 3.3.

3.3 Métodos de sincronia

As múltiplas *threads* acessam três estruturas compartilhadas:

- *vetor<token>*: caso o *codelet* executado pela *thread* possua uma saída externa, a mesma é inserida neste vetor. Note que, como múltiplos *codelets* podem possuir uma saída externa, é necessário sincronizar os acessos a esta estrutura;
- *hash_map*, a cada *codelet* executado, as suas saídas internas são buscadas nesta estrutura, verificando se o *codelet* que terá como entrada aquela saída possui múltiplas entradas:
 - Caso aquela entrada seja a única do *codelet*, o mesmo é inserido diretamente na *queue*;
 - Caso o *codelet* possua múltiplas entradas, o processo é ramificado, através de uma busca na estrutura:
 - * Caso não exista nenhum *codelet* correspondente ao *codelet* destino desta saída, um *codelet em branco* é criado e esta saída será a primeira entrada desse novo *codelet*;

- * Caso já exista um *codelet* correspondente, esta saída é transformada em entrada. Se esta saída era a última necessária para completar o conjunto de entrada do *codelet*, o mesmo é retirado do *hash_map* e inserido na *queue*;
- *queue*, as múltiplas *threads* acessam essa estrutura em duas fases: inserir e retirar os *codelets* para a execução;

Existem diversas maneiras para sincronizar os acessos. Dois métodos foram analisados:

- *spin_lock*: este método faz com que as *threads* que tentem acessar uma região de exclusão mútua já ocupada permaneçam em *loop* até que a região seja liberada pela *thread* anterior; (45)
- Estruturas *lock_free*: essas estruturas permitem o acesso de múltiplas *threads* sem realizar o bloqueio da região. Isto é possível com a utilização de instruções e blocos atômicos, que são executados do começo ao fim sem que haja mudança de contexto. (46)

Nesta etapa, além de buscar o melhor método de sincronização para cada estrutura citada, também será analisado o impacto da estrutura interna — continuando o que foi apresentado na seção anterior 3.2.

3.3.1 *queue*

Outra estrutura onde a performance é extremamente importante é a fila onde os *codelets* que estão prontos para execução ficam armazenados. Afinal, com um número de *codelets* suficientemente grande, os tempos oriundos desta estrutura, por menor que sejam, ocupam uma parcela significativa do tempo de execução.

Esta estrutura tem como finalidade armazenar os *codelets* que estão aptos para a execução, *i.e.* todos os *codelets* que possuem um conjunto completo de entradas — paradigma *dataflow*, e deve permitir que as múltiplas *threads* acessem e retirem os *codelets* para a execução de maneira eficiente: diminuindo não só o tempo de acesso e retirada da estrutura como também o tempo necessário para a sincronização destas operações, já que, como uma única unidade é compartilhada entre as múltiplas *threads* é necessário garantir que um mesmo *codelet* seja extraído uma única vez. A operação inversa, a inclusão de um *codelet* apto para a execução, também requer os mesmos cuidados.

Como esta estrutura deve manter um número grande de *codelets*, o questionamento de qual deve ser o critério que define a prioridade dos *codelets* surge de maneira natural. Antes de verificar qual a melhor estrutura para o armazenamento destes elementos é necessário verificar se existe um critério que seja superior e, a partir desta informação, continuar o desenvolvimento de maneira apropriada.

Para isto foram implementados testes com vários critérios de escalonamento de *codelets* diferentes:

- De acordo com a ordem de inserção — LIFO e FIFO;
- De acordo com o número de saídas: do *codelet* com mais saídas para o com menos saídas e vice-versa;
- De acordo com o a profundidade do *codelet* na representação em forma de dígrafo: partindo do *codelet* mais próximo da raiz e do *codelet* mais distante;
- Aleatório;

Figura 16 – Tempo de execução em número de ciclos dos algoritmos implementados variando o tipo de escalonador utilizado durante a execução. A execução foi realizada na máquina real utilizando o conjunto de entradas **A**. (a) Multiplicação de Matrizes; (b) Produto Interno; (c) Fatoração L.U.; (d) Quick Sort.

Fonte: Elaborada pelo autor.

A comparação entre os diversos critérios de escalonamento dos *codelets*, como visto na figura 16, demonstra que os melhores casos ocorrem com as implementações mais simples, LIFO e FIFO, já que estas não possuem os custos extras necessários para a criação e manutenção das filas de prioridade. Com isto ficou claro que não é necessário utilizar um escalonador que

dependa do *codelet* executado. Assim, o resto do desenvolvimento foi feito considerando uma estrutura do tipo FIFO.

Três alternativas para a implementação da fila de *codelets* foram implementadas:

1. Uma única fila que será acessada por todas as *threads*:
 - a) Utilizando o mesmo *container* da estrutura `vetor<T>` e um `spin_lock` para realizar a sincronia de acesso entre as múltiplas *threads*;
 - b) Utilizando uma estrutura *lock-free* — a escolha da estrutura *lock-free* foi a `moodycamel::BlockingConcurrentQueue` (47), que foi validada e possui uma eficiência conhecida (48, 49);
2. Múltiplas filas, isto é, uma alternativa baseada em *work-stealing*, onde cada *thread* possui a sua própria fila de *codelets* e, caso não haja nenhum *codelet* para a execução na sua respectiva fila, a *thread* rouba um *codelet* de outra *thread*.

Os resultados podem ser vistos na figura 17.

A comparação entre as três implementações para a fila de *codelets* é razoavelmente simples, bastando escolher a estratégia que possua o melhor desempenho. A fila do tipo *lock-free* foi a fila que manteve a regularidade na maioria dos casos e portanto a sua escolha foi facilmente justificada.

A alternativa que utiliza o método de *work-stealing* é promissora, porém, em casos onde o número de *codelets* é razoavelmente baixo, a dificuldade em manter o balanceamento de carga é muito grande, o que acaba limitando fortemente o desempenho e escalabilidade do modelo. O seu pico de eficiência ocorre no algoritmo de multiplicação de matrizes, onde o número de *codelets* disponíveis é extremamente alto desde o início da execução e independentes entre si.

3.3.2 *hash_map*

É fundamental garantir que o sincronismo entre os acessos a esta estrutura sejam eficientes. Afinal, o número de pareamentos que acontece durante uma execução é extremamente alto. Cada *codelet* executado representa um possível pareamento.

Diferentemente da *queue*, neste caso não há tantas opções para o método de sincronismo, já que não foi encontrada uma estrutura adequada do tipo *lock-free* e a divisão dos *codelets* em múltiplas estruturas de pareamento seria altamente ineficiente — já que caso o pareamento de saídas e entradas acontecesse com *codelets* de *threads* distintas, seria necessário percorrer boa parte das *threads* para encontrar o *codelet* destino, aumentando o tempo e realizando acessos sincronizados de maneira muito mais frequente. Logo foi escolhido a utilização de `spin_locks`.

Figura 17 – Tempo de execução dos algoritmos variando o tipo de escalonador. Execução realizada na máquina real utilizando o conjunto de entradas **B**. O eixo das abscissas representa o número de threads. Já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

Para verificar qual a melhor a estrutura de dados os algoritmos, vide apêndice [A](#), que utilizam mais esta estrutura — isto é, aqueles onde existe uma grande dependência entre os múltiplos *codelets* — foram executados e os resultados comparados, como visto na tabela [4](#).

A estrutura escolhida foi a `boost::unordered_map`, pois a mesma foi a estrutura mais eficiente em todas as operações analisadas.

Tabela 4 – Comparação entre as estruturas de pareamento para as diversas operações utilizadas, a letra O sobrescrita representa a execução do código compilado com otimizações.

	Inserção	Inserção ^O	Busca	Busca ^O	Remoção	Remoção ^O
<code>std::map</code>	180ns	38ns	670ns	125ns	1440ns	433ns
<code>std::unordered_map</code>	241ns	37ns	109ns	31ns	221ns	39ns
<code>boost::unordered_map</code>	244ns	36ns	104ns	31ns	151ns	37ns

Fonte: Elaborada pelo autor.

3.3.3 sync

Esta estrutura serve apenas para armazenar os mecanismos de sincronização utilizados, permitindo que a troca dos mesmos seja feita de maneira mais simples.

Devido à utilização da estrutura do tipo *lock free*, são necessários apenas dois `spin_locks`: um para o `hash_map` e outro para o `vetor<Token>`.

Agora que os detalhes do π FlowMT foram explicados, é importante visualizar como um algoritmo é implementado e executado no mesmo, o que será visto nas seções 3.5 e 3.6.

3.4 Códigos utilizando o modelo

A implementação de um código que utilize o modelo começa com a criação de uma instância do modelo, através da criação de um elemento do tipo `Piflow::Piflow` que tem como argumento o número de *threads* desejado. Com isso todos os elementos do modelo estão criados, restando iniciar os mesmos através da chamada `Piflow::Piflow.Start()`.

Com o modelo já iniciado basta começar a criar os *codelets* que serão executados.

Para criar um *codelet* basta fazer: `Piflow::Codelet(id,func,n_ins);`, este *codelet* tem um `id`, um endereço de uma função e o número de entradas. Depois de criado basta adicionar o *codelet* na fila de *codelets* com a chamada `Piflow::Piflow.Insert(codelet)` na instância do modelo criada. Note que o modelo cria $N - 1$ *threads*, e caso não seja feita a chamada `Piflow::Piflow.Run()` a primeira *thread* não executará os *codelets* e ficará esperando as $N - 1$ *threads* terminarem o resto da execução.

Depois disso basta continuar com a criação das saídas dos *codelets* dentro das funções, como por exemplo:

Código-fonte 4: Exemplo de função do modelo.

```

1 Piflow::OutV exemplo(Piflow::InV input){
2   Piflow::OutV outputs;
3   outputs.emplace_back(null,0,0,&exemplo,1);
4 }
```

```

5   return outputs;
6 }

```

Neste exemplo a função `exemplo` cria um elemento do tipo saída, com `id` e `label` com valores 0 e tem como destino a própria função `exemplo`. O *codelet* destino desta saída tem apenas uma entrada, esta própria saída.

3.5 Implementação de um programa

A implementação da fatoração L.U., descrita no algoritmo 3 do apêndice A, no π FlowMT é demonstrada a seguir.

Código-fonte 5: Trecho do main do programa de fatoração L.U. utilizando π FlowMT.

```

1 Piflow::Piflow paralelo(Nthreads);
2
3 paralelo.Start();
4
5 Piflow::Codelet codelet(0,&lufac2,1);
6 codelet.Insert(Piflow::Input(A, 0));
7 paralelo.Insert(codelet);
8
9 paralelo.Run();
10
11 paralelo.Finish();
12
13 A=reinterpret_cast<float**>(paralelo.outputs[0].Data());

```

Primeiro é criado o ambiente paralelo `Piflow` com $N_{threads}$ e o mesmo é iniciado em `paralelo.Start()`, o que significa que as *threads* criadas estão esperando os *codelets* para a execução.

Em seguida, o primeiro *codelet* é criado com a ID 0 que executará a função `lufac2` e possui uma entrada. A única entrada deste *codelet* é adicionada ao mesmo — a matriz *A* que será fatorada e seu identificador, neste caso 0.

Como este *codelet* já está pronto para a execução, o *codelet* só possui uma entrada e ela já está disponível, ao realizar a chamada `paralelo.Insert(codelet)` o mesmo é movido diretamente para a fila de *codelets*.

A chamada `paralelo.Run()` permite que a *thread* principal também execute os *codelets*. Por fim, a chamada `paralelo.Finish()`, depois da execução de todos os *codelets*,

destrói as *threads* criadas. Depois, o endereço da matriz é retirado do vetor de saídas e colocado em *A*.

Resta analisar as outras funções utilizadas no código, começando com a função `lufac2` apresentada no código-fonte 6.

Código-fonte 6: Função `lufac2`.

```

1 Piflow::OutV lufac2(Piflow::InV input){
2     const uint64_t k = input[0].Id();
3     float** A = static_cast<float**>(input[0].Data());
4
5     Piflow::OutV outputs;
6     if(N-k>cutoff){
7         for(uint64_t l=k+1;l<N;l++){
8             outputs.emplace_back(A,k*N+l,k*N+l,&lufac1,1);
9         }
10    }
11    else{
12        outputs.emplace_back(A,k,k,&lufac_seq,1);
13    }
14
15    return outputs;
16 }
```

A função `lufac2`, disponível no código fonte 6, recebe um vetor de entradas e retorna um vetor de saídas.

As duas primeiras operações são realizadas na única entrada presente: o índice k e o endereço da matriz que é utilizada.

O próximo passo envolve a criação do vetor de saídas deste *codelet*. Note que, caso a iteração, representada pelo índice k , indique que o problema já passou do ponto de corte — isto é, não é capaz de gerar paralelismo suficiente para diluir os custos de comunicação — apenas um *codelet* é inserido no vetor de saída e, caso contrário, são inseridos $N - k - 1$ *codelets*.

A criação dos elementos presentes no vetor de saída segue o seguinte escopo:

`outputs.emplace_back(Dado, ID relacionado ao dado, ID do codelet que terá essa saída como entrada, endereço da função, número de entradas.`

No primeiro caso, onde são criados múltiplos *codelets*, todos possuem os mesmos valores para os dados, endereço da função e número de entradas. A diferença está nos campos

de identificação, onde o valor contém a informação de qual linha da matriz cada *codelet* irá operar.

Já para o caso que gera apenas um *codelet* a inserção segue o mesmo molde, onde a única modificação presente ocorre no endereço da função.

Código-fonte 7: Função `lufac1`.

```

1 Piflow::OutV lufac1(Piflow::InV input){
2   const uint64_t i = input[0].Id()\%N;
3   const uint64_t k = input[0].Id()/N;
4   float** A = static_cast<float**>(input[0].Data());
5
6   A[i][k]=A[i][k]/A[k][k];
7   for(uint64_t j=k+1;j<N;j++){
8     A[i][j]=A[i][j] - A[i][k]*A[k][j];
9   }
10
11   Piflow::OutV outputs;
12
13   outputs.emplace_back(A,k+1,k+1,&lufac2,N-k-1);
14   return outputs;
15 }
```

De maneira semelhante, a função começa com a identificação dos valores necessários: os índices i e k e o endereço da matriz e posteriormente a iteração do algoritmo é realizada.

Os vetores de saída oriundos deste *codelet* possuem apenas uma saída. Cada um dos $N - k - 1$ *codelets* desta iteração geram saídas que atuaram como entrada para um único *codelet* com ID $k + 1$.

Código-fonte 8: Função `lufac_seq`.

```

1 Piflow::OutV lufac_seq(Piflow::InV input){
2   uint64_t k = input[0].Id();
3   float** A = static_cast<float**>(input[0].Data());
4
5   for(;k<N;k++){
6     for(uint64_t i=k+1;i<N;i++){
7       A[i][k]=A[i][k]/A[k][k];
8       for(uint64_t j=k+1;j<N;j++){
9         A[i][j]=A[i][j] - A[i][k]*A[k][j];
```

```

10     }
11 }
12
13 }
14
15 Piflow::OutV outputs;
16 outputs.emplace_back(A); // saída que não atua como
    entrada de outro codelet
17 return outputs;
18 }

```

Analogamente, a função `lufac_seq` (ver código-fonte 8) segue os mesmos procedimentos: o índice k e o endereço da matriz são obtidos. Desta vez a saída do *codelet* não é uma entrada de outro, de modo que a mesma é inserida no vetor de saídas.

A representação em dígrafo pode ser vista na figura 18.

Figura 18 – Grafo de dependências da implementação da fatoração L.U., com um corte sequencial em k .

Fonte: Elaborada pelo autor.

Note que a figura 18 permite a identificação de qual função é executada por cada *codelet*, bastando analisar o padrão formado na representação em dígrafo — um estágio com

múltiplos *codelets* seguido por um estágio com apenas um *codelet*. O estágio com múltiplos *codelets* possui *codelets* que executam a função `lufac2`; já o *codelet solitário* executa a função `lufac1`. Note ainda, que o último *codelet* não possui saídas, o que indica que ele executa a função `lufac_seq`.

3.6 Demonstração de uma execução

Um exemplo de execução pode ser visto na figura 19.

Figura 19 – Representação da execução no modelo desenvolvido com dois processadores – P1 e P2. Nesta representação os *codelets* em execução são representados pela presença do mesmo em um dos processadores e a disponibilidade de entradas e saídas e representada pelo preenchimento do seu respectivo quadrado.

Fonte: Elaborada pelo autor.

METODOLOGIA DE AVALIAÇÃO

DIVERSOS são os fatores que podem impactar diretamente a análise de um modelo de execução (50):

- **O que é executado:** como a análise de um modelo de execução é vinculada exclusivamente à execução de um algoritmo implementado no modelo, é natural que os resultados obtidos sejam fortemente influenciados pelo algoritmo utilizado;
- **Onde a tarefa é executada:** além dos resultados do modelo serem vinculados aos algoritmos executados, eles também são vinculados ao ambiente de execução utilizado, isto é, a combinação *hardware* e *software* que — diferentes ambientes de execução podem atuar de maneiras diferentes em modelos de execução diferentes;
- **O que é analisado:** como os resultados de uma execução do modelo são vinculados ao algoritmo implementado e ao ambiente de execução utilizado é natural que as métricas obtidas com esta execução também possuam o mesmo vínculo, ou seja, dentre as métricas obtidas com a execução deve-se escolher aquelas que melhor descrevam o comportamento do modelo.

Embora seja possível argumentar que a importância de cada item não é a mesma, é evidente que todos são fortemente vinculados ao processo de análise de um modelo de execução e comparação de modelos de execução diferentes.

As próximas seções discutem as escolhas realizadas e os efeitos gerados por elas nos resultados.

4.1 Algoritmos

Como os resultados obtidos são vinculados ao algoritmo executado é natural que, para se obter a melhor análise do modelo, sejam executados múltiplos algoritmos. Como o objetivo da execução é a análise do modelo de execução e não do algoritmo, algumas escolhas devem ser feitas para simplificar o processo de execução e análise: cada algoritmo escolhido deve ter sido amplamente estudado, de modo que seu comportamento seja conhecido e facilmente detectado durante a análise.

