Universidade Federal da Bahia
Instituto de Matemática

Departamento de Ciência da Computação

OTIMIZAÇÃO PARALELA PARA O PAREAMENTO PROBABILÍSTICO DE REGISTROS EM SISTEMAS MULTICORE, MULTI-GPU E HÍBRIDO

Pedro Marcelino Mendes Novaes Melo

TRABALHO DE GRADUAÇÃO

Salvador
20 de março de 2017

PEDRO MARCELINO MENDES NOVAES MELO

OTIMIZAÇÃO PARALELA PARA O PAREAMENTO PROBABILÍSTICO DE REGISTROS EM SISTEMAS MULTICORE, MULTI-GPU E HÍBRIDO

Este Trabalho de Graduação foi apresentado ao Departamento de Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Marcos Ennes Barreto

Salvador
20 de março de 2017

Sistema de Bibliotecas - UFBA

Melo, Pedro Marcelino Mendes Novaes.

Otimização Paralela para o Pareamento Probabilístico de Registros em sistemas multicore, multi-gpu e híbrido / Pedro Marcelino Mendes Novaes Melo – Salvador, 2017.

20p.: il.

Orientador: Prof. Dr. Marcos Ennes Barreto.

Monografia (Graduação) – Universidade Federal da Bahia, Instituto de Matemática, 2017.

1. Primeira palavra-chave. 2. Segunda palavra-chave. 3. Terceira palavra-chave. I. Barreto, Marcos Ennes. II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

CDD – XXX.XX

CDU – XXX.XX.XXX

TERMO DE APROVAÇÃO

PEDRO MARCELINO MENDES NOVAES MELO

OTIMIZAÇÃO PARALELA PARA O PAREAMENTO PROBABILÍSTICO DE REGISTROS EM SISTEMAS MULTICORE, MULTI-GPU E HÍBRIDO

Este Trabalho de Graduação foi julgado adequado à obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Departamento de Ciência da Computação da Universidade Federal da Bahia.

Salvador, DIA de MES de ANO

Prof. Dr. Marcos Ennes Barreto
Universidade Federal da Bahia

Prof. Dr. Murilo do Carmo Boratto
Universidade Estadual da Bahia

Prof. Dr. Vinicius Petrucci
Universidade Federal da Bahia

DIGITE A DEDICATORIA AQUI

AGRADECIMENTOS

DIGITE OS AGRADECIMENTOS AQUI

O que sabemos é uma gota; o que ignoramos é um oceano.

— ISAAC NEWTON

RESUMO

COLOQUE O RESUMO. Se preferir, crie um arquivo separado e o inclua via comando include.

Para evitar problemas de formato neste template (de uso geral), usamos acentuação mostrada abaixo.

```
\c{c} \~{a} \'{a} \^{e} \'{\i}
```

Não precisa fazer dessa forma, caso use pacotes adequados (latin1, etc.).

Palavras-chave: PALAVRAS-CHAVE.

ABSTRACT

COLOQUE O RESUMO EM INGLÊS. Se preferir, crie um arquivo separado e o inclua via comando include.

Keywords: PALAVRAS-CHAVE EM INGLÊS.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

INTRODUÇÃO

1.1 INTRODUÇÃO

Com a difusão das redes sociais e a revolução da internet das coisas observadas na última década, o volume de produção de dados tem crescido exponencialmente, impulsionando pesquisas e popularizando o conceito de *Big Data* tanto no domínio industrial quanto acadêmico. Segundo projeções da IBM, cerca de 15 *petabytes* de dados estruturados e não-estruturados são gerados diariamente. Dessa forma, diversos setores, como o de telecomunicações, saúde e economia, tem sofrido mudanças no modo de lidar com essa enorme quantidade de dados e capturá-los não é mais suficiente, é necessário extrair todo o seu conteúdo e transformá-lo em informação útil e disponível. No entanto, à medida que a produção dos dados aumenta, as tarefas de gerenciamento, que envolvem desde a captura, transformação e processamento até a análise do conhecimento obtido, se tornam demasiadamente complexas.

No que diz respeito ao gerenciamento de grandes conjuntos de dados, frequentemente, faz-se necessário combinar informações referentes à uma mesma entidade no mundo real que são provenientes de diferentes fontes de dados a fim de se obter informações mais completas e significativas, capazes de subsidiar futuras etapas de tomada de decisão. Esse processo de integração de dados geralmente possui características que definem o custo da sua execução, tais como esquemas diferentes de bases de dados, inexistência de chaves unificadas de identificação que facilitem a integração, bem como a disponibilidade dos recursos computacionais existentes para um processamento em tempo oportuno. Outra limitação pertinente a este processo de integração de dados diz respeito à confiabilidade e qualidade das informações e na necessidade de contornar problemas como erros de imputação e valores ausentes. Neste cenário, é fundamental a aplicação de técnicas de ETL (*Extraction, Transformation and Loading*), já que as transformações e *data cleasing* são frequentemente utilizadas para garantir integridade e consistência dos dados, empregando métodos para correção de erros e aplicando regras de negócio exigidas na área em questão.

Em casos onde uma determinada base de dado não possui chaves que consigam identificar um atributo em outras bases de forma determinística, deve-se realizar um pare-

amento utilizando abordagem probabilística. Esse tipo de abordagem implica na escolha de múltiplos potenciais atributos-chave e na determinação de diferentes pesos para eles. Realiza-se, então, uma série de comparações entre os registros, utilizando para tal, métodos de transformações e algoritmos de similaridade, para determinar se de fato dois registros em bases diferentes são correspondentes. Quando existe a possibilidade dos registros possuírem grafias diferentes em várias bases de dados, como é o caso de variáveis do tipo *nome* e *endereço*, por exemplo, ou quando os registros necessitam de anonimização, a exemplo de bases que possuem informações sigilosas, um método de transformação frequentemente utilizado é o *Filtro de Bloom*: uma estrutura capaz de verificar se um elemento pertence ou não a um conjunto. Sua vantagem se deve ao fato de que, por representar strings em texto plano através de um vetor de bits, eles são mais facilmente computados além de manter as informações originais anônimas. Para calcular a similaridade entre dois registros em Filtros de Bloom pode ser utilizado o *Coeficiente de Dice*, que tem como objetivo estabelecer uma relação de aproximação entre dois elementos diferentes, identificando o quão semelhantes eles são entre si.

