NOTICE

The definitive published version is:

Yıldırım A.A., Watson D., Tarboton, D.G., Wallace, R.M., A virtual tile approach to raster-based calculations of large digital elevation models in a shared-memory system, Computers & Geosciences, Volume 82, September 2015, Pages 78-88, ISSN 0098-3004,
http://dx.doi.org/10.1016/j.cageo.2015.05.014.

Corresponding Author:
Ahmet Artu Yıldırım
E-mail: ahmetartu@aggiemail.usu.edu

A Virtual Tile Approach to Raster-based Calculations of Large Digital Elevation Models in a Shared-Memory System

Ahmet Artu Yıldırım[a,*], Dan Watson[a], David Tarboton[b], Robert M. Wallace[c]

[a] *Department of Computer Science, Utah State University, Logan, Utah, USA*
[b] *Utah Water Research Laboratory, Utah State University, Logan, Utah, USA*
[c] *US Army Engineer Research and Development Center, Information Technology Lab, Vicksburg, Mississippi, USA*

Abstract

Grid digital elevation models (DEMs) are commonly used in hydrology to derive information related to topographically driven flow. Advances in technology for creating DEMs has increased their resolution and data size with the result that algorithms for processing them are frequently memory limited. This paper presents a new approach to the management of memory in the parallel solution of hydrologic terrain processing using a user-level virtual memory system for shared-memory multithreaded systems. The method includes tailored virtual memory management of raster-based calculations for data sets that are larger than available memory and a novel order-of-calculations approach to parallel hydrologic terrain analysis applications. The method is illustrated for the pit filling algorithm used first in most hydrologic terrain analysis workflows.

Keywords: Multithreaded parallel digital elevation model analysis, pit filling algorithm, user-level virtual memory management

1. Introduction

Grid digital elevation models (DEMs) are commonly used in hydrology to derive information related to topographically driven flow [1, 2, 3]. When dealing with large digital elevation model (DEM), datasets, computational efficiency and memory capacity become important considerations. Prior work in TauDEM [4] has advanced parallel methods for terrain processing using a message passing (MPI) approach that allows memory to be distributed across multiple processors in medium-sized cluster computers [5, 6, 7].

In desktop computers, virtual memory systems are the standard approach to working with data that is too big to fit in memory. Operating systems typically implement virtual memory using page files that hold on disk contents of memory. However repeated swapping (thrashing) occurs when these get large because the system has limited general capability to anticipate the pages needed next. Most operating systems implement the virtual machine in kernel that makes it difficult, sometimes impossible, to change its functionality and page replacement policy. This necessities the implementation of a user-level tailored virtual memory system to handle a programs' locality better for fine-grain control [8].

There is a need to process large DEMs on desktop computers that are often limited in total memory. This is the primary problem addressed in this work. It is also desirable to have parallel terrain analysis algorithms that use multi-threading to take advantage of common multi-core processors [9] for greater efficiency. This is a secondary consideration in this work.

This paper presents a new approach to the management of memory and the parallel solution of the raster-based computations for shared-memory multithreaded systems, such as desktop computers. The contributions of this method in the context of parallelism are a tailored user-level tile based virtual memory manager for raster-based calculations for data sets that are larger than available memory and a novel order-of-calculations approach to parallel hydrology analysis applications.

We implemented a modified version of the Planchon and Darboux pit filling algorithm [10] as implemented in [4, 6] as an application of our tiled virtual memory manager and evaluated its effectiveness for pit removal in DEMs of varying size, with a varying number of operating system threads and memory capacity. The results demonstrate several benefits over the use of standard virtual memory approaches.

Furthermore, this study examines a load-balancing technique to minimize the idle times which might occur in case of uneven load between compute threads due to data variability. We used the GDAL library [11] to enable a wide range of raster file formats within the implemented memory manager. We implemented our memory man-

*Corresponding author
Email addresses: ahmetartu@aggiemail.usu.edu (Ahmet Artu Yıldırım), dan.watson@usu.edu (Dan Watson), dtarb@usu.edu (David Tarboton), robert.m.wallace@usace.army.mil (Robert M. Wallace)

ager using the Microsoft Windows 7 operating system as it is widely used for desktop Geographic Information System terrain analysis and enabling the processing of large DEMs on Windows systems was a goal of this work. The source code of the virtual memory project and the pit-filling algorithm can be found at `https://bitbucket.org/ahmetartu/hydrovtmm`.

The paper is organized as follows: Section 2 provides background and literature review. Section 3 gives specifics of the modified Planchon and Darboux Algorithm used here. Section 4 describes the design of the multi-threaded tiled virtual memory manager for raster-based calculations that is contributed here. Section 5 gives performance results. Finally, we discuss conclusions based on the obtained results in Section 6.

2. Background

This review addresses three subjects that are needed to set the context for this work. First we review existing DEM pit filling approaches focusing on the Planchon and Darboux Algorithm [10] modified and used in this work. We then examine other efforts that use parallel methods for hydrologic terrain analysis and general methods for virtual memory management to enable the processing of large data sets that do not fit in physically available computer memory.

Pits, defined as grid cells or sets of grid cells completely surrounded by higher grid cells often occur during DEM production and are generally considered to be artifacts of the data collection process [12]. Drainage conditioning to remove pits is an important preprocessing step in the hydrologic analysis of DEMs, and is representative of a broad class of raster-based algorithms (e.g. [13, 1]) designed to determine topographically driven flow. Once drainage conditioning has been performed, a DEM that has no pits is referred to as being hydrologically conditioned. The most common approach to drainage conditioning is pit filling, whereby pit grid cells are raised to a level where they are at a minimum equal to the lowest surrounding grid cell and can drain. A well-known effective pit filling algorithm is described by Planchon and Darboux [10]. Pit removal using the Planchon and Darboux (PD) algorithm is one of the multiple hydrologic terrain analysis functions in the TauDEM package. Time-complexity of the direct implementation of the PD algorithm is reported to grow with the number of cells N in DEM as $O(N^{1.5})$ [10]. Planchon et al. also provided an improved implementation that embedded a recursive dry upward cell function that was reported to achieve a time-complexity of $O(N^{1.2})$ [10]. Alternative pit filling algorithms, some claiming better efficiency have been presented by others [14, 15, 16].