Como a lista dos algoritmos (51) que possuem as características desejadas é demasiadamente extensa, foram escolhidos algoritmos que além de serem estudados são usados frequentemente em problemas numéricos ou de ordenação, seja pela sua simplicidade e eficácia, caso do *quick sort*, ou seja pela sua versatilidade de aplicações, como a FFT ou multiplicação de matrizes. Os algoritmos são descritos brevemente a seguir e em detalhes no apêndice A e suas implementações no π FlowMT podem ser vistas no anexo A.

- **Multiplicação de Matrizes:** simples multiplicação de duas matrizes quadradas;
- **Produto Interno:** produto interno entre dois vetores;
- **Fatoração L.U.:** fatoração de uma matriz quadrada como o produto de uma matriz triangular inferior e uma matriz triangular superior;
- **Integração Binária:** integração de uma função $f(x)$ utilizando o método do trapézio;
- **FFT:** implementação da transformada rápida de Fourier utilizando o algoritmo de Cooley-Tukey;
- **Agulha de Buffon:** utilização do método de Monte Carlo para aproximação de π ;
- **Quick Sort:** ordenação de um vetor utilização a técnica de dividir para conquistar;
- **Bucket Sort:** algoritmo de ordenação que divide um vetor em um número finito de recipientes que são ordenados posteriormente.

Além de entender os pseudocódigos apresentados é necessário entender como os mesmos foram implementados. Implementar o algoritmo seguindo completamente o pseudocódigo é possível e funcional, porém sua eficiência, considerando a execução paralela, pode não ser apropriada. Isto acontece devido à inerente dificuldade e impraticabilidade de se dividir a execução de um algoritmo em um grande número de partes com tamanhos próximos e suficientemente grandes para diluir os custos da paralelização durante as suas respectivas execuções, ou seja, uma boa distribuição de trabalho.

Para tornar o processo mais prático e viável, é necessário realizar algumas mudanças durante a implementação. O objetivo destas modificações é facilitar o processo de particionamento do código, onde, em situações limites, é mais eficiente executar trechos do código de maneira sequencial a executá-los de maneira paralela (52) — *i.e.* os custos da paralelização são equivalentes ao custo da execução destes trechos.

Os algoritmos *originais* estão disponíveis no apêndice A, enquanto a seguir serão apresentadas as eventuais modificações realizadas para a paralelização.

- **Multiplicação de Matrizes:** o paralelismo deste algoritmo surge da independência dos elementos da matriz resultante, isto é, no *loop* mais externo — no algoritmo 1 o *loop* em k , matematicamente o somatório do produto linha $\times$ coluna — e não há outras modificações em relação ao pseudocódigo apresentado;
- **Produto Interno:** o paralelismo também é obtido através do *loop* presente, isto é, o vetor é dividido em blocos e esses blocos realizam o produto interno de maneira sequencial; além disso, o resultado dos produtos internos parciais é somado através de uma redução paralela;
- **Fatoração L.U.:** novamente fazendo uso do *loop* presente, porém desta vez o *loop* considerado para o paralelismo é o *loop* em i . Note que o tamanho deste *loop* varia, de modo que, para certos valores de k o *loop* não é capaz de gerar *codelets* suficientes para alimentar as múltiplas *threads*. Com isso uma segunda modificação é feita para a execução paralela: acontece um corte para o *loop* em k , de modo que uma parte do *loop* é feita executando o *loop* interno paralelamente. Já a outra parte é feita de maneira sequencial;
- **Integração Binária**
 - **Recursivo:** nesse caso o paralelismo é obtido através das chamadas duplamente recursivas da função `integral`;
 - **Não-recursivo:** a implementação deste algoritmo no contexto paralelo é muito parecida à implementação do algoritmo de produto interno: o *loop* é paralelizado e o resultado da soma é obtido através de uma redução.
- **FFT:** outro exemplo de algoritmo onde o paralelismo é explorado através das chamadas duplamente recursivas. Note que, assim como no algoritmo de fatoração L.U., o tamanho das iterações diminui com a profundidade da recursão; logo, as camadas mais profundas da recursão não são capazes de fornecer o paralelismo necessário. A alternativa encontrada é exatamente a mesma utilizada no exemplo da fatoração: realiza-se um corte sequencial, onde a partir de determinada profundidade as chamadas deixam de ser realizadas em paralelo e passam a ser realizadas de maneira sequencial;

- **Agulha de Buffon:** a paralelização deste algoritmo é simples, já que não existe dependência entre as iterações do *loop*. Para as implementações realizadas o paralelismo é explorado no *loop* mais externo;
- **Quick Sort:** a implementação paralela deste algoritmo segue a mesma linha da implementação da FFT: o paralelismo é obtido através das chamadas recursivas, onde a cada chamada o paralelismo disponível dobra — ou triplica, no caso da implementação com dois pivôs. E assim como no caso da FFT, é necessário realizar o corte sequencial para manter o algoritmo eficiente;
- **Bucket Sort:** este algoritmo não necessita de modificações para a execução paralela; basta paralelizar o *loop* de ordenação.

4.2 Ambiente de execução

Como o π FlowMT tem como objetivo manter a eficiência da execução em ambientes *many-core*, que ainda não estão disponíveis no mercado, o ambiente de execução onde os testes serão realizados deve ser capaz de fornecer características próximas o suficiente às desejadas.

Duas alternativas surgem de maneira natural: executar as implementações em um simulador, podendo assim simular um processador *many-core* ou executar as implementações em um processador real com o maior número de núcleos disponíveis atualmente.

A escolha feita foi a de utilizar as duas alternativas de maneira combinada já que suas características e limitações são complementares: o simulador permite a execução em um ambiente controlado e totalmente isolado, o que elimina efeitos presentes no sistema real — por exemplo, o escalonador de tarefas do sistema operacional, o uso de *swap*, etc; já a máquina real permite que tais efeitos sejam mensurados, com a contrapartida de que o número de núcleos usados é sempre o mesmo, o que não ocorre no simulador.

4.2.1 Simulador

A escolha de qual simulador usar é bastante direta, já que poucos possuem todas as características desejadas, quais sejam:

1. Ser validado considerando um processador real. Este talvez seja o item mais importante, já que um simulador que não tenha sido validado geraria dados com grau de confiança inferior ao necessário;
2. Ser capaz de simular um processador com um número arbitrário de núcleos. O simulador deve ser capaz de, no mínimo, simular um processador com mais núcleos do que o processador real utilizado, já que do contrário, o simulador não teria valia;

3. Ser eficiente, já que a execução de qualquer programa através de um simulador é notoriamente mais lenta que a execução tradicional. Como o número de vezes que o simulador é executado é alto, se o simulador não for suficientemente eficiente a utilização do mesmo se torna impraticável.

Com estas características, dois simuladores foram encontrados: Sniper (53) e o ZSim (54). Simuladores como o gem5 (55), Flexus (56) e MARSS (57) realizam as simulações de maneira muito lenta: considerando uma velocidade média para estes simuladores de 200 mil instruções por segundo seria necessário executá-los por oito horas para simular um segundo de um processador *single-core*, já a simulação de um segundo de um processador com mil núcleos levaria cerca de um ano para ser realizada.

4.2.1.1 Sniper

Resultado de uma colaboração entre a Intel, *Interuniversity Microelectronics Centre* (IMEC) e universidades belgas. O simulador tem três grandes objetivos:

1. Analisar o desempenho de processadores futuros, *e.g.* processadores *many-core*;
2. Explorar as características de arquitetura desses novos processadores;
3. Permitir que otimizações de *software* sejam realizadas antes dos processadores estarem disponíveis no mercado.

O simulador é construído com base no simulador Graphite (58) e implementa um modelo de abstração de alto nível conhecido como *interval.* (59)

Este modelo realiza a cronometragem dos núcleos simulados sem realizar o rastreamento individual das instruções executadas dentro do *pipeline* do núcleo. Isso é possível, pois, de acordo com o modelo, os eventos que *quebram* o *pipeline* — *e.g.* *cache miss*, uma má predição de desvio, etc — dividem o fluxo de instruções em intervalos determinados pelo tipo de evento. Com os tempos de cada intervalo é possível realizar uma cronometragem das tarefas executadas dentro do simulador de uma maneira rápida e precisa.

O simulador foi validado utilizando um sistema multi-processador com quatro processadores da família Peryn da Intel (Xeon X7460), onde cada processador possui seis núcleos. Os testes de validação foram realizados utilizando os *benchmarks* presentes na ferramenta SPLASH-2 (60).

4.2.1.2 ZSim

Desenvolvido por pesquisadores de Stanford e do MIT com o objetivo de permitir a simulação de processadores com milhares de núcleos, tem como objetivo satisfazer quatro critérios:

- Ser preciso;
- Ser rápido;
- Ser de propósito geral, isto é, permitir a execução de qualquer tipo trabalho;
- Ser de fácil utilização.

Para atingir estes três critérios o simulador faz uso de três técnicas de simulação:

1. Tradução dinâmica dos executáveis, esta técnica tem como objetivo tornar a simulação de sequencial, *i.e.* simulação de um processador *single-core*, mais rápida. Com isso é possível eliminar a necessidade de modelar as instruções x86 no simulador. O tempo de execução de cada instrução é obtido através de um *timing model* construído especificamente para o ZSim para núcleos com execução fora de ordem. Essa técnica faz uso da ferramenta de instrumentação Pin (61).
2. Paralelização do simulador em duas fases utilizando um algoritmo do tipo *bound-weave*, para tornar o simulador ainda mais rápido é necessário que o simulador aproveite os múltiplos núcleos presentes no *host*.
3. Virtualização para o usuário, para permitir que o usuário execute qualquer tipo de tarefa sem que haja a implementação de um simulador *full-system*, é necessário que o simulador seja capaz de lidar com as eventuais chamadas do sistema que a tarefa realize, além de possuir um escalonador para *organizar* a execução das múltiplas *threads* ou múltiplos processos.

O simulador foi validado utilizando um processador da família Westmere da Intel através da execução do *benchmark* SPEC CPU2006 (62) e outros programas *multithreads* desenvolvidos internamente.

Para escolher entre os dois simuladores foram realizados alguns testes de desempenho, apresentados na tabela 5.

Tabela 5 – Tempo de execução dos simuladores na realização de uma multiplicação de matrizes de tamanho 64×64 . As simulações foram realizadas com 1 *core* e 1.024 *cores*.

	1 <i>core</i>	1.024 <i>cores</i>
ZSim	2,9s	5,2s
Sniper	24,8s	132,7s

Fonte: Elaborada pelo autor.

Os resultados demonstram que o ZSim é muito mais eficiente que o Sniper, de modo que a escolha é facilmente justificada.

A configuração usada para a validação do simulador — que simula um processador Intel Xeon L5640 de seis núcleos — serviu como base para a configuração dos processadores simulados, com duas modificações.

1. Variação no número de núcleos — de 1 a 2.048;
2. Utilização da mesma estrutura hierárquica de *cache* do processador real utilizado;

Os processadores simulados, de maneira geral, seguem a especificação listada no quadro 1.

Cache L1I	64KB por núcleo	3 ciclos
Cache L1D	32KB por núcleo	4 ciclos
Cache L2	256KB por núcleo	7 ciclos
Cache L3	64MB compartilhados	27 ciclos
Memória DDR3	∞ ¹	40 ciclos
Frequência da Memória	1.333MHz	
Frequência do processador	3GHz	

Quadro 1 – Especificações para os processadores simulados.
Fonte: Adaptado de INTEL (63);SANCHEZ; KOZYRAKIS (54).

Note que, como os futuros processadores que irão possuir 2.048 núcleos estão razoavelmente longes de serem desenvolvidos, é necessário atuar de maneira especulativa em relação à estrutura dos mesmos. Além de simular os processadores de maneira conservadora — *i.e.* mantendo a hierarquia de *cache* atual — foram realizadas simulações mais arrojadas, onde o tamanho do *cache* variou de acordo com a extrapolação obtida através da figura 20.

4.2.2 Threadripper

A segunda etapa de avaliação utiliza um processador real com muitos núcleos. Encontrar um processador com essa característica e que estivesse disponível no Brasil por um preço acessível foi uma tarefa que exigiu uma exaustiva busca. O processador escolhido, o AMD Ryzen Threadripper 2990WX — foi lançado nos Estados Unidos em 13 de agosto de 2018 (64) e adquirido em 17 de setembro do mesmo ano.

O Threadripper 2990WX é um processador com 32 núcleos com “2-way *simultaneous multithreading*” que opera a 3GHz — 4.2GHz com *boost*, mas como essa tecnologia beneficia o uso individual dos núcleos² a mesma foi desabilitada durante os testes. O restante das especificações está disponível no quadro 2.

Os testes foram realizados utilizando o sistema operacional Linux Debian ‘Sid’ com o *kernel* 4.9.0-27. (65)

² O *boost* na frequência aumenta a frequência de acordo com o uso do processador, *i.e.* se apenas um dos 32 núcleos estiver em funcionamento ele operará a 4.2GHz, com mais núcleos em uso menor o valor da frequência até atingir os 3GHz.

Figura 20 – Tamanho do *cache* L3 em MB em função do número de *threads* de cada processador

Fonte: Elaborada pelo autor.

Cache L1	64KB por núcleo
Cache L1D	32KB por núcleo
Cache L2	256KB por núcleo
Cache L3	64MB compartilhados
Memória DDR4	128GB
Frequência da Memória	1.333MHz
Frequência do processador	3GHz

Quadro 2 – Especificações do computador usado. Os valores de tempo de acesso das camadas do *cache* não estão disponíveis.

Fonte: Adaptado de AMD (64).

4.2.3 Compiladores

Um outro fator que influencia a execução é como e onde o código fonte é compilado. Diversos compiladores existem (66) e escolher qual deve ser utilizado depende de fatores relacionados ao projeto desenvolvido.

A decisão de adicionar mais uma variável — o compilador utilizado — aos dados surge de maneira natural. Um dos modelos usados para a comparação (OmpSs) possui um compilador próprio e este compilador é incompatível com a maioria dos outros modelos testados. Assim se não é possível realizar todos os testes com um único compilador, é natural que o horizonte de testes seja ampliado.

Como existem vários compiladores, a escolha de quais serão utilizados foi feita conside-

rando a referência. (67) Os compiladores e suas respectivas compatibilidades com os modelos de execução podem ser vistos no quadro 3.

Compilador	π FlowMT	OpenMP	T.B.B.	OmpSs
g++-8.2.0 (68)	✓	✓	✓	✗
icpc-19.0.0 (69)	✓	✓	✓	✗
clang-7.0.0 (70)	✓	✓	✓	✗
pgc++-18.4-0 (71)	✓	✓	✓	✗
aocc-1.2.1 (72)	✓	✓	✓	✗
zapcc-7.0.0 (73)	✓	✓	✓	✗
mcxx-2.1.0 (74)	✗	✓	✗	✓

Quadro 3 – Compiladores utilizados e suas respectivas compatibilidades.

Fonte: Elaborada pelo autor.

4.3 Métricas

A primeira e, indiscutivelmente, mais importante métrica utilizada na análise é o tempo de execução, medido através da contagem de número de ciclos de *clock* do processador entre o início e fim da região de interesse do programa. Apenas as operações apresentadas nos pseudocódigo são consideradas, de modo que as operações de inicialização — *e.g.* criação e alocação de vetores — não são incluídas.

Para as execuções realizadas no simulador, a obtenção do número de ciclos é trivial — a execução do simulador gera um arquivo de saída com vários dados sobre o programa executado, dentre eles o número de ciclos. Já para as execuções realizadas no programa real, a medida do número de ciclos é feita através da instrução em *assembly* `rdtsc`. (75) Para isso a instrução é chamada duas vezes, uma no começo da região de interesse e a segunda após a execução da mesma. Assim a diferença entre os dois valores retornados pela instrução contém o número de ciclos gastos para a execução.

Note que para as execuções no simulador, dada a sua natureza determinística, não é necessário repetir o experimento várias vezes, já que o seu resultado não varia. Já para as execuções na máquina real, onde existe variação, os experimentos são executados múltiplas vezes e a média dos tempos de execução é tomada, com o objetivo de obter um resultado preciso. Como na maioria das vezes o número de repetições é o mesmo (100 vezes), este valor só será explicitamente informado em experimentos que utilizem um número de repetições diferente.

A segunda métrica, *speedup*, é derivada diretamente da primeira e indica quão escalável a execução paralela (76) é. O cálculo é simples e utiliza apenas dois valores: o tempo de execução com uma *thread* ($t(1)$) e o tempo com n *threads* ($t(n)$):

$$S(n) = \frac{t(1)}{t(n)}$$

Seguindo com métricas derivadas do tempo de execução, tem-se a métrica chamada eficiência que mostra a utilização média dos n processadores.

$$e(n) = \frac{t(1)}{n \times t(n)}$$

A próxima métrica só é obtida com a execução dos modelos no simulador e é relacionada ao balanceamento de carga entre os n processadores — o simulador permite a contagem de ciclos de todos os n núcleos de maneira independente, o que não é possível de ser feito com a execução no processador real. Uma métrica parecida é obtida com a execução do π FlowMT, já que o modelo desenvolvido conta com a opção de visualizar o número de *codelets* executados por *thread*. Considere que o número de ciclos utilizado pelo i -ésimo núcleo é $t_i(n)$ e n é quantidade total de núcleos.

$$\zeta(n) = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (t_i(n) - \bar{t}(n))^2}}{\sum_{i=1}^n t_i(n)}$$

Outra métrica que só está disponível através da execução no simulador mede o *overhead* causado pelo modelo de execução, onde $O(n)$ é a soma do número de ciclos dos n núcleos.

$$q(n) = \frac{O(n)}{O(1)}$$

Há ainda uma outra métrica, que de maneira qualitativa, mede o impacto da granularidade no *speedup* obtido. Considere o programa de integração binária: nele é muito fácil modificar o tamanho das tarefas, *i.e.* aumentar a granularidade, pois basta modificar a função integrada para uma mais complexa.

Para o conjunto de funções com variadas complexidades foram escolhidos as funções polinomiais, de modo que o grau do polinômio será proporcional à granularidade.

RESULTADOS

Os resultados serão divididos entre aqueles obtidos com a execução no simulador e na máquina real.

5.1 Máquina real

Nesta seção serão analisados os resultados obtidos com as execuções dos algoritmos na máquina real. Primeiramente será analisado o impacto do compilador durante as execuções e em seguida uma comparação entre os modelos de estado de arte e o modelo desenvolvido.

