THIAGO LUANGE GOMES

DETERMINAÇÃO DA REDE DE DRENAGEM EM GRANDES TERRENOS ARMAZENADOS 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
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
G633d
2013

Gomes, Thiago Luange, 1988-
Determinação da rede de drenagem em grandes terrenos armazenados em memória externa / Thiago Luange Gomes. – Viçosa, MG, 2013.
x, 74f. : il. (algumas color.) ; 29cm.

Textos em português e inglês
Orientador: Marcus Vinícius Alvim Andrade
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 71-74

1. Algorítimos computacionais. 2. Sistemas de informação geográfica. 3. Bacias hidrográficas - Programas de computador. 4. Discos rígidos (Computação). I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.1

THIAGO LUANGE GOMES

DETERMINAÇÃO DA REDE DE DRENAGEM EM GRANDES TERRENOS ARMAZENADOS 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: 02 de abril de 2013.

Luciana Brugiolo Gonçalves

Tiago Garcia de Senna Carneiro

Marcus Vinícius Alvim Andrade
(Orientador)

Eu dedico essa dissertação primeiramente a minha família, em especial aos meus pais José Renato Gomes e Maria Alice Soares Gomes, que me deram apoio e incentivo durante todo o processo, dedico também a todos as pessoas que participaram diretamente e indiretamente do processo, Prof. Marcus Vinícius Alvim Andrade, Prof. Salles V. G. Magalhães, Guilherme C. Pena e muitos outros amigos.

AGRADECIMENTOS

Quando eu olho para trás, nos últimos dois anos, eu lembro de um monte de pessoas que influenciaram meu trabalho e minha vida. Primeiro de tudo, quero agradecer ao meu orientador Prof. Marcus Vinícius Alvim Andrade, foi um prazer trabalhar em seu grupo e seu estilo de pesquisa me influenciou muito. Ele me apresentou aos temas, hidrografia e algoritmos para memória externa que renderam bons frutos. Segundo, eu quero agradecer ao Prof. Salles V. G. Magalhães que ajudou muito com seu conhecimento, trabalho e incentivo. Meus agradecimentos especiais são para Guilherme C. Pena e os integrantes do grupo de pesquisa do Prof. Marcus por ouvir idéias e problemas da vida real e científica e ajudar na construção do método EMFlow. Eu também quero agradecer aos outros professores que ministraram matérias importantes para minha formação no mestrado e ao departamento por fornecer apoio e condições extraordinárias de trabalho.

Deixo um agradecimento necessário aos órgãos de financiamento CAPES, FAPEMIG e CNPq que tornaram possível o desenvolvimento do projeto.

Um agradecimento muito especial a todos meus amigos . Vocês sempre acreditara em mim e me ajudaram a não perder as perspectivas. No entanto, eu sou de longe mais grato à minha família e em especial aos meus pais, José Renato Gomes e Maria Alice Soares Gomes que me deram mais amor e incentivo do que eu posso pagar de volta!

"Rose knew almost everything that water can do, there are an awful lot when you think what."

(Gertrude Stein, The World is Round)

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

GOMES, Thiago Luange, M.Sc., Universidade Federal de Viçosa, abril de 2013. **Determinação da rede de drenagem em grandes terrenos armazenados em memória externa.** Orientador: Marcus Vinícius Alvim Andrade.

Este trabalho apresenta um algoritmo muito eficiente, chamado *EMFlow*, para o cálculo da rede de drenagem em grandes terrenos armazenados em memória externa. A rede de drenagem retrata o caminho que a água segue através do terreno (direção de fluxo) e a quantidade de água que flui por cada célula do terreno (fluxo acumulado). Como é conhecido, devido ao rápido aumento da disponibilidade de dados de alta resolução da superfície terrestre, os algoritmos de memória interna não são capazes de processar de forma eficiente esse volume de dados na maioria dos computadores. Portanto, otimizar os algoritmos simultaneamente para a movimentação de dados e processamento tem sido um desafio para os sistemas de informação geográfica (SIG). O *EMFlow* calcula a direção de fluxo usando uma adaptação do método *RWFlood* que utiliza um processo de inundação para obtenção da direção de fluxo e o fluxo acumulado é calculado com base em um método bastante eficiente proposto por Haverkort e Janssen (2012). Para reduzir o número total de operações de entrada e saída, o *EMFlow* adota uma nova estratégia de subdivisão do terrenos em ilhas que são processadas separadamente e agrupa as células do terreno em blocos que são armazenados em uma estrutura de dados especial gerenciada como uma memória *cache*. O tempo de execução do *EMFlow* foi comparado com os dois mais recentes e eficientes métodos descritos na literatura: *TerraFlow* e *r.watershed.seg* e foi, em média, 27 vezes mais rápido que ambos. Como o processamento de grandes terrenos pode levar horas, essa melhora é muito significativa.

Abstract

GOMES, Thiago Luange, M.Sc., Universidade Federal de Viçosa, April, 2013. **Computing the drainage network on massive terrains using external memory.** Adviser: Marcus Vinícius Alvim Andrade.

This work presents a very efficient algorithm, named *EMFlow*, and its implementation to compute the drainage network on huge terrains stored in external memory. The drainage network of a terrain delineates the path that water flows through the terrain (the flow direction) and the amount of water that flows into each terrain cell (the flow accumulation). As it is known, due to the fast increase in the volume of high resolution terrestrial data available, the internal memory algorithms do not run well for huge terrains on most computers and, thus, optimizing the massive data processing algorithm simultaneously for data movement and computation has been a challenge for GIS (Geographic Information System). In *EMFlow*, the flow direction is computed using an adaptation of method *RWFlood* which uses a flooding process to obtain this direction and the flow accumulation is computed based on a very fast method proposed by Haverkort and Janssen (2012). To reduce the total number of I/O operations, *EMFlow* adopts a new strategy to subdivide the terrains into islands which are processed separately and the terrain cells are grouped into blocks, which are stored in a special data structure managed as a cache memory. The *EMFlow* execution time was compared against the two most recent and most efficient published methods: *TerraFlow* and *r.watershed.seg* and it was, in average, 27 times faster than both methods. Since processing large datasets can take hours, this improvement is very significant.

1. Introdução Geral

O avanço da tecnologia do sensoriamento remoto tem produzido um enorme volume de dados sobre a superfície terrestre. O projeto *SRTM (NASA's Shuttle Radar Topography Mission)*, por exemplo, mapeou 80% da superfície da terra com resoluções de 30 metros, formando o mais completo banco de dados de alta resolução da terra, que possui 10 terabytes de dados [Arge et al., 2003].

Segundo Câmara et al. [2001] os métodos de aquisição de dados podem ser: (a) por pontos amostrados com espaçamento irregular e regular ou (b) por mapa de isolinhas. A Figura 1.1 mostra vários tipos diferentes de amostragem por pontos e a Figura 1.2 ilustra o caso de um mapa de isolinhas.

Neste processo de coleta de informação, a superfície da Terra pode ser representada de forma aproximada utilizando um modelo digital de elevação (MDE), que armazena as elevações de pontos amostrados na superfície terrestre. Essas amostras são armazenadas em estruturas de dados definidas de forma a possibilitar uma ma-

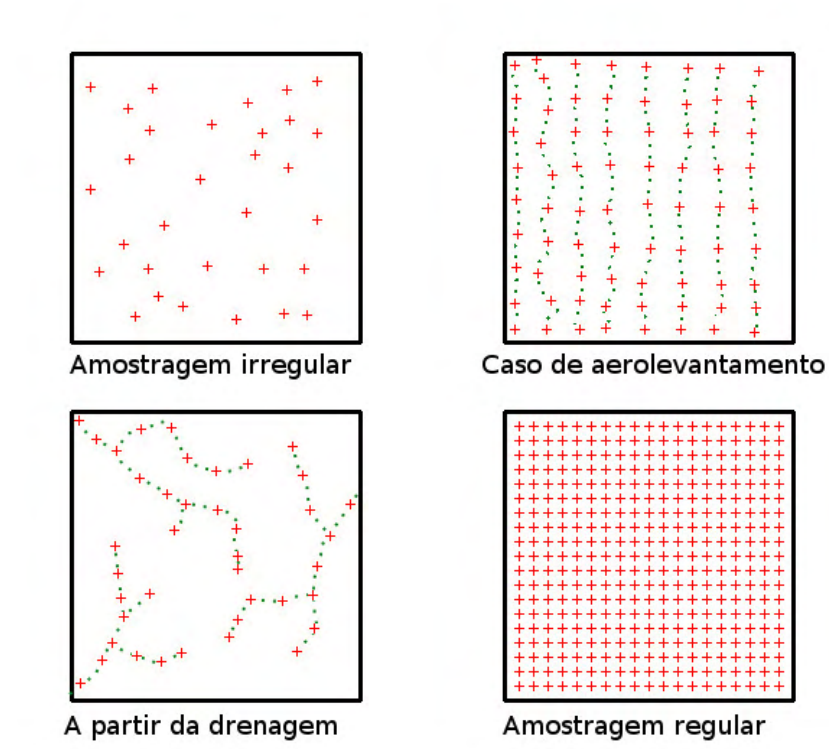

Figura 1.1. Tipos de amostragem por pontos. Fonte: Câmara et al. [2001].

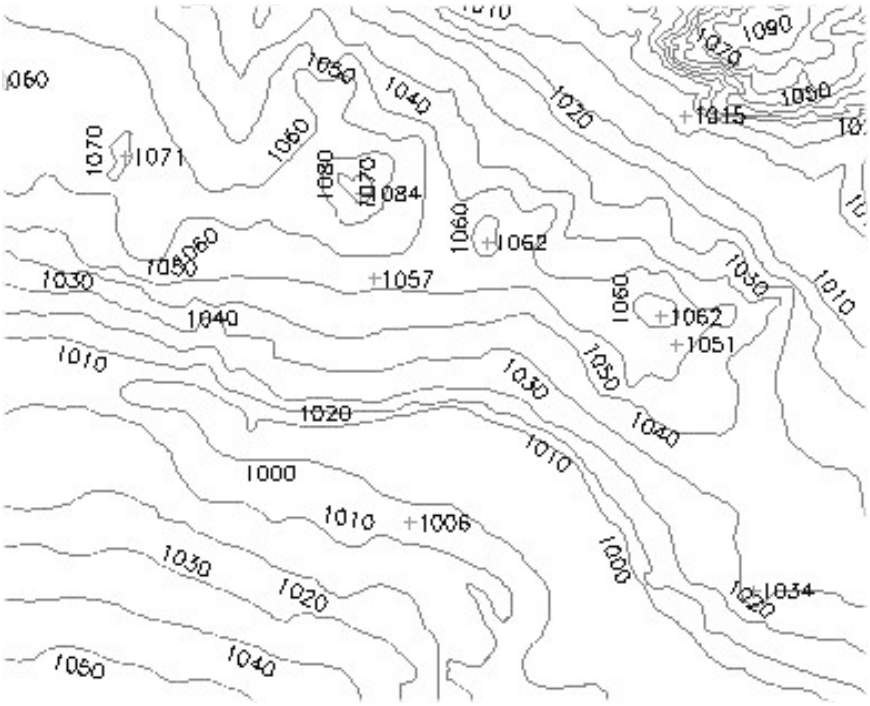

Figura 1.2. Exemplo de mapa de isolinhas. Fonte: SPRING - DPI/INPE [2012].

nipulação eficiente pelos algoritmos de análise contidos nos Sistemas de Informação Geográfica (SIG). As estruturas de dados mais utilizadas são os modelos de grade regular e os modelos de malha triangular(*TIN - Triangulated Irregular Network*).

A grade regular é um modelo digital que aproxima superfícies através de um poliedro de faces retangulares. Os vértices desses poliedros podem ser os próprios pontos amostrados, caso estes tenham sido adquiridos nas mesmas localizações xy que definem a grade desejada [Câmara et al., 2001]- veja a Figura 1.3. Uma malha triangular é um conjunto de poliedros cujas faces são triângulos e os vértices do triângulo são geralmente os pontos amostrados da superfície. Esta modelagem, considerando as arestas dos triângulos, permite que as informações morfológicas importantes, como as descontinuidades representadas por feições lineares de relevo (cristas) e drenagem (vales), sejam consideradas durante a geração da grade triangular, possibilitando assim, modelar a superfície do terreno preservando as feições geomórficas da superfície [Câmara et al., 2001]. Veja a Figura 1.4.

A análise de elementos hidrográficos é uma aplicação sobre modelos digital de elevação que vem sendo realizada há anos. O estudo de elementos hidrográficos pode ser uma ferramenta importante na análise de gestão de riscos de enchentes, previsão de vazão e no estudo geral dos processos geomorfológicos, podendo contribuir para

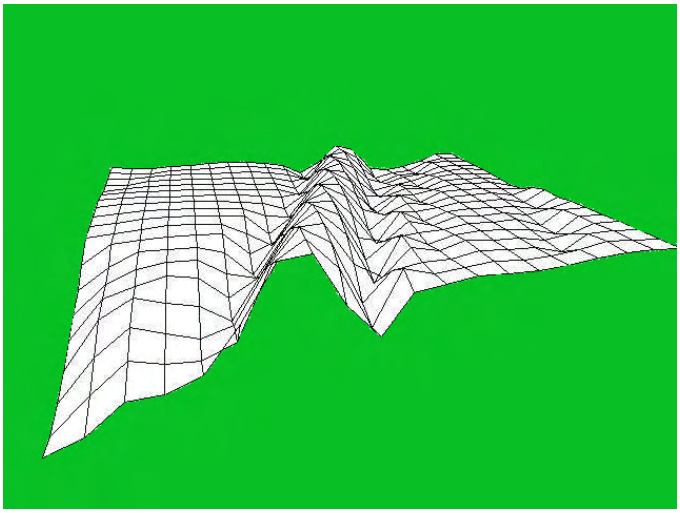

Figura 1.3. Grade regular. Fonte: Brostuen & Cox [2000]

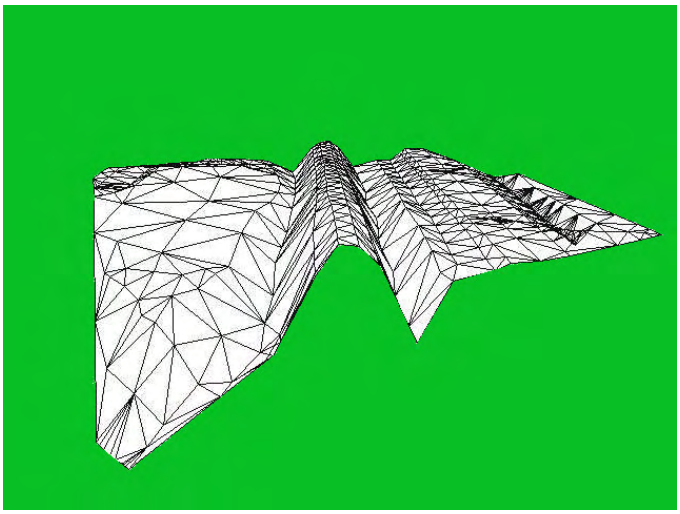

Figura 1.4. Malha triangular. Fonte: Brostuen & Cox [2000]

obtenção de previsões climáticas mais confiáveis [Driemel et al., 2011].

A rede de drenagem é um componente fundamental na análise de elementos hidrográficos, pois é essencial para determinação das propriedades hidrográficas de um terreno. A rede de drenagem é composta pela direção do fluxo de escoamento e pelo fluxo acumulado em cada ponto (célula) do terreno. Intuitivamente, a direção de fluxo corresponde ao caminho que a água deve seguir ao longo do terreno e o fluxo acumulado é a quantidade de água que alcança cada célula supondo que o terreno recebe um determinado volume de água uniformemente distribuído sobre a sua superfície [Moore et al., 1991]. Veja a Figura 1.5.

Segundo Arge et al. [2003], pode-se definir formalmente o problema da direção

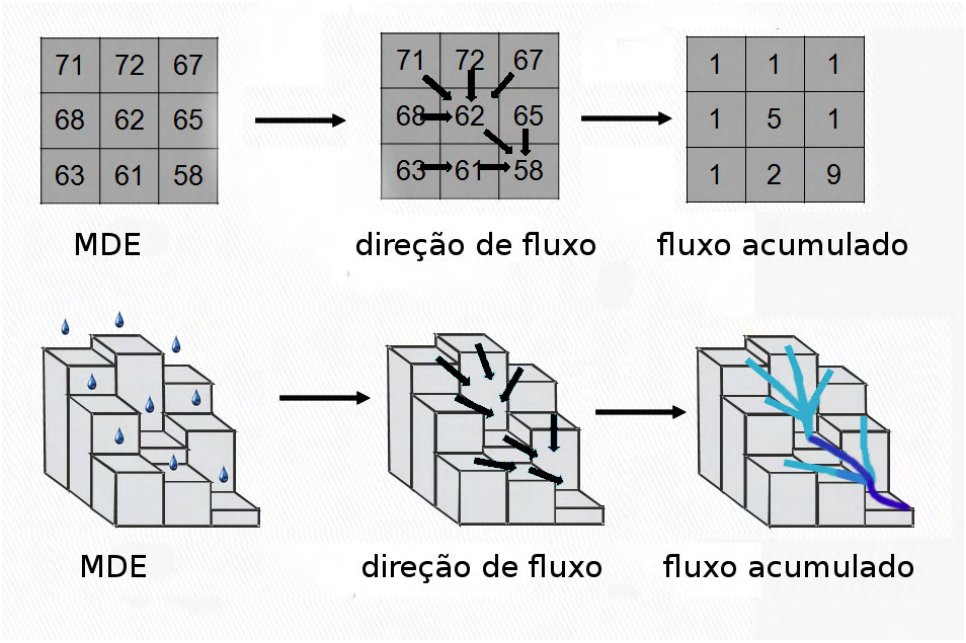

Figura 1.5. Exemplificação da direção de fluxo e fluxo acumulado.

de fluxo como a tarefa de atribuir direções de fluxo para todas as células do terreno de tal modo que as três condições a seguir sejam satisfeitas:

1. Cada célula tem pelo menos uma direção do fluxo;
2. Não existem caminhos de fluxo cíclicos, e
3. Cada célula no terreno possui um caminho de fluxo para a borda do terreno

Considerando que atualmente há um grande volume de dados em alta resolução sobre a superfície terrestre, os algoritmos para cálculo da rede de drenagem podem requerer muito espaço de armazenamento para manipular os terrenos. Por exemplo, um terreno de $100km \times 100km$ amostrado de forma regular com resolução de $1m$ resulta em 10^{10} pontos, o que requer mais de 18GB de espaço de armazenamento caso a elevação em cada ponto seja armazenada utilizando 2 *bytes*. Geralmente, não é possível armazenar este volume de dados na memória interna da maioria dos computadores.

Assim, esse enorme volume de dados requer o desenvolvimento (ou adaptação) de algoritmos para o processamento de dados em memória externa (geralmente discos) onde o acesso aos dados é bem mais lento do que na memória interna [Vitter, 2001; Arge et al., 2003; Dementiev et al., 2005; Magalhães et al., 2012a].

1.1 Algoritmos para memória externa

Enormes conjuntos de dados surgem naturalmente em muitos domínios: Por exemplo, bases de dados espaciais que representam toda a superfície terrestre, na computação de cenas gráficas, simulações precisas do clima que trabalham com *Petabytes* de dados. Estes exemplos são apenas uma amostra das numerosas aplicações que têm de processar uma quantidade enorme de dados.

As memórias internas dos computadores podem manter apenas uma pequena fração do grande conjunto de dados durante o processamento, e portanto, os aplicativos precisam acessar a memória externa (por exemplo, discos rígidos) com muita frequência. Tais acessos são muito mais lentos do que o acesso a memória principal, e assim, os acessos ao disco (operações de entrada e saída) tornam-se o principal gargalo nessas aplicações.

A razão para a alta latência é a natureza mecânica do acesso ao disco. Os dados são armazenados na superfície de um disco magnético, cuja velocidade de rotação variam de 4.200 a 15.000 rotações por minuto (RPM) [Vitter, 2001; Dementiev, 2007]. A informação é armazenada aplicando-se um campo magnético a partir de uma cabeça de leitura/escrita posicionada muito perto da superfície e seguindo uma trajetória concêntrica chamada faixa do disco. 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 estejam sob a cabeça. Depois disso, é necessário esperar até que o segmento fique sob a cabeça (latência rotacional). Apenas a partir deste momento a leitura ou escrita é possível, veja a Figura 1.6. O tempo total necessário para encontrar a posição no disco é chamado de tempo de busca (*seek*), a latência média para discos modernos é cerca de 3-10 ms [Vitter, 2001; Dementiev, 2007]. O tempo de busca depende da velocidade de rotação e dificilmente pode ser reduzido devido a natureza mecânica da tecnologia dos discos rígidos. Note-se que depois de encontrar a requerida posição 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.

Para aumentar o espaço de memória interna disponível para uma aplicação os sistemas operacionais implementam o mecanismo chamado de memória virtual que consiste em mapear os arquivos da memória externa (arquivo de página) para endereços virtuais da memória principal. Esta ideia suporta o modelo de que um programa tem uma memória principal infinitamente grande com custo de acesso

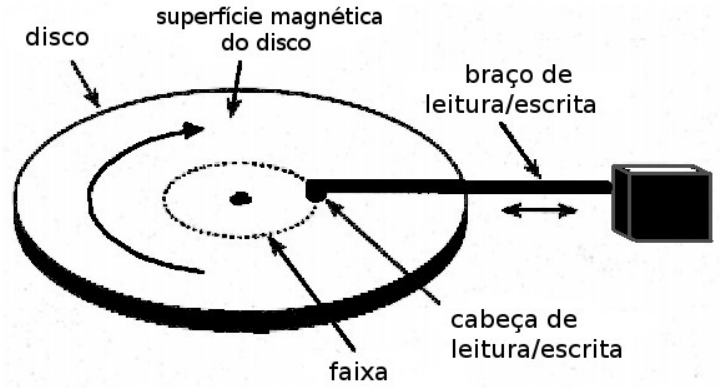

Figura 1.6. Unidade de disco magnético. Fonte: Vitter [2001]

aleatório uniforme. Este modelo tem sido utilizado como principal arquitetura de computador para o desenvolvimento de linguagens de programação.

