JAQUELINE ALVARENGA SILVEIRA

UMA BIBLIOTECA PARA AUXILIAR O DESENVOLVIMENTO DE APLICAÇÕES QUE PROCESSAM GRANDES MATRIZES ARMAZENADAS EM MEMÓRIA EXTERNA

Dissertação apresentada a Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2014

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
S587b
2014

Silveira, Jaqueline Alvarenga, 1988-
Uma biblioteca para auxiliar o desenvolvimento de aplicações que processam grandes matrizes armazenadas em memória externa / Jaqueline Alvarenga Silveira. – Viçosa, MG, 2014.
xi, 69f. : il. (algumas color.) ; 29 cm.

Inclui apêndice.
Orientador: Marcus Vinícius Alvim Andrade.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.58-61.

1. Processamento de dados. 2. Algoritmos. 3. Memória cache. 4. Sistemas de Informação. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.3

JAQUELINE ALVARENGA SILVEIRA

UMA BIBLIOTECA PARA AUXILIAR O DESENVOLVIMENTO DE APLICAÇÕES QUE PROCESSAM GRANDES MATRIZES ARMAZENADAS EM MEMÓRIA EXTERNA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 27 de junho de 2014.

Carlos Antônio Alvares Soares Ribeiro

Ricardo dos Santos Ferreira

Marcus Vinícius Alvim Andrade
(Orientador)

"A single conversation with a wise man is better than ten years of study"
(Chinese Proverb)

AGRADECIMENTOS

Agradeço primeiro a Deus, que sempre segurou-me em todos os momentos difíceis durante toda esta longa jornada.

Aos meus pais, Rogério e Margareth, por estarem sempre ao meu lado apoiando-me e incentivando-me a sempre seguir em frente e nunca desistir dos meus sonhos. Agradeço à toda a minha família que sempre apoiou-me enviando boas energias e torcendo para sempre tudo dar certo.

Aos amigos de Viçosa que sempre propiciaram-me momentos de descontração durante o período que estudei na Universidade Federal de Viçosa.

Gostaria de deixar registrada aqui a minha imensa gratidão ao Prof. Marcus que dedicou seu tempo e atenção para me incentivar e auxiliar a realização das pesquisas e o desenvolvimento desta dissertação. Por fim, quero agradecer ao Prof. Salles que contribuiu muito com seu conhecimento e trabalho e os colegas de laboratório que contribuíram muito para a realização deste trabalho. Por fim, eu também quero agradecer aos outros professores que ministraram matérias importantes para a minha formação no mestrado e ao departamento por fornecer apoio e condições de trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

SILVEIRA, Jaqueline Alvarenga, M.Sc., Universidade Federal de Viçosa, junho de 2014. **Uma biblioteca para auxiliar o desenvolvimento de aplicações que processam grandes matrizes na memória externa.** Orientador: Marcus Vinícius Alvim Andrade.

Este trabalho apresenta uma biblioteca, chamada *TiledMatrix*, para auxiliar o desenvolvimento de aplicações que processam grandes matrizes armazenadas na memória externa. A biblioteca é baseada em algumas estratégias similares ao gerenciamento de uma memória cache e seu objetivo básico é permitir que uma aplicação, desenvolvida para processamento na memória interna, possa ser adaptada para ser processada na memória externa. Ela oferece uma interface para acessar a memória externa similar ao tradicional método de acesso a uma matriz. Além disso, ela divide a matriz em blocos bidimensionais, sendo que os blocos são organizados no disco de modo a tirar proveito do padrão de acesso das aplicações. A fim de reduzir o tempo de transferência do bloco para e do disco, a *TiledMatrix* utiliza um algoritmo de compressão/descompressão de dados que foi implementado em paralelo para melhorar ainda mais a eficiência da biblioteca. Assim, a *TiledMatrix* foi implementada e testada em algumas aplicações que requerem intensivo processamento de matrizes tais como: calcular a transposta da matriz, calcular a área de visibilidade, fluxo acumulado e filtro da mediana. Estas aplicações foram implementadas em duas versões: uma usando a *TiledMatrix* e outra usando a *Segment*, uma biblioteca que está incluída no GRASS, um sistema de informação geográfica *open source*. Em média, elas foram 7 vezes mais rápidas com a *TiledMatrix* e, em alguns casos, foram em torno de 18 vezes mais rápidas. A *TiledMatrix* também foi avaliada com o *ArcGIS* e considerou-se duas aplicações que estão incluídas no ArcGIS: calcular a área de visibilidade e o fluxo acumulado. Como os testes mostraram, em alguns casos, as aplicações foram mais do que 200 vezes mais rápidas ao utilizarem a *TiledMatrix*.

Abstract

SILVEIRA, Jaqueline Alvarenga, M.Sc., Universidade Federal de Viçosa, June of 2014. **A library to support the development of applications that process huge matrices in external memory.** Adviser: Marcus Vinícius Alvim Andrade.

This work presents a new library, named *TiledMatrix*, to support the development of applications that process large matrices stored in external memory. The library is based on some strategies similar to cache memory management and its basic purpose is to allow that an application, designed for internal memory processing, can be adapted for external memory. It provides an interface for external memory access that is similar to the traditional method to access a matrix. Moreover, it divides the matrix into two-dimensional blocks, where the blocks are organized in the disk to take advantage of access pattern of applications. In order to reduce the transfer time of a block to and from the disk, *TiledMatrix* uses an algorithm for compression/decompression of data that was implemented in parallel to further improve the efficiency of the library. The *TiledMatrix* was implemented and tested in some applications that require intensive matrix processing such as: computing the transposed matrix and the computation of viewshed, flow accumulation and median. These applications were implemented in two versions: one using *TiledMatrix* and another one using the *Segment* library that is included in GRASS, an open source GIS. In average, they were 7 times faster with *TiledMatrix* and, in some cases, more than 18 times faster. The *TiledMatrix* performance was also compared with the *ArcGIS*. For these tests, it was considered only two applications that are included in ArcGIS: the *viewshed* and the flow accumulation. As tests have shown, in some cases, the applications were more than 200 times faster using the *TiledMatrix*.

Capítulo 1

Introdução

Na era dos grandes volumes de dados, é muito importante que aplicações sejam desenvolvidas para processar dados que excedem a capacidade da memória principal. Na prática esta tarefa não é muito fácil, visto que, arquiteturas consistem de tipos de memória distintos em que cada tipo tem propriedades, velocidades de acesso e modos de acesso distintos. (Ver Figura 1.1).

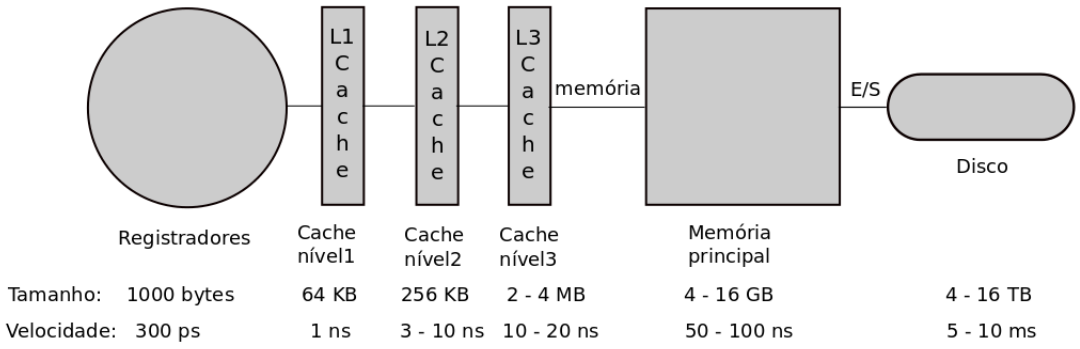

Figura 1.1. Hierarquia de Memória. Fonte: [23]

A memória mais próxima da *central processing unit* CPU é a memória cache que consome aproximadamente um a dois ciclos de *clock* para ser acessada. O tamanho da memória cache pode variar entre poucos *Kilobytes* e diversos *Megabytes*. A memória cache armazena cópias de itens contidos na memória principal e assim, permite a CPU acessar dados ou instruções do programa de maneira muito mais rápida. O tempo de acesso à memória principal é de aproximadamente 50 a 150 nanosegundos e ela pode alcançar alguns *gigabytes*. A memória principal é o local de armazenamento central para códigos e dados de programas. A maior e mais lenta memória é a secundária (memória externa) e o tempo de acesso a ela é mensurado

em aproximadamente 12 a 60 milissegundos. Nos últimos anos, as pesquisas tem focado na otimização sobre os processadores e tem ignorado os efeitos do subsistema de memória. Deste modo, a velocidade do processador tem aumentado de 30% a 50% por ano enquanto que a velocidade do subsistema de memória tem crescido apenas de 7% a 10% por ano [10].

O processamento de matrizes é um requisito central para muitas aplicações como análise de imagens, aplicações de animação gráfica, modelagem numérica de terreno, etc. Em muitos casos, a matriz é muito grande e não pode ser armazenada/processada na memória interna exigindo um processamento externo. Por exemplo, em modelagem de terreno, os recentes avanços tecnológicos na coleta de dados tais como *Light Detection and Ranging* (LIDAR) e *Interferometric Synthetic Aperture Radar* (IFSAR) tem produzido um grande volume de dados e muitos computadores não podem armazenar ou processá-los internamente.

Neste contexto, a memória interna dos computadores pode manter apenas um fração de grandes conjuntos de dados. Durante o processamento, as aplicações precisam acessar a memória externa por diversas vezes. Tais acessos podem ser em torno de 10^6 vezes mais lento do que um acesso a memória principal. Portanto, as operações de entrada e saída são um gargalo na computação, se as aplicações realizam grandes quantidades de operações de entrada e saída, ou a aplicação não é desenvolvida para trabalhar de maneira eficiente na memória secundária. Algumas aplicações confiam no sistema operacional para minimizar o gargalo de operações de entrada e saída.

Sistemas operacionais implementam o mecanismo de memória virtual que estende o espaço de trabalho para aplicações, mapeando um arquivo armazenado na memória externa para endereços na memória virtual. Esta ideia suporta o modelo da memória principal [37]. Com a memória virtual a aplicação não precisa se preocupar onde seus dados estão localizados (na memória principal ou na memória externa). Esta abstração não tem grandes penalidades para padrões de acesso sequencial simples, ou seja, o sistema operacional é capaz de predizer tais padrões e carregar os dados em seguida. No entanto, para padrões mais complexos esta solução não é muito usual e até não indicada, porque o arquivo na memória externa é particionado em blocos e portanto muitas operações de entrada e saída aleatórias são necessárias durante a execução de uma determinada aplicação.

Para evitar esta desvantagem, vários pesquisadores passaram a desenvolver estruturas de dados e algoritmos que explicitamente fazem a comunicação de operações de entrada e saída. Estes são chamados de algoritmos externos ou algoritmos de memória externa que consideram a memória interna com tamanho limitado e um

número de dispositivos de memória secundária. No modelo da memória externa, o desempenho do algoritmo é mensurado ao contar (i) o número de operações de CPU (usando o modelo memória principal), (ii) o número de acessos à memória secundária e (iii) o espaço de trabalho ocupado na memória secundária.

Neste contexto, diversas bibliotecas para processamento de dados na memória externa, tais como *TPIE* (*Transparent Parallel I/O Environment*) [3], *LEDA-SM* [11], *Segment* [20] e *STXXL* [13], têm sido desenvolvidas para tentar reduzir o número de operações de acesso na memória externa, e assim, atuam como uma cache entre a memória principal e o disco. As bibliotecas STXXL e LEDA-SM fornecem estruturas de dados tais como vetores, filas, árvores e funções básicas como busca e ordenação. A biblioteca *Segment*, por outro lado, fornece uma estrutura de dados para armazenar grandes matrizes na memória externa.

Embora essas bibliotecas auxiliem o desenvolvimento de aplicações envolvendo o processamento na memória externa, elas são bastante genéricas e têm como principal objetivo fornecer estruturas de dados como vetores, filas, árvores, etc e funções básicas como busca e ordenação. Ou seja, de acordo com nossos conhecimentos, não existe biblioteca projetada/desenvolvida com o propósito específico de auxiliar o processamento propriamente dito de grandes matrizes armazenadas na memória externa.

1.0.1 Objetivos

O objetivo deste trabalho é propor e implementar uma biblioteca que auxilie o desenvolvimento de aplicações que processam grandes matrizes armazendas na memória externa. Esta biblioteca é baseada em algumas estratégias similares ao gerenciamento de memória cache e seu propósito básico é permitir que uma aplicação, originalmente desenvolvida para processamento em memória interna, possa ser facilmente desenvolvida para memória externa.

Para alcançar o objetivo geral, têm-se alguns objetivos específicos a serem atingidos, como:

1. Uso de algumas estruturas de dados especiais para gerenciar acessos ao disco e, assim, reduzir o número de operações de entrada e saída.

2. Usar um algoritmo para compressão de dados (e adaptá-lo para que ele utilize processamento paralelo) para reduzir o número de operações de entrada e saída durante a transferência de dados para e do disco.

3. Adaptar as estruturas de dados utilizadas pelos algoritmos para permitir que a biblioteca proposta neste trabalho e bibliotecas de propósito semelhante gerenciem os acessos ao disco para as aplicações, a fim de diminuir a latência dos acessos feitos pelas aplicações.

4. Comprovar a eficiência da biblioteca, comparando o tempo de processamento gasto na execução de algumas aplicações que a utilizarão com o tempo gasto na execução de tais aplicações utilizando outra biblioteca de propósito semelhante e avaliar a biblioteca com um sistema de informação geográfica.

1.0.2 Organização da dissertação

No Capítulo 2 são descritos os principais conceitos utilizados nesse trabalho que serviram de base para o projeto. Na seção 2.1, apresentamos as características dos discos rígidos, assim como, salientamos a importância de desenvolver algoritmos que visem reduzir o número de operações de entrada e saída. Na seção 2.2 descrevemos os níveis, o funcionamento e a capacidade de armazenamento da memória cache nos computadores atuais. Na seção 2.3 descrevemos o princípio da localidade de referência definindo os conceitos de localidade temporal e espacial. Na seção 2.4 descrevemos algumas políticas de substiuição de blocos na memória interna. Na seção 2.5 descrevemos as políticas de escrita no disco. Na seção 2.6 apresentamos metodos de acesso ao disco. Na seção 2.7 apresentamos o modelo de representação de terreno utilizado neste projeto. Na seção 2.8 apresentamos as bibliotecas criadas para auxiliar no desenvolvimento de aplicações que requerem processamento em memória externa.

A principal função do Capítulo 3 é descrever a biblioteca desenvolvida neste projeto. A seção 3.1 descreve a maneira como é feito o particionamento da matriz de entrada em blocos pela biblioteca. A seção 3.2 apresenta a estrutura de dados implementada na biblioteca. Na seção 3.3 apresentamos a biblioteca *Segement* que será usada na subseção 4.6.2 para avaliar a biblioteca criada neste projeto.

O Capítulo 4 mostra a análise de eficiência da biblioteca apresentando os resultados das comparações realizadas para atestar seu desempenho. Nas seções 4.1, 4.2, 4.3 e 4.4 descrevemos as aplicações que serão utilizadas nas próximas seções para a avaliação de eficiência da biblioteca deste projeto. Na seção 4.5 apresentamos as fontes de dados utilizadas nos testes realizados neste trabalho. Na seção 4.6 primeiramente, justificamos a escolha do tamanho do bloco para as aplicações. Em seguida, mostramos a eficiência da biblioteca deste projeto.

Capítulo 2

Revisão Bibliográfica

2.1 Algoritmos para memória externa

O disco rígido é um dispositivo magnético ou óptico utilizado como sistema de armazenamento secundário, que consiste em diversos pratos que giram em torno de um eixo. A superfície de cada prato é organizada como coordenadas e os dados são armazenados em trilhas concêntricas nas superfícies de cada prato. Cada trilha é dividida em unidades de tamanho fixo (aproximadamente 512 bytes) denominadas setores[13] e cada trilha possui um número diferentes de setores por serem concentricas.

Os setores são as menores unidades individuais endereçáveis do mecanismo do disco. Operações de entrada e saída sempre transferem dados em múltiplos do tamanho dos setores. Um cilindro descreve o grupo de todas as trilhas localizadas a uma posição específica por meio de todos os pratos. Uma unidade de disco representativa contém acima de 20 pratos e aproximadamente 2000 a 3000 trilhas por polegada e o número de setores por trilha não é fixo[10]. Na Figura 2.1 pode-se observar alguns dos principais componentes do disco.

Neste contexto, a informação é armazenada aplicando-se um campo magnético a partir de uma cabeça de leitura/escrita posicionada muito perto da superfície de um disco mangnético, cuja velocidade de rotação varia de 4.200 a 15.000 rotações por minuto (RPM)[12]. Cada superfície do prato tem uma cabeça do disco associada responsável por escrever ou ler as variações do fluxo magnético sobre a superfície dos pratos [10]. A fim de ler ou gravar uma posição determinada no disco, o controlador de disco move horizontalmente o braço de leitura/escrita de tal modo que a faixa com os dados desejados esteja sob a cabeça. Depois disso, é necessário esperar até que o segmento fique sob a cabeça (latência rotacional). A partir deste momento a

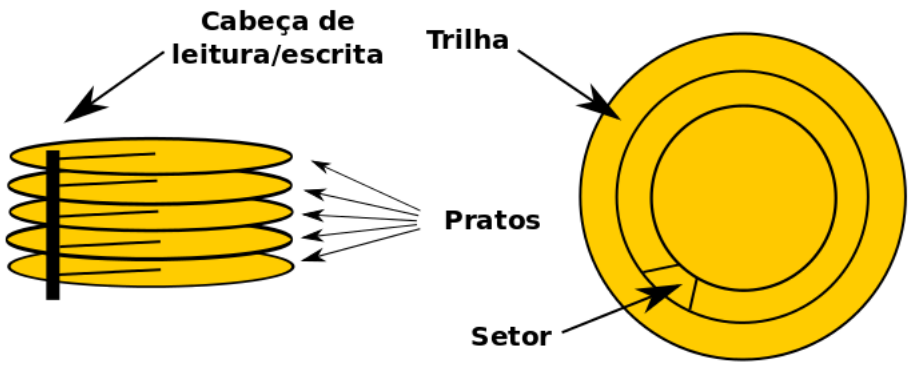

Figura 2.1. Componentes de uma unidade do disco. **Fonte:** [36]

leitura ou escrita é possível.

O processo de leitura/escrita consiste em três estágios:

Latência Rotacional: Uma vez que a cabeça está posicionada na trilha desejada, a cabeça tem que esperar chegar até o setor desejado. Este tempo é denominado *Latência Rotacional* e é inversamente proporcional à velocidade angular (expressa em revoluções por minuto (RPM)) [6]. Em média, a latência rotacional é metade do tempo que o disco precisa para uma revolução, como mostra Deng [14]. O tempo gasto nesta operação é 5,7 ms para um disco com 5400 rpms e 4,2 ms para um disco com 7200 rpms.

Seek Time (**Tempo de busca**): Tempo gasto para posicionar a cabeça de leitura/escrita sobre uma trilha específica. *Seek time* é limitado pela natureza mecânica da tecnologia dos discos rígidos. Uma operação de *seek* é composta de um *speedup*, onde o braço é acelerado, uma *coast* (para longas *seek*), onde o braço move com uma velocidade máxima, *slowdown*, onde a cabeça é posicionada próxima à uma trilha desejada e uma *settle* onde o controlador de disco ajusta a cabeça [11]. O tempo gasto neste movimento é entre 8 a 12 milissegundos e é não linear ao longo do raio.