Também será analisado o impacto do tamanho das tarefas e a eficácia do escalonador do π FlowMT.

5.1.1 Compiladores

Os primeiros resultados consideram os diferentes compiladores utilizados, já que com essa informação é possível definir se o compilador escolhido para os próximos resultados teria algum *bias* em relação a um compilador. Os testes foram realizados utilizando o conjunto de entradas **B** do apêndice **B**.

Figura 21 – Tempo de execução dos algoritmos variando o compilador utilizando o π FlowMT e o conjunto de entradas **B**. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

Com as informações presentes nas figuras 21, 22 e 23 é possível concluir que os compiladores, com exceção ao compilador `mcc++`, são realmente compatíveis com todos os modelos utilizados e que os resultados não variam de maneira significativa entre um compilador e outro. Outro ponto importante para o restante dos resultados é notar que o compilador `mcc++` não permitiu que algumas das implementações do OpenMP fossem executadas, mesmo com a compilação acontecendo de forma aparentemente normal e que as implementações que foram executadas foram consideravelmente mais lentas que em relação aos outros compiladores.

Figura 22 – Tempo de execução dos algoritmos variando o compilador utilizando OpenMP e o conjunto de entradas **B**. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

Como o modelo OmpSs é compatível apenas com o compilador `mc++` é esperado que os resultados que utilizem esse modelo — e consequentemente este compilador — sejam mais lentos comparados aos demais, já que para os demais resultados dos outros modelos o compilador utilizado será o `g++`.

Figura 23 – Tempo de execução dos algoritmos variando o compilador utilizando T.B.B. e o conjunto de entradas **B**. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

5.1.2 Número de ciclos

Os próximos resultados analisados comparam diretamente o tempo de execução, em número de ciclos, entre os modelos. Eles são apresentados na figura 24.

Como esperado e discutido na subseção 5.1.1 os resultados do modelo *OmpSs* são mais lentos que os demais. Embora o compilador tenha sua parcela de responsabilidade, não é possível descartar que o próprio modelo seja menos eficiente que os demais.

Figura 24 – Tempo de execução dos algoritmos. Execução realizada na máquina real utilizando o conjunto de entradas **B**. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

A comparação entre os outros modelos é mais acirrada. Do ponto de vista do π FlowMT os resultados são positivos para os algoritmos onde a granularidade é suficientemente grande, por exemplo a multiplicação de matrizes e a FFT, sendo capaz de ser competitivo com os modelos de estado da arte utilizados. Já para os algoritmos onde a granularidade não é suficiente, neste caso a integração binária com $f(x) = x$ e o produto interno, os resultados confirmam o

que já era esperado; em casos onde a granularidade não é grande o suficiente, os custos de comunicação não conseguem ser diluídos e passam a ter um impacto significativo no tempo de execução.

5.1.3 *Speedup*

A análise do tempo de execução permite determinar qual o modelo mais rápido, mas é insuficiente para determinar qual o modelo mais escalável. Para esses casos, é importante analisar os valores da outra métrica apresentada: o *speedup*, veja figura 25.

A avaliação do *speedup* permite a confirmação de algumas tendências importantes para o restante das análises.

Considere o *speedup* obtido com a execução do algoritmo de Monte Carlo. Nela todos os modelos escalaram de maneira semelhante, o que é esperado, pois neste algoritmo não existe dependência entre os dados e as tarefas são abundantes, permitindo que os escalonadores de cada modelo operem de maneira ideal.

Do ponto de vista do π FlowMT, a análise confirma que o modelo depende da utilização de granularidade grossa: para os casos onde a granularidade é sabidamente pequena os ganhos de escalabilidade obtidos são muito inferiores aos obtidos em casos com granularidade mais grossa, como a FFT.

Outro ponto interessante observado com essa análise é a inconsistência do OmpSs, variando de acordo com os algoritmos de melhor escalabilidade para a pior, indicando novamente a possível ineficiência do modelo

Fato semelhante ocorre com o OpenMP, mas este é significativamente mais rápido, o que pode explicar a variação da escalabilidade, já que é difícil escalar um programa que já é sequencialmente rápido. Neste caso a execução sequencial considera a execução do algoritmo com apenas uma *thread*.

Já o T.B.B. é o modelo que garante a escalabilidade mais constante, embora não seja o modelo que garanta a maior consistência com a execução mais rápida.

Figura 25 – *Speedup* obtido com a execução dos algoritmos. Execução realizada na máquina real utilizando o conjunto de entradas **B**. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

5.1.4 Balanceamento de carga

Para a execução na máquina real esta métrica só é possível de ser obtida com o π FlowMT, devido a capacidade do mesmo de listar o número de *codelets* executados por *threads* e, como o tamanho dos *codelets* disponíveis é sempre próximo — isto é, os *codelets* que estão na mesma profundidade na representação de dígrafo possuem um número próximo de instruções — é possível considerar a razão entre o desvio padrão do número de *codelets* executados por *thread* e o número total de *codelets* executados como uma métrica de

balanceamento de carga, ver figura 26.

Figura 26 – Desvio padrão do número de *codelets* executados em função do número total de *codelets* executados. Execução realizada na máquina real utilizando o conjunto de entradas **B**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) Quick Sort; (k) Quick Sort - 2 Pivôs; (l) Bucket Sort.

Fonte: Elaborada pelo autor.

Os resultados demonstram que o escalonador implementado no π FlowMT é eficiente, mantendo sempre a variação do número de execução dos *codelets* entre as *threads* menor do que 5% do total de *codelets* executados, com exceção para os casos onde a granularidade é muito fina, por exemplo a integração binária de $f(x) = x$, ou em casos onde o número de

codelets é baixo, como por exemplo o algoritmo de ordenação *bucket sort*.

5.1.5 Efeitos da granularidade

Com o programa de integração binária é possível modificar o tamanho das tarefas de maneira bastante direta, o que permite analisar o impacto da granularidade no *speedup* obtido. Executou-se o algoritmo de integrações binárias variando a função integrada — de $f(x) = x$ a $f(x) = \sum_{i=1}^N x^i$ — e mantendo o restante das entradas constante e de acordo com as entradas **B** do apêndice B. Os resultados estão disponíveis na figura 27.

Figura 27 – *Speedup* obtido com a variação do tamanho dos *codelets*. O eixo das ordenadas representa o *speedup* obtido e o eixo das abscissas representa o grau do polinômio integrado. Execução realiza com 8 *threads* utilizando o π FlowMT na máquina real.

Fonte: Elaborada pelo autor.

Novamente o resultado confirma o esperado, o impacto da granularidade é alto para o modelo desenvolvido. Sendo que resultados melhores acontecem com a utilização de *codelets* com a granularidade mais grossa.

5.2 Simulador

De maneira semelhante, os mesmos resultados foram obtidos com a execução no simulador. Desta vez porém, as entradas utilizadas nos algoritmos pertencem ao conjunto **A** do

apêndice B, já que a execução do conjunto B no simulador leva um tempo demasiadamente longo.

A tabela 6 mostra o tempo de execução dos algoritmos executados no simulador.

Tabela 6 – Tempo de execução dos algoritmos no simulador utilizando os valores de entrada A.

Algoritmo		Número de Núcleos Simulados				
		2	8	32	128	512
π FlowMT	Integração Binária - R	6:12.00	6:13.42	6:17.96	6:35.88	7:22.40
	Integração Binária - NR	0:59.19	0:57.77	1:00.16	1:04.38	1:30.85
	Produto Interno	1:32.08	1:35.18	1:41.70	1:57.51	4:04.91
	Multiplicação de Matrizes	3:21.30	3:31.95	3:45.66	3:54.54	4:59.83
	Fatoração L.U.	4:08.18	3:23.46	2:01.52	6:24.37	7:22.22
	FFT	4:17.29	4:21.47	4:27.71	4:45.40	5:20.43
	Quick Sort	10:35.18	10:25.06	11:29.87	14:40.65	18:29.45
OpenMP	Integração Binária - R	7:07.19	7:05.75	7:13.33	7:22.47	7:32.51
	Integração Binária - NR	1:08.27	1:03.27	1:03.35	1:00.90	1:09.12
	Produto Interno	1:36.69	1:40.96	1:47.55	2:00.57	2:13.98
	Multiplicação de Matrizes	6:40.80	7:05.10	6:54.89	7:07.04	7:11.35
	Fatoração L.U.	3:09.64	3:12.50	2:50.42	3:23.31	6:17.46
	FFT	4:25.92	4:29.23	4:40.12	4:59.44	5:48.31
	Quick Sort	9:22.31	9:50.312	10:07.53	11:57.99	15:09.23
T.B.B	Integração Binária - R	6:12.13	6:13.09	6:19.57	6:27.22	6:37.12
	Integração Binária - NR	1:10.13	1:19.21	1:22.90	1:27.95	1:33.36
	Produto Interno	1:32.70	1:37.16	1:40.90	1:59.86	3:21.16
	Multiplicação de Matrizes	6:31.11	6:54.74	6:50.09	6:58.95	7:18.94
	Fatoração L.U.	3:09.14	3:37.73	5:08.56	12:45.53	20:27.55
	FFT	5:03.77	5:05.54	5:28.69	6:06.99	7:18.44
	Quick Sort	10:39.95	12:42.95	26:07.91	1:20:11	2:42:02
OmpSs	Integração Binária - R	8:43.29	9:14.65	10:11.89	23:42.40	1:43:03
	Integração Binária - NR	1:46.13	2:07.34	6:36.34	30:36.83	2:05:15
	Produto Interno	3:04.16	10:47.44	41:28.46	2:50:32	5:18:50
	Multiplicação de Matrizes	6:32.21	12:55.03	51:20.09	3:47:44	14:12:22
	Fatoração L.U.	4:52.98	13:18.42	58:16.94	4:07:30	16:41:21
	FFT	1:26.59	2:36.75	7:32.83	29:31.87	1:56:29
	Quick Sort	6:21.86	8:42.19	28:11.02	2:00:13	8:02:26

Fonte: Elaborada pelo autor.

A comparação entre os compiladores foi omitida, já que os resultados utilizando apenas os compiladores g++ e msvc++ demonstraram o mesmo padrão encontrado durante a execução na máquina real.

5.2.1 Speedup

Seguindo a metodologia adotada para a execução dos algoritmos no simulador os algoritmos foram executados simulando um processador com um cache similar ao cache dos

processadores atuais e uma execução considerando a implementação de um *cache* que siga a tendência de crescimento observada. Os resultados estão disponíveis nas figuras 28 e 29.

A comparação é feita primariamente com a comparação do *speedup*, já que é nesta métrica que a diferença deve ser mais evidente, já que para as entradas utilizadas o tamanho do *cache* não deve impactar de maneira significativa o tempo de execução.

Figura 28 – *Speedup* obtido com a execução dos algoritmos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*, enquanto que o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

Figura 29 – *Speedup* obtido com a execução dos algoritmos. Execução realizada no simulador com o *cache* projetado utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de núcleos simulados. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

Os resultados com a variação do *cache* não são tão distintos. A maior diferença acontece para um número elevado de núcleos, o que é esperado, pois há mais acessos à memória. Há também casos onde não há diferença significativa, como a integração binária, onde não há muitos acessos à memória.

Como os resultados não apresentam uma discrepância grande, o restante dos resultados será apresentado considerando apenas a simulação com o *cache atual*, já que é uma execução

mais pessimista, de modo que os resultados com qualquer aumento do *cache* devem ser melhores do que os obtidos.

5.2.2 Número de ciclos

O próximo passo é analisar o tempo de execução dos algoritmos, como visto na figura 30.

Figura 30 – Tempo de execução dos algoritmos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa o tempo de execução em número de ciclos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) Quick Sort; (k) Quick Sort - 2 Pivôs; (l) Bucket Sort.

Fonte: Elaborada pelo autor.

Com a execução no simulador com um número maior de *threads* é possível identificar algumas tendências de maneira mais clara. Um exemplo nítido é a utilização das técnicas de *work-stealing* presentes no T.B.B. e no OmpSs, observe que ambas as variações do *quick sort* são mais lentas nesses modelos. Isso acontece, pois esse algoritmo tem como característica ser duplamente recursivo e, como estes modelos possuem múltiplas filas de tarefas, é necessário que as técnicas de *work-stealing* sejam aplicadas muitas vezes, já que o algoritmo começa com apenas uma tarefa em uma única *thread* de modo que as divisões do *quick sort* são mantidas nesta mesma fila, o que força as outras *threads* a *roubarem* as tarefas desta primeira *thread*.

Um outro fator importante é que os algoritmos utilizados não possuem um *tamanho de problema* suficiente para explorar o paralelismo de processadores com muitos núcleos. Observe por exemplo a execução do produto interno, note que o desempenho de todos os modelos é suficientemente próximo, mas a partir de 64 núcleos simulados não é possível ganhar desempenho. Alguns algoritmos conseguem saturar o processador em modelos diferentes, como o caso da integração binária não recursiva com $f(x) = J_{\pi}(x)$, que consegue saturar o processador com o OpenMP e o T.B.B. enquanto que com o π FlowMT o ganho de desempenho para de acontecer com 256 *threads* e que com o OmpSs é ainda mais cedo com 16 *threads*.

É também importante observar que para a execução em um regime *many-core* o π FlowMT é o modelo mais constante, sendo muitas vezes o melhor modelo de execução.

A análise da integração binária com $f(x) = x$ é importante, pois com ela é possível notar que os modelos de estado de arte são bastante prejudicados com a execução de programas com granularidade fina no regime *many-core*, o que indica que estes modelos não estão otimizados para serem executados nestes ambientes.

Ainda é possível notar que os resultados obtidos com a execução no simulador seguem a mesma tendência da execução na máquina real, no sentido de que o melhor modelo encontrado durante a execução na máquina real também é o melhor modelo para as execuções do simulador.

5.2.3 Balanceamento de carga

Seguindo com a metodologia empregada, agora será analisado o balanceamento de carga entre os múltiplos núcleos simulados.

A execução dos algoritmos no simulador permite que a obtenção do número de ciclos de cada núcleo presente no simulador e portanto é possível determinar se as tarefas foram divididas de maneira igualitária, isto é, se cada núcleo simulado executou o mesmo número de ciclos, como visto na figura 31.

Esta análise é bastante interessante, o primeiro ponto importante a ser observado nos gráficos da figura 31 é o comportamento do OpenMP: em boa parte dos casos este foi o modelo com o maior desbalanceamento, mas de maneira não intuitiva o desbalanceamento acontece quando o modelo é executado com um número baixo de *threads* e depois tem um

Figura 31 – Razão da variação entre o número de ciclos por núcleo simulado e o número total de ciclos. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

comportamento de queda. Além do OpenMP também é possível observar esse comportamento em um número menor de casos no T.B.B..

Os picos observados nos outros modelos são bem menores do que os citados anteriormente e, de maneira geral, o balanceamento de carga destes modelos é bastante eficiente.

Já o π FlowMT demonstrou possuir o melhor escalonador: não só foi o mais constante,

como foi o modelo com as menores variações.

5.2.4 Overhead

O objetivo desta etapa é avaliar se o número total de ciclos — isto é, a soma do número de ciclos de todos os núcleos — não varia com o aumento do número de *threads*, como visto na figura 32.

O primeiro item observado na análise do *overhead* é o desempenho do OmpSs: em todos os casos o OmpSs teve valores muito elevados. O resultado permite duas conclusões: a primeira é de que o modelo não está otimizado para ser executado em um ambiente com muitas *threads* e a segunda é de que o compilador próprio utilizado pelo modelo é incapaz de otimizar os códigos de maneira eficiente.

Já o *overhead* observado no π FlowMT acontece devido aos tempos de comunicação do processo que torna uma saída em uma entrada para outro *codelet*, pois as *threads* permanecem em um estado de espera ocupada. Esse impacto é ressaltado com a execução de programas onde a granularidade é pequena, já que nesses casos o impacto dos custos de comunicação é maior.

O OpenMP foi o modelo com os menores índices de *overhead*, com a única exceção acontecendo com a execução do produto interno, que também é um algoritmo de granularidade fina.

Por sua vez o T.B.B. exibiu um *overhead* elevado para um tipo de algoritmo bastante comum: o que utiliza recursão, mas isto é novamente explicado pela utilização das técnicas de *work-stealing* necessárias para executar estes algoritmos.

5.3 Eficiência do Quick Sort

Um resultado que se destaca negativamente é o *speedup* obtido com a execução do *quick sort*. Este algoritmo é do tipo duplo recursivo e por isso gera uma grande quantidade de paralelismo com o decorrer de sua execução, todavia esta grande quantidade de paralelismo não consegue ser *convertida* em *speedup*.

Logo a análise do algoritmo torna-se fundamental para entender a limitação de escalabilidade observada, pois, se a análise determinar que o algoritmo deveria possuir uma escalabilidade maior do que a obtida será necessário analisar a origem do fator que limita a escalabilidade obtida.

O cálculo teórico da eficiência do *quick sort* pode ser visto no apêndice D. A comparação entre a função encontrada e os resultados obtidos pode ser visto na figura 33.

O resultado encontrado, além de indicar que as aproximações realizadas nos cálculos

Figura 32 – *Overhead* obtido com a execução dos algoritmos no simulador. Execução realizada no simulador com o *cache* dos processadores atuais utilizando o conjunto de entradas **A**. O eixo das abscissas representa o número de *threads*. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

são válidas, permite a demonstração de outros dois fatos:

- O algoritmo *quick sort*, embora possua uma grande quantidade de paralelismo, não é um bom algoritmo para ser utilizado no ambiente paralelo. Isso acontece porque o tamanho de cada nova tarefa é estritamente menor que o da tarefa anterior;

Figura 33 – Eficiência do *quick sort* com a execução no ambiente real (entradas **B**) e no simulado (entradas **A**) comparada com a eficiência teórica. O eixo das abscissas representa o número de *threads*, já o eixo das ordenadas representa a eficiência de execução. (a) Máquina Real; (b) Simulador.

Fonte: Elaborada pelo autor.

- π FlowMT é capaz de extrair todo o paralelismo disponível no algoritmo.