Além das questões acerca do pareamento de dados já expostas e, considerando que nos dias de hoje grande parte das informações presentes nas empresas, órgãos governamentais e pesquisas científicas possuem um grande volume de dados, as tecnologias tradicionais de integração de dados não são suficientes para garantir a execução, em tempo hábil. Essa limitação é em parte explicada pelo fato de que antes do surgimento das redes sociais e difusão dos nós sensores, transações na internet e *streams*, os dados não eram gerados na velocidade com que acontece atualmente além da limitação da capacidade do hardware que se refletia em poucas oportunidades para uma maior exploração do paralelismo. Por conta disso, grande parte dos softwares desenvolvidos não possuíam alguns requisitos que estão presentes nas tecnologias de processamento de alto desempenho atualmente, tais como escalabilidade e distribuição de tarefas.

Diante disso, inúmeras pesquisas estão sendo desenvolvidas no intuito de prover ferramentas de integração de dados que consigam suprir os presentes desafios da área. Para tanto, alguns paradigmas de programação estão sendo exigidos para direcionar essas pesquisas e um, em especial, é o paradigma de *computação paralela*. Nesse paradigma, divide-se um grande problema em um conjunto de partes menores que podem ser resolvidas de forma simultânea, já que os núcleos de processamento são interligados entre si. Aplicações baseadas nesse paradigma se beneficiam da característica de escalabilidade, suportando um aumento de carga sem perda de desempenho.

Englobando a computação paralela, a *computação heterogênea* é uma importante técnica para resolver problemas de computação intensiva. Para isso, coordena o uso de diferentes tipos de arquiteturas para maximizar o desempenho conjunto. Nos últimos anos as arquiteturas *multicore*, onde os processadores são compostos por múltiplos núcleos, vem viabilizando cada vez mais o ganho de desempenho das aplicações. Uma API (*Application Programming Interface*) que é utilizada com frequência para a programação *multicore* é o OpenMP (*Open Multi-Processing*), que permite desenvolver aplicações paralelas ao nível de *threads*, baseadas no princípio de compartilhamento da memória principal. Outras unidades de processamento podem ser incorporadas às plataformas de alto desempenho a fim de se acelerar a execução das aplicações. As placas gráficas (*GPUs - Graphics Proces-*

sing Unit), por exemplo, deixaram de ser exclusivamente utilizadas para processamento de imagem e passaram a acelerar aplicações de propósito geral. Uma plataforma usual para essa arquitetura é o CUDA (*Compute Unified Device Architecture*), que fornece abstrações sobre a organização das *threads*, memória e sincronização, permitindo paralelizar uma aplicação entre a grande quantidade de ALUs (*Arithmetic Logical Units*) presentes em uma GPU.

Este trabalho pretende apresentar modelos paralelos em plataformas heterogêneas *multicore* e GPU para o problema do pareamento probabilístico de registros, considerando principalmente o desempenho, escalabilidade e acurácia. Pretende-se também estabelecer uma análise comparativa das execuções em cada plataforma específica, destacando questões de distribuição de carga de trabalho. Para isso, foi implementado versões do *AtyImo*, software de correlação probabilística em sistemas distribuídos, para as etapas que demandam maior poder computacional, como é o caso da etapa de comparação, usando as APIs OpenMP e CUDA. Como estudo de caso, utilizou-se bases de dados do projeto *Coorte de 100 milhões de brasileiros*, na qual objetivou-se parear dados governamentais do Brasil com dados socioeconômicos e de saúde do SUS (*Sistema Único de Saúde*).

Como resultado deste trabalho, foi obtido três otimizações paralelas para correlação probabilística: (i) versão *multicore*, usando OpenMP; (ii) versão *single* e *multi-GPU*, usando CUDA; e (iii) versão híbrida (multicore + multi-GPU). Foi utilizado (i) o tamanho de entrada do problema; (ii) número de processadores usados, para as versões *multicore* e híbrida; e (iii) tamanho de *grid* e *blocks*, para as versões GPU e híbrida, como parâmetros do algoritmo. Como métricas de avaliação, utilizou-se o *speedup* e o tempo de execução para cada versão implementada. O trabalho obteve resultados satisfatórios garantindo alta escalabilidade para problemas de entrada de até **XXX** milhões de registros, sendo pioneiro na paralelização probabilística de registros de grandes bases de dados, como as utilizadas como prova de conceito desta pesquisa, em arquiteturas heterogêneas.

1.2 MOTIVAÇÃO

O Brasil dispõe de um grande acervo de bases de dados, tal como citado na introdução. Muitas vezes, estas possuem milhares de registros e são objetos de inúmeros estudos.

1.3 OBJETIVOS

O principal objetivo deste trabalho foi desenvolver otimizações paralelas da ferramenta *AtyImo*, utilizada para realizar correlação probabilística de dados, para a etapa de comparação de registros. Estas otimizações devem ser capazes de lidar com bases de dados no contexto de *big data*.

Os objetivos específicos do projeto foram:

1. Desenvolvimento de uma ferramenta de correlação probabilística de dados utilizando a API *OpenMP*;

2. Desenvolvimento de uma ferramenta de correlação probabilística de dados utilizando a API *CUDA*;

3. Desenvolvimento de uma ferramenta híbrida de correlação probabilística de dados, utilizando as APIs *OpenMP* e *CUDA*;

4. Comparação dos métodos criados com os métodos existentes;

5. Avaliação dos diferentes *speedup's* e tempo de execução entre as versões implementadas;

6. Obtenção da melhor configuração de distribuição de carga para a versão híbrida;

1.4 ORGANIZAÇÃO DO TRABALHO

No próximo capítulo são apresentados os conceitos de computação de alto desempenho, pareamento de registros e demais conteúdos importantes para o entendimento do trabalho. Um levantamento do estado da arte sobre ferramentas de correlação de dados, determinística e probabilística, bem como de técnicas de computação heterogênea para integração de dados serão apresentados no Capítulo 3. Uma breve descrição sobre a ferramenta inspiradora deste trabalho, o *AtyImo*, bem como uma descrição detalhada do processo de desenvolvimento das otimizações paralelas propostas será mostrada no Capítulo 4. O Capítulo 5 exibirá os resultados obtidos e suas respectivas discussões. A conclusão e direcionamento para trabalhos futuros estão no Capítulo 6.

Capítulo

2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os principais fundamentos teóricos envolvidos no processo de desenvolvimento de uma ferramenta otimizada para integração de dados, utilizando a abordagem probabilística, no contexto de *Big Data*.