Taxa de transferência: Mede a quantidade de dados que o disco pode transferir para a memória em um intervalo de tempo. A taxa de transferência de dados geralmente é expressa em *Megabytes* por segundo.

Tempo de transição de cabeça: O braço move todas as cabeças de maneira síncrona. No entanto, apenas uma cabeça está ativa em um determinado

momento. O tempo necessário para transitar entre duas cabeças distintas é mensurado em ms.

Tempo de transição de trilha/cilindro: Uma transição de trilha/cilindro requer mover o braço do controlador do disco para a próxima trilha. Neste caso, é mensurado o tempo que o controlador de disco gasta para levar a cabeça para o próximo cilindro/trilha quando lê ou escreve dados. Este tempo é mensurado em ms.

Tempo de acesso aos dados: Mensura-se o tempo gasto para ler/escrever em um determinado setor. Portanto, o tempo de acesso aos dados é a soma de tempo de busca, tempo de transição de cabeça e latência rotacional.

Note-se, que depois de se encontrar a posição requerida sobre o disco, os dados podem ser transferidos a uma velocidade que está limitada pela largura de banda da interface de conexão da CPU com o disco rígido. Para amortizar a latência, os dados são lidos ou escritos em blocos.

Sendo assim é possível calcular o tempo total de transferência de um bloco como sendo a soma dos tempos de latência, de busca e de transferência. Como se pode perceber, o tempo de busca é o estágio mais demorado do processo de leitura/escrita no disco. A fim de evidenciar o impacto que as operações de entrada e saída têm no tempo de execução final de uma aplicação, considere dois algoritmos apresentados na Figura 2.2 (a) e (b), consistindo em imprimir uma matriz W com n $\times$ n células armazenadas na memória externa.

Algorithm 1 Imprime matriz

```
1: for (i = 1; i <= n; i++) do
2:     for (j = 1; j <= n; j++) do
3:         cout << M[i, j];
4:     end for
5: end for
```

(a)

Algorithm 2 Imprime matriz

```
1: for (j = 1; j <= n; j++) do
2:     for (i = 1; i <= n; i++) do
3:         cout << M[i, j];
4:     end for
5: end for
```

(b)

Figura 2.2. O algoritmo em (a) imprime uma matriz por linhas e o algoritmo em (b) imprime uma matriz por colunas.

A principal diferença entre os dois algoritmos está no modo de acesso à matriz. O algoritmo em (a) acessa a matriz linha por linha, já o algoritmo em (b) coluna por

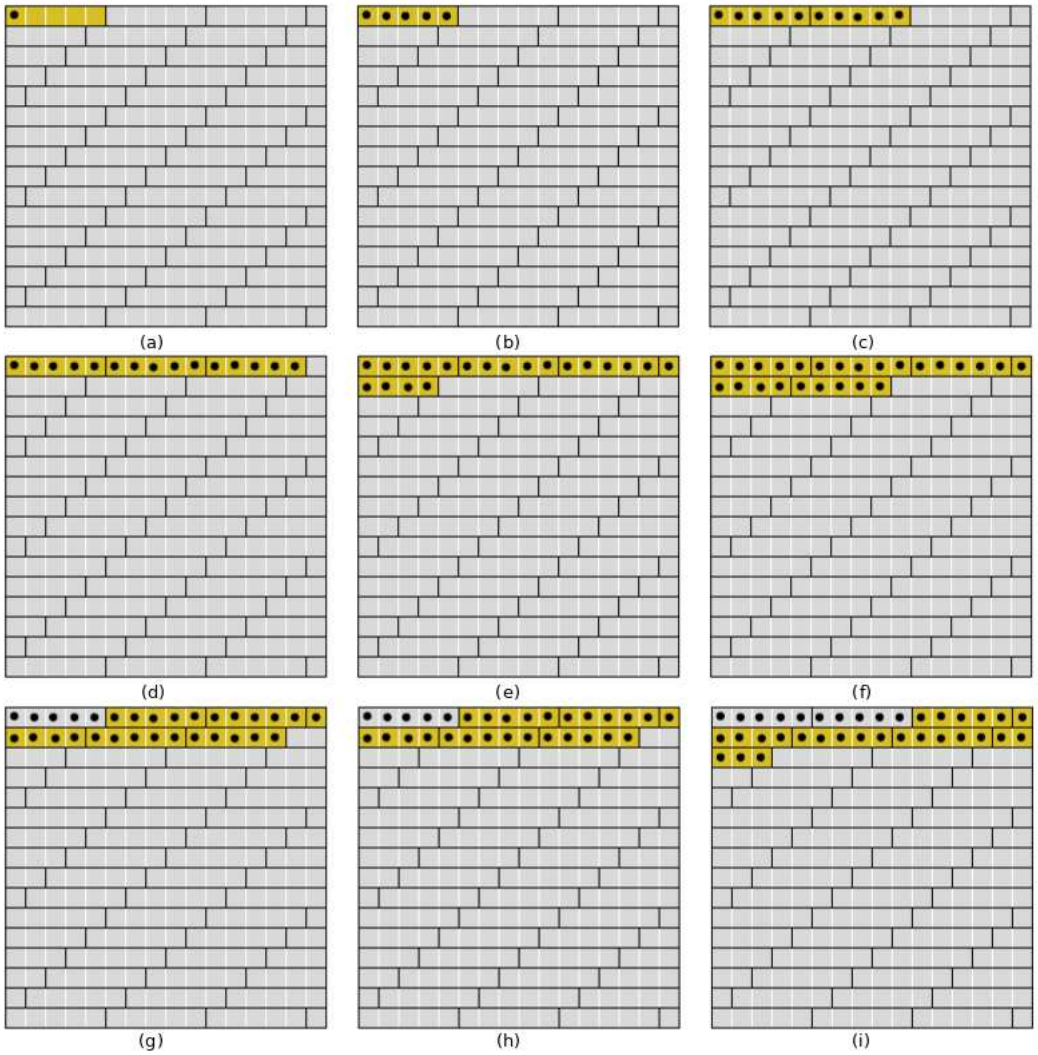

Figura 2.3. Modo de acesso a matriz feito pelo algoritmo mostrado na Figura 2.2 de (a) até (i), sendo que a matriz é dividida em blocos de 5 células. Fonte: [4]

coluna. Com base em instruções de CPU, os dois algoritmos são $\theta(n^2)$. No entanto, ao considerar operações de entrada e saída, caso o tamanho do bloco B fosse menor do que o tamanho de uma linha de W o algoritmo em (a) executaria $\theta(n^2/B)$ operações de entrada e saída e o algoritmo em (b) executaria $\theta(n^2)$ operações de entrada e saída. Isso porque o algoritmo em (a), ao carregar uma célula, carrega também as próximas células a serem processadas pela aplicação, isto é, um bloco de células é carregado e as células do bloco serão as próximas a serem processadas (impressas), como mostra a Figura 2.3. Por outro lado, o algoritmo em (b) carrega um bloco para cada célula que será processada, sendo necessário carregar o mesmo bloco várias vezes na memória, como mostra a Figura 2.4.

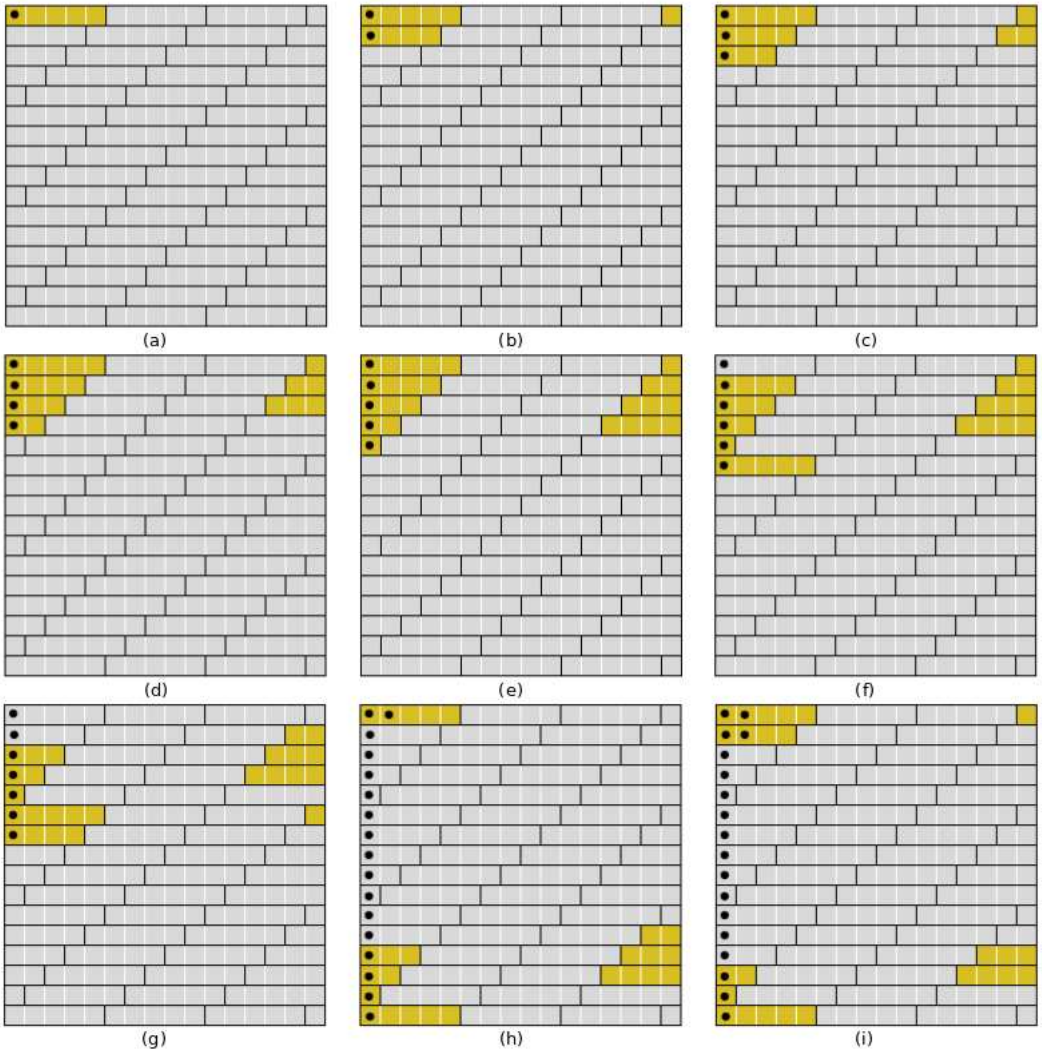

Figura 2.4. Modo de acesso a matriz feito pelo algoritmo mostrado nas Figura 2.2 de (a) até (i), sendo que a matriz é dividida em blocos de 5 células. Fonte: [4]

Para se ter uma ideia da diferença neste processo, suponha uma máquina em que a memória cache pode armazenar 10 mil células e que o tempo para ler um bloco é de 10 milissegundos (9 de latência e 1 para ler). Assim, o tempo para imprimir uma matriz com 50000^2 células por meio do algoritmo em (a) é de aproximadamente 4 minutos e, por meio do algoritmo em (b) é de cerca de 10 meses. Sendo assim, fica evidente a importância de se desenvolver algoritmos que visem reduzir o número de operações de entrada e saída.

2.1.1 O modelo da memória secundária

Aggarwal e Vitter [1] foram os primeiros pesquisadores a introduzirem um modelo da memória secundária. Em seu modelo, uma máquina consiste de uma CPU e uma memória interna rápida de tamanho M. A memória secundária é modelada por um disco tendo $P \geq 1$ cabeças independentes. Uma operação de entrada e saída move B itens do disco para a memória interna ou vice versa. Se $P > 1$, $P * B$ itens podem ser movidos de posições independentes do disco para a memória interna em uma operação de E/S. Do ponto de vista prático, $P > 1$ não existe em disco modernos. Embora cada disco possua diversas cabeças, elas não podem ser movidas independentemente uma da outra. Os principais parâmetros utilizados por este modelo são:

N = número de elementos na instância do problema;

M = número de elementos que podem ser armazenados na memória interna;

B = número de elementos por bloco de disco;

onde $M < N$ e $1 \leq B \leq \frac{M}{2}$. Uma operação de E/S é definida como o processo em que um disco pode transferir um bloco de B itens contíguos. A principal medida de desempenho para algoritmos e estruturas de dados neste modelo é o número de operações de E/S realizadas, que geralmente é expresso em função dos limites para algumas operações fundamentais, tais como:

1. *Scanning* ou varredura de N itens contíguos do disco, cuja complexidade é $\Theta(\frac{N}{B})$;
2. *Sorting* ou ordenação de N itens contíguos do disco, cuja complexidade é $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$;

É importante salientar que normalmente $scan(N) < sort(N) << N$ e, então, é significantemente melhor ter um algoritmo que realiza $sort(N)$ em vez de N operações de E/S. Assim, muitos algoritmos tentam reorganizar os dados na memória externa com o objetivo de diminuir o número de operações de E/S realizadas.

2.2 Memória cache

A memória cache é a memória mais "próxima"(veja a Figura 1.1) da CPU e pode ser acessada, em geral, em um a dois ciclos de clock de CPU. O tamanho da memória

cache pode variar entre poucos kilobytes e vários megabytes sendo que, armazena cópias de alguns dados armazenados na memória principal. A memória cache é dividida em dois grupos, caches de nível 1 e nível 2, comumente chamadas de cache L1 e L2. A cache L1 é um tipo de memória que é incorporada à mesma CPU e é o primeiro local que a CPU tenta acessar. A cache L2 é outra memória que, ao invés de alimentar a CPU, alimenta a cache L1 e, desta maneira, a cache L2 pode ser compreendida como uma cache da cache L1. A principal diferença entre as duas memórias é o tamanho, sendo que a cache L1 é menor do que a cache L2 e portanto, o dado é mais fácil de ser encontrado na cache L1 do que na cache L2.

Neste contexto, a memória cache é configurada de maneira que se o dado está para ser lido da memória principal, primeiramente verifica-se se o dado está contido na cache. Se o dado estiver na cache, é recuperado e usado pela CPU. Contudo, se o dado não está na cache, ele é lido da memória principal e, enquanto está sendo transferido para a CPU, é carregado na cache. Portanto, o objetivo da memória cache é atuar como um *buffer* entre os registradores contidos na CPU e a memória principal.

Em termos de capacidade de armazenamento, a memória cache é muito menor do que a memória principal e, portanto, é necessário estabelecer critérios para o gerenciamento dessa memória. Quando for necessário carregar um novo item e não houver mais espaço para carregá-lo, é necessário remover algum item que está armazenado. Para resolver esta questão, uma política deve ser definida para remover o item. Como será descrito nas próximas seções, há diferentes políticas para realizar esse gerenciamento.

2.2.1 Memória cache nos discos rígidos

Nos discos rígidos (*hard drives* ou *HD?s*) atuais existem as memórias *cache* que auxiliam a melhoria do desempenho dos mesmos. Dessa forma, suponha que a leitura de um arquivo seja realizada de maneira sequencial, em que serão lidos vários setores sequenciais. Na teoria, a maneira mais rápida de ler esses dados é fazer com que a cabeça de leitura leia de uma vez todos os setores de uma trilha, passe para a próxima trilha seguinte, leia todos os seus setores, e assim por diante. Isso permitiria obter o melhor desempenho possível. Entretanto, na prática não é assim que funciona. O sistema pede o primeiro setor do arquivo e somente solicita o próximo após recebê-lo e certificar-se de que não existem erros. Caso não houvesse cache, a cabeça de leitura do HD acabaria tendo que passar várias vezes sobre a mesma trilha, lendo um setor a cada passagem, já que não daria tempo de ler os

setores sequencialmente depois do tempo transcorrido antes de cada novo pedido [10].

Este problema pode ser resolvido utilizando-se uma memória cache pois, a cada passagem, a cabeça de leitura lê todos os setores próximos, independentemente de terem sido solicitados ou não. Após fazer a verificação de rotina, o sistema solicitará o próximo setor que, por já está carregado na cache, será recuperado muito mais rapidamente. Desta forma, a cache de leitura tenta tirar vantagem do fato de que muitos acessos são sequenciais e, portanto, os dados contiguamente próximos muito provavelmente serão requisitados a seguir. Isto permite que o tempo de transferência de dados seja em torno de microsegundos, já que os dados requisitados residem na cache.

2.3 Princípio da localidade de referência

A localidade de referência é a tendência de o processador ao longo de uma execução referenciar instruções e dados da memória principal localizados em endereços próximos. Muitas aplicações de computador são regidas por acessos a dados que apresentam um padrão de localidade, em que existe uma relação entre os dados acessados. Existem dois tipos diferentes de localidade: temporal e espacial [19, 30]. A localidade temporal estabelece que se um item é referenciado, ele tende a ser referenciado novamente em um curto espaço de tempo. A localidade espacial estabelece que se um item é referenciado, outros itens, cujos endereços estejam próximos a ele tendem a ser acessados em breve.

Assim, quando os conceitos de localidade são aplicados na memória principal, consegue-se obter uma melhoria na eficiência da memória, sendo que as células presentes na memória principal são constantemente referenciadas [24]. Caso os conceitos de localidade não sejam aplicados, existe uma grande chance de um bloco que está sendo muito referenciado, ser removido da memória principal; mas este bloco deverá retornar à memória em um curto intervalo de tempo, aumentando assim o número de operações de E/S e reduzindo a eficiência da aplicação.

Contudo, a localidade espacial para um dado tamanho de memória varia significativamente de acordo com o comportamento de cada aplicação. A localidade espacial varia não apenas por meio de diferentes aplicações mas também dentro das diferentes partes da mesma aplicação. Poucas aplicações têm o comportamento da localidade espacial uniforme e executam bem quando usam um tamanho fixo de memória [24]. Além disso, para a maioria das aplicações, a localidade espacial é

não-uniforme. Esta variação é difícil de predizer estatisticamente, sendo que explorar a localidade espacial efetivamente requer mecanismos para se predizer o tempo de execução.

Visto que os blocos mais acessados estão na cache L1 e os menos acessados estão na cache L2 [38], caso um bloco seja requisitado da CPU, primeiramente tal bloco é buscado na cache L1; se a página é encontrada na cache L1, a página então é transferida em alta velocidade para a CPU. Se o bloco não estiver na cache L1, então a cache L2 é acessada. Caso o bloco não esteja em nenhuma das caches, a CPU procura o bloco na memória principal. O bloco poderia não estar na memória principal também e, neste caso, é necessário recuperá-lo transferindo uma cópia do bloco para a cache L1, prevendo um novo acesso baseado no princípio de localidade espacial, como mostrado na Figura 2.5.

