PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

André Luiz Teodoro Alves

Paralelismo na Geração de Conceitos Baseado no Particionamento do Contexto Formal

Belo Horizonte

2024

André Luiz Teodoro Alves

Paralelismo na Geração de Conceitos Baseado no Particionamento do Contexto Formal

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Mark Alan Junho Song

Coorientador: Prof. Henrique Cota de Freitas

Belo Horizonte

2024

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Alves, André Luiz Teodoro
A474p Paralelismo na geração de conceitos baseado no particionamento do contexto formal / André Luiz Teodoro Alves. Belo Horizonte, 2024.
61 f. : il.

Orientador: Mark Alan Junho Song
Coorientador: Henrique Cota de Freitas
Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática

1. Computação de alto desempenho. 2. Conceitos - Análise. 3. Percepção de padrões. 4. Processamento paralelo (Computadores). 5. Processamento eletrônico de dados. 6. Mineração de dados (Computação). 7. Algoritmos computacionais. I. Song, Mark Alan Junho. II. Freitas, Henrique Cota de. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.

CDU: 681.3-11
Ficha catalográfica elaborada por Daniela Luzia da Silva Gomes - CRB 6/2505

André Luiz Teodoro Alves

Paralelismo na Geração de Conceitos Baseado no Particionamento do Contexto Formal

Dissertacao apresentada ao Programa de Pos-Graduacao em Informatica como requisito parcial para qualificacao ao Grau de Mestre em Informatica pela Pontificia Universidade Catolica de Minas Gerais.

Prof. Dr. Mark Alan Junho Song - PUC Minas
(Orientador)

Prof. Dr. Henrique Cota de Freitas - PUC Minas
(Coorientador)

Prof. Dr. Luis Enrique Zárate Gálvez - PUC Minas
(Banca Examinadora)

Prof. Dr. Adriano César Pereira - UFMG

(Banca Examinadora)

Belo Horizonte, 14 de Novembro de 2024

RESUMO

Os avanços tecnológicos recentes ampliaram o acesso ao conhecimento, especialmente com a expansão da Internet como uma fonte significativa de dados. No entanto, a capacidade humana de processar e interpretar grandes volumes de informações é limitada, o que demanda o desenvolvimento de técnicas eficazes de extração de conhecimento, como a Análise Formal de Conceitos (FCA). A FCA organiza dados binários e identifica padrões e relações entre objetos e atributos. No entanto, com o crescimento do volume e da dimensionalidade dos dados, os métodos tradicionais da FCA enfrentam desafios devido à explosão combinatória e aos altos custos computacionais. Esta dissertação propõe uma abordagem baseada em paralelismo, utilizando C++ e OpenMP, para particionar contextos formais em subcontextos e processá-los simultaneamente em múltiplos núcleos de processamento. Os resultados experimentais mostram ganhos de performance, com uma aceleração de até 22,957x e eficiência de 91,3%, indicando que a técnica proposta melhora a escalabilidade e o tempo de execução da FCA, além de expandir seu potencial de aplicação em áreas como mineração de dados, inteligência artificial e bioinformática.

Palavras-chave: Análise Formal de Conceitos, Particionamento, Paralelismo, Geração de Conceito, In-Close 4

ABSTRACT

Recent technological advancements have expanded access to knowledge, particularly with the growth of the internet as a significant data source. However, human capacity to process and interpret large volumes of information is limited, necessitating the development of effective knowledge extraction techniques such as Formal Concept Analysis (FCA). FCA organizes binary data and identifies patterns and relationships between objects and attributes. However, as the volume and dimensionality of data increase, traditional FCA methods face challenges due to combinatorial explosion and high computational costs. This dissertation proposes an approach based on parallelism, utilizing C++ and OpenMP, to partition formal contexts into subcontexts and process them simultaneously across multiple processing cores. Experimental results show performance gains, with acceleration of up to 22.957x and efficiency of 91.3%, indicating that the proposed technique improves FCA scalability and execution time, while also expanding its application potential in areas such as data mining, artificial intelligence, and bioinformatics.

Keywords: Formal Concept Analysis, Partitioning, Parallelism, Concept Generation, In-Close 4

Aos meus pais, à minha família
e principalmente à Ana.

AGRADECIMENTOS

Gostaria de agradecer a todos que contribuíram para a realização desta dissertação. Em primeiro lugar, agradeço ao meu orientador, Prof. Dr. Mark Alan Junho Song, pela orientação valiosa, paciência e incentivo ao longo de todo o processo, desde a graduação até na sua orientação no mestrado. Toda sua ajuda foi fundamental para o desenvolvimento deste trabalho. Também agradeço ao Prof. Dr. Henrique Cota de Freitas pela coorientação e toda a sua contribuição ao decorrer do projeto. Sem a orientação deles, esse projeto não seria possível.

Aos colegas e amigos do CArT (Computer Architecture and Parallel Processing Team), sou grato pelo apoio (mesmo que indireto) e pelas ajudas pontuais que precisei ao decorrer desta dissertação.

A minha família merece um agradecimento especial pelo suporte incondicional e pela compreensão durante os períodos mais intensos desta jornada. Seu apoio emocional e encorajamento foram indispensáveis para que eu pudesse concluir este trabalho. Em especial, devo muitos agradecimentos à minha noiva Ana Christina, a qual me apoiou sempre, na minha melhor e na minha pior fase. Amo todos vocês.

Agradeço também à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa e pela confiança concedida, assim como todo o trabalho feito no país, auxiliando o desenvolvimento acadêmico nacional. Também agradeço à FAPEMIG, ao CNPq e a PUC Minas por todo o suporte fornecido para a realização do projeto.

Finalmente, agradeço a todos que, direta ou indiretamente, contribuíram para a realização deste estudo. Sem o apoio e a colaboração de cada um, este trabalho não teria sido possível.

"A persistência é o caminho do êxito."

Charles Chaplin

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

CPU – *Central Processing Unit*

FCA – *Formal Concept Analysis*

FPGA – *Field Programmable Gate Array*

GCC – *GNU Compiler Collection*

GPU – *Graphics Processing Unit*

MIMD – *Multiple instructions, multiple data*

MISD – *Multiple instructions, single data*

OpenCL – *Open Computing Language*

OpenMP – *Open Multi-Processing*

RAM – *Random Access Memory*

SCGaz – *Synthetic Context Generator*

SIMD – *Single instrution, multiple data*

SISD – *Single instruction, single data*

SUMÁRIO

1 INTRODUÇÃO

Os avanços tecnológicos dos últimos anos têm ampliado consideravelmente o acesso ao conhecimento, promovendo o desenvolvimento de novos métodos para a recuperação de informações e aprimorando os já existentes. A Internet, como um vasto repositório de dados, transformou-se em uma plataforma essencial para a comunidade científica, acelerando a aquisição de conhecimento e facilitando a disseminação de informações atualizadas e relevantes (GODINHO et al., 2022).

Diante do enorme volume de dados disponíveis na Internet, a capacidade humana de processá-los é intrinsecamente limitada, principalmente devido à complexidade necessária para compreender e avaliar a relevância das informações. Para enfrentar esse desafio, diversas técnicas de mineração de dados foram desenvolvidas com o objetivo de extrair conhecimento útil e significativo dos dados disponíveis (DOMINGOS-SILVA; VIEIRA, 2008).

Neste cenário, a Análise Formal de Conceitos, originalmente *Formal Concept Analysis* (FCA), destaca-se como uma metodologia eficaz para identificar e explorar as relações entre diferentes elementos (CARPINETO; ROMANO, 2004). Adaptada para a análise de dados, a FCA enfatiza as associações e dependências entre objetos e atributos que frequentemente derivam de conjuntos de dados reais ou sintéticos (GANTER; WILLE, 2012). A estrutura de um Contexto Formal, que inclui atributos e objetos, é caracterizada por incidências — pares de subconjuntos de objetos e atributos correspondentes. A partir desses contextos, os conceitos formais são derivados, permitindo a organização hierárquica dentro de redes formais que revelam o conhecimento de maneira estruturada. Essa hierarquia encapsula as relações intrínsecas entre os conceitos da base de dados, facilitando uma compreensão mais profunda e detalhada da estrutura subjacente dos dados.

A aplicabilidade da FCA em diversas áreas do conhecimento é ampla, abrangendo campos como psicologia, inteligência artificial, biologia e linguística (PRISS, 2006; JINDAL; SEEJA; JAIN, 2020). Isso se deve principalmente à sua capacidade de esclarecer, de forma hierárquica, as relações complexas entre conjuntos de dados.

Entretanto, o esforço computacional necessário para a análise e extração de informações tende a aumentar significativamente quando se lida com bancos de dados densamente povoados e de alta dimensão, tornando a extração de conceitos formais inviável em tais cenários. Este desafio destaca a necessidade de desenvolver técnicas mais eficientes e escaláveis, capazes de lidar com a complexidade e a volumetria dos dados contemporâneos.

1.1 Problema

Com o crescimento do volume de dados, a complexidade das operações necessárias para processar esses dados também cresce exponencialmente. Em outras palavras, à medida que a dimensão do conjunto de dados se expande, a quantidade de combinações possíveis de objetos e atributos aumenta drasticamente, resultando em um aumento substancial no tempo e recursos computacionais necessários para realizar as análises.

Além disso, a alta dimensionalidade dos dados impõe desafios adicionais. Em bases de dados de alta dimensionalidade, os atributos podem ser altamente variados e numerosos, o que pode levar a uma explosão combinatória. Isso significa que, mesmo com técnicas avançadas de mineração de dados, o tempo de processamento pode se tornar proibitivamente longo, e os recursos computacionais podem ser insuficientes para lidar com a análise completa dos dados.

Existem diversas técnicas para o processamento dessas operações de combinação que são popularmente utilizadas, como algoritmos paralelos e métodos de otimização (FU; NGUIFO, 2004; WU et al., 2021). No entanto, muitas dessas técnicas ainda enfrentam problemas significativos de performance quando aplicadas a bases de dados de alta dimensionalidade. A eficiência do processamento continua a ser um desafio crítico, pois métodos tradicionais podem não escalar adequadamente com o aumento do volume de dados.

Portanto, a busca por abordagens mais eficientes e escaláveis para a FCA permanece um problema central na área. O desenvolvimento de novos algoritmos e técnicas capazes de lidar de forma eficaz com a complexidade e volumetria dos dados contemporâneos é crucial para avançar o estado da arte na análise de conceitos formais. Esses novos métodos devem não apenas melhorar a eficiência do processamento, mas também garantir que a integridade e a precisão da análise dos dados sejam mantidas, permitindo a extração de conhecimento significativo de grandes conjuntos de dados.

A necessidade de soluções inovadoras e robustas é, portanto, imperativa para superar os desafios atuais e permitir que a FCA continue a ser uma ferramenta poderosa e aplicável em um mundo cada vez mais orientado por dados.

1.2 Objetivo

Diante do problema citado na Seção 1.1, este estudo propõe uma abordagem baseada em C++ e OpenMP para reduzir o tempo de execução na obtenção de conceitos formais em arquiteturas com múltiplos núcleos de processamento e memória compartilhada. A abordagem particiona os contextos formais em subcontextos, permitindo o processamento paralelo eficiente dos subconjuntos. Os resultados mostram um ganho

significativo, abrindo novas possibilidades para a análise de dados em diversos campos.

1.2.1 Objetivos específicos

Almejando a construção e a validação do protótipo, foram necessárias as realizações dos seguintes objetivos específicos, sendo eles:

1. Desenvolver um algoritmo para particionar os contextos formais em subcontextos formais, utilizando a linguagem C++.
2. Desenvolver um algoritmo para unificar os conceitos formais gerados e validar se são conceitos formais do contexto formal original, utilizando a linguagem C++.
3. Avaliar ganho e performance para os algoritmos desenvolvidos quando submetidos contextos formais de diferentes dimensionalidades.

1.3 Justificativa

A era digital propiciou um acesso sem precedentes à informação, transformando a forma como coletamos, armazenamos e analisamos dados. No entanto, essa abundância de dados apresenta desafios significativos, especialmente no que diz respeito à análise eficiente e precisa de grandes volumes de informações. A Análise de Conceito Formal (FCA) emergiu como uma ferramenta para enfrentar essa complexidade, oferecendo uma metodologia robusta para a organização e análise de dados binários. A FCA permite a identificação de relações intrínsecas entre objetos e atributos, facilitando a extração de conhecimento a partir de dados complexos.

Apesar de suas vantagens, a aplicação da FCA em conjuntos de dados densos enfrenta sérias limitações computacionais. À medida que o número de objetos e atributos aumenta, a quantidade de combinações possíveis cresce exponencialmente, resultando em um aumento substancial no tempo e recursos necessários para realizar as análises. Este desafio é exacerbado pela alta dimensionalidade dos dados contemporâneos, que pode tornar inviável a aplicação de métodos tradicionais da FCA devido ao seu alto custo computacional.

Para mitigar essas limitações, diversas estratégias foram desenvolvidas para aumentar a eficiência da geração de conceitos formais. Uma dessas abordagens promissoras é o paralelismo, que oferece uma solução eficaz ao distribuir as tarefas de computação entre múltiplos processadores. O paralelismo não só reduz significativamente o tempo total de processamento, mas também melhora a escalabilidade das análises, permitindo a exploração de conjuntos de dados maiores e mais complexos.

A implementação do paralelismo na FCA pode acelerar a análise de dados, tornando-a mais viável para aplicações em larga escala. Além disso, essa abordagem possibilita a exploração de novas áreas de aplicação para a FCA, como mineração de dados, inteligência artificial e bioinformática.

Na mineração de dados, por exemplo, a capacidade de analisar rapidamente grandes volumes de dados pode levar à descoberta de padrões e tendências que seriam impraticáveis de identificar com métodos tradicionais. Na inteligência artificial, a FCA pode ser usada para melhorar algoritmos de aprendizado de máquina, fornecendo uma base sólida para a compreensão das relações entre diferentes variáveis. Na bioinformática, a análise eficiente de dados genômicos e proteômicos pode acelerar a descoberta de novas relações biológicas e avanços na medicina.

Portanto, a adoção de estratégias de paralelismo para a FCA não apenas supera as limitações computacionais atuais, mas também expande significativamente o potencial de aplicação dessa metodologia. Isso torna a FCA uma ferramenta ainda mais valiosa em um mundo cada vez mais orientado por dados, permitindo que pesquisadores e profissionais de diversas áreas aproveitem ao máximo as vastas quantidades de informações disponíveis na era digital.

1.4 Organização da Dissertação

Essa seção aborda sobre a organização do restante desta dissertação. O Capítulo 2 apresenta a fundamentação teórica sobre a análise formal de conceitos, tal como conceitos importantes sobre computação paralela e em como ela foi utilizada no presente trabalho. O Capítulo 3 mostra alguns trabalhos presentes na literatura que abordam aplicações de otimização em extração de conceitos formais. O Capítulo 4 descreve a metodologia utilizada neste trabalho, definindo as operações utilizadas para a obtenção dos conceitos formais a partir dos quasi-conceitos. No Capítulo 5 são apresentados exemplos e resultados dos experimentos executados neste trabalho. Por fim, no Capítulo 6 são abordadas as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este trabalho fundamenta-se em dois pilares essenciais: Análise Formal de Conceitos e Computação Paralela. FCA é uma teoria matemática que explora relações entre objetos e atributos através da formação de conceitos formais, facilitando a organização e interpretação de grandes volumes de dados.

Por sua vez, Computação Paralela utiliza técnicas como paralelismo de dados e de tarefas para aumentar a eficiência computacional, essencial para o processamento rápido e escalável de grandes conjuntos de dados. Integrando esses conceitos, este trabalho visa otimizar a aplicação da FCA através de técnicas avançadas de computação paralela.

Com o objetivo de aprofundar melhor sobre estes conceitos, esta seção será separada da seguinte forma: a Seção 2.1 contextualiza sobre FCA e sobre essa área de estudo, assim como o que é um contexto formal e um conceito formal, a Seção 2.2 aborda sobre o conceito de computação paralela, sua origem, assim como suas classificações e as ferramentas neste contexto utilizadas.

2.1 Análise Formal de Conceitos

A Análise Formal de Conceitos (FCA) é uma área da análise de dados baseada na teoria dos reticulados (lattices) que organiza e representa a relação entre um conjunto de objetos e um conjunto de atributos. Introduzida por Rudolf Wille na década de 1980, a FCA proporciona uma estrutura matemática para identificar e explorar conceitos formados por agrupamentos de objetos que compartilham atributos comuns.

A FCA tem suas raízes na *lattice theory*, uma área da matemática que estuda estruturas algébricas conhecidas como reticulados. O primeiro trabalho significativo de Wille sobre o assunto foi publicado em 1982, intitulado *Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts* (WILLE, 1982). Wille desenvolveu a FCA com o objetivo de criar uma metodologia sistemática para a análise de dados qualitativos, inspirado por ideias da teoria dos conjuntos e da teoria das relações.

A partir desse trabalho, a FCA evoluiu como uma abordagem para a classificação e interpretação de dados. Desde então, a FCA tem sido aplicada em uma ampla gama de áreas, incluindo ciência da computação, linguística, biologia e ciências sociais.

2.1.1 Contexto Formal

Na Análise Formal de Conceitos (FCA), um contexto formal $K = (G, M, I)$ é uma estrutura fundamental que representa a interação entre um conjunto de objetos G, um conjunto de atributos M, e uma relação binária I que indica quais objetos possuem quais atributos. Esta abordagem oferece uma representação poderosa para a organização e análise de dados complexos, onde cada objeto pode ser descrito por um conjunto de características ou atributos.

O conjunto G, muitas vezes denominado conjunto de entidades, representa os objetos de interesse em um domínio específico. Por exemplo, em um contexto de análise de pacientes, os objetos podem ser indivíduos específicos. O conjunto M, por outro lado, consiste nos atributos que descrevem os objetos. Esses atributos podem variar amplamente dependendo do domínio de aplicação, incluindo características físicas, sintomas médicos, preferências de consumo, entre outros. A relação I especifica quais objetos possuem quais atributos, formando uma matriz binária onde cada célula indica a presença (ou ausência) de um atributo em um objeto.