ANÁLISE E PERSPECTIVAS

Os resultados obtidos suportam as hipóteses apresentadas como base para o desenvolvimento da pesquisa:

A utilização de fluxos de dados como alternativa para a extração da capacidade de processamentos dos processadores *multi-core* foi bastante efetiva. Para os casos analisados é possível afirmar que a utilização de um modelo de execução é tão ou mais efetiva que a utilização dos outros modelos de execução analisados. O que não só confirma a viabilidade de se utilizar o fluxo de dados como também encoraja o uso do mesmo.

Também é possível afirmar que o modelo desenvolvido e implementado é de propósito geral e compatível com os processadores *multi-core* e *many-core*:

- A implementação de diversos algoritmos distintos demonstram que o modelo é flexível e consegue aproveitar das características dos diversos algoritmos, permitindo que o paralelismo presente nos algoritmos seja aproveitado de maneira eficiente — um ponto importante é notado com a análise da eficiência do *quick sort*: o impacto do algoritmo na eficiência é notório, o que indica que mesmo com o melhor modelo de execução possível, em certos casos não é possível extrair toda a capacidade de processamento do processador;
- Como o modelo foi testado em ambientes com até 2.048 núcleos é possível afirmar que o modelo é de fato compatível com estes processadores, já que todas as execuções foram bem sucedidas;

Já a comparação entre o modelo com os modelos de estado de arte é bastante positiva. O modelo desenvolvido não só foi bastante competitivo em relação aos modelos comparados como foi melhor em diversos cenários, principalmente quando é analisado o comportamento dos modelos durante a execução em um ambiente com centenas e até mesmo milhares de

núcleos. Um ponto importante que foi ressaltado anteriormente é o impacto da granularidade no modelo desenvolvido e, de fato, os resultados confirmam esta teoria e demonstram que os efeitos da granularidade são importantes não só para o modelo desenvolvido como também para os modelos de estado de arte analisados.

Há ainda de se ressaltar que os resultados obtidos comparam um modelo ainda em desenvolvimento com modelos de execução já estabelecidos e que estão em desenvolvimento por um longo período de tempo — a primeira versão do OpenMP foi lançada em 1997, do TBB em 2006 e do OmpSs em 2011 — e passaram por uma série de otimizações que ainda não aconteceram com o π FlowMT, o que torna o desempenho obtido com o modelo desenvolvido ainda mais promissor.

Uma segunda contribuição desta pesquisa é a estrutura de dados *codelet*. Parte do sucesso do π FlowMT é oriunda da utilização desta estrutura, que separa os códigos em segmentos de código, e como esta estrutura não depende do ambiente criado para o π FlowMT é possível que a mesma seja utilizada em outros contextos.

O desenvolvimento do π FlowMT ainda possui lacunas a serem preenchidas futuramente, como visto a seguir:

- **Otimizações:** o modelo foi levemente otimizado, mas ainda há espaço para maiores otimizações e especializações;
- **Escalonador dinâmico:** implementar um escalonador que, de maneira dinâmica, analise o tamanho dos *codelets* executados e tente aglutinar os mesmos caso a granularidade deles não seja grossa o suficiente para manter a eficiência desejada;
- **Facilitação do uso para usuários finais:** desenvolvimento de um manual de utilização e simplificação da sintaxe;
- **Automatização:** implementação de uma ferramenta que permita que os *codelets* sejam gerados de maneira mais fácil, ou até mesmo, de maneira automática;
- **Heterogeneização:** permitir e incentivar a utilização do modelo com outros elementos processantes, como GPUs e FPGAs.

Um artigo com os resultados apresentados nesta tese foi submetido e encontra-se na fase final de correção.

REFERÊNCIAS

- 1 DONGARRA, J.; LUSZCZEK, P. TOP500. *In*: DONGARRA, J.; LUSZCZEK, P. (ed.). **Encyclopedia of Parallel Computing**. Boston: Springer, 2011. p. 2055–2057. ISBN 978-0-387-09766-4. DOI 10.1007/978-0-387-09766-4_157.
- 2 DONGARRA, J. The linpack benchmark: an explanation. *In*: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1987, Berlin. **Proceedings** [...]. Berlin, Heidelberg: Springer-Verlag, 1987. p. 456–474.
- 3 DENNIS, J. B.; BOUGHTON, G. A.; LEUNG, C. K. Building blocks for data flow prototypes. *In*: ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 7., 1980, New York. **Proceedings** [...]. New York: ACM, 1980. p. 1–8. DOI 10.1145/800053.801902.
- 4 ACKERMAN, W. B.; DENNIS, J. B. **Val**: a value-oriented algorithmic language, preliminary reference manual. Massachusetts: Massachusetts Institute of Technology, 1979.
- 5 WATSON, I.; GURD, J. A practical data flow computer. **Computer**, v. 15, n. 2, p. 51–57, 1982.
- 6 BURGER, D.; KECKLER, S. W.; MCKINLEY, K. S.; DAHLIN, M.; JOHN, L. K.; LIN, C.; MOORE, C. R.; BURRILL, J.; MCDONALD, R. G.; YODER, W. Scaling to the end of silicon with edge architectures. **Computer**, v. 37, n. 7, p. 44–55, 2004.
- 7 GEBHART, M. *et al.* **An evaluation of the TRIPS computer system**. 2009. Disponível em: <https://www.cs.utexas.edu/ftp/dburger/papers/ASPLOS09.pdf>. Acesso em: 23 jan. 2019.
- 8 PELL, O.; MENCER, O.; TSOI, K. H.; LUK, W. Maximum performance computing with dataflow engines. *In*: VANDERBAUWHEDE, W.; BENKRID, K. (ed.). **High-Performance Computing Using FPGAs**. New York: Springer, 2013. p. 747–774.
- 9 FLYNN, M. J.; PELL, O.; MENCER, O. Dataflow supercomputing. *In*: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 22., 2012, Oslo **Proceedings** [...]. Oslo: IEEE, 2012. p. 1–3.
- 10 GIORGI, R. **TERAFLUX**: exploiting dataflow parallelism in teradevices. *In*: CONFERENCE ON COMPUTING FRONTIERS, 9., 2012. p. 303–304. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.446.4447&rep=rep1&type=pdf>. Acesso em: 23 jan. 2019.
- 11 PORTERO, A.; YU, Z.; GIORGI, R. Teraflux: exploiting tera-device computing challenges. **Procedia Computer Science**, v. 7, p. 146–147, 2011. DOI 10.1016/j.procs.2011.09.081.
- 12 SWANSON, S.; MICHELSON, K.; SCHWERIN, A.; OSKIN, M. **WaveScalar**. *In*: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 36., 2003. p. 291. Disponível em: <https://dl.acm.org/doi/pdf/10.5555/956417.956546?download=true>. Acesso em: 23 jan. 2019.

- 13 SWANSON, S.; SCHWERIN, A.; MERCALDI, M.; PETERSEN, A.; PUTNAM, A.; MICHELSON, K.; OSKIN, M.; EGGERS, S. J. The wavescalar architecture. **ACM Transactions on Computer Systems (TOCS)**, v. 25, n. 2, p. 4, 2007.
- 14 GRAMA, A. **Introduction to parallel computing**. London: Pearson Education, 2003.
- 15 KYRIACOU, C.; EVRIPIDOU, P.; TRANCOSO, P. Data-driven multithreading using conventional microprocessors. **IEEE Transactions on Parallel and Distributed Systems**, v. 17, n. 10, p. 1176–1188, 2006.
- 16 CICOTTI, P.; BADEN, S. B. Latency hiding and performance tuning with graph-based execution. *In*: WORKSHOP ON DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, 1., 2011, Galveston Island. **Proceedings [...]**. Galveston Island: IEEE, 2011. p. 28–37.
- 17 OROZCO, D.; GARCIA, E.; PAVEL, R.; KHAN, R.; GAO, G. Tideflow: The time iterated dependency flow execution model. *In*: WORKSHOP ON DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, 1., 2011, Galveston Island. **Proceedings [...]**. Galveston Island: IEEE, 2011. p. 1–9.
- 18 KULKARNI, A.; LANG, M.; LUMSDAINE, A. **GoDEL**: a multidirectional dataflow execution model for large-scale computing. 2011. p. 10–18. Disponível em: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6176400>. Acesso em: 23 jan. 2019.
- 19 BRAVO, M.; LI, Z.; ROY, P. V.; MEIKLEJOHN, C. Derflow: Distributed deterministic dataflow programming for erlang. *In*: ACM SIGPLAN WORKSHOP ON ERLANG, 30., 2014, New York. **Proceedings [...]**. New York: ACM, 2014. p. 51–60. DOI 10.1145/2633448.2633451.
- 20 BORKAR, S. Design challenges of technology scaling. **IEEE Micro**, v. 19, n. 4, p. 23–29, 1999.
- 21 BELL, J.; CASASENT, D.; BELL, C. G. An investigation of alternative cache organizations. **IEEE Transactions on Computers**, v. 100, n. 4, p. 346–351, 1974.
- 22 BONCZ, P. Processor cache. *In*: LIU L., ÖZSU M.T. (ed.). **Encyclopedia of database systems**. Boston: Springer, 2009. p. 2195–2196 DOI 10.1007/978-0-387-39940-9_684.
- 23 GIANNAKAS, T.; SERMPEZIS, P.; SPYROPOULOS, T. Show me the cache: Optimizing cache-friendly recommendations for sequential content access. *In*: INTERNATIONAL SYMPOSIUM ON A WORLD OF WIRELESS, MOBILE AND MULTIMEDIA NETWORKS (WoWMoM), 19., 2018, Chania, Greece. **Proceedings [...]**. Chania, Greece: IEEE, 2018 DOI 10.1109/wowmom.2018.8449731.
- 24 SPRANGLE, E.; CARMEAN, D. Increasing processor performance by implementing deeper pipelines. *In*: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 29., 2002, Anchorage. **Proceedings [...]**. Anchorage: IEEE, 2002. p. 25–34.
- 25 HARTSTEIN, A.; PUZAK, T. R. The optimum pipeline depth for a microprocessor. **ACM Sigarch Computer Architecture News**, v.30, n.2, p. 7-13, 2002.
- 26 YEAGER, K. C. The mips r10000 superscalar microprocessor. **IEEE micro**, v. 16, n. 2, p. 28–41, 1996.

- 27 MUTLU, O.; STARK, J.; WILKERSON, C.; PATT, Y. N. Runahead execution: an alternative to very large instruction windows for out-of-order processors. *In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE*, 9., 2003, Anaheim. **Proceedings** [...]. Anaheim: IEEE, 2003. p. 129–140.
- 28 EVERS, M.; PATEL, S. J.; CHAPPELL, R. S.; PATT, Y. N. An analysis of correlation and predictability: what makes two-level branch predictors work. *In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, 25., 1998, Barcelona. **Proceedings** [...]. Barcelona: IEEE, 1998. p. 52–61.
- 29 FLYNN, M. Flynn's taxonomy. *In: PADUA, D. (ed.). Encyclopedia of Parallel Computing*. Boston: Springer, 2011. p. 689–697. DOI 10.1007/978-0-387-09766-4_2.
- 30 SCHWARZMEIER, J. L. Cray vector computers. *In: PADUA, D. (ed.). Encyclopedia of Parallel Computing*. Boston: Springer, 2011. p. 441–453. DOI 10.1007/978-0-387-09766-4_479.
- 31 KOZYRAKIS, C.; PATTERSON, D. Overcoming the limitations of conventional vector processors. *In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, 30., 2003, San Diego. **Proceedings** [...]. San Diego: IEEE, 2003. p. 399–409.
- 32 ETIEMBLE, D. **45-year CPU evolution: one law and two equations**. 2018. Disponível em: <https://arxiv.org/ftp/arxiv/papers/1803/1803.00254.pdf>. Acesso em: 23 jan. 2019.
- 33 AGARWAL, A.; LEVY, M. The kill rule for multicore. *In: ACM/IEEE DESIGN AUTOMATION CONFERENCE*, 44., 2007, San Diego. **Proceedings** [...]. San Diego: IEEE, 2007. p. 750–753.
- 34 KOWARSCHIK, M.; WEISS, C. An overview of cache optimization techniques and cache-aware numerical algorithms. *In: MEYER, U.; SANDERS, P.; SIBEYN, J. (ed.). Algorithms for memory hierarchies*. Berlin: Springer, 2003. p. 213–232. DOI 10.1007/3-540-36574-5_10.
- 35 UNGERER, T.; ROBIČ, B.; ŠILC, J. Multithreaded processors. **Computer Journal**, v. 45, n. 3, p. 320–348, 2002.
- 36 SINHARROY, B.; NORSTRAND, J. A. V.; EICKEMEYER, R. J.; LE, H. Q.; LEENSTRA, J.; NGUYEN, D. Q.; KONIGSBURG, B.; WARD, K.; BROWN, M. D.; MOREIRA, J. E.; LEVITAN, D.; TUNG, S.; HRUSECKY, D.; BISHOP, J. W.; GSCHWIND, M.; BOERSMA, M.; KROENER, M.; KALTENBACH, M.; KARKHANIS, T.; FERNSLER, K. M. IBM POWER8 processor core microarchitecture. **IBM Journal of Research and Development**, IBM, v. 59, n. 1, p. 2:1–2:21, 2015. DOI 10.1147/jrd.2014.2376112.
- 37 LI, Z.; XU, X.; HU, W.; TANG, Z. Microarchitecture and performance analysis of godson-2 smt processor. *In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN*, 2006, San José. **Proceedings** [...]. San José: IEEE, 2006. p. 485–490. DOI 10.1109/ICCD.2006.4380860.
- 38 SINGH, A. K.; SHAFIQUE, M.; KUMAR, A.; HENKEL, J. Mapping on multi/many-core systems: survey of current and emerging trends. *In: ACM/EDAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC)*, 50., 2013, Austin. **Proceedings** [...]. Austin: IEEE, 2013. p. 1–10. DOI 10.1145/2463209.2488734.

- 39 CONTRERAS, G.; MARTONOSI, M. Characterizing and improving the performance of intel threading building blocks. *In*: INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, 2008, Seattle. **Proceedings** [...]. Seattle: IEEE, 2008. p. 57–66. DOI 10.1109/IISWC.2008.4636091.
- 40 DURAN, A.; AYGUADÉ, E.; BADIA, R. M.; LABARTA, J.; MARTINELL, L.; MARTORELL, X.; PLANAS, J. OmpSs: a proposal for programming heterogeneous multi-core architectures. **Parallel Processing Letters**, v. 21, n. 02, p. 173–193, 2011. DOI 10.1142/S0129626411000151.
- 41 SILVA JÚNIOR, J. T. **PIFLOW**: projeto, simulação e implementação de um protótipo dataflow em FPGA. 2016. 148 p. Dissertação (Mestrado em Ciências) — Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2016.
- 42 TEIXEIRA, J.; RUGGIERO, C. A.; MATIAS, P. π FlowMR: um protótipo baseado no modelo a fluxo de dados, escalável, implementado em um cluster de fpgas de baixo custo. *In*: SEMANA INTEGRADA DO INSTITUTO DE FÍSICA DE SÃO CARLOS — SIFSC, 9., 2019, São Carlos. **Resumos** [...]. São Carlos: IFSC, 2019.
- 43 ZUCKERMAN, S.; SUETTERLEIN, J.; KNAUERHASE, R.; GAO, G. R. Using a “codelet” program execution model for exascale machines: Position paper. *In*: INTERNATIONAL WORKSHOP ON ADAPTIVE SELF-TUNING COMPUTING SYSTEMS FOR THE EXAFLOP ERA, 11., 2011, San José. **Proceedings** [...]. New York: ACM, 2011. p. 64–69. DOI 10.1145/800053.801902.
- 44 DOLBEAU, R.; BIHAN, S.; BODIN, F. **HMPP**: a hybrid multi-core parallel programming environment. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0DF6AF8C258A6EC1E6D2053787F73AF0?doi=10.1.1.357.3627&rep=rep1&type=pdf>. Acesso em: 23 jan. 2019.
- 45 ANDREWS, G. R. **Concurrent programming**: principles and practice. San Francisco: Benjamin/Cummings Publishing Company, 1991.
- 46 VALOIS, J. D. **Lock-free data structures**. 1996. Ph. D. Thesis (Doctor) – Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1996.
- 47 DESROCHERS, C. **moodycamel::concurrentQueue**. Disponível em: <https://github.com/cameron314/concurrentqueue>. Acesso em: 23 jan. 2020.
- 48 DESROCHERS, C. **A performance benchmark of queue with std::mutex against boost::lockfree::queue and moodycamel::concurrentQueue**. Disponível em: https://github.com/Qihoo360/evpp/blob/master/docs/benchmark_lockfree_vs_mutex.md. Acesso em: 27 maio 2019.
- 49 DESROCHERS, C. **A fast general purpose lock-free queue for c++**. Disponível em: <http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++>. Acesso em: 27 maio 2019.
- 50 CULLER, D. E.; SINGH, J. P.; GUPTA, A. **Parallel computer architecture**: a hardware/software approach. San Francisco: Morgan Kaufmann Publishers, 1999.
- 51 KAO, M.-Y. **Encyclopedia of Algorithms**. 2nd. ed. Berlin: Springer, 2016.