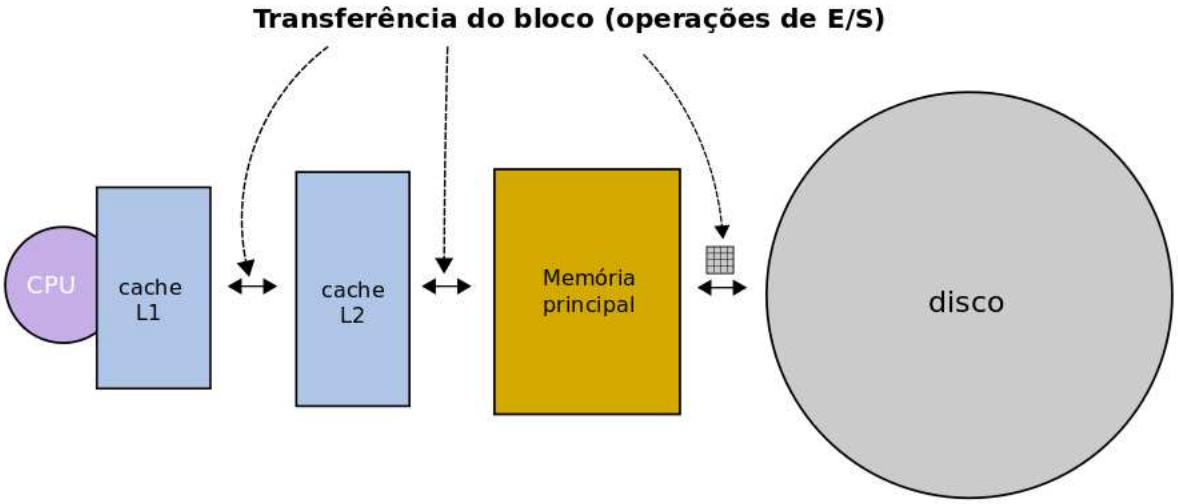

Figura 2.5. Transferência de um bloco do disco para outros níveis da hierarquia de memória. Fonte: [4]

2.4 Políticas de substituição de memória

A estratégia (a política) de como os blocos contidos na memória são substituídos por outros em um determinado momento, influencia consideralvemente o desempenho do processamento das informações em uma CPU. As políticas de substituição mais frequentemente adotadas são:

- *LRU (Least-Recently Used)* [8] remove o bloco menos recentemente usado, baseando-se no princípio da localidade temporal.

- *FIFO (first-in-first-out)* [32] remove o bloco que foi carregado na cache há mais tempo. Este método refere-se apenas ao carregamento, e nem sempre a antiguidade é um bom critério, pois um bloco muito muito frequentemente usado, ainda que antigo, poderia ser escolhido neste caso. De acordo com, Chrobak e Noga [7], a política de substituição na memória LRU é melhor do que a FIFO;

- *Aleatório (random)* [5] caracteriza-se por escolher um bloco ao acaso. Resultados experimentais e simulações mostram que este método funciona razoavelmente, ou seja, o peso de eventuais situações de injustiça, em que um bloco "útil" é escolhido para ser retirado da memória, não é muito grande.

2.5 Políticas de escrita no disco

Quando um bloco contido na memória é modificado, existem duas possibilidades de escrita no disco. A primeira política estabelece que o bloco é escrito apenas quando precisa ser removido da memória. Esta política é chamada *write-back* (escrita retardada). Uma segunda possibilidade é escrever o bloco no disco no momento que ele for modificado. Esta política é nomeada *write-through* (escrita imediata) e, em geral, ela exige um maior número de operações de entrada e saída.

2.6 Métodos de acesso ao disco

Durante o processo de armazenamento de dados no disco deve-se escolher entre os dois métodos de acesso aos dados: sequencial ou aleatório. No acesso sequencial os dados são acessados numa sequência ordenada, um após o outro, na sequência em que foram escritos no disco. Por outro lado, o acesso aleatório permite o acesso a um elemento arbitrário armazenado no disco.

Comparar operações aleatórias e sequenciais é uma maneira de se avaliar a eficiência de uma aplicação em termos do uso do disco. Acessar os dados sequencialmente é mais rápido do que acessar os dados aleatoriamente, visto a maneira em que o disco trabalha. Leituras aleatórias implicam uma taxa de transferência baixa, pois o processo de leitura aleatória envolve um número elevado de operações de *seek*. Isso também ocorre na escrita aleatória [34].

2.7 Representação de superfícies de terreno

Dentre as várias aplicações que se baseiam no processamento de matrizes uma que é bastante importante é a modelagem de terrenos. Um Modelo Digital de Terreno (MDT) é um modelo em que a superfície do terreno S é construída a partir de um conjunto finito de dados digitais (valores de elevação) e, possivelmente, um conjunto de segmentos de linha não-sobrepostos cujas extremidades pertencem a S. Um MDT construído com pontos de S representa uma superfície que interpola, ou aproxima, as medidas de elevação dos pontos de S.

O tipo de modelo depende da forma como os pontos de S são escolhidos no terreno. A escolha de pontos distribuídos sobre uma grade regular caracteriza o modelo digital de elevação raster, também chamado de Matriz de Elevação que muitas vezes é chamado de DEM. Uma forma de representação de terrenos se baseia em armazenar a elevação de pontos regularmente espaçados numa DEM. Ver Figura 2.6.

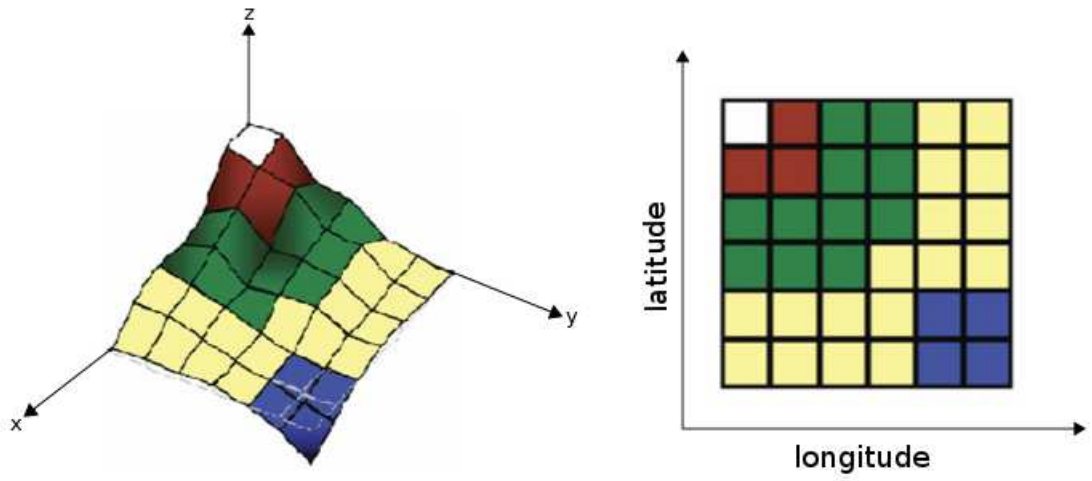

Figura 2.6. Forma de representação de terrenos por meio de uma DEM.

2.8 Bibliotecas para projeto de algoritmos na memória externa

Algumas bibliotecas tais como *TPIE* [3], *LEDA-SM* [11], *STXXL* [13] e *Segment* [20] tem sido criadas para auxiliar no desenvolvimento de aplicações que requerem

processamento em memória externa. A ideia é que estas bibliotecas sejam desenvolvidas para tentar reduzir o número de operações de acesso na memória externa.

A TPIE foi o primeiro grande projeto de software a implementar algoritmos e estruturas de dados para realizar operações de E/S de maneira eficiente. A biblioteca fornece implementações de algoritmos de ordenação, merging e muitas estruturas de dados de busca (B^+-*tree*, *persistent*B^+-tree, $R-tree$, $K-D-B-tree$, $KD-tree$, $Bkd-tree$).

A biblioteca *LEDA-SM* [11] foi desenvolvida como uma extensão para a biblioteca *LEDA* [28] para manipular grandes conjuntos de dados. Ela fornece implementações de alguns algoritmos eficientes de E/S como ordenação e algoritmos em grafos e estruturas de dados na memória externa tais como pilha, fila, *heap* e B^+-*tree*. No entanto, as estruturas de dados e algoritmos não podem manipular mais do que 2^{31} elementos.

STXXL [13] é uma biblioteca *open source* para manipular grandes conjuntos de dados; ela fornece algoritmos para varredura, ordenação e união de dados e diversos contêiners na memória externa. É uma implementação orientada à *framework* da STL (*Standard Template Library*) em C++ para a manipulação de dados na memória externa.

Segment [20] é uma biblioteca incluída em *GRASS* [20] (um Sistema de Informações Geográficas (SIG) *open source*) que foi desenvolvida para manipular grandes matrizes na memória externa. Basicamente, ela subdivide a matriz original em pequenas submatrizes (segmentos) os quais são armazenados na memória externa e alguns destes segmentos são colocados na memória interna e gerenciados como uma memória cache.

É importante salientar que a maioria destas bibliotecas listadas acima, exceto a *Segment*, não permitem operações com matrizes na memória externa. Sendo assim, a proposta deste trabalho é desenvolver uma biblioteca denominada *TiledMatrix* para auxiliar o desenvolvimento de aplicações que processam grandes matrizes armazenadas na memória externa a fim de reduzir o número de operações de entrada e saída do disco e tentar tornar as aplicações mais eficientes.

2.9 Compressão de dados

A compressão de dados consiste em reduzir o número de bits necessários para armazenar ou transmitir dados. A compressão pode ser com perdas ou sem perdas. Dados comprimidos sem perdas, quando são descomprimidos, mantém os mesmos

valores de antes da compressão. Um exemplo é o código de Morse [35].

Todos os algoritmos de compressão de dados consistem de pelo menos de um modelo e de um codificador. Um modelo estima a distribuição de probabilidade (E é mais comum do que Z). O codificador atribui códigos a símbolos, sendo que existem soluções ótimas e eficientes para o problema de codificação. No entanto, o problema de modelagem é não-computável, caracterizando-se como um problema de inteligência artificial [27].

A compressão com perdas, descarta dados "sem importância", por exemplo, detalhes de uma imagem, que não são perceptíveis a olho nu. Um exemplo é o padrão para transmissão de TV a cores. O olho humano é menos sensível a pequenos detalhes entre as cores de brilho igual (como o vermelho e o verde) do que é para brilho (preto e branco). Assim, o sinal da cor é transmitido com menos resolução sobre uma faixa de frequência limitada do que o sinal monocromático. A compressão com perdas consiste em separar dados importantes de dados não-importantes, seguida compressão sem perdas da parte importante descartando o resto.

Capítulo 3

Biblioteca para o processamento de grandes matrizes na memória externa

A *TiledMatrix* foi projetada para auxiliar o desenvolvimento de aplicações que requerem a manipulação de grandes matrizes, que não podem ser armazenadas ou processadas na memória interna. Tal bibloteca permite fácil adaptação de algoritmos originalmente desenvolvidos para memória interna, para processar grandes matrizes armazenadas na memória externa. A idéia é tornar os acessos à matriz transparente para a aplicação, isto é, os acessos a matriz serão gerenciados pela *TiledMatrix*.

3.1 Particionamento da matriz em blocos

A memória principal dos computadores, em geral, pode armazenar apenas uma fração de grandes conjuntos de dados. Assim, para processar grandes matrizes na memória principal, tais matrizes precisam ser particionadas em pequenos blocos, cujo tamanho permite que possam ser carregados na memória. Assim, quando uma célula da matriz precisa ser acessada, o bloco contendo aquela célula é carregado na memória e então qualquer acesso na célula naquele bloco pode ser feito de maneira eficiente.

Contudo, muitas aplicações possuem um padrão de acesso não sequencial e são regidas por acessos a dados que apresentam um padrão de localidade como descrito na seção 2.3. A fim de tirar vantagem do padrão de localidade, a biblioteca

TiledMatrix particiona a matriz de modo que as células cujos endereços estejam próximos na matriz são armazenados de maneira sequencial no disco, como pode ser observado na Figura 3.1.

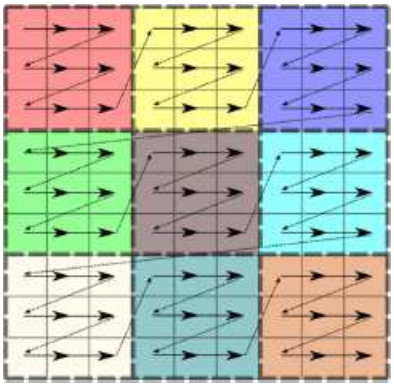

Figura 3.1. Modo de armazenamento das submatrizes no disco.

Desta maneira, é possível reduzir o número de operações de entrada e saída ao reordenar os dados considerando a sequência de acesso [2, 4, 17, 21] ou ao adaptar o algoritmo para usar um padrão de acesso mais eficiente [22, 29].

3.2 Estrutura de dados utilizada pela *TiledMatrix*

A biblioteca *TiledMatrix* é composta por:

- um vetor determinado *MemBlocks* que armazena os blocos na memória principal.
- um vetor nomeado *Dirty* que serve como uma flag para cada bloco na memória interna para indicar se o bloco precisa ser escrito de volta para o disco quando é removido.
- uma matriz nomeada *Position* que armazena a posição onde o bloco da matriz correspondente está armazenado no vetor *MemBlocks* (o valor -1 indica que o bloco correspondente não está carregado na memória). Além disso, se uma célula em um bloco é acessada, a matriz *Position* indica se o bloco está armazenado em *MemBlocks* ou se o bloco precisa ser recarregado para a memória.

3.2.1 Políticas de substituição implementadas na biblioteca *TiledMatrix*

A biblioteca foi desenvolvida para ser geral e oferece muitas opções para o desenvolvimento da aplicação. É possível definir o tamanho do bloco (isto é, o número de linhas e colunas), o tamanho de *MemBlocks*, isto é, o número de blocos que serão armazenados na memória principal e, também, é possível selecionar a política de substituição. Visto que não é possível armazenar todos os blocos na memória interna, a estrutura *MemBlocks* é gerenciada como uma memória cache adotando uma política de substituição. A biblioteca fornece as seguintes políticas: LRU - *Least Recently Used* [8], FIFO - *first in first out* [32] e seleção aletória [5].

A política LRU foi implementada utilizando um *timestamp* atribuído para cada bloco na memória. Assim, quando uma célula contida em um bloco B precisa ser acessada, basta acessar a posição do bloco correspondente na matriz *Position*, se o bloco estiver carregado na memória interna a matriz *Position* informa em qual posição o bloco está carregado. Desta maneira, basta atualizar o *timestamp* de B na posição informada pela matriz *Position* sendo que o *timestamp* daquele bloco passa a ser maior do que os outros e, portanto, quando um bloco precisa ser removido da memória interna, o bloco com o menor *timestamp* é escolhido. Na política FIFO foi utilizado um ponteiro para último elemento inserido na FIFO, assim evita-se ter que deslocar todos os blocos ao inserir um novo bloco na fila.

3.2.2 Política de escrita

A política de escrita implementada na biblioteca *TiledMatrix* é a *Write back* (escrita somente no retorno). Tal política foi implementada considerando uma *flag dirty* atribuída para cada bloco na memória. Assim, quando um bloco precisa ser removido da memória, a *TiledMatrix* verifica se a *flag* que corresponde ao bloco está marcada com *true* ou *false*. Caso esteja marcada com *true*, o bloco é escrito para o disco, por outro lado, caso esteja marcada com *false*, o bloco é apenas descartado, reduzindo operações de escrita.

3.2.3 Algoritmo de compressão LZ4

Adicionalmente, a fim de reduzir o número de operações de E/S durante a transferência de bloco para e do disco, a biblioteca usa um algoritmo de compressão de dados e, por questões e eficiência, foi escolhido o algoritmo LZ4 [25]. A idéia é comprimir o bloco antes de escrevê-lo no disco e descomprimi-lo quando é carregado

para a memória. A principal vantagem deste algoritmo é que é possível alcançar uma compressão de dados rápida e sem perdas de informação a uma velocidade de 400 MB/s por núcleo da CPU. O algoritmo LZ4 possui, ainda, um decodificador bastante rápido, que atinge uma velocidade acima de 1GB/s por núcleo da CPU, tipicamente alcançando a velocidade da *Random Acess Memory* (RAM) em sistemas multi-núcleos[25]. Tais características influenciaram na escolha deste algoritmo para este trabalho e também ao oferecer uma compressão de dados mais rápida do que a taxa que outros algoritmos presentes na literatura oferecem [9]. Como os testes mostraram, esta estrégia torna a biblioteca mais eficiente e, em alguns casos, as aplicações executam duas vezes mais rápido quando a compressão é usada.

No algoritmo LZ4 existe uma função que retorna o tamanho máximo do bloco comprimido no pior caso. Este valor serve para reservar um espaço para cada bloco comprimido no arquivo em disco mesmo que o bloco não alcance esse tamanho. Adicionalmente, durante a compressão de um bloco, o tamanho do bloco comprimido é armazenado em uma matriz. Assim, durante a leitura de um bloco, é lido apenas o tamanho do bloco comprimido e não o espaço total reservado para o bloco. Na Figura 3.2 pode-se observar um exemplo de pior e melhor caso em que no melhor caso um bloco de 100 *bytes* passa a ter um tamanho de 30 *bytes* e, no pior caso, passa a ter um tamanho de 101 *bytes* (o tamanho do bloco tornou-se maior com a compressão porque os dados incluídos naquele bloco não permitem realizar uma boa compressão e devido às informações adicionais que o algoritmo LZ4 deve incluir em cada bloco comprimido). Além disso, há também uma matriz contendo o tamanho do bloco comprimido e o arquivo com os blocos no disco que, independentemente do espaço alocado, ocupam apenas os primeiros bytes da posição designada ao bloco no arquivo em disco.

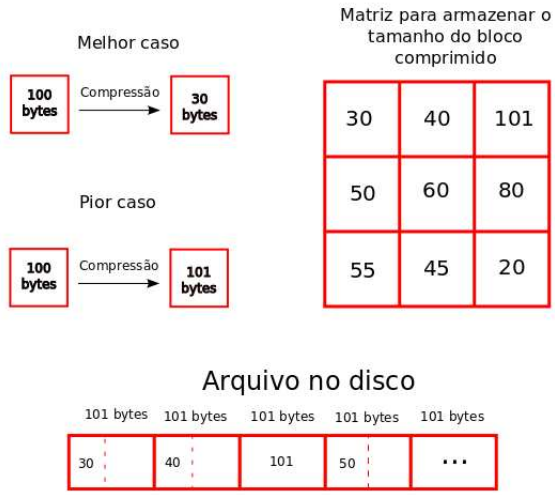

Figura 3.2. Estrutura de dados implementada na *TiledMatrix* para auxiliar o algoritmo LZ4.

Na Figura 3.3 pode-se observar a estrutura de dados usada pela biblioteca *TiledMatrix*: neste exemplo, a matriz no disco é particionada em 16 blocos quadrados e o vetor *MemBlocks* é capaz de armazenar 5 blocos; a política de substituição usada foi a LRU. Suponha que *MemBlocks* esteja cheia. Se uma célula no bloco $(1, 2)$ é acessada, este bloco precisa ser carregado e, neste caso, um bloco armazenado em *MemBlocks* é expulso para abrir espaço para o novo bloco. De acordo com o vetor *timestamp*, o bloco $(2, 3)$ tem o menor *timestamp*. Assim, o bloco será escrito para o disco e a célula correspondente na matriz *Position* é atualizada como -1. Em seguida, o bloco $(1, 2)$ é carregado, descomprimido, armazenado no *slot 4* de *MemBlocks* e a matriz *Position* é atualizada para indicar a posição do bloco em *MemBlocks* como mostra na Figura 3.4.