A Tabela 1 exemplifica um contexto formal típico com $|\mathbf{G}| = 8$ objetos e $|\mathbf{M}| = 9$ atributos, mostrando como os objetos G estão associados aos atributos M.

Tabela 1 – Contexto formal diático

	precisa de água para viver	vive na água	vive na terra	precisa de clorofila	dicotiledônea	monocotiledônea	pode se mover	tem membros	amamentam
sanguessuga	X	X					X		
brema	X	X					X	X	
sapo	X	X	X				X	X	
cão	X		X				X	X	X
ervas daninhas	X	X		X		X			
junco	X	X	X	X		X			
feijão	X		X	X	X				
milho	X		X	X		X			

Fonte: Elaborada pelo autor.

Nesta tabela, cada célula (g, m) contém um marcador (como 'x') para indicar a presença do atributo m no objeto g. Por exemplo, o objeto "sanguessuga"possui os atributos ("precisa de água para viver", "vive na água", "pode se mover"), enquanto o objeto "brema"possui ("precisa de água para viver", "vive na água", "pode se mover", "tem membros").

2.1.2 Conceito Formal

Um conceito formal em um contexto formal $K = (G, M, I)$ é representado por um par (A, B), onde $A \subseteq G$ e $B \subseteq M$. Esses conjuntos são definidos de acordo com operadores de derivação que refletem a interação entre objetos e atributos no contexto:

$$A' = \{b \in M \mid \forall a \in A, aIb\} \tag{2.1}$$

$$B' = \{a \in G \mid \forall b \in B, aIb\} \tag{2.2}$$

Onde A' representa a extensão de A, isto é, todos os objetos em G que possuem todos os atributos em B. Por outro lado, B' é a intenção de B, ou seja, todos os atributos em M que são compartilhados por todos os objetos em A.

Esse conceito de par (A, B) permite uma visão hierárquica dos dados, identificando grupos de objetos que compartilham características comuns. Por exemplo, na Tabela 1, o par $(\{"sapo", "junco"\}, \{"precisadeáguaparaviver", "vivenaágua", "vivenaterra"\})$ pode ser considerado um conceito formal. Aqui, $\{"sapo", "junco"\}$ é A, que possui os atributos $\{"precisadeáguaparaviver", "vivenaágua", "vivenaterra"\}$ como B. Extrair a extensão para B retorna os objetos $\{"sapo", "junco"\}$, enquanto extrair a intenção para A retorna os atributos $\{"precisadeáguaparaviver", "vivenaágua", "vivenaterra"\}$.

A Tabela 2 ilustra alguns conceitos formais derivados dos dados apresentados na Tabela 1, exemplificando como diferentes conjuntos de objetos e atributos podem formar conceitos formais significativos.

Tabela 2 – Exemplos de Conceitos Formais

A (Objetos)	B (Atributos)
{"sanguessuga", "brema"}	{"precisa de água para viver", "vive na água", "pode se mover"}
{"gato", "cachorro"}	{"tem cauda", "anda sobre quatro patas"}
{"maçã", "laranja"}	{"é uma fruta", "tem sementes"}

Esses exemplos demonstram como a FCA pode ser aplicada para identificar relações complexas entre objetos e atributos, oferecendo uma metodologia poderosa para a

organização e interpretação de dados. A capacidade de extrair conceitos formais não apenas facilita a análise exploratória de dados, mas também suporta a descoberta de padrões e estruturas ocultas em conjuntos de dados diversos.

2.1.3 Algoritmos para obtenção de conceitos formais

Algoritmos para obtenção de conceitos formais desempenham um papel crucial na Análise Formal de Conceitos (FCA), uma metodologia robusta utilizada para a análise e organização de dados complexos.

Esses algoritmos são projetados para identificar e representar estruturas hierárquicas de conceitos formados por conjuntos de objetos que compartilham atributos comuns. Em um contexto formal, eles exploram as relações intrínsecas entre objetos e atributos, permitindo não apenas a geração sistemática de conceitos, mas também facilitando a interpretação e a extração de conhecimento significativo dos dados.

Um dos algoritmos mais utilizados na FCA é o In-Close (ANDREWS; ORPHANIDES, 2010), que se destaca por sua capacidade de lidar com contextos formais complexos. O In-Close opera identificando conjuntos de atributos que são máximos em relação aos objetos em um contexto, garantindo que não possam ser expandidos sem alterar a definição dos conceitos encontrados.

A última versão, In-Close 4, representa um progresso significativo ao enfrentar desafios apresentados por conjuntos de dados densos e de alta dimensionalidade. Essa versão permite uma análise mais profunda da estrutura dos dados, o que é essencial para explorar relações complexas e extrair insights valiosos para aplicações diversas, incluindo mineração de dados, inteligência artificial e bioinformática.

O algoritmo In-Close é amplamente adotado por pesquisadores e profissionais que buscam compreender a estrutura dos dados e identificar padrões ocultos. Sua capacidade de lidar eficientemente com grandes volumes de dados e de capturar nuances nas relações entre objetos e atributos o torna uma ferramenta indispensável em áreas onde a análise detalhada e a interpretação de dados são cruciais para avanços científicos e tecnológicos. Portanto, o desenvolvimento e aprimoramento contínuos desses algoritmos são fundamentais para expandir os horizontes da FCA e explorar novas fronteiras na análise de dados complexos.

2.2 Computação Paralela

A computação paralela, conforme descrita nos trabalhos de Pacheco *et al.*(PACHECO, 2011)) e Robey *et al.*(ROBEY; ZAMORA, 2021), refere-se a uma abordagem computa-

cional onde vários cálculos ocorrem simultaneamente. Este método baseia-se no conceito de que grandes problemas podem ser particionados em problemas menores, permitindo que sejam resolvidos simultaneamente, reduzindo assim o tempo total de processamento. Aplicações de computação paralela são encontradas em diversos campos, como criptografia/descriptografia de dados, treinamento de modelos e processamento de imagens 3D, conforme indicado nos estudos de Yazdeen *et al.*(YAZDEEN et al., 2021), Kitrungrotsakul *et al.* (KITRUNGROTSAKUL et al., 2020)) e Zeng *et al.*(ZENG et al., 2021).

Aproveitar núcleos de processamento para gerenciar múltiplas tarefas simultaneamente melhora substancialmente o desempenho geral. Além disso, as ferramentas de computação paralela otimizam os recursos computacionais ao particionar uma tarefa maior em várias subtarefas, executando-as simultaneamente em vários núcleos.

Uma dessas ferramentas é a OpenMP API (MATTSON; HE; KONIGES, 2019; PAS; STOTZER; TERBOVEN, 2017), que oferece um caminho para aproveitar o paralelismo em máquinas com memória compartilhada. Além disso, o OpenMP pode ser utilizado para computação heterogênea, abrangendo Unidades Centrais de Processamento (CPUs) e Unidades de Processamento Gráfico (GPUs), permitindo processamento paralelo versátil e eficiente.

Discutir fórmulas de ganho (S) e eficiência (E) na análise de algoritmos paralelos é essencial. Estas métricas são fundamentais para quantificar as melhorias de desempenho alcançadas através da paralelização. o ganho é comumente expressa como a razão entre o tempo de execução de um algoritmo sequencial (T_1) e o de um algoritmo paralelo com processadores P (T_P), encapsulado pela fórmula:

$$S = \frac{T_1}{T_P}$$

Notavelmente, o ganho superlinear surge quando o tempo de execução do algoritmo paralelo ultrapassa o limite teórico previsto pelo ganho linear definida pela lei de Amdalh (PACHECO, 2011).

Apesar da limitação do algoritmo paralelo devido à sua porção sequencial, existem mais recursos para execução paralela. Por exemplo, existem muitos caches para execução paralela. Em outras palavras, a limitação de recursos que resulta em perdas de cache para uma versão sequencial não está presente na mesma condição para uma versão paralela. Assim, o ganho superlinear não envolve apenas algoritmos paralelos, mas também arquiteturas paralelas.

A eficiência, denotada por E, fornece ideias sobre a utilização de processadores adicionais e é definida matematicamente como a razão entre o ganho (S_P) e o número de processadores:

$$E = \frac{S_P}{P}$$

Além disso, a discussão se estende à escalabilidade, abrangendo conceitos de escalabilidade forte e fraca. A escalabilidade forte mede a eficiência de um algoritmo paralelo, pois o tamanho do problema é fixo enquanto o número de processadores varia. Por outro lado, a escalabilidade fraca avalia a eficiência do algoritmo à medida que o tamanho do problema e o número de processadores são dimensionados proporcionalmente.

2.2.1 Escalabilidade

Na computação paralela, *escalabilidade* refere-se à capacidade de um sistema ou algoritmo de manter ou melhorar o desempenho conforme o número de recursos, como processadores ou núcleos, é aumentado. A escalabilidade pode ser classificada em duas categorias principais: *escalabilidade forte* e *escalabilidade fraca*.

Escalabilidade forte, ou escalabilidade de problema fixo, refere-se à capacidade de um sistema de manter o desempenho constante quando o problema é mantido fixo e o número de processadores é aumentado. Essa forma de escalabilidade é medida pelo *speedup* (velocidade), que é a razão entre o tempo de execução com um único processador e o tempo de execução com p processadores. Idealmente, o speedup deve ser aproximadamente igual ao número de processadores p.

A Lei de Amdahl fornece uma perspectiva sobre a escalabilidade forte. Segundo a Lei de Amdahl, o speedup máximo de um programa paralelo é limitado pela fração do código que não pode ser paralelizada. A Lei é expressa pela fórmula:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{2.3}$$

onde $S(p)$ é o speedup com p processadores, e f é a fração do código que pode ser paralelizada. A fórmula mostra que a melhoria no desempenho é limitada pela parte não paralelizável do código, mesmo que a fração paralelizável f seja alta.

Escalabilidade fraca, ou escalabilidade de problema variável, refere-se à capacidade de um sistema de manter o tempo de execução constante quando o tamanho do problema é aumentado proporcionalmente ao número de processadores. Na escalabilidade fraca, o tamanho do problema aumenta com o número de processadores de forma que a carga de trabalho total é proporcional ao número de processadores. O objetivo é manter o tempo de execução constante, independentemente do aumento da carga de trabalho. Assim, a carga de trabalho deve aumentar proporcionalmente ao número de processadores para que o tempo de execução permaneça constante.

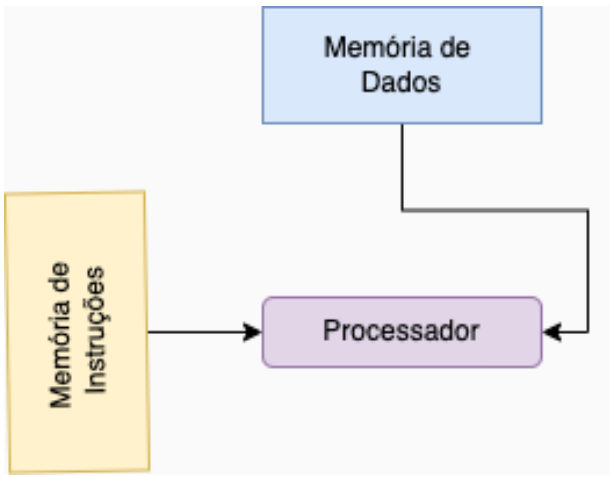
Portanto, a escalabilidade forte e a escalabilidade fraca são usadas para avaliar a eficiência dos sistemas de computação paralela. A escalabilidade forte foca na redução do tempo de execução para um problema fixo com mais processadores, enquanto a escalabilidade fraca concentra-se em manter o tempo de execução constante à medida que o tamanho do problema é ajustado proporcionalmente ao número de processadores. A Lei de Amdahl ilustra as limitações da escalabilidade forte, mostrando como a parte não paralelizável do código pode limitar o desempenho máximo possível.

2.2.2 Taxonomia de Flynn

Em 1966, Michael J. Flynn propôs um sistema inovador para categorizar arquiteturas de computadores, utilizando como base o fluxo de instruções e o fluxo de dados. Essa taxonomia (FLYNN, 1966) diferencia as arquiteturas em quatro classes distintas:

1. *Single instruction, single data* (SISD): Cada unidade de processamento executa uma única instrução em um único dado por vez, como apresentado na Figura 1 . Essa é a arquitetura sequencial proposta por John von Neumann.

Figura 1 – SISD



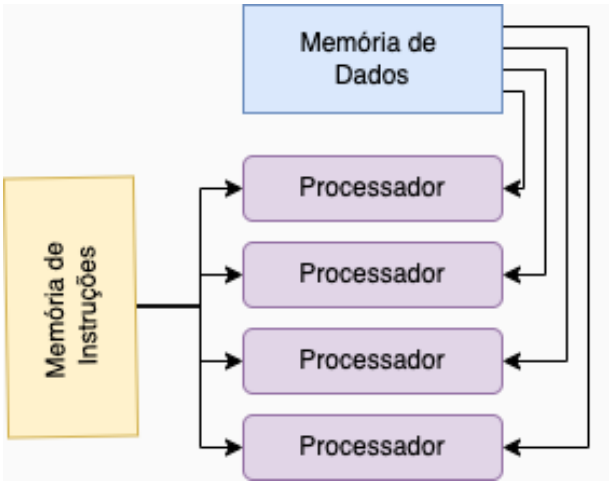
Fonte: Elaborada pelo autor.

2. *Single instrution, multiple data* (SIMD):Todas as unidades de processamento recebam a mesma instrução, aplicando-a simultaneamente a diferentes conjuntos de dados. Essa abordagem é amplamente utilizada em arquiteturas vetoriais e processadores gráficos (GPUs), onde o processamento de grandes volumes de dados semelhantes pode ser realizado de maneira eficiente.

   Uma das principais vantagens dessa abordagem é a eficiência no processamento paralelo, já que a mesma operação é aplicada simultaneamente a diferentes elementos

de dados. Isso reduz a quantidade de ciclos de clock necessários em comparação com métodos que requerem operações separadas para cada dado, resultando em um desempenho superior em tarefas que envolvem processamento intensivo de dados. Além disso, o espaço de endereçamento compartilhado simplifica o gerenciamento de dados e coordena o acesso a eles, minimizando a necessidade de comunicação entre threads e potencialmente reduzindo problemas de sincronização.

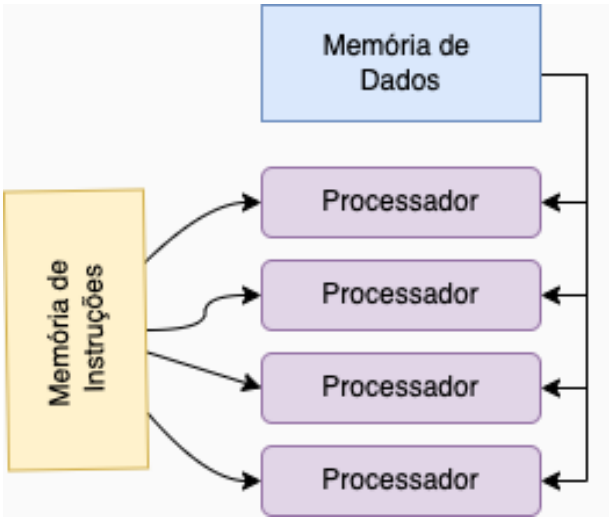
Figura 2 – SIMD



Fonte: Elaborada pelo autor.

3. *Multiple instructions, single data* (MISD): Cada unidade de processamento executa instruções distintas, mas todas operam sobre o mesmo dado, conforme a Figura 3. Essa arquitetura, menos comum, é utilizada em pesquisas e aplicações específicas (VALLEJO-MANCERO; ZAPATA, 2021).

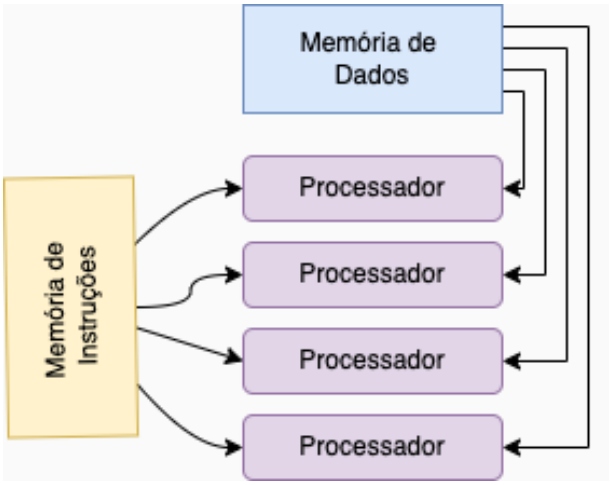
Figura 3 – MISD



Fonte: Elaborada pelo autor.

4. *Multiple instructions, multiple data* (MIMD): Cada unidade de processamento executa instruções diferentes e opera sobre dados distintos de forma simultânea, mostrado na Figura 4. Essa arquitetura é prevalente em computadores multiprocessados e em sistemas de computação em nuvem, proporcionando alto poder de processamento para tarefas complexas.

Figura 4 – MIMD



Fonte: Elaborada pelo autor.

A classificação de Flynn tornou-se um pilar fundamental na compreensão da diversidade de arquiteturas de computadores, permitindo que usuários, desenvolvedores e pesquisadores identifiquem a solução ideal para cada necessidade.

2.2.3 Memória Compartilhada

Na taxonomia de Flynn, a memória compartilhada oferece um meio comum de comunicação entre um grupo de processadores, permitindo que diversas unidades de processamento executem suas instruções, acessando e modificando dados compartilhados ou privados de forma simultânea.

Esse trabalho em conjunto torna-se possível em arquiteturas MIMD (Múltiplas Instruções, Múltiplos Dados), onde a comunicação interprocessador se torna mais rápida e eficiente, otimizando o desempenho geral do sistema.

As vantagens da memória compartilhada é que cada processador recebe a mesma instrução, mas a aplica em diferentes dados simultaneamente, ideal para tarefas que exigem processamento massivo de conjuntos de dados semelhantes, como imagens e vídeos.

No entanto, surgem alguns desafios: garantir que todos os processadores estejam na mesma página em relação aos dados (coerência de cache), evitar conflitos de acesso (sincronização de acesso) e manter a escalabilidade exige técnicas mais sofisticadas e inovadoras.