2.1 PAREAMENTO DE REGISTROS

O pareamento de registros, também conhecido como *record linkage* ou *vinculação de dados*, tem como objetivo combinar informações que estão contidas em diferentes arquivos computadorizados. Um método básico é comparar informações como *nome* e *endereço* para determinar quais pares de registros correspondem à mesma entidade (WINKLER, 2006). O crescente interesse pela vinculação se dá por alguns motivos, dentre eles: (i) arquivos de dados que contém informações importantes tem seu valor aumentado a partir da integração de registros individuais que estão presentes em diferentes bases; (ii) o crescente avanço das arquiteturas computacionais tem permitido executar dados no contexto de *big data*; e (iii) aumenta-se paulatinamente a conscientização de empresas e órgãos governamentais sobre o potencial do *record linkage* para diversas áreas, dentre elas a pesquisa médica e genética (FELLEGI; SUNTER, 1969). Por conta disso, inúmeros trabalhos tem sido desenvolvidos utilizando técnicas de pareamento de registros (JR; COELI, 2000; GÓEZ; MEDRONHO; COELI, 2006; PINTO et al., ; SEHILI et al., 2015).

(FELLEGI; SUNTER, 1969) propuseram um modelo matemático para conceituar o significado de *record linkage*. Para isso, denota-se duas populações, ou base de dados, A e B e seus respectivos elementos, ou registros, a e b, onde cada elemento possui informações diversas, que podem ser nome, sexo e endereço, por exemplo. Assumindo que alguns elementos estão presentes tanto na base A quanto na B, o objetivo é classificar pares do produto espacial

$$A \times B = \{(a, b); a \in A, b \in B\} \tag{2.1}$$

em um dos conjuntos disjuntos

$$M = \{(a, b); a = b, a \in A, b \in B\} \tag{2.2}$$

$$U = \{(a, b); a \neq b, a \in A, b \in B\} \tag{2.3}$$

sendo o M (*matched*), conjunto dos pares que foram pareados, e o U (*unmatched*), conjunto dos pares que não foram pareados. Considerando $\alpha(a)$ e $\beta(b)$ como sendo as informações inerentes aos elementos a e b, respectivamente, e K, os atributos independentes a serem comparados, o *record linkage* deve decidir quais elementos de $A \times B$ irão compor cada um dos conjuntos disjuntos. Isso é possível a partir da construção de um *vetor de comparação*, ou índice de similaridade, dado por

$$\gamma[\alpha(a), \beta(b)] = \{\gamma^1[\alpha(a), \beta(b)], ..., \gamma^K[\alpha(a), \beta(b)]\} \tag{2.4}$$

Esse vetor de comparação pode ser simplificado como $\gamma(a, b)$ ou $\gamma(\alpha, \beta)$ e denota a verificação de similaridade entre cada um dos elementos comuns às duas populações. Por fim, o autor define um *espaço de comparação*, denotado por Γ, que engloba todas as possíveis realizações de γ.

2.1.1 Método Probabilístico

Durante o processo de geração dos dados para preenchimento das populações, diferentes tipos de erros podem ser introduzidos nos registros. Dentre eles, destacam-se o preenchimento e/ou codificação diferente dos dados, já que estes são provenientes de fontes diversas, e incompletude dos registros (FELLEGI; SUNTER, 1969). O *método probabilístico* constrói, também, um vetor de comparação γ com K parâmetros

$$\gamma[\alpha(a), \beta(b)] = \{\gamma^1[\alpha(a), \beta(b)], ..., \gamma^K[\alpha(a), \beta(b)]\} \tag{2.5}$$

sendo esses parâmetros os atributos independentes que serão utilizados para comparação entre os registros a e b. Isso acontece quando as bases de dados não possuem chaves estrangeiras que consigam relacionar entre si os registros deterministicamente. Como mostra a figura 2.5, eventualmente, existirão casos em que pares verdadeiros negativos sejam classificados no grupo U, como também verdadeiros negativos sejam classificados no grupo M. São os chamados pares *falsos negativos* e *falsos positivos*.

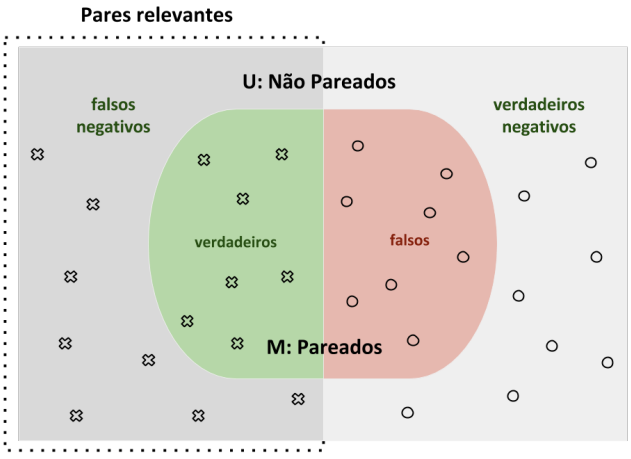

Figura 2.1 Classificação dos pares no processo de pareamento probabilístico de registros

Essa classificação se dá por conta da inexistência da chave e pela presença de erros nos registros. A *acurácia* do método probabilístico, etapa realizada após o pareamento, é dada a partir das seguintes avaliações:

a) $sensibilidade = \frac{verdadeiros\ positivos}{verdadeiros\ positivos + falsos\ negativos}$

b) $especificidade = \frac{verdadeiros\ positivos}{verdadeiros\ positivos + falsos\ positivos}$

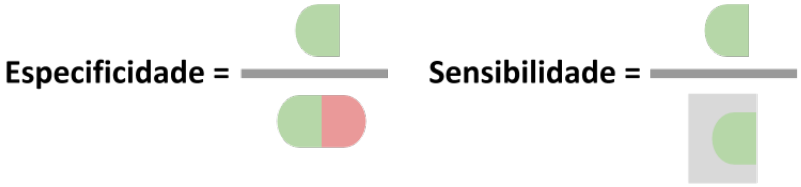

Figura 2.2 Classificação dos pares no processo de pareamento probabilístico de registros

Na prática, o pareamento probabilístico de registros é determinado por funções de similaridade ou distância (KIM; LEE, 2007), denotado por $dist(parametro_1, parametro_2)$, no pseudo-algoritmo abaixo

para cada registro a_i ($\in A$)
para cada registro b_j ($\in B$)
se $dist(a_i, b_j) < \Theta$ então $a_i \approx b_j$

Quando dois registros, a e b se referem à mesma entidade no mundo real, é escrito como $a \approx b$. De acordo com (KIM; LEE, 2007), quando esses dois registros são classificados no grupo M, quatro relacionamentos, ilustrado na figura X, podem ocorrer: (i) $a \sqsupseteq b$: todas as informações de b aparecem em a, (ii) $a \sqsubseteq b$: todas as informações de a aparecem em b, (iii) $a \equiv b$: as informações de a e b são idênticas, e (iv) $a \oplus b$: parte das informações de a e b se equivalem além de um *ponto de corte* Θ determinado. A notação $a \not\approx b$ refere-se quando os registros a e b não representam a mesma entidade no mundo real ou se as informações referentes à esses registros estão muito abaixo do ponto de corte.