The Planchon and Darboux (PD) approach fills pits by covering the whole surface with a layer of "water" up to a level greater than or equal to the highest point in the DEM, then removes the excess water in an iterative manner. Doing so, the algorithm naturally leaves the water in the depressions at the height of their outlet. Let $Z \in \mathbb{R}^2$ be the set of input elevation points (i.e., the input DEM with size m, where each member is an elevation point x_i, $1 \leqslant i \leqslant m$) and let $W \in \mathbb{R}^2$ be the output DEM consisting of "filled" elevation points y_i. The goal of the PD algorithm is to increase each elevation point x_i with a minimal difference of elevation. The PD algorithm initializes all grid cells to a large value (greater than the highest point in the domain). It then uses iterative scans across the domain to lower elevation values to the lowest elevation greater than or equal to the original terrain hydrologically conditioned so that they drain to one of their neighbors.

Wallis et al. [4, 6] implemented a parallel version of the PD algorithm. This adopted a distributed memory domain partitioning approach to parallelism and divided the domain into horizontal stripes, one stripe per parallel process. The PD algorithm was applied separately to each stripe in parallel, with a step to exchange information across stripe boundaries at the end of each iteration so as to ensure convergence to the same global solution as obtained by a serial implementation. The original PD algorithm and the Wallis et al. parallel implementation visit each grid cell on each iteration. Scans of the grid cycle through all eight possible combinations of row and column scan orders. PD also offered an improved implementation that used a recursive dry upward tree search each time a cell was set to the original elevation to enhance efficiency. However each iterative pass across the DEM still examined each grid cell.

Subsequent to the Wallis et al. [6] work, the TauDEM team [4] identified the visiting of each grid cell on each iteration as an inefficiency and developed a stack based approach whereby unresolved grid cells are placed on a stack on the first scan, then removed from the first stack on each subsequent scan and placed on a second stack. Stacks are then switched. This limits the scanning to two directions rather than eight, but was found to result in a speedup of a factor of 2 for small datasets and 4.3 for a modestly large 1.5 GB dataset in comparison to the eight combination full grid scanning. The benefits of the stack thus seem to outweigh the inefficiency of fewer scan directions. The TauDEM team did not evaluate the recursive dry upward approach of the improved PD algorithm. Recursive methods use the system stack to expand system memory, posing a challenge for memory management in large data computations. They also pose a challenge for a domain partitioned parallel approach as cross partition calls are less predictable. They are also hard to implement on a stack as they would require additional code to track the stack position of each grid cell and to handle changing the order in which grid cells on the stack are processed. The two direction stack based modified PD algorithm was incorporated into the publicly released version of TauDEM [4] that was the starting point for this work. The focus of this paper is on virtual memory management for large DEM data, and the modified PD algorithm as used by TauDEM [4, 6] is used as an example to illustrate a gen-

eral approach.

Beyond pit filling several parallel computing technologies have been employed in the implementation of the hydrologic algorithms to achieve higher performance by effectively utilizing available processors in the system including MPI for distributed memory architectures [17, 5, 6], OpenMP [18] or low-level standard threading library for multi-threaded shared memory architectures, and NVIDIA CUDA for general-purpose computing on graphics processing units (GPUs) [19, 20]. Xu et al. [18] present a grid-associated algorithm to improve the performance of a D8 algorithm [21] via OpenMP technology which is a thread-level standard of parallel programming. CUDA algorithms for drainage network determination are presented by Ortega et al. [20], achieving up to 8x speed-up improvement with respect to corresponding CPU implementation. Do et al. introduced a parallel algorithm to compute the global flow accumulation in a DEM using MPI [17] where hierarchical catchment basins are computed by means of a parallel spanning tree algorithm to model the flow of water. Each processor computes a local flow direction graph using the *D8* flow routing model.

There have been a number of general approaches to better manage virtual memory. The Tempest [22] interface was proposed for customizing the memory policies of a given application on parallel computers. Translation lookaside buffer (TLB) is a limited memory cache that is utilized for efficient memory address translation. However, TLB misses are also common in memory-hungry applications such as databases and in-memory caches. To eliminate these, Basu et. al [23] proposed mapping part of a process' linear virtual address space with a direct segment. Huang et. al [24] introduced a power-aware virtual memory system to reduce the energy consumed by the memory for data-centric applications.

Software programs may implement their own level of memory management to overcome the inefficiencies of system-level virtual memory. For example, ArcGIS reportedly uses quad-tree tiling to organize on disk file storage in a way that facilitated better virtual memory management [25]. However full details of their implementation are not available.

3. Modified Planchon and Darboux Algorithm

We employed a modified version of the PD algorithm, as used in TauDEM [4], in this study. The PD algorithm has two phases, initialization (Algorithm 1) and filling (Algorithm 2). Initialization starts by allocating memory space for elevation data, W. Then, if the cell is located at the edge, the algorithm assigns the value of cell in Z to each cell of W with the same cell location, otherwise a maximum number is assigned. Here edge is defined to include cells at the edge of the DEM or internal cells adjacent to cells with no data values, or cells marked as internally draining and not to be filled in a separate input file. Thus, it is assumed that water may drain from the edges of the input DEM or into internally draining or no data cells (because data may not fit neatly into the rectangular DEM domain). We employ stacks to hold indexes of the cells not yet solved. The indexes of the cells having excess "water" in W are pushed onto the first stack, (S_1), which is used later in the filling phase. An upper bound on the number of elements the stack needs to hold is the number of internal (non-edge) grid cells in the DEM.