Quando o padrão de acesso é simples, o sistema operacional adota algumas estratégias para tentar prever os próximos endereços a serem acessados e carrega esses dados na memória antes que eles sejam efetivamente requisitados. Porém, essa estratégia não é muito eficiente quando o padrão de acesso não segue uma sequência pré-estabelecida; Nesses casos, é necessário utilizar algum mecanismo para auxiliar o processo de gerenciamento dos acessos à memória externa de forma a reduzir o número de acessos e melhorar o desempenho das aplicações.

1.2 Objetivos

O objetivo geral do trabalho foi desenvolver um método capaz de calcular a rede de drenagem em grandes terrenos, isto é, calcular a direção de fluxo e fluxo acumulado para terrenos que não possam ser armazenados na memória interna do computador.

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

- Transformar as estruturas de dados ultilizadas pelos algoritmos a fim de diminuir a latência dos acessos feitos pelos algoritmos.
- Avaliar e produzir melhorias nos algoritmos para reduzir o número de acessos ao disco.
- Realizar comparações de tempo e resultado em relação aos algoritmos descritos na literatura para memória externa.

1.3 Resultados obtidos

Nos capítulos 2, 3 e 4 são apresentados os artigos que descrevem os resultados obtidos neste trabalho. Mais especificamente, o capítulo 2 se refere ao artigo *Determinação da rede de drenagem em grandes terrenos armazenados em memória externa* aceito no GEOINFO 2012 (XII Brazilian Symposium on GeoInformatics) [Gomes et al., 2012b]. O capítulo 3 se refere ao artigo *Computing the drainage network on huge grid terrains* apresentado no BIGSPATIAL 2012 (The First ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BIGSPATIAL) 2012) [Gomes et al., 2012a] que foi realizado em conjunto com a 20th ACM SIGSPATIAL (International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2012)) e o capítulo 4 ao artigo *Efficiently computing the drainage network on massive terrains using external memory flooding process* submetido ao Journal Computers & Geosciences [Gomes et al., 2013]. A seguir é feito uma breve descrição do conteúdo destes artigos.

O artigo incluído no capítulo 2 descreve uma primeira versão do método *EMFlow* para cálculo da rede de drenagem em memória externa, onde a ideia básica é adaptar o método *RWFlood* [Magalhães et al., 2012b] que obtém a rede de drenagem de um terreno simulando o processo de inundação do terreno supondo que a água entra no terreno pela sua borda vindo da parte externa. Neste caso, é importante observar que o caminho que a água percorre à medida que vai inundando o terreno é o mesmo caminho que a água percorreria se fosse proveniente da chuva que cai sobre o terreno e escoa descendentemente.

O método *RWFlood* original processa o terreno, representado por uma matriz, acessando essa matriz de forma não sequencial e, portanto, o processamento de grandes terrenos armazenados em memória externa não é eficiente.

Para contornar estes problemas diminuindo o número de acessos ao disco, foi elaborado o método *EMFlow* que adapta o método *RWFlood* de forma que os acessos realizados à matriz sejam gerenciados por uma biblioteca denominada *TiledMatrix* [Magalhães et al., 2012a], que é capaz de armazenar e gerenciar grandes matrizes em memória externa. Na verdade, a ideia básica desta adaptação é modificar a forma de gerenciamento da memória (reorganizando a matriz) para tirar proveito da localidade espacial de acesso. Esta primeira versão foi comparada com *TerraFlow* [Arge et al., 2003] e *r.watershed.seg* [GRASS Development Team, 2010], os principais métodos descritos na literatura. A proposta foi cerca de 10 vezes mais eficiente que o *TerraFlow* para os terrenos grandes e o *r.watershed.seg* não foi capaz de processar

os terrenos em um tempo razoável.

O artigo referente ao capítulo 3 descreve uma melhoria importante no método *EMFlow* onde foi adotada uma estratégia (original) de divisão do terreno que gera uma considerável melhoria no tempo de processamento.

A adaptação apresentada no capítulo 2 pode não ser muito eficiente quando o processo de inundação não mantém um padrão de acesso espacial, ou seja, em um dado momento as células acessadas não estão, na sua maioria, próximas umas das outras na matriz. Tal fato ocorre, por exemplo, com terrenos que possuem duas regiões distintas com mesma elevação, fazendo com que o processo de inundação fique alternando entre as regiões. Para contornar esse problema foi desenvolvido uma estratégia de divisão do terreno que se aproveita do fato do método *EMFlow*, assim como o *RWFlood*, supor que o terreno é uma ilha cercada por água em todos os lados sendo que o nível do "oceano" é iterativamente elevado. À medida que o nível deste oceano sobe, o terreno se torna um conjunto de ilhas . Veja a Figura 3.2. Note que, uma ilha é um conjunto de células conectadas que ainda não foram inundadas. Assim, as ilhas podem ser identificadas computando as componentes conexas de células não inundadas e podem ser processadas separadamente devido ao fato de o processo de inundação em uma ilha não afetar outra ilha.

Para existir uma ilha é necessário ter um grupo de células inundadas (processadas) cercando um grupo de células não inundadas (não processadas). No entanto, o processo de identificação baseado na obtenção de componentes conexas seria muito ineficiente principalmente quando o terreno não pode ser armazenado na memória interna. Assim, o algoritmo adota uma estratégia menos precisa onde as ilhas são identificadas usando um terreno com menor resolução que pode ser processado rapidamente na memória interna. Isto é, no terreno com menor resolução uma célula representa um bloco de células do terreno original, sendo que ela é considerada processada se todas as células do bloco já tiverem sido processadas e não processada se pelo menos uma célula do bloco ainda não tiver sido processada. Note que, as ilhas identificadas no terreno com menor resolução podem ser processadas separadamente e que ilhas do terreno original podem ser identificadas como conexas, mas o resultado não muda pois duas ou mais ilhas podem ser processadas ao mesmo tempo. Novamente, essa versão melhorada foi comparada com o *TerraFlow* e *r.watershed.seg* e ela foi cerca de 17 vezes mais eficiente que o *TerraFlow* para os terrenos grandes e o *r.watershed.seg*, como já foi falado, não foi capaz de processar os terrenos em um tempo razoável.

Finalmente, o capítulo 4 descreve uma extensão do artigo apresentado no capítulo 3, onde foi incluído ao método *EMFlow* um eficiente algoritmo para cálculo

de fluxo acumulado, cuja implementação foi realizada baseada no método proposto por Haverkort & Janssen [2012]. A utilização deste método produziu melhorias significativas no tempo de processamento.

Os algoritmos para fluxo acumulado nas versões do método *EMFlow* apresentadas no capítulo 2 e 3 são baseados no método de ordenação topológica, tal método se mostra muito eficiente quando o terreno pode ser processado na memória interna do computador. Mas, quando o terreno não pode ser armazenado na memória interna, esse método pode exigir muitas operações de entrada e saída devido ao acesso não sequencial apresentado pelo método.

Para reduzir o número de acessos ao disco realizados pelo método de ordenação topológica, o fluxo passou a ser obtido utilizando uma implementação do algoritmo proposto por Haverkort & Janssen [2012]. A ideia principal deste método é subdividir o terreno em blocos que possam ser armazenados na memória interna, processar os blocos de forma isolada e unir os resultados de modo que o fluxo acumulado fique correto. Veja a Figura 4.3. Conforme demonstrado por Haverkort & Janssen [2012], este método é da ordem de $O(Scan(N))$ operações de entrada e saída no pior caso, sendo o método de ordenação topológica da ordem de $O(Sort(N))$. De acordo com os testes, essa versão do *EMFlow* foi em média 27 vezes mais eficiente que os métodos *TerraFlow* e *r.watershed.seg*.

2. Determinação da rede de drenagem em grandes terrenos armazenados em memória externa

Abstract

An important component of terrain analysis in GIS is the computation of the drainage network. But, this is not a trivial task for huge terrains stored in the external memory since, in this case, the time required to access the external memory is much larger than the internal processing time. Thus, such external memory algorithms should be designed under a computational model where the main cost is the number I/O operations instead of CPU time. In this context, this paper presents an efficient algorithm for computing the drainage network in huge terrains where the main idea is to adapt the method *RWFlood* [Magalhães et al., 2012b], designed to compute the drainage network in the internal memory, to reduce the number of disk access. The proposed method was compared against some classic methods as *TerraFlow* and *r.watershed.seg* and, as the tests shown, it was much faster (in many cases, more than 30 times) than both methods.

2.1 Introdução

O avanço da tecnologia do sensoriamento remoto tem produzido um enorme volume de dados sobre a superfície terrestre. O projeto *SRTM (NASA's Shuttle Radar Topography Mission)*, por exemplo, mapeou 80% da superfície da terra com resoluções de 30 metros, formando o mais completo banco de dados de alta resolução da terra, que possui 10 terabytes de dados [Arge et al., 2003].

Neste processo de coleta de informação, a superfície da Terra é representada de forma aproximada utilizando um modelo digital de elevação (MDE), que armazena as elevações de pontos amostrados na superfície terrestre. Essas amostras podem ser obtidas de maneira irregular e serem armazenadas como uma rede triangular irregular (*TIN - Triangulated Irregular Network*) ou de maneira regular, sendo armazenados em uma matriz [Felgueiras, 2001] e, em ambas as formas, elas necessitam de muito espaço de armazenamento, geralmente maior do que se tem disponível na

memória interna da maioria dos computadores atuais. Por exemplo, um terreno de $100km \times 100km$ amostrado de forma regular com resolução de $1m$ resulta em 10^{10} pontos, o que requer mais de 18GB de espaço de armazenamento caso a elevação em cada ponto seja armazenada utilizando 2 *bytes*. Esse enorme volume de dados requer o desenvolvimento (ou adaptação) de algoritmos para o processamento de dados em memória externa (geralmente discos) onde o acesso aos dados é bem mais lento do que na memória interna [Dementiev et al., 2005]. Então, os algoritmos para processamento de grande volume de dados (armazenados em memória externa) precisam ser projetados e analisados utilizando um modelo computacional que considera não apenas o uso da CPU, mas também o tempo de acesso ao disco. Um desses modelos, proposto por Aggarwal & Vitter [1988], analisa a complexidade dos algoritmos com base no número de operações de acesso a disco executadas. Esse modelo será apresentado com maiores detalhes na seção 2.2.2.

Uma importante aplicação na área de sistemas de informação geográfica (SIGs) relacionada a modelagem de terrenos é a determinação das estruturas hidrológicas tais como a direção de fluxo, o fluxo acumulado, bacias de acumulação, etc. Essas estruturas são utilizadas no cálculo de atributos do terreno, tais como convergência topográfica, rede de drenagem, bacias hidrográficas,etc. Que por sua vez, são usadas para modelar vários processos hidrológicos, geomorfológicos e biológicos no terreno, como água do solo, potencial de erosão, distribuição de espécies e plantas [I.Moore et al., 1991].

Este trabalho apresenta o método *EMFlow* para a obtenção da rede de drenagem em grandes terrenos representados por matriz de elevação armazenadas em memória secundária. A ideia básica deste novo método é adaptar o algoritmo *RWFlood* [Magalhães et al., 2012b], alterando a forma como os dados em memória externa são acessados. Para isto, é utilizada uma biblioteca que gerencia as transferências de dados entre as memórias interna e externa, buscando diminuir o número de acessos ao disco.

2.2 Referencial teórico

2.2.1 Determinação da rede de drenagem

Conforme mencionado na seção 2.1, uma importante aplicação em SIGs é a determinação das propriedades hidrográficas de um terreno, sendo que o elemento básico da hidrografia é a rede de drenagem que é composta pela direção do fluxo de escoamento e pelo fluxo acumulado em cada ponto (célula) do terreno. Intuitivamente, a

direção de fluxo corresponde ao caminho que a água deve seguir ao longo do terreno e o fluxo acumulado é a quantidade de água que alcança cada célula supondo que o terreno recebe um determinado volume de água uniformemente distribuído sobre a sua superfície [Moore et al., 1991].

A direção de fluxo pode ser modelada de duas formas: fluxo em direção única (SFD - *Single-flow-direction*) em que o fluxo é direcionado para uma única célula vizinha que possua o menor valor de elevação e que seja menor do que a elevação da célula em questão e o fluxo em várias direções (MFD - *Multi-flow-directions*) onde o fluxo é dividido proporcionalmente em função da diferença de elevação entre a célula em questão e as suas vizinhas que possuam elevação menor [Arge et al., 2003]. Do ponto de visto computacional, a escolha dos modelos SFD ou MFD não é crítica, pois a direção de fluxo pode ser computada com uma mesma complexidade assintótica usando ambos modelos. No entanto, do ponto de vista prático, esta escolha é importante, pois o modelo SFD geralmente produz uma rede de fluxo com um menor número de trechos convergentes que são mais longos, enquanto o modelo MFD produz uma rede mais difusa, com um maior número de trechos mais curtos [Arge et al., 2003]. Neste trabalho será adotado o modelo SFD por questão de simplicidade.

Segundo Arge et al. [2003], pode-se definir formalmente o problema da direção de fluxo como a tarefa de atribuir direções de fluxo para todas as células do terreno de tal modo que as seguintes três condições sejam satisfeitas:

1. Cada célula tem pelo menos uma direção do fluxo;
2. Não existem caminhos de fluxo cíclicos, e
3. Cada célula no terreno possui um caminho de fluxo para a borda do terreno

Há diversos métodos para a obtenção da rede de drenagem [O'Callaghan & Mark, 1984; Jenson & Domingue, 1988; Soille & Gratin, 1994; Tarboton, 1997] e, conforme descrito pelos autores, a maior dificuldade neste processo é a ocorrência de células onde não é possível determinar a direção de fluxo diretamente porque ou a célula é um mínimo local ou pertence a uma região horizontalmente plana. Um mínimo local é uma célula do terreno cuja elevação é menor ou igual à elevação de todas as suas vizinhas e uma região plana corresponde a um conjunto de células adjacentes com uma mesma elevação. As células situadas na borda de uma região plana e que não são mínimos locais, isto é, que possuem pelo menos uma célula vizinha com elevação menor que a sua são denominadas pontos de escoamento.

Assim, as regiões planas podem ser classificadas de duas formas: um *platô*, que é uma região plana que possui pelo menos um ponto de escoamento e um *fosso*, que é uma região plana sem ponto de escoamento [Arge et al., 2003; Jenson & Domingue, 1988]. Intuitivamente, o fluxo em um platô é orientado na direção dos pontos de escoamento e, em um fosso, a água se acumula até que ela "transborde" escoando através das células vizinhas com menor elevação.

Vários métodos de obtenção da rede de drenagem como, por exemplo, os apresentados em Soille & Gratin [1994]; Metz et al. [2011]; Danner et al. [2007], eliminam os fossos realizando um pré-processamento do terreno para preenchê-lo até que um ponto da grade com valor de elevação menor do que a elevação máxima do fosso seja encontrado. Ou seja, cada fosso é preenchido com o objetivo de transformá-lo em um platô.

O processamento de um platô, em geral, é realizado após uma primeira etapa da obtenção das direções de fluxo quando a direção de fluxo em todas as células é conhecida, exceto nas células dos platôs. Então, conhecendo-se os pontos de escoamento de cada platô, as direções de fluxo das células daquele platô são definidas de modo que o fluxo seja orientado para suas células de escoamento.

Após a obtenção da direção de fluxo, o próximo passo é a determinação do fluxo acumulado em cada célula do terreno, isto é, a quantidade de água que atinge cada célula supondo que cada uma receba inicialmente uma unidade de água e que esta água seguirá as direções obtidas no passo anterior. Diversos métodos para a obtenção do fluxo acumulado como O'Callaghan & Mark [1984]; Arge et al. [2003]; Soille & Gratin [1994] se baseiam no método convencional de seguir as direções de fluxo. Outros [Muckell et al., 2007, 2008] modelam este problema como um sistema de equações lineares cuja solução fornece o fluxo acumulado em cada célula.

Uma vez obtido o fluxo acumulado, a rede de drenagem pode ser computada selecionando todas as células cujo fluxo acumulado é maior do que um certo limite pré-estabelecido. A partir da direção de fluxo e do fluxo acumulado outros elementos hidrográficos podem ser obtidos, por exemplo, as bacias de acumulação.

Conforme afirmado por Planchon & Darboux [2002], esse processo de obtenção da rede de drenagem exige uma quantidade considerável de processamento, principalmente devido à etapa de remoção dos fossos e tratamento dos platôs. Na verdade, na maioria dos métodos baseados nesta estratégia, mais de 50% do tempo total de processamento é consumido nesta etapa. Para evitar a necessidade da execução desta etapa, recentemente foi proposto em Magalhães et al. [2012b] um novo método para obtenção da rede de drenagem, denominado *RWFlood*, descrito resumidamente na seção 2.3.1, que evita a realização desta etapa de pré-processamento

simulando um processo de alagamento onde as depressões e os platôs são tratados naturalmente. Conforme apresentado em Magalhães et al. [2012b], este método chega a ser 100 vezes mais rápido do que os principais métodos descritos na literatura.

2.2.2 Algoritmos para processamento de dados em memória secundária

Durante o processamento de grandes volumes de dados, a transferência de informações entre as memórias interna e externa frequentemente domina o tempo de processamento dos algoritmos. Portanto, o projeto e análise de algoritmos utilizados para manipular esses dados precisa ser feito com base em um modelo computacional que avalia o número de operações de entrada e saída (E/S) realizadas. Um desses modelos, que vem sendo frequentemente utilizado pelos pesquisadores, foi proposto por Aggarwal & Vitter [1988] e considera que o computador é composto por um processador, uma memória interna de tamanho M e um disco (memória externa) que armazena os dados em blocos de tamanho B. Assim, a complexidade de um algoritmo é avaliada nesse modelo com base no número de operações de E/S realizadas, sendo que cada operação de E/S consiste na transferência de um bloco de tamanho B entre o disco e a memória interna.

Baseado neste modelo, os autores demonstraram que problemas fundamentais como a varredura (*scan*) e a ordenação (*sort*) de N elementos armazenados de forma sequencial no disco podem ser realizadas de maneira eficiente em memória externa, sendo que o número de operações de E/S efetuadas por esses problemas é:

$$scan\,(N) = \Theta\left(\frac{N}{B}\right) \quad \text{e} \quad sort\,(N) = \Theta\left(\frac{N}{B} log_{(\frac{M}{B})}\left(\frac{N}{B}\right)\right)$$

Vale mencionar que, em situações práticas, $scan(N) < sort(N) << N$ e, dessa forma, um algoritmo que realiza $sort(N)$ operações de E/S é muito mais eficiente do que um algoritmo que realiza $O(N)$ operações. Portanto, muitos algoritmos reorganizam os dados em disco utilizando um método de ordenação em memória externa para, então, poder acessá-los de forma sequencial e, com isso, processá-los de forma eficiente.

2.2.3 Algoritmos para determinação da rede de drenagem em memória secundária

Vários sistemas de informação geográfica, como por exemplo o ArcGis [ESRI, 2012] e o GRASS [GRASS Development Team, 2010], incluem algoritmos para cálculo da direção de fluxo e do fluxo acumulado. Mas, muitos destes algoritmos são projetados para minimizar o tempo de processamento e frequentemente não se ajustam para grande volume de dados [Arge et al., 2003]. Dentre os métodos desenvolvidos para o tratamento de grande volume de dados em memória externa pode-se destacar os módulos *TerraFlow* e *r.watershed.seg* disponíveis no GRASS [GRASS Development Team, 2010]. As seções 2.2.3.1 e 2.2.3.2 descrevem esses dois métodos.

2.2.3.1 TerraFlow

O *TerraFlow* é atualmente o sistema que resolve o problema de cálculo de elementos da hidrografia como rede de drenagem e bacia de acumulação (*watershed*) em grandes terrenos de forma mais eficiente [Arge et al., 2003; Toma et al., 2001]. Para aumentar o desempenho dos algoritmos, eles foram desenvolvidos utilizando métodos para o gerenciamento, reposicionamento e movimento dos dados com base no modelo proposto por Aggarwal & Vitter [1988].

O algoritmo de direção de fluxo é composto de 4 etapas, sendo cada uma delas otimizada para memória externa:

1. Identificação das áreas planas (platôs e fossos) e determinação das direções de fluxos na outras áreas.
2. Atribuição das direções de fluxos nos platôs
3. Preenchimento dos fossos, transformando-os em platôs.
4. Determinação das direções de fluxo para todo o terreno "corrigido".

O algoritmo de fluxo acumulado toma como entrada o terreno original e a grade de direções de fluxo e assume que cada célula recebe uma unidade de água e esta água escoa pelo terreno seguindo as direções de fluxo. Para obter o fluxo acumulado de forma eficiente, o método reordena as células no disco com base na ordem em que elas deverão ser processadas.

Além de reorganizar os dados no disco, o *TerraFlow* também utiliza uma técnica que "amplia" cada célula do terreno para que elas armazenem informações relevantes sobre as suas 8 vizinhas e, assim, evitem buscas (acessos ao disco). Como

resultado, o algoritmo trabalha com arquivos temporários que podem ser de até $80N$ *bytes*, onde N é o número de células do terreno.

2.2.3.2 Módulo r.watershed do GRASS

O *r.watershed* é um módulo do GRASS que pode ser utilizado para a obtenção da rede de drenagem em terrenos. O *r.watershed* foi inicialmente desenvolvido para memória interna e, então, adaptado para processamento em memória externa [Metz et al., 2011] com o uso da biblioteca *segmented* do GRASS, que permite a manipulação de grandes matrizes em memória externa.

A biblioteca *segmented* [GRASS Development Team, 2010] fornece um conjunto de rotinas para manipulação de grandes matrizes, que são divididas em blocos (segmentos) e armazenadas em arquivos temporários em disco. Por motivos de eficiência, uma quantidade k desses blocos é mantida na memória interna.