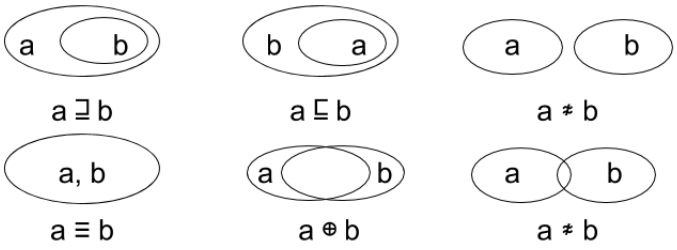

Figura 2.3 Classificação dos pares no processo de pareamento probabilístico de registros

2.1.2 Filtros de Bloom

Em muitos bancos de dados, alguns identificadores possuem informações de cunho pessoal, por exemplo: *nome, endereço, referências sobre doença e/ou salário, etc.* Em muitos países, a distribuição e uso para pesquisas científicas de tais bases de dados é um tanto restrita e, por conta disso, é necessário o uso de técnicas de criptografia de dados que consigam tanto garantir o sigilo das informações quanto manter as características necessárias para realização do pareamento. Alguns algoritmos que usam criptografia baseada em funções *hash* conseguem anonimizar as informações e preservar o seu conteúdo, no entanto requerem um pareamento exato dos identificadores. Dessa forma, são ineficientes no *record linkage* quando esses dados possuem diferenças entre si (SCHNELL; BACHTELER; REIHER, 2009). Nesse cenário, o *Filtro de Bloom*, método para aproximação de strings, se apresenta como uma alternativa viável.

Inicialmente proposto por (BLOOM, 1970), o *Filtro de Bloom* é utilizado para determinar se dois elementos são aproximadamente idênticos (JAIN; DAHLIN; TEWARI, 2005). É constituído de um vetor binário V de tamanho n que inicia todos os seus *bits* com o valor *0*. O objetivo do algoritmo de Filtro de Bloom é armazenar um determinado conjunto $S = \{x_1, x_2, ..., x_n\}$, onde cada elemento desse conjunto pode ser caracteres de uma única *string* ou atributos utilizados para o pareamento, em V. Para isso, k funções *hash* independentes $h_1, ..., h_k$ são utilizadas sobre cada elemento $x_i \in S$ para definir quais posições dentro de V terão seu valor alterado para *1*.

Como exemplo, a figura XX exemplifica a atuação do Filtro de Bloom sobre um conjunto S, que armazena informações sobre os atributos *nome, município de residência e sexo.*

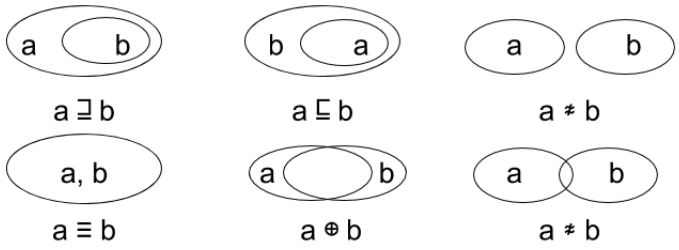

Figura 2.4 Classificação dos pares no processo de pareamento probabilístico de registros

2.1.3 Cálculo de Similaridade e Classificação

No pareamento probabilístico de registros, a etapa de *comparação* dos registros presentes no produto espacial $A \times B$ é baseada no cálculo da probabilidade de concordância e discordância entre os elementos de um determinado par. Essa probabilidade é empregada usando alguma metodologia para verificação de similaridade, principalmente na etapa $\gamma(a, b)$. O *Coeficiente de Dice*, ou Coeficiente de Sorensen-Dice, foi criado para expressar de forma quantitativa, em estudos ecológicos, o grau em que duas espécies distintas estão

associadas em diferentes ecossistemas (DICE, 1945). Dado duas espécies, A e B, o autor propõe inicialmente um *índice de associação*, que é obtido a partir da divisão de um número a de amostras aleatórias de uma dada série em que a espécie A ocorre por um número h de amostras nas quais as espécies A e B ocorrem em conjunto:

$$Indice\ de\ associacao\ B/A\ = \frac{h}{a} \tag{2.6}$$

$$Indice\ de\ associacao\ A/B\ = \frac{h}{b} \tag{2.7}$$

O Coeficiente de Dice é gerado a partir dos índices de associação e possui um valor intermediário entre os dois índices de associação:

$$Coeficiente\ de\ Dice\ = \frac{2h}{a+b} \tag{2.8}$$

onde (i) h representa o somatório dos números de espécies a e b que coincidem entre si, (ii) a corresponde ao número de ocorrências de uma determinada espécie A, e (iii) b corresponde ao número de ocorrências de uma determinada espécie B. Essa técnica pode ser aplicada no contexto do pareamento de registros para uso como coeficiente de similaridade entre dados de bases diferentes, sendo a e b contadores de um determinado carácter nos Filtros de Bloom A e B, respectivamente. A figura XX exemplifica como o Coeficiente de Dice realiza o cálculo de similaridade entre dois filtros.

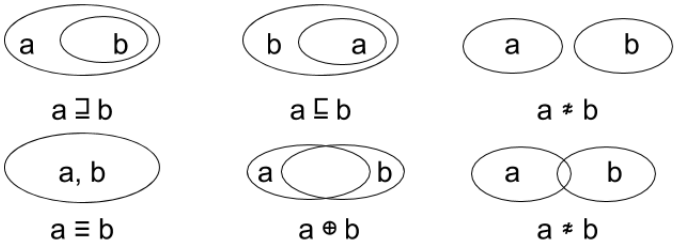

Figura 2.5 Classificação dos pares no processo de pareamento probabilístico de registros

2.2 COMPUTAÇÃO DE ALTO DESEMPENHO

Em 1988, um artigo do *Wall Street Journal*, intitulado *Attack of the Killer Micros*, descreveu como os computadores *commodities*, ou computadores pessoais (PCs, do inglês *Personal Computers*), poderiam ser equivalentes aos supercomputadores. De fato, existem casos em que um agrupamento de computadores pessoais pode alcançar níveis de desempenho próximos ou equivalentes aos níveis de um supercomputador, já que há muito mais tecnologias sendo desenvolvidas com a finalidade de melhorar o desempenho de tais computadores do que os supercomputadores. Além disso, o mercado dos PCs se mostrou extremamente promissor com o surgimento de aplicações empresariais, jogos, sistemas multimídia, entre outras. A computação de alto desempenho (HPC, do

inglês *High-Performance Computing*) se refere ao uso de supercomputadores ou *clusters* de vários computadores *commodities* no intuito de agregar desempenho para acelerar aplicações que quererem grandes recursos computacionais.