Algorithm 1 Initialisation Phase of Stack-based Planchon and Darboux Algorithm

Require: Input DEM Z
Ensure: Elevation data W, stack S_1

```
1:  procedure INITIALIZEPLANCHON(Z)
2:      for Cell number i ← 1 to |Z| do
3:          if c_i ∈ Z is on the edge then
4:              W(i) ← Z(i)
5:          else
6:              W(i) ← +∞
7:              S_1 ← push(S_1, i);
8:          end if
9:      end for
10:     return W, S_1
11: end procedure
```

In the filling phase (Algorithm 2), the cell values of W are decreased in an iterative manner until no cell value is changed. The procedure pulls an unresolved grid cell from stack, S_1. The operation of $\min N_8(W(i))$ denotes finding the minimum value among 8 neighbours of the cell $W(i)$. In Line 6, the algorithm checks whether the value of $Z(i)$ is greater than or equal to the minimum elevation of its 8 neighbours plus a small parameter ϵ to enforce strictly positive slopes. This parameter may be 0, and is defaulted to 0 where minimally altered hydrologically conditioned surfaces are desired. If "true", this implies that the cell drains, so the value of $Z(i)$ is assigned to the cell $W(i)$. Otherwise the value $N_{min} + \epsilon$ is assigned to the cell, lowering its value to the elevation at which it drains (line 10). Cells that are set to their original elevation value (line 7) do not need to be revisited so are not placed on the stack S_2. Cells lowered to a neighbor value (plus ϵ) may need to be revisited in a later iteration so are pushed to the stack S_2. The stacks decrease in size progressively as cells are processed. After each iteration, in Line 16, the indexes pointing to the stacks are swapped.

The original PD algorithm did not employ a stack as described above. Rather at each iteration it scanned the entire DEM examining for each cell whether $W(i) > Z(i)$, and if true examined neighbor elevations and applied the excess water removal logic.

4. Virtual tile approach to the parallel raster-based algorithm

We implemented a parallel modified PD algorithm for a multi-threaded shared memory system with limited mem-

Algorithm 2 Filling Phase of Planchon and Darboux Algorithm

Require: Input DEM Z, elevation data W, stacks S_1 and S_2, ϵ-descending path parameter, i denotes to the visited cell index, $\min N_8(W(i))$ denotes to minimum elevation of the neighbouring cells of cell i

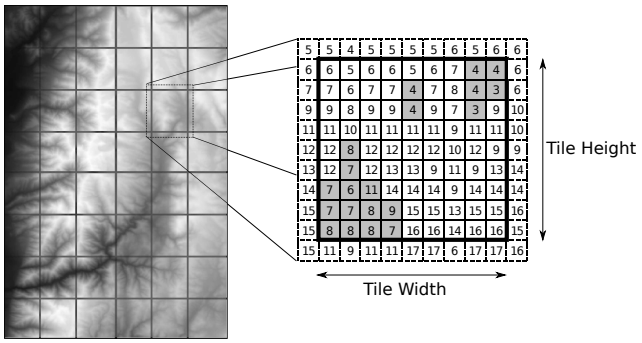
Ensure: Elevation data W

```
1:  procedure FILLPLANCHON(Z, W, S1, S2, ε)
2:      do
3:          for all cell number i in S1 do
4:              if W(i) > Z(i) then
5:                  N_min ← min N8(W(i))
6:                  if Z(i) ⩾ N_min + ε then
7:                      W(i) ← Z(i)
8:                  else
9:                      if W(i) > N_min + ε then
10:                         W(i) ← N_min + ε
11:                     end if
12:                     S2 ← push(S2, i)
13:                 end if
14:             end if
15:         end for
16:         call swapStacks(S1, S2)
17:     while any cell c ∈ W is changed
18:     return W
19: end procedure
```

ory to process large DEMs. The input DEM is divided into a number of generally square $q \times q$ tiles where q is tile size that can be considered a generalized domain decomposition approach when compared to the TauDEM stripe approach. Tiles may be rectangular along the edges to accommodate domain sizes that are not multiples of q. Each tile is processed individually by one thread. Tiles are distributed among the threads with respect to number of cells in the stacks using a load balancing mechanism. In order to process big DEMs, which might be larger than available physical memory, we adopt a tile-based user-level virtual memory management system. The memory system swaps out tiles and their related data to hard-disk to free the memory space automatically when the predefined memory limit is reached. Tiles are chosen for swapping out by the memory manager according to a least recently used rule.

In our algorithm, we use a single input/output (I/O) thread and multiple compute threads. *Compute threads* are responsible for the execution of the PD algorithm while the *I/O thread* is used to avoid performance bottlenecks due to overlapping disk I/O and compute operations. The I/O thread services all compute threads. The main thread is regarded as a compute thread and is responsible for spawning other compute threads and the I/O thread. Memory address space is shared among all threads, alleviating the overhead associated with interprocess communication in shared memory systems.

Figure 1: Partition of the DEM into tiles with one-cell border. Grid cells contain elevation values with nondraining cells (pits) in the tile detail depicted in gray.



We adopt data parallelism among compute threads where each thread works on one tile at a time and executes the same algorithm (Algorithm 3). Each tile is a rectangular subset of the DEM that consists of primary data and a one-cell border that overlaps into the primary data from adjacent tiles as illustrated in Figure 1, that is used to facilitate transfer of information between tiles. The primary data of all tiles completely covers the domain without overlap. A tile page, T_k, has the corresponding subset of elevation data, W_k, input DEM data, Z_k and two stacks. Tile pages are indexed by tile page number k that define the page uniquely in the address space. The *initializePlanchon* and *fillPlanchon* functions change values only within the primary data, but refer to values from adjacent grid cells using the edge data held locally as part of each tile page.