Dessa forma, ao se acessar um elemento da posição (i, j) da matriz, é determinado qual é o número *id* do bloco que contém esse elemento. Então, varre-se a lista de segmentos carregados na memória para verificar se o bloco *id* já está carregado. Se ele estiver carregado, o elemento (i, j) é retornado ou, caso contrário, o segmento é carregado na memória e, então, o elemento é retornado. Para evitar que a lista de segmentos carregados seja varrida sempre que uma posição da matriz for acesada, o último segmento utilizado é sempre armazenado no início da lista e, dessa forma, acessos consecutivos em um mesmo segmento são realizados de forma mais eficiente.

Ao carregar um segmento na memória, pode não haver espaço disponível para armazená-lo. Nesse caso, o bloco que estiver a mais tempo sem ser acessado é removido da memória para ceder sua posição ao novo segmento.

2.3 O método *EMFlow*

Conforme descrito anteriormente, os algoritmos existentes para o cálculo de elementos da hidrografia, em geral, consomem a maior parte de seu tempo numa etapa de pré-processamento para eliminar as depressões e tratar as regiões planas. Porém, em Magalhães et al. [2012b] é apresentado um novo método chamado *RWFlood* que é bem mais eficiente do que os outros algoritmos tradicionais, pois neste método não é necessário realizar esta etapa de pré-processamento e estas regiões são tratadas durante o próprio processamento.

Assim, como mencionado em 2.1, o objetivo desse trabalho é adaptar o método *RWFlood* para o processamento em memória externa. A seguir, este método será

descrito de forma resumida e, então, será apresentada a adaptação proposta para torná-lo eficiente no processamento de dados em memória externa.

2.3.1 O método *RWFlood*

A ideia básica do *RWFlood*, apresentado no Algoritmo 1, para obter a rede de drenagem de um terreno é simular o processo de inundação do terreno, supondo que a água entra no terreno pela sua borda vindo da parte externa. Neste caso, é importante observar que o caminho que a água percorre à medida que vai inundando o terreno é o mesmo caminho que a água percorreria se fosse proveniente da chuva que cai sobre o terreno e escoa descendentemente.

Em outras palavras, o método supõe que o terreno é uma ilha cercada por água em todos os lados sendo que o nível do "oceano" é iterativamente elevado. À medida que o nível deste oceano sobe, as células do terreno são gradativamente inundadas. Assim, ao atingir uma depressão, ela é preenchida por "água" e a elevação das células pertencentes a essa depressão é incrementada.

Mais especificamente, no início, o nível do oceano de água é definido como sendo igual à elevação do valor mais baixo entre as células da borda do terreno. Então, é realizado um processo iterativo que, a cada passo, eleva o nível do oceano e inunda as células do terreno que são adjacentes à borda deste oceano. Se a elevação dessas células é menor do que o nível da água então sua elevação é alterada para ficar igual ao nível do oceano.

Além de preencher as depressões, o *RWFlood* também calcula a direção de fluxo durante o processo de inundação. Note que a direção de fluxo corresponde a direção contrária àquela da inundação, isto é, se na inundação a água inundou uma célula c_i vindo da célula c_k então a direção de fluxo da célula c_i é direcionado para a célula c_k. Inicialmente, a direção das células da borda do terreno é definida apontando para fora do terreno (isto é, indicando que naquelas células a água escoa para fora do terreno) e a direção de cada célula c que não pertence à borda é definida como apontando para a célula vizinha a c de onde a água vem para inundar a célula c.

Depois de inundar todas as depressões e todas as células com elevação igual ao nível da água e que são adjacentes à borda do oceano, o nível da água é elevado para a elevação da célula mais baixa que é adjacente à borda deste oceano. Para obter esta célula que irá definir o nível da água, o método *RWFlood* utiliza um array Q de filas para armazenar as células que precisam ser posteriormente processadas. Ou seja, Q contém uma fila para cada elevação existente no terreno, sendo que a fila $Q[m]$

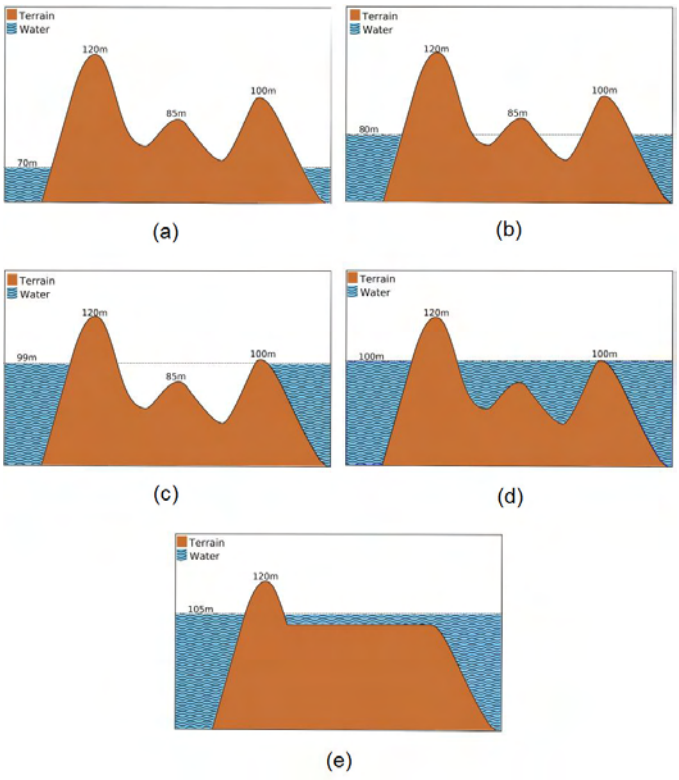

Figura 2.1. Processo de inundação em 5 diferentes níveis: (a) 70m, (b) 80m, (c) 99m, (d) 100m, (e) 105m. Fonte: Magalhães et al. [2012b]

armazena as células (a serem processadas) com elevação m. Inicialmente, as células na fronteira do terreno são inseridas na fila correspondente. Assim, supondo que a célula mais baixa na borda do terreno tem elevação k, então o processo começa na fila $Q[k]$ (isto corresponde a supor que nível da água se inicia com elevação k) e seja c a célula na primeira posição da fila $Q[k]$; esta célula é removida da fila e é processada da seguinte forma: as células vizinhas a c que ainda não foram "visitadas" (isto é, que ainda não têm a direção de fluxo definida) têm a sua direção de fluxo definida para a célula c e elas são inseridas nas respectivas filas. É importante observar que, se uma célula vizinha a c que ainda não foi visitada tem elevação menor do que c, então a elevação desta célula é aumentada (conceitualmente, isto corresponde a inundar a célula) e depois ela é inserida na fila correspondente a esta nova elevação. Quando todas as células na fila $Q[k]$ são processadas, o processo continua na próxima fila não vazia no vetor Q.

A figura 2.1 ilustra o processo de inundação. Em 2.1(a) o nível da água é 70m (as depressões ainda não foram inundadas). Em 2.1(b) é mostrado o nivel da água depois de algumas iterações (10 no caso), as depressões no centro do terreno estão abaixo do nível da água, mas elas não são inundadas ainda porque elas não são

adjacentes à borda do oceano. Em 2.1 (c), o nível de água é 99m e as células na fila $Q[99]$ são processadas (apenas as vizinhas à borda do oceano foram inseridas nas filas). Neste processo, as células com elevações de 100m, o pico mais à direita , são inseridas na fila $Q[100]$ e, quando o nível de água é definido em 100m, as células em $Q[100]$ são processadas (Figura 2.1(d)). Assim, a depressão passa a estar adjacente à borda da água e a elevação das células na depressão são definidas para 100m. Em 2.1(e) o nível da água é 105m.

Vale ressaltar que o método *RWFlood* determina a direção do fluxo de cada célula durante a inundação. Quando uma célula c é processada, todas as células vizinhas a c que ainda não foram visitadas (isto é, que não tem a sua direção de fluxo definida) têm o seu sentido de fluxo definido para c e depois, são inseridas na fila correspondente.

Após o cálculo da direção de fluxo, o algoritmo *RWFlood* calcula o fluxo acumulado no terreno utilizando uma estratégia baseada em ordenação topológica. Conceitualmente, a ideia é supor a existência de um grafo onde cada vértice representa uma célula do terreno e há uma aresta ligando um vértice v a um vértice u se, e somente se, a direção de escoamento de v aponta para u. Os vértices são inicializados com 1 unidade de água e o processamento se inicia num vértice v cujo grau de entrada é 0. Este vértice é marcado como visitado e, supondo que v direciona o fluxo para o vértice u, então o fluxo do vértice v é adicionado ao fluxo atual do vértice u. Além disso, a aresta que conecta o vértice v ao vértice u é removida reduzindo assim o grau de entrada do vértice u - este vértice u será processado (visitado) quando o seu grau de entrada se tornar 0.

2.3.2 Adaptação do método *RWFlood* para processamento em memória externa

O método *RWFlood* original processa o terreno, representado por uma matriz, acessando essa matriz de forma não sequencial e, portanto, o processamento de grandes terrenos armazenados em memória externa pode não ser eficiente. Conforme pode ser observado no algoritmo 1, o processo iterativo da linha 10 realiza, para cada célula c removida da fila, acessos a células vizinhas a c. Note que há um padrão de acessos espacial: em um dado momento as células acessadas estão, na maioria das vezes, próximas umas das outras na matriz. O problema é que, normalmente uma matriz bidimensional é armazenada de forma linear na memória e, por isso, células que são vizinhas (Na representação bidimensional) frequentemente ficam armazenadas em posições distantes umas das outras. Assim, se esta matriz estiver

armazenada em memória externa, seria necessário acessar o disco várias vezes durante o processamento de cada célula. Suponha que uma matriz de dimensões 6×6 seja armazenada em um disco cujo bloco é capaz de armazenar 4 células. Conforme ilustrado na figura Figura 2.2, para se processar a célula cujo rótulo é 11, é necessário acessar suas células vizinhas e, com isso, são realizados 5 acessos ao disco para carregar os blocos cujos rótulos são a,b,c,d e e.

Algorithm 1 RWFlood - Preenche depressões e calcula a direção de fluxo

```
1: Seja Q[minElev...maxElev] um arranjo de filas
2: for all célula c na borda do terreno do
3:    c.dir ← NULL
4: end for
5: for all célula c na borda do terreno do
6:    Q[c.elev].insert(c)
7:    c.dir ← OutsideTerrain
8: end for
9: for z = minElev → maxElev do
10:   while Q[z] não estiver vazia do
11:      c ← Queues[z].remove()
12:      for all célula n vizinha a c com n.dir = NULL do
13:         n.dir ← c
14:         if n.elev < z then
15:            n.elev ← z
16:         end if
17:         Q[n.elev].insert(n)
18:      end for
19:   end while
20: end for
```

Observe também que a ordem em que as células a serem processadas são armazenadas nas filas não garante que células adjacentes nessas filas estejam próximas na matriz. Veja a Figura 2.3: as células a serem processadas (marcadas de branco com X) são adjacentes à região já processada (marcada de cinza) e elas são adicionadas às filas em uma ordem que depende da forma com que o terreno é alagado. Assim, as células que são processadas em iterações consecutivas não estão, necessariamente, próximas no terreno.

Para contornar estes problemas diminuindo o número de acessos ao disco, este trabalho propõe um novo método denominado EMFlow, cuja estratégia consiste em adaptar o método *RWFlood* de forma que os acessos realizados à matriz sejam gerenciados por uma biblioteca denominada *TiledMatrix* [Magalhães et al., 2012a], que é capaz de armazenar e gerenciar grandes matrizes em memória externa. Na verdade,

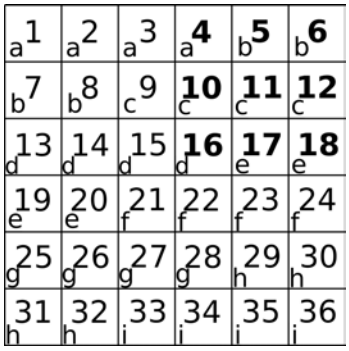

Figura 2.2. Exemplo de matriz com dimensões 6×6 armazenada em um disco onde o cada bloco é capaz de armazenar 4 células. A letra presente em cada célula indica o bloco na qual essa célula está armazenada.

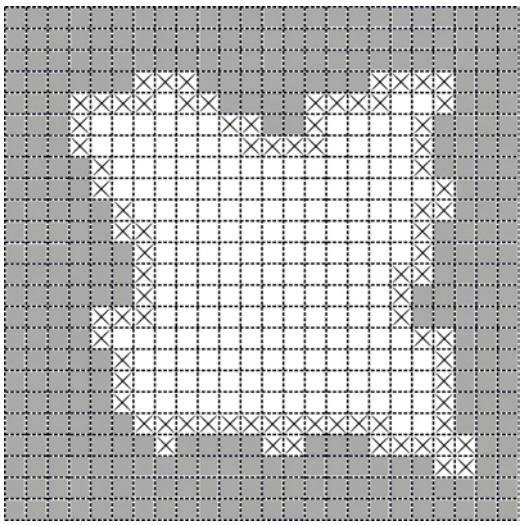

Figura 2.3. Exemplo de estado das células do terreno que está sendo processado: as células de cinza representam células já processdas (alagadas), as de branco as células não processadas e as de branco com X as que estão nas filas de processamento.

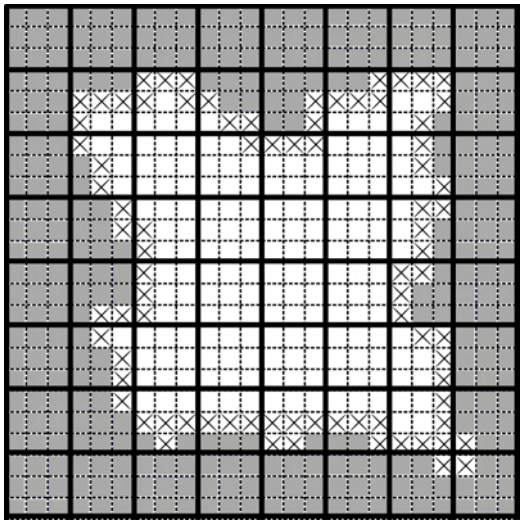

Figura 2.4. Exemplo de matriz dividida em blocos (representados pela linha contínua) e armazenada em disco.

a idéia básica desta adaptação é modificar a forma de gerenciamento da memória (reorganizando a matriz) para tirar proveito da localidade espacial de acesso.

Assim, as matrizes que armazenam a elevação do terreno, a direção de escoamento e o fluxo acumulado são substituídas por matrizes em memória externa que são gerenciadas pela biblioteca *TiledMatrix*. Essa biblioteca subdivide a matriz em blocos menores que são armazenados de forma sequencial em um arquivo na memória externa, sendo que a transferência destes blocos entre as memórias interna e externa também são gerenciados pela biblioteca que permite a adoção de diferentes políticas de gerenciamento. Em outras palavras, alguns blocos da matriz ficam armazenados em memória interna enquanto estiverem sendo processados e são transferidos de volta para a memória externa quando não forem mais necessários, dando lugar a outros blocos. Desta forma, a biblioteca adota uma estratégia semelhante ao gerenciamento de uma memória cache buscando predizer quais serão os próximos blocos da matriz que terão posições acessadas no processamento, mantendo-os na memória interna.

A Figura 2.4 ilustra a subdivisão de uma matriz em blocos. Observe que se todos os blocos que contiverem células a serem processadas (destacadas em branco com X) forem copiados para a memória principal haverá uma diminuição do número de operações de acesso a disco realizadas.

Uma questão importante a se considerar na implementação da biblioteca *TiledMatrix* refere-se à política utilizada para determinar qual bloco será escolhido para ceder espaço a novos blocos. Neste trabalho utilizou-se a estratégia de reti-

rar da memória interna aquele bloco que está a mais tempo sem ter sido acessado pela aplicação. Isto é, sempre que uma célula de um bloco é acessada, este bloco é marcado com um *timestamp*. Quando for necessário retirar um bloco da memória interna para carregar um outro bloco, será escolhido aquele bloco que tiver o menor *timestamp*. Esta estratégia foi adotada baseado no fato de que, durante o processamento do algoritmo *RWFlood*, há uma certa localidade de acesso às células do terreno. Ou seja, se há um bloco que está a algum tempo sem ser acessado (nenhuma de suas células é processada) então há uma grande chance de que nenhuma outra célula daquele bloco será processada (acessada) nos próximos acessos realizados pelo algoritmo ou então de que todas as células daquele bloco já tenham sido processadas e, portanto, o bloco não precisará mais ser acessado.

Mais precisamente, seja M uma matriz de dimensões $nrows \times ncolumns$ que se deseja armazenar em memória externa. Assim, primeiramente, a biblioteca *TiledMatrix* subdivide a matriz M em blocos de dimensões $m \times n$ que serão armazenados na memória secundária, sendo que m e n podem ser definidos pelo usuário. Para casos onde essa divisão não seja inteira, a biblioteca completa os blocos na fronteira da matriz com valores nulos - no caso de processamento de terrenos, estas células contém *NODATA*. Veja Fig 2.5. Então, cria-se uma cache que irá armazenar k blocos em memória interna sendo que essa cache é, na verdade, um vetor C de dimensão k de modo que cada posição do vetor armazena um bloco de dimensão $m \times n$. Para determinar se um bloco está armazenado ou não na cache e, em caso afirmativo, indicar em qual posição da cache ele está armazenado é criada uma matriz auxiliar Pos com dimensões $\frac{nrows}{m} \times \frac{ncols}{n}$ sendo que $Pos[i][j] = p$ indica que o bloco (i, j) está armazenado na posição p do vetor C - caso o bloco (i, j) não esteja carregado na cache então $Pos[i][j] = -1$.

Para gerenciar as transferências dos blocos entre as memórias externa e interna, a biblioteca armazena, para cada bloco, um *timestamp* indicando o momento em que o respectivo bloco foi acessado pela ultima vez.

2.3.3 Considerações sobre o *EMFlow*

Os métodos *EMFlow* e *r.watershed.seg* (incluído no GRASS) adotam estratégias semelhantes para obter a rede de drenagem em grandes terrenos armazenados em memória externa, ambas se baseiam em utilizar um biblioteca para gerenciar o acesso e a transferência de dados entre as memórias interna e externa[1]. No caso

[1]Na verdade essa estratégia é adotada pela grande maioria dos métodos desenvolvidos para processamento em memória externa em diversas aplicações.

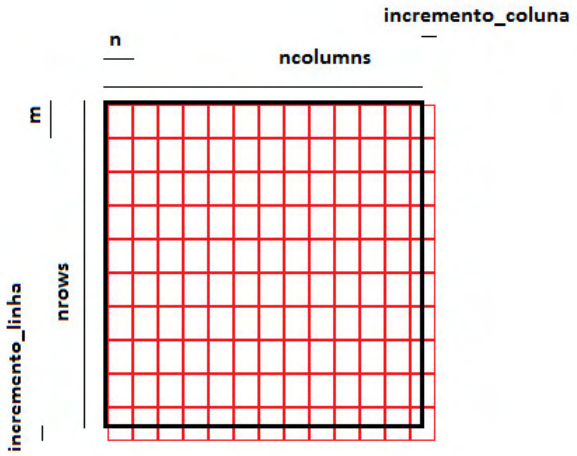

Figura 2.5. Modelo de divisão da Matriz M.

r.watershed.seg é utilizada a biblioteca *segmented* e no *EMFlow*, a biblioteca *TiledMatrix*. Embora essas duas bibliotecas sejam utilizadas para o mesmo intuito, elas possuem diferenças importantes, em particular:

- na biblioteca *segmented*, as posições dos blocos que estão na memória são armazanadas numa lista e para acessar uma célula de um bloco é necessário percorrer esta lista para obter a posição do bloco na memória, ou seja, o acesso a uma posição (i, j) é feita, no pior caso, em tempo linear. Na *TiledMatrix* este tempo é sempre constante porque as posições do bloco são armazenadas em uma matriz. Note que essa busca deve ser feita inúmeras vezes, ou seja, a eficiência desta operação afeta diretamente a eficiência do algoritmo que utiliza a biblioteca.

- em ambas bibliotecas, a substituição de blocos se baseia na política LRU, porém o processo de marcação dos blocos é diferente na *segmented*, os blocos são marcados com um valor numérico e estes valores são atualizados da seguinte forma: seja s o bloco corrente (isto é, acessado pela última vez); caso seja necessário acessar um bloco k diferente de s então os marcadores de todos os blocos, exceto o do bloco k, são incrementados em uma unidade. Desta forma, será substituído o bloco com o maior valor. Note que o processo de marcação requer tempo linear. Na *TiledMatrix*, os blocos são marcados com o *timestamp* do último acesso, isto é, ao acessar um bloco apenas o marcador deste bloco deve ser atualizado. Assim, na substituição, é removido o bloco com

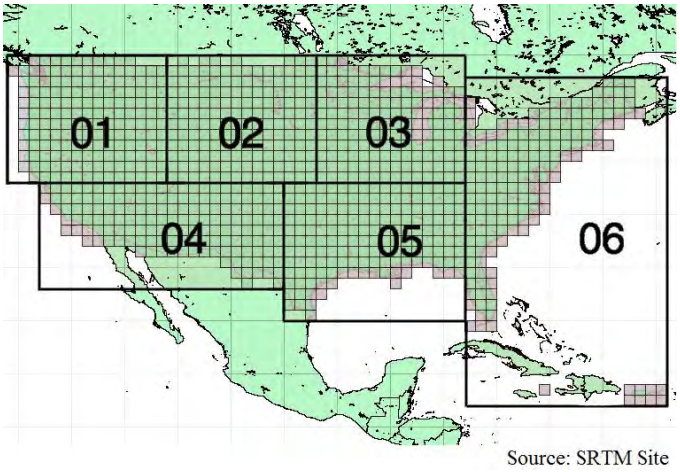

Figura 2.6. Regiões SRTM para EUA.

o menor *timestamp*. Vale observar que, neste caso, o processo de marcação requer tempo constante.

Todas essas diferenças tornam a biblioteca *TiledMatrix* bem mais eficiente do que a *segmented* do GRASS.

Uma questão importante sobre a eficiência do método *EMFlow* é o tamanho das filas que armazenam as células a serem posteriormente processadas, pois o espaço total ocupado por estas filas poderia ser maior do que a memória disponível, o que reduziria a eficiência do método. Porém, os vários testes realizados, indicaram que esta situação não ocorre na prática, uma vez que em nenhum dos testes este tamanho total não ultrapassou 1% do tamanho do terreno.