- 52 IWASAKI, S.; TAURA, K. A static cut-off for task parallel programs. *In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPITALION-PACT*, 16., 2016, Haifa. **Proceedings** [...]. Haifa: ACM Press, 2016. DOI 10.1145/2967938.2967968.
- 53 CARLSON, T. E.; HEIRMAN, W.; EYERMAN, S.; HUR, I.; EECKHOUT, L. An evaluation of high-level mechanistic core models. **ACM Transactions on Architecture and Code Optimization (TACO)**, v. 2, n. 3, 2014. DOI 10.1145/2629677.
- 54 SANCHEZ, D.; KOZYRAKIS, C. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. **ACM SIGARCH Computer Architecture News**, v. 41, n. 3, p. 475–486, 2013.
- 55 BINKERT, N.; BECKMANN, B.; BLACK, G.; REINHARDT, S. K.; SAIDI, A.; BASU, A.; HESTNESS, J.; HOWER, D. R.; KRISHNA, T.; SARDASHTI, S.; SEN, R.; SEWELL, K.; SHOAB, M.; VAISH, N.; HILL, M. D.; WOOD, D. A. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, v.39, n.2, 2011. DOI 10.1145/2024716.2024718.
- 56 WENISCH, T. F.; WUNDERLICH, R. E.; FERDMAN, M.; AILAMAKI, A.; FALSAFI, B.; HOE, J. C. Simflex: Statistical sampling of computer system simulation. **IEEE Micro**, v. 26, n. 4, p. 18–31, 2006.
- 57 PATEL, A.; AFRAM, F.; CHEN, S.; GHOSE, K. Marss: A full system simulator for multicore x86 cpus. *In: ACM/EDAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC)*, 48. 2011, New York. **Proceedings** [...]. New York: IEEE., 2011., p. 1050–1055.
- 58 MILLER, J. E.; KASTURE, H.; KURIAN, G.; GRUENWALD, C.; BECKMANN, N.; CELIO, C.; EASTEP, J.; AGARWAL, A. Graphite: a distributed parallel simulator for multicores. *In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER*, 16., 2010, Bangalore, India. **Proceedings** [...]. Bangalore, India: IEEE, 2010.
- 59 GENBRUGGE, D.; EYERMAN, S.; EECKHOUT, L. Interval simulation: raising the level of abstraction in architectural simulation. *In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER*, 16., 2010, Bangalore, India. **Proceedings** [...]. Bangalore, India: IEEE, 2010.
- 60 WOO, S. C.; OHARA, M.; TORRIE, E.; SINGH, J. P.; GUPTA, A. The SPLASH-2 programs: characterization and methodological considerations. **ACM SIGARCH Computer Architecture News**, v.23, n.2, 1995. DOI 10.1145/225830.223990.
- 61 LUK, C.-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. **ACM SIGPLAN Notices**, v. 40, n. 6, p. 190–200, 2005. DOI 10.1145/1064978.1065034.
- 62 KEJARIWAL, A.; VEIDENBAUM, A. V.; NICOLAU, A.; TIAN, X.; GIRKAR, M.; SAITO, H.; BANERJEE, U. Comparative architectural characterization of spec CPU2000 and CPU2006 benchmarks on the Intel® Core™ 2 Duo processor. *In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING, AND SIMULATION*, 2008, Samos, Greece. **Proceedings** [...]. Samos, Greece, 2008. p. 456–474. DOI 10.1109/ICSAMOS.2008.4664832.

- 63 INTEL. **Xeon L5640 specifications**. Disponível em: [https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20L5640%20-%20AT80614005133AB%20\(BX80614L5640\).html](https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20L5640%20-%20AT80614005133AB%20(BX80614L5640).html). Acesso em: 27 maio 2019.
- 64 AMD. **Ryzen Threadripper 2990WX specifications**. Disponível em: <http://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%20Threadripper%202990WX.html>. Acesso em: 27 maio 2019.
- 65 DEBIAN. **The unstable distribution (sid)**. Disponível em: <https://www.debian.org/releases/sid/>. Acesso em: 27 maio 2019.
- 66 **A list of compilers**. Disponível em: https://en.wikipedia.org/wiki/List_of_compilers#C++_compilers. Acesso em: 27 maio 2019.
- 67 KASLIWAL, V. **A performance-based comparison of C/C++ compilers**. 2017. Disponível em: <https://colfaxresearch.com/compiler-comparison>. Acesso em: 27 maio 2019.
- 68 GNU. **GCC 8 releas series changes, new features, and fixes**. Disponível em: <https://gcc.gnu.org/gcc-8/changes.html>. Acesso em: 27 maio 2019.
- 69 INTEL. **Intel C++ compiler-19.0 for linux release notes for intel parallel studio XE 2019**. Disponível em: <https://software.intel.com/en-us/articles/intel-c-compiler-190-for-linux-release-notes-for-intel-parallel-studio-xe-2019>. Acesso em: 27 maio 2019.
- 70 CLANG. **Clang 7.0.0 release notes**. Disponível em: <https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html>. Acesso em: 27 maio 2019.
- 71 TOOLS, P. C. . **PGI version 18.4 documentation for x86 and NVIDIA processors**. Disponível em: <https://www.pgroup.com/resources/docs/18.4/x86/index.htm>. Acesso em: 27 maio 2019.
- 72 AMD. **AMD optimizing C/C++ compiler**. Disponível em: <https://developer.amd.com/amd-aocc>. Acesso em: 27 maio 2019.
- 73 ZAPCC. **Zapcc**. Disponível em: <https://www.zapcc.com>. Acesso em: 27 maio 2019.
- 74 CENTER, B. S. **Mercurium C/C++/Fortran source-to-source compiler**. Disponível em: <https://github.com/bsc-pm/mcxx>. Acesso em: 27 maio 2019.
- 75 MUIR, J. **Using the rdtsc instruction for performance monitoring**. 1997. Disponível em: <https://www.ccsf.carleton.ca/~jamuir/rdtscpm1.pdf>. Acesso em: 23 jan. 2019.
- 76 SUN, X.-H.; NI, L. M. Another view on parallel speedup. *In*: CONFERENCE ON SUPERCOMPUTING, 90., 1990, New York. **Proceedings** [...]. New York: IEEE, 1990. p. 324–333.
- 77 CAYLEY, A. A memoir on the theory of matrices. **Philosophical Transactions of the Royal Society of London**, v. 148, p. 17–37, 1858. DOI 10.1098/rstl.1858.0002.
- 78 PAN, V. How can we speed up matrix multiplication? **SIAM Review**, v. 26, n. 3, p. 393–415, 1984. DOI 10.1137/1026076.
- 79 GIBBS, J.; WILSON, E. **Vector Analysis**. C. Scribner's sons, 1901. (Yale bicentennial publications). Disponível em: <https://archive.org/details/117714283>. Acesso em: 29 jan. 2020.

- 80 KOBBELT, L. A fast dot-product algorithm with minimal rounding errors. **Computing**, v. 52, n. 4, p. 355–369, 1994. DOI 10.1007/BF02276882.
- 81 TURING, A. M. Rounding-off errors in matrix processes. **Quarterly Journal of Mechanics and Applied Mathematics**, v. 1, n. 1, p. 287–308, 01 1948. DOI 10.1093/qjmam/1.1.287.
- 82 LIU, Z.; CHEUNG, D. Efficient parallel algorithm for dense matrix LU decomposition with pivoting on hypercubes. **Computers & Mathematics with Applications**, v. 33, n. 8, p. 39–50, 1997. DOI 10.1016/s0898-1221(97)00052-7.
- 83 KUNCIR, G. F. Algorithm 103: Simpson's rule integrator. **Communications of the ACM**, v. 5, n. 6, p. 347, 1962. DOI 10.1145/367766.368179.
- 84 GAUSS, C. F. Nachlass: theoria interpolationis methodo nova tractata. *In: Werke*. Göttingen: Königliche Gesellschaft der Wissenschaften, 1866. v. 3, p. 265–330. Disponível em: <https://archive.org/stream/werkecarlf03gausrich#page/n277/mode/2up>. Acesso em: 29 jan. 2020.
- 85 COOLEY, J. W.; TUKEY, J. W. An algorithm for the machine calculation of complex fourier series. **Mathematics of Computation**, v. 19, n. 90, p. 297–297, 1965. DOI 10.1090/s0025-5718-1965-0178586-1.
- 86 LECLERC, G.-L.; BUFFON, C. D. Essai d'arithmétique morale. **Supplément à l'Histoire Naturelle**, v. 4, 1777.
- 87 HWANG, C.-O.; KIM, Y.; IM, C.; LEE, S. Buffon's needle algorithm to estimate π . **Applied Mathematics**, v. 08, n. 03, p. 275–279, 2017. DOI 10.4236/am.2017.83022.
- 88 HOARE, C. A. R. Quicksort. **Computer Journal**, v. 5, n. 1, p. 10–16, 1962. DOI 10.1093/comjnl/5.1.10.
- 89 YAROSLAVSKIY, V. **Dual-pivot quicksort algorithm, 2009**. Disponível em: <http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf>. Acesso em: 05 maio 2019.
- 90 FAUJDAR, N.; SARASWAT, S. The detailed experimental analysis of bucket sort. *In: INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, DATA SCIENCE & ENGINEERING-CONFLUENCE, 7., 2017, Noida. Proceedings [...]*. Noida: IEEE, 2017. DOI 10.1109/confluence.2017.7943114.

ALGORITMOS

Os pseudocódigos de cada algoritmo implementado são apresentados a seguir:

- **Multiplicação de Matrizes:** algoritmo que realiza a multiplicação de duas matrizes quadradas de tamanho N com a atribuição do resultado a uma terceira matriz (77, 78), como visto no algoritmo 1;

Algoritmo 1: Algoritmo para multiplicação de matrizes quadradas.

Input: A, B
Output: C

```

1 for  $i \leftarrow 0$  to  $N - 1$  do
2   for  $j \leftarrow 0$  to  $N - 1$  do
3     for  $k \leftarrow 0$  to  $N - 1$  do
4        $C[i][j] \leftarrow A[i][j] \times B[j][k]$ 
```

- **Produto Interno:** exemplificado no algoritmo 2, realiza o produto interno entre dois vetores de tamanho N ; (79, 80)

Algoritmo 2: Algoritmo para a realização do produto interno entre dois vetores.

Input: A, B
Output: r

```

1  $r \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $N - 1$  do
3    $r \leftarrow r + A[i] \times B[i]$ 
```

- **Fatoração L.U.:** decompõe uma matriz em um produto de uma matriz triangular inferior e triangular superior (81, 82) — realizado *in-place*, disponível no algoritmo 3;

Algoritmo 3: Algoritmo de fatoração L.U..**Input:** A **Output:** A

```

1 for  $k \leftarrow 1$  to  $N - 1$  do
2   for  $i \leftarrow k + 1$  to  $N - 1$  do
3      $A[i][k] \leftarrow A[i][k] / A[k][k]$ 
4     for  $j \leftarrow k + 1$  to  $N - 1$  do
5        $A[i][j] \leftarrow A[i][j] - A[i][k] \times A[k][j]$ 

```

- **Integração Binária:** integra uma função arbitrária em um intervalo determinado. Pode ser feito de maneira recursiva (algoritmo 4) ou não-recursiva (algoritmo 5); (83)

Algoritmo 4: Algoritmo recursivo para o cálculo da integral da função $f(x)$.**Input:** $x_i, x_f, \Delta x$ **Output:** I

```

1 Function integral( $a, b, \Delta x$ ):
2   if  $(b - a) > \Delta x$  then
3      $m \leftarrow (a + b) / 2$ 
4     return integral( $a, m, \Delta x$ ) + integral( $m, b, \Delta x$ )
5   else
6     return  $(b - a) \times (f(a) + f(b)) / 2$ 
7 Function main():
8    $I \leftarrow$  integral( $x_i, x_f, \Delta x$ )

```

Algoritmo 5: Algoritmo não-recursivo para o cálculo da integral da função $f(x)$.**Input:** $x_i, x_f, \Delta x$ **Output:** I

```

1 Function integral( $x_i, x_f, \Delta x$ ):
2    $r \leftarrow 0$ 
3    $min \leftarrow x_i$ 
4    $max \leftarrow x_i + \Delta x$ 
5   do
6      $r \leftarrow r + (max - min) \times (f(min) + f(max)) / 2$ 
7      $min \leftarrow max$ 
8      $max \leftarrow max + \Delta x$ 
9   while  $max < x_f$ 
10 Function main():
11    $I \leftarrow$  integral( $x_i, x_f, \Delta x$ )

```

- **Fast Fourier Transform (FFT):** implementação da transformada rápida de Fourier utilizando o algoritmo de Cooley-Tukey (84, 85), exemplificada no algoritmo 6;

Algoritmo 6: pseudocódigo do algoritmo de Cooley-Tukey para a FFT.

```

Input:  $x, N$ 
Output:  $w$ 
1 Function separate( $a, n$ ):
2    $m \leftarrow n/2$ 
3    $b[m]$ 
4   for  $i \leftarrow 0$  to  $m$  do
5      $b[i] \leftarrow a[i * 2 + 1]$ 
6      $a[i] \leftarrow a[i * 2]$ 
7      $a[i + m] \leftarrow b[i]$ 
8 Function fft( $w, n$ ):
9   if  $n > 1$  then
10     $m \leftarrow n/2$ 
11    separate( $w, n$ )
12    fft( $w, m$ )
13    fft( $w + m, m$ )
14    for  $i \leftarrow 0$  to  $m$  do
15       $e \leftarrow w[i]$ 
16       $o \leftarrow w[i + m]$ 
17       $f \leftarrow e^{-2i\pi/n} \times o$ 
18       $w[i] \leftarrow e + f$ 
19       $w[i + m] \leftarrow e - f$ 
20 Function main():
21    $w \leftarrow x$ 
22   fft( $w, N$ )

```

- **Agulha de Buffon:** este algoritmo utiliza a técnica de Monte Carlo para calcular a probabilidade de uma agulha cruzar a linha de uma folha de caderno ao ser jogada de modo aleatório (86, 87). O algoritmo foi implementado de modo a calcular a probabilidade da agulha cruzar a linha de uma folha de caderno em função do seu tamanho — *i.e.* $P(l)$, onde l é o tamanho da agulha, como visto no algoritmo 7;
- **Quick Sort:** popular algoritmo de ordenação que faz uso de recursividade para ordenar o vetor, foram utilizadas a versão tradicional (88) (algoritmo 8) e a versão com dois pivôs (89) (algoritmo 9);
- **Bucket Sort:** outro algoritmo de ordenação, é conhecido por ser mais *parallel-friendly* (90), sua implementação pode ser vista no algoritmo 10.

Algoritmo 7: Algoritmo para o cálculo da função $P(I)$ utilizando o método de Monte Carlo.

Input: $Size, N_{samples}, S_{min}, S_{max}, \Delta S$
Output: $P(S)$

```

1  $N_{needles} \leftarrow (S_{max} - S_{min}) / \Delta S$ 
2 for  $j \leftarrow 0$  to  $N_{needles}$  do
3    $needle \leftarrow S_{min} + j \times \Delta S$ 
4    $hit \leftarrow 0$ 
5   for  $i \leftarrow 0$  to  $N_{samples}$  do
6      $x_1 \leftarrow S \times \text{rand}(0, 1) + 1$ 
7      $y_1 \leftarrow S \times \text{rand}(0, 1) + 1$ 
8      $\theta \leftarrow 2 \cdot \pi \times \text{rand}(0, 1)$ 
9      $x_2 \leftarrow x_1 + needle \times \cos(\theta)$ 
10     $y_2 \leftarrow y_1 + needle \times \sin(\theta)$ 
11     $hit \leftarrow hit + (\text{ceil}(\max(y_1, y_2)) - \text{floor}(\min(y_1, y_2)) > 1)$ 
12   $P[j] \leftarrow hit / N_{samples}$ 

```

Algoritmo 8: Implementação com um pivô do algoritmo de ordenação *quick sort*.

Input: x, N
Output: x

```

1 Function quick_sort( $x, a, b$ ):
2    $i \leftarrow a$ 
3    $j \leftarrow b$ 
4    $center \leftarrow x[(a + b) / 2]$ 
5   if  $b > a$  then
6     while  $i \leq j$  do
7       while  $x[i] < center$  do
8          $i \leftarrow i + 1$ 
9       while  $x[j] > center$  do
10         $j \leftarrow j - 1$ 
11        if  $i \leq j$  then
12          swap( $x[i], x[j]$ )
13           $i \leftarrow i + 1$ 
14           $j \leftarrow j - 1$ 
15        if  $a < j$  then
16          quick_sort( $x, a, j$ )
17        if  $i < b$  then
18          quick_sort( $x, i, b$ )
19 Function main():
20   quick_sort( $x, 0, N$ )

```

Algoritmo 9: Implementação do algoritmo *quick sort* utilizando dois pivôs.

Input: x, N
Output: x

```

1 Function quick_sort( $x, a, b$ ):
2    $i \leftarrow a + 1$ 
3    $j \leftarrow b - 1$ 
4    $k \leftarrow i$ 
5   if  $b > a$  then
6     if  $x[a] > x[b]$  then
7        $\text{swap}(x[a], x[b])$ 
8      $p \leftarrow x[a]$ 
9      $q \leftarrow x[b]$ 
10    while  $k \leq j$  do
11      if  $x[k] < p$  then
12         $\text{swap}(x[k], x[i])$ 
13      else if  $x[k] \geq q$  then
14        while  $x[j] > q$  and  $k < j$  do
15           $j \leftarrow j - 1$ 
16         $\text{swap}(x[k], x[j])$ 
17         $j \leftarrow j - 1$ 
18        if  $x[k] < p$  then
19           $\text{swap}(x[k], x[i])$ 
20           $i \leftarrow i + 1$ 
21       $k \leftarrow k + 1$ 
22     $j \leftarrow j + 1$ 
23     $i \leftarrow i - 1$ 
24     $\text{swap}(x[a], x[i])$ 
25     $\text{swap}(x[b], x[j])$ 
26    if  $a < i - 1$  then
27       $\text{quick\_sort}(x, a, i - 1);$ 
28    if  $i + 1 < j - 1$  then
29       $\text{quick\_sort}(x, i + 1, j - 1);$ 
30    if  $j + 1 < b$  then
31       $\text{quick\_sort}(x, j + 1, b);$ 
32 Function main():
33    $\text{quick\_sort}(x, 0, N)$ 

```

Algoritmo 10: Algoritmo de ordenação *Bucket Sort*.

Input: x, x_{min}, x_{max}, N_b
Output: x

```

1 Function main():
2    $b[N_b]$ ;
3    $\Delta b \leftarrow x_{max} - x_{min} / N_b$ ;
4   for  $i \leftarrow 0$  to size ( $x$ ) do
5      $idx \leftarrow \text{floor}(x[i] / \Delta b)$ ;  $b[idx] \leftarrow x[i]$ ;
6   for  $i \leftarrow 0$  to  $N_b$  do
7     sort ( $b[i]$ );
8    $k \leftarrow 0$ ;
9   for  $i \leftarrow 0$  to  $N_b$  do
10    for  $j \leftarrow 0$  to size ( $b[i]$ ) do
11       $x[k] \leftarrow b[i][j]$ ;  $k \leftarrow k + 1$ ;

```

ENTRADAS

VALORES de entrada para os algoritmos executados.