After the initialization pass (Lines 3-5) and each filling pass (Lines 9-13) the resulting elevation values are hydrologically conditioned within the local context of each tile, but may need to be modified as each tile is juxtaposed with its surrounding tiles. Hence, local elevation data on the edges must be updated from surrounding tiles. Neighboring threads exchange the border of the tiles. Thus, we implement a barrier to synchronize all threads at this point to make sure all the threads are at the same position. Barriers are shown in Algorithm 3. Although barriers are desired to be avoided to fully exploit the parallelism in algorithm design, in order to circumvent race conditions it is required. Then the *exchangeBorders* function is called for each tile by each owner thread. This accesses the data from each bordering tile and updates its edge data with the corresponding primary data from the adjacent tile.

Remaining cells to be processed at the next iteration can be determined by the number of cells in the stack. This determines the require processing time for each tile. After locally filling the pits of the tiles, imbalance might occur in which some threads finish the local filling before other threads. To avoid this inefficiency, we implement load balancing functionality that distributes the tiles to the threads evenly based on the number of cells in the stacks. Before distributing the tiles, all threads must wait

Algorithm 3 Multithreaded PD algorithm

Require: Input DEM Z, tile size TS, memory limit ML, number of compute threads NC, each with start index, SI_i and end index, EI_i, referencing the tiles it processes.
Ensure: Output DEM W (final elevation data)

```
1:  procedure PARALLELPLANCHON(Z, TS, ML, NC)
2:      for all Compute Threads i in parallel do
3:          for k ← SI_i, EI_i do
4:              call initializePlanchon(T_k)
5:          end for
6:          call exchangeBorders() with barrier()
7:          call performLoadBalancing() with barrier()
8:          do
9:              for k ← SI_i, EI_i do
10:                 if T_k is not finished then
11:                     call fillPlanchon(T_k)
12:                 end if
13:             end for
14:             call exchangeBorders() with barrier()
15:             call performLoadBalancing() with barrier()
16:         while any cell elevation in W is changed
17:     end for
18:     call barrier()
19:     call writeOutputDEM()
20: end procedure
```

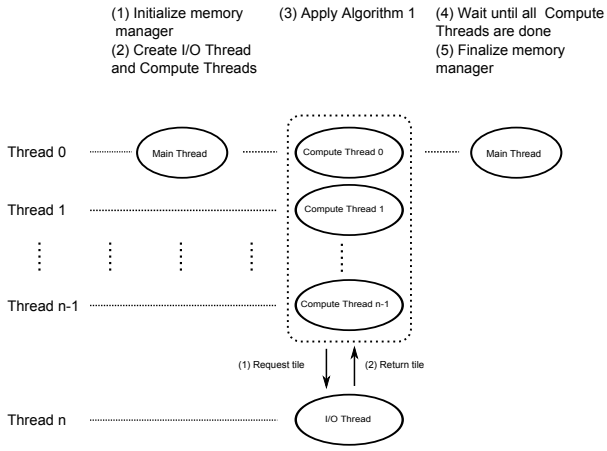
for the main thread that executes the *loadBalancing* function. *loadBalancing* examines the size of the unresolved stack in each tile and adjusts the tiles assigned to each thread by setting SI_i and EI_i prior to the next iteration. More discussion about the load balancing can be found in *Load balancing* section.

A virtual memory management strategy (Figure 2) was developed to support implementation of this algorithm in systems where the physical memory is limited. A memory manager is responsible for performing swapping operations as memory reaches its maximum capacity. In system-level virtual memory, each program on the operating system has its own address space that is consisting of memory chunks (pages) [26]. In our approach, each thread is provided a set of tiles on the DEM file and its associated data whose memory is managed by the virtual memory through a well-defined software interface.

All memory management functionality is performed by the I/O thread, using the GDAL library [11]) so that processing in the compute threads may proceed in parallel with I/O swapping of tiles to disk. One of the reasons for a single I/O thread is that GDAL did not support parallel I/O. However even with a parallel I/O library on the PC platforms we are targeting, the parallel I/O capability is limited and a single I/O thread avoids cross thread I/O bottlenecks.

The main thread initializes the memory manager and creates the tile page table. Tile page table is in shared

Figure 2: Shared virtual memory management strategy



memory accessible to the compute threads and I/O thread. The tile page table contains tile data structures mainly including tile state, memory addresses of original and conditioned DEM data and stack data. We implemented a tailored stack data structure where the data are kept in memory in a contiguous fashion in order to serialize the writing and reading of stack memory content efficiently. The stack data structure contains a dynamically increasing heap array (using the *realloc* utility), and size and capacity variables.

Figure 3: Tile distribution and sharing among the compute threads in pit-remove algorithm. Tiles are numbered with the owner compute thread number

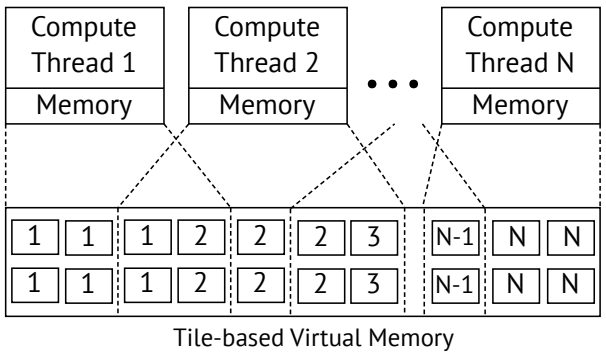


Figure 3 illustrates how tiles are assigned to compute threads. Note that the *exchangeBorders* function results in tiles assigned to one thread being accessed by adjacent threads. This is a non-locking read access so that contention that may occur when a thread attempts to lock a memory block that is owned by another thread [27] can be avoided. Data sharing in shared memory systems is one source of performance degradation and should be avoided where possible, thus the *exchangeBorders* function is called only at infrequent intervals.