2.4 Resultados

O algoritmo *EMFlow* foi implementado em C++, compilado com o g++ 4.5.2, e vários testes foram realizados para avaliar seu tempo de execução e seu comportamento em diferentes situações comparando-o contra os métodos *TerraFlow* e *r.watershed.seg*, ambos incluídos no GRASS. Os testes foram executados em uma máquina com processador Intel Core 2 Duo com 2,8GHz, HD de 5400 RPM e sistema operacional Ubuntu Linux 11.04 64 bits. Esse computador foi configurado com diferentes capacidades de memória RAM: 4GB, 2GB e 1GB para avaliar os métodos em diferentes exigências de uso da memória externa.

Os terrenos utilizados nos testes foram gerados a partir de dados dos EUA disponibilizados pelo projeto SRTM [Jet Propulsion Laboratory NASA, 2012] com

Região	Tamanho	*EMFlow* Tempo(s)	TerraFlow Tempo(s)	r.watershed.seg Tempo(s)
Região 2	1000^2	0,66	24,43	6,25
	5000^2	14,18	661,37	622,66
	10000^2	74,56	2329,71	25784,71
	15000^2	326,15	7588,33	∞
	20000^2	717,87	12937,30	∞
	25000^2	2006,14	22220,89	∞
	30000^2	2848,13	35408,11	∞
	40000^2	5653,93	67076,04	∞
	50000^2	10649,04	98221,64	∞

Tabela 2.1. Tempo de processamento para diferentes tamanhos. Memória disponível de 1GB.

Região	Tamanho	*EMFlow* Tempo(s)	TerraFlow Tempo(s)	r.watershed.seg Tempo(s)
Região 2	1000^2	0,67	19,32	6,03
	5000^2	14,51	400,84	630,60
	10000^2	66,87	2251,66	5290,46
	15000^2	181,40	5870,34	34252,23
	20000^2	467,57	13066,63	∞
	25000^2	1024,84	19339,79	∞
	30000^2	1558,53	30364,31	∞
	40000^2	3119,20	56421,36	∞
	50000^2	5958,47	82673,22	∞

Tabela 2.2. Tempo de processamento para diferentes tamanhos. Memória disponível de 2GB.

resolução horizontal de 30 metros. Os dados utilizados nos testes foram extraídos da Região 2 (indicada na Fig.2.6) pois esta região está no centro do continente e portanto os terrenos não incluem parte do oceano que normalmente são definidas como *NODATA*.

As tabelas 2.1, 2.2 e 2.3 exibem os tempos de processamento (em segundos) utilizando, respectivamente, memórias de 1GB, 2GB e 4GB, sendo que no método *EMFlow* foram utilizados blocos com 200×200 células para a memória de 1GB,

Região	Tamanho	*EMFlow* Tempo(s)	TerraFlow Tempo(s)	r.watershed.seg Tempo(s)
Região 2	1000^2	0,81	19,32	6,34
	5000^2	15,04	400,84	616,53
	10000^2	65,38	2251,70	5186,70
	15000^2	153,60	5870,30	22276,00
	20000^2	295,35	13067,00	41493,00
	25000^2	529,50	19340,00	77729,00
	30000^2	850,53	30364,00	∞
	40000^2	1826,80	56421,00	∞
	50000^2	2897,60	82673,00	∞

Tabela 2.3. Tempo de processamento para diferentes tamanhos. Memória disponível de 4GB.

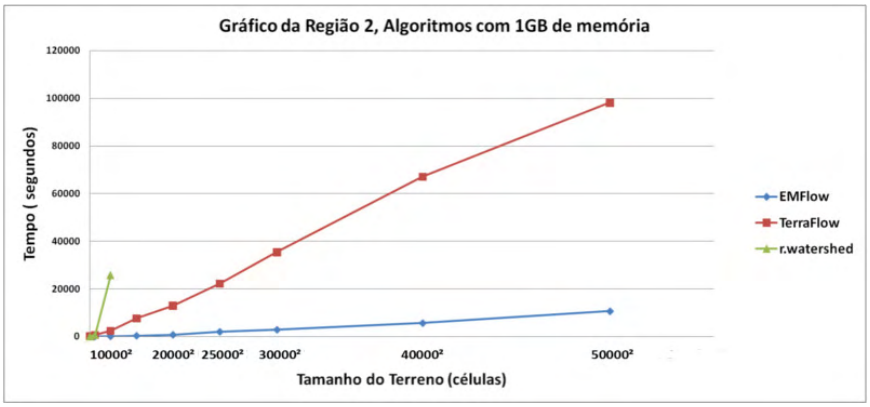

Figura 2.7. Padrão dos algoritmos para 1GB.

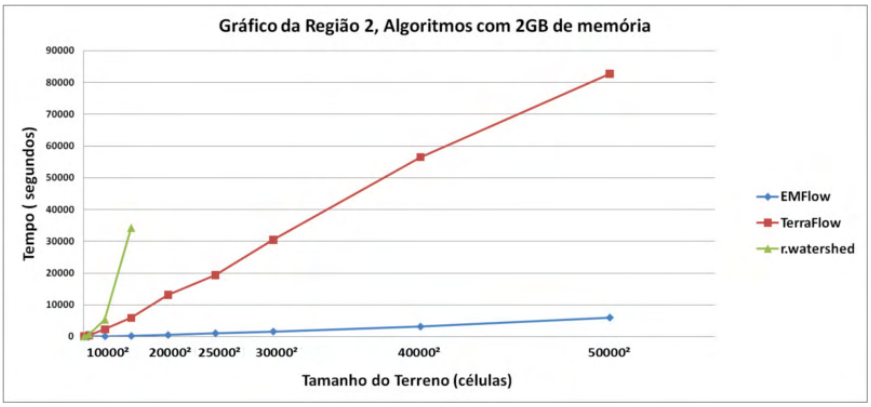

Figura 2.8. Padrão dos algoritmos para 2GB.

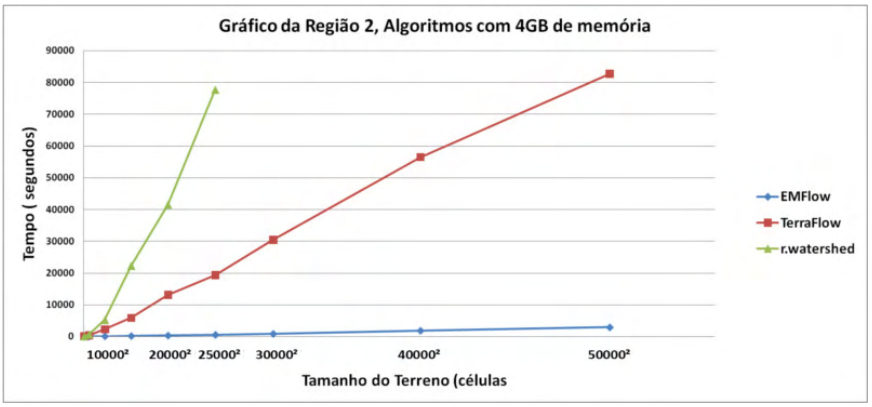

Figura 2.9. Padrão dos algoritmos para 4GB.

400×400 para a memória de 2GB e 800×800 para 4GB. No caso do *TerraFlow*, a versão disponível no GRASS utiliza, no máximo, 2GB de memória. No caso *r.watershed.seg*, o símbolo ∞ indica que, naquela situação, a execução do método foi interrompida quando o tempo de processamento ultrapassou 100000 segundos. As Figuras 2.7, 2.8 e 2.9 apresentam os gráficos referentes às tabelas 2.1, 2.2 e 2.3 respectivamente.

Como é possível verificar, em todas as situações, o método *EMFlow* apresentou um desempenho bem melhor do que os outros dois métodos, chegando a ser mais de 30 vezes mais rápido.

2.5 Conclusões e trabalhos futuros

Neste trabalho foi apresentado o algoritmo *EMFlow* para cálculo da rede de drenagem em grandes terrenos armazenados em memória externa e, como mostrado pelos testes, o método proposto apresenta uma eficiência muito superior aos principais métodos disponíveis. Em particular, vale destacar que, em situações extremas (terrenos muito maiores do que a memória interna), o *EMFlow* foi cerca de 30 vezes mais rápido do que o *TerraFlow* e, em muitas dessas situações, não foi possível obter o resultado (num tempo razoável) utilizando o método *r.watershed.seg*.

Outra observação interessante é que a estratégia utilizada no método *EMFlow* pode ser adaptada para outras aplicações que utilizem grandes matrizes armazenadas em memória externa. Como trabalhos futuros, pretende-se utilizar a biblioteca *TiledMatrix* em outras aplicações com este perfil. Além disso, também será realizado um estudo mais detalhado para avaliar a influência do tamanho dos blocos no desempenho do algoritmo e desenvolver uma estratégia para determinar automati-

camente o tamanho do bloco mais adequado.

3. Computing the drainage network on huge grid terrains

Abstract

We present a very efficient algorithm, named *EMFlow*, and its implementation to compute the drainage network, that is, the flow direction and flow accumulation on huge terrains stored in external memory. It is about 20 times faster than the two most recent and most efficient published methods: *TerraFlow* and *r.watershed.seg*. Since processing large datasets can take hours, this improvement is very significant.

The *EMFlow* is based on our previous method *RWFlood* which uses a flooding process to compute the drainage network. And, to reduce the total number of I/O operations, *EMFlow* is based on grouping the terrain cells into blocks which are stored in a special data structure managed as a cache memory. Also, a new strategy is adopted to subdivide the terrains in islands which are processed separately.

Because of the recent increase in the volume of high resolution terrestrial data, the internal memory algorithms do not run well on most computers and, thus, optimizing the massive data processing algorithm simultaneously for data movement and computation has been a challenge for GIS.

3.1 Introdution

Many important applications in Geographical Information Science (GIS) as hydrology, visibility, routing, etc require terrain data processing and these applications have become a challenge for GIS because they have to process a huge volume of high resolution terrestrial data. On most computers, the internal memory algorithms do not run well for such volume of data since a large number of I/O operations is necessary. For example, NASA's Shuttle Radar Topography Mission (SRTM) acquired 30 meters resolution terrain data for much of the world, generating about 10 terabytes of data. The datasets can be even bigger considering the technological advances which allow data acquisition at sub-meter resolution.

Thus, it is important to optimize the massive data processing algorithms simultaneously for computation and data movement between external and internal

memory since processing data in external memory takes much more time. That is, the algorithms for external memory processing must be designed and implemented to minimize the number of I/O operations for swapping data between main memory and disk.

More precisely, the algorithms for external memory processing should be designed and analyzed considering a computational model where the algorithm complexity is evaluated based on data transfer operations instead of CPU processing operations. A model often used, proposed by Aggarwal & Vitter [1988], defines an I/O operation as the transfer of one disk block of size B between the external and internal memory; the performance is measured by number of such I/O operations. The internal computation time is assumed to be comparatively insignificant. The algorithm complexity is defined based on the number of I/O operations executed by fundamental operations such as scanning or sorting n contiguous elements stored in external memory. Those are $\text{scan}(n) = \theta(n/B)$ and $\text{sort}(n) = \theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$, where M is the internal memory size.

Hydrological applications generally require the drainage network computation of a terrain, consisting of the flow direction and flow accumulation. Intuitively, they are the path that water flows through the terrain and the amount of water that flows into each terrain cell supposing that each cell receives a rain drop [Moore et al., 1991]. As broadly described Magalhães et al. [2012b]; Arge et al. [2003]; Metz et al. [2011]; Danner et al. [2007], it is a very time-consuming process, mainly on huge terrains requiring external memory processing. Indeed, in many situations, the flow direction can not be straightforwardly determined as for example, in a local minimum terrain cell.

In this paper, we present a new method, named *EMFlow*, for computing the drainage network on huge terrains represented by a digital elevation matrix stored in external memory. This new method is based on the adaptation of the *RWFlood* algorithm Magalhães et al. [2012b] where the idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses to the terrain matrix executed by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts a new (original) strategy to subdivide the terrain matrix into smaller pieces (islands) that can be processed separately.

The performance of *EMFlow* was compared against the most recent and efficient methods *TerraFlow* [GRASS Development Team, 2010] and *r.watershed.seg* [GRASS Development Team, 2010], both included in the open source GIS GRASS [GRASS Development Team, 2010]. As the tests showed, the *EMFlow* can be more than 20 times faster than the fastest of them. Since processing of large

terrains can take hours, this is a significant improvement.

3.2 Background and Previous Work

3.2.1 Drainage Network Computation

As described previously, the drainage network of a terrain delineates the path that water flows through the terrain (the flow direction) and the amount of water that flows into each terrain cell (the flow accumulation). As formulated by Arge et al. [2003], the flow direction problem is to assign the flow directions to all cells in the terrain such that the following three conditions are fulfilled:

1. Every cell has at least one flow direction;
2. No cyclic flow paths exist; and
3. Every cell in the terrain has a flow path to the edge of the terrain.

The flow direction can be modeled considering single flow direction (SFD) or multiple flow directions (MFD). In SFD, for each terrain cell it is assigned a direction towards the steepest downslope neighbor, while in MFD, each cell has directions to all downslope neighbors. The use of SFD or MFD is essentially a modeling choice since the computational complexity of the flow routing problem is the same in both models. This paper will use SFD.

There are several methods to obtain the drainage network Arge et al. [2003]; Metz et al. [2011]; Danner et al. [2007]; Jenson & Domingue [1988]; Tarboton [1997]. As described by those authors, the major challenge in the process is the flow routing in local minimum and flat areas. A local minimum is a cell with no *downslope neighbor* and a flat area is a set of adjacent cells with a same elevation. Given a cell c, a neighbor cell is called a downslope neighbor if it has a strictly lower elevation than c and a cell in a flat area that has a downslope neighbor is called a *spill-point*. Also, the flat areas can be classified as a *plateau* or a *sink* where the plateau has, at least, a spill point and a sink doesn't. Intuitively, water will accumulate in a sink until it fills up and water flows out of it [Jenson & Domingue, 1988] and in the plateau the water should flow towards spill points.

Usually, most drainage network computation methods, as for example Arge et al. [2003]; Soille & Gratin [1994]; Metz et al. [2011]; Danner et al. [2007], use a preprocessing step to remove the sinks and the flat areas. Initially, the elevation of the cells belonging to a sink are increased to transform it into a plateau. Next,

the directions on the plateau are assigned to ensure that there is a path (along flow directions) from each cell to the nearest spill point.

After obtaining the flow direction, the next step is to compute the flow accumulation in each terrain cell, that is, the amount of water flowing to each cell supposing that all cells receive a drop of water and this water follows the direction obtained in the previous step. Several methods for flow accumulation computation are based on graph topological sorting [O'Callaghan & Mark, 1984; Arge et al., 2003; Soille & Gratin, 1994] while others [Muckell et al., 2007, 2008] model this problem as a linear equations system.

According to Planchon & Darboux [2002], the drainage network computation requires a considerable amount of processing, mainly due the preprocessing step to remove depressions and flat areas. In fact, in most methods based on this strategy, more than 50% of the total processing time is spent by this step. To avoid this time-consuming step, recently Magalhães et al. [2012b] proposed a new method, named *RWFlood*, which is shortly described in section 3.3.1. As shown in Magalhães et al. [2012b], this method is more than 100 times faster than other recent methods but it does not scale well when the terrain does not fit in internal memory.

3.2.2 Computing Drainage Network Algorithms in External Memory

Several GIS implement algorithms for flow direction and flow accumulation but most of these algorithms were designed assuming that the terrain can be stored in internal memory and therefore they often do not scale well to large datasets [Arge et al., 2003]. On the other hand, there are some methods recently developed to process huge volume of data in external memory such as *TerraFlow* [GRASS Development Team, 2010] and *r.watershed.seg* [GRASS Development Team, 2010] both available in GRASS GIS.

3.2.2.1 TerraFlow

The *TerraFlow* is an efficient method, proposed by Arge et al. [2003]; Toma et al. [2001], to compute hydrological elements as drainage network and watershed in large terrains stored in external memory. It was implemented based on the model proposed by Aggarwal & Vitter [1988]. For performance improvements, it uses some specific methods for data management, replacement and movement between internal and external memory.

The flow direction is computed in several steps. Initially, the plateaus and sinks are identified and the flow directions on non-flat areas are determined. Next, the flow directions on plateaus are assigned and then the depressions are identified and filled (removed). Finally, the flow directions on these areas are determined.

The flow accumulation is computed taking the elevation grid and the flow direction grid as input. Then, assuming that each cell receives a unit of water which flows according to the flow direction, the cells are processed using a strategy called *time forward processing* which uses a priority queue to process the cells in a topological order.

As described by the authors, the *TerraFlow* complexity is $\Theta(sort(n))$ and it uses some temporarily files whose total size may be up 80 times larger than the original terrain file.

3.2.2.2 GRASS module r.watershed

The *r.watershed* is another GRASS module to obtain the drainage network. It was initially developed for internal memory processing and adapted for external memory [Metz et al., 2011] using the GRASS *segment* library [GRASS Development Team, 2010], which allows an efficient processing of huge matrices in external memory.

The *segment* library provides a set of functions to manage huge matrices stored in external memory. Basically, the matrix is subdivided into segments (blocks) that are stored in temporary disk files. To improve the efficiency, a given number of these segments are kept in internal memory. Thus, to access a given matrix position, firstly, it is determined which segment contains that position and, then, the list of segments stored in internal memory is swept to check if the corresponding segment is already loaded. If yes, the position is accessed as usual, otherwise, the corresponding segment need to be transferred to internal memory. To avoid the segment list sweeping at each matrix access, the last accessed segment is kept in the first position of the list and, thus, consecutive accesses in a same segment are more efficient.

When loading a segment in memory, there may be no space available to store the new segment and, in this case, the segment having the longest time without being accessed is evicted to open space for the new segment. In the *segment* library implementation, the segments have a “access time" field represented by an integer and every time a new segment is accessed (that is, a segment that is not in the front of the list) its access time is set to zero and the access time of all other segments are

incremented by 1. Thus, in some cases, the segment access can have a large CPU overhead.

3.3 The *EMFlow* method

As described in section 3.2.1, most methods for flow direction computation use a very time-consuming preprocessing step to remove depressions and flat areas. However, in Magalhães et al. [2012b] we presented a new method, named *RWFlood*, which is much more efficient than other classical methods, mainly because it does not perform this preprocessing step and the depressions and flat areas are naturally handled during the processing. Thus, as mentioned in Section 3.1, the purpose of this work is to adapt the *RWFlood* method for external memory processing.

3.3.1 *RWFlood* method

To avoid the time-consuming preprocessing step, *RWFlood* computes the drainage network using a reverse order. Instead of determining the downhill flow it uses a flooding process. More precisely, the method is based on the following observation: if a terrain is flooded by water coming from outside and getting into the terrain through its boundary then the course of the water getting into the terrain will be the same as the water coming from rain and flowing downhill (that is, the flow direction). Thus, the idea is to suppose the terrain is surrounded by water (as an island) and to simulate a flooding process raising the water level iteratively. When the water level raises, it gradually floods the terrain cells and when it reaches a depression, that depression is filled by “water".

Figure 3.1 illustrates the flooding process: in Figure 3.1(a) the whole terrain is an island and next, in 3.1(b), the water level achieves the lowest cell in the terrain boundary. The raising water process continues and in 3.1(c) the water starts to get into the terrain and a terrain depression is filled — see 3.1(d). The flooding process can generates new islands as in 3.1(e). Finally, the process ends when the whole terrain is flooded — see 3.1(f).

More formally, in the beginning, the water level is set to the elevation of the lowest cell in the terrain boundary. Then, two steps are executed iteratively: flooding a cell and raising the water level. When flooding a cell c, all cells neighbors to c are processed as following: given a neighbor cell d, if the elevation of d is smaller than the elevation of c, then the elevation of d is raised to the elevation of c; also, the flow direction of d is set to the cell c.

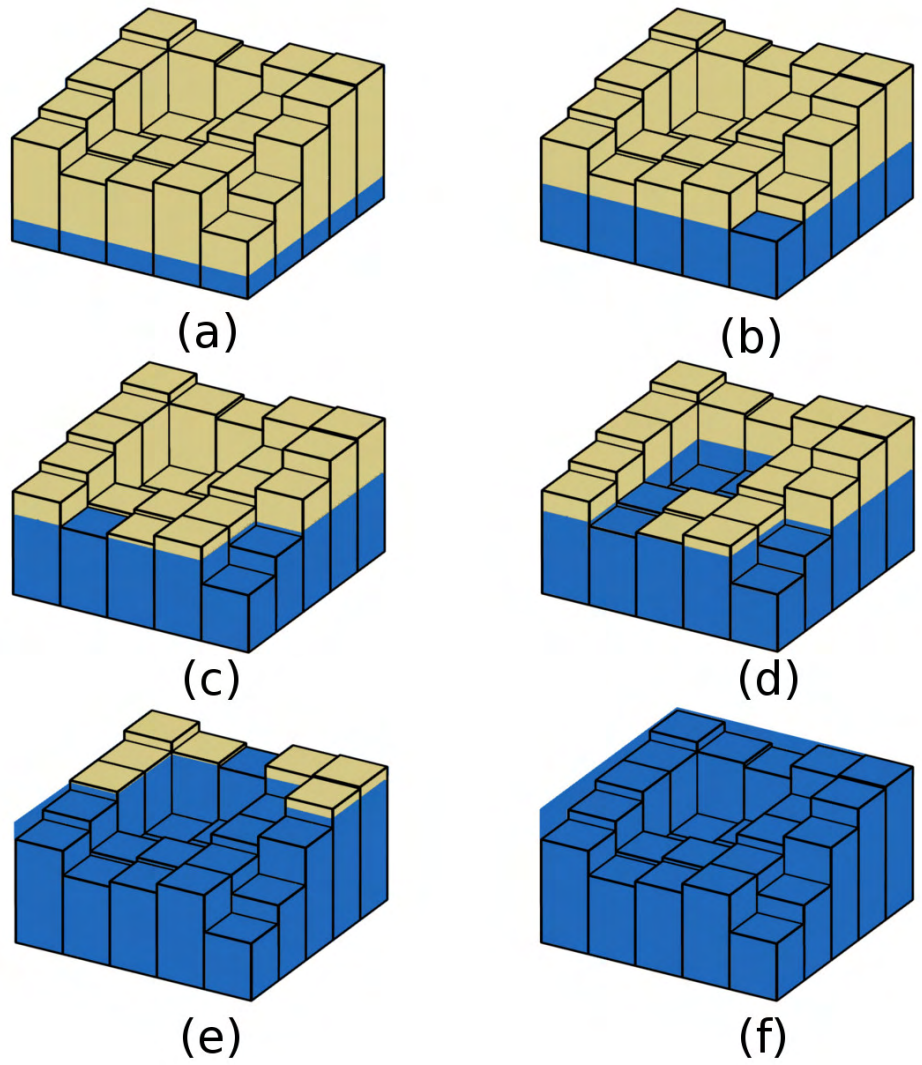

Figura 3.1. The flooding process: (a) the whole terrain is an island; (b) the water level is on the lowest cell in the terrain boundary; (c) the water level is raised; (d) a depression is flooded; (e) the flooding process creates two islands; (f) the flooding process is complete.