Programa		Conjunto A	Conjunto B
Multiplicação de Matrizes	N	1.024	4.096
Produto Interno	N	33.554.432	134.217.728
	Corte Sequencial	32.768	32.768
	Redução Multiestágio	32	64
Fatoração L.U.	N	1.024	4.096
	Corte Sequencial	64	64
Integração Binária - R	δx	10^{-4}	$2,5 \times 10^{-5}$
	Redução multiestágio	8	8
Integração Binária - NR	N_{div}	2^{15}	2^{17}
	Redução Multiestágio	8	8
FFT	N	1.048.576	4.194.304
Bucket Sort	N	33.554.432	134.217.728
Quick Sort	N	33.554.432	134.217.728
	Corte Sequencial	1.024	1.024
Monte Carlo	N_i	10^4	10^6
	L_s	0.0	0.0
	L_f	10.0	10.0
	δ_L	0.01	0.01

Quadro 4 – Conjuntos de entradas utilizados durante as várias execuções.

Os programas de integração binária foram executados com duas funções diferentes, vistas nas equações B.1 e B.2.

$$I = \int_0^{\pi} x dx \quad (\text{B.1})$$

$$I = \int_0^\pi J_\pi(x) dx = \int_0^\pi \left[\frac{1}{\pi} \int_0^\pi \cos(\pi\tau - x \sin \tau) d\tau - \frac{\sin(\pi^2)}{\pi} \int_0^\infty e^{-x \sinh(t) - \pi t} dt \right] dx \quad (\text{B.2})$$

FLOPS

UMA métrica bastante utilizada para a comparação entre modelos é a que conta o número de operações de ponto flutuante realizadas em um segundo, *Floating point Operations Per Second* (FLOPS), e por causa desta importância esta métrica foi analisada no contexto desta pesquisa.

Para as execuções no simulador não foi possível a obtenção do número de operações de ponto flutuante, já que o simulador não separa as instruções de ponto flutuante das demais.

Já para a execução na máquina real foi utilizado o programa perf com a *tag stat*, que conta o número de execuções de um determinado evento.

O processador utilizado, AMD Threadripper 2990WX, possui dois tipos de eventos que são relacionados a aritmética de pontos flutuantes: um que conta o número de instruções e o outro que retorna diretamente o número de FLOPS, como visto a seguir:

- **Contadores:**

- RETIRED_MMX_FP_INSTRUCTIONS:SSE_INSTR
- RETIRED_MMX_FP_INSTRUCTIONS:MMX_INSTR
- RETIRED_MMX_FP_INSTRUCTIONS:X87_INSTR

- **FLOPS:**

- RETRED_SSE_AVX_OPERATIONS:DP_MULT_ADD_FLOPS
- RETRED_SSE_AVX_OPERATIONS:DP_DIV_FLOPS
- RETRED_SSE_AVX_OPERATIONS:DP_MULT_FLOPS
- RETRED_SSE_AVX_OPERATIONS:DP_ADD_SUB_FLOPS
- RETRED_SSE_AVX_OPERATIONS:SP_MULT_ADD_FLOPS

- RETRED_SSE_AVX_OPERATIONS:SP_DIV_FLOPS
- RETRED_SSE_AVX_OPERATIONS:SP_MULT_FLOPS
- RETRED_SSE_AVX_OPERATIONS:SP_ADD_SUB_FLOPS

Todavia os eventos do segundo tipo retornam sempre o valor zero. Testes puramente sintéticos foram desenvolvidos com o objetivo de entender e identificar a eventual presença de algum problema de compatibilidade com a execução do perf com os modelos de execução paralelas, mas a perfilação destes também não foi bem sucedida.

Os eventos que contam as instruções funcionam da maneira esperada, como visto na figura 34.

Entretanto a utilização destes eventos não permite que a obtenção do número de FLOPS de maneira coerente: como os eventos contam o número de instruções de determinado tipo é possível que as escolhas realizadas pelo compilador impactem de maneira distinta os diferentes modelos de execução.

Considere o caso extremo onde um mesmo código fonte é compilado utilizando diferentes compiladores, figura 35.

Note que existe uma diferença significativa entre o número de FLOPS entre as execuções de um mesmo programa, diferença esta que não é compatível com a diferença observada nos tempos de execução — vide subseção 5.1.1.

Portanto não é possível utilizar as ferramentas disponíveis para a obtenção desta métrica.

Figura 34 – Comparação do número de GFlops entre os diversos modelos de execução. Execução realizada utilizando o conjunto de entradas **B**. O eixo das ordenadas representa o número de GFlops, enquanto o eixo das abcissas representa o número de núcleos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) Quick Sort; (k) Quick Sort - 2 Pivôs; (l) Bucket Sort.

Fonte: Elaborada pelo autor.

Figura 35 – Comparação do número de GFlops com a variação do compilador. Execução realizada utilizando o conjunto de entradas **B**. O eixo das ordenadas representa o número de GFlops, enquanto o eixo das abcissas representa o número de núcleos. (a) Integração Binária - Recursiva - $f(x) = x$; (b) Integração Binária - Recursiva - $f(x) = J_{\pi}(x)$; (c) Integração Binária - Não recursiva - $f(x) = x$; (d) Integração Binária - Não recursiva - $f(x) = J_{\pi}(x)$; (e) Produto Interno; (f) Multiplicação de Matrizes; (g) Fatoração L.U.; (h) FFT; (i) Monte Carlo; (j) *Quick Sort*; (k) *Quick Sort* - 2 Pivôs; (l) *Bucket Sort*.

Fonte: Elaborada pelo autor.

EFICIÊNCIA DO QUICK SORT

CONSIDERE que a escolha dos pivôs é ótima, i.e., o pivô irá sempre dividir o vetor na metade, nesse caso, o tempo de execução sequencial já é conhecido e é $\mathcal{O}(N \log N)$ para N suficientemente grande.

Nos cálculos abaixo os tempos apresentados consideram apenas o fator dominante com N suficientemente grande, de modo que $t_{seq} = N \log N$.

Figura 36 – Divisões geradas pelo algoritmo do *quick sort*.

Fonte: Elaborada pelo autor.

Na execução paralela, a divisão através do pivô é feita até o número de folhas ser igual ao número de processadores.

O tempo para realizar a divisão é proporcional ao tamanho do segmento, ou seja, na raiz é proporcional a N , no segundo nível proporcional a $N/2$ e assim sucessivamente. Como estamos considerando que a divisão é feita até que o número de folhas seja igual ao número de

processadores, as duas divisões do segundo nível podem ser feitas em paralelo.

Ou seja, o tempo necessário para realizar as divisões é o seguinte:

$$t_{par}^{(1)} = \sum_{i=0}^{\log_2 p - 1} \frac{N}{2^i} = 2N - \frac{2N}{P} \quad (D.1)$$

Esses segmentos de vetores de tamanho N/P são ordenados com o seguinte tempo de execução: $t_{par}^{(2)} = (N/P) \log_2(N/P)$.

Portanto o tempo total de execução do algoritmo é

$$t_{par} = t_{par}^{(1)} + t_{par}^{(2)} = \frac{N}{P} \log_2 \frac{N}{P} + 2N - \frac{2N}{P} \quad (D.2)$$

De posse do tempo de execução paralelo e do tempo de execução sequencial é calculada a eficiência:

$$E = \frac{t_{seq}}{t_{par} P} \quad (D.3)$$

Substituindo e simplificando obtém-se a seguinte fórmula para a eficiência do *quick sort*:

$$E = \frac{1}{\frac{4^p}{\log_2 \frac{4^p}{4p}} + 1 + \frac{\log_2 \frac{4^p}{4p}}{\log_2 N}} \quad (D.4)$$

Utilizando a relação $S = E \times p$:

$$S(p) = \frac{p}{\frac{4^p}{\log_2 \frac{4^p}{4p}} + 1 + \frac{\log_2 \frac{4^p}{4p}}{\log_2 N}} \quad (D.5)$$

ALGORITMOS – π FLOWMT

Os algoritmos implementados no π FlowMT podem ser vistos neste anexo.

Código-fonte 9: Integração binária - Recursiva

```
1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 #include <cmath>
8
9 double dx;
10 double pi = 4.*atan(1.);
11 int fx;
12 int degree;
13
14 size_t logdx;
15 size_t N;
16
17 float max=pi;
18 float min=0.;
19
20 size_t RB=8;
21
22 struct helper{
23     float start;
```

```
24  float finish;
25  float pi;
26 };
27
28 double function(double x){
29  double f;
30  switch(fx){
31  case(1):{
32      double a=pi;
33
34      double dtau=0.01;
35      double stau=0;
36
37      for(double tau=0;tau<pi;tau=tau+dtau){
38          stau+= dtau*( cos(a*tau - x*sin(tau)) +cos(a*(tau+
39          dtau) - x*sin(tau+dtau)))/2.;
40      }
41
42      double dt=0.05;
43      double tm=pi;
44      double st=0;
45
46      for(double t=0;t<tm;t=t+dt){
47          st+=dt*(exp(-x*sinh(t)-a*t)+exp(-x*sinh(t+dt)-a*(t+
48          dt)))/2.;
49      }
50
51      f=(1./pi)*stau - (sin(a*pi)/pi)*st;
52  }break;
53
54  case(2):{
55      f=0;
56
57      for(int i=0;i<degree+1;i++) f+=(i+1)*pow(x,
58      static_cast<double>(i));
59  }break;
```

```

60     }
61     return f;
62 }
63
64 Piflow::OutV sum(Piflow::InV input){
65     helper* h;
66     helper* aux;
67
68     auto id=input[0].Id();
69     auto label=input[0].Label()+1;
70
71     h = reinterpret_cast<helper*>(input[0].Data());
72
73     for(size_t i=1;i<RB;i++){
74         aux=reinterpret_cast<helper*>(input[i].Data());
75
76         if(h->start > aux->start) h->start = aux->start;
77         if(h->finish < aux->finish) h->finish = aux->finish;
78
79         h->pi = h->pi + aux->pi;
80         delete aux;
81     }
82
83     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
84     outputs.reserve(1);
85
86     if(h->start==min && h->finish==max){
87         Piflow::End();
88         outputs.emplace_back(new float(h->pi));
89         delete h;
90     }
91     else{
92         outputs.emplace_back(h,id/RB,label,label*N+id/RB, &sum,
93             RB);
94     }
95     return outputs;
96 }
97

```

```

98 Piflow::OutV integration(Piflow::InV input){
99     float* lim = reinterpret_cast<float*>(input[0].Data());
100
101     auto id=input[0].Id();
102     auto label=input[0].Label()+1;
103
104     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
105
106
107     if((lim[1]-lim[0])>dx){
108         float* lim_aux = new float[2];
109
110         lim_aux[0]=lim[0];
111         lim_aux[1]=(lim[1]+lim[0])/2.;
112
113         lim[0]=lim_aux[1];
114
115         outputs.reserve(2);
116         outputs.emplace_back(lim_aux,2*id,label,label*N+ 2*id,
117             &integration, 1);
117         outputs.emplace_back(lim,2*id+1,label,label*N+ 2*id+1,
118             &integration, 1);
118     }
119     else{
120         helper* h = new helper;
121         h->start=lim[0];
122         h->finish=lim[1];
123         h->pi=(lim[1]-lim[0])*(function(lim[0])+function(lim
124             [1]))/2.;
124         delete[] lim;
125         outputs.reserve(1);
126         outputs.emplace_back(h,id/RB,label,label*N+id/RB,&sum,
127             RB);
127     }
128
129     return outputs;
130 }
131
132 uint64_t flog2(uint64_t x){

```

```
133  uint64_t log=0;
134  while(x>=1) log++;
135  return log;
136 }
137
138 int main(int argc, char* argv[]){
139  dx = std::stof(argv[1]);
140  size_t Nthreads = std::stoul(argv[2]);
141  RB = std::stoul(argv[3]);
142  fx = std::stoi(argv[4]);
143  degree=std::stoi(argv[5]);
144
145  auto t1 = cycles();
146
147  logdx=flog2( (uint64_t)((max-min)/dx) ) + 1;
148  N = pow(2.,logdx);
149
150  while(logdx%flog2(RB)!=0) RB/=2;
151
152  Piflow::Piflow paralelo(Nthreads,false);
153
154  paralelo.Start();
155
156  float* limite = new float[2];
157  limite[0]=min;
158  limite[1]=max;
159
160  Piflow::Codelet codelet(0,&integration,1);
161  codelet.Insert(Piflow::Input(limite,0,0));
162  paralelo.Insert(codelet);
163
164  paralelo.Run();
165
166  paralelo.Finish();
167
168  float bi=*(reinterpret_cast<float*>(paralelo.outputs[0].
    Data()));
169
170  auto t2 = cycles();
```

```
171 auto tempo = t2-t1;
172
173 std::clog << "[%T]\t" << dx << '\t' << RB << '\t' <<
    Nthreads << '\t' << tempo << std::endl;
174
175 std::clog << "[%C]";
176 for (size_t i=0;i<Nthreads; i++) std::clog << '\t' <<
    paralelo.N_exec(i);
177 std::clog << std::endl;
178
179 return 0;
180 }
```

Código-fonte 10: Integração binária - Não recursiva

```
1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 #include <cmath>
8
9 double dx;
10 double pi = 4.*atan(1.);
11 int fx;
12 int degree;
13
14 float max=pi;
15 float min=0.;
16
17 size_t RB=8;
18 size_t Block;
19
20 struct helper{
21     float start;
22     float finish;
23     float pi;
24 };
```



```
25
26 double function(double x){
27     double f;
28     switch(fx){
29         case(1):{
30             double a=pi;
31
32             double dtau=0.01;
33             double stau=0;
34
35             for(double tau=0;tau<pi;tau=tau+dtau){
36                 stau+= dtau*( cos(a*tau - x*sin(tau)) +cos(a*(tau+
37 dtau) - x*sin(tau+dtau)))/2.;
38             }
39
40             double dt=0.05;
41             double tm=pi;
42             double st=0;
43
44             for(double t=0;t<tm;t=t+dt){
45                 st+=dt*(exp(-x*sinh(t)-a*t)+exp(-x*sinh(t+dt)-a*(t+
46 dt)))/2.;
47             }
48
49             f=(1./pi)*stau - (sin(a*pi)/pi)*st;
50         }break;
51
52         case(2):{
53             f=0;
54             for(int i=0;i<degree+1;i++) f+=(i+1)*pow(x,
55 static_cast<double>(i));
56         }break;
57
58         default:{
59             f=x;
60         }break;
61     }
62     return f;
63 }
```

```

61
62
63 Piflow::OutV sum(Piflow::InV input){
64     helper* h;
65     helper* aux;
66
67     h = reinterpret_cast<helper*>(input[0].Data());
68
69     size_t it = input[0].Label()+1;
70
71     for(size_t i=1;i<RB;i++){
72         aux=reinterpret_cast<helper*>(input[i].Data());
73
74         if(h->start > aux->start) h->start = aux->start;
75         if(h->finish < aux->finish) h->finish = aux->finish;
76
77         h->pi = h->pi + aux->pi;
78         delete aux;
79     }
80
81     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
82     outputs.reserve(1);
83
84     if(h->start==min && h->finish==max){
85         Piflow::End();
86         outputs.emplace_back(new float(h->pi));
87         delete h;
88     }
89     else{
90         outputs.emplace_back(h,input[0].Id()/RB, it,(Block*it)+
            input[0].Id()/RB, &sum, RB);
91     }
92
93     return outputs;
94 }
95
96 Piflow::OutV integration(Piflow::InV input){
97     float* lim = reinterpret_cast<float*>(input[0].Data());
98

```

```

99  helper* h = new helper;
100 h->start = lim[0];
101 h->finish = lim[1];
102 h->pi = (lim[1]-lim[0])*(function(lim[0])+function(lim
    [1]))/2.;
103
104 Piflow::OutV outputs(Piflow::OutV_type::Lockless);
105 outputs.reserve(1);
106
107 outputs.emplace_back(h, input[0].Id()/RB,0, input[0].Id()
    /RB, &sum, RB);
108
109 return outputs;
110 }
111
112 uint64_t flog2(uint64_t x){
113     uint64_t log=0;
114     while(x>=1) log++;
115     return log;
116 }
117
118
119 int main(int argc, char* argv[]){
120     Block = pow(2.,std::stoul(argv[1]));
121     size_t Nthreads = std::stoul(argv[2]);
122     RB = std::stoul(argv[3]);
123     fx = std::stoi(argv[4]);
124     degree=std::stoi(argv[5]);
125
126     auto t1 = cycles();
127
128     dx=(max-min)/Block;
129
130     while(log(Block)/log(RB)!=(long)(log(Block)/log(RB))) RB
        /=2;
131
132     Piflow::Piflow paralelo(Nthreads,false);
133
134     paralelo.Start();

```

```

135
136     float start;
137     float finish;
138
139     for(size_t i=0;i<Block;i++){
140         start = min+i*dx;
141         finish = min+(i+1)*dx;
142         Piflow::Codelet codelet(i,&integration,1);
143         codelet.Insert(Piflow::Input(new float[2]{start,finish
144             },i));
145         paralelo.Insert(codelet);
146     }
147     paralelo.Run();
148
149     paralelo.Finish();
150
151     float bi=*(reinterpret_cast<float*>(paralelo.outputs[0].
152         Data()));
153
154     auto t2 = cycles();
155     auto tempo = t2-t1;
156
157     std::clog << "[%T]\t" << dx << '\t' << RB << '\t' <<
158         Nthreads << '\t' << tempo << std::endl;
159
160     std::clog << "[%C]";
161     for (size_t i=0;i<Nthreads; i++) std::clog << '\t' <<
162         paralelo.N_exec(i);
163     std::clog << std::endl;
164
165     return 0;
166 }

```

Código-fonte 11: *Bucket sort*

```

1 #include <iostream>
2 #include <random>
3

```

```

 4 #include <vector>
 5
 6 #include "../piflow/piflow.h"
 7 #include "../cycles.h"
 8
 9 size_t N;
10 size_t Nthreads;
11 size_t NB;
12
13 float max;
14 float min;
15 float tn;
16
17 float* V;
18
19 size_t cutoff = 15;
20
21 void insertionsort_t(std::vector<float> &V, size_t start,
    size_t N){
22     float tmp;
23     for(size_t i=1; i<N; i++){
24         for(size_t j=i; j>0 && V[j+start-1]>V[j+start]; (j>0 ? j
            --:0)){
25             tmp=V[j+start];
26             V[j+start]=V[j+start-1];
27             V[j+start-1]=tmp;
28         }
29     }
30 }
31
32 void quicksort_t(std::vector<float> &V, size_t start,
    size_t finish){
33     size_t i=start;
34     size_t j=finish;
35
36     float center=V[(i+j)/2];
37
38     float tmp;
39