2.2.4 OpenMP

Em meio ao conjunto de processadores em arquiteturas MIMD, o *framework Open Multi-Processing* (OpenMP) surge como uma peça-chave na exploração paralela de forma eficiente e otimizada. O OpenMP, um conjunto de diretivas de pré-processador e funções de biblioteca, permite aos programadores explorar o poder da memória compartilhada de forma simples e intuitiva, maximizando o desempenho de aplicações *multithreading* em sistemas com memória compartilhada.

Através de diretivas como *parallel*, *for* e *sections*, o OpenMP direciona a execução de seções de código em paralelo, distribuindo tarefas entre os diferentes processadores da orquestra. Mecanismos de sincronização, como critical e barrier, garantem a harmonia entre os *threads*, evitando conflitos de acesso à memória compartilhada e assegurando a integridade dos dados.

O OpenMP se destaca por sua portabilidade, adaptando-se a diversas linguagens de programação como C, C++ e Fortran, permitindo que programadores de diferentes origens explorem os benefícios da memória compartilhada.

Em resumo, o OpenMP se configura como uma ferramenta eficaz para o desenvolvimento de aplicações *multithreading* em arquiteturas MIMD com memória compartilhada, simplificando a programação e otimizando o desempenho.

3 TRABALHOS RELACIONADOS

Moraes *et al.* (MORAES et al., 2016) contribuíram significativamente ao introduzir uma versão paralelizada do algoritmo Next Closure, focando na otimização do desempenho e na redução do tempo de execução, especialmente ao lidar com bancos de dados de alta dimensão. A implementação paralela foi uma resposta direta à necessidade de processar grandes volumes de dados de forma mais eficiente e escalável.

A principal conquista do estudo foi a redução significativa no tempo de execução do algoritmo, alcançada através da execução concorrente de até 16 *threads*. Esse aumento na eficiência permitiu que o algoritmo Next Closure fosse aplicado com sucesso em cenários onde o tempo de processamento anteriormente seria proibitivo.

No entanto, o trabalho de Moraes *et al.* também destacou desafios e áreas para futuras melhorias. A escalabilidade do algoritmo com o aumento do número de *threads* e a robustez em diferentes configurações de dados foram áreas identificadas para investigações adicionais. Essas considerações são cruciais para garantir que as implementações paralelas não apenas acelerem o processamento, mas também preservem a precisão e a integridade dos resultados em contextos complexos.

Além disso, o estudo de Moraes *et al.* sublinhou a importância contínua de estratégias de paralelização no campo da análise formal de conceitos, abrindo caminho para avanços adicionais na aplicação de técnicas paralelas para lidar com desafios emergentes em grandes conjuntos de dados e aplicações computacionais intensivas.

Kodagoda *et al.* (KODAGODA; ANDREWS; PULASINGHE, 2017) apresentaram uma contribuição significativa ao propor uma versão paralela do algoritmo In-Close (versão 3), utilizando a *Application Programming Interface* (API) *OpenMP* para aumentar sua eficiência computacional. A motivação principal por trás de seu trabalho foi explorar a capacidade de paralelização de algoritmos recursivos, como aqueles baseados em CBO (Colliding Bodies Optimization), que são conhecidos por sua estrutura recursiva complexa.

Ao implementar o *OpenMP*, os pesquisadores puderam explorar o paralelismo de forma eficaz, distribuindo tarefas entre vários núcleos de processamento. Isso resultou em melhorias significativas no tempo de execução do algoritmo In-Close, especialmente em contextos de dados densos e de alta dimensionalidade, onde a paralelização pode mitigar o impacto da complexidade computacional.

Embora Kodagoda *et al.* tenham alcançado sucesso ao apresentar a viabilidade

do *OpenMP* para paralelizar o algoritmo In-Close, eles também identificaram áreas para futuras melhorias. Em particular, sugeriram realizar comparações mais abrangentes com outros algoritmos paralelos para validar e aprimorar sua metodologia.

Essas comparações poderiam incluir métricas como tempo de execução, escalabilidade com o aumento do número de *threads* e eficiência na manipulação de grandes volumes de dados. Esse enfoque é crucial para garantir que as técnicas paralelas aplicadas não apenas acelerem o processamento, mas também mantenham a precisão e a robustez na análise de conceitos formais em diferentes cenários aplicados.

Novais *et al.* (NOVAIS et al., 2021) realizaram um avanço significativo ao desenvolver um algoritmo baseado em *Open Computing Language (OpenCL)* para extração de conceitos formais, utilizando uma abordagem de força bruta paralela em uma arquitetura heterogênea, que incluía tanto *Central Processing Unit* (CPU)+*Graphics Processing Unit* (GPU) quanto CPU+*Field Programmable Gate Array* (FPGA). A escolha de *OpenCL* como plataforma permitiu que o algoritmo fosse executado de forma eficiente em diferentes dispositivos de hardware, aproveitando ao máximo o potencial de processamento paralelo dessas unidades de processamento.

Um dos principais destaques do trabalho foi o desempenho excepcional alcançado. Comparado ao algoritmo *Data-Peeler* no mesmo contexto, o algoritmo proposto por Novais *et al.* superou significativamente, apresentando um fator de melhoria notável de 18. Essa diferença substancial ilustra a eficácia da abordagem paralela em *OpenCL* para acelerar a análise de conceitos formais, mesmo em contextos de dados complexos e volumosos.

Além do desempenho aprimorado, Novais *et al.* também destacaram a eficiência energética superior alcançada por seu algoritmo. A redução no consumo de energia em comparação com algoritmos alternativos em diversas arquiteturas sublinha a importância não apenas da velocidade de processamento, mas também da sustentabilidade e economia de recursos em aplicações computacionais intensivas.

Contudo, o estudo também identificou desafios a serem abordados em pesquisas futuras, como a otimização adicional do uso de recursos heterogêneos e a adaptação do algoritmo para diferentes classes de problemas e conjuntos de dados. Essas considerações são fundamentais para impulsionar ainda mais o campo da análise formal de conceitos em direção a soluções mais eficientes e adaptáveis às demandas computacionais contemporâneas.

Fu e Nguifo (FU; NGUIFO, 2004) contribuíram significativamente para o campo da análise formal de conceitos ao desenvolver um algoritmo paralelo voltado para a extração eficiente de conceitos em espaços de busca. O cerne de sua abordagem estava na remoção de atributos redundantes do contexto antes da geração da partição, baseando-se no algoritmo *NextClosure*. Este algoritmo é conhecido por sua eficácia em identificar fe-

chamentos em conjuntos de dados, o que é crucial para a geração de conceitos formais de maneira eficiente.

No entanto, um desafio identificado foi a falta de balanceamento adequado da carga de trabalho entre os diferentes processadores utilizados no paralelismo. Isso resultou em flutuações de desempenho dependendo do fator de carga escolhido para distribuir as tarefas, o que impactou a consistência e a previsibilidade dos tempos de execução.

Apesar dessa limitação, o estudo de Fu e Nguifo mostraram o potencial das abordagens paralelas na manipulação de contextos de entrada de alta dimensão. A capacidade de processar grandes conjuntos de dados de forma distribuída é crucial para lidar com o aumento na complexidade e no tamanho dos dados encontrados em aplicações práticas, como na mineração de dados, inteligência artificial e bioinformática. Assim, a pesquisa não apenas destacou os desafios, mas também abriu novas perspectivas para melhorias futuras na eficiência e escalabilidade dos algoritmos de análise formal de conceitos.

Zhou *et al.* (ZHOU et al., 2021) contribuíram significativamente ao apresentar uma versão aprimorada do já estabelecido algoritmo In-Close, denominada In-Close 5. Uma das principais inovações introduzidas foi o uso de uma matriz de bits vertical para armazenar o contexto do conceito e sua extensão. Essa abordagem foi projetada especificamente para mitigar as ineficiências encontradas na versão anterior, In-Close 4, especialmente durante a intersecção de conjuntos de atributos.

Os resultados experimentais obtidos pelos autores apresentaram a eficácia substancialmente superior do algoritmo proposto, mesmo em cenários onde o In-Close 4 encontrava dificuldades para produzir resultados precisos e eficientes. Essa melhoria não apenas aumentou a velocidade de processamento, mas também melhorou a precisão na identificação de conceitos formais, essenciais para a análise estruturada e hierárquica de conjuntos de dados complexos.

A proposta do In-Close 5 representa um avanço significativo no campo da análise formal de conceitos, oferecendo uma solução mais robusta e eficiente para lidar com desafios computacionais em grandes volumes de dados. Além disso, abre novas possibilidades para aplicações práticas em áreas como mineração de dados, inteligência artificial e bioinformática, onde a análise precisa e rápida de relações entre objetos e atributos é fundamental para a descoberta de conhecimento e tomada de decisões informadas.

Em contraste com esses estudos, o presente trabalho se destaca ao desenvolver uma abordagem que envolve a criação de um *wrapper* para executar o algoritmo In-Close 4, juntamente com três algoritmos para separar, computar e unificar os quasi-conceitos, gerando os conceitos formais. Este método distingue este estudo dos demais na área.

A Tabela 3 fornece uma visão abrangente das semelhanças e distinções entre a

nossa abordagem e os trabalhos relacionados.

Tabela 3 – Comparação das características entre o trabalho atual e os trabalhos relacionados

Características	**Trabalho proposto**	(MORAES et al., 2016)	(KODAGODA; ANDREWS; PULA-SINGHE, 2017)	(NOVAIS et al., 2021)	(FU; NGUIFO, 2004)	(ZHOU et al., 2021)
Algoritmo Paralelo	In-Close 4	Next Closure	In-Close 3	Customizado	Next Closure	In-Close 5
Framework de Paralelização	C++/OpenMP	OpenMP	OpenMP	OpenCL	N/A	N/A
Foco dos experimentos	Contextos sintéticos	Bases de dados de alta dimensionalidade	Algoritmo Recursivo baseado no CBO	Arquitetura Heterogênea	Espaços de busca	Contexto formal com conceitos formais e extensão
Principal contribuição	Abordagem de divisão de Contexto Formal	Redução do tempo de execução	Algoritmos de paralelização recursiva	Paralelismo heterogêneo	Remoção de atributos de maneira redundante	Aperfeiçoamento do In-Close 4
Ênfase de comparação	Obtenção de Conceitos Formais em paralelo	–	–	Outros algoritmos paralelos	Impacto do fator de carga	Limitações do In-Close 4

Fonte: Elaborada pelo autor.

4 ABORDAGEM PROPOSTA

Para o desenvolvimento deste trabalho, foi estabelecida uma abordagem que apresenta os recursos e os métodos utilizados para a produção dos algoritmos. A estrutura seguirá a seguinte forma: na Seção 4.1 é discutida qual arquitetura foi utilizada neste trabalho, assim como a justificativa do porquê a mesma foi escolhida. Na Seção 4.2 é definido o termo *quasi-conceito*, tal como as operações necessárias para a obtenção dos conceitos formais a partir dos mesmos. Na Seção 4.3 são enumerados todos os recursos necessários para o desenvolvimento do projeto. Na Seção 4.4 são apresentados as etapas de desenvolvimento dos algoritmos propostos neste trabalho, explicando resumidamente as etapas do mesmos, assim como os conceitos utilizados.

4.1 Arquitetura

A arquitetura escolhida para a implementação da extração de conceitos formais no presente trabalho é baseada no modelo MIMD, que se destaca por sua capacidade de executar múltiplas instruções simultaneamente sobre dados distintos. Esse modelo é particularmente adequado para tarefas que envolvem grandes volumes de dados, como a extração de conceitos a partir de contextos formais. A principal vantagem deste modelo é sua capacidade de realizar operações independentes em paralelo, permitindo que diferentes subconjuntos de dados sejam processados simultaneamente por diferentes núcleos ou threads, sem a necessidade de interdependência entre eles.

A sua utilização oferece benefícios significativos em termos de desempenho e escalabilidade. Em contextos formais, onde é necessário processar grandes quantidades de objetos e atributos, a paralelização das operações pode reduzir substancialmente o tempo de execução. Além disso, à medida que a quantidade de dados aumenta, a arquitetura MIMD garante que o sistema continue a escalar de forma eficiente, utilizando plenamente os recursos computacionais disponíveis, como núcleos de processadores em sistemas multiprocessados.

A escolha deste modelo também assegura flexibilidade no processamento, uma vez que cada processador pode executar uma tarefa distinta, adaptando-se a diferentes operações ou subconjuntos de dados. Esse comportamento é essencial quando lidamos com contextos heterogêneos ou grandes volumes de dados distribuídos. Por fim, a compatibilidade com arquiteturas modernas, como clusters de servidores e processadores multinúcleos, faz do MIMD uma escolha natural para garantir alto desempenho e eficiência no processamento da extração de conceitos formais.

4.2 Quasi-conceitos

Para a elaboração deste trabalho, utilizou-se o termo denominado *Quasi-conceito*. Um *Quasi-conceito* é um conceito obtido a partir de uma partição de um contexto formal (também chamado neste projeto de subcontexto formal). O seu nome foi dado para indicar que pode ou não ser um conceito formal, quando comparado ao contexto formal original.

A ideia dos quasi-conceitos surge da necessidade de lidar com grandes volumes de dados, onde a análise direta dos conceitos formais pode se tornar impraticável devido à complexidade e ao tempo computacional requerido. Ao particionar o contexto formal original em subcontextos menores, torna-se possível identificar conceitos formais em cada subcontexto de maneira mais eficiente. Esses conceitos, obtidos dos subcontextos, são denominados quasi-conceitos.

4.2.1 Processo de Extração de Quasi-conceitos

O processo de extração de quasi-conceitos envolve várias etapas:

1. **Particionamento do Contexto Formal Original**: O contexto formal original é particionado em vários subcontextos menores. Esse particionamento pode ser baseada em diversos critérios, como a distribuição dos objetos e atributos, ou mesmo pela aplicação de algoritmos específicos de particionamento.

2. **Identificação de Conceitos Formais em Subcontextos**: Para cada subcontexto, são identificados os conceitos formais utilizando técnicas tradicionais de FCA. Esses conceitos identificados são os quasi-conceitos.

3. **Agregação dos Quasi-conceitos**: Os quasi-conceitos extraídos de cada subcontexto são então agregados para formar um conjunto representativo de conceitos. Essa agregação é essencial para garantir que as relações importantes entre os objetos e atributos sejam preservadas e analisadas de forma holística.

4. **Validação dos Quasi-conceitos**: A etapa final envolve a validação dos quasi-conceitos para verificar se eles correspondem ou não a conceitos formais no contexto original. Essa validação é crucial para assegurar a integridade e a utilidade dos conceitos extraídos.

4.2.2 Operações da obtenção de quasi-conceitos

Dado $K(G, M, I)$ uma tripla que representa um contexto formal, particionado em subcontextos $K_i(G_i, M, I_i)$, $1 \leq i \leq n$, onde $G_p \cap G_q = \emptyset$ para todos $p \neq q$ (considerando p e q como representações de dois subcontextos formais, e $G_1 \cup G_2 \cup ... \cup G_n = G$. É importante notar que G_n consiste em uma partição de G.

Considere os quasi-conceitos (A_p, B_p) e (A_q, B_q), obtidos dos subcontextos K_p e K_q, A representando um grupo de objetos de K e B representando um conjunto de atributos de K. Para derivar os conceitos finais com base nos quasi-conceitos gerados a partir de cada subcontexto K_i, são necessárias as seguintes operações:

1. Se $B_p \cap B_q = \emptyset$, então os quasi-conceitos são de fato conceitos.

   Prova: Como $B_p \cap B_q = \emptyset$, segue-se que $B'_p = A_p$ e $A'_p = B_p$ é válido para todo o contexto K. O mesmo princípio se aplica a qualquer (A_q, B_q).

   A Tabela 4 ilustra o contexto K. Para simplificar, empregamos dois subcontextos K_1 e K_2; entretanto, o método idêntico pode ser facilmente estendido a n subcontextos.

Tabela 4 – Exemplo para interseção

	a	b	c
1	X	X	
2	X		
3	X	X	
4			X
5			X
6			X

Fonte: Elaborada pelo autor.

   Os quasi-conceitos ({1, 2, 3}, {a, b}) e ({1}, {a, b, c}) são derivados do subcontexto K_1, enquanto ({4, 5, 6}, {a}) é um quasi-conceito obtido do subcontexto K_2. Dado que $\{a\} \cap \{b\} = \emptyset$, o par ({1, 2, 3}, {a}) incorpora um conceito dentro do contexto mais amplo K. Isto é evidente como {a}' = {1, 2, 3}, e {1, 2, 3}' = {a}, abrangendo todo o contexto K.

   Da mesma forma, ({1, 3}, {a, b}) também é um conceito para todo o contexto K, uma vez que {a, b}' = {1, 3}, e {1, 3}' = {a, b}, dado que {a, b} ∩ {c} = ∅. Da mesma forma, ({4, 5, 6}, {c}) é um conceito devido ao mesmo princípio.

2. Se $B_p \cap B_q \neq \emptyset$, o conjunto de atributos $B_p \cap B_q$ é compartilhado por todos os conjuntos de objetos A_p e A_q, significando a presença de um conceito formal com objetos $A_p \cup A_q$ e atributos $B_p \cap B_q$.

   Prova: Seja X = $A_p \cup A_q$ e Y = $B_p \cap B_q$. Se $B_p \cap B_q \neq \emptyset$, isso implica $Y' = X$. Considerando $X' = Y$, confirma que ($A_p \cup A_q$, $B_p \cap B_q$) é de fato um conceito dentro de K.

   A Tabela 5 ilustra uma instância de dois subcontextos dentro do contexto K, mostrando que a mesma abordagem se estende facilmente a n subcontextos.

Tabela 5 – Exemplo para quasi-conceitos que não são conceitos

	a	b	c
1	X	X	X
2	X	X	X
3	X	X	X
4	X	X	X
5	X		X
6	X		X

Fonte: Elaborada pelo autor.