Each tile page has state, μ_k where $\mu_k = \{notloaded, present, swappedout, finished\}$: *notloaded*, indicating that the page has not yet been requested by any compute thread; *present*, the page is resident in the memory; *swappedout*, the page is swapped out to the disk; or *finished*, all elevation points in the tile have been hydrologically conditioned and no more pit-filling is required. The condition *finished* is set on a tile when its stack size becomes 0. The algorithm iterates until all the tiles have achieved the state of *finished*, or until all stacks on all processes remain unchanged. The page table is illustrated in Figure 4(a) with tile states illustrated in Figure 4(b).

A distinguishing property in our model, as compared to most GIS software, e.g., ArgGIS, is that a tile page does not consist only of raster data, but also contains algorithm-specific data such as the input, output and associated data that comprise the memory space of the program.

A *message queue* is used for interprocess communication between compute threads and the I/O thread. The I/O thread constantly retrieves the messages in the message loop in a First In, First Out (FIFO) manner. The message passing between the compute threads and I/O thread is performed using *PostThreadMessage* and *PeekMessage* of Win32 functions. If a compute thread requests a tile retrieval, it sends a message to the I/O thread and waits until the page is returned. If the page is present in memory, the operation is completed without waiting. Each tile has a time stamp variable to keep track of the last request time. The time stamp is updated by the I/O thread when requested by any compute thread. Using this time stamp, the memory manager swaps out the oldest tiles according to the Least Recently Used (LRU) replacement algorithm.

The granularity of the virtual memory manager is an important parameter that affects the system's performance. A small page/tile size can result in larger page tables, which might cause the system to spend its CPU time mostly in I/O operations. On the other hand, too large a tile size can hinder the benefit of using memory management and memory usage can exceed the memory limit by a factor of tile page size.

Thrashing is a well-known phenomenon encountered when a process idles excessively waiting for the referenced page to be loaded by the operating system. In this algorithm, the effect of thrashing is alleviated by pre-fetching tiles using a pre-specified pattern based on knowledge of the sequence in which the algorithm accesses tiles. The prefetching technique has been applied for sequential programs in which predicting patterns of program execution and data access in the near future improves the system efficiency [28]. In the implemented virtual memory system, when a tile is requested, the memory manager caches the next α tiles (from the $flist$ array) in this pattern asynchronously. The goal of this prefetching technique is to increase the *hit rate* of tiles found in present memory when requested. We define the hit rate as the number of tile references whose memory present in RAM divided by the total number of tile references.

4.1. API to access to the virtual memory manager

The user-level virtual memory is designed to be used as a general-purpose virtual memory for raster-based computations to execute on a single machine with limited memory through a well-defined interface. The raster-based algorithm accesses the VM through a well-defined interface so as to enable the potential for application with other raster-based algorithms. The API is explained below.

- $initializeMemoryManager(S)$: The function accepts array S that contains the setting values of the virtual memory manager including maximum allowable memory, and the DEM file accessed to determine extent and tile sizes. Using the parameter S, the virtual memory is initialized and memory allocated for resources such as the messaging system and the data structures used by the main thread.

- $getTile(tid, flist[])$: A compute thread requests the tile with uniquely-defined tile id *tid* from VM. A tile can be owned by only one compute thread for writing at a time to avoid race condition error. During the tile-retrieval process with the *getTile* function, the tile's reference count is incremented by 1. The virtual memory manager prevents access to a shared resource from other threads using the *EnterCriticalSection* and *LeaveCriticalSection* Win32 functions. As an advantage with respect to most operating system level virtual memory systems, the tiles to be prefetched are determined by the algorithm from the $flist$ array containing tile ids. Prefetching occurs in parallel. The next tile on the list is prefetched by the virtual memory manager. After loading each tile, the virtual memory manager checks the memory limit and if this has been exceeded swaps out the least-recently accessed tile not in use. This strategy is a heuristic attempt to ensure that the next tiles needed by the compute threads will be in memory when they are requested.

- $unlockTile(tid)$: After the compute thread is finished with the tile, the tile is explicitly released by the owner thread. The virtual memory manager decrements the tile's reference count where the tile is requested with $getTile$ function. When the tile's reference count reaches 0, the tile is returned to the memory pool. Then, virtual memory manager decides to keep the tile in memory or swaps out to the disk based on the page replacement algorithm to free memory space.

- $saveTile(tid)$: If the tile is finished completely by the compute thread, the result is saved to the disk via $saveTile$ function and then the tile is marked as *finished* state which is the final state of a tile.

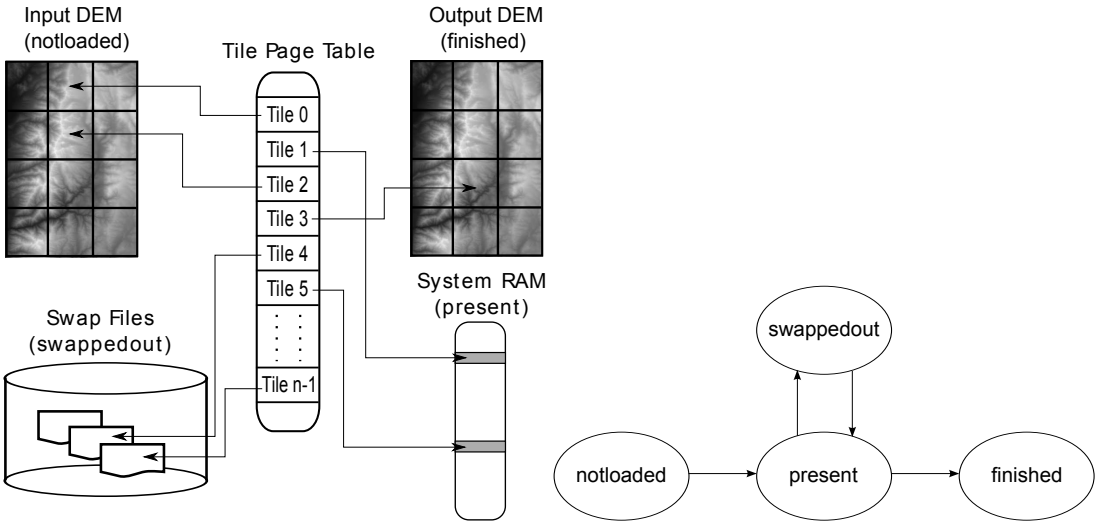


Figure 4: (a)Tile page table maps each page to a region in a DEM dataset, data block in a physical memory or data stored in a swap file on the hard disk (b) Tile state diagram