After flooding all cells with the same elevation as c, the next step is executed, that is, the water level is raised to the elevation of the lowest cell higher than c and the process continues from this cell. To get this cell quickly, the method uses an array Q of queues to store the cells that need to be processed later. Thus, Q contains one queue for each elevation — queue $Q[m]$ will store the cells with elevation m that were already visited and need to be processed later. Initially, each cell in the terrain boundary is inserted into the corresponding queue. Supposing the lowest cells have elevation k, the process starts at queue $Q[k]$ and, after processing

Algorithm 2 RWFlood - computes the flow direction

```
1: Let Q[minElev...maxElev] be an array of queues
2: for all cell c in the terrain boundary do
3:    c.dir ← NULL
4:    Q[c.elev].insert(c)
5:    c.dir ← OutsideTerrain
6: end for
7: for z = minElev → maxElev do
8:    while Q[z] is not empty do
9:       c ← Queues[z].remove()
10:      for all cell d neighbor to c such that d.dir = NULL do
11:         d.dir ← c
12:         if d.elev < z then
13:            d.elev ← z
14:         end if
15:         Q[d.elev].insert(d)
16:      end for
17:   end while
18: end for
```

all cells in that queue, the process proceeds with the next non-empty queue in the array Q (intuitively, meaning that the water level is raised). Let $Q[z]$ be this next non-empty queue, then the front cell is dequeued (conceptually, it is flooded) and its neighbors are visited. That is, given a neighbor cell v, if v has already been visited, it is done; on the other hand, if v has not been visited yet, and if its elevation is not lower than z, it is inserted in its corresponding queue; otherwise, if its elevation is lower than z, its elevation is set to z and the cell is inserted into $Q[z]$. This latter case corresponds to flooding a depression point.

Thus, the next cell to be processed can be easily obtained by getting the next cell in the current queue (if it is not empty) or the first cell in the next non-empty queue. See algorithm 2.

The flow direction of each cell can be determined during the flooding process since, when a cell c is processed, all cells adjacent to c which are inserted in a queue can have their flow direction set to c. That is, conceptually, the flow direction is set to the opposite direction as the water gets into the cells and, thus, the water in the adjacent cells will flow to the cell c. Initially, the flow direction of all cells in the terrain boundary is set to out of the terrain (i.e., indicating that in those cells the water flows out of the terrain).

After computing the flow direction, *RWFlood* uses an algorithm based on graph topological sorting to compute the flow accumulation. Conceptually, the idea is to

process the flow network as a graph where each terrain cell is a vertex and there is a directed edge connecting a cell c_1 to a cell c_2 if and only if c_1 flows to c_2. Initially, all vertices in the graph have 1 unit of flow. Then, in each step, a cell c with in-degree 0 is set as visited and its flow is added to the $next(c)$'s flow where $next(c)$ is the cell following c in the graph. After processing c, the edge connecting c to $next(c)$ is removed (i.e., $next(c)$'s in-degree is decremented) and if the in-degree of $next(c)$ becomes 0, the $next(c)$ cell is similarly processed.

3.3.2 Adapting *RWFlood* for external memory processing

As presented in Magalhães et al. [2012b], the *RWFlood* method is very efficient when the whole terrain can be processed in internal memory. However, its performance decreases significantly whenever the terrain does not fit in internal memory and it is necessary to perform external processing. The main reason for this inefficiency is the non-sequential access to the terrain matrix. Indeed, according to the flooding process, the cells are accessed (processed) following the elevation order from the lowest to highest elevation. Also, when a cell is processed, its neighbors need to be accessed but, although these cells are close in the two-dimensional matrix representation, they may not be close in the memory because, usually, a matrix is stored using a linear row-major order.

To circumvent this problem and reduce the number of disk accesses, we propose a new method, named *EMFlow*, whose basic idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses carried out by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts a new (original) strategy to subdivide the terrain matrix in smaller pieces which can be processed separately.

Conceptually, the main idea of *RWFlood* is to store the cells in the boundary of the flooded regions — see Figure 3.2(c) and (d). At each step, the lowest cell in this boundary is processed. When a cell c is processed, all neighbors of c that were not processed yet and whose elevation is smaller or equal to the elevation of c are flooded, that is, the flooding boundary moves toward these cells. This flooding process can generate interior islands — see Figures 3.2(a) and (b) — and these islands can be processed (flooded) separately since the flooding process of an island does not affect any other island. Based on this fact, the *EMFlow* subdivides the terrain into islands that are processed one by one.

More precisely, initially, the whole terrain is processed as one island which is flooded using the *RWFlood* strategy. Next, at some moment (described below), the

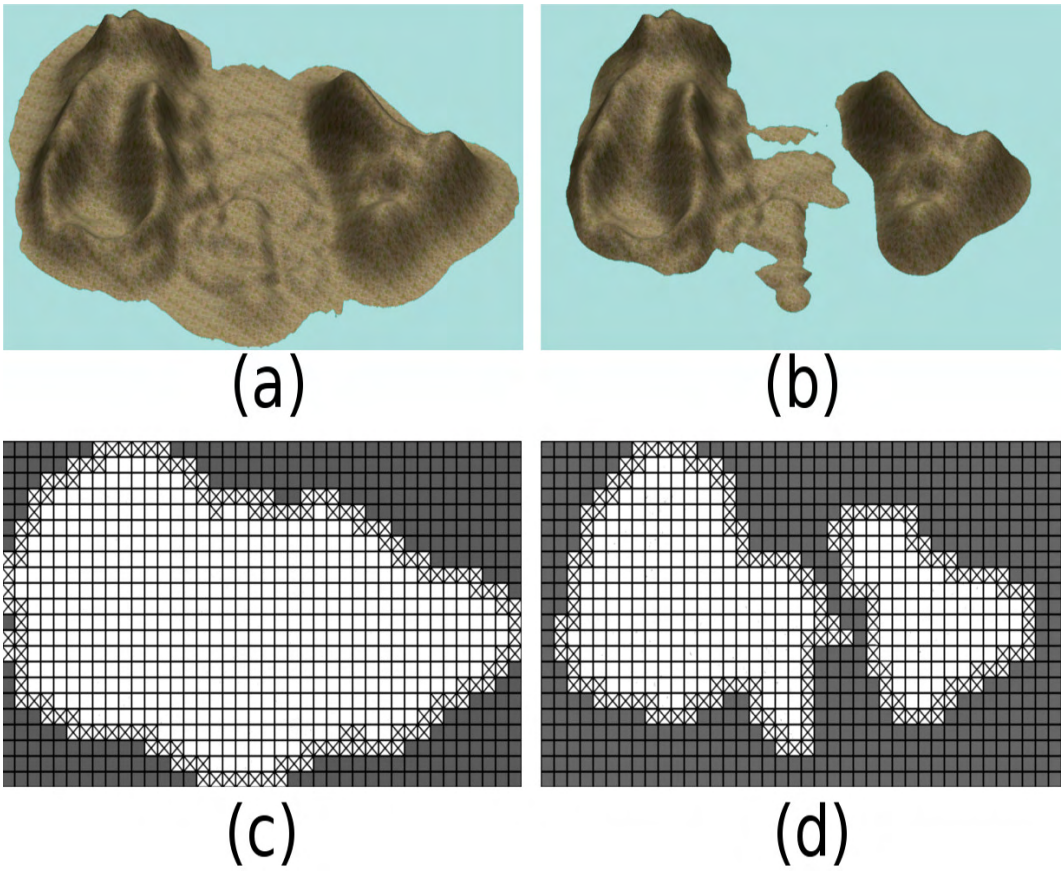

Figura 3.2. (a) Flooding the terrain; (b) The flooding process generated two islands; (c) and (d) The cells in the flooding boundary are labeled with $\times$.

algorithm analyzes if the flooding process generated internal islands. Notice that, an island is a group of connected cells which were not flooded (that is, processed) yet. Thus, the islands can be identified computing the connected components composed of non processed cells. After identifying the islands, each one is processed independently.

However, this subdivision strategy does not assure that the process can be entirely executed in internal memory. The islands can be too large and have too many cells that do not fit in internal memory. Thus, to improve the algorithm performance, the terrain matrix accesses are managed by the *TiledMatrix*[Magalhães et al., 2012a] library which was designed to store and manage huge matrices in external memory. Basically, in *TiledMatrix*, a matrix is subdivided in blocks whose size allows that a given number of blocks can be stored in internal memory. Then, all blocks are stored in external memory and they are loaded to internal memory on demand. That is, when a cell c needs to be accessed, the library determines which block contains that cell and, if the block is not in the internal memory, it is loaded.

Since eventually there may not be space to store a new block, the data structure storing the blocks is managed as a cache memory. More precisely, the library adopts a replacement policy to evict a block and open room for the new block[1]. *EMFlow* uses the *LRU - least recently used* policy.

Furthermore, to reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [Collet, 2012]. Before storing a block in the disk, it is compressed and when a block is loaded to internal memory it is uncompressed.

3.3.2.1 Implementation details

In the *EMFlow* implementation, we adopted some strategies for performance improvement:
(1) Islands identification: an island generated during the flooding process is composed of a group of connected cells that were not flooded yet, and this group is surrounded by flooded cells. That is, an island is a maximal connected component of non-flooded (or non-processed) cells and, to have an island, it is necessary to have a group of flooded (processed) cells surrounding the island. But, since the connected component computation is a time-consuming process, mainly when the terrain matrix can not be stored in internal memory, the algorithm adopts a less accurate strategy where the islands are identified using a lower resolution terrain. More precisely, the algorithm creates an auxiliary matrix C where each cell corresponds to a square block in the terrain matrix and a C cell stores the number of corresponding terrain cells which were not processed yet. That is, the cells of C are initialized with the number of terrain cells in each corresponding square block and, during the flooding process, this value is decremented whenever a corresponding terrain cells is processed. When the value in a C cell becomes zero it indicates that all cells in the corresponding terrain block were already processed. Thus, the islands identification process is reduced to the computation of the maximal connected component of non zero cells in the matrix C.

Notice that if two blocks are disconnected in C then the cells in each block will belong to different islands and, thus, they can be processed separately. On the other hand, two different islands in the terrain may be identified as connected in C (because C has a lower resolution), that is, they may be identified as one island. But, the final result does not change if two islands are processed as one island. This

[1]The library provides the following policies: LFU - Least Frequently Used, FIFO - first in first out, and random selection.

may only lead to a larger processing time because the number of cells which need to be stored in internal memory may increase.

Since the islands identification is not a trivial process, it is executed only occasionally. The idea is to execute it when there are evidences that some islands were generated. In the *EMFlow* algorithm, the lenght of the flooded region boundary was used to trigger this process, that is, it is executed when the number of cells in the flooded region boundary achieves a given threshold.
(2) Scheduling the islands processing: as described previously, during the terrain flooding, the island generation follows a recursive sequence, but these islands can be processed in any order since they are independent and their processing is self-contained. Thus, in *EMFlow*, the processing of the islands is scheduled trying to process first those islands that (probably) will require a smaller number of external memory accesses. Since the cells in the islands boundary are already stored in internal memory then the external memory accesses will be required only if there exist some cells adjacent to the islands boundary that are not in internal memory yet. Then, the algorithm computes, for each island, the percentage of cells adjacent to the island boundary that are already in internal memory and the islands with higher percentage are processed first. In fact, since the matrix cells accesses are managed by the *TiledMatrix* library using blocks, the algorithm computes the percentage of blocks containing cells adjacent to the boundary that are already in internal memory.
(3) The islands boundary size: when an island is processed, all cells on its boundary need to be loaded into internal memory and also, during the cell processing, the neighbor cells must be loaded too. Thus, if the algorithm tries to process many islands simultaneously and if these islands have long boundaries (with too many cells), this large number of cells may not fit in internal memory. In this case, some cells (in fact, some blocks) need to be evicted and reloaded again later. To avoid these time-consuming operations, the algorithm defines a threshold to limit the number of islands that could be processed at a same time, that is, which could be loaded in internal memory.

3.3.3 *EMFlow* versus r.watershed.seg

Both methods *EMFlow* and *r.watershed.seg* (included in GRASS) try to improve their performance by using libraries to manage the external memory accesses; *EMFlow* uses the *TiledMatrix* library [Magalhães et al., 2012a] and *r.watershed.seg* uses *segment* [GRASS Development Team, 2010]. Although these two libraries have similar purposes and both are based on to subdivide the matrix in blocks and manage

them using a cache strategy, they have some important differences described below:

- Both libraries store a set of blocks in internal memory using an array. However, when a terrain cell is accessed, they use different methods to check if the block containing that cell is already loaded in internal memory. In the *segment*, the array positions where the blocks are stored are kept in a list of pairs (b_n, b_p) where b_n is the block number (referent to the terrain matrix) and b_p is the block position in the internal memory array. Then, to check if the block is loaded in internal memory (and get it), the list is searched. Thus, in the worst case, the access to a terrain cell can take $O(n)$ time, where n is the number of blocks stored in internal memory. Trying to reduce this time, the library keeps the last block accessed in the front of the list to avoid the worst case of searching operation when the next accessed cell is also in the same block. On the other hand, in *TiledMatrix*, the terrain cell access always takes a constant time since the blocks' positions are stored in a matrix of size $\frac{N}{h} \times \frac{M}{w}$ where N and M are respectively the terrain matrix height and width and h and w are respectively the block height and width. Thus, if a block is not loaded in internal memory, the matrix position corresponding to that block is set to -1, otherwise, it is set to the array position where that block is stored. As this operation is executed many times during the whole process, its efficiency affects directly the algorithm performance.

- The block replacement policy is LRU in both libraries, but the libraries use different strategies for block marking. In *segment*, the blocks are marked with an integer value which is updated every time a block is accessed. Initially, all blocks are marked with zero and when a new block b is accessed (that is, when a cell contained in a new block b is accessed), the value of all blocks, except b, are incremented. Thus, the block replacement will evict the block with the smaller value. In *TiledMatrix*, the blocks are marked using a *timestamp*, that is, when a block is accessed, it is marked with the current timestamp. Then, the block with the smaller timestamp will be evicted. Therefore, the block marking takes $O(n)$ time in *segment* and a constant time in *TiledMatrix*.

- To reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 Collet [2012]. Thus, before writing a block to the disk, it is compressed using LZ4 and, after reading a block from the disk, it is uncompressed. As presented in Magalhães et al. [2012a], the *EMFlow* is more

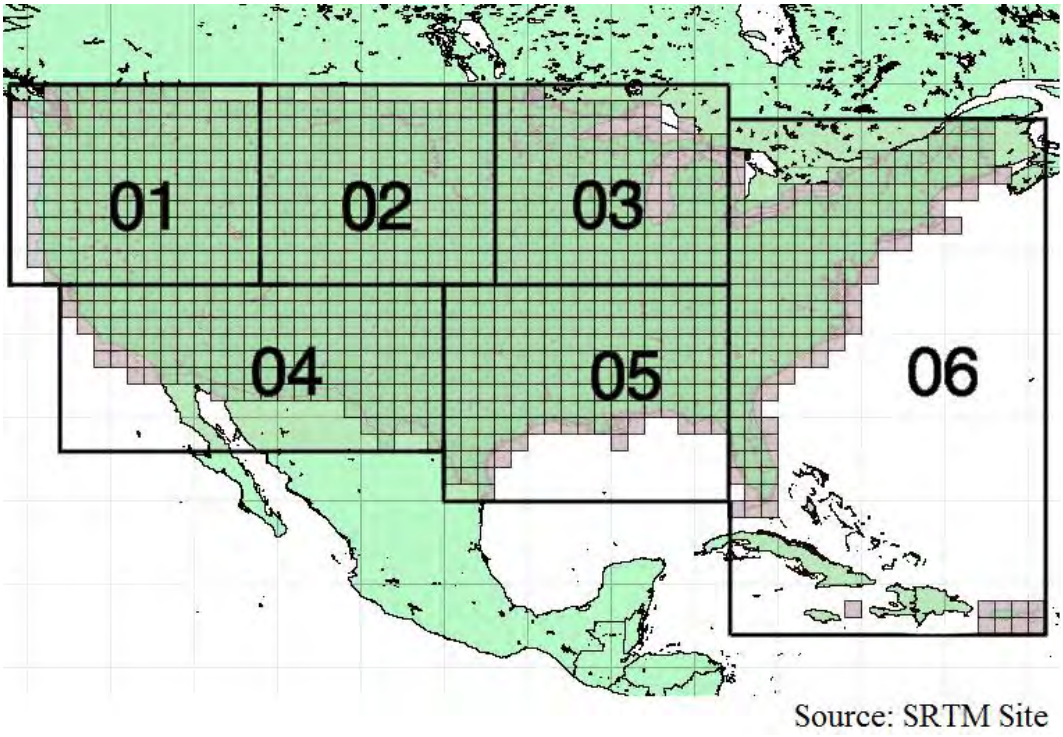

Figura 3.3. SRTM USA Regions.

than two time faster when this compression strategy is used. On the other hand, the *segment* does not adopt any similar strategy.

3.4 Experimental Results

EMFlow was implemented in C++ and compiled with g++ 4.5.2. It was compared against the most efficient algorithms described in the literature: *TerraFlow* GRASS Development Team [2010] and *r.watershed.seg* GRASS Development Team [2010] both available in GRASS. The tests were executed in a machine with an Intel Core 2 Duo with 2,8GHz and 5400 RPM SATA HD (Samsung HD103SI) running the Ubuntu Linux 11.04 64 bits operation system. This machine was configured with different internal memory sizes: 1GB and 2GB to evaluate the algorithms performance in different scenarios.

The tests used different datasets generated from two distinct USA regions (regions 02 and 03 in Figure 3.3) sampled at $30m$ horizontal resolution using 2 bytes per elevation value. These two regions were selected because they are in the central part of USA, do not include ocean, and therefore have few NODATA elements.

Tables 3.1 and 3.2 show the execution time (in seconds) of the three algorithms

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,93	24,43	6,25	0,92	28,22	5,91
5000^2	18,80	661,37	622,66	19,11	907,50	508,90
10000^2	81,67	2329,71	25784,71	81,09	3358,42	55182,80
15000^2	251,14	7588,33	∞	248,39	9046,13	∞
20000^2	579,84	12937,30	∞	605,38	14404,76	∞
25000^2	980,14	22220,89	∞	1065,78	24974,77	∞
30000^2	1522,61	35408,11	∞	1890,35	41251,21	∞
40000^2	3055,39	67076,04	∞	4117,65	78056,28	∞
50000^2	7173,84	98221,64	∞	7618,78	110394,74	∞

Tabela 3.1. Processing time (in seconds) for different terrain sizes from regionws R2 and R3 considering a memory size of 1GB.

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,74	19,32	6,03	0,98	19,44	5,79
5000^2	20,02	400,84	630,60	19,98	442,97	513,88
10000^2	87,66	2251,66	5290,46	86,94	2552,93	3911,23
15000^2	209,02	5870,34	34252,23	202,36	6869,33	32518,89
20000^2	437,58	13066,63	∞	415,37	13873,60	∞
25000^2	776,98	19339,79	∞	764,86	22492,14	∞
30000^2	1179,31	30364,31	∞	1196,58	33337,07	∞
40000^2	2254,80	56421,36	∞	2162,17	59149,27	∞
50000^2	4011,72	82673,22	∞	3470,99	86670,30	∞

Tabela 3.2. Processing time (in seconds) for different terrain sizes from regionws R2 and R3 considering a memory size of 2GB.

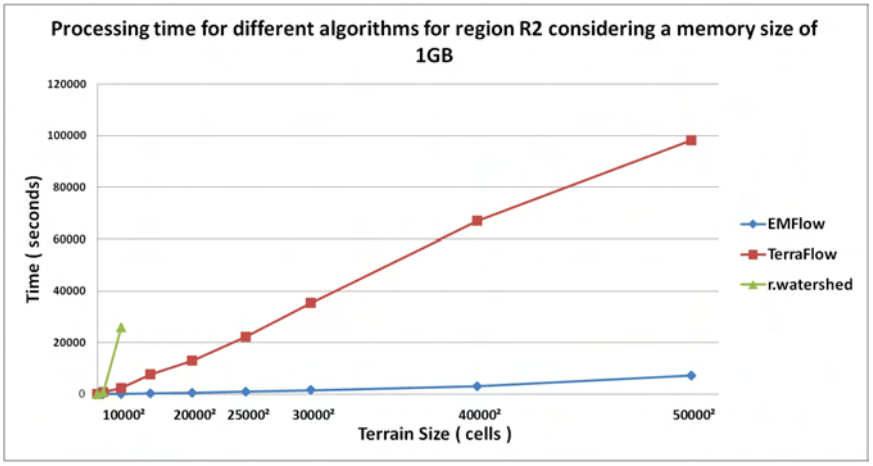

Figura 3.4. Chart corresponding the region R2 considering memory size 1GB.