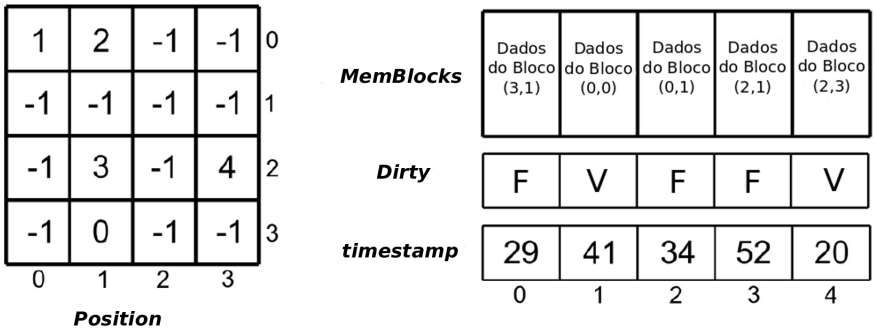

Figura 3.3. Exemplo de estrutura de dados que será usada pela biblioteca proposta neste projeto.

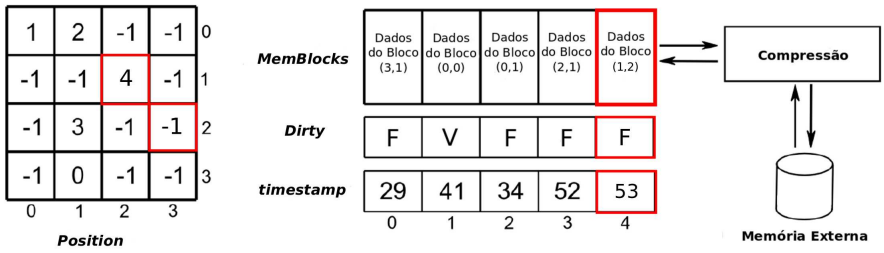

Figura 3.4. Estrutura de dados após a transferência de um bloco do disco para *MemBlocks*.

3.2.3.1 Algoritmo LZ4 paralelo

A fim de melhorar ainda mais a eficiência da biblioteca, o algoritmo LZ4 foi implementado em Paralelo. A ideia do LZ4 paralelo consiste um dividir um bloco pelo número de núcleos da CPU, gerando subblocos, e então comprimir e descomprimir esses subblocos em paralelo. Na Figura 3.5 pode-se observar a estrutura de dados utilizada na biblioteca *TiledMatrix* para implementar o algoritmo LZ4 Paralelo. Primeiramente a *TiledMatrix* particiona um bloco pelo número de núcleos disponíveis na CPU e chama a função *tamanhoMaximoBlocoComprimido()* contida no algoritmo LZ4. Por meio do valor retornado por esta função, reserva-se um espaço em disco para cada subbloco mesmo que o tamanho do subbloco, comprimido seja menor do que o tamanho reservado. Em seguida, a biblioteca comprime os subblocos em paralelo.

O tamanho dos subblocos comprimidos é guardado em um *array bytesCompressaoBloco*. Como não se sabe o tamanho dos subblocos comprimidos, visto que os blocos ainda serão comprimidos, utiliza-se um *array blocosComprimidos* para armazenar os subblocos comprimidos, cujo tamanho é o número de núcleos da CPU disponíveis no computador multiplicado pelo tamanho máximo que um subbloco comprimido pode alcançar. Na Figura 3.6 (a) pode-se observar como os blocos ficam posicionados no *array blocosComprimidos*. Note que o espaço à direita do tracejado é um lixo que deve ser descartado antes de realizar a operação de escrita dos subblocos no disco. Para eliminar o lixo no final de cada posição do *array blocosComprimidos*, por meio do *array bytesCompressaoBloco*, realiza-se uma cópia apenas dos subblocos comprimidos para outro *array auxBlocosCompressao* de modo que, entre cada subbloco armazenado no *array auxBlocosCompressao*, não exista lixo, como mostra a Figura 3.6 (b). Assim, para escrever os subblocos no disco, a *TiledMatrix* soma o tamanho de cada subbloco comprimido e escreve no disco apenas a quantidade de bytes resultante dessa soma, que são os subblocos comprimidos. O processo de

leitura dos subblocos é semelhante: soma-se o tamanho de cada subbloco em bytes e lê-se somente essa quantidade de bytes do disco, que são os subblocos comprimidos. Os subblocos são armazenados no *array blocosComprimidos* e descomprimidos em paralelo e unidos em uma matriz que armazena os blocos carregados na memória, de modo que todos os subblocos formem um único bloco na memória.

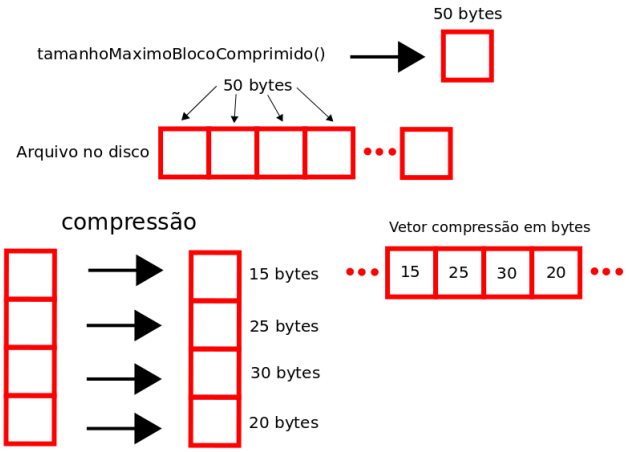

Figura 3.5. Estrutura de dados utilizada na *TiledMatrix* para implementar o algoritmo LZ4 Paralelo, considerando que o computador tenha 4 núcleos da CPU disponíveis.

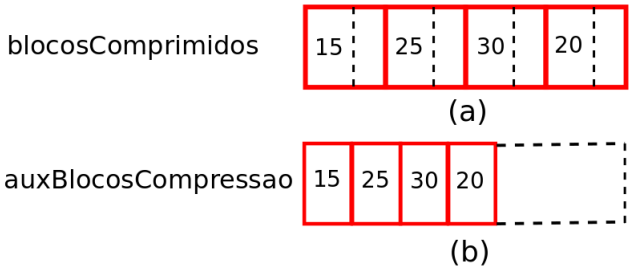

Figura 3.6. Processo de remoção de lixo e união dos subblocos em um *array*, considerando que o computador tenha 4 núcleos da CPU disponíveis.

Para casos em que a divisão da matriz não seja inteira (ver Figura 3.7), a biblioteca completa os blocos na fronteira da matriz com valores nulos - no caso de processamento de terrenos estas células contém valores *NODATA*[1].

[1] Valores *NODATA* representam a ausência de dados.

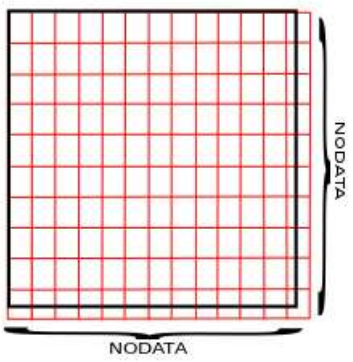

Figura 3.7. Bloco na fronteira de uma matriz, em que a divisão não é inteira, preenchido com valores *NODATA*.

3.3 *TiledMatrix* versus *Segment*

É importante mencionar que, embora a biblioteca *Segment*, incluída em GRASS [20], seja baseada em conceitos similares da *TiledMatrix*, na implementação da *TiledMatrix* foram usadas algumas estratégias diferentes que a tornam mais efficiente do que a *Segment*.

Primeiramente, *Segment* não usa compressão da dados para reduzir operações de E/S do disco. Além disso, a *Segment* permite apenas a política de substituição LRU. E, além disso, esta política foi implementada baseada de forma diferente nas duas bibliotecas, principalmente considerando o método para marcação de blocos. Na *Segment*, os blocos são marcados com um *timestamp* (um valor numérico) que é atualizado da seguinte forma: quando um novo bloco é acessado, seu *timestamp* é zerado e todos os outros *timestamp* dos blocos (na memória) são incrementados em uma unidade; assim, se um bloco precisa ser expulso, o bloco com maior *timestamp* é selecionado. Como pode-se observar, se existem n blocos na memória, cada passo de marcação do bloco tem custo $O(n)$. Por outro lado, na *TiledMatrix*, quando um bloco é acessado, apenas o *timestamp* daquele bloco é incrementado; assim, o bloco que tem o menor *timestamp* é expulso. Neste caso, o processo de marcação do bloco requer tempo constante.

Outra importante diferença entre *TiledMatrix* e *Segment* é o processo para determinar se um bloco já está carregado na memória principal e, se sim, onde ele está armazenado. Na *TiledMatrix*, quando uma célula em um bloco precisa ser acessada, é fácil (e rápido) determinar se um bloco está carregado na memória interna (e onde o bloco está armazenado) visto que basta acessar a posição do bloco correspondente na matriz *Position*. Assim, esta operação pode ser executada em tempo constante. Mas, na *Segment*, é necessário varrer o vetor que armazena os

blocos na memória interna para verificar (e acessar) o bloco; para tentar reduzir este *overhead*, a biblioteca mantém a posição do último bloco carregado. Novamente, no pior caso, esta operação requer tempo linear.

Portanto, na *TiledMatrix*, um acesso a um bloco carregado na memória, em todos os casos, tem custo $O(1)$, enquanto na *Segment* tem custo $O(n)$. Note que esta melhoria da eficiência na operação de acesso ao bloco torna a *TiledMatrix* mais eficiente do que a *Segment*, visto que o acesso aos blocos é uma operação básica que é executada várias vezes.

Capítulo 4

Avaliação de Eficiência

A fim de avaliar a eficiência da *TiledMatrix*, alguns testes experimentais foram realizados para compará-la com a biblioteca *Segment*. Os testes foram baseados em três aplicações que requerem intensivo processamento de matrizes: calcular a transposta da matriz, calcular o viewshed (mapa de visibilidade) e o fluxo acumulado em terrenos representados por uma matriz de elevação. Para cada aplicação foi selecionado um algoritmo que foi adaptado para processamento em memória externa, isto é, para processar a matriz armazenada em disco. Assim, para cada algoritmo foi gerada duas versões: uma usando *TiledMatrix* para gerenciar o acesso a matriz armazenada na memória externa e outra usando a *Segment*.

Os algoritmos usados nos testes são descritos abaixo. Tais algoritmos foram implementados em C++, compilados usando gcc 4.5.2 e os testes foram executados em um computador *Core i5*, memória RAM de 4GB, 1TB de disco *Sata* executando no sistema operacional *Linux Ubuntu* 12.10 64*bits*. Em todas as aplicações as bibliotecas foram configuradas para usar, no máximo 3GB de RAM. Os resultados destes testes são apresentados na seção 4.6.2.

4.1 Transposição da matriz

O algoritmo para transposição da matriz é baseado no método trivial onde a matriz de entrada é lida sequencialmente (linha por linha) do disco e cada célula *(i,j)* é armazenada na posição *(j,i)* em uma matriz temporária M manipulada pela biblioteca *TiledMatrix* ou pela biblioteca *Segment*. Então, a matriz transposta M é escrita para o disco.

4.2 Viewshed

Dado um terreno representado por uma matriz de elevação, seja O um ponto no espaço (o *observador*) de onde se deseja visualizar (ou comunicar) com outros pontos do terreno (os alvos). Tanto o observador como o alvo podem estar a uma dada altura acima do terreno. Usualmente, assume-se que o observador tem um alcance de visão ρ, o *raio de interesse*, que significa que o observador pode ver pontos a uma dada distância ρ. Assim, um alvo T é visível a partir de O se e apenas se a distância de T de O é, no máximo ρ, e a linha de visão de O a T é sempre estritamente acima do terreno, isto é, não existe um ponto nesta linha cuja altura é menor ou igual do que a elevação do ponto do terreno correspondente. Ver Figura 4.1.

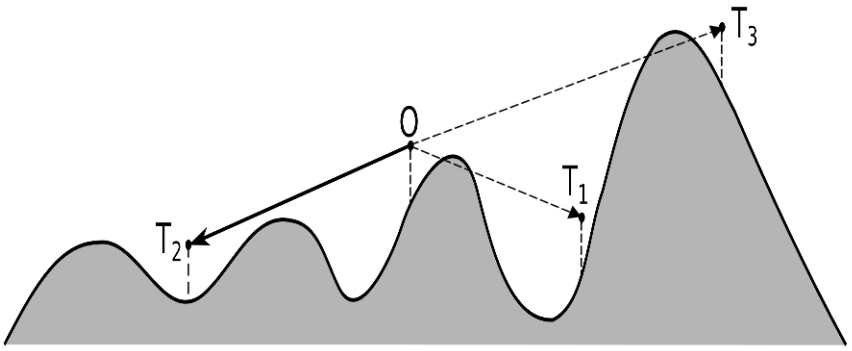

Figura 4.1. Visibilidade do alvo: T_2 é visível, mas T_1 e T_3 não são.

O *viewshed* de O corresponde a todos os pontos do terreno que podem ser vistos por O; formalmente,

$$viewshed(O) = \{p \in T \mid \text{um alvo em } p \text{ é visível de } O\} \tag{4.1}$$

onde o raio de interesse ρ é deixado implícito. Visto que o terreno é representado por MDEs, o *viewshed* pode ser representado por uma matriz $(2\rho + 1) \times (2\rho + 1)$ quadrada de bits onde 1 indica que o ponto correspondente é visível e O não é. Por definição, o observador está no centro desta matriz.

Então, a visibilidade de um alvo sobre uma célula c_t pode ser determinada ao verificar a inclinação da linha conectando o observador e o alvo e a elevação da célula sobre o segmento rasterizado. Mais precisamente, suponha que o segmento é composto pelas células $c_0, c_1, \cdots, c_t$ onde c_0 e c_t são respectivamente as células correspondentes ao observador e posições do alvo. Seja α a inclinação da linha conectando o observador a célula c_i, isto é,

$$\alpha_i = \frac{\zeta(c_i) - (\zeta(c_0) + h_o)}{dist(c_0, c_i)} \tag{4.2}$$

onde $\zeta(c_0)$ e $\zeta(c_i)$ são, respectivamente, a elevação das células c_0 e c_i e $dist(c_0, c_i)$ é a "distância" (o número de células) entre estas duas células e h_o e h_t é a altura do observador e do alvo, respectivamente. Assim, o *target* sobre c_t é visível se e apenas se a inclinação $\frac{\zeta(c_t)+h_t-(\zeta(c_0)+h_o)}{dist(c_0,c_p)}$ é maior do que α, para todo $0 < i < t$. Se sim, a célula correspondente na matriz *viewshed* é marcada como 1; caso contrário, é marcada como 0.

Note que o cálculo do *viewshed* demanda um processamento intensivo da matriz. Para terrenos cuja matriz não cabe na memória interna, é necessário um grande número de acessos à memória externa, visto que a matriz é acessada não-sequencialmente.

Existem muitos métodos para o cálculo do *viewshed* [18, 21] e, em particular, em [18] é proposto um método de tempo linear em que a visibilidade é calculada ao longo de um raio (segmento) conectando o observador O ao centro da uma célula na borda de um quadrado de lado $2R + 1$ centrado em O. Este raio será rotacionando em sentido anti-horário em torno do observador seguindo as células na borda (ver Figura 4.2).

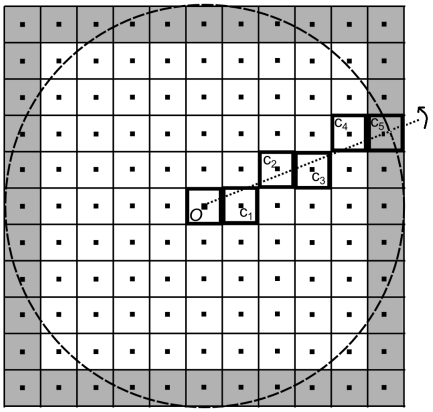

Figura 4.2. Percorrendo o terreno.

Este algoritmo pode ser facilmente adaptado para processamento em memória externa utilizando a biblioteca *TiledMatrix* ou *Segment*. A idéia básica é, que quando uma célula da matriz precisa ser acessada, este acesso é gerenciado por uma destas bibliotecas descritas anteriormente.

4.3 Fluxo acumulado

Outra aplicação importante que requer o processamento intensivo da matriz é o cálculo da rede de drenagem em terrenos. Informalmente, a rede de drenagem de um terreno delineia o caminho que a água deve seguir ao longo de um terreno (a direção de fluxo) e a quantidade de água que flui em cada célula do terreno (o fluxo acumuldo). Em outras palavras, a direção de fluxo consiste em atribuir direções para células do terreno tal que estas direções ditam o caminho da água no terreno [33]. A direção na célula c pode ser definida como a direção da célula vizinha de menor elevação dentre as que têm elevação menor do que c. Adicionalmente, o fluxo acumulado de uma célula c é o número de outras células cujo fluxo alcança c (informalmente, o fluxo acumulado significa a quantidade de água que alcança cada célula) supondo que cada célula recebe uma gota de chuva. Na Figura 4.3 pode-se observar um exemplo de uma matriz de direção de fluxo e a matriz de fluxo acumulado calculada a partir da matriz de direção de fluxo.

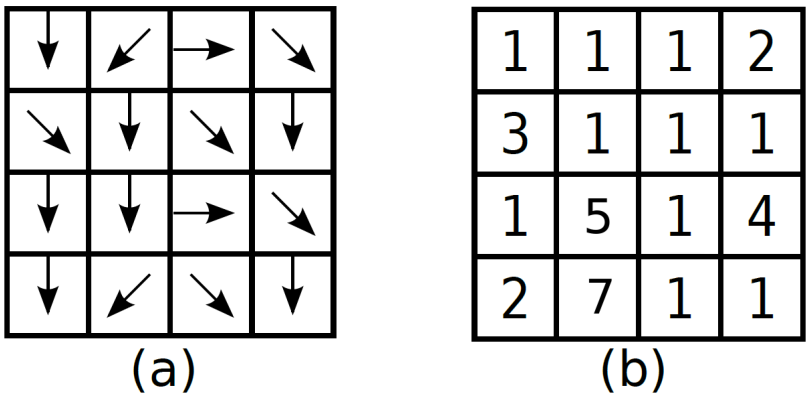

Figura 4.3. Cálculo da rede de drenagem: (a) Direção de fluxo; (b) Fluxo acumulado.

É importante notar que o cálculo da direção de fluxo precisa incluir alguns passos para tratar depressões e áreas planas sobre os terrenos e, em geral, estes passos não são muito simples e usam algumas estruturas de dados complexas. Assim, a adaptação de algoritmos de direção de fluxo para memória externa não é muito fácil, pois exige algumas modificações importantes na implementação do algoritmo. Visto que o objetivo dos testes é avaliar a eficiência da biblioteca *TiledMatrix*, então decidimos adaptar apenas a parte relacionada à obtenção do fluxo acumulado a partir da direção de fluxo.

Assim, dada uma matriz de direção de fluxo obtida pelo algoritmo *RWFlood* [26], o fluxo acumulado pode ser obtido realizando uma ordenação to-

pológica num grafo criado a partir das direções de fluxo da seguinte forma: cada célula é um vértice e existe uma aresta direcionada conectando uma célula c_1 a uma célula c_2 se e apenas se c_1 flui para c_2. Inicialmente, todos os vértices no grafo tem 1 unidade de fluxo. Então, em cada passo, uma célula c com grau 0 é marcada como visitada e seu fluxo é adicionado ao fluxo de $next(c)$, que é a célula seguinte c no grafo. Depois de processar c, a aresta conectando c a $next(c)$ é removida (ou seja, o grau de $next(c)$ é decrementado). Para os testes, nós adaptamos o algoritmo apresentado por Haverkort e Janssen [22] para usar a *TiledMatrix* e *Segment*.