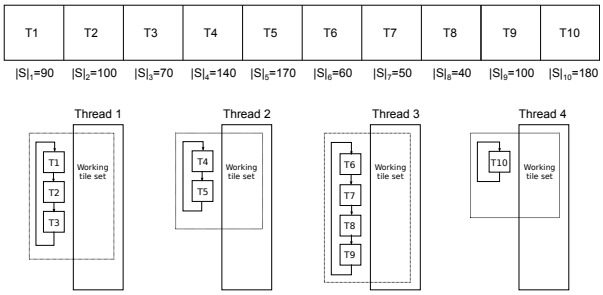
- $performLoadBalancing(tiles[], ctid)$: Given a set of tile ids $tiles[]$ and compute thread id $ctid$, *performLoadBalancing* returns the working set of tiles for the compute thread $ctid$. The virtual memory manager distributes the tiles evenly among the compute threads. Load balancing is a program option that can be disabled if desired.

- $finalizeMemoryManager()$: The memory resources used by the virtual memory manager are freed.

4.2. Load balancing

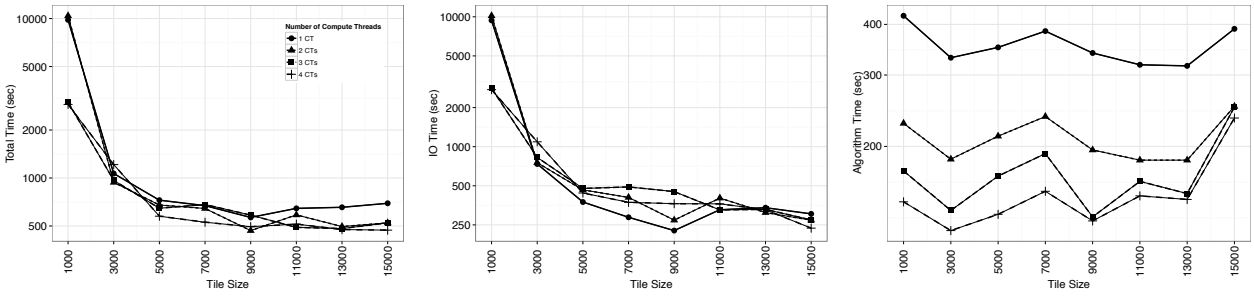
The variable nature of the topography that the algorithm operates on gives rise to variability in the number of cells on the unresolved stack for each tile at each iteration. Load balancing has been applied for concurrent MPI execution streams that exhibit irregular structure and dynamic load patterns [29] and to ensure that the load on each core is proportional to its computing power in multi-core architectures [30]. We developed a load balancing mechanism that is specifically tailored for the parallel modified PD algorithm. The modified PD algorithm applied to a tile iteratively processes unresolved cells whose indices are stored on a stack. The size of the stack on a tile indicates the work load per tile. The load balancing rule distributes the tiles based on tile stack size ($|S|$). Figure 5 illustrates the load balancing mechanism. Let T be the set of all tiles and CT be the set of compute threads. Then $\sum |S| = \sum_{i=1}^{|T|} |S|_i$ is the sum of stack sizes, and $\overline{|S|} = \sum |S|/|CT|$ is used as the target number of elements in the stacks per compute thread. The load balancing algorithm assigns consecutive tiles to each compute thread to take advantage of CPU caches through the principle of memory locality. The number of assigned cells might be greater than or equal to $\overline{|S|}$ as the tile granularity affects the distribution of the load.

Figure 5: Illustration of load balancing among the compute threads; In this example, $\sum |S| = 1000$, $\overline{|S|} = 250$ for 4 compute threads and 10 tiles



5. Experimental Results

Numerical experiments were performed to evaluate the virtual memory manager using five DEM datasets obtained from the National Elevation Dataset web site (`http://ned.usgs.gov`) (Table 1). These ranged from a relatively small test dataset to a dataset that exceeded the physical memory capacity of the test computer. Tile sizes, number of compute threads and memory capacity were all varied in the runs. The experiments were conducted on a machine equipped with an Intel Core I7 processor with 3.40 GHz and configured with either 4 GB or 16 GB of RAM. We used the 64-bit version of Microsoft Windows 7. Although not popular in HPC systems, Windows is widely used for desktop Geographic Information System terrain analysis



(a) (b) (c)

Figure 6: Time results in seconds for DEM3 as a function of tile size where y-axis is in logarithmic scale. (a) Total execution time (b) Waiting time (c) Algorithm time. (for varying number of compute threads up to 4 and prefetch count $\alpha = 1$)

and enabling the processing of large DEMs on these systems was a goal of this work.

Tile size is one of the most important parameters impacting the performance of the parallel PD algorithm. To examine the impact of tile size, the parallel PD algorithm was executed with shared virtual memory management enabled and a varying number of compute threads up to 4 using DEM3 with a range of tile sizes from 1000 to 15000 on a machine with 16 GB RAM. The maximum memory capacity for this experiment was set to half memory size of DEM3 dataset for these runs. Maximum memory is the size of the allocated memory for the entire page structure. In this experiment it was purposely set less than the total memory required by the algorithm, around 4 times the file size, or 16 GB to induce swapping and exercise the user level virtual memory management.