2.2.1 Conceitos na Computação Paralela

Segundo a *Lei de Moore*, o número de transístores em um circuito eletrônico dobraria a cada dezoito meses sem que os custos de produção aumentassem (MOORE et al., 1998). Essa lei se tornou a base norteadora das metas de produção dos processadores pelas grandes fabricantes. No entanto, ao passo que a predição de Moore se mostrava acertada, o espaço disponível para inserção desses transístores diminuía e o nível de consumo de energia, implicado pelas altas taxas do *clock*, aumentava. Tornava-se cada vez mais complexo contornar os problemas de dissipação térmica produzida pelos componentes do *hardware*.

Nesse cenário, surgiu as *arquiteturas de computação paralela* (VAJDA, 2011), comumente conhecidas como *arquiteturas multicore* (exemplificada na figura 2.6), onde um processor com diversos núcleos (*cores*) explora técnicas de paralelismo, tanto em nível de instruções (ILP - *Instruction-level Parallelism*) quanto em nível de *threads* (TLP - *Threads-level Parallelism*), *pipelining* e processamento com *hyperthreading* (MARR et al., 2002). Essas arquiteturas *multicore* possuem dois, quatro, seis ou mais núcleos de processamento. Seguindo essa linha de raciocínio, poderiam incluir centenas de núcleos, no entanto, devido às altas frequências necessárias ao funcionamento de tais arquiteturas, o calor gerado seria excessivo. A partir disso, surgiram as arquiteturas *manycore*, que tem como base as unidades de processamento gráfico (GPU), compostas de centenas de processadores capazes de explorar o paralelismo em nível de dados (*DLP - Data-level Parallelism*).

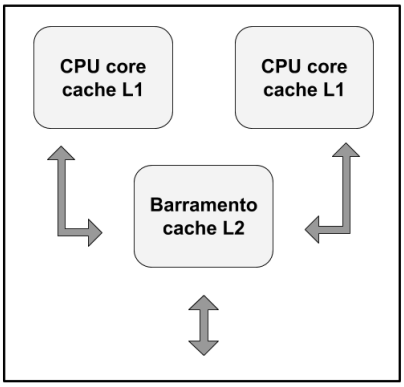

Figura 2.6 Visão genérica da organização de um processador *dual-core* com memória *cache* interna e compartilhada

Antes do surgimento dessas arquiteturas, os softwares eram projetados para serem executados em um único *core*. Após a introdução das arquiteturas *multicore* e a adoção

dessa por parte das aplicações, alguma mudanças ocorreram no processo de desenvolvimento de software e hardware: (i) surgiu a necessidade de se implementar novas técnicas de gerenciamento e comunicação entre os recursos do computador e, principalmente, (ii) mudou-se o paradigma utilizado para o desenvolvimento das aplicações, passando a ser utilizado o paradigma da *programação paralela*. Como consequência, os desenvolvedores passaram a necessitar de um conhecimento mais aprofundado das novas arquiteturas para poderem extrair o máximo de desempenho. Abaixo é listado alguns conceitos presentes na computação paralela.

- **Paralelismo:** múltiplos *threads* executando uma tarefa simultaneamente
- **Concorrência:** duas aplicações são ditas concorrentes quando estas precisam acessar um recurso compartilhado, uma informação na memória, por exemplo, ao mesmo tempo
- **Programação paralela:** técnicas de programação para uso de múltiplos *cores* ou múltiplos computadores
- **Computação *multicore*:** computação com sistemas que proveem múltiplos circuitos computacionais por CPU
- **Computação distribuída:** computação com sistemas que consiste em múltiplos computadores conectados por uma rede de comunicação
- **Paralelismo em nível de instrução:** um programa consiste em um fluxo de instruções executadas pelo processador. Reordenar e combinar em grupos as intruções para que sejam executadas em paralelo sem que altere o resultado final do programa é conhecido como paralelismo em nível de instrução
- **Paralelismo em nível de *threads*:**
- **Paralelismo em nível de dados:** o mesmo processamento é aplicado à um grande conjunto de dados em paralelo
- **Passagem de memória:** modelo de comunicação entre processos onde esta é executada na forma de operações do tipo *send* e *receive*. A comunicação pode ser tanto síncrona, onde o receptor deve estar pronto para receber a mensagem, ou assíncrona, onde a mensagem pode ser enviada antes do receptor estar pronto para recebê-la (EL-REWINI; ABD-EL-BARR, 2005)
- **Memória compartilhada:** modelo de memória que pode ser acessada, tanto para escrita quanto para leitura, simultaneamente por múltiplos *threads* e/ou processos, tendo como objetivo principal prover comunicação entre eles (EL-REWINI; ABD-EL-BARR, 2005). Esse modelo deve prover também controle de acesso, sincronização e proteção dos dados

2.2.2 Computação para Multicore

Com a mudança da arquitetura dos computadores, passando de monoprocessado para multiprocessado, diversas ferramentas de programação paralela surgiram com o objetivo de prover abstrações na exploração eficiente dos recusos de tais arquiteturas. Tanto ferramentas desenvolvidas pelos fabricantes de *software*, quanto de código aberto oferecem aos desenvolvedores de software bibliotecas para os diversos modelos de arquiteturas. No contexto da computação para *multicore*, destacam-se o MPI (*Message Passing Interface*) e o OpenMP (*Open Multi-Processing*).

2.2.2.1 MPI é um padrão industrial que utiliza a troca de mensagem como a principal forma de comunicação entre os processos (MPI. . . , 2017). Para realizar tal comunicação entre os processos, o MPI move os dados de um espaço de armazenamento de um processo para o espaço de armazenamento de outro processo e esse trabalho é realizado através de operações, já implementadas, de troca de mensagem, tais como *send* e *receive*. As principais vantagens do estabelecimento de um padrão de troca de mensagens são a *portabilidade* e a *facilidade de utilização*, já que o MPI possui funções que controlam a passagem de mensagem, a exemplo do *socket* (RIBEIRO; FERNANDES, 2013). Isso permite eximir o programador de obter esse conhecimento.