No exemplo dado, o subcontexto K_1 produz o quasi-conceito $(\{1\}, \{a, b, c\})$, enquanto K_2 produz $(\{4\}, \{a, b, c\})$. Esses quasi-conceitos compartilham o conjunto de atributos $\{a, b, c\}$, indicando a presença de um conceito formal $(\{1, 4\}, \{a, b, c\})$. Este conceito emerge da interseção dos conjuntos de intenção, resultando em $\{a, b, c\}$, e da união dos conjuntos de extensão $\{1\} \cup \{4\}$ igualando a $\{1, 4\}$. K_1 gera o quasi-conceito $(\{1, 2, 3\}, \{a, b\})$, e K_2 dá origem a $(\{4, 5, 6\}, \{a\})$.

Esses quasi-conceitos compartilham o conjunto de atributos $\{a\}$, levando ao conceito formal $(\{1, 2, 3, 4, 5, 6\}, \{a\})$. Este conceito surge da intersecção dos conjuntos de intenção, resultando em $\{a\}$, e da união dos conjuntos de extensão $\{1, 2, 3\} \cup \{4, 5, 6\}$ igualando $\{1, 2, 3, 4, 5, 6\}$.

3. Quando $B_p \subseteq B_q$, os quasi-conceitos tornam-se subconceitos dentro de um conceito no contexto K. Para verificar isso, é essencial confirmar se um dos conjuntos de atributos é um subconjunto de o outro e se o conjunto de objetos de um quasi-conceito for um subconjunto dos objetos de um conceito extraído (Tabela 6).

Tabela 6 – Exemplo para quasi-conceitos que são conceitos

	a	b	c
1	X		X
2	X	X	X
3	X		X
4	X	X	
5	X		X
6	X		X

Fonte: Elaborada pelo autor.

No subcontexto inicial K_1, existe um quasi-conceito $(\{1,2,3\},\{a,c\})$, e no subcontexto subsequente K_2, outro quasi-conceito $(\{4,5,6\},\{a,c\})$ está presente. No entanto, nenhum deles se qualifica como conceitos, uma vez que compartilham a intenção $\{a,c\}$, que está englobada no conceito $(\{1,2,3,4,5,6\},\{a,c\})$ derivado da regra 2.

4. Supremo e ínfimo

   Para calcular o supremo e o ínfimo, quando eles não são gerados como quasi-conceitos, é necessário definir o ínfimo como o conjunto de objetos que compartilham todos os atributos e o supremo como o conjunto de atributos compartilhados por todos os objetos:

   - Se nenhum conjunto ínfimo foi definido ainda, o conceito formal ínfimo é definido como $(\emptyset,\{b_1 \cup ... \cup b_n\})$, onde b_1 até b_n representam todos os atributos do contexto K.
   - Se nenhum conjunto supremo foi definido ainda, conceito formal supremo é definido como $(\{a_1 \cup ... \cup a_n\},\emptyset)$, onde a_1 até a_n representam todos os objetos do contexto K.

4.3 Recursos

Os recursos utilizados neste trabalho foram essenciais para o desenvolvimento e a avaliação do algoritmo de análise de conceitos formais. São eles:

- **Linguagem C++:** Utilizada para implementar o algoritmo de análise de conceitos formais devido à sua eficiência e flexibilidade na manipulação de grandes volumes de dados.
- **Biblioteca OpenMP:** Utilizada para a paralelização do processamento, permitindo a distribuição de tarefas entre múltiplos núcleos de CPU e melhorando significativamente o desempenho computacional.
- **Compilador *GNU Compiler Collection* (GCC):** Utilizado para compilar e otimizar o código-fonte em C++, oferecendo suporte robusto ao padrão OpenMP e maximizando a eficiência do código paralelizado.
- ***Synthetic Context Generator* (SCGaz) para geração de base de dados sintética:** Utilizado para criar conjuntos de dados sintéticos que simulam cenários complexos de análise, permitindo testes extensivos e controle sobre as características dos dados utilizados.

Além disso, foram utilizadas duas máquinas para testes em diferentes estágios do desenvolvimento:

- **Máquina inicial para os primeiros testes:** Apple® MacBook Pro® com processador Intel® Core i7® @ 2.60GHz, 16GB de *Random Access Memory* (RAM) e 12 threads. Esta máquina foi utilizada na fase inicial para desenvolvimento e testes preliminares do algoritmo.

- **Máquina dos testes finais:** Intel Xeon E5645 com 2.40GHz, doze núcleos, caches L2 privados, 64GB de RAM e 1TB de armazenamento em disco. Esta máquina foi utilizada para os testes finais e avaliação de desempenho do algoritmo em cenários de uso intensivo de recursos computacionais.

Esses recursos proporcionaram um ambiente adequado para desenvolvimento, otimização e validação do algoritmo de análise de conceitos formais, garantindo resultados confiáveis e escaláveis para diferentes configurações de dados e cargas de trabalho computacional.

4.4 Algoritmo proposto: ConceptsFinder

Diante disso, um algoritmo foi proposto, o *ConceptsFinder*, com o intuito facilitar a análise de grandes volumes de dados utilizando técnicas de geração e manipulação de conceitos formais.

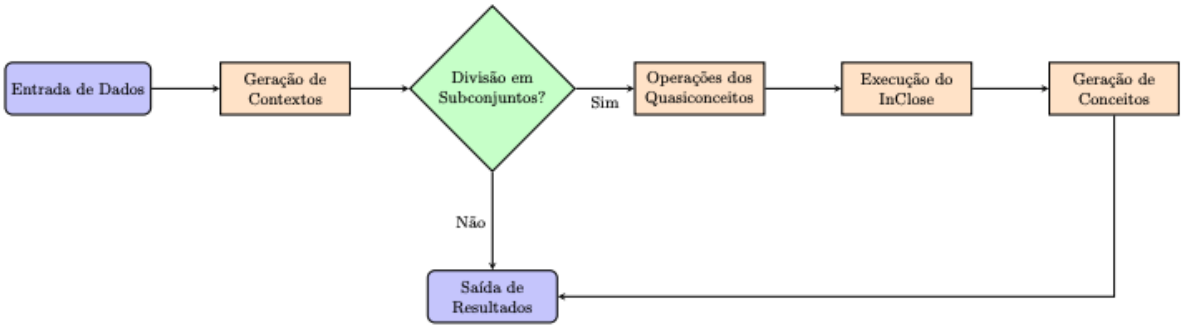
Baseado em princípios de paralelismo e em estratégias como a aplicação do In-Close, o sistema se destina a particionar arquivos de contexto em partes menores e processá-los simultaneamente para otimizar o tempo de execução.

Posteriormente, os conceitos formais gerados são agregados e analisados para identificar padrões significativos e relações entre objetos e atributos nos conjuntos de dados.

Esse enfoque não apenas melhora a eficiência da análise, mas também amplia a capacidade de exploração de informações em ambientes com grandes quantidades de dados, como na área de mineração de dados e inteligência artificial. Para explicar o seu funcionamento, foram separadas seções que incluem as respectivas funcionalidades realizadas pelo algoritmo proposto.

Na figura 5 temos a estrutura do algoritmo representada no formato de fluxograma, onde cada processo é abordado em uma seção ao decorrer do texto.

Figura 5 – Fluxograma do funcionamento do algoritmo



Fonte: Elaborada pelo autor.

O algoritmo apresentado abaixo mostra o corpo principal do algoritmo, o qual é responsável pela geração e particionamento de arquivos de contexto, seguido pela execução de um algoritmo de fechamento formal (InClose) nos contextos gerados. O processo é iniciado com a inicialização de vetores para armazenar objetos, atributos, partições e densidades. Os vetores objects, attributes, divisions e densities são criados para organizar e manipular os dados necessários ao longo do algoritmo. Em seguida, os valores são inseridos nesses vetores: os valores para objects variam de 100.000 a 1.000.000 com incrementos de 100.000; os valores para attributes variam de 10 a 100 com incrementos de 10; e os valores para densities variam de 10 a 70 com incrementos de 10.

O algoritmo começa com a inicialização de quatro vetores: ‘objects‘, ‘attributes‘, ‘divisions‘ e ‘densities‘ (linha 2 a 5). Os vetores ‘objects‘ e ‘attributes‘ são populados com intervalos específicos de números usando loops ‘for‘ (linhas 7-13). O vetor ‘objects‘ recebe valores de 100.000 a 1.000.000 com incrementos de 100.000 (linha 7-9), enquanto o vetor ‘attributes‘ recebe valores de 10 a 100 com incrementos de 10 (linhas 10-12). O vetor ‘densities‘ é preenchido com valores de 10 a 70, também com incrementos de 10 (linhas 13-15).

Depois de preencher esses vetores, o algoritmo gera arquivos de contexto utilizando os valores dos vetores ‘objects‘, ‘attributes‘ e ‘densities‘, chamando a função ‘ContextFile::generateContextFiles‘ (linha 16). Em seguida, os contextos formais sintéticos são gerados por meio da função ‘SCGazAutomator::generateSynteticFormalContexts‘ (linha 18).

O algoritmo utiliza uma barreira OpenMP (‘#pragma omp barrier‘) para sincronizar as operações paralelas antes de prosseguir para a função ‘divideFiles‘, que divide os arquivos de contexto conforme especificado no vetor ‘divisions‘ (linha 20). Após outra barreira de sincronização (linha 22), o algoritmo executa o fechamento formal nos contextos usando a função ‘InCloseAutomator::runInCloseForFormalContexts‘ (linha 23).

Algoritmo 1 Algoritmo para geração e divisão de arquivos de contexto, seguido da execução do InClose

```
Input: int argc, char* argv[]
Result: Geração de contextos formais e execução do InClose
Function main(ARGC, ARGV):
    Data: vector<int> objects{};
    Data: vector<int> attributes{};
    Data: vector<int> divisions{2, 4, 8};
    Data: vector<int> densities{};
    Data: std::vector<int>::iterator it;
    for n ← 100000 to 1000000 by 100000 do
        objects.insert(it, n)
    end
    for n ← 10 to 100 by 10 do
        attributes.insert(it, n)
    end
    for n ← 10 to 70 by 10 do
        densities.insert(it, n)
    end
    vector<ContextFile> contextFiles ← generateContextFiles (objects, attributes,
     densities)
    generateSynteticFormalContexts (contextFiles)
    #pragma omp barrier divideFiles(divisions, contextFiles)
    #pragma omp barrier runInCloseForFormalContexts (contextFiles)
    foreach CONTEXTFILE in CONTEXTFILES do
        foreach DIVISION in DIVISIONS do
            concepts ← generateConcepts (contextFile, division)
            printf("CONCEITOS PARA %S COM %I DIVISÕES: %S",
            contextFileName(), DIVISION, conceptsPrintConcepts())
        end
    end
    return 0
```

Finalmente, para cada arquivo de contexto e cada partição especificada, o algoritmo gera conceitos e imprime os resultados. Isto é realizado através de dois loops aninhados que iteram sobre os arquivos de contexto e as partições, respectivamente (linhas 25-29). Para cada combinação, a função ‘generateConcepts‘ é chamada, e os conceitos gerados são impressos com uma mensagem formatada usando ‘printf‘ (linha 28). O algoritmo termina retornando 0, indicando uma execução bem-sucedida (linha 31).

4.4.1 Inicialização e Geração de Dados

Nesta etapa inicial, os vetores de objetos, atributos, partições e densidades são inicializados com valores específicos. Esses vetores são fundamentais para configurar os parâmetros de entrada necessários para o processo de geração e análise dos contextos formais.

Os objetos e atributos definem o tamanho e a estrutura dos conjuntos de dados a serem analisados, enquanto as partições e densidades são utilizadas posteriormente para configurar o processamento paralelo e a geração de contextos sintéticos com diferentes níveis de complexidade.

Após a inicialização dos vetores, os contextos formais sintéticos são gerados com base nos parâmetros fornecidos, como o número de objetos, atributos e densidades especificados.

Utilizando a ferramenta SCGaz, esses contextos são criados de forma automatizada, garantindo cenários de teste controlados e consistentes. A geração sintética é crucial para a validação e teste dos algoritmos de análise formal de conceitos, proporcionando um ambiente controlado para avaliar o desempenho e a eficácia das técnicas implementadas.

Para facilitar nos testes realizados, foi criado um algoritmo chamado runSCGaz. O algoritmo runSCGaz foi projetado para executar o SCGaz, gerando contextos formais a partir de um arquivo de contexto e repetindo o processo para um número especificado de conceitos. O código-fonte está apresentado e explicado a seguir.

Algoritmo 2 Execução da Ferramenta SCGaz para Geração de Contextos Formais

```
Input: ContextFile contextFile, int numConcepts
Function runSCGaz(contextFile, numConcepts):
    int currentContext = 1
    while currentContext ≤ numConcepts do
        const string path = contextFile.pathName()
        const string filename = contextFile.name(currentContext)
        const string args = "bin/scgaz -c "+ filename
        + -b "+ to_string(contextFile.numObjects) + -
        a "+ to_string(contextFile.numAttributes) + -d "+
        to_string(float(contextFile.density)/100)
        FILE *fpipe
        char line[65536]

        if (fpipe = (FILE*)popen (args.c_str(), "r")) == NULL then
            puts ("Error on execute SCGaz")
        end
        while fgets (line, sizeof line, fpipe) do
            puts (line)
        end
        if feof (fpipe) then
            pclose (fpipe)
        end
        else
            puts ("Error: Failed to read the pipe to the end.")
        end
        currentContext += 1
        renameFile (filename)
        Helper::replaceFile(filename, path,
        Helper::getFormalContextFileExtensionName())

    end
    puts ("FormalContexts generated successfully!")
```

O algoritmo 2 começa inicializando a variável currentContext com o valor 1, servindo como um contador para o número de conceitos a serem processados (linha 2). Este contador é utilizado dentro de um loop while que continua a executar enquanto currentContext for menor ou igual a numConcepts, garantindo que todos os conceitos sejam processados iterativamente (linha 3).

Dentro do loop, o algoritmo constrói os caminhos e nomes de arquivos necessários para a execução da ferramenta scgaz. Ele cria a string path usando o método pathName do objeto contextFile (linha 4) e utiliza o método name do mesmo objeto para criar a string filename, incorporando o número atual do conceito (linha 5). A string args é formada concatenando vários parâmetros necessários para a execução do comando scgaz,

incluindo o nome do arquivo, o número de objetos e atributos, e a densidade do contexto, ajustada para uma representação percentual (linha 6).

Após a construção dos caminhos e argumentos, o algoritmo abre um pipe para executar o comando scgaz usando a função popen (linha 8). Se a abertura do pipe falhar, ele imprime uma mensagem de erro (linha 9). Caso contrário, lê a saída do comando linha por linha, utilizando a função fgets, e imprime cada linha (linhas 10-11). Depois de ler toda a saída, o algoritmo fecha o pipe com a função pclose e verifica se houve algum erro durante a leitura. Se houver um erro, imprime uma mensagem de erro apropriada (linhas 12-14).

Depois de processar a saída do comando scgaz, o algoritmo incrementa o valor de currentContext (linha 17), renomeia o arquivo de saída para um nome apropriado (linha 18) e substitui o arquivo original pelo novo utilizando métodos auxiliares como renameFile e Helper::replaceFile (linha 19). O loop while continua até que todos os conceitos especificados tenham sido processados. Após a conclusão do loop, o algoritmo imprime uma mensagem indicando que todos os contextos formais foram gerados com sucesso, sinalizando o término bem-sucedido do processo (linha 21).

4.4.2 Partição de Contextos Formais e Processamento Paralelo

Após a geração dos contextos sintéticos, os grupos de contextos formais são particionados em grupos menores utilizando técnicas apropriadas de partição. Essa abordagem permite segmentar os conjuntos de dados para facilitar o processamento paralelo, onde cada parte pode ser analisada simultaneamente por diferentes núcleos de processamento. O uso de paralelismo, facilitado pela biblioteca OpenMP, acelera significativamente o tempo de processamento, permitindo uma análise eficiente e escalável dos conjuntos de dados.

O algoritmo 3, proposto para essa finalidade, chamado divideFiles, é responsável por particionar os grupos de contextos formais em várias seções, executar o algoritmo InClose e normalizar os contextos resultantes. Ele utiliza a biblioteca OpenMP para paralelização, permitindo que as operações sejam executadas simultaneamente em múltiplas threads.

Algoritmo 3 Particionamento de Grupos de Contextos Formais e Execução do InClose

```
Input: vector<int> divisions, vector<ContextFile> contextFiles
Function divideFiles(divisions, contextFiles):
    omp_set_num_threads (divisions.size())
     #pragma omp parallel for
     for int division ∈ divisions do
        Splitter::divideFiles(contextFiles, division)
         InCloseAutomator::runInCloseForFormalContexts(contextFiles, 1,
         division)
    end
    #pragma omp barrier
     omp_set_num_threads (contextFiles.size())

    #pragma omp parallel for for auto contextFile ∈ contextFiles do
        Context currentContext = Helper::getContext(
         Helper::getContentForContextFile(contextFile.fullPath()))
         Helper::normalize(contextFile.fullPath(), currentContext)
    end
```

Primeiramente, o algoritmo define o número de threads com base no tamanho do vetor divisions (linha 2). Em seguida, um loop paralelo (#pragma omp parallel for) percorre cada valor em divisions e chama a função Splitter::divideFiles para dividir os arquivos de contexto conforme a divisão especificada (linha 4). Após a divisão, a função InCloseAutomator::runInCloseForFormalContexts é chamada para executar o fechamento formal nos contextos divididos (linha 5).

Depois que todas as divisões foram processadas, o algoritmo utiliza uma barreira OpenMP (#pragma omp barrier) para sincronizar as threads (linha 8). Em seguida, o número de threads é redefinido com base no tamanho do vetor contextFiles (linha 9). Outro loop paralelo percorre cada arquivo de contexto (linha 11), obtendo o contexto atual usando funções auxiliares Helper::getContext e Helper::getContentForContextFile (linha 12). Finalmente, a função Helper::normalize é chamada para normalizar cada contexto (linha 13).

Adentrando no contexto da classe Splitter, também temos a função Splitter::divideFiles, onde fica concentrada toda a regra de divisão de contextos formais por número de objetos. O algoritmo a seguir mostra o funcionamento do mesmo.

O algoritmo 4 executa o processo de dividir arquivos de contexto em vários subcontextos. Ele utiliza a função Helper::getContext para obter o contexto atual a partir do caminho do arquivo completo, calcula o número de objetos por divisão e trata objetos restantes, se houver. Em seguida, itera sobre cada divisão para selecionar os objetos e incidências correspondentes, cria um novo subcontexto e normaliza esse subcontexto usando a função Helper::normalize. Cada subcontexto resultante é armazenado no vetor subContexts para processamento posterior.