The total execution time is a combination of *Algorithm time* and I/O *Waiting time* (i.e., the elapsed time between the time a tile is requested and the completion of that request by the I/O thread). In Figure 6(a), the total execution time is seen to decrease sharply when more than 2 compute threads are utilized for a tile size of 1000. Parallelization is seen here to be effective in overlapping I/O operations with computation by the I/O thread. We observe that total execution time continues to decrease as the tile size increases up to 9000, in which case the minimum time is achieved with 468 seconds when 2 compute threads are used. After a tile size of 9000, the total execution time begins to increase due to a *thrashing* effect because the memory manager performs an excessive number of disk swapping operations. The waiting time accounts for the majority of this total execution time, indicating an increased I/O overhead cost associated with small tile sizes and correspondingly more distinct disk accesses (Figure 6(b)). The algorithm time is also dependent on tile size, reflecting the additional computation involved with edge exchanges for smaller tile sizes, although this effect is significantly smaller than the I/O effect quantified by the wait time (Figure 6(c)). These experiments show that the tile size can have a dramatic effect on performance of the program but this effect diminishes as the tile size increases, and then becomes more prominent due to better data locality. This experiment suggests limiting tile sizes to approximately 9000 for this system for the best performance using 3 compute threads. Multithreading also helps decrease the algorithm time. However, we observe that as the memory limit decreases, the performance benefit of multithreading diminishes because the algorithm becomes more I/O intensive. Because I/O overhead increases as tile sizes decrease, experiments for tile sizes less than 1000 were not performed. Conversely, note that when the tile size is 15000, the algorithm execution times with 3 and 4 compute threads approach the algorithm time with 2 compute threads because of the unfair load distribution among the threads. We also performed the experiment for tile size 24000 to investigate the effect of the biggest tile size on the virtual memory manager but the experiments failed due to insufficient memory.

Table 1: Properties of DEM datasets used in the experiments

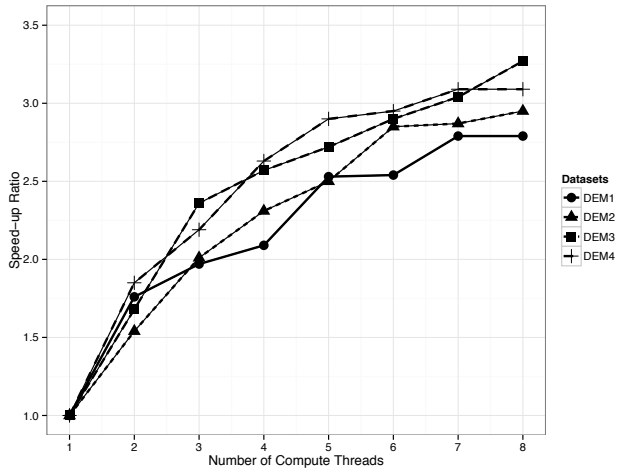
Label	Location	Domain Size	File Size
DEM1	GSL Basin (100 m cells)	4045x7402	117 MB
DEM2	GSL Basin (27.3 m cells)	14849x27174	1.61 GB
DEM3	Boise River Basin (10 m cells)	24856x41000	3.98 GB
DEM4	Wasatch Front (10 m cells)	34649x44611	6.05 GB
DEM5	Chesapeake Bay (10 m cells)	45056x49152	8.65 GB

Memory use during the run of the parallel PD algorithm to evaluate the effect of tile size with the DEM3 dataset was profiled to find the average and maximum memory used (Table 2) where the memory limit was set to 2000 MB. In practice, because memory is allocated in tile size increments for each thread, the maximum memory usage is larger than the target memory capacity parameter. The amount of this excess varies with tile size. When we set the tile size to 24000, the algorithm is failed to allocate memory space because the test machine doesn't have sufficient memory space and, therefore, we completely lost the benefit of the virtual memory system. The experi-

Table 2: Memory profile result of DEM3 in which expected maximum memory limit is 2000 MB

Tile Size (q)	Avg. Memory Usage	Max. Achieved Memory Usage
1000	1828 MB	2046 MB
3000	1851 MB	2396 MB
5000	1780 MB	3477 MB
7000	2089 MB	4955 MB
9000	2249 MB	6392 MB
11000	2438 MB	9253 MB
13000	3308 MB	12242 MB
15000	3456 MB	13683 MB
24000	×	×

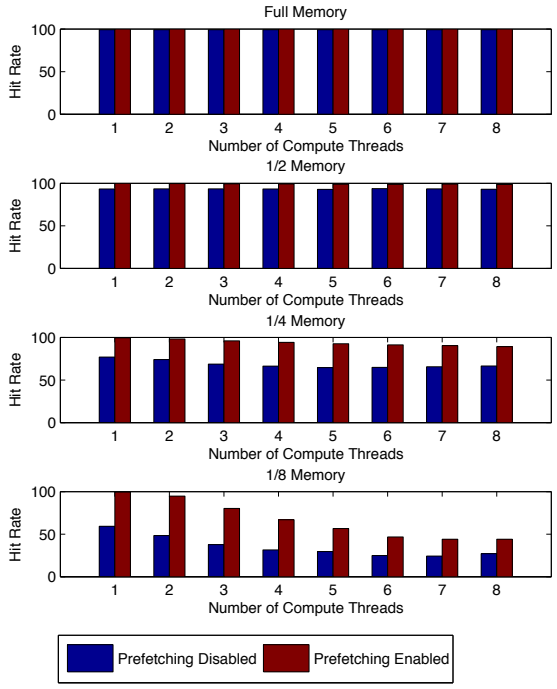
Figure 7: Speedup Ratio of the parallel PD algorithm for DEM1, DEM2, DEM3 and DEM4



ments show that the amount by which actual memory use exceeds target memory capacity increases with tile size. Maximum memory use exceeding the target parameter is a disadvantage of this virtual memory management approach as it limits the control on memory use by the program, resulting in potential for operating system virtual memory to kick in or for the program to crash due to insufficient memory.