4.4 Filtro da mediana

O filtro da mediana é um método não-linear usado para reduzir ruídos em imagem. Este método percorre a matriz pixel por pixel, substituindo cada valor pelo valor mediano dos pixels vizinhos [39] como mostra na Figura 4.4.

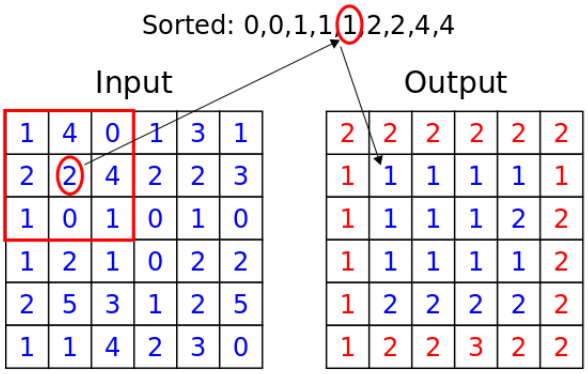

Figura 4.4. Calculando o filtro da mediana.

4.5 Fontes de dados

Os testes realizados com a *TiledMatrix* e a *Segment* utilizaram matrizes representando terrenos obtidas a partir da página Nasa [31] com resolução horizontal de 30 metros. Os dados utilizados nos testes foram extraídos das Regiões 2 e 3 (como mostra na Figura 4.5) pois estas regiões estão no centro do continente e portanto os terrenos não incluem parte do oceano que normalmente são definidas como *NO-DATA*. A Região 2 caracteriza-se por ser mais montanhosa e a Região 3 por ser mais suave.

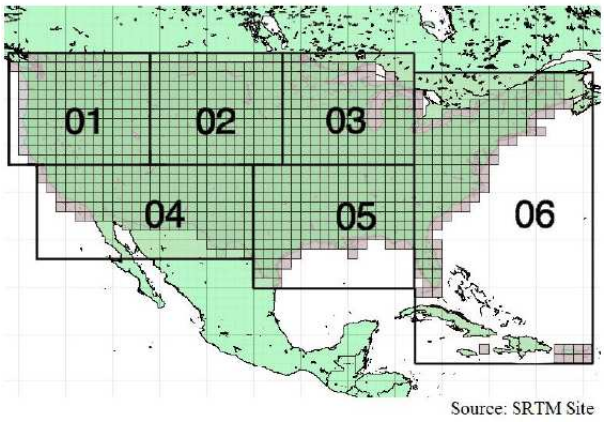

Figura 4.5. Regiões SRTM para EUA.

Nos testes para avaliar as políticas de substituição (ver seção 4.6.4) geraram-se terrenos fictícios a fim de mostrar o desempenho das aplicações em terrenos montanhosos e terrenos mais suaves, visto que o tempo de processamento para calcular o viewshed e o fluxo acumulado depende da topografia do terreno. Os terrenos aleatórios foram gerados considerando um tamanho de bloco de 1000×1000 e foi utilizada a função *rand()* do *C++* para gerar números aleatórios, o quais foram atribuídos para cada célula do terreno. Os terrenos suaves foram gerados considerando um tamanho de bloco 1000×1000, sendo que as elevações dos valores atribuidos às células foram geradas de modo a ter pouca variação de uma elevação para outra.

4.6 Experimentos e avaliação de eficiência

4.6.1 Cálculo do tamanho do bloco da *TiledMatrix*

A eficiência de um algoritmo que utiliza a *TiledMatrix* para processar uma matriz armazenada na memória externa é afetada por alguns elementos como: o tamanho da memória interna, o tamanho do bloco bloco adotado pela *TiledMatrix* e a forma da matriz (número de células nas linhas e colunas). Idealmente, o melhor caso para o algoritmo seria ter qualquer bloco já carregado na memória interna quando uma de suas células for acessada e um bloco deveria ser removido apenas quando o mesmo não fosse mais necessário. Mas, usualmente, não é fácil alcançar este caso quando o algoritmo tem que processar conjuntos de dados que são muito grandes para caber na memória interna.

Além disso, quando a biblioteca precisa armazenar (carregar) um bloco para

e do disco, é necessário executar uma operação de *seek* para alcançar a posição do bloco no disco e, então, transferi-lo para e do disco. Mas, note que, o custo da operação *seek* pode ser amortizado usando um bloco de tamanho grande, isto é, quanto maior o tamanho de bloco, menor a significancia do custo da operação de *seek* para a operação de transferência de dados. Por outro lado, tamanhos de blocos muito grandes usualmente aumentam a penalidade da falta do bloco, visto que o tempo necessário para armazenar (carregar) um grande bloco para e da memória é alta.

Assim, para alcançar uma boa eficiência ao se usar a *TiledMatrix*, o tamanho do bloco precisa ser definido a fim de tentar reduzir o número de vezes que o mesmo bloco é removido e recarregado enquanto alcança uma alta taxa de transferência de disco. E, é importante notar que esta definição depende da aplicação, porque ela precisa ser feita, principalmente, baseada nos padrões de acesso usados pelas aplicações. Para ilustrar como o tamanho do bloco pode ser definido baseado na análise da aplicação, vamos descrever a definição do tamanho do bloco para três aplicações usadas nos testes.

No algoritmo de transposição da matriz, a matriz é acessada sequencialmente linha por linha. Assim, o tamanho do bloco deve ser definido de modo que seja possível armazenar, na memória interna, os blocos necessários para cobrir uma linha da matriz. Como a maior matriz usada nos testes tem 60.000 células por linha e como o tamanho da memória disponível é 3GB, definimos cada bloco com 1000 x 1000 células totalizando 4.000.0000 de bytes por bloco, pois uma célula requer 4 bytes.

No cálculo do *viewshed*, o tamanho do bloco foi definido baseado nos padrões de acesso do algoritmo de Franklin e Ray. Mais precisamente, como foi mostrado em [16], se a mémoria interna for capaz de armazenar $2\left(\frac{\rho}{\omega}+2\right)$ blocos onde ρ é o raio de interesse e o tamanho do bloco é $\omega \times \omega$ células então quando um bloco é removido pela política *LRU*, ele não será mais necessário. Assim, como o tamanho da memória é 3GB, nós definimos blocos com 1000×1000 células para permitir que o número requerido de blocos possa ser armazenado na memória interna.

Finalmente, no problema do fluxo acumulado, a definição do tamanho do bloco é um pouco mais complexa visto que o padrão de acesso à matriz não é muito bem definido por causa da ordenação topológica, que é baseada na direção de fluxo, que pode ser muito diferente para cada terreno. Note que o algoritmo usado para a ordenação topológica acessa a matriz de duas maneiras: na maioria das vezes, a matriz é acessada linha por linha e, ocasionalmente, a sequência de acesso segue uma cadeia de células cujo grau de entrada é 1. Na Figura 4.6 pode-se observar

estes acessos: inicialmente, a matriz é processada linha por linha (e todos blocos que interceptam a primeira linha são carregado na memória) e quando a célula c é processada, visto que seu grau é 0, o acesso sequencial precisa seguir sua direção de fluxo visto que o grau do seu vizinho também torna 0. Este processo é repetido para todas as células indicadas usando uma com a cor preta (mais escura) que requer que 8 blocos sejam acessados (carregados). Depois de processar esta cadeia, o acesso sequencial linha por linha é restaurado na célula d.

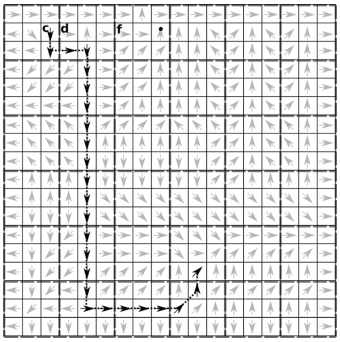

Figura 4.6. Calculando o fluxo acumulado em uma matriz de 18×18 células (dividida em blocos com 3×3 células). A cadeia de células destacadas com uma linha pontilhada é processada quando a célula c é visitada por um acesso linha por linha.

Note que quando uma cadeia de células com grau 0 é seguida, a sequência de acesso poderia requerer vários blocos carregados e este processo poderia forçar a remoção de um bloco que poderia ser acessado novamente. Por exemplo, na Figura 4.6, o bloco contendo a célula d foi carregado durante o processamento da primeira linha e será acessado novamente depois de processar a sequência refererida. Assim, se o processamento da sequência força que o bloco contendo a célula d seja removido, este bloco será recarregado para processar d.

Portanto, no problema do fluxo acumulado, nós escolhemos um pequeno tamanho de bloco a fim de aumentar a quantidade de blocos que podem ser armazenados na memória interna. De acordo com alguns experimentos empíricos, um bom tamanho de bloco para este problema é aquele que contem 250×250 células, visto que blocos maiores evitam operações de E/S e blocos menores tornam a taxa de transferência do disco baixa.

4.6.2 Avaliação de desempenho

Como dito anteriormente, a *TiledMatrix* foi avaliada comparando o tempo de execução das três aplicações usando duas bibliotecas: *TiledMatrix* e *Segment*. Além disso,

para avaliar o impacto da compressão do bloco para o desempenho da *TiledMatrix*, nós usamos duas versões da *TiledMatrix*: uma incluindo compressão e outra não incluindo. Em todos os casos, as bibliotecas *TiledMatrix* e *Segment* foram configuradas para utilizar a política LRU como a política de substituição e o tamanho do bloco foi definido como segue: para a transposição da matriz e *viewshed*, foi usado um bloco quadrado com células 1000 x 1000 e no fluxo acumulado, blocos de 250 x 250 células.

As matrizes usadas nos testes foram obtidas do site STRM [31] e elas representam terrenos com resolução de $30m \times 30m$ correspondendo a regiões dos Estados Unidos da América. Para a transposição da matriz, nós usamos diferentes tamanhos de matrizes de uma mesma região e, visto que o tempo de processamento para calcular o *viewshed* e o fluxo acumulado depende da topografia do terreno, nestes dois casos nós usamos diferentes tamanhos do terreno de duas diferentes regiões: uma correspondente a um terreno montanhoso e a outra, a um terreno mais suave. Mais precisamente, no fluxo acumulado, as entradas são as matrizes de direção de fluxo obtidas pelo algoritmo *RWFlood* [26] usando-se estes conjuntos de dados de terrenos. E, no cálculo do *viewshed*, foi usado o raio de interesse tão grande quanto possível, isto é, um raio de interesse cobrindo todo o terreno e o observador foi posicionado 50 metros acima do terreno.

É importante notar que, embora uma matriz de entrada possa ser armazenada na memória interna, talvez o algoritmo de processamento não possa ser executado internamente visto que ele precisa usar algumas estruturas de dados adicionais. Assim, considerando que as matrizes do terreno são representadas usando $2bytes$ por célula, um terreno com N células requer $2Nbytes$. Nos testes, o algoritmo da transposição da matriz usa duas matrizes: a matriz do terreno de (entrada) com $2Nbytes$ e a matriz de *viewshed* (saída) com $Nbytes$ (um *byte* por célula), assim requerendo $3Nbytes$ no total. E, finalmente, o algoritmo de fluxo acumulado usa 3 matrizes: a matriz de direção de fluxo ($Nbytes$), o grau da matriz ($Nbytes$) e a matriz de fluxo acumulado ($4Nbytes$) requerindo $6Nbytes$ no total.

Os resultados dos testes são apresentados nas Tabelas 4.1, 4.2 e 4.3 e também, nos gráficos nas Figuras 4.7, 4.8 e 4.9. De acordo com os resultados, todas as aplicações executam mais rapidamente usando *TiledMatrix* do que *Segment* e, em geral, foram 7 vezes mais rápidas com a *TiledMatrix*. Mas, note que, em alguns casos, a *TiledMatrix* pode tornar a aplicação 18 vezes mais rápido (por exemplo, fluxo acumulado em terrenos de 3.4 GB). Este aumento na eficiência pode ser explicado por duas razões: a transferência de dados é reduzida por causa da compressão do bloco e o uso da estratégias mais eficientes para gerenciamento do bloco (ver

seção 3.3).

Tabela 4.1. Tempo (em segundos) para transpor uma matriz usando as bibliotecas *TiledMatrix* (com e sem compressão) e *Segment*.

Transposta da matriz				
Tamanho da Matriz		**TiledMatrix**		
linha × coluna	GB	sem comp.	com comp.	**Segment**
5000^2	0.1	1.34	1.08	2.97
10000^2	0.4	5.47	4.09	11.59
20000^2	1.5	25.49	18.49	62.97
30000^2	3.4	92.18	79.28	251.51
40000^2	6.1	234.32	178.67	486.35
50000^2	9.5	365.38	249.72	800.42
60000^2	13.7	507.63	335.39	1246.18

Tabela 4.2. Tempo (em segundos) para calcular o *viewshed* em um terreno utilizando as bibliotecas *TiledMatrix* (com e sem compressão) e *Segment*.

Viewshed					
Terrenos			**TiledMatrix**		
Região	linha × coluna	GB	sem comp.	com comp.	**Segment**
2	5000^2	0.1	3.69	3.15	6.33
	10000^2	0.4	14.24	12.35	26.96
	20000^2	1.5	58.50	49.49	115.66
	30000^2	3.4	182.76	147.16	344.24
	40000^2	6.1	365.82	278.45	648.47
	50000^2	9.5	591.41	426.39	1036.57
	60000^2	13.7	906.19	609.76	1568.51
3	5000^2	0.1	3.90	3.19	7.03
	10000^2	0.4	15.08	12.65	28.76
	20000^2	1.5	61.21	51.46	132.81
	30000^2	3.4	211.85	151.56	378.65
	40000^2	6.1	352.41	270.04	631.750
	50000^2	9.5	643.91	434.58	1106.22
	60000^2	13.7	872.69	575.72	1552.40

Tabela 4.3. Tempo (em segundos) para calcular o fluxo acumulado em um terreno utilizando as bibliotecas *TiledMatrix* (com e sem compressão) e *Segment*.

Fluxo Acumulado					
Terrenos			**TiledMatrix**		
Região	linha × coluna	GB	sem comp.	com comp.	**Segment**
2	5000^2	0.1	7.10	3.86	65.25
	10000^2	0.4	26.79	15.29	277.75
	20000^2	1.5	119.7	65.26	1038.15
	30000^2	3.4	612.11	242.78	4336.54
	40000^2	6.1	1685.65	938.57	8542.93
	50000^2	9.5	2927.60	1927.28	14640.51
	60000^2	13.7	4736.27	3243.15	22977.13
3	5000^2	0.1	6.82	3.89	65.03
	10000^2	0.4	26.10	15.07	276.51
	20000^2	1.5	113.60	64.98	1032.23
	30000^2	3.4	642.27	237.24	4305.04
	40000^2	6.1	1557.33	798.03	8703.05
	50000^2	9.5	2624.07	1713.16	14299.41
	60000^2	13.7	4056.73	2663.41	20625.91

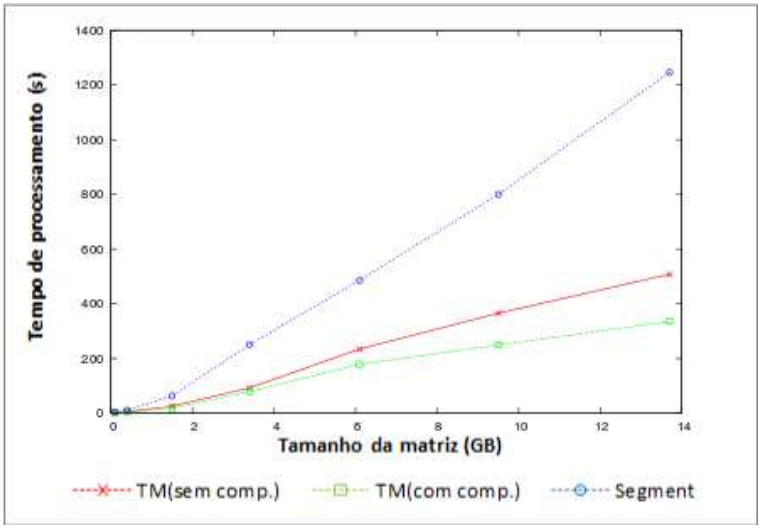

Figura 4.7. Gráfico de tempo de execução para a transposição da matriz considerando um conjunto de dados cujas matrizes tem tamanhos distintos (em número de células).

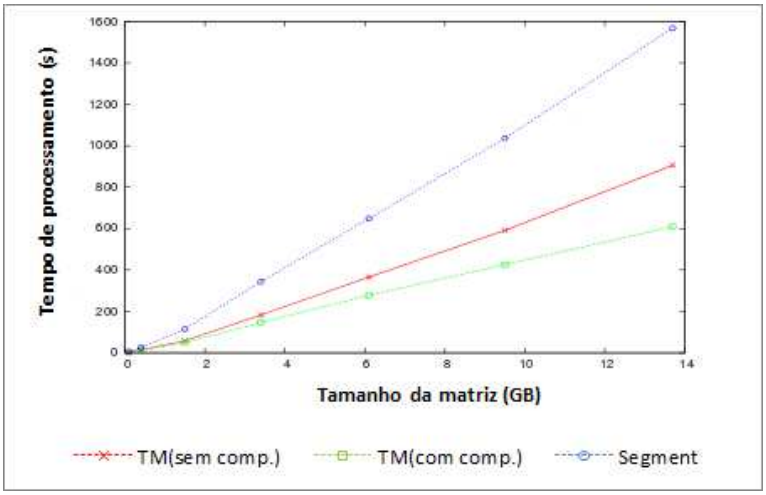

Figura 4.8. Gráfico de tempo de execução para o cálculo *viewshed* considerando diferentes tamanhos do terreno da região 2.

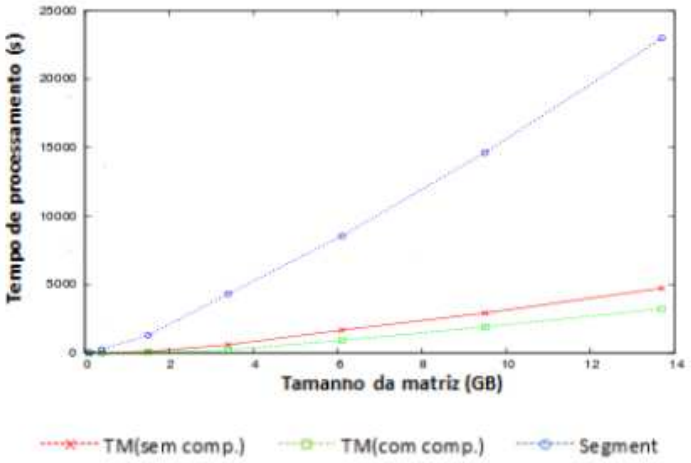

Figura 4.9. Gráfico de tempo de execução para o cálculo do fluxo acumulado considerando diferentes tamanhos do terreno da região 2.

Outra observação interessante é a velocidade alcançada quando a compressão foi usada na *TiledMatrix*. Em todos os casos, a eficiência da aplicação usando a *TiledMatrix* foi melhor quando a compressão foi ligada do que quando não foi. Em média, o uso da compressão melhorou a eficiência da aplicação em 50% e, em algumas situações, esta melhoria da eficiência foi mais de 200%, como pode-se observar na Tabela 4.3 nas Regiões 2 e 3 para os terrenos de tamanho 30.000×30.000 células.