Algoritmo 4 Algoritmo para divisão de arquivos de contexto em subcontextos.

```
Function divideFiles(vector<ContextFile> cxtFiles, int numDivs):
    for cxtFile in cxtFiles do
        vector<Context> subContexts
        Context currentContext = Helper::getContext(Helper::getContentForContextFile
        (cxtFile.fullPath()))

        int numObjsForDiv = currentContext.objects.size() / numDivs
        int noRemainingObjs = currentContext.objects.size() % numDivs

        for currDiv from 0 to numDivs do
            int firstIdx = currDiv * numObjsForDiv
            int lastIdx = firstIdx + numObjsForDiv
            if noRemainingObjs ≠ 0 then
                lastIdx += noRemainingObjs
                noRemainingObjs = 0

            end
            Context subContext = currentContext

            vector<string> objects(&currentContext.objects[firstIdx], &currentCon-
            text.objects[lastIdx])
            vector<string> incidences(&currentContext.incidences[firstIdx], &current-
            Context.incidences[lastIdx])

            subContext.objects = objects
            subContext.incidences = incidences

            subContexts.pushback(subContext)

            Helper::normalize(cxtFile.fullPath(1, currDiv, numDivs), subContext)

        end
    end
```

Inicialmente, nas linhas 8 e 9, para cada arquivo de contexto (cxtFile), o algoritmo obtém o conteúdo completo do arquivo e extrai o contexto atual utilizando funções auxiliares. Em seguida, nas linhas 10 e 11, calcula-se o número de objetos por divisão (numObjsForDiv) e o número de objetos restantes (noRemainingObjs) que não se distribuem igualmente nas divisões.

Para cada divisão (currDiv), são calculados os índices inicial (firstIdx) e final (lastIdx) dos objetos que pertencem à divisão corrente, conforme mostrado nas linhas 12 e 13. Caso existam objetos restantes, eles são adicionados à última divisão, ajustando o índice final (lastIdx) e zerando o contador de objetos restantes, conforme descrito nas linhas 14 a 16. O subcontexto é então criado a partir do contexto atual nas linhas 17 a 19, onde são selecionados os objetos e incidências correspondentes aos índices calculados para a divisão atual. Nas linhas 20 e 21, os objetos e incidências do subcontexto são atualizados

com os valores selecionados.

Finalmente, nas linhas 22 e 23, o subcontexto é adicionado ao vetor de subcontextos e, em seguida, normalizado e salvo utilizando uma função auxiliar. Essas funcionalidades chave são essenciais para a divisão eficiente e organizada dos arquivos de contexto em subcontextos menores, facilitando a manipulação e processamento subsequente dos dados.

Para cada divisão (currDiv), são calculados os índices inicial (firstIdx) e final (lastIdx) dos objetos que pertencem à divisão corrente, conforme mostrado nas linhas 12 e 13. Caso existam objetos restantes, eles são adicionados à última divisão, ajustando o índice final (lastIdx) e zerando o contador de objetos restantes, conforme descrito nas linhas 14 a 16. O subcontexto é então criado a partir do contexto atual nas linhas 17 a 19, onde são selecionados os objetos e incidências correspondentes aos índices calculados para a divisão atual. Nas linhas 20 e 21, os objetos e incidências do subcontexto são atualizados com os valores selecionados.

Finalmente, nas linhas 22 e 23, o subcontexto é adicionado ao vetor de subcontextos e, em seguida, normalizado e salvo utilizando uma função auxiliar. Essas funcionalidades chave são essenciais para a divisão eficiente e organizada dos arquivos de contexto em subcontextos menores, facilitando a manipulação e processamento subsequente dos dados.

4.4.3 Execução do Algoritmo In-Close e Geração dos Conceitos Formais

O algoritmo In-Close é aplicado a cada parte dos contextos formais divididos, visando identificar e agrupar os conceitos formais presentes nos dados. Este algoritmo é responsável por analisar as relações entre objetos e atributos dentro de cada contexto, gerando assim os quasiconceitos que representam grupos de elementos com características semelhantes. A execução do In-Close é essencial para a extração de padrões e relações nos conjuntos de dados, fornecendo insights valiosos sobre a estrutura dos dados analisados.

Para cada contexto formal e configuração de divisão especificada, são gerados conceitos utilizando métodos específicos, como a combinação de partições de conceitos ou operações de quasiconceitos. Esses conceitos são então agregados para fornecer uma visão abrangente dos padrões e relações nos dados, permitindo uma análise mais profunda e detalhada da estrutura dos conjuntos de dados analisados. Esta etapa é crucial para identificar insights significativos e validar a eficácia das técnicas de análise formal de conceitos implementadas.

Algoritmo 5 Algoritmo para execução do In-Close em contextos formais.

```
Function runInCloseForFormalContexts(vector<ContextFile> cxtFiles, int numCxt, int numDivs):
    for cxtFile in cxtFiles do
        if numDivs > 1 then
            for currDiv from 0 to numDivs do
                runInClose(cxtFile, numCxt, currDiv, numDivs)

            end
        end
        runInClose(cxtFile, numCxt)

    end
```

O algoritmo 5 descreve o processo de dividir arquivos de contexto em vários subcontextos. Ele utiliza a função Helper::getContext para obter o contexto atual a partir do caminho do arquivo completo, calcula o número de objetos por divisão e trata objetos restantes, se houver. Em seguida, itera sobre cada divisão para selecionar os objetos e incidências correspondentes, cria um novo subcontexto e normaliza esse subcontexto usando a função Helper::normalize. Cada subcontexto resultante é armazenado no vetor subContexts para processamento posterior.

A função *runInClose* implementada no algoritmo apresentado é um encapsulamento do algoritmo In-Close 4. Essa função será detalhadamente abordada em uma seção subsequente deste trabalho, onde serão discutidos os detalhes da sua implementação.

4.4.4 Algoritmo para a obtenção de conceitos formais a partir dos quasi-conceitos

Para a obtenção dos conceitos formais, foi criado o algoritmo a seguir, o `return ConceptsByQuasiConcepts`. Ele processa todos os grupos de quasiconceitos resultantes, sempre realizando as operações entre dois grupos. Esse processamento é feito pelo algoritmo `returnConceptsByQuasiConceptsTwoByTwo`.

Algoritmo 6 Algoritmo para obtenção de conceitos formais a partir de quasiconceitos

```
Function returnConceptsByQuasiConcepts(vector<ConceptGroup> partitions):
    vector<ConceptGroup> newPartitions = partitions
    timer.startTimer()

    if newPartitions.empty() then
        puts("Invalid data")

    end
    while newPartitions.size() > 1 do
        newPartitions[0] = (newPartitions[0], newPartitions[1])
        newPartitions.pop_back()

    end
    timer.endTimer()

    return newPartitions[0]
```

O algoritmo 6 é implementado na função returnConceptsByQuasiConcepts, que recebe como entrada um vetor de partitions (linha 2). Inicialmente, o vetor newPartitions é configurado para ser uma cópia de partitions (linha 3) e o timer é iniciado para medir o tempo de execução do processo (linha 4).

Em seguida, o algoritmo verifica se newPartitions está vazio. Caso esteja, uma mensagem de erro "Invalid data"é impressa (linhas 5-7). Essa verificação é crucial para garantir que o algoritmo não prossiga com dados inválidos.

O núcleo do algoritmo é um laço while que continua executando enquanto o tamanho de newPartitions for maior que 1 (linha 9). Dentro do laço, a função returnConceptsByQuasiConceptsTwoByTwo é chamada para combinar as duas primeiras partições de newPartitions (linha 10). Após a combinação, a última partição do vetor é removida (newPartitions.pop_back(), linha 11). Esse processo de combinação e redução continua até que reste apenas uma partição.

Após a conclusão do laço, o timer é parado (linha 13), e a partição resultante, que agora contém os conceitos combinados, é retornada (linha 14).

Foi criado também o algoritmo 7, denominado `returnConceptsByQuasiConceptsTwoByTwo`. Ele realiza as operações de obtenção de conceitos a partir de quasiconceitos obtidos de dois grupos de quasiconceitos (`groupConceptsA` e `groupConceptsB`). Este algoritmo é essencial para combinar e derivar novos conceitos baseados nas interseções e uniões de atributos e objetos entre os grupos de conceitos fornecidos.

Algoritmo 7 Algoritmo para a obtenção de conceitos formais a partir dos quasiconceitos

```
Function returnConceptsByQuasiConceptsTwoByTwo(ConceptGroup groupConceptsA, ConceptGroup groupConceptsB):
    // Instancia arrays vazios para armazenar os conceitos gerados a
       partir dos quasiconceitos
    set<Concept> newConceptsArray{};
     set<Concept> auxArray{};

    // Retorna todos os atributos através da união dos atributos únicos de
       todos os ConceptGroups
    set<string> allAttributes = groupConceptsA.uniqueAttributes();
     allAttributes.merge(groupConceptsB.uniqueAttributes());

    // Retorna todos os objetos através da união dos objetos únicos de
       todos os ConceptGroups
    set<string> allObjects = groupConceptsA.uniqueObjects();
     allObjects.merge(groupConceptsB.uniqueObjects());

    // Realiza as operações listadas para a obtenção dos conceitos a
       partir dos quasiconceitos
    ...
    // Obtêm-se o Ínfimo e o Supremum
    ...
    return conceptGroup;
```

O algoritmo returnConceptsByQuasiConceptsTwoByTwo visa a obtenção de conceitos formais a partir de quasiconceitos, instanciando inicialmente arrays vazios para armazenar os conceitos gerados. Essa inicialização ocorre nas linhas 7 e 8, onde são criados os conjuntos newConceptsArray e auxArray. Em seguida, o algoritmo retorna todos os atributos através da união dos atributos únicos de todos os ConceptGroups, realizada nas linhas 11 e 12. Essa união é essencial para garantir que todos os atributos relevantes sejam considerados.

De forma semelhante, o algoritmo retorna todos os objetos através da união dos objetos únicos de todos os ConceptGroups, conforme implementado nas linhas 15 e 16. Essas operações garantem que a base de dados inclua todos os objetos possíveis antes de proceder com a geração dos conceitos formais. Após essas etapas de preparação, o algoritmo realiza diversas operações (não especificadas no trecho fornecido) para obter os conceitos a partir dos quasiconceitos.

Finalmente, o algoritmo calcula o ínfimo e o supremum dos conceitos gerados, assegurando a completude e correção da estrutura conceitual. O resultado final é retornado como um conceptGroup, encapsulando todos os conceitos formais obtidos a partir dos

quasiconceitos processados. As operações específicas de cálculo do ínfimo e do supremum, assim como outras operações intermediárias, são indicadas no algoritmo detalhadas no apêndice B.

4.4.5 Armazenamento de Resultados

Por fim, os resultados da análise são armazenados, incluindo tempos de execução e outras métricas relevantes, em arquivos ou outros formatos de saída. Esses dados são fundamentais para avaliar o desempenho do sistema, comparar diferentes configurações de análise e validar os padrões identificados nos conjuntos de dados sintéticos. O armazenamento dos resultados permite uma análise posterior detalhada e auxilia na tomada de decisões para otimizar e melhorar os processos de análise formal de conceitos. O algoritmo 8 ilustra o código apresentado.

Algoritmo 8 Salvar o tempo de execução do ConceptsFinder

Function `saveCFTime(`STRING TIME, STRING CXTNAME`):`

```
// Abrir o arquivo em modo de escrita (append)
std::ofstream file;
 file.open("ConceptsFinder.csv", ios::out | ios::app);

// Escrever o nome do contexto e o tempo no arquivo
file « cxtName + ":"+ time + "";

// Fechar o arquivo
file.close();
```

4.5 Encapsulamento do algoritmo In-Close 4

Neste projeto, diante da necessidade de explorar e hierarquizar relações complexas entre conjuntos de objetos e atributos em grandes volumes de dados, foi crucial a implementação de uma classe para encapsular o algoritmo escolhido para extração de conceitos formais.

O algoritmo selecionado foi o In-Close 4, uma versão avançada do algoritmo In-Close, projetada para lidar com conjuntos de dados densos e de alta dimensionalidade.

A classe desenvolvida facilita a aplicação automatizada do encapsulador do In-Close 4 em múltiplos contextos formais, configurando e coordenando a execução do algoritmo, além de gerenciar os resultados obtidos. Essa abordagem não apenas simplifica a análise e interpretação hierárquica das relações entre os dados, mas também melhora significativamente a eficiência e escalabilidade das operações de mineração de dados.

O algoritmo encapsulado do In-Close implementado é fundamental na análise de conceitos formais, uma técnica da Análise Formal de Conceitos (FCA) utilizada para

explorar e hierarquizar relações entre conjuntos de objetos e atributos.

A função ‘runInClose‘ coordena a execução do algoritmo para múltiplos contextos formais, onde cada contexto é processado para identificar conceitos formais. Ela opera recebendo arquivos de contexto como entrada, configurando os parâmetros necessários para a execução do encapsulador do In-Close e capturando os resultados de tempo de execução.

Ao final de cada iteração, os resultados são organizados e arquivos temporários são gerenciados para manter a integridade e a organização dos conceitos identificados. Essa abordagem automatizada permite a aplicação eficiente da ACF em conjuntos de dados complexos, facilitando a análise e a interpretação hierárquica das relações entre objetos e atributos.

O algoritmo 9 implementa o encapsulamento do algoritmo In-Close para a geração de conceitos formais a partir de múltiplos contextos formais.

Primeiramente, para cada contexto no intervalo definido por numCxt, o algoritmo configura os parâmetros necessários para a execução do In-Close 4 (linhas 5-9). Isso inclui a definição do caminho do arquivo de contexto (contextFile.pathName(), linha 6), o nome do arquivo (contextFile.name(currCxt, currDiv, numDivs), linha 7), e os argumentos para a chamada do comando In-Close (contextFile.fullPath(numCxt, currDiv, numDivs), linha 8). Essa preparação garante que o comando será executado com os parâmetros corretos para cada iteração.

A seguir, o algoritmo realiza a execução do comando In-Close (linhas 11-23). Para isso, ele abre um pipe para executar o comando In-Close com os argumentos configurados. Ele verifica se a execução foi bem-sucedida (linha 13) e, em seguida, lê a saída do comando linha por linha (linha 15). Se a linha lida contiver a string "Total time", o tempo de execução é salvo (saveICTime(getIntFromString(lineString), filename), linha 17). Todas as linhas são impressas no console (linha 19), permitindo monitorar a execução do comando.

Após a leitura completa da saída do comando, o pipe é fechado (linhas 20-24). Caso haja um erro ao fechar o pipe, uma mensagem de erro é exibida (linha 23). Esta etapa é crucial para garantir que todos os recursos sejam liberados corretamente e que a execução do comando seja finalizada de maneira apropriada.

Em seguida, o algoritmo realiza a substituição do arquivo de contexto pelo novo arquivo gerado e a limpeza dos arquivos não utilizados (linhas 26-27). Esta etapa envolve a chamada das funções replaceFile(path, filename) (linha 26) e removeUnusedFiles() (linha 27). Essas funções garantem que os arquivos antigos sejam substituídos pelos novos e que quaisquer arquivos temporários ou desnecessários sejam removidos, mantendo a organização do sistema de arquivos.

Finalmente, o algoritmo imprime uma mensagem indicando o sucesso na geração dos conceitos (linha 28). Esta mensagem finaliza a execução do algoritmo, fornecendo um *feedback* ao usuário sobre o sucesso da operação. Em suma, o algoritmo de encapsulamento do In-Close 4 é um processo robusto e eficiente para a execução automatizada do algoritmo In-Close em múltiplos contextos, garantindo a correta configuração dos parâmetros, execução, leitura da saída, e gerenciamento dos arquivos de entrada e saída.

Algoritmo 9 Algoritmo para o encapsulamento do algoritmo In-Close 4.

```
Function runInClose(ContextFile contextFile, int numCxt, int currDiv,
int numDivs):
    for currCxt from 0 to numCxt do
        // Configuração dos parâmetros
        string path = contextFile.pathName()
         string filename = contextFile.name(currCxt, currDiv, numDivs)
         string args = "bin/inclose "+ contextFile.fullPath(numCxt, currDiv, numDivs)
         + "0 0 y 2 n"

        // Execução do comando InClose
        FILE *fpipe
         char *line
         if (fpipe = (FILE*)popen(args.c_str(),"r")) == NULL then
            printf("Error on execute InClose")
        end
        while fgets(line, sizeof line, fpipe) do
            string lineString = string(line)
             if lineString.find("Total time") != std::string::npos then
                saveICTime(getIntFromString(lineString), filename)
            end
            printf("%s", lineString.c_str())
             end
            if feof(fpipe) then
                pclose(fpipe)
            end
            else
                printf( "Error: Failed to read the pipe to the end.")
            end
            // Substituição do arquivo e limpeza
            replaceFile(path, filename)
             removeUnusedFiles()
        end
        printf("generated successfully!")
```

4.6 Classe Helper

Neste projeto, também se utilizou diversos algoritmos auxiliares para apoiar a execução de tarefas pequenas, como a normalização dos objetos e/ou atributos, ou até a manipulação de textos para a exibição. Para apoio, o código-fonte deste algoritmo está disponível no apêndice C.

5 RESULTADOS E DISCUSSÕES

Esta seção define as métricas utilizadas para avaliar o êxito deste trabalho, apresenta e discute os resultados obtidos a partir da execução do algoritmo In-Close 4, tanto em sua versão sequencial quanto na versão paralela implementada em C++/OpenMP.

Os dados foram normalizados para facilitar a execução do algoritmo, e uma série de experimentos foi conduzida com diferentes tamanhos de bancos de dados, variando de 10.000 a 1.000.000 de objetos, com 50 atributos e uma densidade de 50%. A densidade foi escolhida devido à melhor compatibilidade na geração dos contextos sintéticos com os diferentes números de objetos apresentados.

As tabelas e figuras apresentadas ao longo da seção ilustram as melhorias obtidas com a abordagem paralela, destacando as tendências de escalabilidade e eficiência à medida que o número de *threads* e núcleos é aumentado.