```

```

40  if(finish-start>cutoff){
41      while(i<=j){
42          while (V[i]<center) i++;
43          while (V[j]>center) (j>0 ? j--:0);
44          if(i<=j){
45              tmp=V[i];
46              V[i]=V[j];
47              V[j]=tmp;
48              i++;
49              (j>0 ? j--:0);
50          }
51      }
52
53      if(start<j) quicksort_t(V,start,j);
54      if(i<finish) quicksort_t(V,i,finish);
55  }
56  else{
57      insertionsort_t(V,start,finish-start+1);
58  }
59 }
60
61
62 Piflow::OutV bs3(Piflow::InV input){
63     const size_t n_in=input.size();
64     size_t n0;
65
66     size_t start=0, finish=0;
67     std::vector<float> *v;
68
69     for(size_t i=0;i<n_in;i++){
70         if(input[i].Id() == NB+1){
71             v=reinterpret_cast<std::vector<float>*>(input[i].Data
72             ());
73             finish = v->size();
74         }
75         else{
76             start = start + input[i].Label();
77             if (input[i].Id()==0) n0=input[i].Label();
78         }
79     }
80 }

```

```

78     }
79     start = start - n0;
80     finish = start + finish;
81
82     for(size_t i=0;start<finish;start++, i++){
83         V[start]=(*v)[i];
84     }
85
86     delete v;
87     Piflow::End();
88     return Piflow::OutV(Piflow::OutV_type::Bypass);
89 }
90
91 Piflow::OutV bs2(Piflow::InV input){
92     const size_t n_in = Nthreads;
93     size_t n=0;
94
95     std::vector<float> *B = new std::vector<float>;
96     std::vector<float> *aux;
97
98     size_t j = input[0].Id();
99     const size_t n_out = NB+1-j;
100
101     B->reserve(n);
102
103     for(size_t i=0;i<n_in;i++){
104         aux=reinterpret_cast<std::vector<float>*>(input[i].Data
105             ());
106
107         n=n+aux->size();
108         B->reserve(n);
109
110         B->insert(B->end(),std::make_move_iterator(aux->begin()
111             ),std::make_move_iterator(aux->end()));
112         delete aux;
113     }
114
115     const size_t size = B->size();
116

```

```

115  quicksort_t ((*B), 0, size-1);
116
117  Piflow::OutV outputs(Piflow::OutV_type::Lock);
118  outputs.reserve(n_out);
119
120  outputs.emplace_back(B, NB+1, j, j, &bs3, j+2);
121
122  for(size_t i=0; i<n_out-1; i++, j++){
123      outputs.emplace_back(NULL, i, size, j, &bs3, j+2);
124  }
125  return outputs;
126 }
127
128 Piflow::OutV bs1(Piflow::InV input){
129     size_t start = input[0].Id();
130     size_t finish = input[0].Label();
131
132     std::vector<float> **B = new std::vector<float>* [NB];
133     for(size_t i=0; i<NB; i++){
134         B[i]=new std::vector<float>;
135     }
136
137     size_t bi;
138     for(size_t i=start; i<finish; i++){
139         bi=int((V[i]-min)/tn);
140         B[(bi==NB ? bi-1:bi)]->push_back(V[i]);
141     }
142
143     Piflow::OutV outputs(Piflow::OutV_type::Lock);
144     outputs.reserve(NB);
145     for(size_t i=0; i<NB; i++){
146         outputs.emplace_back(B[i], i, i, &bs2, Nthreads);
147     }
148     return outputs;
149 }
150
151
152 int main(int argc, char* argv[]){
153     N = pow(2., std::stoul(argv[1]));

```

```

154  NB = std::stoul(argv[2]);
155  Nthreads = std::stoul(argv[3]);
156  int seed = (argc==4 ? 1:std::stoul(argv[4]));
157
158  V=new float[N];
159
160  //parallel inatialization
161  uint64_t nb_ = pow(2.,10.);
162  uint64_t start_[Nthreads];
163  uint64_t finish_[Nthreads];
164
165  for(uint64_t i=0;i<Nthreads;i++){
166      start_[i]=(i==0 ? 0:finish_[i-1]);
167      finish_[i]= start_[i] + nb_/Nthreads + ( i < (nb_ %
        Nthreads) ? 1:0);
168  }
169
170  for(uint64_t j=0; j<Nthreads;j++){
171      {
172          for(uint64_t k=start_[j]; k< finish_[j];k++){
173              std::random_device rd;
174              std::uniform_real_distribution<> v(-10,10);
175              std::mt19937 gen(seed*1337 + k);
176
177              for(uint64_t i=0;i<N/nb_;i++){
178                  V[k*(N/nb_)+i]=v(gen);
179              }
180          }
181      }
182  }
183
184  max=-10.;
185  min=10.;
186  for(uint64_t i=0;i<N;i++){
187      max=(V[i]>max ? V[i]:max);
188      min=(V[i]<min ? V[i]:min);
189  }
190
191  tn=(max-min)/float(NB);

```

```

192
193  auto t1 = cycles();
194
195  Piflow::Piflow paralelo(Nthreads, false);
196
197  paralelo.Start();
198
199  size_t block_size = N/Nthreads;
200  size_t block_r = N%Nthreads;
201  size_t start=0, finish;
202
203  for(size_t i=0; i<Nthreads;i++){
204      finish = start + block_size + (i<block_r ? 1:0);
205
206      Piflow::Codelet codelet(i,&bs1,1);
207      codelet.Insert(Piflow::Input(NULL, start, finish));
208
209      paralelo.Insert(codelet);
210      start=finish;
211  }
212
213  paralelo.Run();
214
215  paralelo.Finish();
216
217  auto t2 = cycles();
218  auto tempo = t2-t1;
219
220  delete[] V;
221
222  std::cerr << "[%%T]\t" << N << '\t' << NB << '\t' <<
    Nthreads << '\t' << tempo << std::endl;
223
224  std::cerr << "[%%C]";
225  for (size_t i=0;i<Nthreads; i++) std::cerr << '\t' <<
    paralelo.N_exec(i);
226  std::cerr << std::endl;
227
228  return 0;

```

229 }

Código-fonte 12: Produto interno

```

1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 #include <cmath>
8
9 size_t N;
10 size_t Nthreads;
11 size_t Block;
12 size_t SBlock;
13 size_t Np;
14
15 float *V1, *V2;
16
17 Piflow::OutV sum(Piflow::InV input){
18     const size_t n_in=input.size();
19
20     const size_t id=input[0].Id()/SBlock;
21     const size_t label=input[0].Label()+1;
22
23     float* dpp = (reinterpret_cast<float*>(input[0].Data()));
24     float* h;
25
26     float aux=0;
27     for(size_t i=1; i<n_in; i++){
28         h=(reinterpret_cast<float*>(input[i].Data()));
29         aux=aux+*h;
30         delete h;
31     }
32     *dpp=*dpp+aux;
33
34     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
35     outputs.reserve(1);

```

```

36
37   if(label!=Np) outputs.emplace_back(dpp,id,label,(N*label)
      +id,&sum,SBlock);
38   else{ Piflow::End(); outputs.emplace_back(dpp);}
39
40   return outputs;
41 }
42
43 Piflow::OutV dotproduct(Piflow::InV input){
44   size_t start = input[0].Id();
45   const size_t finish = input[0].Label();
46
47   float dpp=0;
48   for(size_t i=start; i<finish;i++) dpp=dpp+V1[i]*V2[i];
49
50   Piflow::OutV outputs(Piflow::OutV_type::Lockless);
51   outputs.reserve(1);
52
53   const size_t id = (finish/Block - 1)/SBlock;
54
55   outputs.emplace_back(new float(dpp),id,0, id, &sum,
      SBlock);
56
57   return outputs;
58 }
59
60 int main(int argc, char* argv[]){
61   N = pow(2.,std::stoul(argv[1]));
62   Nthreads = std::stoul(argv[2]);
63   Block = std::stoul(argv[3]);
64   SBlock = std::stoul(argv[4]);
65
66   Np = log2(N/Block)/log2(SBlock);
67
68   V1=new float[N];
69   V2=new float[N];
70
71   float dp=0;
72

```

```

73  std::random_device rd;
74  std::mt19937 gen(rd());
75  std::uniform_real_distribution<> v(-10,10);
76
77  for(size_t i=0;i<N;i++){
78      V1[i]=v(gen);
79      V2[i]=v(gen);
80  }
81
82
83  auto t1 = cycles();
84
85  Piflow::Piflow paralelo(Nthreads,false);
86
87  const size_t Nb = N/Block;
88  size_t start=0,finish=Block;
89  paralelo.Start();
90
91  for(size_t i=0;i<Nb;i++){
92      Piflow::Codelet codelet(i,&dotproduct,1);
93      codelet.Insert(Piflow::Input(NULL,start,finish));
94      paralelo.Insert(codelet);
95      start = finish;
96      finish = finish + Block;
97  }
98
99  paralelo.Run();
100
101  paralelo.Finish();
102
103  dp=*(reinterpret_cast<float*>(paralelo.outputs[0].Data()))
    );
104
105  auto t2 = cycles();
106  auto tempo = t2-t1;
107
108
109  std::cout << "[%T]\t" << N << '\t' << Block << '\t' <<
    Nb << '\t' << Nthreads << '\t' << tempo << std::endl;

```

```

110
111     std::cout << " [%%C] ";
112     for (size_t i=0; i<Nthreads; i++) std::cout << '\t' <<
        paralelo.N_exec(i);
113     std::cout << std::endl;
114
115     delete[] V1;
116     delete[] V2;
117
118     return 0;
119 }

```

Código-fonte 13: FFT

```

1 #include <iostream>
2 #include <random>
3 #include <complex>
4
5 #include <cmath>
6
7 #include "../piflow/piflow.h"
8
9 #include "../cycles.h"
10
11 uint64_t N;
12 uint64_t cutoff;
13
14 double inline pi() { return std::acos(-1.); }
15
16 void separate(std::complex<double>* a, uint64_t n){
17     const uint64_t n2 = n/2.;
18     std::complex<double>* b = new std::complex<double>[n2];
19
20     for(uint64_t i=0; i<n2; i++) b[i]=a[i*2+1];
21     for(uint64_t i=0; i<n2; i++) a[i]=a[i*2];
22     for(uint64_t i=0; i<n2; i++) a[i+n2]=b[i];
23
24     delete[] b;
25 }

```

```

26
27 void fft_seq(std::complex<double>* w, uint64_t n){
28     if(n>1){
29         const uint64_t n2 = n/2.;
30         separate(w,n);
31         fft_seq(w,n2);
32         fft_seq(w+n2,n2);
33
34         for(uint64_t i=0;i<n2;i++){
35             std::complex<double> e = w[i];
36             std::complex<double> o = w[i+n2];
37
38             std::complex<double> f = std::exp(std::complex<double>
>(0,-2.*pi()*i/n));
39
40             w[i]=e+f*o;
41             w[i+n2]=e-f*o;
42         }
43     }
44 }
45
46 Piflow::OutV fft2(Piflow::InV input){
47     std::complex<double>* w1 = static_cast<std::complex<
double>*>(input[0].Data());
48     std::complex<double>* w2 = static_cast<std::complex<
double>*>(input[1].Data());
49     std::complex<double>* w = (w1>w2 ? w2 : w1);
50     const uint64_t n=input[0].Id();
51     const uint64_t n2 = n/2.;
52     const uint64_t k=input[0].Label();
53     const uint64_t it=N/n;
54
55     Piflow::OutV outputs(Piflow::OutV_type::Bypass);
56     if(n<N){
57         for(uint64_t i=0;i<n2;i++){
58             std::complex<double> e = w[i];
59             std::complex<double> o = w[i+n2];
60

```

```

61     std::complex<double> f = std::exp(std::complex<double>
        >(0,-2.*pi()*i/n))*o;
62
63     w[i]=e+f;
64     w[i+n2]=e-f;
65 }
66
67     outputs.tc(Piflow::OutV_type::Lockless);
68     outputs.emplace_back(w,2*n,k/2,it*N+k/2, &fft2,2);
69 }
70 else Piflow::End();
71
72 return outputs;
73 }
74
75 Piflow::OutV fft1(Piflow::InV input){
76     std::complex<double>* w = static_cast<std::complex<double>
        >>(input[0].Data());
77     const uint64_t n=input[0].Id();
78     const uint64_t k=input[0].Label();
79
80     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
81     if(n>cutoff){
82         const uint64_t n2 = n/2.;
83         separate(w,n);
84
85         outputs.emplace_back(&w[0],n2,2*k,2*k,&fft1,1);
86         outputs.emplace_back(&w[n2],n2,2*k+1,2*k+1,&fft1,1);
87     }
88     else{
89         fft_seq(w,n);
90         outputs.emplace_back(&w[0],2*n,k/2,k/2, &fft2,2);
91     }
92     return outputs;
93 }
94
95 int main(int argc, char* argv[]){
96     N = pow(2.,std::stoul(argv[1]));
97     uint64_t Nthreads = std::stoul(argv[2]);

```

```

98  cutoff = pow(2.,std::stoul(argv[3]));
99  const double tf=0.5*std::stoul(argv[1]);
100 const double rate = N/tf;
101 const double resolution = rate/N;
102
103 std::complex<double>* x = new std::complex<double>[N];
104 std::complex<double>* w = new std::complex<double>[N];
105
106 const uint64_t nfreq=static_cast<uint64_t>(std::log10(((N
    /2)-1)*resolution));
107
108 double freq[nfreq];
109
110 for(uint64_t i=0;i<nfreq;i++) freq[i]=std::pow(10,i);
111
112 for(uint64_t i=0;i<N;i++){
113     x[i]=0.0;
114     for(uint64_t j=0;j<nfreq;j++) x[i]+=sin(2*pi()*freq[j]*
        i/N);
115     w[i]=x[i];
116 }
117
118 auto t1 = cycles();
119
120 Piflow::Piflow paralelo(Nthreads,false);
121
122 paralelo.Start();
123
124 Piflow::Codelet codelet(0,&fft1,1);
125 codelet.Insert(Piflow::Input(&w[0],N,0));
126 paralelo.Insert(codelet);
127
128 paralelo.Run();
129 paralelo.Finish();
130
131 auto t2 = cycles();
132 auto tempo = t2-t1;
133
134 delete[] x;

```

```

135  delete[] w;
136
137  //for(uint64_t i=0;i<N;i++) std::cout << i*rate << '\t'
    << x[i].real() << std::endl;
138
139  //for(uint64_t i=0;i<N/2.;i++) std::cout << i << '\t' <<
    std::abs(w[i]) << std::endl;
140
141  std::clog << "[%T]\t" << N << '\t' << Nthreads << '\t'
    << cutoff << '\t' << tempo << std::endl;
142  std::clog << "[%C]";
143  for (int64_t i=0;i<Nthreads; i++) std::clog << '\t' <<
    paralelo.N_exec(i);
144  std::clog << std::endl;
145 }

```

Código-fonte 14: Fatoração L.U.

```

1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 uint64_t N;
8 uint64_t Nthreads;
9 uint64_t cutoff;
10
11 uint64_t out;
12
13 Piflow::OutV lufac1(Piflow::InV input);
14 Piflow::OutV lufac2(Piflow::InV input);
15 Piflow::OutV lufac_seq(Piflow::InV input);
16
17 Piflow::OutV lufac_seq(Piflow::InV input){
18     uint64_t k = input[0].Id();
19
20     float** A = static_cast<float**>(input[0].Data());
21

```

```

22   for(;k<N;k++){
23       for(uint64_t i=k+1;i<N;i++){
24           A[i][k]=A[i][k]/A[k][k];
25           for(uint64_t j=k+1;j<N;j++){
26               A[i][j]=A[i][j] - A[i][k]*A[k][j];
27           }
28       }
29
30   }
31
32   Piflow::End();
33
34   return Piflow::OutV(Piflow::OutV_type::Bypass);
35 }
36
37 Piflow::OutV lufac1(Piflow::InV input){
38     const uint64_t i = input[0].Id()%N;
39     const uint64_t k = input[0].Id()/N;
40     float** A = static_cast<float**>(input[0].Data());
41
42     A[i][k]=A[i][k]/A[k][k];
43     for(uint64_t j=k+1;j<N;j++){
44         A[i][j]=A[i][j] - A[i][k]*A[k][j];
45     }
46
47     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
48
49     outputs.emplace_back(A,k+1,k,&lufac2,N-k-1);
50     return outputs;
51 }
52
53
54 Piflow::OutV lufac2(Piflow::InV input){
55     const uint64_t k = input[0].Id();
56
57     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
58     float** A = static_cast<float**>(input[0].Data());
59     if(N-k>cutoff){
60         for(uint64_t l=k+1;l<N;l++){

```

```

61     outputs.emplace_back(A,k*N+1,k,&lufac1,1);
62 }
63 }
64 else{
65     outputs.emplace_back(A,k,k,&lufac_seq,1);
66 }
67 if(outputs.empty()) outputs.tc(Piflow::OutV_type::Bypass)
68 ;
69 return outputs;
70 }
71
72 int main(int argc, char* argv[]){
73     N = std::stoul(argv[1]);
74     Nthreads = std::stoul(argv[2]);
75     cutoff = (argc==4 ? std::stoul(argv[3]):20);
76
77     float **A = new float*[N];
78
79     for(uint64_t i=0;i<N;i++) A[i]=new float[N];
80
81     std::random_device rd;
82     std::mt19937 gen(rd());
83     std::uniform_real_distribution<> v(-10,10);
84
85     for(uint64_t i=0;i<N;i++){
86         for(uint64_t j=0;j<N;j++){
87             auto aux = v(gen);
88             A[i][j]=aux;
89             if(Validate==1) B[i][j]=aux;
90         }
91     }
92
93     auto t1 = cycles();
94
95     Piflow::Piflow paralelo(Nthreads,false);
96
97     paralelo.Start();
98