4.6.3 Avaliação da TiledMatrix com o ArcGIS

A *TiledMatrix* também foi avaliada comparando o tempo de execução de duas aplicações que usam a *TiledMatrix* e o *ArcGIS* [15]. As aplicações escolhidas foram *viewshed* e o fluxo acumulado, visto que estas aplicações estão contidas no *ArcGIS* e exigem maior processamento. A biblioteca *TiledMatrix* foi configurada para utilizar a política LRU como a política de substituição e o tamanho do bloco foi definido como segue: para *viewshed*, foi usado um bloco quadrado com 1000 x 1000 células e no fluxo acumulado, blocos de 250 x 250 células.

As matrizes usadas nos testes foram obtidas do site STRM [31] e elas representam terrenos com resolução de $30m \times 30m$ correspondendo a regiões dos Estados Unidos da América. As entradas para a aplicação que calcula o fluxo acumulado utilizando a *TiledMatrix*, são as matrizes de direção de fluxo obtidas pelo algoritmo *RWFlood* [26] usando-se estes conjuntos de dados de terrenos. E, no cálculo do *viewshed*, foi usado o raio de interesse tão grande quanto possível, isto é, um raio de interesse cobrindo todo o terreno e o observador foi posicionado 50 metros acima do terreno.

Os resultados dos testes são apresentados nas Tabelas 4.4 e 4.5. De acordo com os resultados, todas as aplicações executam mais rapidamente usando *TiledMatrix* do que *ArcGIS* e, em geral, o *viewshed* e o fluxo acumulado foram, respectivamente, 7 e 133 vezes mais rápido com a *TiledMatrix*. Mas, note que, em alguns casos, a *TiledMatrix* pode tornar a aplicação 214 vezes mais rápida (por exemplo, fluxo acumulado no terreno da região 2 de tamanho 1.5 GB). Este aumento na eficiência pode ser explicado por duas razões: a transferência de dados é reduzida por causa da compressão do bloco e o uso da estratégias mais eficientes para gerenciamento do bloco (ver seção 3.3).

Tabela 4.4. Tempo (em segundos) para calcular o *viewshed* em um terreno utilizando a biblioteca *TiledMatrix* e *ArcGIS*.[1]

Viewshed				
Terrenos				
Região	linha × coluna	GB	**TiledMatrix**	**ArcGIS**
2	5000^2	0.1	3.15	20.60
	10000^2	0.4	12.35	67
	20000^2	1.5	49.49	315
	30000^2	3.4	147.16	999
	40000^2	6.1	278.45	1796
	50000^2	9.5	426.39	∞
	60000^2	13.7	609.76	∞
3	5000^2	0.1	3.19	19.73
	10000^2	0.4	12.65	68
	20000^2	1.5	51.46	361
	30000^2	3.4	151.56	867
	40000^2	6.1	270.04	1782
	50000^2	9.5	434.58	∞
	60000^2	13.7	575.72	∞

Tabela 4.5. Tempo (em segundos) para calcular o fluxo acumulado em um terreno utilizando a biblioteca *TiledMatrix* e *ArcGIS*.[1]

Fluxo Acumulado				
Terrenos				
Região	linha × coluna	GB	**TiledMatrix**	**ArcGIS**
2	5000^2	0.1	3.86	452
	10000^2	0.4	15.29	2354
	20000^2	1.5	65.26	13975
	30000^2	3.4	242.78	50358
	40000^2	6.1	938.57	142882
	50000^2	9.5	1927.28	∞
	60000^2	13.7	3243.15	∞
3	5000^2	0.1	3.89	371
	10000^2	0.4	15.07	2335
	20000^2	1.5	64.98	10507
	30000^2	3.4	237.24	32669
	40000^2	6.1	798.03	65792
	50000^2	9.5	1713.16	∞
	60000^2	13.7	2663.41	∞

4.6.4 Avaliação das políticas de substituição de bloco LRU, FIFO e *Random* na biblioteca *TiledMatrix*

Nesta subseção são apresentados os resultados de uma análise experimental para avaliar o desempenho dos algoritmos para obter a transposta da matriz, *viewshed* e fluxo acumulado utilizando três políticas de substituição de bloco LRU, FIFO e *Random* implementadas na biblioteca *TiledMatrix*. Os testes experimentais foram constituídos de terrenos fictícios (*Random* e Plano) e um terreno real correspondente a uma região dos Estados Unidos da América. O terreno *Random* foi gerado com o intuito de forçar uma determinada aplicação a realizar mais acessos ao disco, visto que cada célula do terreno possui um valor de elevação escolhido de maneira aleatória. O terreno Plano foi gerado para que a aplicação tenha menos acessos ao disco, já que as células do terreno possuem valores de elevação similares. O tamanho dos blocos escolhidos para as aplicações transposta e *viewshed* foram 1000 × 1000

[1] O valor ∞ indica que não foi possível processar terrenos de tamanho superiores a 6,1GB no ArcGIS.

células e para o fluxo acumulado foi de 400 × 400 células. Em todas as aplicações a biblioteca foi configurada para usar, no máximo 3GB de RAM. Cada teste foi executado três vezes e então tirou-se a média tanto para os tempos de execução quanto para a quantidade de blocos carregados, removidos e escritos.

Na Tabela 4.6 pode-se observar o tempo de execução gasto para executar a transposta utilizando diferentes políticas de substituição de blocos como LRU, FIFO e *Random*. A Tabela 4.7 mostra a quantidade de blocos lidos, carregados, removidos e escritos para as políticas LRU, FIFO e *Random* respectivamente. Ao utilizar a política LRU, a aplicação ficou mais rápida para os terrenos de tamanho 5000 × 5000, 20000 × 20000 e 50000 × 50000. Para a FIFO a aplicação ficou mais rápida com os terrenos de tamando 30000 × 30000 e 40000 × 40000. E por último, ao utilizar a política *Random*, a aplicação ficou mais rápida para o terreno de tamanho 10.000 × 10.000. Nota-se que, para a aplicação transposta o uso da política LRU é o mais indicado, visto que a aplicação é mais rápida ao utilizar a política LRU na maioria dos casos.

Na Tabela 4.8 pode-se observar o tempo de execução gasto para executar o *viewshed* utilizando diferentes políticas de substituição de blocos como LRU, FIFO e *Random* em terrenos distintos. A Tabela 4.9 mostra a quantidade de blocos lidos, carregados, removidos e escritos para as políticas LRU, FIFO e *Random* respectivamente. Para o terreno *Random*, ao utilizar a política LRU, a aplicação ficou mais rápida para os terrenos de tamanho 20.000 × 20.000 e 30.000 × 30.000. Para a FIFO a aplicação ficou mais rápida com os terrenos de tamando 5.000 × 5.000, 10.000 × 10.000, 40.000 × 40.000 e 50.000 × 50.000. E por último, ao utilizar a política *Random* a aplicação ficou mais lenta para todos os tamanhos de terreno. Para os terrenos da Região 2, ao utilizar a política LRU a aplicação ficou mais rápida para os terrenos de tamanho 40.000 × 40.000 e 50.000 × 50.000. Com uso da política FIFO a aplicação ficou mais rápida para os terrenos de tamanho 5.000 × 5.000, 10.000 × 10.000, 20.000 × 20.000 e 30.000 × 30.000. Ao utilizar a política *Random* a aplicação ficou mais lenta para todos os terrenos. Para o terreno Plano, a aplicação ficou em todos os testes mais lenta ao utilizar a política LRU. Com uso da política FIFO a aplicação ficou mais rápida para os terrenos 5.000 × 5.000, 10.000 × 10.000, 20.000 × 20.000 e 30.000 × 30.000. Com o uso da política *Random* a aplicação ficou mais rápida para os terrenos de 40.000 × 40.000 e 50.000 × 50.000. É importante salientar que a aplicação ficou mais rápida com uso da política FIFO do que com o uso das políticas LRU e *Random* na maioria dos testes para todos os tipos de terreno usados. Portanto, os testes indicam que é mais interessante o uso da política FIFO para a aplicação *viewshed* do que o uso das políticas LRU e *Random*.

Na Tabela 4.10 pode-se observar o tempo de execução gasto para executar o fluxo acumulado utilizando diferentes políticas de substituição de blocos, e.g., LRU, FIFO e *Random* em terrenos distintos. A Tabela 4.11 mostra a quantidade de blocos lidos, carregados, removidos e escritos para as políticas LRU, FIFO e *Random* respectivamente. Para o terreno *Random*, ao utilizar a política LRU, a aplicação ficou mais rápida para os terrenos de tamanho 10.000×10.000, 20.000×20.000 e 30.000×30.000. Para a FIFO, a aplicação ficou mais lenta para todos os terrenos. E, por último, ao utilizar a política *Random*, a aplicação ficou mais rápida para o terreno de tamanho 5.000×5.000. Para os terrenos *Random* de tamanho 40.000×40.000 e 50.000×50.000, o tempo de execução gasto pela aplicação ultrapassou 100.000 segundos. Assim, para estes casos considerou-se o tempo gasto pela aplicação como sendo infinito. Para os terrenos da Região 2, ao utilizar a política LRU, a aplicação ficou mais lenta para todos os tamanhos de terreno. Com uso da política FIFO, a aplicação ficou mais rápida para os terrenos de tamanho 5.000×5.000, 10.000×10.000, 20.000×20.000, 40.000×40.000 e 50.000×50.000. Ao utilizar a política *Random*, a aplicação ficou mais rápida para o terreno de tamanho 30.000×30.000. Para o terreno Plano, a aplicação ficou, em todos os testes, mais lenta ao se utilizar a política LRU. Com uso da política FIFO, a aplicação ficou mais rápida para os terrenos 5.000×5.000, 10.000×10.000, 20.000×20.000, 30.000×30.000 e 40.000×40.000. Com o uso da política *Random*, a aplicação ficou mais rápida para o terreno de 50.000×50.000.

É importante salientar que a aplicação fluxo acumulado ficou mais rápida com uso da política FIFO do que com o uso das políticas LRU e *Random* na maioria dos testes para todos os terrenos da Região 2 e para os terrenos Planos. Portanto, para estes dois tipos de terreno, os testes indicam que é mais interessante o uso da política FIFO para a aplicação fluxo acumulado do que o uso das políticas LRU e *Random*. Para terrenos *Random*, a política LRU mostrou tornar a aplicação mais eficiente em relação às outras políticas de substituição implementadas na biblioteca e, portanto, de acordo com os testes, o uso da política *LRU* é mais interessante no terreno *Random*.

Tabela 4.6. Tempo (em segundos) para calcular a transposta da matriz em um terreno utilizando a biblioteca *TiledMatrix* configurada para usar as políticas de substituição LRU, FIFO e *Random*.

Tamanho da matriz				
linha × coluna	GB	**LRU**	**FIFO**	**Random**
5000^2	0,1	1,05	1,18	1,14
10000^2	0,4	6,25	6,45	5,84
20000^2	1,5	18,94	21,82	25,16
30000^2	3,4	82,91	73,13	94,13
40000^2	6,1	219,57	210,85	251,73
50000^2	9,5	359,56	419,77	431,20

Tabela 4.7. Número de blocos lidos, carregados, removidos e escritos para calcular a transposta da matriz utilizando as políticas de substituição de bloco LRU, FIFO e Random.

Tamanho da matriz		B. lidos	**B. carregados**			**B. Removidos**			**B. Escritos**		
linha × coluna	GB		LRU	FIFO	RAM	LRU	FIFO	RAM	LRU	FIFO	RAM
5000^2	0.1	25	25	25	25	0	0	0	0	0	0
10000^2	0.4	100	100	100	105	0	0	6	0	0	6
20000^2	1.5	400	400	400	517	0	0	147	0	0	136
30000^2	3.4	900	1.275	1.103	1.411	489	317	742	470	317	619
40000^2	6.1	1.600	2.983	2.926	2.921	2.197	2.140	2.147	1.600	1.600	1.501
50000^2	9.5	2.500	4.874	4.860	4.919	4.088	4.074	4.133	2.500	2.500	2.551

Tabela 4.8. Tempo (em segundos) para calcular a *viewshed* da matriz em um terreno utilizando a biblioteca *TiledMatrix* configurada para usar as políticas de substituição LRU, FIFO e *Random*.

Terrenos			**LRU**	**FIFO**	**Random**
Dados	linha × coluna	GB			
Random	5000^2	0,1	3,02	2,70	3,10
	10000^2	0,4	12,54	12,43	12,69
	20000^2	1,5	52,31	51,85	56,36
	30000^2	3,4	138,18	141,56	149,21
	40000^2	6,1	309,55	298,88	297,42
	50000^2	9,5	488,30	470,76	485,23
Região 2	5000^2	0,1	3,10	3,05	3,09
	10000^2	0,4	12,64	12,55	12,77
	20000^2	1,5	50,53	50,22	53,43
	30000^2	3,4	133,25	133,22	145,86
	40000^2	6,1	282,12	298,83	283,53
	50000^2	9,5	435,55	459,84	455,79
Plano	5000^2	0,1	3,66	3,22	3,25
	10000^2	0,4	13,69	13,31	13,52
	20000^2	1,5	53,06	51,79	52,87
	30000^2	3,4	121,82	119,04	121,03
	40000^2	6,1	229,82	226,87	224,55
	50000^2	9,5	376,88	368,71	366,35

Tabela 4.9. Número de blocos lidos, carregados, removidos e escritos para calcular *Viewshed* utilizando a política de substituição de blocos LRU, FIFO e *Random*.

Terrenos			B. lidos	**B. carregados**			**B. Removidos**			**B. Escritos**		
Dados	linha × coluna	GB		LRU	FIFO	RAM	LRU	FIFO	RAM	LRU	FIFO	RAM
Random	5000^2	0.1	25	50	50	51	0	0	2	0	0	2
	10000^2	0.4	100	200	200	207	0	0	16	0	0	16
	20000^2	1.5	400	800	800	1.083	0	0	364	0	0	345
	30000^2	3.4	900	2.302	2.295	2.398	1.044	1.037	1.332	1.044	1.037	1.139
	40000^2	6.1	1.600	4.338	4.340	4.619	3.080	3.082	3.407	2.571	2.571	2.616
	50000^2	9.5	2.500	7.151	7.101	7.510	5.893	5.843	6.261	4.371	4.371	4.487
Região 2	5000^2	0.1	25	50	50	52	0	0	2	0	0	2
	10000^2	0.4	100	200	200	215	0	0	18	0	0	18
	20000^2	1.5	400	800	800	968	0	0	307	0	0	283
	30000^2	3.4	900	2.342	2.336	2.452	1.084	1.078	1.366	1.084	1.078	1.164
	40000^2	6.1	1.600	4.587	4.589	4.803	3.329	3.331	3.581	2.651	2.651	2.695
	50000^2	9.5	2.500	7.289	7.239	7.591	6.031	5.981	6.340	4.371	4.371	4.496
Plano	5000^2	0.1	25	50	50	52	0	0	2	0	0	2
	10000^2	0.4	100	200	200	117	0	0	17	0	0	17
	20000^2	1.5	400	800	800	1.087	0	0	372	0	0	349
	30000^2	3.4	900	2.950	2.604	2.950	1.791	1.346	5.975	1.430	1.346	3.185
	40000^2	6.1	1.600	5.881	5.823	5.975	4.623	4.565	4.727	3.220	3.220	3.185
	50000^2	9.5	2.500	9.672	9.625	9.948	8.414	8.367	8.690	5.025	5.028	5.231

Tabela 4.10. Tempo (em segundos) para calcular o fluxo acumulado da matriz em um terreno utilizando a biblioteca *TiledMatrix* configurada para usar as políticas de substituição LRU, FIFO e *Random*.[2]

Terrenos			**LRU**	**FIFO**	**Random**
Dados	linha × coluna	GB			
Random	5000^2	0,1	18,31	18,27	18,01
	10000^2	0,4	46,03	72,34	72,46
	20000^2	1,5	266,64	381,45	381,65
	30000^2	3,4	610,68	877,11	911,02
	40000^2	6,1	∞	∞	∞
	50000^2	9,5	∞	∞	∞
Região 2	5000^2	0,1	10,42	9,27	9,81
	10000^2	0,4	44,21	40,92	41,96
	20000^2	1,5	264,90	257,31	258,75
	30000^2	3,4	596,30	591,07	589,64
	40000^2	6,1	1.108,72	1.059,73	1.070,25
	50000^2	9,5	1.736,22	1.622,15	1.699,00
Plano	5000^2	0,1	8,82	8,13	8,41
	10000^2	0,4	36,81	31,73	32,09
	20000^2	1,5	227,59	215,68	217,15
	30000^2	3,4	522,00	487,25	496,62
	40000^2	6,1	930,75	905,60	908,00
	50000^2	9,5	1.530,00	1.481,53	1.446,37

[2]Para terrenos *Random* de tamanho 6,1 e 9,5 GB, o tempo ultrapassou 100.000 segundos e então, atribuiu-se tempo infinito para estes terrenos.

Tabela 4.11. Número de blocos lidos, carregados, removidos e escritos para calcular o fluxo acumulado utilizando a política de substituição de blocos LRU, FIFO e *Random*.[3]

Terrenos			B. lidos	**B. carregados**			**B. Removidos**			**B. Escritos**		
Dados	linha × coluna	GB		LRU	FIFO	RAM	LRU	FIFO	RAM	LRU	FIFO	RAM
	5000^2	0.1	49	204	196	204	0	0	8	0	0	8
	10000^2	0.4	169	676	676	721	0	0	47	0	0	47
	20000^2	1.5	625	3.750	3.750	4.079	2.498	2.498	2.827	0	0	329
Random	30000^2	3.4	1.044	8.664	8.664	12.284	5.774	5.774	10.838	0	0	3.611
	40000^2	6.1	2.500	∞	∞	∞	∞	∞	∞	∞	∞	∞
	50000^2	9.5	3.969	∞	∞	∞	∞	∞	∞	∞	∞	∞
	5000^2	0.1	49	196	196	204	0	0	8	0	0	8
	10000^2	0.4	169	676	676	723	0	0	48	0	0	48
	20000^2	1.5	625	3.750	3.750	4.040	2.498	2.498	2.799	0	0	301
Região 2	30000^2	3.4	1.044	8.664	8.664	10.859	5.774	5.774	8.235	0	0	2.292
	40000^2	6.1	2.500	21.681	21.632	22.495	18.325	18.276	19.212	5.432	5.720	7.409
	50000^2	9.5	3.969	34.566	35.210	37.272	31.210	31.854	33.924	9.282	10.085	12.669
	5000^2	0.1	49	196	196	204	0	0	8	0	0	8
	10000^2	0.4	169	676	676	719	0	0	45	0	0	45
	20000^2	1.5	625	3.750	3.750	3.995	2.498	2.498	2.765	0	0	260
Plano	30000^2	3.4	1.044	8.664	8.664	10.442	5.774	5.774	7.924	0	0	1.877
	40000^2	6.1	2.500	20.901	19.111	20.246	17.545	15.755	17.072	5.042	4.934	5.082
	50000^2	9.5	3.969	35.069	35.085	34.948	31.713	31.729	31.610	9.578	9.594	10.151

4.6.5 Avaliação da taxa de compressão da biblioteca

O objetivo desta seção é apresentar os resultados da avaliação da velocidade de compressão (MB/s) alcançado pela biblioteca *TiledMatrix* durante a transferência de dados. Para isso foram desenvolvidos programas de leitura e escrita consecutiva de blocos e leitura e de escrita aleatória de blocos para o disco. Inicialmente, a memória é inicializada e, em seguida, os blocos são carregados na memória, de maneira consecutiva. Esse algoritmo também é válido para carregar blocos de maneira aleatória; porém a ordem em que os blocos serão carregados é feita de maneira aleatória. Nos programas "escreve bloco sequencialmente" e "carrega blocos aleatoriamente", a memória é limpa para garantir que não existem blocos carregados na memória; os blocos são carregados de maneira sequencial e aleatória respectivamente e a memória novamente é limpa. Neste último passo em que a memória é limpa, os blocos são escritos no disco.