5.1 Métricas

As métricas principais incluem o número de conceitos criados, o tempo de execução e o impacto do número de threads na eficiência do processamento para cada contexto formal.

5.1.1 Número de Conceitos Criados

O número de conceitos criados é uma métrica crucial para avaliar a capacidade do algoritmo em identificar relações e padrões nos conjuntos de dados analisados. Quanto maior o número de conceitos formais gerados, maior é a capacidade de representar diferentes níveis de abstração e complexidade nas relações entre objetos e atributos. Também é importante ressaltar que o número de conceitos criados foi utilizado também para assegurar a assertividade do método.

5.1.2 Tempo de Execução

O tempo de execução refere-se ao período necessário para processar e analisar os dados em cada contexto formal. É uma métrica crítica que influencia diretamente a eficiência do algoritmo, especialmente quando lidando com grandes volumes de dados. Reduzir o tempo de execução permite uma análise mais rápida e responsiva, facilitando a tomada de decisões e a exploração de dados em tempo real.

5.1.3 Número de Threads Utilizados

O número de threads utilizados durante o processamento paralelo também é uma métrica importante. A paralelização do algoritmo permite distribuir o trabalho entre múltiplos núcleos de processamento, acelerando o tempo de execução geral. No entanto, o aumento no número de threads nem sempre resulta em uma melhoria linear no desempenho devido a sobrecargas e custos de sincronização. Portanto, encontrar o equilíbrio ideal entre o número de threads e o tamanho do problema é crucial para maximizar a eficiência computacional.

5.1.4 Comparação e Análise

Para cada contexto formal analisado, são coletadas métricas detalhadas de desempenho, incluindo o número de conceitos criados, o tempo total de execução e o impacto da variação no número de threads. Essas métricas são então comparadas e analisadas para entender melhor como o algoritmo se comporta em diferentes configurações e tamanhos de conjunto de dados. A comparação sistemática desses resultados ajuda a otimizar o algoritmo, identificar gargalos de desempenho e orientar ajustes para melhorar ainda mais a eficiência da análise de conceitos formais.

5.2 Preparação dos Dados

Um processo de normalização foi realizado antes da execução do algoritmo In-Close 4 para facilitar na legibilidade. Esta normalização visou a substituição dos termos "object" e "attribute" pelas letras do alfabeto ordenadas, convertendo o valor decimal correspondente para a letra correspondente do alfabeto, ou números correspondentes, devida estrutura retornada pelo algoritmo de geração de contextos sintéticos. Caso o número dos atributos fosse igual ou superior a 26, era adicionada mais uma letra ao lado para corresponder ao índice e assim sucessivamente. Ex: attribute[26] $\rightarrow AA$. Para objetos foi atribuído um conjunto de números, e para atributos, um conjunto de letras. Por exemplo: object[0] $\rightarrow$ 1 e attribute[5] $\rightarrow F$.

5.3 Configuração dos Experimentos

Para realizar os testes, geramos 100 contextos sintéticos usando *SCGaz*(RIMSA; SONG; ZÁRATE, 2013), variando o Contexto Formal com 10.000, 25.000, 50.000, 100.000 e 1.000.000 objetos, 50 atributos e densidade de 50%. Esses contextos formais foram então divididos em 2, 4, 8, 12, 16, 18 e 24 subgrupos para garantir a geração completa de conceitos formais. Além disso, cinco contextos formais distintos foram gerados para cada conjunto de dados para garantir uma cobertura abrangente dos testes.

5.4 Ambiente de Execução

Nossos experimentos foram realizados em uma máquina baseada em um Intel Xeon E5645 com 2,40 GHz, 12 núcleos. 24 *threads* SMT e *caches* L2 privados, utilizando 64 GB de RAM e 1 TB de armazenamento em disco. Ao longo de todos os testes, os conceitos formais derivados dos quase-conceitos, segregados em subconjuntos, mostraram-se equivalentes ao conjunto original de conceitos formais. A avaliação de desempenho foi baseada nos tempos de execução da versão serial do In-Close 4 e em nossa implementação paralela em C++/OpenMP.

5.5 Resultados Obtidos

A seção a seguir apresenta uma análise detalhada do desempenho do algoritmo proposto sob diferentes configurações de paralelismo e volumes de dados. Por meio de experimentos realizados com variações no número de objetos, atributos e *threads*, foram avaliados parâmetros como tempo de execução, ganho de desempenho (speedup). Esses resultados permitiram compreender como o algoritmo se comporta em cenários variados, destacando tanto os benefícios do paralelismo quanto os desafios inerentes ao escalonamento eficiente.

Para cada uma das configurações apresentadas (objetos, atributos, densidade e número de *threads*), foram gerados cem contextos formais os quais foram executados em um tempo determinado. O valor de tempo apresentado em cada um destes cenários é resultado de uma média calculada a partir das cem execuções. Para verificar que não houve uma discrepância entre os valores apresentados, foram calculados os valores de desvio padrão e variância para cada um dos conjuntos de dados, disponíveis nas tabelas abaixo.

5.5.1 Contexto Formal com 10.000 Objetos, 50 Atributos e 50% de Densidade

Na Tabela 7, ao considerar um Contexto Formal de 10.000 objetos e 50 atributos, a execução sequencial utilizando o algoritmo In-Close 4 levou 0,4781 segundos para gerar 282.115 conceitos.

Ao adotar abordagens paralelas com 2, 4, 8, 12, 16, 18 e 24 *threads* (teste de escalabilidade forte), foi observada uma melhoria nos tempos de execução, registrando 0,3356, 0,3374, 0,3590, 0,3359, 0,3214, 0,3124 e 0,3065 segundos, respectivamente. Esta ligeira redução no tempo de execução sugere melhorias potenciais com base na escalabilidade fraca, ou seja, conjuntos de dados maiores.

De acordo com os resultados apresentados na Figura 6, o tempo de execução diminui significativamente à medida que aumenta o número de *threads* e núcleos. Inicialmente, o tempo total de execução diminui consideravelmente à medida que o paralelismo é introduzido, destacando-se um ganho de até 1,560x com 24 *threads* e 12 núcleos em comparação à execução sequencial.

Tabela 7 – Resultados para contextos com 10.000 objetos e 50 atributos

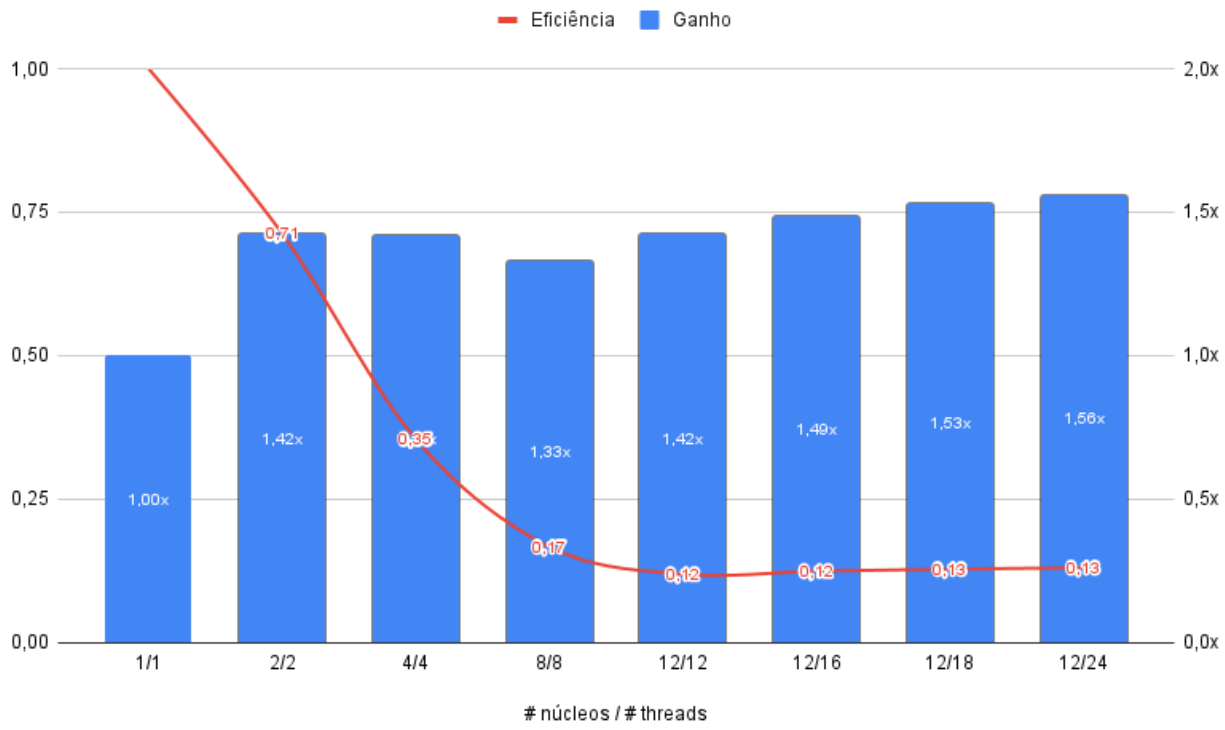
# Conceitos		Tempo (s)		# *threads*	Tempo (s)	Desvio Padrão	Variância
282.115	In-Close 4	0,4781	Abordagem paralela	2	0,3356	0,0212	0,00045
				4	0,3374	0,0163	0,00027
				8	0,3590	0,0166	0,00028
				12	0,3359	0,0145	0,00021
				16	0,3214	0,0153	0,00023
				18	0,3124	0,0141	0,00020
				24	0,3065	0,0137	0,00019

Fonte: Elaborada pelo autor.

O pico de ganho foi de 1,425x, e a abordagem paralela foi 28,8% menos eficiente que o tempo de execução serial quando 2 *threads* e 2 núcleos foram usados.

No entanto, a eficiência caiu à medida que mais *threads* foram empregados, indicando que a escalabilidade pode ser afetada por fatores como sobrecarga de comunicação ou sincronização. Notavelmente, apesar do ganho de desempenho, a eficiência diminui com mais *threads*, sugerindo a necessidade de uma carga de trabalho ou conjunto de dados maior.

Figura 6 – Ganho e eficiência para 10.000 objetos, 50 atributos, e 50% de densidade



Fonte: Elaborada pelo autor.

5.5.2 Contexto Formal com 25.000 Objetos, 50 Atributos e 50% de Densidade

Com 25.000 objetos e 50 atributos, conforme mostrado na Tabela 8, a execução do algoritmo In-Close 4 sem paralelismo exigiu um tempo de 1,1953 segundos para processar 298.512 conceitos.

O paralelismo com diferentes números de *threads* (2, 4, 8, 12, 16, 18 e 24) reduziu claramente o tempo de execução, diminuindo para 0,8390, 0,8435, 0,8975, 0,8398, 0,8035, 0,7810 e 0,7663 segundos, respectivamente. Isto destaca a capacidade do método paralelo em otimizar o desempenho do algoritmo, proporcionando uma melhoria significativa no processamento formal do conceito.

Tabela 8 – Resultados para contextos com 25.000 objetos e 50 atributos

# Conceitos		Tempo (s)		# *threads*	Tempo (s)	Desvio Padrão	Variância
298.512	In-Close 4	1,1953	Abordagem paralela	2	0,8390	0,0456	0,00208
				4	0,8435	0,0398	0,00159
				8	0,8975	0,0402	0,00161
				12	0,8398	0,0423	0,00179
				16	0,8035	0,0395	0,00156
				18	0,7810	0,0376	0,00142
				24	0,7663	0,0364	0,00132

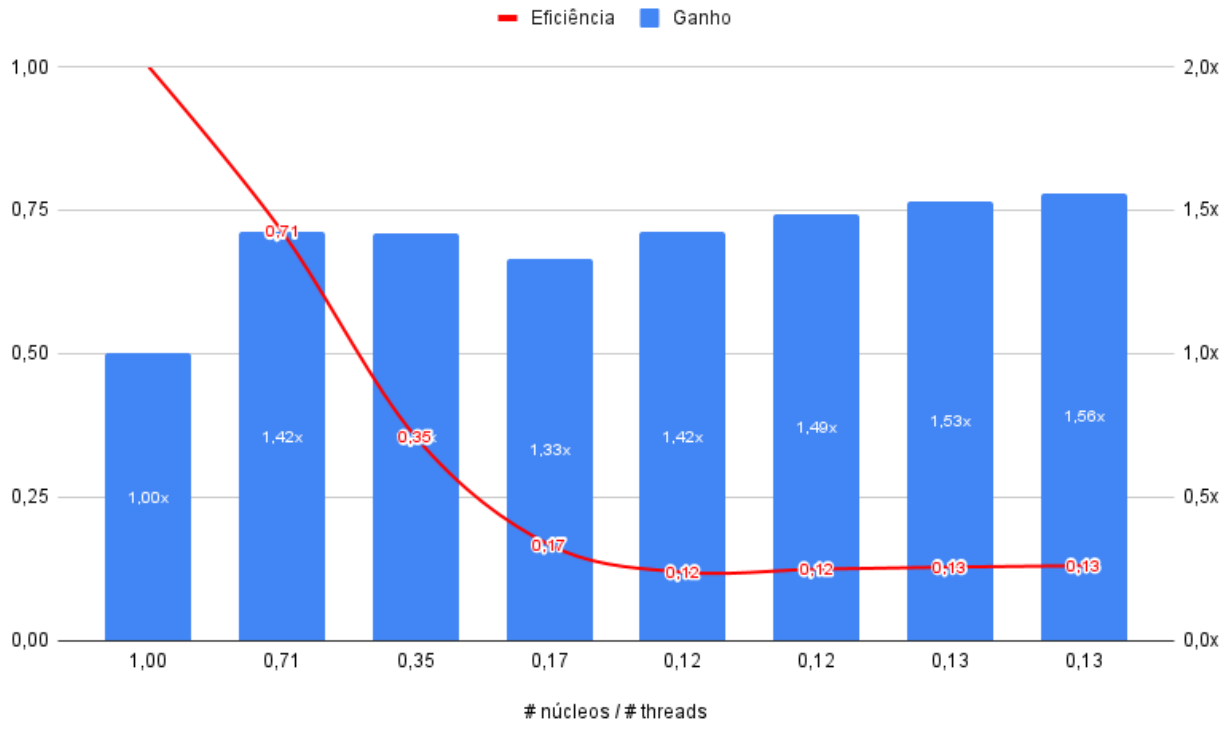
Fonte: Elaborada pelo autor.

Conforme visto na Figura 7, a execução do algoritmo In-Close 4 sem o uso de paralelismo exigiu um tempo de 0,4796 segundos para processar 198.423 conceitos. Com o uso de diferentes números de *threads* (2, 4, 8 e 12), foi possível observar uma redução no tempo de execução, que diminuiu para 0,3356, 0,3374, 0,3590 e 0,3359 segundos, respectivamente. No entanto, os ganhos obtidos foram modestos, com o pico de ganho atingindo apenas 1,56x quando foram utilizadas 12 *threads* e 24 núcleos.

A eficiência apresentou uma queda significativa ao longo do aumento do número de *threads*, com valores variando de 0,71 (com 2 *threads*) até 0,13 (com 24 *threads*). Este comportamento evidencia a presença de overheads consideráveis na paralelização, como contenção de recursos e custos de sincronização entre as *threads*, que comprometeram o desempenho e limitaram a escalabilidade do algoritmo.

Embora a abordagem paralela tenha oferecido algum ganho de desempenho, a baixa eficiência destaca a necessidade de melhorias no balanceamento de carga e na gestão de recursos. Além disso, é possível que o algoritmo tenha características que dificultam o aproveitamento do paralelismo em escalas menores. Trabalhos futuros poderão investigar estratégias de otimização para mitigar esses problemas, como a reformulação da divisão de tarefas ou o uso de técnicas avançadas de paralelização que maximizem o aproveitamento dos recursos computacionais disponíveis.

Figura 7 – Ganho e eficiência para 25.000 objetos, 50 atributos, e 50% de densidade



Fonte: Elaborada pelo autor.

5.5.3 Contexto Formal com 50.000 Objetos, 50 Atributos e 50% de Densidade

Com um conjunto de dados composto por 50.000 objetos e 50 atributos, conforme mostrado na Tabela 9, os resultados detalhados comparam a execução do algoritmo In-Close 4 com uma abordagem sequencial e paralela. A implementação paralela apresentou uma melhoria significativa no tempo de execução em comparação com a versão sequencial, por exemplo, de 20235,84 segundos para 881,47 segundos com 24 *threads*.

Utilizando o algoritmo com 428.215 conceitos, a execução sequencial demorou cerca de 20235,84 segundos, enquanto a abordagem paralela, empregando diferentes números de *threads* - 2, 4, 8, 12, 16, 18 e 24 - reduziu esse tempo para valores entre 5058,96 e 881,47 segundos. Isso destaca a eficiência do paradigma do paralelismo na redução do tempo computacional, mostrando ganhos substanciais no desempenho do algoritmo à medida que aumenta o número de *threads* utilizadas.