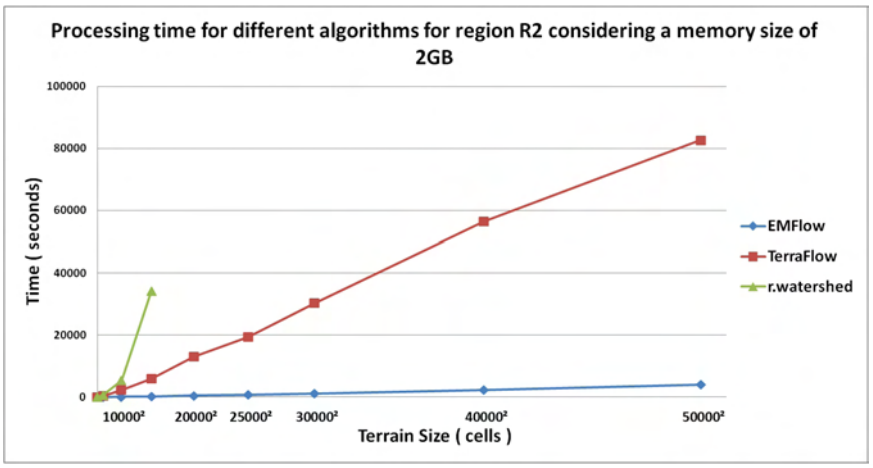

Figura 3.5. Chart corresponding the region R2 considering memory size 2GB.

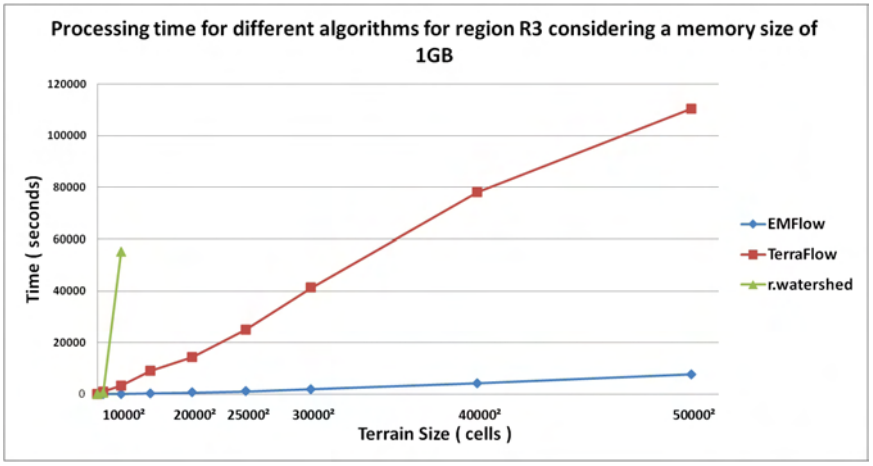

Figura 3.6. Chart corresponding the region R3 considering memory size 1GB.

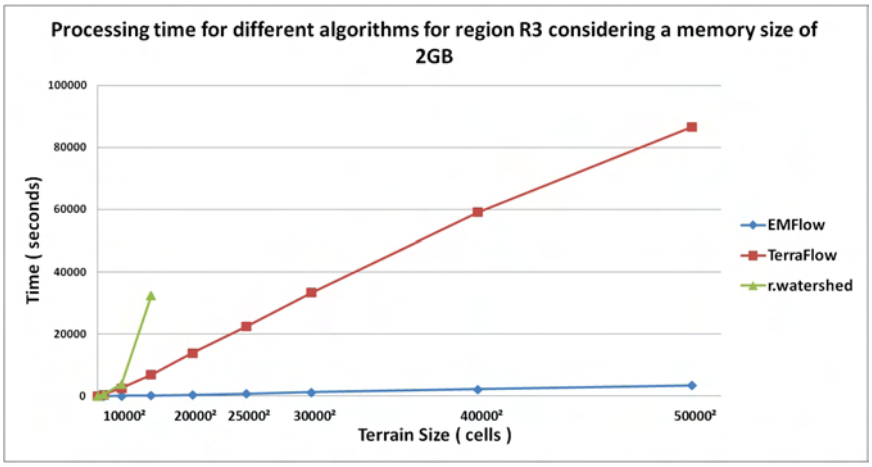

Figura 3.7. Chart corresponding the region R3 considering memory size 2GB.

in the R2 and R3 regions using respectively 1GB and 2GB of RAM. In these tests, the *TiledMatrix* library, used by *EMFlow*, was configured as following: for 1GB of RAM it was used blocks with 200×200 cells and for 2GB the block size was 400×400 cells. In the tables, the symbol ∞ is used to indicate that the execution was interrupted after 150000 seconds (40 hours). Figures 3.4, 3.5, 3.6 and 3.7 presents the charts corresponding to the tables.

Note that *EMFlow* was faster than the other two algorithms in all situations and, for very huge terrains (as for 50000×50000), *EMFlow* was more than 20 times faster than *TerraFlow* while *r.watershed.seg* was not able to conclude the terrain processing after 40 hours.

It is worth to mention that, since *EMFlow* is based on *RWFlood*, the drainage networks computed by these two algorithms are the same. Additionally, as presented in Magalhães et al. [2012b], the drainage networks obtained by *RWFlood* are very similar (almost the same) to those computed by *TerraFlow* and *r.watershed*. For example, Figure 3.8 presents the drainage networks computed by the three methods: *EMFlow*, *TerraFlow* and *r.watershed.seg* in two terrains: the R3 region and a terrain from Tapajos[2] region. The Figures 3.8 (a) and (d) show the networks computed by *EMFlow* in the regions R3 and Tapajos respectively, the Figures 3.8 (b) and (e) show the networks from R3 and Tapajos computed by *TerraFlow* and Figures 3.8 (c) and (f) show the networks computed by *r.watershed* in those regions. As it is possible to see, the corresponding networks are very similar.

[2]Tapajos is an important tributary river of the Amazon basin.

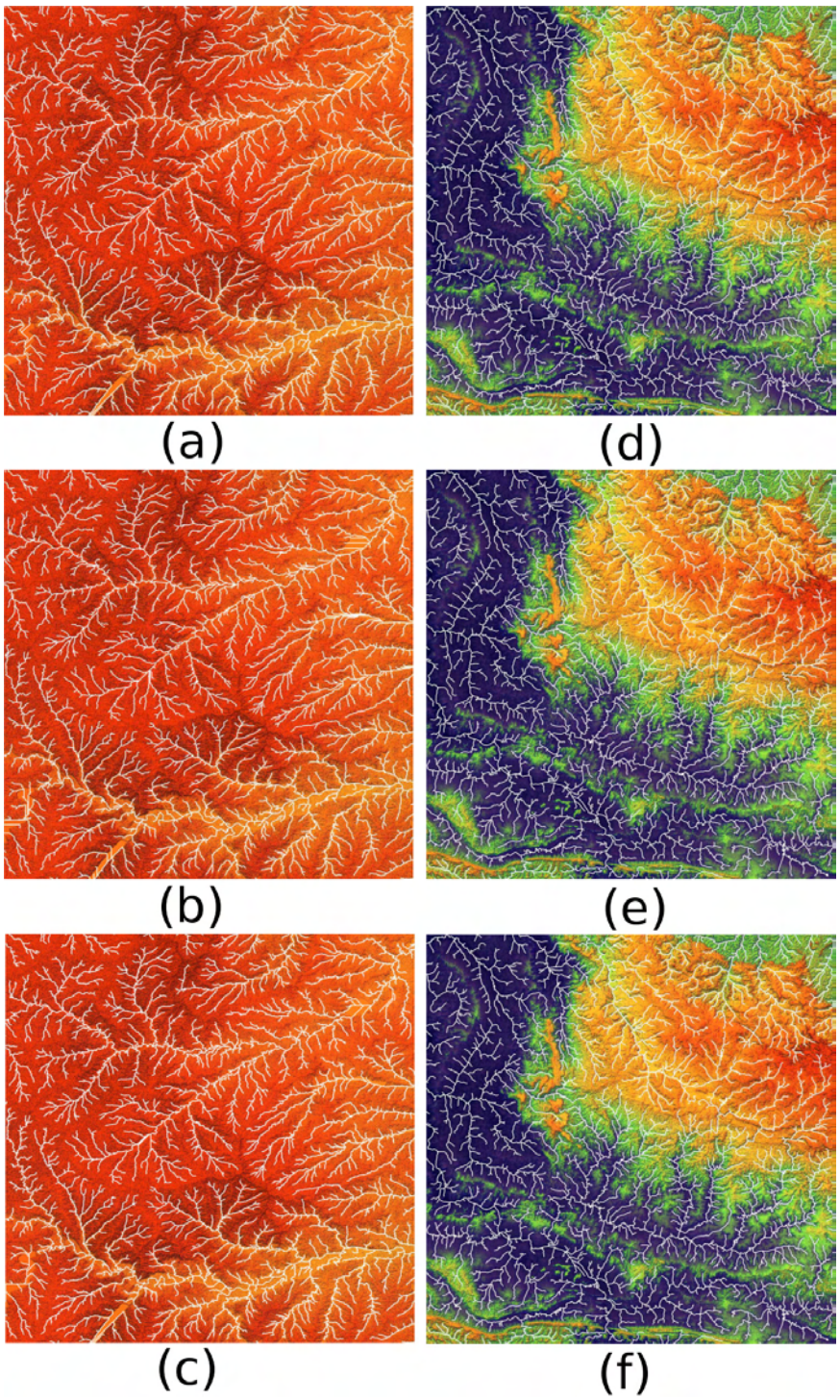

Figura 3.8. Drainage networks of two terrains R3, in (a), (b) and (c), and Tapajos, in (d), (e) and (f), computed by three methods: (a) and (d) EMFlow, (b) and (e) TerraFlow, (c) and (f) r.watershed.seg.

3.5 Conclusion

This paper presents *EMFlow*, a new algorithm for drainage network computation on huge terrains stored in external memory. *EMFlow*'s performance was compared against the most efficient methods described in the literature: *TerraFlow* and *r.watershed.seg* using many different terrains sizes and, in all situations, *EMFlow* was much faster (in some cases, more than 20 times) than both.

EMFlow adopts a new strategy for terrain subdivision, and uses a cache strategy to improve the external memory access.

4. Efficiently computing the drainage network on massive terrains using external memory flooding process.

Abstract

We present a very efficient algorithm, named *EMFlow*, and its implementation to compute the drainage network (i.e. the flow direction and flow accumulation) on huge terrains stored in external memory. As it is known, due to the fast increase in the volume of high resolution terrestrial data available, the internal memory algorithms do not run well for huge terrains on most computers and, thus, optimizing the massive data processing algorithm simultaneously for data movement and computation has been a challenge for GIS.

In *EMFlow*, the flow direction is computed using an adaptation of our previous method *RWFlood* which uses a flooding process to obtain this direction and the flow accumulation is computed based on a very fast method proposed by Haverkort and Janssen (2012).

To reduce the total number of I/O operations, *EMFlow* adopts a new strategy to subdivide the terrains in islands which are processed separately and the terrain cells are grouped into blocks, which are stored in a special data structure managed as a cache memory.

The *EMFlow* execution time was compared against the two most recent and most efficient published methods: *TerraFlow* and *r.watershed.seg* and it was, in average, 27 times faster than both methods. Since processing large datasets can take hours, this improvement is very significant.

4.1 Introdution

Many important applications in Geographical Information Science (GIS), such as hydrology, visibility and routing, require terrain data processing. These applications have become a challenge for GIS because they have to process a huge volume of high resolution terrestrial data. On most computers, the internal memory algorithms do not run well on such volumes of data since a large number of I/O operations

is necessary. For example, NASA's Shuttle Radar Topography Mission (SRTM) acquired 30 meter resolution terrain data for much of the world, generating about 10 terabytes of data. The datasets can be even bigger considering the technological advances that allow data acquisition at sub-meter resolution.

Thus, it is important to optimize the massive data processing algorithms simultaneously for computation and data movement between external and internal memory since processing data in external memory takes much more time. That is, the algorithms for external memory processing must be designed and implemented to minimize the number of I/O operations for swapping data between main memory and disk.

More precisely, the algorithms for external memory processing should be designed and analyzed considering a computational model where the algorithm complexity is evaluated based on data transfer operations instead of CPU processing operations. A model often used, proposed by Aggarwal and Vitter [Aggarwal & Vitter, 1988], defines an I/O operation as the transfer of one disk block of size B between the external and internal memory; the performance is measured by number of such I/O operations. The internal computation time is assumed to be comparatively insignificant. The algorithm complexity is defined based on the number of I/O operations executed by fundamental operations such as scanning or sorting n contiguous elements stored in external memory. Those are $\text{scan}(n) = \theta(n/B)$ and $\text{sort}(n) = \theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$, where M is the internal memory size.

Hydrological applications generally require the drainage network computation of a terrain, consisting of the flow direction and flow accumulation. Intuitively, they are the path that water flows through the terrain and the amount of water that flows into each terrain cell supposing that each cell receives a rain drop [Moore et al., 1991]. As broadly described [Magalhães et al., 2012b; Arge et al., 2003; Metz et al., 2011; Danner et al., 2007], it is a very time-consuming process, mainly on huge terrains requiring external memory processing. Indeed, in many situations, the flow direction can not be straightforwardly determined as for example, in a local minimum terrain cell.

In this paper, we present a new method, named *EMFlow*, for computing the drainage network on huge terrains represented by a digital elevation matrix stored in external memory. This new method is based on the adaptation of the *RWFlood* algorithm [Magalhães et al., 2012b] where the idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses to the terrain matrix executed by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts a new (original) strategy to subdivide the terrain matrix

into smaller pieces (islands) that can be processed separately and uses The *Cache-AwareAccumulation* algorithm for calculating the flow accumulation [Haverkort & Janssen, 2012].

The performance of *EMFlow* was compared against the most recent and efficient methods *TerraFlow* [GRASS Development Team, 2010] and *r.watershed.seg* [GRASS Development Team, 2010], both included in the open source GIS GRASS [GRASS Development Team, 2010]. As the tests showed, the *EMFlow* can be more than 20 times faster than the fastest of them. Since processing of large terrains can take hours, this is a significant improvement.

4.2 Background and Previous Work

4.2.1 Drainage Network Computation

As described previously, the drainage network of a terrain delineates the path that water flows through the terrain (the flow direction) and the amount of water that flows into each terrain cell (the flow accumulation). As formulated by Arge et al. [2003], the flow direction problem is to assign the flow directions to all cells in the terrain such that the following three conditions are fulfilled:

1. Every cell has at least one flow direction;
2. No cyclic flow paths exist; and
3. Every cell in the terrain has a flow path to the edge of the terrain.

The flow direction can be modeled considering single flow direction (SFD) or multiple flow directions (MFD). In SFD, for each terrain cell it is assigned a direction towards the steepest downslope neighbor, while in MFD, each cell has directions to all downslope neighbors. The use of SFD or MFD is essentially a modeling choice since the computational complexity of the flow routing problem is the same in both models. This paper will use SFD.

There are several methods to obtain the drainage network [Arge et al., 2003; Metz et al., 2011; Danner et al., 2007; Jenson & Domingue, 1988; Tarboton, 1997]. As described by those authors, the major challenge in the process is the flow routing in local minimum and flat areas. A local minimum is a cell with no *downslope neighbor* and a flat area is a set of adjacent cells with a same elevation. Given a cell c, a neighbor cell is called a downslope neighbor if it has a strictly lower elevation than c and a cell in a flat area that has a downslope neighbor is called a *spill-point*.

Also, the flat areas can be classified as a *plateau* or a *sink* where the plateau has, at least, a spill point and a sink doesn't. Intuitively, water will accumulate in a sink until it fills up and water flows out of it [Jenson & Domingue, 1988] and in the plateau the water should flow towards spill points.

Usually, most drainage network computation methods, as for example Arge et al. [2003]; Soille & Gratin [1994]; Metz et al. [2011]; Danner et al. [2007], use a preprocessing step to remove the sinks and the flat areas. Initially, the elevation of the cells belonging to a sink are increased to transform it into a plateau. Next, the directions on the plateau are assigned to ensure that there is a path (along flow directions) from each cell to the nearest spill point.

After obtaining the flow direction, the next step is to compute the flow accumulation in each terrain cell, that is, the amount of water flowing to each cell supposing that all cells receive a drop of water and this water follows the direction obtained in the previous step. Several methods for flow accumulation computation are based on graph topological sorting [O'Callaghan & Mark, 1984; Arge et al., 2003; Soille & Gratin, 1994] while others [Muckell et al., 2007, 2008] model this problem as a linear equations system.

According to Planchon et al. [Planchon & Darboux, 2002], the drainage network computation requires a considerable amount of processing, mainly due the preprocessing step to remove depressions and flat areas. In fact, in most methods based on this strategy, more than 50% of the total processing time is spent by this step. To avoid this time-consuming step, recently Magalhães et al. [Magalhães et al., 2012b] proposed a new method, named *RWFlood*, which is shortly described in section 4.3.1. As shown in Magalhães et al. [2012b], this method is more than 100 times faster than other recent methods but it does not scale well when the terrain does not fit in internal memory.

4.2.2 Computing Drainage Network Algorithms in External Memory

Several GIS implement algorithms for flow direction and flow accumulation but most of these algorithms were designed assuming that the terrain can be stored in internal memory and therefore they often do not scale well to large datasets [Arge et al., 2003]. On the other hand, there are some methods recently developed to process huge volume of data in external memory such as *TerraFlow* [GRASS Development Team, 2010] and *r.watershed.seg* [GRASS Development Team, 2010] both available in GRASS GIS.

4.2.2.1 TerraFlow

The *TerraFlow* is an efficient method, proposed by Arge et al. [Arge et al., 2003; Toma et al., 2001], to compute hydrological elements as drainage network and watershed in large terrains stored in external memory. It was implemented based on the model proposed by Aggarwal and Vitter [Aggarwal & Vitter, 1988]. For performance improvements, it uses the special library *TPIE* for data management, replacement and movement between internal and external memory.

The flow direction is computed in several steps. Initially, the plateaus and sinks are identified and the flow directions on non-flat areas are determined. Next, the flow directions on plateaus are assigned and then the depressions are identified and filled (removed). Finally, the flow directions on these areas are determined.

The flow accumulation is computed taking the elevation grid and the flow direction grid as input. Then, assuming that each cell receives one unit of water which flows according to the flow direction, the cells are processed using a strategy called *time forward processing* which uses a priority queue to process the cells in a topological order.

As described by the authors, the *TerraFlow* complexity is $\Theta(sort(n))$ and it uses some temporarily files whose total size may be up 80 times larger than the original terrain file.

4.2.2.2 GRASS module r.watershed

The *r.watershed* is another GRASS module to obtain the drainage network. It was initially developed for internal memory processing and adapted for external memory [Metz et al., 2011] using the GRASS *segment* library [GRASS Development Team, 2010], which allows an efficient processing of huge matrices in external memory.

The *segment* library provides a set of functions to manage huge matrices stored in external memory. Basically, the matrix is subdivided into segments (blocks) that are stored in temporary disk files. To improve the efficiency, a given number of these segments are kept in internal memory. Thus, to access a given matrix position, firstly, it is determined which segment contains that position and, then, the list of segments stored in internal memory is swept to check if the corresponding segment is already loaded. If yes, the position is accessed as usual, otherwise, the corresponding segment need to be transferred to internal memory. To avoid the segment list sweeping at each matrix access, the last accessed segment is kept in the

first position of the list and, thus, consecutive accesses in a same segment are more efficient.

When loading a segment in memory, there may be no space available to store the new segment and, in this case, the segment having the longest time without being accessed is evicted to open space for the new segment. In the *segment* library implementation, the segments have a “access time"field represented by an integer and every time a new segment is accessed (that is, a segment that is not in the front of the list) its access time is set to zero and the access time of all other segments are incremented by 1. Thus, in some cases, the segment access can have a large CPU overhead.

4.3 The *EMFlow* method

As described in section 4.2.1, most methods for flow direction computation use a very time-consuming preprocessing step to remove depressions and flat areas. However, in Magalhães et al. [2012b] we presented a new method, named *RWFlood*, which is much more efficient than other classical methods, mainly because it does not perform this preprocessing step and the depressions and flat areas are naturally handled during the processing. Thus, as mentioned in Section 3.1, the purpose of this work is to adapt the *RWFlood* method for external memory processing.

4.3.1 *RWFlood* method

To avoid the time-consuming preprocessing step, *RWFlood* computes the drainage network using a reverse order. Instead of determining the downhill flow it uses a flooding process. More precisely, the method is based on the following observation: if a terrain is flooded by water coming from outside and getting into the terrain through its boundary then the course of the water getting into the terrain will be the same as the water coming from rain and flowing downhill (that is, the flow direction). Thus, the idea is to suppose the terrain is surrounded by water (as an island) and to simulate a flooding process raising the water level iteratively. When the water level raises, it gradually floods the terrain cells and when it reaches a depression, that depression is filled by “water".