2.2.2.2 OpenMP é uma API que oferece suporte para a programação paralela. Para isso, possui diretivas de compilação, funções provenientes da biblioteca padrão e variáveis de ambiente que, em conjunto com a memória compartilhada, possibilitam o desenvolvimento de aplicações em sistemas *multicore* (OPENMP, 2015). Por possuir compartilhamento de memória, o paralelismo é atingido na utilização do modelo *fork-join* e na criação de múltiplos *threads*, que executaram em determinadas regiões paralelas (ver figura 2.7).

As *diretivas de compilação* permitem criar regiões paralelas, distribuir o trabalho a ser realizado entre os *threads* criados, controlar o acesso aos dados (especificando se são privados ou compartilhados) e gerir a sincronização desses *threads*. Já as *funções da biblioteca* são responsáveis por definir o número de *threads* que será criado, obter informações sobre os mesmos bem como gerir *locks*. As *variáveis de ambiente* definem, também, a quantidade de *threads* além de controlar o escalonamento e definir o máximo aninhamento de regiões paralelas. Abaixo segue um exemplo de código extraído em (OPENMP-EXAMPLES, 2015), na linguagem C, que realiza a soma de dois vetores, armazenando-a em um terceiro vetor, com comentários a seguir.

Algorithm 1 Exemplo de soma de vetores utilizando a API OpenMP

```
1:    #include <stdio.h>
2:    #include <omp.h>
3:
4:    #define N 10000
5:
6:    int main(int argc, char *argv[]) {
7:        int vetorA[N];
8:        int vetorB[N];
9:        int vetorC[N];
10:
11:       #pragma omp parallel for
12:       {
13:           for (int i = 0; i < N; i++)
14:           vetorC[i] = vetorA[i] + vetorB[i];
15:       }
16:
17:       return 0;
18:   }
```

- É necessário a inclusão da biblioteca OpenMP (`omp.h`) presente na linha 2
- `#pragma omp` ⟨`nome da diretiva`⟩ ⟨`[cláusula, ...]`⟩ é o formato padrão de uma diretiva no OpenMP
- `#pragma omp parallel` ⟨`[cláusula, ...]`⟩ é conhecido como construtor paralelo além de ser a diretiva mais importante do OpenMP, uma vez que é o responsável pela indicação da região do código que será executada em paralelo. Esse é o construtor utilizado na linha 9, em conjunto com a cláusula `for`
- Para a compilação, é necessário utilizar um compilador que consiga interpretar os `#pragmas`. Para utilizar o *gcc* (*GNU Compiler COllection*), é necessário utilizar a opção `-fopenmp`
- Para definir a quantidade de *threads* que será criada, pode-se utilizar a cláusula `num_threads(`*`n_th`*`)`, sendo o `n_th` a quantidade exata de *threads*. Outra opção é utilizar uma variável de ambiente, `OMP_NUM_THREADS=`*`n_th`*, no momento da execução do código

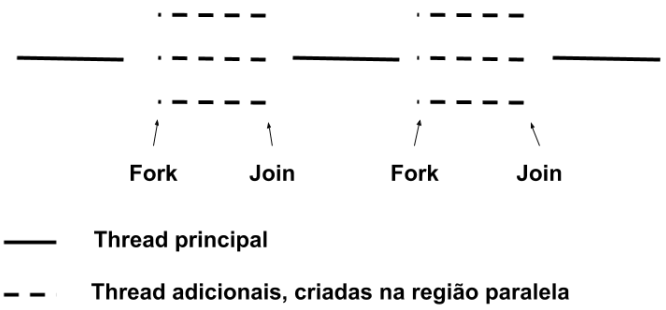

Figura 2.7 Modelo *fork-join* para paralelização, utilizado pelo OpenMP

2.2.3 Computação para GPU

As unidades de processamento gráfico (GPU) se popularizaram à medida em que aumentou-se o consumo de aplicações que empregavam gráficos 3D, a exemplo dos jogos para computador. Ao perceber a alta demanda por *hardware* que conseguisse processar essas aplicações em específico, a empresa americana NVIDIA lançou sua primeira GPU em 1999, a *GeForce 256* (GEFORCE. . . , 1999). Inicialmente, a GPU era constituída de um processador que exercia uma única função: executar aplicações gráficas em três dimensões. Sua característica peculiar é a presença de uma grande quantidade de unidades lógicas (ALU - *Arithmetic Logic Unit*) para processamento do que para *caching* de dados e controle de fluxo, característica das CPUs (BORATTO; COMPUTADORES; BARRETO,) (ver figura 2.8).

Devido à grande capacidade computacional das GPUs, a NVIDIA criou então a GPGPU (*General-Purpose Computing on Graphics Processing Units*), uma GPU de propósito geral capaz de realizar operações complexas da classe SIMD (*Single Instruction Multiple Data*) com extrema facilidade e rapidez. Essas GPUs de propósito geral são divididas em vários SMs (*Streaming Multiprocessors*), onde são executados grupos de *threads*, chamados de *warps*. Vários núcleos, chamados de CUDA *cores*, estão presentes em cada SM e é onde são executadas as operações aritméticas. Esse número considerável de núcleos explica a rapidez com que a GPU executa suas aplicações.

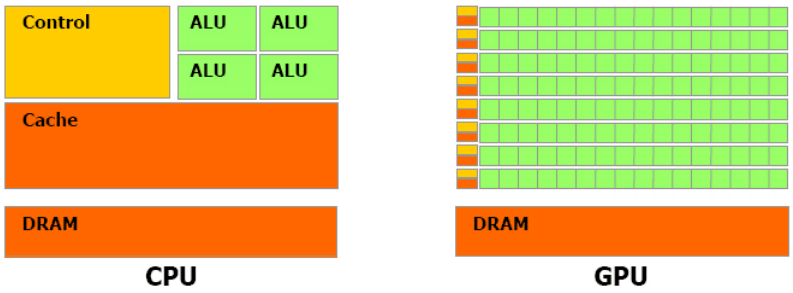

Figura 2.8 Arquitetura de uma CPU *vs* Arquitetura de uma GPU (NVIDIA, 2017)

Assim como as ferramentas de computação *multicore*, o surgimento de ferramentas, como DirectX e OpenGL, forneceu aos programadores diversas bibliotecas capazes de extrair o máximo das placas gráficas, permitindo o desenvolvimento de aplicações diversas para serem executadas nessa arquitetura. Um das mais populares é a plataforma CUDA (*Compute Unified Device Architecture*, que será explorada no tópico abaixo.