A Tabela 4.12 apresenta a taxa de compressão durante a transferência de blocos de modo sequencial e aleatório da biblioteca *TiledMatrix* em um terreno fictício(gerado de maneira aleatória por um programa de computador). É importante salientar que a leitura consecutiva é 1,33 vezes mais rápida do que a leitura aleatória. A escrita sequencial é 1,39 vezes mais rápida do que a escrita aleatória. A compressão durante a leitura sequencial foi em média 66,46 MB/s e na escrita sequencial foi de 65,82 MB/s. A compressão para operações aleatórias diminuem para 49,95 MB/s para a leitura (47,21 MB/s para escrita).

[3]Para terrenos *Random* de tamanho 6,1 e 9,5 GB, o tempo ultrapassou 100.000 segundos e então, atribuiu-se tempo infinito para estes terrenos.

Tabela 4.12. Taxa de compressão em MB/s durante transferência de blocos para e da memória principal da biblioteca *TiledMatrix* para a leitura e escrita aleatória usando terrenos aleatórios de tamanho 1,5, 3,4, 6,1, 9,5 GB, em que B é o tamanho do bloco.

Aleatório-Terreno aleatório								
B	**leitura**				**escrita**			
	Tamanho do terreno				**Tamanho do terreno**			
	1,5	3,4	6,1	9,5	1,5	3,4	6,1	9,5
100	50	23	13	8	26	13	7	5
200	50	24	16	13	28	15	10	8
400	54	33	28	26	35	21	18	18
800	66	50	50	42	50	40	35	35
1000	66	60	50	50	54	46	46	42
1500	66	60	54	54	66	54	54	50
2000	66	60	60	60	75	66	60	60
4000	66	66	66	66	85	85	75	75
5000	85	66	66	66	85	85	75	75
Sequencial-Terreno aleatório								
B	**leitura**				**escrita**			
	Tamanho do terreno				**Tamanho do terreno**			
	1,5	3,4	6,1	9,5	1,5	3,4	6,1	9,5
100	60	60	60	60	37	37	35	37
200	60	60	60	66	42	40	37	40
400	60	60	60	66	54	50	50	50
800	66	66	60	66	66	60	66	66
1000	66	66	66	75	75	66	66	66
1500	66	66	66	75	85	75	75	75
2000	66	66	66	75	85	85	85	85
4000	75	66	66	85	85	85	85	85
5000	75	66	66	85	85	85	85	85

A Tabela 4.13 apresenta a taxa de compressão durante transferência de blocos de modo sequencial e aleatório da biblioteca *TiledMatrix* em um terreno real correspondente à região 2 do mapa como mostra na Figura 4.5. É importante salientar que a leitura consecutiva é 1,11 vezes mais rápida do que a leitura aleatória. A escrita sequencial é 1,003 vezes mais rápida do que a escrita aleatória. A velocidade da taxa de compressão durante a leitura sequencial foi em média 63,5075 MB/s e

na escrita sequencial foi de 54,27 MB/s. A velocidade da taxa de compressão para operações aleatórias diminuem para 56,775 MB/s para a leitura (54,1075 MB/s para escrita).

Tabela 4.13. Taxa de compressão em MB/s durante a transferência de blocos para e da memória principal da biblioteca *TiledMatrix* para a leitura e escrita aleatória usando terrenos aleatórios de tamanho 1,5, 3,4, 6,1, 9,5 GB, em que B é o tamanho do bloco.

Aleatório-R2								
B	**leitura**				**escrita**			
	Tamanho do terreno				**Tamanho do terreno**			
	1,5	3,4	6,1	9,5	1,5	3,4	6,1	9,5
100	50	24	13	9	22	13	7	5
200	50	25	18	16	27	15	10	10
400	50	33	31	22	30	21	20	21
800	50	35	46	75	35	40	31	37
1000	50	54	54	75	46	46	42	50
1500	66	66	60	75	60	54	60	75
2000	66	75	66	100	75	66	75	85
4000	75	75	75	120	100	85	100	150
5000	75	75	75	120	100	85	100	150
Sequencial-R2								
B	**leitura**				**escrita**			
	Tamanho do terreno				**Tamanho do terreno**			
	1,5	3,4	6,1	9,5	1,5	3,4	6,1	9,5
100	60	60	54	50	25	25	25	25
200	60	60	54	50	37	37	33	33
400	60	66	54	50	40	40	40	33
800	60	66	54	50	40	40	40	37
1000	60	66	54	60	46	46	50	42
1500	60	66	66	60	60	66	60	54
2000	60	66	66	75	66	66	66	75
4000	75	75	75	85	85	85	85	100
5000	75	75	75	85	100	85	100	100

Como esperado, ao aumentar o tamanho do bloco, a taxa de compressão da biblioteca *TiledMatrix* durante a transferência de dados também aumenta. Isso porque o custo da operação de *seek* pode ser amortizado à medida que se usa um

tamanho de bloco maior.

4.6.6 Comparação algoritmo *LZ4 Sequencial* e *LZ4 Paralelo*

Como foi descrito na seção 4.6.2, a compressão reduz a transferência de dados melhorando a eficiência de uma aplicação. Sendo assim, para tentar melhorar ainda mais essa eficiência, o algoritmo LZ4 foi implementado em paralelo. A idéia é que os blocos sejam particionados em sub-blocos, sendo que cada um desses sub-blocos será comprimido por uma das threads disponíveis, isto é, supondo que existam k threads, o bloco será particionado em k sub-blocos. Desta forma, a compressão e a descompressão serão realizadas em paralelo.

A fim de avaliar o ganho que esta estratégia produz em relação ao uso da compressão Sequencial, alguns testes foram realizados em algumas aplicações como: transposta, *viewshed* e mediana. O tamanho do bloco escolhido para a transposta, *viewshed* e filtro da mediana foi de 1000×1000 células. Em todas as aplicações a biblioteca foi configurada para usar, no máximo, 2GB de RAM. Para a transposição da matriz e o cálculo da mediana utilizaram-se diferentes tamanhos de matrizes e para o cálculo do *viewshed*, utilizou-se diferentes tamanhos de terrenos. No cálculo do *viewshed*, foi usado o raio de interesse tão grande quanto possível, isto é, um raio de interesse cobrindo todo o terreno, e o observador posicionado 50 metros acima do terreno.

Os testes consistiram em mensurar o tempo total gasto para a execução de cada aplicação e o tempo gasto para carregar os blocos durante a execução de cada aplicação. Os resultados destes testes são apresentados nas Tabelas 4.14, 4.16 e 4.18. De acordo com os resultados, as aplicações executam mais mais rápidas usando LZ4 em Paralelo do que LZ4 sequencial na maioria dos testes. O uso da compressão Paralela melhorou a eficiência das aplicações, em média, em 4%. Em alguns casos onde o terreno é pequeno, o ganho com a paralelização foi negativo. Isso acontece porque estes terrenos cabem na memória interna e, portanto, não há operações de troca, logo o uso da compressão de dados paralela torna a *TiledMatrix* menos eficiente, pois aloca-se mais memória para a compressão paralela do que para a compressão sequencial. Nas Tabelas 4.15, 4.17 e 4.19 pode-se observar que, em todos os testes, o tempo para carregar os blocos durante a execução das aplicações é menor quando a compressão Paralela é utilizada do que quando se usa a compressão Sequencial.

Note que os ganhos obtidos nos testes podem ser explicados devido à redução na transferência de dados, visto que os subblocos são comprimidos, reduzindo o

tamanho deles. Mais precisamente, sejam $b_1, b_2, ..., b_i$ os tamanhos em bytes de cada subbloco, $S = b_1 + b_2 + ... + b_i$ a soma do tamanho dos subblocos comprimidos e B o tamanho do bloco original comprimido. Visto que tanto na implementação do algoritmo LZ4 paralelo quanto na implementação do algoritmo LZ4 sequencial são realizadas uma única escrita para escrever todos os subblocos no disco e uma única leitura para ler todos os subblocos do disco, se $S < B$ então a transferência de dados no algoritmo Paralelo é menor do que a transferência de dados no algoritmo Sequencial e portanto o algoritmo LZ4 em Paralelo tende a melhorar a eficiência das aplicações.

Tabela 4.14. Tempo (em segundos) gasto para calcular a transposta de uma matriz, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
	5000^2	0.1	1,07	1,67	-56%
	10000^2	0.4	4,97	6,27	-26%
	20000^2	1.5	19,96	21,52	-7%
2	30000^2	3.4	98,79	95,02	4%
	40000^2	6.1	195,87	182,06	7%
	50000^2	9.5	303,95	280,02	8%

Tabela 4.15. Tempo (em segundos) gasto para carregar blocos durante o cáculo da transposta de uma matriz, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
	5000^2	0.1	0,0	0,0	0%
	10000^2	0.4	0,0	0,0	0%
	20000^2	1.5	0,0	0,0	0%
2	30000^2	3.4	50	44	12%
	40000^2	6.1	92	79	14%
	50000^2	9.5	131	107	18%

Tabela 4.16. Tempo (em segundos) gasto para calcular o *viewshed* de um terreno, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
2	5000^2	0.1	3,08	3,19	-3%
	10000^2	0.4	12,62	12,73	-12,62%
	20000^2	1.5	62,34	57,81	7%
	30000^2	3.4	149,59	146,2	2%
	40000^2	6.1	285,29	277,17	3%
	50000^2	9.5	440,15	430,78	2%

Tabela 4.17. Tempo (em segundos) gasto para carregar blocos durante o cálculo do *viewshed* de um terreno, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
2	5000^2	0.1	0,0	0,0	0%
	10000^2	0.4	0,0	0,0	0%
	20000^2	1.5	7,0	2,0	71%
	30000^2	3.4	22,0	16,0	27%
	40000^2	6.1	48,0	37,0	23%
	50000^2	9.5	75,0	55,0	27%

Tabela 4.18. Tempo (em segundos) gasto para calcular o filtro da mediana de um terreno, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
2	5000^2	0.1	8,83	8,76	0,8%
	10000^2	0.4	34,23	34,06	0,5%
	20000^2	1.5	144,86	141,49	2%
	30000^2	3.4	354,48	337,70	5%
	40000^2	6.1	671,46	660,03	2%
	50000^2	9.5	1002,04	994,8	0,7%

Tabela 4.19. Tempo (em segundos) gasto para calcular o filtro da mediana de um terreno, utilizando a biblioteca *TiledMatrix* (com compressão paralela e sequencial).

Terrenos			**TiledMatrix**		Ganho com a
Região	linha × coluna	GB	comp. Sequencial	comp. Paralela	Paralelização
2	5000^2	0.1	0,0	0,0	0%
	10000^2	0.4	0,0	0,0	0%
	20000^2	1.5	5,0	2,0	60%
	30000^2	3.4	23,0	5,0	78%
	40000^2	6.1	59,0	50,0	15%
	50000^2	9.5	83,0	65,0	22%

Capítulo 5

Conclusões

Nós apresentamos uma nova biblioteca, nomeada *TiledMatrix*, para auxiliar o desenvolvimento de aplicações que processam grandes matrizes armazenadas na memória externa, principalmente, discos. Esta biblioteca usa algumas estruturas de dados especiais e algoritmos de gerenciamento de memória *cache* para reduzir o número de acessos ao disco rígido. O objetivo básico é permitir que uma aplicação originalmente desenvolvida para processamento na memória interna possa ser facilmente adaptada para a memória externa. A biblioteca fornece uma interface para acessar a memória externa que é similar ao tradicional acesso a matrizes na memória interna. Uma interessante estratégia incluída nesta biblioteca foi o uso da compressão de dados para reduzir a transferência entre as memórias interna e externa.

Apresentamos no Capítulo 3 a descrição da biblioteca *TiledMatrix* mostrando a estrutura de dados utilizada pela TiledMatrix, incluindo a descrição de como a *TiledMatrix* faz uso do algoritmo LZ4 [9], um algoritmo utilizado para uma compressão de dados mais rápida e sem perdas reduzindo o número de operações de E/S durante a transferência de bloco para e do disco. Ainda no Capítulo 3, mostramos uma melhoria no algoritmo LZ4 realizada neste trabalho que permite que a compressão dos blocos seja feita de maneira paralela.

Os critérios e algoritmos utilizados para mostrar a eficiência da *TiledMatrix* foram apresentados no Capítulo 4. A *TiledMatrix* foi implementada e testada em algumas aplicações que requerem intensivo processamento de matrizes tais como: calcular a transposta da matriz e o cálculo do *viewshed* e fluxo acumulado em terrenos representados por uma matriz de elevação. Estas aplicações foram implementadas em duas versões: uma usando a bibliteca *TiledMatrix* e outra usando a biblioteca *Segment*, sendo que todas estas aplicações foram descritas neste mesmo capítulo. As aplicações foram executadas utilizando muitos conjuntos de dados com

diferentes tamanhos e, de acordo com os testes, todas as aplicações executaram mais rápidos utilizando a *TiledMatrix* do que com a *Segment*. Em média, os testes foram 7 vezes mais rápido com a *TiledMatrix* e, em alguns casos, 18 vezes mais rápidos. Além disso, os testes mostraram que o uso da compressão melhorou a eficência da aplicação em 50% e, em algumas situações, quase 200%. Neste mesmo Capítulo, avaliamos a taxa de compressão da *TiledMatrix* durante a transferência de dados para e do disco realizando leituras e escritas aleatórias e consecutivas de blocos. Por meio desta avaliação foi possível concluir que a velocidade da taxa de compressão durante a leitura consecutiva foi 1,33 mais rápida do que a leitura aleatória e a taxa de compressão durante a escrita consecutiva foi 1,39 vezes mais rápida do que a escrita aleatória. Além disso, mostra-se que a compressão de dados paralela em média melhorou a eficiência das aplicações que foram testadas em 3,6%. Contudo, a *TiledMatrix* foi comparada com o *ArcGIS* e ela foi bem mais eficiente do que ele, sendo que, em média, as aplicações foram 70 vezes mais rápidas com a *TiledMatrix* e, em alguns casos, a *TiledMatrix* pode tornar a aplicação 214 vezes mais rápida (como foi feito no último parágrafo da seção 4.6.3 pág. 39).

A partir de todos os experimentos realizados concluímos que a *TiledMatrix* é bastante simples e capaz de auxiliar aplicações a processarem matrizes muito maiores que a memória interna em um tempo menor do que outras bibliotecas existentes com propósito semelhante.

Como um trabalho futuro, nós pretendemos avaliar a *TiledMatrix* em algumas outras aplicações e elaborar algumas estratégias para definir o tamanho do bloco de acordo com o padrão de acesso de memória da aplicação e o tamanho da memória.

Referências Bibliográficas

[1] Aggarwal, A. & Vitter, J. S. (1988). The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116--1127.

[2] Andrade, M. V. A.; aes, S. V. G. M.; Franklin, W. R. & Cutler, B. M. (2011). Efficient viewshed computation on terrain in external memory. pp. 381--397. GeoInformatica.

[3] Arge, L.; Procopiuc, O. & Vitter, J. S. (2002). Implementing i/o-efficient data structures using tpie. Em *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pp. 88--100, London, UK, UK. Springer-Verlag.

[4] Arge, L.; Toma, L. & Vitter, J. S. (2001). I/o-efficient algorithms for problems on grid-based terrains. *J. Exp. Algorithmics*, 6.

[5] Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78--101.

[6] Bostoen, T.; Mullender, S. & Berbers, Y. (2013). Power-reduction techniques for data-center storage systems. *ACM Comput. Surv.*, 45(3):33:1--33:38.

[7] Chrobak, M. & Noga, J. (1999). Lru is better than fifo. *Algorithmica*, 23(2):180–185.

[8] Coffman, Jr., E. G. & Denning, P. J. (1973). *Operating Systems Theory*. Prentice Hall Professional Technical Reference.

[9] Collet, Y. (2011). Lz4 explained. http://fastcompression.blogspot.com.br/2011/05/lz4-explained.html. Acessado em 12 de setembro de 2013.

[10] Crauser, A. (2001). *LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice*. Doctoral dissertation, Universität des Saarlandes.

[11] Crauser, A. & Mehlhorn, K. (1999). Leda-sm : Extending leda to secondary memory. Em Vitter, J. S. & Zaroliagis, C. D., editores, *Algorithm engineering (WAE-99) : 3rd International Workshop, WAE'99*, volume 1668 of *Lecture Notes in Computer Science*, pp. 228--242, London, UK. Springer.

[12] Dementiev, R. (2006). *Algorithm Engineering for Large Data Sets.* Doctoral dissertation, Universität des Saarlandes.

[13] Dementiev, R.; Kettner, L. & Sanders, P. (2005). Stxxl : Standard template library for xxl data sets. http://stxxl.sourceforge.net/. Acessed July 15, 2012.

[14] Deng, Y. (2011). What is the future of disk drives, death or rebirth? *ACM Comput. Surv.*, 43(3):23:1--23:27.

[15] ESRI (Environmental Systems Resource Institute) (2012). ArcMap 10.2.1. ESRI, Redlands, California.

[16] Ferreira, C. R.; Magalhães, S. V. G.; Andrade, M. V. A.; Franklin, W. R. & Pompermayer, A. M. (2012). More efficient terrain viewshed computation on massive datasets using external memory. Em *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pp. 494--497, New York, NY, USA. ACM.

[17] Fishman, J.; Haverkort, H. J. & Toma, L. (2009). Improved visibility computation on massive grid terrains. Em Wolfson, O.; Agrawal, D. & Lu, C.-T., editores, *GIS*, pp. 121–130. ACM.

[18] Franklin, W. R. & Ray, C. (1994). Higher isnt́ necessarily better – visibility algorithms and experiments. 6th Symposium on Spatial Data Handling, Edinburgh, Scotland.

[19] González, A.; Aliagas, C.; Valero, M. & Nord, C. (1995). A data cache with multiple caching strategies tuned to different types of locality. pp. 338--347.

[20] GRASS, D. T. (2011). Geographic resources analysis support system (GRASS GIS) software. http://grass.osgeo.org. Accessed July 15, 2012.

[21] Haverkort, H.;], L. & Zhuang, Y. (2007). Computing visibility on terrains in external memory. Em *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments / Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALCO).*

[22] Haverkort, H. & Janssen, J. (2012). Simple i/o-efficient flow accumulation on grid terrains. *CoRR - Compting Research Repository*, abs/1211.1857.

[23] Hennessy, J. L. & Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edição.

[24] Kumar, S. & Wilkerson, C. B. (1998). Exploiting spatial locality in data caches using spatial footprints. Em *25 Years ISCA: Retrospectives and Reprints*, pp. 357–368.

[25] Lz4 (2012). Extremely fast compression algorithm. http://code.google.com/p/lz4/. Acessado em 1 de junho de 2012.

[26] Magalhães, S. V. G.; Andrade, M. V. A.; Franklin, W. R. & Pena, G. C. (2012). A new method for computing the drainage network based on raising the level of an ocean surrounding the terrain. *15th AGILE International Conference on Geographical Information Science*, pp. 391--407.

[27] Mahoney's, M. (2011). Data compression explained. http://mattmahoney.net/dc/dce.html. Accessado em 25 de julho de 2014.

[28] Mehlhorn, K. & Näher, S. (1995). Leda: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96--102.