Figure 4.1 illustrates the flooding process: in Figure 4.1(a) the whole terrain is an island and next, in 4.1(b), the water level achieves the lowest cell in the terrain boundary. The raising water process continues and in 4.1(c) the water starts to get into the terrain and a terrain depression is filled — see 4.1(d). The flooding process

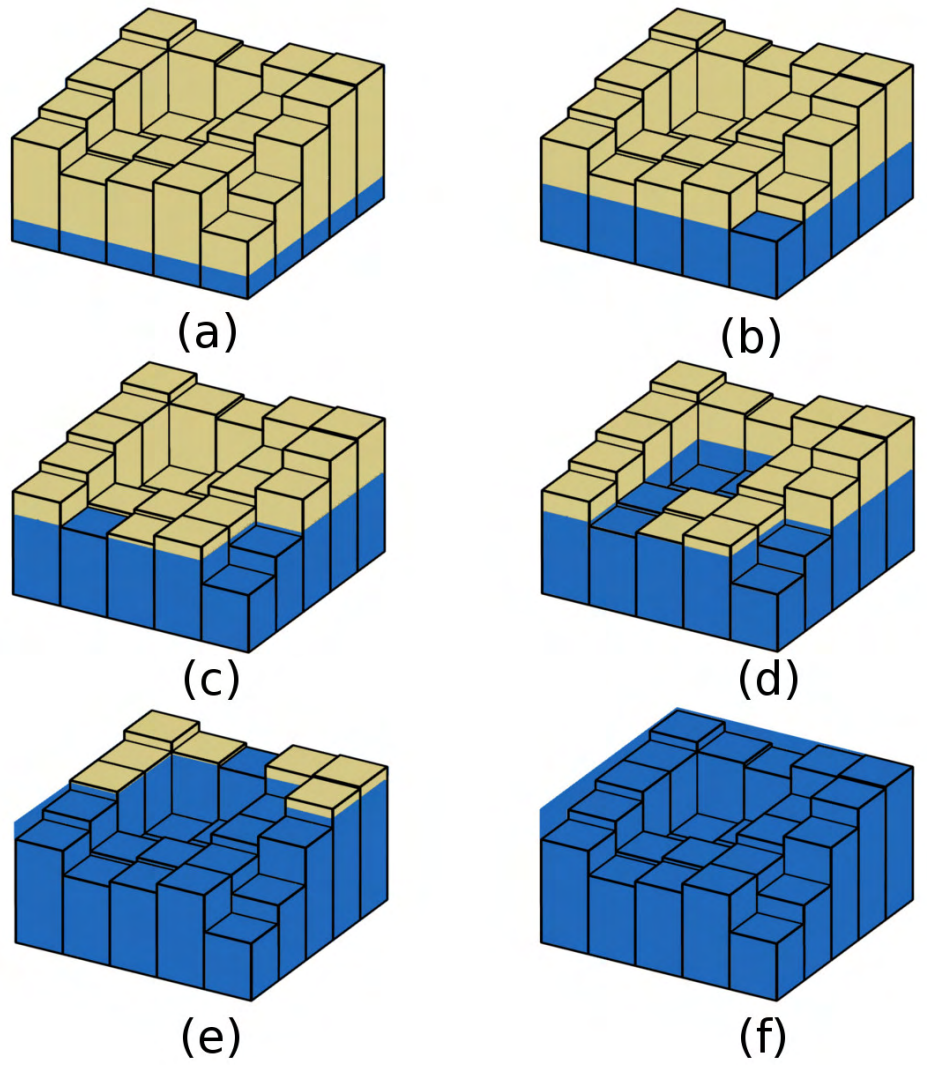

Figura 4.1. The flooding process: (a) the whole terrain is an island; (b) the water level is on the lowest cell in the terrain boundary; (c) the water level is raised; (d) a depression is flooded; (e) the flooding process creates two islands; (f) the flooding process is complete.

can generates new islands as in 4.1(e). Finally, the process ends when the whole terrain is flooded — see 4.1(f).

More formally, in the beginning, the water level is set to the elevation of the lowest cell in the terrain boundary. Then, two steps are executed iteratively: flooding a cell and raising the water level. When flooding a cell c, all cells neighbors to c are processed as following: given a neighbor cell d, if the elevation of d is smaller than the elevation of c, then the elevation of d is raised to the elevation of c; also, the flow direction of d is set to the cell c.

Algorithm 3 RWFlood - computes the flow direction

```
1: Let Q[minElev...maxElev] be an array of queues
2: for all cell c in the terrain boundary do
3:    c.dir ← NULL
4:    Q[c.elev].insert(c)
5:    c.dir ← OutsideTerrain
6: end for
7: for z = minElev → maxElev do
8:    while Q[z] is not empty do
9:       c ← Queues[z].remove()
10:      for all cell d neighbor to c such that d.dir = NULL do
11:         d.dir ← c
12:         if d.elev < z then
13:            d.elev ← z
14:         end if
15:         Q[d.elev].insert(d)
16:      end for
17:   end while
18: end for
```

After flooding all cells with the same elevation as c, the next step is executed, that is, the water level is raised to the elevation of the lowest cell higher than c and the process continues from this cell. To get this cell quickly, the method uses an array Q of queues to store the cells that need to be processed later. Thus, Q contains one queue for each elevation — queue $Q[m]$ will store the cells with elevation m that were already visited and need to be processed later. Initially, each cell in the terrain boundary is inserted into the corresponding queue. Supposing the lowest cells have elevation k, the process starts at queue $Q[k]$ and, after processing all cells in that queue, the process proceeds with the next non-empty queue in the array Q (intuitively, meaning that the water level is raised). Let $Q[z]$ be this next non-empty queue, then the front cell is dequeued (conceptually, it is flooded) and its neighbors are visited. That is, given a neighbor cell v, if v has already been visited, it is done; on the other hand, if v has not been visited yet, and if its elevation is not lower than z, it is inserted in its corresponding queue; otherwise, if its elevation is lower than z, its elevation is set to z and the cell is inserted into $Q[z]$. This latter case corresponds to flooding a depression point.

Thus, the next cell to be processed can be easily obtained by getting the next cell in the current queue (if it is not empty) or the first cell in the next non-empty queue. See algorithm 3.

The flow direction of each cell can be determined during the flooding process

since, when a cell c is processed, all cells adjacent to c which are inserted in a queue can have their flow direction set to c. That is, conceptually, the flow direction is set to the opposite direction as the water gets into the cells and, thus, the water in the adjacent cells will flow to the cell c. Initially, the flow direction of all cells in the terrain boundary is set to out of the terrain (i.e., indicating that in those cells the water flows out of the terrain).

After computing the flow direction, *RWFlood* uses an algorithm based on graph topological sorting to compute the flow accumulation. Conceptually, the idea is to process the flow network as a graph where each terrain cell is a vertex and there is a directed edge connecting a cell c_1 to a cell c_2 if and only if c_1 flows to c_2. Initially, all vertices in the graph have 1 unit of flow. Then, in each step, a cell c with in-degree 0 is set as visited and its flow is added to the $next(c)$'s flow where $next(c)$ is the cell following c in the graph. After processing c, the edge connecting c to $next(c)$ is removed (i.e., $next(c)$'s in-degree is decremented) and if the in-degree of $next(c)$ becomes 0, the $next(c)$ cell is similarly processed.

4.3.2 Adapting *RWFlood* for external memory processing

As presented in Magalhães et al. [2012b], the *RWFlood* method is very efficient when the whole terrain can be processed in internal memory. However, its performance decreases significantly whenever the terrain does not fit in internal memory and it is necessary to perform external processing. The main reason for this inefficiency is the non-sequential access to the terrain matrix. Indeed, according to the flooding process, the cells are accessed (processed) following the elevation order from the lowest to highest elevation. Also, when a cell is processed, its neighbors need to be accessed but, although these cells are close in the two-dimensional matrix representation, they may not be close in the memory because, usually, a matrix is stored using a linear row-major order.

To circumvent this problem and reduce the number of disk accesses, we propose a new method, named *EMFlow*, whose basic idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses carried out by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts a new (original) strategy to subdivide the terrain matrix in smaller pieces which can be processed separately. Also, to compute the flow accumulation, we implemented the external memory method *CacheAwareAccumulation* described in [Haverkort & Janssen, 2012].

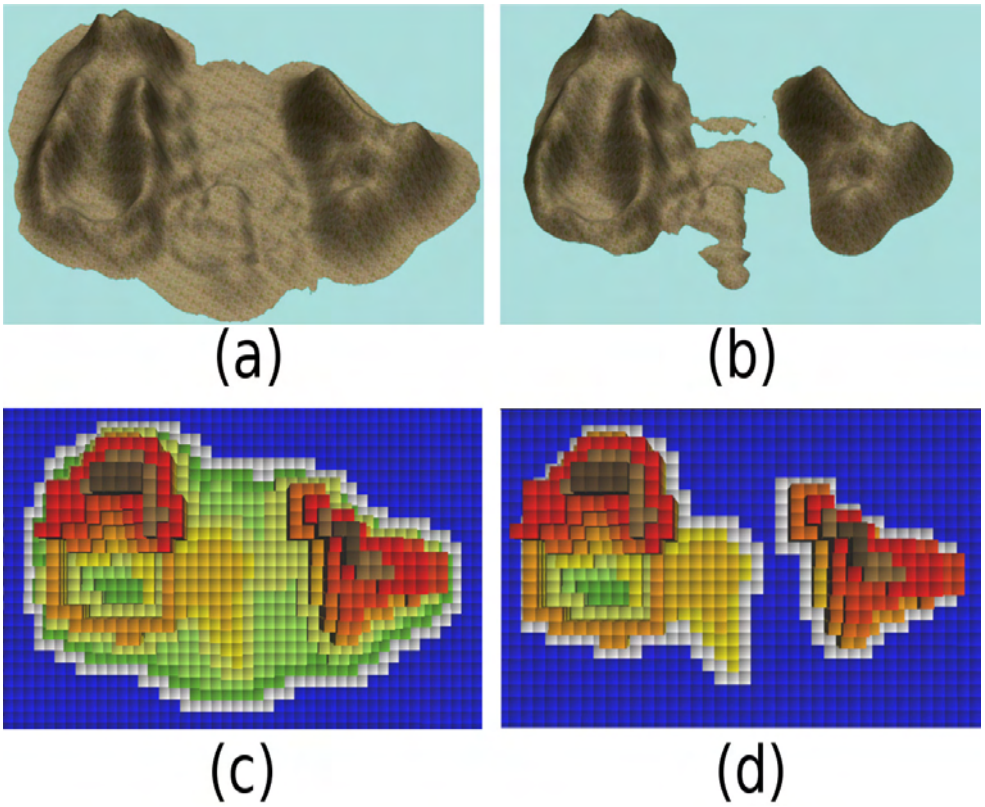

Figura 4.2. a) Flooding the terrain; (b) The flooding process generated two islands; (c) and (d) The cells in the flooding boundary are labeled with white.

4.3.2.1 The flow direction

Conceptually, the main idea of *RWFlood* is to store the cells in the boundary of the flooded regions — see Figure 4.2(c) and (d). At each step, the lowest cell in this boundary is processed. When a cell c is processed, all neighbors of c that were not processed yet and whose elevation is smaller or equal to the elevation of c are flooded, that is, the flooding boundary moves toward these cells. This flooding process can generate interior islands — see Figures 4.2(a) and (b) — and these islands can be processed (flooded) separately since the flooding process of an island does not affect any other island. Based on this fact, the *EMFlow* subdivides the terrain into islands that are processed one by one.

More precisely, initially, the whole terrain is processed as one island which is flooded using the *RWFlood* strategy. Next, at some moment (described below), the algorithm analyzes if the flooding process generated internal islands. Notice that, an island is a group of connected cells which were not flooded (that is, processed) yet. Thus, the islands can be identified computing the connected components composed of non processed cells. After identifying the islands, each one is processed

independently.

However, this subdivision strategy does not assure that the process can be entirely executed in internal memory. The islands can be too large and have too many cells that do not fit in internal memory. Thus, to improve the algorithm performance, the terrain matrix accesses are managed by the *TiledMatrix*[Magalhães et al., 2012a] library which was designed to store and manage huge matrices in external memory. Basically, in *TiledMatrix*, a matrix is subdivided in blocks whose size allows that a given number of blocks can be stored in internal memory. Then, all blocks are stored in external memory and they are loaded to internal memory on demand. That is, when a cell c needs to be accessed, the library determines which block contains that cell and, if the block is not in the internal memory, it is loaded. Since eventually there may not be space to store a new block, the data structure storing the blocks is managed as a cache memory. More precisely, the library adopts a replacement policy to evict a block and open room for the new block[1]. *EMFlow* uses the *LRU - least recently used* policy.

Furthermore, to reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [Collet, 2012]. Before storing a block in the disk, it is compressed and when a block is loaded to internal memory it is uncompressed.

4.3.2.2 The flow accumulation

In the *RWFlood*, the flow accumulation is computed using a method based on topological sorting and, as the tests showed, this method is very efficient when the terrain can be processed in internal memory. However, as in the flow direction computation, for external memory processing this strategy is not very effective since it can require many non-sequential accesses. Therefore, in *EMFlow*, the flow accumulation is computed using another (more efficient) strategy based on the method described in Haverkort & Janssen [2012]. Shortly, the main idea of this method is to subdivide the terrain in blocks each one small enough to fit into internal memory and, also, such that the boundary cells of each block are shared with the neighboring blocks (except on the outer boundary of the terrain). - see figure 4.3(a). Thus, the flow accumulation is computed in three steps:
Step 1: considering the flow direction matrix (which was given as input), the flow accumulation of all cells in the boundary block is computed using the conventional

[1]The library provides the following policies: LFU - Least Frequently Used, FIFO - first in first out, and random selection.

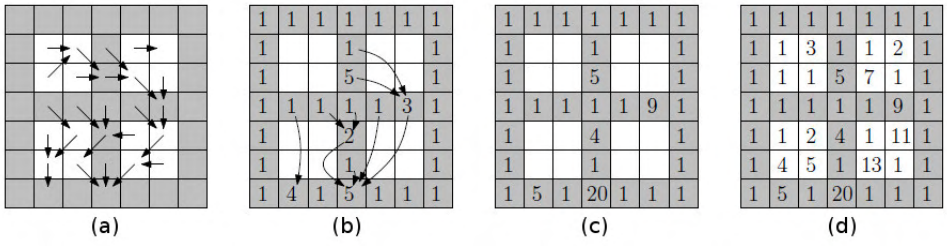

Figura 4.3. Flow accumulation steps: (a) terrain subdivision (the cells in grey are boundary cells shared among the blocks); (b) the flow accumulation value in the boundary cells and the corresponding flow direction for the boundary cells; (c) updating the boundary cells flow accumulation; (d) computing the flow accumulation in the interior cells.

topological sorting strategy (each block is processed independently in internal memory). Additionally, given a block B, for each cell c in the boundary of B that flows to an interior cell of B, it is determined the boundary cell of B to where the water in the cell c flows - see figure 4.3(b).

Step 2: then, the flow accumulation value of each boundary cell c is updated adding the corresponding values of the (same) cell c in different blocks and also adding the values of all boundary cells that flows to c, that is, this last part corresponds to compute the flow accumulation considering only the boundary cells - see figure 4.3(c).

Step 3: now, the flow accumulation of the interior cells in each block is (re)computed using the conventional approach considering the boundary cell values obtained in the previous step.

As the tests showed, computing the flow accumulation using the method described above instead of using topological sorting in external memory made *EMFlow* about 10% faster.

4.3.2.3 Implementation details

In the *EMFlow* implementation, we adopted some strategies for performance improvement:

(1) Islands identification: an island generated during the flooding process is composed of a group of connected cells that were not flooded yet, and this group is surrounded by flooded cells. That is, an island is a maximal connected component of non-flooded (or non-processed) cells and, to have an island, it is necessary to have a group of flooded (processed) cells surrounding the island. But, since the connected component computation is a time-consuming process, mainly when the terrain matrix can not be stored in internal memory, the algorithm adopts a less accurate

strategy where the islands are identified using a lower resolution terrain. More precisely, the algorithm creates an auxiliary matrix C where each cell corresponds to a square block in the terrain matrix and a C cell stores the number of corresponding terrain cells which were not processed yet. That is, the cells of C are initialized with the number of terrain cells in each corresponding square block and, during the flooding process, this value is decremented whenever a corresponding terrain cells is processed. When the value in a C cell becomes zero it indicates that all cells in the corresponding terrain block were already processed. Thus, the islands identification process is reduced to the computation of the maximal connected component of non zero cells in the matrix C.

Notice that if two blocks are disconnected in C then the cells in each block will belong to different islands and, thus, they can be processed separately. On the other hand, two different islands in the terrain may be identified as connected in C (because C has a lower resolution), that is, they may be identified as one island. But, the final result does not change if two islands are processed as one island. This may only lead to a larger processing time because the number of cells which need to be stored in internal memory may increase.

Since the islands identification is not a trivial process, it is executed only occasionally. The idea is to execute it when there are evidences that some islands were generated. In the *EMFlow* algorithm, the lenght of the flooded region boundary was used to trigger this process, that is, it is executed when the number of cells in the flooded region boundary achieves a given threshold.

(2) Scheduling the islands processing: as described previously, during the terrain flooding, the island generation follows a recursive sequence, but these islands can be processed in any order since they are independent and their processing is self-contained. Thus, in *EMFlow*, the processing of the islands is scheduled trying to process first those islands that (probably) will require a smaller number of external memory accesses. Since the cells in the islands boundary are already stored in internal memory then the external memory accesses will be required only if there exist some cells adjacent to the islands boundary that are not in internal memory yet. Then, the algorithm computes, for each island, the percentage of cells adjacent to the island boundary that are already in internal memory and the islands with higher percentage are processed first. In fact, since the matrix cells accesses are managed by the *TiledMatrix* library using blocks, the algorithm computes the percentage of blocks containing cells adjacent to the boundary that are already in internal memory.

(3) The islands boundary size: when an island is processed, all cells on its boundary need to be loaded into internal memory and also, during the cell processing, the

neighbor cells must be loaded too. Thus, if the algorithm tries to process many islands simultaneously and if these islands have long boundaries (with too many cells), this large number of cells may not fit in internal memory. In this case, some cells (in fact, some blocks) need to be evicted and reloaded again later. To avoid these time-consuming operations, the algorithm defines a threshold to limit the number of islands that could be processed at a same time, that is, which could be loaded in internal memory.

4.3.3 *EMFlow* versus r.watershed.seg

Both methods *EMFlow* and *r.watershed.seg* (included in GRASS) try to improve their performance by using libraries to manage the external memory accesses; *EMFlow* uses the *TiledMatrix* library [Magalhães et al., 2012a] and *r.watershed.seg* uses *segment* [GRASS Development Team, 2010]. Although these two libraries have similar purposes and both are based on to subdivide the matrix in blocks and manage them using a cache strategy, they have some important differences described below:

- Both libraries store a set of blocks in internal memory using an array. However, when a terrain cell is accessed, they use different methods to check if the block containing that cell is already loaded in internal memory. In the *segment*, the array positions where the blocks are stored are kept in a list of pairs (b_n, b_p) where b_n is the block number (referent to the terrain matrix) and b_p is the block position in the internal memory array. Then, to check if the block is loaded in internal memory (and get it), the list is searched. Thus, in the worst case, the access to a terrain cell can take $O(n)$ time, where n is the number of blocks stored in internal memory. Trying to reduce this time, the library keeps the last block accessed in the front of the list to avoid the worst case of searching operation when the next accessed cell is also in the same block. On the other hand, in *TiledMatrix*, the terrain cell access always takes a constant time since the blocks' positions are stored in a matrix of size $\frac{N}{h} \times \frac{M}{w}$ where N and M are respectively the terrain matrix height and width and h and w are respectively the block height and width. Thus, if a block is not loaded in internal memory, the matrix position corresponding to that block is set to -1, otherwise, it is set to the array position where that block is stored. As this operation is executed many times during the whole process, its efficiency directly affects the algorithm performance.

- The block replacement policy is LRU in both libraries, but the libraries use different strategies for block marking. In *segment*, the blocks are marked with an integer value that is updated every time a block is accessed. Initially, all blocks are marked with zero and when a new block b is accessed (that is, when a cell contained in a new block b is accessed), the value of all blocks, except b, are incremented. Thus, the block replacement will evict the block with the smaller value. In *TiledMatrix*, the blocks are marked using a *timestamp*, that is, when a block is accessed, it is marked with the current timestamp. Then, the block with the smaller timestamp will be evicted. Therefore, the block marking takes $O(n)$ time in *segment* and a constant time in *TiledMatrix*.

- To reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [Collet, 2012]. Thus, before writing a block to the disk, it is compressed using LZ4 and, after reading a block from the disk, it is uncompressed. As presented in Magalhães et al. [2012a], the *EMFlow* is more than twice as fast when this compression strategy is used. On the other hand, the *segment* does not adopt any similar strategy.

4.4 Experimental Results

EMFlow was implemented in C++ and compiled with g++ 4.5.2. It was compared against the most efficient algorithms described in the literature: *TerraFlow* [GRASS Development Team, 2010] and *r.watershed.seg* [GRASS Development Team, 2010] both available in GRASS. The tests were executed in a machine with an Intel Core 2 Duo with 2,8GHz and 5400 RPM SATA HD (Samsung HD103SI) running Ubuntu Linux 11.04 (64 bits). This machine was configured with different internal memory size, 1GB and 2GB, to evaluate the algorithm's performance in different scenarios.

The tests used different datasets generated from two distinct USA regions (regions 02 and 03 in Figure 4.4) sampled at $30m$ horizontal resolution using 2 bytes per elevation value. These two regions were selected because they are in the central part of the USA, do not include ocean, and therefore have few NODATA elements.

Tables 4.1 and 4.2 show the execution time (in seconds) of the three algorithms in the R2 and R3 regions using respectively 1GB and 2GB of RAM. In these tests, the *TiledMatrix* library, used by *EMFlow*, was configured as following: for 1GB of RAM it used blocks with 200×200 cells and for 2GB the block size was 400×400 cells. In the tables, the symbol ∞ is used to indicate that the execution was interrupted after

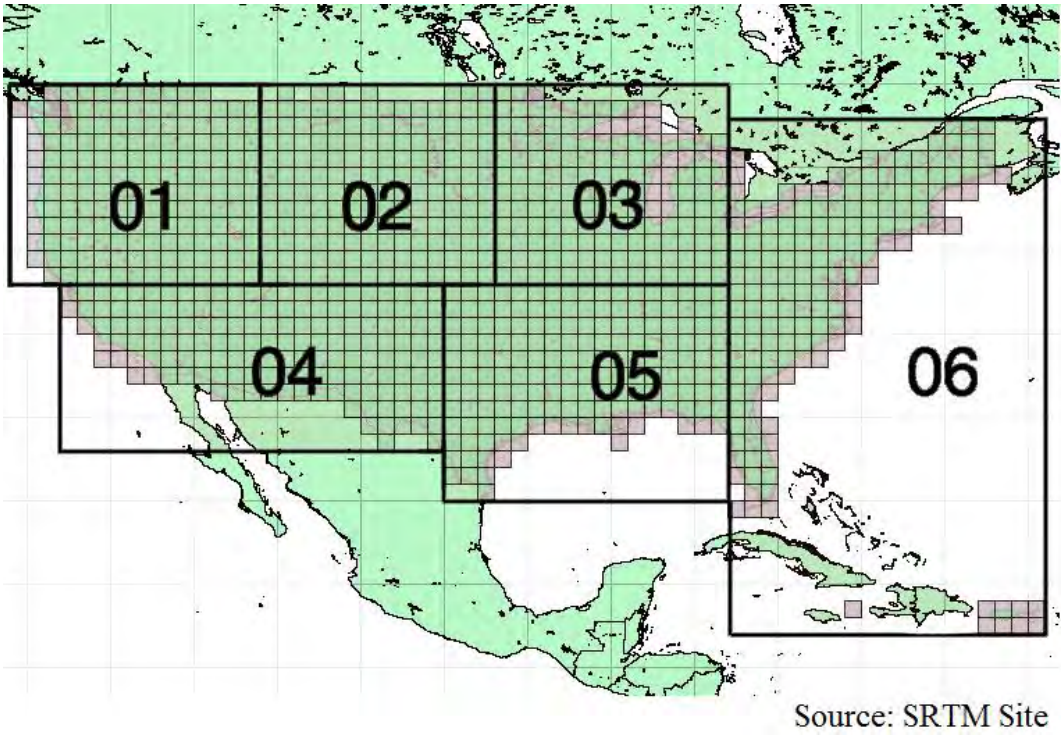

Figura 4.4. SRTM USA Regions.