```

```

99   Piflow::Codelet codelet(0,&lufac2,1);
100   codelet.Insert(Piflow::Input(A, 0));
101   paralelo.Insert(codelet);
102
103   paralelo.Run();
104
105   paralelo.Finish();
106
107   auto t2 = cycles();
108   auto tempo = t2-t1;
109
110   std::clog << "[%%T]\t" << N << '\t' << cutoff << '\t' <<
        Nthreads << '\t' << tempo << std::endl;
111
112   std::clog << "[%%C]";
113   for (uint64_t i=0;i<Nthreads; i++) std::clog << '\t' <<
        paralelo.N_exec(i);
114   std::clog << std::endl;
115
116
117   for(uint64_t i=0;i<N;i++){
118       delete[] A[i];
119   }
120   delete[] A;
121   if(Validate==1) delete[] B;
122
123   return 0;
124 }

```

Código-fonte 15: Multiplicação de Matrizes

```

1 #include <iostream>
2 #include <random>
3
4 #include <vector>
5 #include <queue>
6 #include <stack>
7
8 #include <thread>

```

```

 9 #include <pthread.h>
10 #include "../piflow/piflow.h"
11 #include "../cycles.h"
12
13
14 #include <boost/any.hpp>
15
16 float** M1;
17 float** M2;
18 float** Md;
19
20 size_t Block_j;
21 size_t Block_i;
22
23 size_t N;
24
25
26 Piflow::OutV MatMult(Piflow::InV input){
27     size_t i=input[0].Id();
28     size_t j=input[0].Label();
29
30     size_t imax=(i+Block_i > N ? N:i+Block_i);
31     size_t jmax=(j+Block_j > N ? N:j+Block_j);
32
33     for(size_t m=i;m<imax;m++){
34         for(size_t l=j;l<jmax;l++){
35
36             float res=0;
37             for(size_t k=0;k<N;k++) res+=M1[m][k]*M2[l][k];
38
39             Md[m][l]=res;
40         }
41     }
42     return Piflow::OutV(Piflow::OutV_type::Bypass);
43 }
44
45 Piflow::OutV Divider(Piflow::InV input){
46     Piflow::OutV outputs(Piflow::OutV_type::Lockless);
47     outputs.reserve(N);

```

```

48
49     size_t i = input[0].Id();
50     for(size_t j=0;j<N;j=j+Block_j){
51         outputs.emplace_back(NULL,i,j, j+i*N,&MatMult,1);
52     }
53
54     if(i+Block_i>=N-1) Piflow::End();
55
56     return outputs;
57 }
58
59
60 int main(int argc, char* argv[]){
61     N = std::stoul(argv[1]);
62     size_t Nthreads = std::stoul(argv[2]);
63     Block_j = (argc>=4 ? std::stoul(argv[3]) : 16);
64     Block_i = (argc==5 ? std::stoul(argv[4]) : 16);
65
66     M1= new float*[N];
67     M2= new float*[N];
68     Md= new float*[N];
69     for(size_t i=0;i<N;i++){
70         M1[i]=new float[N];
71         M2[i]=new float[N];
72         Md[i]=new float[N];
73     }
74
75     std::random_device rd;
76     std::mt19937 gen(rd());
77     std::uniform_real_distribution<> v(-10,10);
78
79     for(size_t i=0;i<N;i++){
80         for(size_t j=0;j<N;j++){
81             M1[i][j]=v(gen);
82             M2[i][j]=v(gen);
83             Md[i][j]=0.0;
84         }
85     }
86

```

```
87  auto t=cycles();
88
89  Piflow::Piflow paralelo(Nthreads,false);
90
91  paralelo.Start();
92
93  for(size_t i=0;i<N;i=i+Block_i){
94      Piflow::Codelet codelet(i,&Divider,1);
95      codelet.Insert(Piflow::Input(NULL,i));
96      paralelo.Insert(codelet);
97  }
98
99  paralelo.Run();
100
101  paralelo.Finish();
102  t=cycles()-t;
103
104  std::clog << "[%T]\t" << N << '\t' << Nthreads << '\t'
    << t << std::endl;
105
106  std::clog << "[%C]";
107  for (size_t i=0;i<Nthreads; i++) std::clog << '\t' <<
    paralelo.N_exec(i);
108  std::clog << std::endl;
109
110
111  for(size_t i=0;i<N;i++){
112      delete[] M1[i];
113      delete[] M2[i];
114      delete[] Md[i];
115  }
116  delete[] M1;
117  delete[] M2;
118  delete[] Md;
119
120  return 0;
121 }
```

```
1 #include <iostream>
2 #include <cmath>
3 #include <random>
4
5 #include "../piflow/piflow.h"
6
7 #include "../cycles.h"
8
9 uint64_t N;
10 float needle_min;
11 float needle_max;
12 float dn;
13 uint64_t Nn;
14
15 uint64_t Size;
16
17 uint64_t Nthreads;
18
19 const constexpr float pi = 4.*atan(1.);
20
21
22 Piflow::OutV monte_carlo(Piflow::InV input){
23     const size_t j = input[0].Id();
24
25     std::random_device rd;
26     std::uniform_real_distribution<> v(0,1);
27     std::mt19937 gen(rd());
28
29     float *p = static_cast<float*>(input[0].Data());
30
31     uint64_t nw = 0;
32     float needle = needle_min + j*dn;
33     for(uint64_t i=0;i<N;i++){
34         float x1= Size*v(gen) + 1;
35         float y1 = Size*v(gen) + 1;
36
37         float theta = 2.*pi*v(gen);
38
```

```

39     float x2 = x1 + needle*cos(theta);
40     float y2 = y1 + needle*sin(theta);
41
42     nw+= (ceil((y1>y2 ? y1:y2)) - floor((y1<y2 ? y1:y2))
         >1);
43 }
44 p[j] = float(nw)/float(N);
45
46 if(j==Nn-1) Piflow::End();
47
48 return Piflow::OutV(Piflow::OutV_type::Bypass);
49 }
50
51 int main(int argc, char* argv[]){
52     N = std::stof(argv[1]);
53     needle_min = std::stof(argv[2]);
54     needle_max = std::stof(argv[3]);
55     dn = std::stof(argv[4]);
56     Nthreads = std::stoul(argv[5]);
57
58     Size = ceil(needle_max);
59     Nn = (needle_max-needle_min)/dn;
60
61     float* p;
62     p = new float[Nn];
63
64     auto t1 = cycles();
65
66     Piflow::Piflow paralelo(Nthreads,false);
67
68     paralelo.Start();
69
70     for(size_t i=0;i<Nn;i++){
71         Piflow::Codelet codelet(i,&monte_carlo,1);
72         codelet.Insert(Piflow::Input(p,i));
73         paralelo.Insert(codelet);
74     }
75
76     paralelo.Run();

```

```

77
78     paralelo.Finish();
79
80     auto t2 = cycles();
81     auto tempo = t2-t1;
82
83     delete[] p;
84
85     std::clog << "[%%T]\t" << N << '\t' << Nn << '\t' <<
        Nthreads << '\t' << tempo << std::endl;
86     std::clog << "[%%C]";
87     for (size_t i=0;i<Nthreads; i++) std::clog << '\t' <<
        paralelo.N_exec(i);
88     std::clog << std::endl;
89
90 }
```

Código-fonte 17: *Quick sort*

```

1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 #include <cmath>
8
9 int64_t N;
10 int64_t Nthreads;
11 int64_t Block;
12
13 int64_t cutoff=15;
14
15 float* V;
16
17 inline void insertionsort_t(const int64_t start, const
    int64_t finish){
18     float tmp;
19     const int64_t N=finish-start+1;
```

```
20  float* Va = &V[start];
21  for(int64_t i=1;i<N;++i){
22      for(int64_t j=i;j>0 && Va[j-1]>Va[j];--j){
23          tmp=Va[j];
24          Va[j]=Va[j-1];
25          Va[j-1]=tmp;
26      }
27  }
28
29 }
30
31 inline void quicksort_t(const int64_t start, const int64_t
    finish){
32     if(finish-start>cutoff){
33         int64_t i=start;
34         int64_t j=finish;
35
36         float tmp;
37
38         float center=V[(i+j)/2];
39
40         while(i<=j){
41             while (V[i]<center) ++i;
42             while (V[j]>center) --j;
43             if(i<=j){
44                 tmp=V[i];
45                 V[i]=V[j];
46                 V[j]=tmp;
47                 ++i;
48                 --j;
49             }
50         }
51
52         if(start<j) quicksort_t(start,j);
53         if(i<finish) quicksort_t(i,finish);
54     }
55     else{
56         insertionsort_t(start,finish);
57     }
```

```

58 }
59
60 Piflow::OutV quicksort_s(Piflow::InV input){
61     const int64_t start=input[0].Id();
62     const int64_t finish=input[0].Label();
63
64     quicksort_t(start,finish);
65
66     return Piflow::OutV(Piflow::OutV_type::Bypass);
67 }
68
69 Piflow::OutV quicksort(Piflow::InV input){
70     const int64_t start=input[0].Id();
71     const int64_t finish=input[0].Label();
72
73     int64_t i=start;
74     int64_t j=finish;
75
76     float tmp;
77
78     float center=V[(i+j)/2];
79
80     Piflow::OutV outputs(Piflow::OutV_type::Bypass);
81     outputs.reserve(2);
82
83     while(i<=j){
84         while (V[i]<center) ++i;
85         while (V[j]>center) --j;
86         if(i<=j){
87             tmp=V[i];
88             V[i]=V[j];
89             V[j]=tmp;
90             ++i;
91             --j;
92         }
93     }
94
95     if(start<j){

```

```

96     if(j-start>Block)outputs.emplace_back(NULL,start,j,
        start, &quicksort,1);
97     else outputs.emplace_back(NULL,start,j, start, &
        quicksort_s,1);;
98 }
99 if(i<finish){
100     if(finish-i>Block)outputs.emplace_back(NULL,i,finish, i
        , &quicksort,1);
101     else outputs.emplace_back(NULL,i,finish, i, &
        quicksort_s,1);
102 }
103 if (!outputs.empty()) outputs.tc(Piflow::OutV_type::
        Lockless);
104 return outputs;
105 }
106
107 int main(int argc, char* argv[]){
108     N = pow(2.,std::stoul(argv[1]));
109     Nthreads = std::stoul(argv[2]);
110     Block = std::stoul(argv[3]);
111     int seed = (argc==4 ? 1:std::stoul(argv[4]));
112
113     V=new float[N];
114
115     //parallel inatialization
116     uint64_t nb_ = pow(2.,10.);
117     uint64_t start_[Nthreads];
118     uint64_t finish_[Nthreads];
119
120
121     for(uint64_t i=0;i<Nthreads;i++){
122         start_[i]=(i==0 ? 0:finish_[i-1]);
123         finish_[i]= start_[i] + nb_/Nthreads + ( i < (nb_ %
            Nthreads) ? 1:0);
124     }
125
126     for(uint64_t j=0; j<Nthreads;j++){
127         {
128             for(uint64_t k=start_[j]; k< finish_[j];k++){

```

```
129         std::random_device rd;
130         std::uniform_real_distribution<> v(-10,10);
131         std::mt19937 gen(seed*1337 + k);
132
133         for(uint64_t i=0;i<N/nb_;i++){
134             V[k*(N/nb_)+i]=v(gen);
135         }
136     }
137 }
138 }
139
140 auto t1 = cycles();
141
142 Piflow::Piflow paralelo(Nthreads,true);
143
144 paralelo.Start();
145
146 Piflow::Codelet codelet(0,&quicksort,1);
147 codelet.Insert(Piflow::Input(NULL,0,N-1));
148 paralelo.Insert(codelet);
149
150 paralelo.Run();
151 paralelo.Finish();
152
153 auto t2 = cycles();
154 auto tempo = t2-t1;
155
156 std::clog << "[%%T]\t" << N << '\t' << Block << '\t' <<
    Nthreads << '\t' << tempo << std::endl;
157
158 std::clog << "[%%C]";
159 for (int64_t i=0;i<Nthreads; i++) std::clog << '\t' <<
    paralelo.N_exec(i);
160 std::clog << std::endl;
161
162 delete[] V;
163
164 return 0;
165 }
```

Código-fonte 18: Quick sort - 2 pivôs

```
1 #include <iostream>
2 #include <random>
3
4 #include "../piflow/piflow.h"
5 #include "../cycles.h"
6
7 #include <cmath>
8
9 int64_t N;
10 int64_t Nthreads;
11 int64_t Block;
12
13 int64_t cutoff=15;
14
15 float* V;
16
17 inline void insertionsort_t(const int64_t start, const
    int64_t finish){
18     float tmp;
19     const int64_t N=finish-start+1;
20     float* Va = &V[start];
21     for(int64_t i=1;i<N;++i){
22         for(int64_t j=i;j>0 && Va[j-1]>Va[j];--j){
23             tmp=Va[j];
24             Va[j]=Va[j-1];
25             Va[j-1]=tmp;
26         }
27     }
28
29 }
30
31 inline void quicksort_t(const int64_t start, const int64_t
    finish){
32     if(finish-start>cutoff){
33         int64_t i=start;
34         int64_t j=finish;
35
36         float tmp;
```

```

37
38     float center=V[(i+j)/2];
39
40     while(i<=j){
41         while (V[i]<center) ++i;
42         while (V[j]>center) --j;
43         if(i<=j){
44             tmp=V[i];
45             V[i]=V[j];
46             V[j]=tmp;
47             ++i;
48             --j;
49         }
50     }
51
52     if(start<j) quicksort_t(start,j);
53     if(i<finish) quicksort_t(i,finish);
54 }
55 else{
56     insertionsort_t(start,finish);
57 }
58 }
59
60 Piflow::OutV quicksort_s(Piflow::InV input){
61     const int64_t start=input[0].Id();
62     const int64_t finish=input[0].Label();
63
64     quicksort_t(start,finish);
65
66     return Piflow::OutV(Piflow::OutV_type::Bypass);
67 }
68
69 Piflow::OutV quicksort(Piflow::InV input){
70     const int64_t start=input[0].Id();
71     const int64_t finish=input[0].Label();
72
73     int64_t i=start+1;
74     int64_t j=finish-1;
75     int64_t k=i;

```

```
76
77     float tmp;
78
79     if(V[start] > V[finish]){
80         tmp=V[start];
81         V[start]=V[finish];
82         V[finish]=tmp;
83     }
84
85     const float p1=V[start];
86     const float p2=V[finish];
87
88     Piflow::OutV outputs(Piflow::OutV_type::Bypass);
89     outputs.reserve(3);
90
91     while(k<=j){
92         if(V[k]<p1){
93             tmp=V[k];
94             V[k]=V[i];
95             V[i]=tmp;
96             ++i;
97         }
98         else if(V[k] >= p2){
99             while(V[j]>p2 && k<j) --j;
100             tmp=V[k];
101             V[k]=V[j];
102             V[j]=tmp;
103             --j;
104
105             if(V[k]<p1){
106                 tmp=V[k];
107                 V[k]=V[i];
108                 V[i]=tmp;
109                 ++i;
110             }
111         }
112         ++k;
113     }
114     ++j;
```

```
115  --i;
116
117  tmp=V[start];
118  V[start]=V[i];
119  V[i]=tmp;
120
121  tmp=V[finish];
122  V[finish]=V[j];
123  V[j]=tmp;
124
125  if(start<i-1){
126      if(i-1-start>Block) outputs.emplace_back(NULL,start,i-1,
127          start, &quicksort,1);
128      else outputs.emplace_back(NULL,start,i-1, start, &
129          quicksort_s,1);
130  }
131
132  if(i+1<j-1){
133      if(i-1-start>Block) outputs.emplace_back(NULL,i+1,j-1, i
134          , &quicksort,1);
135      else outputs.emplace_back(NULL,i+1,j-1, i, &quicksort_s
136          ,1);
137  }
138
139  if(j+1<finish){
140      if(finish-i>Block) outputs.emplace_back(NULL,j+1,finish,
141          j, &quicksort,1);
142      else outputs.emplace_back(NULL,j+1,finish, j, &
143          quicksort_s,1);
144  }
145  if (!outputs.empty()) outputs.tc(Piflow::OutV_type::
146      Lockless);
147  return outputs;
148 }
149
150
151
152
153
154 int main(int argc, char* argv[]){
155     N = pow(2.,std::stoul(argv[1]));
156     Nthreads = std::stoul(argv[2]);
```

```

147  Block = std::stoul(argv[3]);
148  int seed = (argc==4 ? 1:std::stoul(argv[4]));
149
150  V=new float[N];
151
152  //parallel inatialization
153  uint64_t nb_ = pow(2.,10.);
154  uint64_t start_[Nthreads];
155  uint64_t finish_[Nthreads];
156
157
158  for(uint64_t i=0;i<Nthreads;i++){
159      start_[i]=(i==0 ? 0:finish_[i-1]);
160      finish_[i]= start_[i] + nb_/Nthreads + ( i < (nb_ %
161          Nthreads) ? 1:0);
162  }
163
164  for(uint64_t j=0; j<Nthreads;j++){
165      {
166          for(uint64_t k=start_[j]; k< finish_[j];k++){
167              std::random_device rd;
168              std::uniform_real_distribution<> v(-10,10);
169              std::mt19937 gen(seed*1337 + k);
170
171              for(uint64_t i=0;i<N/nb_;i++){
172                  V[k*(N/nb_)+i]=v(gen);
173              }
174          }
175      }
176
177  auto t1 = cycles();
178
179  Piflow::Piflow paralelo(Nthreads,true);
180
181  paralelo.Start();
182
183  Piflow::Codelet codelet(0,&quicksort,1);
184  codelet.Insert(Piflow::Input(NULL,0,N-1));

```

```
185     paralelo.Insert(codelet);
186
187     paralelo.Run();
188     paralelo.Finish();
189
190     auto t2 = cycles();
191     auto tempo = t2-t1;
192
193     std::clog << "[%%T]\t" << N << '\t' << Block << '\t' <<
        Nthreads << '\t' << tempo << std::endl;
194
195     std::clog << "[%%C]";
196     for (int64_t i=0;i<Nthreads; i++) std::clog << '\t' <<
        paralelo.N_exec(i);
197     std::clog << std::endl;
198
199     delete[] V;
200
201     return 0;
202 }
```