Tabela 9 – Resultados para contextos com 50.000 objetos e 50 atributos

# Conceitos		Tempo (s)		# *threads*	Tempo (s)	Desvio Padrão	Variância
428.215	In-Close 4	20.235,84	Abordagem paralela	2	5.058,96	0,7438	0,553
				4	2.792,27	0,3875	0,15
				8	1.927,42	0,2762	0,076
				12	1.497,43	0,225	0,051
				16	1.218,54	0,2324	0,054
				18	1.016,85	0,1912	0,037
				24	881,47	0,1349	0,018

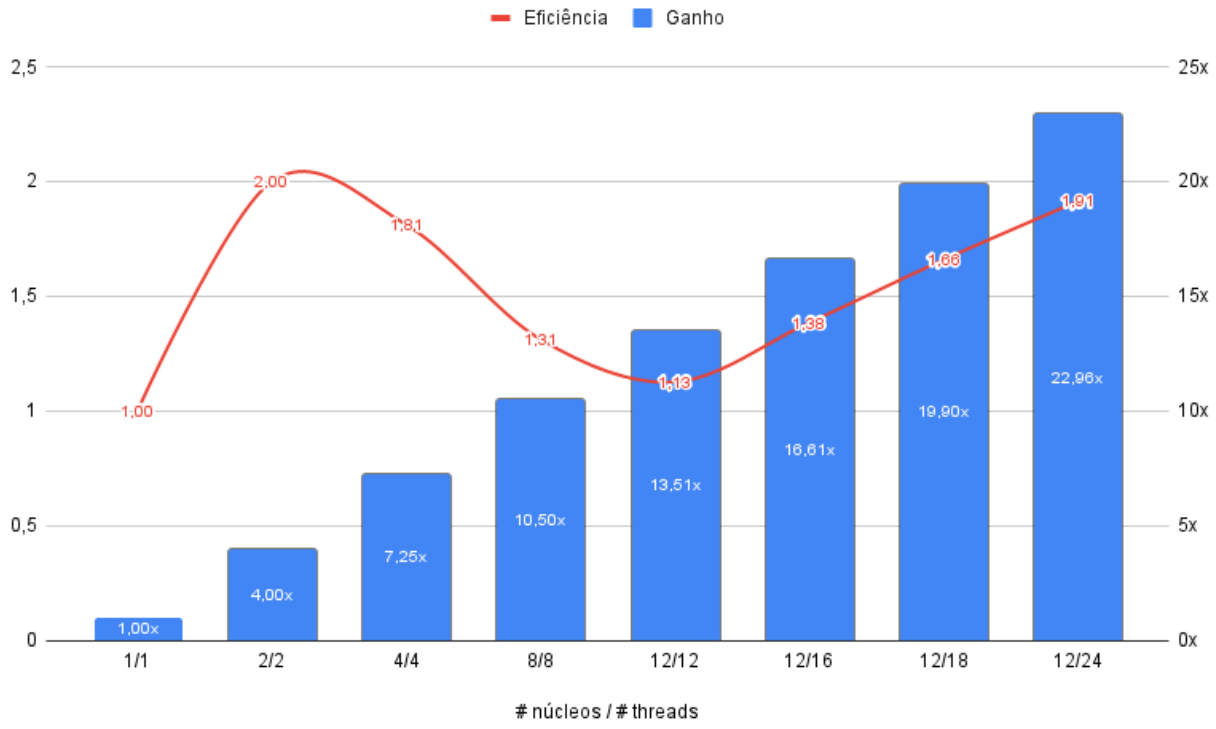
Fonte: Elaborada pelo autor.

Conforme visto na Figura 8, a execução do algoritmo In-Close 4 sem uso de paralelismo exigiu um tempo de 20235,84 segundos para processar 428.215 conceitos. Porém, ao adotar o paralelismo com diferentes números de *threads* (2, 4 e 8), foi observada uma clara redução no tempo de execução, diminuindo para 5058,96, 2792,27 e 1927,42 segundos, respectivamente. Isto destaca a capacidade do método paralelo em otimizar o desempenho do algoritmo, proporcionando uma melhoria significativa na eficiência do processamento dos conceitos formais em questão.

O pico de ganho foi de 22,96x, e a abordagem paralela foi 91% mais eficiente que o tempo de execução serial quando foram utilizados 12 *threads* e 24 núcleos.

É importante ressaltar que nesses cenários onde a eficiência atingiu valores superiores a 1 (acima do ideal, de acordo com a Lei de Amdahl) foi proveniente de um melhor aproveitamento de recursos, podendo ter sido ocasionada, e.g., devido à uma maior quantidade de *caches* disponíveis juntamente com o compartilhamento de dados entre as mesmas. Essa investigação será realizada em trabalhos futuros.

Figura 8 – Ganho e eficiência para 50.000 objetos, 50 atributos, e 50% de densidade



Fonte: Elaborada pelo autor.

5.5.4 Contexto Formal com 100.000 Objetos, 50 Atributos e 50% de Densidade

A Tabela 10 apresenta resultados relevantes para o desempenho de um algoritmo que emprega paralelismo na geração de conceitos formais, considerando um conjunto de 100.000 objetos e 50 atributos. Com uma quantidade significativa de 582.219 conceitos formados pelo algoritmo In-Close 4, o tempo de execução sem abordagem paralela foi de 33.996,2112 segundos.

Com o paradigma paralelo utilizando 2 *threads*, houve uma redução notável no tempo, chegando a 19.512,26035 segundos. A adição de mais *threads* mostrou melhorias incrementais, com 4 *threads* atingindo 11.129,51326 segundos e 8 *threads* atingindo 5.754,4621 segundos.

O pico de ganho para este contexto formal ocorreu quando o algoritmo utilizou 24 *threads*/12 núcleos, com tempo de execução de 2.561,9234 segundos. Esses resultados mostram uma tendência de melhoria de desempenho com a introdução do paralelismo. No entanto, a Figura 8 mostra que a eficiência diminui com o aumento de *threads*, o que é típico de uma forte escalabilidade. Esse comportamento sugere um melhor equilíbrio entre a carga de trabalho e o número de núcleos e *threads*.

Tabela 10 – Resultados para contextos com 100.000 objetos e 50 atributos

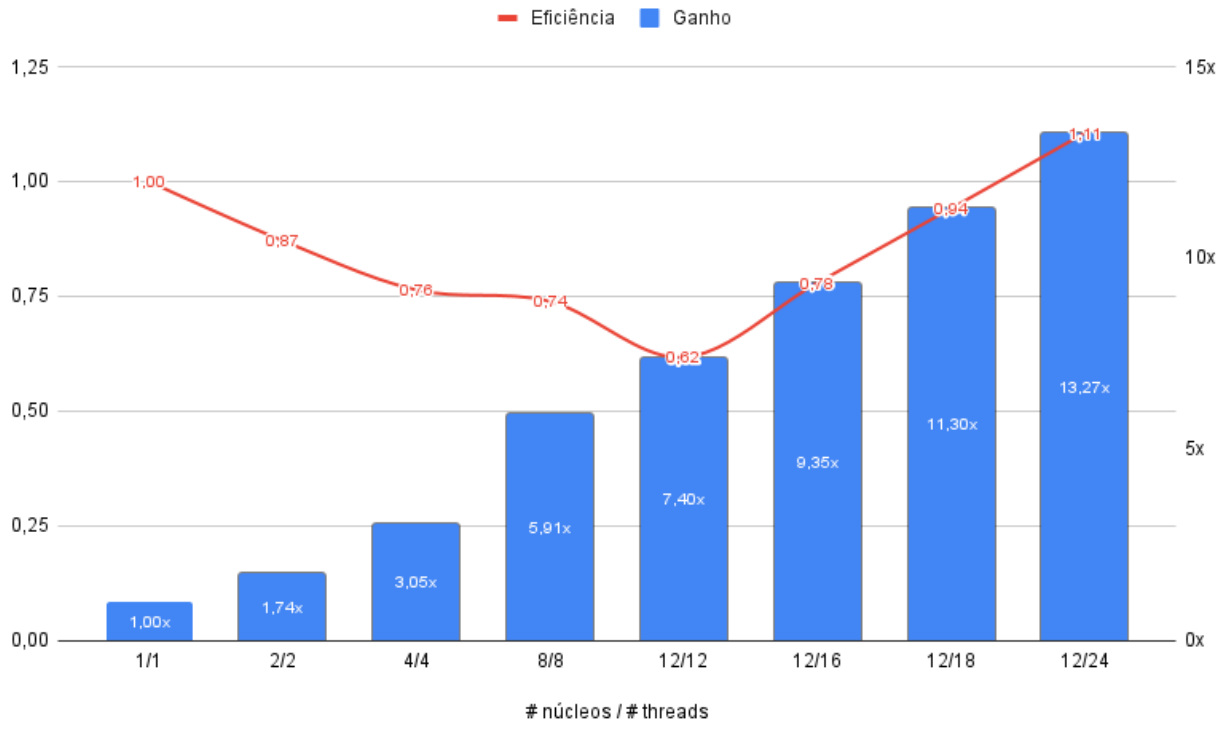
# Conceitos		Tempo (s)		# *threads*	Tempo (s)	Desvio Padrão	Variância
582.219	In-Close 4	33.996,2112	Abordagem paralela	2	19.512,26035	1,2314	1,514
				4	11.129,51326	0,8342	0,696
				8	5.754,462101	0,5121	0,262
				12	4.593,337729	0,3498	0,122
				16	3.636,386227	0,2719	0,074
				18	3.009,63289	0,241	0,058
				24	2.561,923405	0,2145	0,046

Fonte: Elaborada pelo autor.

A Figura 9 apresenta os resultados da execução de um algoritmo que emprega paralelismo para gerar conceitos formais, variando o número de *threads* e núcleos utilizados. O tempo total de execução diminui significativamente à medida que aumenta o número de *threads*, demonstrando a eficiência do paralelismo no algoritmo proposto. Observa-se que, com o aumento de *threads* até 12 ocorre um declínio de eficência, enquanto há um aumento no ganho, porém quando executadas com mais de uma *thread* por núcleo (a partir de 12/18), temos uma eficiência crescente.

O pico de ganho foi de 13,27x e a abordagem paralela foi 11% mais eficiente que o tempo de execução serial, quando foram utilizados 24 *threads* e 12 núcleos.

Figura 9 – Ganho e eficiência para 100.000 objetos, 50 atributos, e 50% de densidade



Fonte: Elaborada pelo autor.

5.5.5 Contexto Formal com 1.000.000 Objetos, 50 Atributos e 50% de Densidade

A Tabela 11 compara duas abordagens para processar um conjunto de dados com 1.000.000 de objetos e 50 atributos. A abordagem serial requer 109.273,536 segundos de execução.

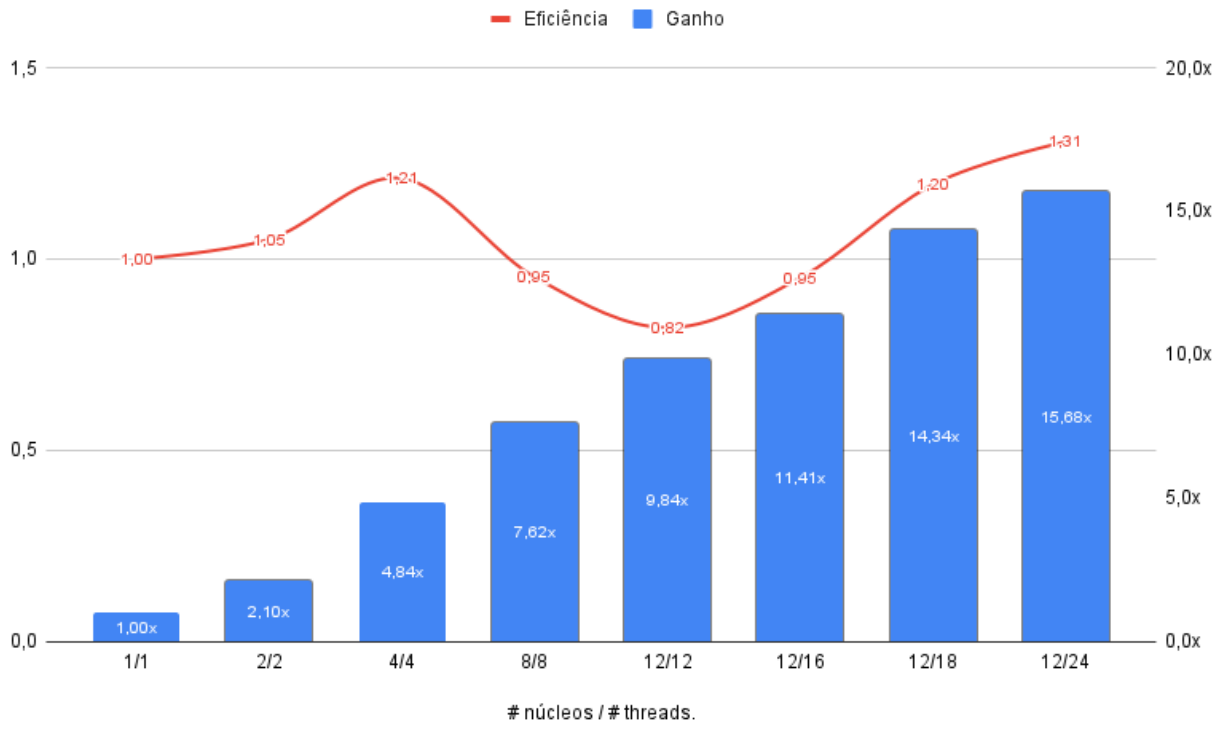
Observamos uma redução considerável no tempo de processamento à medida que o número de *threads* aumenta. Com 24 *threads*, por exemplo, a abordagem paralela atinge um tempo de 6.968,97551 segundos, o que representa um ganho de aproximadamente 15x em relação à abordagem serial.

Tabela 11 – Resultados para contextos com 1.000.000 objetos e 50 atributos

# Conceitos		Tempo (s)		# *threads*	Tempo (s)	Desvio Padrão	Variância
2.328.876	In-Close 4	109.273,536	Abordagem paralela	2	51.960,79	4,6728	21,85
				4	22.563,19	3,5823	12,825
				8	14.338,47737	2,4159	5,835
				12	11.102,77748	1,5684	2,459
				16	9.580,355602	1,1524	1,327
				18	7.620,190795	1,0312	1,064
				24	6.968,97551	0,9372	0,878

Fonte: Elaborada pelo autor.

Figura 10 – Ganho e eficiência para 1.000.000 objetos, 50 atributos, e 50% de densidade



Fonte: Elaborada pelo autor.

Os resultados da Figura 10 também mostram uma redução significativa no tempo total de execução à medida que aumenta o número de *threads* e núcleos disponíveis.

Notavelmente, de uma configuração de um *thread* e um núcleo para quatro *threads* e núcleos, houve uma redução considerável no tempo total de execução de 8.094.336 para 1.671.347, resultando em um ganho de 4,843x.

No entanto, à medida que o número de *threads* aumenta além do número de núcleos

disponíveis (casos de 16, 18 e 24 *threads* com 12 núcleos), é observado um enorme aumento no ganho em relação à execução sequencial.

A eficiência do algoritmo, representada pela razão entre o ganho e o número de *threads*, atinge seu pico em 24 *threads* com valor de 1,307x, indicando um uso mais eficiente dos recursos disponíveis naquele ponto. Porém, quando o número de *threads* está entre 4 e 18, embora o ganho aumente, a eficiência diminui em comparação aos resultados com outro número de *threads*.

O pico de ganho foi de 15,680x e a abordagem paralela foi 30,7% mais eficiente que o tempo de execução serial, quando foram utilizados 12 *threads* e 24 núcleos.

É importante considerar que a queda de eficiência é natural em programas paralelos quando há aumento no número de núcleos (denominador na fórmula de eficiência). No entanto, como observado, o aumento no número de *threads* com quantidade fixa de núcleos (limite de 12) possibilitou aumentar o ganho (numerador da fórmula de eficiência). Sendo assim, tem-se um aumento na eficiência, porque a quantidade de núcleos utilizados não varia com o aumento do ganho.

5.6 Considerações sobre os resultados obtidos

Os resultados obtidos mostram a capacidade de explorar escalabilidades fracas e fortes na execução paralela. A escalabilidade fraca é evidenciada pela capacidade de aumentar a carga de trabalho proporcionalmente ao adicionar mais núcleos, enquanto a escalabilidade forte é observada quando a eficiência do sistema é mantida com a constante quantidade de dados, mesmo com o aumento do número de núcleos de processamento.

Foram observadas acelerações superlineares em certas configurações, conforme ilustrado na Figura 9. Por exemplo, foram alcançados ganhos de 14x e 15x ao executar a versão paralela do algoritmo utilizando doze núcleos. Esses ganhos indicam que a eficiência do sistema pode superar a soma esperada das capacidades dos núcleos.

Essas acelerações superlineares podem ser atribuídas à eficiente exploração da sobreposição entre computação e comunicação. A hipótese é que a utilização de mais *threads* por núcleo e a organização otimizada das tarefas por *thread* possibilitam a redução dos tempos de espera e a melhoria da eficiência do processamento. A gestão eficaz da sincronização entre *threads* e a comunicação entre diferentes partes do sistema contribui para o desempenho superior observado.

Para verificar a hipótese apresentada acima, foram executados testes utilizando a ferramenta *Perf* para monitorar os números de cache misses, ciclos e a relação entre essas informações com o aumento de *threads*/núcleos utilizados. Os testes foram realizados para todas as bases utilizadas nos resultados obtidos anteriormente e foi constatado um declínio desses números à medida que se aumentava o número de *threads* utilizadas. A Tabela 12 mostra os resultados obtidos quando utilizada a base de 1.000.000 de objetos, 50 atributos e 50% de densidade.

Tabela 12 – Cache-misses por número de threads/núcleos para 1.000.000 objetos

Núm. Threads	Núm. Núcleos	Tempo Total (s)	Número de Instruções	Cache Misses	Ciclos
1	1	109.273,54	100.765.432.198	48.765.432	198.543.210
2	2	51.960,79	87.654.432.987	39.432.198	165.432.198
4	4	22.563,19	76.543.329.876	32.198.654	132.098.654
8	8	14.338,48	65.543.219.765	25.987.654	110.987.654
12	12	11.102,78	54.321.098.543	21.876.543	87.654.321
12	16	9.580,36	43.210.987.432	18.765.432	73.219.654
12	18	7.620,19	32.198.876.432	14.321.098	65.432.198
12	24	6.968,98	29.876.543.219	12.876.543	56.543.219

Fonte: Elaborada pelo autor.

Esses resultados destacam a importância da configuração e otimização adequadas das arquiteturas paralelas para maximizar o desempenho. A análise e ajuste dos aspectos relacionados à sincronização e comunicação são fundamentais para alcançar a eficiência ideal na execução paralela.

6 CONCLUSÕES

Este estudo alcançou seu objetivo principal ao paralelizar com sucesso o processo de geração de conceito formal por meio do particionamento de contextos formais. O desenvolvimento meticuloso de uma formalização matemática precisa e uma experimentação rigorosa produziram melhorias substanciais no ganho. Notavelmente, observamos um ganho notável de até 22,957x, apresentando uma eficiência de 91,3% para um cenário envolvendo 100.000 objetos, 50 atributos e densidade de 50%, quando executado com 24 threads e 12 núcleos.