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,93	24,43	6,25	0,92	28,22	5,91
5000^2	18,80	661,37	622,66	19,11	907,50	508,90
10000^2	81,67	2329,71	25784,71	81,09	3358,42	55182,80
15000^2	288,14	7588,33	∞	303,39	9046,13	∞
20000^2	542,84	12937,30	∞	566,38	14404,76	∞
25000^2	971,14	22220,89	∞	996,78	24974,77	∞
30000^2	1501,61	35408,11	∞	1811,35	41251,21	∞
40000^2	3045,39	67076,04	∞	3824,65	78056,28	∞
50000^2	5875,84	98221,64	∞	6244,78	110394,74	∞

Tabela 4.1. Processing time (in seconds) for different terrain sizes from regions R2 and R3 with 1GB of memory.

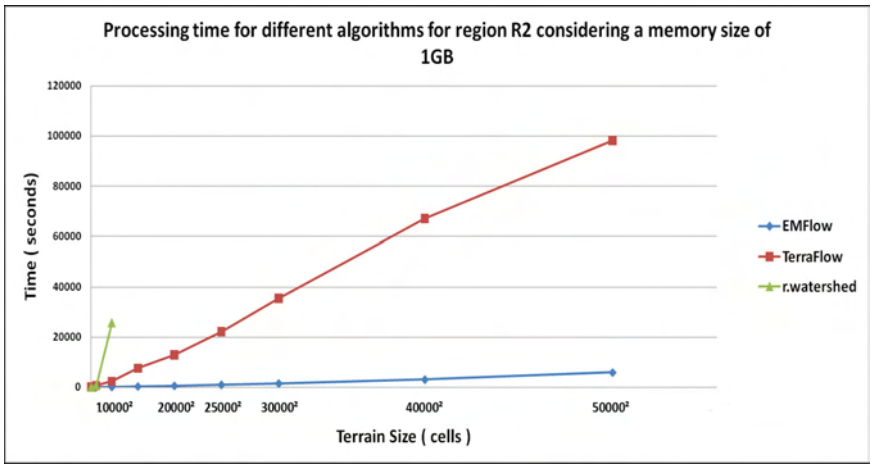

Figura 4.5. Chart corresponding the region R2 considering memory size 1GB.

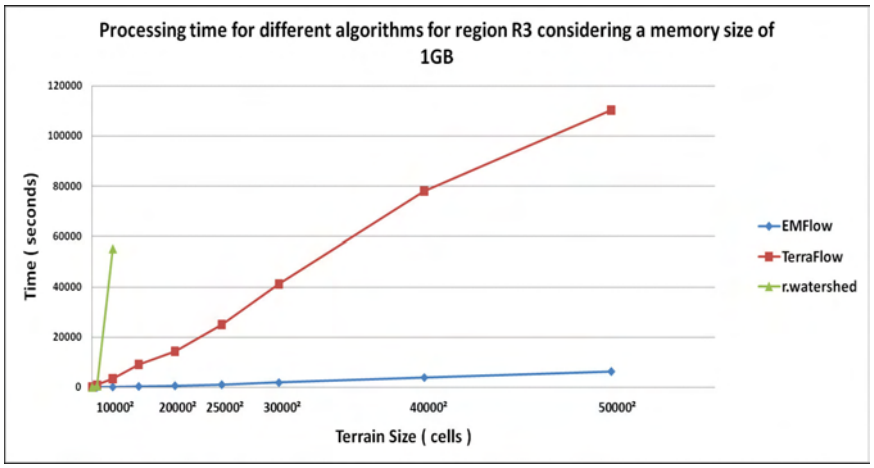

Figura 4.6. Chart corresponding the region R3 considering memory size 1GB.

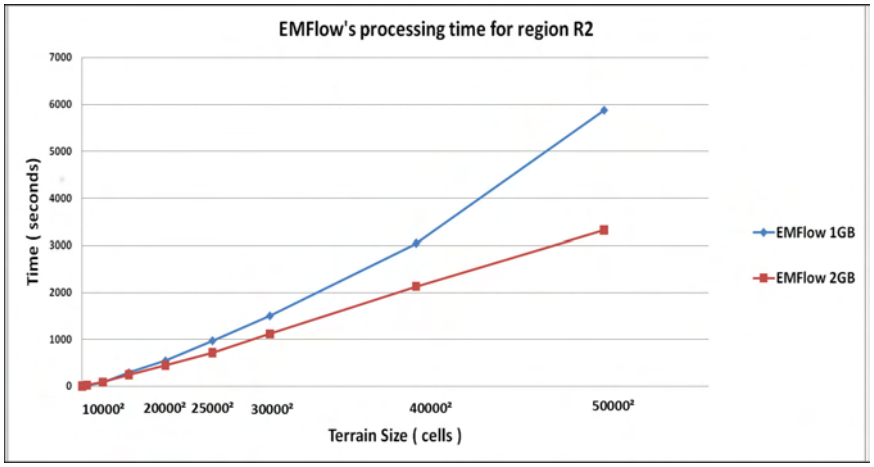

Figura 4.7. Chart corresponding the region R2 considering memory size 1GB and 2GB for EMFlow.

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,74	19,32	6,03	0,98	19,44	5,79
5000^2	20,02	400,84	630,60	19,98	442,97	513,88
10000^2	87,66	2251,66	5290,46	86,94	2552,93	3911,23
15000^2	242,02	5870,34	34252,23	233,36	6869,33	32518,89
20000^2	443,58	13066,63	∞	413,37	13873,60	∞
25000^2	713,98	19339,79	∞	686,86	22492,14	∞
30000^2	1113,31	30364,31	∞	1094,58	33337,07	∞
40000^2	2126,80	56421,36	∞	1943,17	59149,27	∞
50000^2	3315,72	82673,22	∞	2996,99	86670,30	∞

Tabela 4.2. Processing time (in seconds) for different terrain sizes from regions R2 and R3 with 2GB of memory.

150000 seconds (40 hours). Figures 4.5 and 4.6 present the charts corresponding to the table 4.1. Figure 4.7 compares *EMFlow*'s performance using 1GB and 2GB of RAM.

Notice that *EMFlow* was faster than the other two algorithms in all situations. In average, it was 27 times faster and, for very huge terrains (such as 40000×40000), *EMFlow* was more than 30 times faster than *TerraFlow* while *r.watershed.seg* was still running after 40 hours.

It is worth to mention that, since *EMFlow* is based on *RWFlood*, the drainage networks computed by these two algorithms are the same. Additionally, as presented in Magalhães et al. [2012b], the drainage networks obtained by *RWFlood* are almost the same as those computed by *TerraFlow* and *r.watershed*. For example, Figure 4.8 presents the drainage networks computed by the three methods: *EMFlow*, *TerraFlow* and *r.watershed.seg* in two terrains: the R3 region and a terrain from the Tapajos[2] region. Figures 4.8 (a) and (d) show the networks computed by *EMFlow* in the regions R3 and Tapajos respectively, the Figures 4.8 (b) and (e) show the networks from R3 and Tapajos computed by *TerraFlow* and Figures 4.8 (c) and (f) show the networks computed by *r.watershed* in those regions. We see that, the corresponding networks are very similar.

[2]Tapajos is an important tributary river of the Amazon basin.

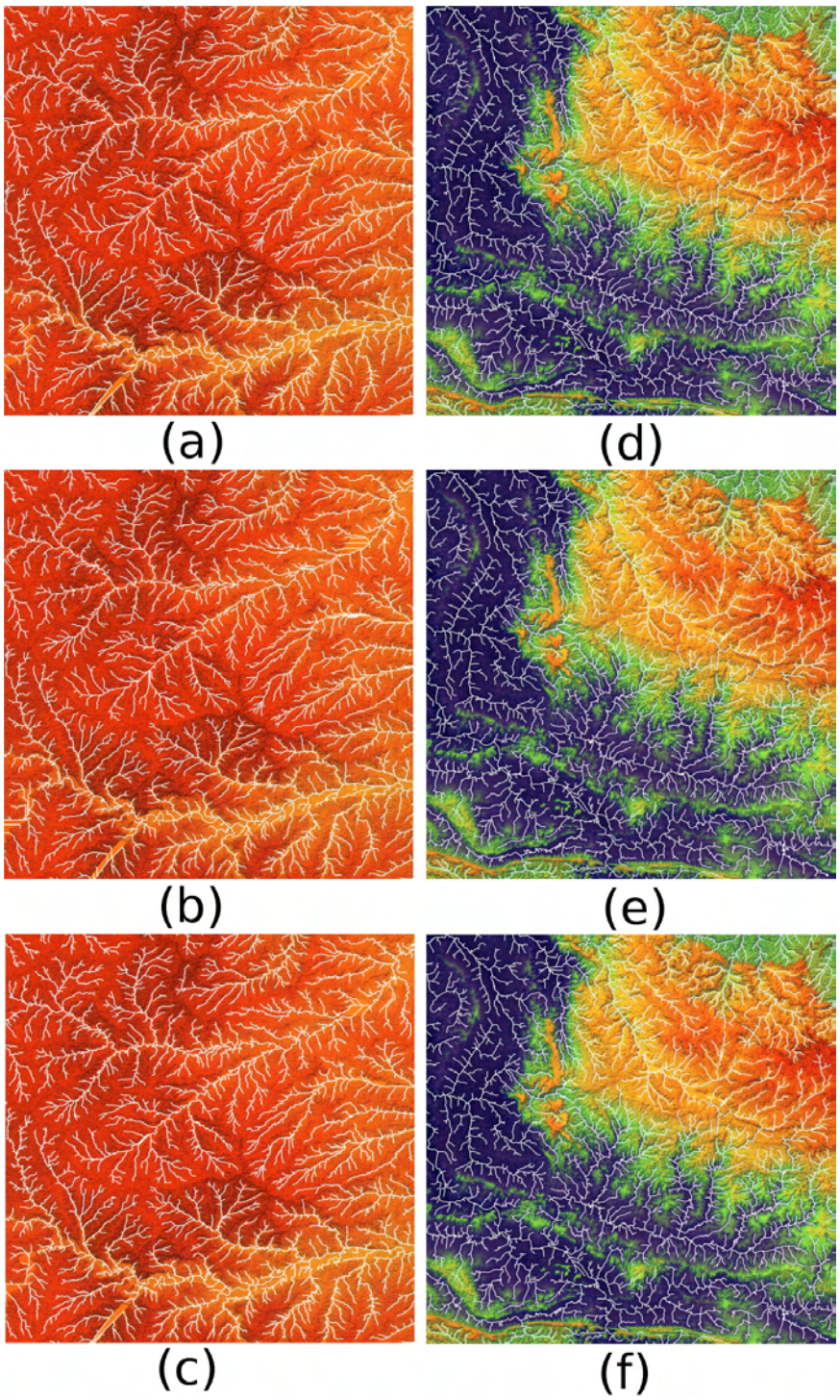

Figura 4.8. Drainage networks of two terrains R3, in (a), (b) and (c), and Tapajos, in (d), (e) and (f), computed by three methods: (a) and (d) EMFlow, (b) and (e) TerraFlow, (c) and (f) r.watershed.seg.

4.5 Conclusion

This paper presented *EMFlow*, a new algorithm for drainage network computation on huge terrains stored in external memory. *EMFlow* uses a cache strategy to improve the external memory access and also it adopts a new strategy for terrain subdivision (based on island creation during the flooding process).

EMFlow's performance was compared against the most efficient methods described in the literature: *TerraFlow* and *r.watershed.seg* using many different terrains sizes and, in all situations, *EMFlow* was much faster (in some cases, more than 30 times) than either.

5. Conclusões e Trabalhos Fututos

Neste trabalho foi apresentado um método chamado *EMFlow* para cálculo da direção de fluxo e do fluxo acumulado em memória externa. *EMFlow* foi projetado e analisado levando em consideração um modelo computacional onde a complexidade do algoritmo é avaliada com base na transferência de dados em vez do número de operações da CPU, o modelo adotado foi o proposto por Aggarwal & Vitter [1988].

No capítulo 2 foi apresentada a primeira versão do método *EMFlow*, essa versão faz adaptação para memória externa do método *RWFlood* [Magalhães et al., 2012b] que é bastante eficiente quando o terreno é processado em memória interna. Porém, esta eficiencia não é escalável quando o processamento do terreno necessita ser realizado em disco. No processo de inundação do terreno, as células são acessadas (processadas) seguindo a ordem das elevações das mais baixas para as mais altas sendo que ao processar uma célula, suas vizinhas devem ser acessadas. Apesar do método *RWFlood* não ter um padrão de acesso sequencial, ele possui um padrão de acesso espacial: as células acessadas estão perto umas das outras na representação bidimensional. Baseado nisso foi feito uma adaptação para modificar a forma de gerenciamento da memória (reorganizando a matriz) para tirar proveito da localidade espacial de acesso. Esta primeira versão foi comparada com *TerraFlow* [Arge et al., 2003] e *r.watershed.seg* [GRASS Development Team, 2010], os principais métodos descritos na literatura , e foi cerca de 10 vezes mais eficiente que o *TerraFlow* para os terrenos grandes e o *r.watershed.seg* não foi capaz de processar os terrenos em um tempo razoável.

No capítulo 3 foi incluído uma melhoria importante no método *EMFlow* utilizando uma estratégia (original) de divisão do terreno durante o processamento. O processo de inundação do método *RWFlood* gera ilhas no interior do terreno, estas ilhas causam um maior número de acessos devido ao fato do processamento ficar alternando entre elas. No entanto, essas ilhas podem ser processadas separadamente porque o processo de inundação em uma ilha não afeta outra ilha. Baseado neste fato, o *EMFlow* subdivide o terreno em ilhas que são processadas separadamente, tornando o método mais eficiente e mais robusto. Novamente, essa versão melhorada foi comparada com o *TerraFlow* e *r.watershed.seg* e ela foi cerca de 17 vezes mais eficiente que o *TerraFlow* para os terrenos grandes e o *r.watershed.seg*, como já foi falado, não foi capaz de processar os terrenos em um tempo razoável.

No capítulo 4 foi incluído ao método *EMFlow* um eficiente algoritmo para calculo de fluxo acumulado baseado no método proposto por Haverkort & Janssen [2012] levando em conta o número de acessos ao disco, o que produziu melhorias significativas no tempo de processamento do método. De acordo com os testes essa versão do *EMFlow* foi em média 27 vezes mais eficiente que os métodos *TerraFlow* e *r.watershed.seg*.

Concluindo, esse trabalho produziu um método bastante eficiente para obtenção da rede de drenagem em terrenos armazenados em memória externa e também deu origem à algumas estratégias que podem ser adotadas em outras aplicações. Em particular o processo de inundação pode ser aplicado, por exemplo, em métodos de processamento de imagens onde o objetivo é identificar certas características da imagem.

Outra estratégia que pode ser adotada em outras aplicações é o processo de divisão em ilhas que poderia ser utilizado para desenvolver algoritmos para processamento paralelo e também para obtenção das bacias de contribuição (Watershed).

Referências Bibliográficas

Aggarwal, A. & Vitter, J. S. (1988). The input/output complexity of sorting and related problems. *Communications of the ACM*, 9:1116–1127.

Arge, L.; Chase, J. S.; Halpin, P.; Toma, L.; Vitter, J. S.; Urban, D. & Wickremesinghe, R. (2003). Efficient flow computation on massive grid terrain datasets. *Geoinformatica*, 7.

Brostuen, D. & Cox, S. (2000). Minimizing subjectivity in digital orthopho imagery. Em *20th Annual Esri International User Conference*, San Diego, California, USA.

Câmara, G.; Davis, C. & Monteiro, A. M. (2001). *Introdução à Ciência da Geoinformação*. Instituto de Pesquisa Espacial- INPE, São Jose dos Campos, SP, Brasil,Disponivel em: http://www.dpi.inpe.br/gilberto/livro/introd/ Acesso em: 10 março 2013.

Collet, Y. (2012). Extremely fast compression algorithm.

Danner, A.; Molhave, T.; Yi, K.; P.; Agarwal, K.; Arge, L. & Mitasova, H. (2007). Terrastream: from elevation data to watershed hierarchies. Em *Proc. of ACM GIS*, pp. 117–124.

Dementiev, R. (2007). *Algorithm Engineering for Large Data Sets*. VDM Verlag, Saarbrucken, Germany, Germany.

Dementiev, R.; Kettner, L. & Sanders, P. (2005). Stxxl : Standard template library for xxl data sets. Technical report, Fakultat fur Informatik, Universitat Karlsruhe. http://stxxl.sourceforge.net/ (acessado em 05/03/2012).

Driemel, A.; Haverkort, H.; Löffler, M. & Silveira, R. I. (2011). Flow computations on imprecise terrains. Em *Proceedings of the 12th international conference on Algorithms and data structures*, WADS'11, pp. 350--361, Berlin, Heidelberg. Springer-Verlag.

ESRI (2012). Arcgis. Disponível em: http://www.esri.com/software/arcgis/arcgis-for-desktop/index.html. (acessado em 17/05/2012).

Felgueiras, C. A. (2001). Modelagem numérica de terreno. Em In G. Câmara, C. Davis, A. M. V. M., editor, *Introdução à Ciência da Geoinformação*, volume 1. INPE.

Gomes, T. L.; Magalhães, S. V. G.; Andrade, M. V. A.; Franklin, W. R. & Pena, G. C. (2012a). Computing the drainage network on huge grid terrains. Em *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '12, pp. 53--60, New York, NY, USA. ACM.

Gomes, T. L.; Magalhães, S. V. G.; Andrade, M. V. A.; Franklin, W. R. & Pena, G. C. (2013). Efficiently computing the drainage network on massive terrains using external memory flooding process. *Submitted for publication.* Computers & Geosciences.

Gomes, T. L.; Magalhães, S. V. G.; Andrade, M. V. A. & Pena, G. C. (2012b). Determinação da rede de drenagem em grandes terrenos armazenados em memória externa. Em *XII Brazilian Symposium on GeoInformatics*, Campos do Jordão,São Paulo, Brazil.

GRASS Development Team (2010). *Geographic Resources Analysis Support System (GRASS GIS) Software.* Open Source Geospatial Foundation, http://grass.osgeo.org (acessado 17/05/2012).

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

I.Moore; Grayson, R. & Ladson, A. (1991). Digital terrain modelling: a review of hydrological, geomorphological and biological aplications. Em *Hydrological Processes 5*, number 3-30.

Jenson, S. & Domingue, J. (1988). Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11):1593–1600.

Jet Propulsion Laboratory NASA (2012). *NASA Shuttle Radar Topography Mission (SRTM).* National Geospatial-Intelligence Agency (NGA) and National Aeronautics and Space Administration (NASA), http://srtm.usgs.gov/mission.php(acessado 17/05/2012).

Magalhães, S. V. G.; Andrade, M. V. A.; Ferreira, C. R.; Pena, G. C.; Luange, T. G. & Pompermayer, A. M. (2012a). Uma biblioteca para o gerenciamento de grandes

matrizes em memória externa. Technical report, Departamento de Informática, Universidade Federal de Viçosa.

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

Metz, M.; Mitasova, H. & Harmon, R. S. (2011). Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. *Hydrology and Earth System Sciences*, 15(2):667--678.

Moore, I. D.; Grayson, R. B. & Ladson, A. R. (1991). Digital terrain modelling: a review of hydrological, geomorphological and biological aplications. *Hydrological Processes*, 5:3–30.

Muckell, J.; Andrade, M.; Franklin, W. R.; Cutler, B.; Inanc, M.; Xie, Z. & Tracy, D. M. (2007). Drainage network and watershed reconstruction on simplified terrain. Em *17th Fall Workshop on Computational Geometry*, IBM TJ Watson Research Center, Hawthorne NY.

Muckell, J.; Andrade, M.; Franklin, W. R.; Cutler, B.; Inanc, M.; Xie, Z. & Tracy, D. M. (2008). Hydrology-aware terrain simplification. Em *5th International Conference on Geographic Information Science*, Park City, Utah, USA.

O'Callaghan, J. & Mark, D. (1984). The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing*, 28:328–344.

Planchon, O. & Darboux, F. (2002). A fast, simple and versatile algorithm to fill the depressions of digital elevation models. *Catena*, 46(2-3):159--176.

Soille, P. & Gratin, C. (1994). An efficient algorithm for drainage network extraction on dems. *Journal of Visual Communication and Image Representation*, 5(2):181–189.

SPRING - DPI/INPE (2012). *Spring - Sistema de Processamento de Informaçõs Georeferenciadas*. INPE/DPI (Divisão de Processamento de Imagens), http://www.dpi.inpe.br/spring/(acessado 05/01/2013).

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.

Toma, L.; Wickremesinghe, R.; Arge, L.; Chase, J. S.; Vitter, J. S.; Halpin, P. N. & Urban, D. (2001). Flow computation on massive grids. Em *GIS 2001 Proceedings of the 9th ACM international symposium on Advances in geographic information systems*.

Vitter, J. S. (2001). External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33.