2.2.3.1 CUDA é uma API que permite o desenvolvimento de software para GPUs, fornecendo abstrações com respeito à organização hierárquica dos *threads*, memória e sincronização. Foi desenvolvida pela NVIDIA e possui suporte a diversas linguagens, dentre elas C/C++. Abaixo são listadas algumas características presentes no desenvolvimento de aplicações com essa API, seguido de um código, extraído de CITAR, exemplificando uma soma de vetores.

- Ao programar para placas gráficas, é necessário deixar explícito o que será executado na CPU (chamada de *host*) e o que será executado na GPU (chamada de *device*). A função principal a ser executada no *device* recebe o nome de *kernel*

- Os qualificadores de tipo da função definem quais funções serão executadas no *host* e no *device*. O qualificador `__global__` define uma função que será executada na GPU e é chamada a partir da CPU, comumente chamada de *kernel*. O qualificador `__device__` define uma função que será executada na GPU e tem sua chamada a partir da GPU, apenas. Já o qualificador `__host__` define uma função que é executada somente na CPU.

- A API CUDA introduz dois novos conceitos para o escalonamento dos *threads*: *bloco* e *grid*. Um bloco é a unidade básica de organização dos *threads* e de mapeamento para o *hardware*, podendo possuir até três dimensões (1D, 2D ou 3D). O *grid* é a unidade básica onde estão distribuídos os blocos e é onde está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU, para uma determinada função. Como o bloco, um *grid* pode possuir até três dimensões. A figura 2.8 exibe a estrutura de um grid de dimensões 2x3 e blocos de tamanho 3x4. Essa API fornece, também, acesso aos índices e dimensões dos *threads*, blocos e *grid*. Cada *thread* possui um índice e este pode ser acessado através da variável

`threadIdx`. Cada bloco dentro do *grid* fornece também um identificador, dado pela variável `blockIdx`, enquanto que `blockDim` armazena a dimensão de um bloco de *threads*.

- Para realizar a computação na GPU, é necessário alocar os dados e transferí-los para o *device*. Isso é feito a partir das funções da API `cudaMalloc()` e `cudaMemcpy()`, para alocar e transferir os dados, respectivamente. Após realizar a computação, `cudaFree()` libera o espaço de memória utilizado na GPU

- Uma vez alocado e transferido os dados, na chamada de função do *kernel* é necessário informar as dimensões do *grid* e bloco, além dos argumentos que forem necessários para a aplicação. As informações do *grid* e bloco são delimitadas pelos caracteres <<< e >>> e devem ter dimensões compatíveis com o tamanho de entrada do problema.

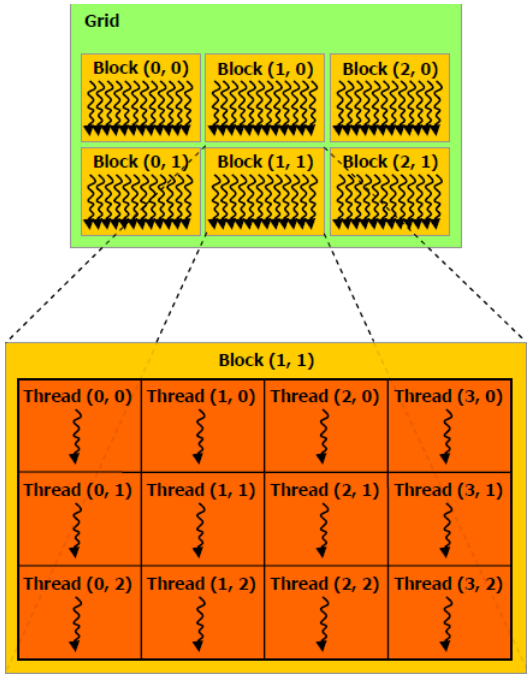

Figura 2.9 Organização do *grid* e bloco (NVIDIA, 2017)

Algorithm 2 Exemplo de soma de vetores utilizando a API OpenMP

```
1:    #include <stdio.h>
2:    #include <cuda.h>
3:
4:    #define N 100000
5:
6:    __global__ void kernel(int *A, int *B, int *C) {
7:        int i = threadIdx.x;
8:        C[i] = A[i] + B[i];
9:    }
10:
11:   int main() {
12:       int A[N], B[N], C[N];
13:       int *A_device, *B_device, *C_device;
14:
15:       cudaMalloc((int **) &A_device, sizeof(int) * N);
16:       cudaMalloc((int **) &B_device, sizeof(int) * N);
17:       cudaMalloc((int **) &C_device, sizeof(int) * N);
18:
19:       cudaMemcpy(A_device, A, sizeof(int) * N, cudaMemcpyHostToDevice);
20:       cudaMemcpy(B_device, B, sizeof(int) * N, cudaMemcpyHostToDevice);
21:
22:       // Chamda do kernel com 1 bloco e N threads
23:       kernel<<<1, N>>>(A, B, C);
24:
25:       cudaMemcpy(C, C_device, sizeof(int) * N, cudaMemcpyDeviceToHost);
26:
27:       cudaFree(A_device);
28:       cudaFree(B_device);
29:       cudaFree(C_device);
30:
31:       return 0;
32:   }
```

- Utilizando a linguagem C, é necessário importar a biblioteca `cuda.h` da API CUDA (linha 2)
- Na linha 6 inicia-se a função *kernel*, com o qualificador `__global__`
- Como o código manipula um vetor de dimensão 1, utiliza-se o identificador `threadIdx.x` para notação do índice do vetor. Caso fosse uma matriz (2 dimensões, por exemplo), utilizaria a seguinte notação: `threadIdx.y` para a segunda dimensão

- Nas linhas 15, 16 e 17 aloca-se as variáveis na GPU, enquanto que nas linhas 19 e 20 é copiado os dados das variáveis no *host* para as variáveis no *device*
- A chamada do *kernel* é feita na linha 23, onde são passados a dimensão do *grid* ou quantidade de blocos (no código tem dimensão 1) e a quantidade de *threads* por bloco (no código possui N threads), respectivamente, entre <<< e >>>
- O resultado da computação na GPU é transferida para uma variável do *host* na linha 25, enquanto que é liberado a memória do *device* nas linhas 27, 28 e 29

2.2.4 Computação heterogênea

Como explanado nas seções 2.2.1 e 2.2.3 desse capítulo, por possuírem um grande número de núcleos de processamento, as GPUs deixaram de ser usadas exclusivamente para processamento gráfico e passaram a suportar aplicações de propósito geral. Assim como as GPUS, as arquiteturas *multicore* tem possibilitado obter um alto ganho de desempenho frente às arquiteturas monoprocessadas. Os avanços no desenvolvimento dessas arquiteturas, mais robustas e paralelizáveis, possibilitaram utilizá-las em conjunto no intuito de prover soluções para problemas que envolvem a computação intensiva, a exemplo de cálculos matemáticos e processamento de grandes bancos de dados. Esse uso coordenado das diferentes arquiteturas é denominado *Computação Heterogênea* (HC, do inglês *Heterogeneous Computing*) (MAHESWARAN; BRAUN; SIEGEL, 1999). Um *cluster* composto por diferentes tipos (modelos ou gerações) de máquinas pode constituir um sistema de computação heterogênea ou, alternativamente, ser tratado como uma simples máquina em um sistema HC mais amplo.