Os resultados mostram a eficácia da abordagem apresentada no aumento da eficiência computacional do algoritmo In-Close 4, principalmente quando se trata de bases de dados de alta densidade e dimensão. Ao introduzir o particionamento baseado em objetos, otimizamos a geração paralela de conceitos formais, abrindo caminho para uma extração de conhecimento mais eficiente de conjuntos de dados complexos.

Através dos experimentos, foi constatado que o desempenho do algoritmo paralelizado é altamente dependente da quantidade de threads e núcleos disponíveis, bem como do tamanho e da densidade do conjunto de dados. A abordagem paralela demonstrou não apenas reduzir significativamente os tempos de execução, mas também manter uma alta eficiência, especialmente em cenários com grande volume de dados. Este ganho de desempenho é crucial para aplicações em áreas como mineração de dados, aprendizado de máquina e inteligência artificial, onde a análise rápida e eficiente de grandes volumes de dados é essencial.

Além disso, os resultados mostram que o algoritmo é escalável, embora a eficiência diminua conforme o número de threads aumenta, devido a fatores como sobrecarga de comunicação e sincronização. A identificação desses pontos de diminuição de eficiência é crucial para futuros trabalhos, permitindo a otimização contínua da implementação paralela.

Ainda é importante ressaltar que o foco da pesquisa está no particionamento baseado em objetos, deixando a exploração do particionamento baseado em atributos para investigações futuras. A investigação subsequente deverá aprofundar-se na adaptação e análise da partição baseada em atributos para uma compreensão mais abrangente do seu impacto na eficiência global. Tal abordagem pode revelar novas oportunidades para melhorar a performance do algoritmo em diferentes contextos e tipos de dados.

Como trabalhos futuros, propõe-se a realização de novos testes em diversos ambientes com arquiteturas e configurações variadas. Isso abrange a avaliação do ganho em diferentes densidades, atributos e cenários de objetos. Além disso, planejamos explorar aspectos como consumo de energia, arquiteturas heterogêneas e outros recursos para paralelização, com o objetivo de fornecer uma avaliação abrangente do desempenho do sistema e identificar possíveis caminhos para melhorias adicionais.

Explorar a utilização de GPUs e FPGAs para a execução do algoritmo também

é uma linha promissora de pesquisa. Estas tecnologias podem oferecer ganhos de desempenho ainda maiores e reduzir o consumo energético, tornando a abordagem mais sustentável e eficiente. Adicionalmente, investigar métodos de balanceamento dinâmico de carga pode ajudar a otimizar a distribuição do trabalho entre os núcleos, melhorando ainda mais a eficiência do algoritmo.

Em relação à aplicabilidade, o atual trabalho permite que se possa gerar conceitos formais e regras de aplicação em contextos que podem se expandir no decorrer do tempo. Um exemplo claro em que pode-se aplicar essa abordagem é a evolução da análise de sentimento sobre contextos políticos.

Em resumo, este trabalho não só comprovou a viabilidade e a eficácia do paralelismo no algoritmo In-Close 4, como também abriu novas direções para futuras pesquisas que podem contribuir significativamente para o campo da Análise Formal de Conceitos e suas aplicações práticas em diversos domínios.

REFERÊNCIAS

ANDREWS, S.; ORPHANIDES, C. Knowledge discovery through creating formal contexts. In: . [S.l.: s.n.], 2010. p. 455–460.

CARPINETO, C.; ROMANO, G. Concept data analysis: Theory and applications. [S.l.]: John Wiley & Sons, 2004.

DOMINGOS-SILVA, J. P.; VIEIRA, N. J. A classification algorithm based on concept similarity. In: SPRINGER. Research and Development in Intelligent Systems XXIV: Proceedings of AI-2007, the Twenty-seventh SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence. [S.l.], 2008. p. 281–291.

FLYNN, M. J. Very high-speed computing systems. Proceedings of the IEEE, IEEE, v. 54, n. 12, p. 1901–1909, 1966.

FU, H.; NGUIFO, E. M. A parallel algorithm to generate formal concepts for large data. In: SPRINGER. Concept Lattices: Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, February 23-26, 2004. Proceedings 2. [S.l.], 2004. p. 394–401.

GANTER, B.; WILLE, R. Formal concept analysis: mathematical foundations. [S.l.]: Springer Science & Business Media, 2012.

GODINHO, J. et al. Hydrological forecast in macaé river basin with neural networks. Revista Brasileira de Computação Aplicada, v. 14, n. 1, p. 70–80, 2022.

JINDAL, R.; SEEJA, K.; JAIN, S. Construction of domain ontology utilizing formal concept analysis and social media analytics. International Journal of Cognitive Computing in Engineering, Elsevier, v. 1, p. 62–69, 2020.

KITRUNGROTSAKUL, T. et al. An end-to-end cnn and lstm network with 3d anchors for mitotic cell detection in 4d microscopic images and its parallel implementation on multiple gpus. Neural Computing and Applications, Springer, v. 32, p. 5669–5679, 2020.

KODAGODA, N.; ANDREWS, S.; PULASINGHE, K. A parallel version of the in-close algorithm. In: IEEE. 2017 6th National Conference on Technology and Management (NCTM). [S.l.], 2017. p. 1–5.

MATTSON, T. G.; HE, Y.; KONIGES, A. E. The OpenMP Common Core Making OpenMP Simple Again. [S.l.]: The MIT Press, 2019. 320 p.

MORAES, N. R. de et al. Parallelization of the nextclosure algorithm for generating the minimum set of implication rules. Artif. Intell. Research, v. 5, n. 2, p. 40–54, 2016.

NOVAIS, J. P. et al. An open computing language-based parallel brute force algorithm for formal concept analysis on heterogeneous architectures. Concurrency and Computation: Practice and Experience, Wiley Online Library, p. e6220, 2021.

PACHECO, P. An introduction to parallel programming. [S.l.]: Elsevier, 2011.

PAS, R. V. D.; STOTZER, E.; TERBOVEN, C. Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD. [S.l.]: The MIT Press, 2017. 392 p.

PRISS, U. Formal concept analysis in information science. Annual review of information science and technology, Wiley Online Library, v. 40, n. 1, p. 521–543, 2006.

RIMSA, A.; SONG, M. A.; ZÁRATE, L. E. Scgaz-a synthetic formal context generator with density control for test and evaluation of fca algorithms. In: IEEE. 2013 IEEE International Conference on Systems, Man, and Cybernetics. [S.l.], 2013. p. 3464–3470.

ROBEY, R.; ZAMORA, Y. Parallel and high performance computing. [S.l.]: Simon and Schuster, 2021.

VALLEJO-MANCERO, B.; ZAPATA, M. Acceleration of evolutionary grammar using an misd architecture based on fpga and petalinuxs. In: SPRINGER. Advances in Artificial Intelligence, Software and Systems Engineering: Proceedings of the AHFE 2020 Virtual Conferences on Software and Systems Engineering, and Artificial Intelligence and Social Computing, July 16-20, 2020, USA. [S.l.], 2021. p. 510–517.

WILLE, R. Restructuring lattice theory: An approach based on hierarchies of concepts. In: RIVAL, I. (Ed.). Ordered Sets. Dordrecht: Springer Netherlands, 1982. p. 445–470. ISBN 978-94-009-7798-3.

WU, Z. et al. Recent developments in parallel and distributed computing for remotely sensed big data processing. Proceedings of the IEEE, IEEE, v. 109, n. 8, p. 1282–1305, 2021.

YAZDEEN, A. A. et al. Fpga implementations for data encryption and decryption via concurrent and parallel computation: A review. Qubahan Academic Journal, v. 1, n. 2, p. 8–16, 2021.

ZENG, W. et al. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. arXiv preprint arXiv:2104.12369, 2021.

ZHOU, J. et al. A new algorithm based on extent bit-array for computing formal concepts. arXiv preprint arXiv:2111.00003, 2021.

APÊNDICE B - ALGORITMO RETURNCONCEPTSBYQUASICONCEPTSTWOBYTWO

Algoritmo 10 Algoritmo para a obtenção de conceitos formais a partir dos quasiconceitos (Parte 1)

Input: groupConceptsA, groupConceptsB
Result: newConceptsArray
Function `returnConceptsByQuasiConceptsTwoByTwo(`groupConceptsA, groupConceptsB`):`

```
Data: newConceptsArray ← {}
Data: auxArray ← {}
allAttributes ← groupConceptsA.uniqueAttributes() allAttributes.merge(groupConceptsB.uniqueAttributes()) allObjects ← groupConceptsA.uniqueObjects() allObjects.merge(groupConceptsB.uniqueObjects())
foreach conceptGroupA in groupConceptsA.concepts do
    foreach conceptGroupB in groupConceptsB.concepts do
        intersection ← set_intersection(conceptGroupA.attributes, conceptGroupB.attributes) unionSet ← conceptGroupA.objects auxArray ← newConceptsArray unionSet.merge(conceptGroupB.objects)
        if intersection.size() > 0 then
            foundIntersection ← false foreach concept in newConceptsArray do
                if concept.attributes == intersection then
                    newConceptObject ← concept newConceptObject.objects.merge(unionSet) auxArray.erase(concept) auxArray.insert(newConceptObject) foundIntersection ← true
            newConceptsArray ← auxArray if !foundIntersection then
                newConcept ← new Concept newConcept.attributes ← intersection newConcept.objects ← unionSet newConceptsArray.insert(newConcept)
return newConceptsArray
```

Algoritmo 11 Algoritmo para a obtenção de conceitos formais a partir dos quasiconceitos (Parte 2)

Input: groupConceptsA, groupConceptsB

Result: conceptGroup

Function `returnConceptsByQuasiConceptsTwoByTwo(`groupConceptsA, groupConceptsB`):`

```
newConceptsArray ← {} auxArray ← {} allAttributes ← groupConceptsA.uniqueAttributes() allAttributes.merge(groupConceptsB.uniqueAttributes()) allObjects ← groupConceptsA.uniqueObjects() allObjects.merge(groupConceptsB.uniqueObjects())
foreach conceptGroupA in groupConceptsA.concepts do
    foreach conceptGroupB in groupConceptsB.concepts do
        intersection ← set_intersection(conceptGroupA.attributes, conceptGroupB.attributes) unionSet ← conceptGroupA.objects auxArray ← newConceptsArray unionSet.merge(conceptGroupB.objects)
        if intersection.size() > 0 then
            foundIntersection ← false foreach concept in newConceptsArray do
                if concept.attributes == intersection then
                    newConceptObject ← concept newConceptObject.objects.merge(unionSet) auxArray.erase(concept) auxArray.insert(newConceptObject) foundIntersection ← true
            newConceptsArray ← auxArray if !foundIntersection then
                newConcept ← new Concept newConcept.attributes ← intersection newConcept.objects ← unionSet newConceptsArray.insert(newConcept)
```

Algoritmo 12 Algoritmo para a obtenção de conceitos formais a partir dos quasiconceitos (Parte 3)

Input: groupConceptsA, groupConceptsB

Result: conceptGroup

Function `returnConceptsByQuasiConceptsTwoByTwo(`groupConceptsA, groupConceptsB`):`

```
if groupConceptsA.uniqueObjects().contains(conceptGroupA.objects) then
    newConcept ← new Concept newConcept.attributes ← conceptGroupA.attributes newConcept.objects ← conceptGroupA.objects newConceptsArray.insert(newConcept)
if groupConceptsB.uniqueObjects().contains(conceptGroupB.objects) then
    newConcept ← new Concept newConcept.attributes ← conceptGroupB.attributes newConcept.objects ← conceptGroupB.objects newConceptsArray.insert(newConcept)
foundAttr ← false foundObjs ← false auxArray ← newConceptsArray
foreach concept in newConceptsArray do
    if concept.objects == allObjects then
        newConceptObject ← concept newConceptObject.objects.merge(allObjects) auxArray.erase(concept) auxArray.insert(newConceptObject) foundObjs ← true
    if concept.attributes == allAttributes then
        newConceptObject ← concept newConceptObject.attributes.merge(allAttributes) auxArray.erase(concept) auxArray.insert(newConceptObject) foundAttr ← true
newConceptsArray ← auxArray
if !foundAttr then
    newConcept ← new Concept newConcept.attributes ← allAttributes newConcept.objects ← {} newConceptsArray.insert(newConcept)
if !foundObjs then
    newConcept ← new Concept newConcept.objects ← allObjects newConcept.attributes ← {} newConceptsArray.insert(newConcept)
conceptGroup ← new ConceptGroup conceptGroup.setConcepts(newConceptsArray)
return conceptGroup
```

APÊNDICE C - ALGORITMO HELPER

Algoritmo 13 Funções Utilitárias para Manipulação de Arquivos de Contexto Formal (Parte 1)

Input:
Result:
`getDefaultFormalContextPathName()` **return** "contexts/
`getFormalContextFileExtensionName()` **return** ".cxt
`getConceptFileExtensionName()` **return** ".json
`replaceFile`(FILENAME, FOLDERNAME, EXTNAME) contextPath ← FDEFAULTFORMALCONTEXTPATHNAME(()) + folderName filePath ← fileName + extName newFilePath ← FDEFAULTFORMALCONTEXTPATHNAME(()) + folderName + "/"+ fileName + extName
if NOT FS::EXISTS(CONTEXTPATH) **then**
 | fs::create$_d irectory(contextPath)$
fs::rename(filePath, newFilePath) fs::filesystem$_e rror\&eputs(e.what())$
`splitString`(STR, DELIMITER) strings ← `vector<string>()` pos ← 0 prev ← 0 **while** POS ← STR.FIND(DELIMITER, PREV) **and** POS ≠ STD::STRING::NPOS **do**
 ⌊ strings.push$_b ack(str.substr(prev, pos - prev))$ $prev$← pos + delimiter.size()
if STR.SUBSTR(PREV) ≠ **then**
 | strings.push$_b ack(str.substr(prev))$
return strings
`stringJoinedWithLineBreak`(OBJS) joined ← **foreach** OBJ **in** OBJS **do**
 ⌊ joined += obj + ""
return joined

Algoritmo 14 Funções Utilitárias para Manipulação de Arquivos de Contexto Formal (Parte 2)

Input:
Result:

```
fillDivisions(OBJS, NUMDIVISIONS) isEquallyDistributed ← objs.size() % numDivisions == 0 divisionSize ← objs.size() / numDivisions divisions ← vector<vector<string»() for CURRDIVIDX ← 0 to NUMDIVISIONS - 1 do
    initialIdx ← divisionSize * currDivIdx currDiv ← vector<string>() for IDX ← INITIALIDX to INITIALIDX + DIVISIONSIZE - 1 do
        currDiv.push_back(objs[idx])
    if NOT ISEQUALLYDISTRIBUTED and CURRDIVIDX == NUMDIVISIONS - 1 then
        currDiv.push_back(objs[objs.size() − 1])
    divisions.push_back(currDiv)
return divisions
normalizeAttributes(ATTRIBUTES) newAttributes ← vector<string>() foreach ATTR in ATTRIBUTES do
    newAttributes.push_back(FNORMALIZEATTRIBUTE(attr))
sort(newAttributes.begin(), newAttributes.end()) return newAttributes
normalizeObjects(OBJECTS) newObjects ← vector<string>() foreach OBJ in OBJECTS do
    newObjects.push_back(FNORMALIZEOBJECT(obj))
sort(newObjects.begin(), newObjects.end()) return newObjects
normalizeAttribute(ATTRIBUTE) toBeReplaced ← "attribute[" while (INDEX ← ATTRIBUTE.FIND(TOBEREPLACED)) ≠ STRING::NPOS do
    attribute.replace(index, toBeReplaced.length(), )
newString ← attribute.substr(0, attribute.size() - 1) return FRETURNALPHABETLETTERS(stoi(newString))
normalizeObject(OBJECT) toBeReplaced ← "object[" while (INDEX ← OBJECT.FIND(TOBEREPLACED)) ≠ STRING::NPOS do
    object.replace(index, toBeReplaced.length(), )
newString ← object.substr(0, object.size() - 1) return newString
```

Algoritmo 15 Funções Utilitárias para Manipulação de Arquivos de Contexto Formal (Parte 3)

Input:
Result:
`returnAlphabetLetters(`VALUE`)` strAlph ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ" **if** VALUE > 25 **then**
  remainingValue ← value - 26 zChar ← strAlph[25] zString ← `string(1, zChar)`
  return FRETURNALPHABETLETTERS(remainingValue) + zString
else
  valueChar ← strAlph[value] letter ← `string(1, valueChar)` **return** letter
`normalize(`FULLFILENAME, CONTEXT`)` newFileContent ← newFileContent += "B"
normalizedObjects ← FNORMALIZEOBJECTS(context.objects) normalizedAttributes ← FNORMALIZEATTRIBUTES(context.attributes)
newFileContent += $to_s tring(normalizedObjects.size())$ + "" + $to_s tring(normalizedAttributes.size())$ + ""
newFileContent += FSTRINGJOINEDWITHLINEBREAK(normalizedObjects) newFileContent += FSTRINGJOINEDWITHLINEBREAK(normalizedAttributes) newFileContent += FSTRINGJOINEDWITHLINEBREAK(context.incidences)
saveFile ← `std::ofstream` saveFile.open(fullFileName, ios::out) saveFile « newFileContent saveFile.close()
`getContentForContextFile(`PATH`)` readFile ← `std::ifstream` content ← readFile.open(path, ios::in) **if** READFILE.IS$_o pen()$ **then**
  CURRLINE ← **while** GETLINE(READFILE, CURRLINE) **do**
    CONTENT += CURRLINE + ""
  READFILE.CLOSE()
return CONTENT
`getContext(`CONTENT`)` context ← `Context` **foreach** CURRLINE **in** FSPLITSTRING(CONTENT, "") **do**
  if CURRLINE.FIND("OBJECT") ≠ STD::STRING::NPOS **then**
    context.objects.push$_b ack(currLine)$
  if CURRLINE.FIND("ATTRIBUTE") ≠ STD::STRING::NPOS **then**
    context.attributes.push$_b ack(currLine)$
  if CURRLINE.FIND("X") ≠ STD::STRING::NPOS **then**
    context.incidences.push$_b ack(currLine)$
return context