[29] Meyer, U. & Zeh, N. (2012). I/o-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths. *ACM Transactions on Algorithms*, 8(3):22.

[30] Milutinovic, V.; Markovic, B.; Tomasevic, M. & Tremblay, M. (1996). The split temporal/spatial cache: Initial performance analysis. Em *Proceedings of SCIzzL-5, Santa Clara, California, USA, March*, pp. 63–69.

[31] Rabus, B.; Eineder, M.; Roth, A. & Bamler, R. (2003). The shuttle radar topography mission (STRM). Htpp://www2.jpl.nasa.gov/strm/. Accessed June 6, 2012.

[32] Smith, A. J. (1982). Cache memories. *ACM Comput. Surv.*, 14(3):473--530.

[33] Tarboton, D. (1997). A new method for the determination of flow directions and contributing areas in grid digital elevation models. *Water Resources Research*, 33:309–319.

[34] Technet (2013). Random and sequential data access. http://technet.microsoft.com/en-us/library/cc938619.aspx. Acessado em 23 de setembro de 2013.

[35] Union, I. T. (2009). Recommendation itu-r m.1677-1 – International Morse code. Technical report.

[36] Vitter, J. S. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv.*, 33(2):209--271.

[37] von Neumann, J. (1945). First draft of a report on the EDVAC. Technical report, University of Pennsylvania.

[38] Zhou, Y.; Bilas, A.; Jagannathan, S.; Xinidis, D.; Dubnicki, C. & Li, K. (2005). VI-attached database storage. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):35--50.

[39] Zhu, Y. & Huang, C. (2012). An improved median filtering algorithm for image noise reduction. *Physics Procedia*, 25(0):609 – 616. International Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao.

Apêndice A

Como utilizar a biblioteca *TiledMatrix*

A biblioteca *TiledMatrix* foi desenvolvida em *C++* e utiliza o algoritmo *lz4* desenvolvido na linguagem *c* para compressão e descompressão de blocos. As principais características da biblioteca são facilidade de uso, portabiblidade e eficiência.

1. *Facilidade de uso*
   Para usar a biblioteca para auxiliar uma determinada aplicação a lidar com quantidade enormes de dados, é necessário definir o tamanho do bloco, a quantidade de memória disponível para a aplicação, instanciar um objeto da classe *tiledMatrix.cpp*, escolher a política de substituição de blocos adequada para a aplicação e utilizar os métodos get e set para retornar um bloco e atribuir um valor a um bloco respectivamente.

2. *Portabilidade*
   A biblioteca é portável para a maioria dos sistemas operacionais e plataformas de *hardware*. Isso implica que o gerencimento de memória secundária é portável para a maioria das plataformas de computação.

3. *Eficiência*
   A biblioteca é eficiente visto que reduz o uso de CPU, a memória usada internamente, o espaço de disco e o número de operações de E/S.

Para acelerar o processo de busca de blocos entre a CPU e a memória principal existem as memórias *cache*. No entanto, nas arquiteturas atuais, entre o disco e a memória principal, não existe um mecanismo que torne o processo de busca de bloco ao disco eficiente. Para isto, a biblioteca *TiledMatrix* foi desenvolvida com o

propósito de tornar o processo de transferência de bloco para e do disco eficiente. Sendo assim, nesta seção descrevem-se os trechos de códigos mais relevantes para que o usuário possa utilizar a biblioteca.

Para utilizar a *TiledMatrix*, o usuário precisa instanciar um objeto do tipo *TiledMatrix* por meio do construtor, como se apresenta na Figura A.1. Como pode ser observado, no construtor da biblioteca exige-se que o usuário passe alguns parâmetros. A descrição desses parâmetros encontra-se na Tabela A.1. Para escolher a política de substituição, o usuário deve digitar "L" para escolher a política LRU, "F" para escolher FIFO e "R" para escolher a política aleatória.

```
1  /*
2   * nrows um numero inteiro de linhas.
3   * ncolumns um numero inteiro de colunas.
4   * tilesRows tamanho da linha do bloco.
5   * tilesColuns tamanho da coluna do bloco
6   * nTilesMemoria numero de blocos na memoria.
7   * policy politica de substituicao de blocos
8   * fileBase arquivo que armazena os valores das celulas da matriz de entrada
9   * nas celulas do blocos.
10  * fileMatrix arquivo para armazenar dados temporariamente.
11  */
12 tiledMatrix( int nRows, int nColumns, int tilesRows, int tilesColumns, int nTilesMemoria,
13        string policy, string fileBase, string fileMatrix="" );
```

Figura A.1. Construtor da biblioteca *TiledMatrix*.

Tabela A.1. Descrição dos parâmetros do método mostrado na Figura A.1.

Tipo	Nome	Descrição
int	*nRows*	Número de linhas que contem a matriz de entrada.
int	*nColums*	Número de colunas que contem a matriz de entrada.
int	*tilesRows*	Número de linhas que contem o bloco.
int	*tilesColums*	Número de colunas que contem o bloco.
int	*nTilesMemoria*	Número de blocos na memória.
string	*policy*	Política de substituição de blocos escolhida pelo usuário.
string	*fileBase*	Arquivo que armazena os valores das células da matriz de entrada nas células do blocos.
string	*fileMatrix*	Arquivo para armazenar dados temporariamente.

Após instanciar um objeto *TiledMatrix* o usuário deve ler a matriz de entrada. Os valores lidos devem ser atribuidos ao blocos por meio do método mostrado na Figura A.2. A Tabela A.2 mostra e descreve quais parâmetros devem ser passados para método.

A biblioteca disponibiliza também um método para inicializar todas as células de um bloco com um determinado valor como mostra na Figura A.3.

```
1  //Insere o elemento (i,j) na matriz com "valor"
2  template <class T>
3  void tiledMatrix<T>::set(int i,int j, T valor) {
4
5    int posTile = posTiles[i/tilesWidth][j/tilesWidth];
6
7    if (posTile == -1) {
8
9      loadTile(i/tilesWidth,j/tilesWidth);
10
11     posTile = posTiles[i/tilesWidth][j/tilesWidth];
12
13   }
14   sujo[i/tilesWidth][j/tilesWidth] = true;
15
16   timeStamps[posTile] = timeStamp++;
17
18   matrizes[posTile][ (i%tilesWidth)*tilesWidth + (j%tilesWidth) ] = valor;
19 }
```

Figura A.2. Carrega um bloco *i,j* na matriz e atribui um valor.

Tabela A.2. Descrição dos parâmetros do método mostrado na Figura A.2.

Tipo	Nome	Descrição
int	*i*	Posição da célula na i-ésima linha da matriz de entrada.
int	*j*	Posição da célula na j-ésima coluna da matriz de entrada.
int	*valor*	Valor da célula na matriz de entrada.

```
1  //Inicializa os blocos com um valor qualquer
2  template <class T>
3  void tiledMatrix<T>::set(T valor) {
4
5    for(int i =0;i<nrowsTiles;i++)
6
7      for(int j=0;j<ncolumnsTiles;j++) {
8
9        statusBloco[i][j] = STATUS_INICIALIZADO_VALOR_CONSTANTE;
10
11       sujo[i][j] = true;
12     }
13     valorSetadoParaTodosBlocos = valor;
14   }
```

Figura A.3. Atribui um valor qualquer a todos os blocos.

O método na Figura A.4 retorna um bloco de acordo com os parâmetros passados pelo usuário. A tabela A.4 descreve esses parâmetros.

A *TiledMatrix* fornece três status para o bloco, como pode ser observado na Figura A.5. O *STATUS_NÃO_INICIALIZADO* significa que o bloco ainda não foi inicializado com nenhum valor, o *STATUS_INICIALIZADO* significa que o bloco já está inicializado com um valor e pode ser carregado. Já o *STATUS_INICIALIZADO _VALOR_CONSTANTE* significa que todas as células do bloco receberam um valor constante definido pela aplicação.

Tabela A.3. Descrição dos parâmetros do método mostrado na Figura A.2.

Tipo	Nome	Descrição
int	*valor*	Valor atribuído a todas as células do bloco.

```
1  //Busca elemento da matriz
2  template <class T>
3  T tiledMatrix<T>::get(int i, int j) {
4    int posTile = posTiles[i/tilesWidth][j/tilesWidth];
5    if (posTile == -1) {
6      loadTile(i/tilesWidth,j/tilesWidth);
7      posTile = posTiles[i/tilesWidth][j/tilesWidth];
8    }
9    timeStamps[posTile] = timeStamp++;
10   return matrizes[posTile][ (i%tilesWidth)*tilesWidth + (j%tilesWidth) ];
11 }
```

Figura A.4. Retorna um elemento da matriz.

Tabela A.4. Descrição dos parâmetros do método mostrado na Figura A.4.

Tipo	Nome	Descrição
int	*i*	Posição da célula na i-ésima linha da matriz de entrada.
int	*j*	Posição da célula na j-ésima coluna da matriz de entrada.

Para carregar um bloco existe o método *loadTile2* como pode ser observado na Figura A.6. Inicialmente o método utiliza uma das políticas de substituição de bloco definida no construtor pelo usuário para remover um bloco da memória quando não houver mais espaço na memória para armazenar um bloco. Em seguida, atualiza-se o vetor *memBlocks* com os dados do bloco e a matriz *Position* com a posição em que o bloco foi inserido no vetor *memBlocks*. Caso o status do bloco seja *STATUS_INICIALIZADO*, o bloco é lido do disco comprimido e descomprimido e armazenado na memória. No entanto, caso o status do bloco seja *STATUS_NÃO_INICIALIZADO* o bloco é carregado na memória pela primeira vez. Caso o status do bloco seja *STATUS_INICIALIZADO _VALOR_CONSTANTE*, todas as células do bloco são inicializadas com um determinado valor definido pela aplicação. O status do bloco finalmente é marcado como *STATUS_INICIALIZADO* e atualizado para sujo. A descrição dos parâmetros do método está na Tabela A.5.

Tabela A.5. Descrição dos parâmetros do método mostrado na Figura A.6.

Tipo	Nome	Descrição
int	*i*	Posição da célula na i-ésima linha da matriz de entrada.
int	*j*	Posição da célula na j-ésima coluna da matriz de entrada.

```
1  // Indica que o bloco esta inicializado com um determinado valor
2  const char STATUS_INICIALIZADO = 1;
3  // Bloco ainda não foi inicializado
4  const char STATUS_NAO_INICIALIZADO = 2;
5  // Inicializa o bloco com um determinado valor
6  const char STATUS_INICIALIZADO_VALOR_CONSTANTE = 3;
```

Figura A.5. Status do bloco.

Para remover um bloco existe o método *removeTileDaMemoria* como pode ser observado na Figura A.7. Inicialmente, o método verifica se na posição *id* de *memblocks* existe um bloco para ser removido. Caso um bloco não esteja carregado na posição *id* uma mensagem é impressa para o usuário e o programa é interrompido. Caso exista um bloco na posição *id*, então bloco precisa ser removido. Primeiramente, encontra-se a posição do bloco no disco. Se o status do bloco for *STATUS_INICIALIZADO _VALOR_CONSTANTE* e *sujo*, todas as células do bloco deveram receber um valor definido pelo usuário. Caso o bloco esteja marcado como sujo o bloco deve ser gravado no disco com compressão e a posição do bloco na matriz *Position* é marcada com -1, indicando que o bloco não está mais carregado na memória. A Tabela A.6 descreve os parâmetros do método.

Tabela A.6. Descrição dos parâmetros do método mostrado na Figura A.7.

Tipo	Nome	Descrição
int	*i*	Posição da célula na i-ésima linha da matriz de entrada.
int	*j*	Posição da célula na j-ésima coluna da matriz de entrada.

Para carregar um bloco de maneira aleatória existe o método *carregaBlocoAleatorio* como pode ser observado na Figura A.8. Inicialmente, o método sorteia posições i, j até atingir o limite de blocos que cabem na memória. Em seguida, carrega os blocos cujas posições foram sorteadas no passo anterior. A descrição dos parâmetros está na Tabela A.7. Mais especificamente, este método foi utilizado para medir a taxa de compressão durante a transferência de blocos do disco para a memória da *TiledMatrix*. A descrição dos parâmetros do método está na Tabela A.7.

Tabela A.7. Descrição dos parâmetros do método mostrado na Figura A.7.

Tipo	Nome	Descrição
int	*numBlocos*	Número de blocos que cabem na memória.

Para carregar um bloco de maneira sequencial existe o método *carregaBlocoSequencial*, como pode ser observado na Figura A.9. O método carrega os blocos

```
428 //Private
429 template <class T>
430 void tiledMatrix<T>::loadTile2(int i,int j) {
431
432   //tilesCarregadosMemoria++;
433
434   int id = 0;
435
436   if (choosePolicy == "L"){
437     id = lruReplacementPolicy();
438     timeStamp[id] = timeStamp++; // Marca o timeStamp dele com um novo timeStamp
439   }
440
441   if (choosePolicy == "F")
442     id = fifoReplacementPolicy();
443
444   if (choosePolicy == "R")
445     id = randomReplacementPolicy();
446
447   tilesCarregados[id].first = i; //Agora o tile (i,j) estara carregado na posicao id
448   tilesCarregados[id].second = j;
449   posTiles[i][j] = id;
450
451   //Carrega o bloco (i,j) na posição id
452
453   unsigned long long tilePos = ((unsigned long long )i)*ncolumnsTiles+j;
454
455   if (statusBloco[i][j] == STATUS_INICIALIZADO) {
456
457     arquivoAssociado.seekg(tilePos*tamanhoMaximoBlocoComprimido, ios::beg);
458
459     arquivoAssociado.read(bufferTempCompressao, bytesCompressaoBloco[i][j]);
460
461     int bytesUsadosDescriptor = numSlices*sizeof(bytesUsadosSlice[0]);
462         memcpy(bytesUsadosSlice,bufferTempCompressao, bytesUsadosDescriptor );
463
464     offset[0] = 0;
465     for(int k=1;k<numSlices;k++) {
466       offset[k] =  offset[k-1] + bytesUsadosSlice[k-1];
467     }
468
469     #pragma omp parallel for
470     for(int k=0;k<numSlices;k++) {
471       LZ4_uncompress (bufferTempCompressao + bytesUsadosDescriptor + offset[k],
472       reinterpret_cast<char *>(matrizes[id] +  k*sliceSize), sliceSize*sizeof(T) );
473     }
474       sujo[i][j] = false;
475   } else
476     if (statusBloco[i][j] == STATUS_NAO_INICIALIZADO) {
477       tilesLidosMemoria++;
478     } else
479         if (statusBloco[i][j] == STATUS_INICIALIZADO_VALOR_CONSTANTE) {
480
481           for(int i=0;i<linhasTiles;i++)
482             for(int j=0;j<colunasTiles;j++)
483               matrizes[id][i*colunasTiles + j]  = valorSetadoParaTodosBlocos;
484
485               statusBloco[i][j] = STATUS_INICIALIZADO;
486               sujo[i][j] = true;
487         }
488         else {
489           cerr << "ERRO!!!" << endl;
490           exit(1);
491         }
492 }
493
```

Figura A.6. Método para carregar um bloco.

na memória de maneira consecutiva até atingir o limite de blocos que cabem na memória. Os parâmetros do método estão descritos na Tabela A.8

```
494 //Remove um bloco da memória
495 template <class T>
496 void tiledMatrix<T>::removeTileDaMemoria(int i,int j) {
497
498   int id = posTiles[i][j];
499
500   //Verifica se o bloco i,j está carregado na metriz posTiles
501   if (id==-1) {
502     cerr << "Erro inesperado: tentando armazenar em disco um tile
503          << "que nao foi carregado em memoria!" <<endl;
504     exit(1);
505   }
506
507   //Encontra a posição do bloco no disco
508   unsigned long long tilePos = ( (unsigned long long) i)*ncolumnsTiles+j;
509
510   if (statusBloco[i][j] == STATUS_INICIALIZADO_VALOR_CONSTANTE && sujo[i][j]) {
511     for(int i=0;i<linhasTiles;i++)
512       for(int j=0;j<colunasTiles;j++)
513   matrizes[id][i*colunasTiles + j]  = valorSetadoParaTodosBlocos;
514   }
515
516   //Caso o bloco tenha sido processado enquanto esteve na memória ele deve ser
517   //escrito no disco
518   if (sujo[i][j]) {
519
520     arquivoAssociado.seekp(tilePos*tamanhoMaximoBlocoComprimido,ios::beg);
521     // O bloco é gravado no disco utilizando a compressão paralela
522     #pragma omp parallel for
523     for(int k=0;k<numSlices;k++) {
524       bytesUsadosSlice[k] = LZ4_compress(reinterpret_cast<char *>(matrizes[id] +k*sliceSize  ),
525       bufferTempCompressaoSlice[k], sliceSize*sizeof(T) );
526     }
527
528     int bytesUsadosDescriptor = numSlices*sizeof(bytesUsadosSlice[0]);
529     memcpy(bufferTempCompressao,bytesUsadosSlice, bytesUsadosDescriptor );
530
531     bytesCompressaoBloco[i][j] = bytesUsadosDescriptor;
532     for(int k=0;k<numSlices;k++)
533       bytesCompressaoBloco[i][j] += bytesUsadosSlice[k];
534
535     offset[0] = 0;
536     for(int k=1;k<numSlices;k++)
537       offset[k] = offset[k-1] + bytesUsadosSlice[k-1];
538
539     #pragma omp parallel for
540     for(int k=0;k<numSlices;k++) {
541       memcpy( bufferTempCompressao + bytesUsadosDescriptor + offset[k],
542       bufferTempCompressaoSlice[k], bytesUsadosSlice[k] );
543     }
544
545     arquivoAssociado.write( bufferTempCompressao ,bytesCompressaoBloco[i][j] );
546
547     sujo[i][j] = false;
548
549     statusBloco[i][j] = STATUS_INICIALIZADO;
550   }
551   //Fala que ele nao estah mais carregado...
552   posTiles[tilesCarregados[id].first][tilesCarregados[id].second] =  -1;
553
554 }
```

Figura A.7. Método para remover um bloco.

Tabela A.8. Descrição dos parâmetros do método mostrado na Figura A.7.

Tipo	Nome	Descrição
int	*numBlocos*	Número de blocos que cabem na memória.

```
1  // Carrega um bloco na memória de maneira aleatória
2  template <class T>
3  void tiledMatrix<T>::carregaBlocoAleatorio( int numBlocos) {
4
5    assert(numBlocos>0);
6
7    srand (time(NULL));
8
9
10   assert(nrowsTiles*ncolumnsTiles >= numBlocos);
11
12
13   std::set< pair<int,int> > st;
14   while(st.size() < numBlocos)
15     st.insert(make_pair(rand() % (nrowsTiles), rand() % (ncolumnsTiles)));
16
17   vector< pair<int,int> > indices(st.begin(),st.end());
18
19   for(int i=0;i<indices.size();i++ )
20     loadTile(indices[i].first ,indices[i].second);
21 }
```

Figura A.8. Método para carregar um bloco de maneira aleatória.

```
1  template <class T>
2  void tiledMatrix<T>::carregaBlocoSequencial(int numBlocos) {
3
4    assert(numBlocos>0);
5
6    for( int i=0;i<nrowsTiles;i++ ) {
7      for( int j=0;j<ncolumnsTiles;j++ ) {
8
9        loadTile(i,j);
10       numBlocos--;
11
12       if (numBlocos == 0) return;
13     }
14   }
15 }
```

Figura A.9. Método para carregar um bloco de maneira sequencial.