O autor (BRAUN; SIEGEL; MACIEJEWSKI, 2001) define que um problema clássico da HC: uma aplicação é composta por uma ou mais tarefas independentes e que algumas dessas tarefas podem ser decompostas em duas ou mais subtarefas. Essas subtarefas possuem dependência de dados entre elas, mas podem ser atribuídas a diferentes máquinas para execução, onde estas possuem algum canal de comunicação. Dado o poder computacional de cada componente do sistema HC e quais tipos de tarefas se comportam melhor em cada um desses, o objetivo é distribuir as subtarefas entre os componentes a fim de obter ganho de desempenho na aplicação (ver figura XX). [poderia falar sobre os desafios da area: comunicacao, arquiteturas diferentes => balanco de carga, etc]

Além das arquiteturas *multicore* e GPU, os *coprocessadores*, principalmente os baseados na arquitetura Intel MIC (*Many Integrated Core*), tem sido largamente utilizado na computação heterogênea (HUANG et al., 2015), sendo adotados no intuito de empregar soluções utilizando o paralelismo para alcançar tanto um alto ganho de desempenho quanto escalabilidade e baixo consumo de energia. A arquitetura geral de um Intel MIC (*Many Integrated Core*), um dos coprocessadores mais utilizados na indústria, é composta de vários núcleos de processamento interligados através de um barramento de alta velocidade. O *Knights Corner*, coprocessador da linha Intel *Xeon Phi* lançado em 2011, possui 61 *cores* de baixo consumo de energia e cada *core* executa quatro *threads* em paralelo. Essa arquitetura utiliza o modelo de memória compartilhada e permite utilizar modelos tradicionais da computação *multicore*, tais como OpenMP.

REFERÊNCIAS BIBLIOGRÁFICAS

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, ACM, v. 13, n. 7, p. 422–426, 1970.

BORATTO, M.; COMPUTADORES, N. d. A. de; BARRETO, M. Programação em arquiteturas paralelas utilizando sistemas multicore, multi-gpu e co-processadores. *Livro dos Minicursos do WSCAD 2016*.

BRAUN, T. D.; SIEGEL, H. J.; MACIEJEWSKI, A. A. Heterogeneous computing: Goals, methods, and open problems. In: SPRINGER. *International Conference on High-Performance Computing*. [S.l.], 2001. p. 307–318.

DICE, L. R. Measures of the amount of ecologic association between species. *Ecology*, Wiley Online Library, v. 26, n. 3, p. 297–302, 1945.

EL-REWINI, H.; ABD-EL-BARR, M. *Advanced computer architecture and parallel processing*. [S.l.]: John Wiley & Sons, 2005.

FELLEGI, I. P.; SUNTER, A. B. A theory for record linkage. *Journal of the American Statistical Association*, Taylor & Francis Group, v. 64, n. 328, p. 1183–1210, 1969.

GEFORCE 256. 1999. ⟨http://www.nvidia.com/page/geforce256.html⟩. Acessado: 23-05-2017.

GÓEZ, S. M. C.; MEDRONHO, R. d. A.; COELI, C. M. Relacionamento probabilístico entre bases de dados sobre medicamentos e notificação: Uma aplicação na vigilância da aids. *Cad. saúde colet.,(Rio J.)*, v. 14, n. 2, p. 313–326, 2006.

HUANG, M. et al. Study of parallel programming models on computer clusters with intel mic coprocessors. *International Journal of High Performance Computing Applications*, SAGE Publications, p. 1094342015580864, 2015.

JAIN, N.; DAHLIN, M.; TEWARI, R. Using bloom filters to refine web search results. In: *WebDB*. [S.l.: s.n.], 2005. p. 25–30.

JR, K. R. d. C.; COELI, C. M. Reclink: aplicativo para o relacionamento de bases de dados, implementando o método probabilistic record linkage. *Cadernos de Saúde Pública*, SciELO Public Health, v. 16, n. 2, p. 439–447, 2000.

KIM, H.-s.; LEE, D. Parallel linkage. In: ACM. *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. [S.l.], 2007. p. 283–292.

MAHESWARAN, M.; BRAUN, T. D.; SIEGEL, H. J. Heterogeneous distributed computing. *Wiley encyclopedia of electrical and electronics engineering*, Wiley Online Library, 1999.

MARR, D. et al. Hyper-threading technology in the netburst® microarchitecture. *14th Hot Chips*, 2002.

MOORE, G. E. et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, v. 86, n. 1, p. 82–85, 1998.

MPI. Message Passing Interface. 2017. ⟨http://mpi-forum.org/⟩. Accessed: 2017-05-21.

NVIDIA. 2017. ⟨http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4hppS02FZ⟩. Acessado: 23-05-2017.

OPENMP. 2015. ⟨http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf⟩. Accessed: 2017-05-21.

OPENMP-EXAMPLES. 2015. ⟨http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf⟩. Accessed: 2017-05-21.

PINTO, C. et al. Correlação probabilística de bancos de dados governamentais.

RIBEIRO, N. S.; FERNANDES, L. G. L. *Programação Híbrida: MPI e OpenMP*. 2013.

SCHNELL, R.; BACHTELER, T.; REIHER, J. Privacy-preserving record linkage using bloom filters. *BMC medical informatics and decision making*, BioMed Central, v. 9, n. 1, p. 41, 2009.

SEHILI, Z. et al. Privacy preserving record linkage with ppjoin. In: *BTW*. [S.l.: s.n.], 2015. p. 85–104.

VAJDA, A. Multi-core and many-core processor architectures. In: *Programming Many-Core Chips*. [S.l.]: Springer, 2011. p. 9–43.

WINKLER, W. E. Overview of record linkage and current research directions. In: CITESEER. *Bureau of the Census*. [S.l.], 2006.