In a second experiment we evaluated the parallel performance of the algorithm on DEM datasets that fit within RAM and where swapping was not needed. With the hardware reconfigured to 16 GB RAM we tested the speed on the first four DEM datasets. Although we achieved best time result with tile size 9000, tile size was set to 1000. Speed up ratios for this test (Figure 7) show a beneficial speed-up relative to the sequential modified PD algorithm with one I/O thread. A speed-up ratio of around 3 was obtained with 8 compute threads. Speed-up ratios are generally slightly more for the larger datasets, though this pattern is not consistent across the full range of the number of threads tested, the variability presumably being due to differences in the data. The machine used in the study has four cores, but the experiments were expanded to 8 compute threads to observe the effect of the hyper-threading feature of the processor, which improves processor performance by enabling the execution of pipeline-scaled interleaving of multiple threads on a single core, resulting in a modest performance benefit.

Figure 8: Hit rate results in percentage for virtual memory manager without prefetching (blue bars) and with prefetching (red bars) with respect to memory capacity and number of compute thread for DEM1



Next, the prefetching capability of the shared virtual memory manager module, described at the end of Section 3, was evaluated (Figure 8). The experiments were conducted with respect to the fraction of tiles that can be *present* in the available system memory. For example, 1/4 memory means there can be at most 1/4 of total number of tiles present in memory at the same time. The experiments show that as the size of the memory limit allowed to be used by the application decreases, the hit rate (i.e., the fraction of tiles already in memory when requested) also decreases due to an increasing number of swapping out operations. This may lead to higher loads on the I/O thread and disk. By contrast, because the implemented system uses one I/O thread, prefetching more than one tile ahead may also result in other threads experiencing higher I/O times. This I/O bottleneck is exacerbated in the presence of multiple compute threads and low memory limits.

Table 3 shows average ($\overline{HRD}$) and standard deviation ($\sigma_{\overline{HRD}}$) values of the difference of hit rates (HRD) between the VM with prefetching (P) and without prefetching (WP) in percentage with respect to memory capacity (M) and a number of compute threads (CT). The values

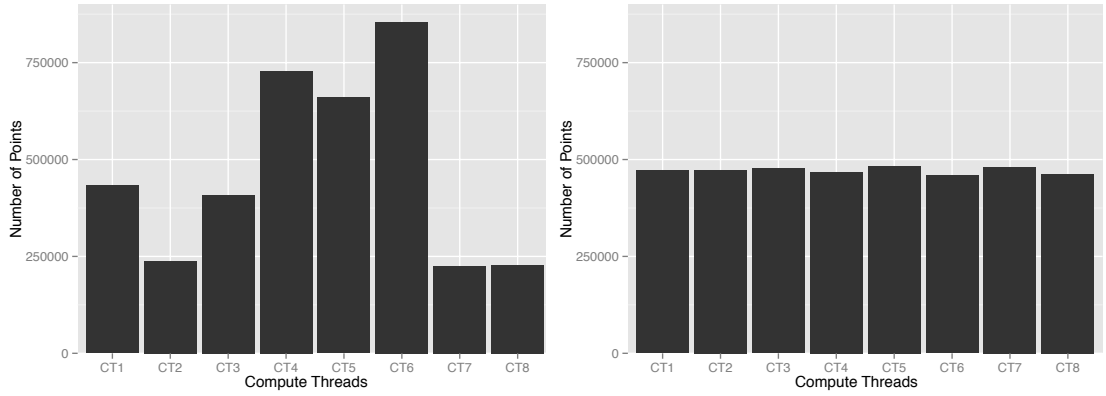


Figure 9: Compute thread loads with respect to a number of processed points (a) Without load balancing (b) With load balancing

Table 3: Average ($\overline{HRD}$) and standard deviation ($\sigma_{\overline{HRD}}$) of the difference of hit ratio between the VM with prefetching and without prefetching in percentage with respect to a number of compute threads for varying memory limits

Statistics	Full Mem.	1/2 Mem.	1/4 Mem.	1/8 Mem.
Average	0.784	5.834	22.664	31.279
Std. Dev.	0.015	0.450	9.355	11.301

are computed as follows:

$$HRD(CT, M) = (HR_{CT,M,P} - HR_{CT,M,WP}) \quad (1)$$

$$\overline{HRD}(M) = \frac{\sum_{i=1}^{CT} HRD_{i,M}}{CT} \quad (2)$$

$$\sigma_{\overline{HRD}}(M) = \sqrt{\frac{\sum_{i=1}^{CT} (HRD(i, M) - \overline{HRD}(M))^2}{CT - 1}} \quad (3)$$

As the virtual memory limit decreases, the average of difference of hit rate values increase that shows the benefit of using prefetching for low memory limit. However, the variation of hit ratios also increase. This is because increasing compute threads affects the hit ratio more under lower memory limits when compared to higher memory limits.

We evaluated the load balancing mechanism in which the tiles are distributed among the compute threads. Without a load balancing mechanism the number of points (grid cells) evaluated by each compute thread is quite unevenly distributed (Figure 9 a), resulting in the threads with fewer points to evaluate waiting for the threads with more points to evaluate before each synchronization that occurs with each call of the exchangeBorders function. When load balancing is not enabled, the compute thread 6 (CT6) processes nearly 6×10^5 more cells than the compute thread 7 (CT7). However, the load balancing approach resulted in a much more even distribution of the number of points processed by each thread (Figure 9 b) where the difference of processed cells between the compute threads dropped significantly. Fair data distribution can be enhanced by the use of smaller tile size that affects the granularity of the virtual memory.

Table 4: DEM datasets for which Pitremove was successfully run using TauDEM 5.1, the parallel PD algorithm with Operating System's built-in virtual memory system (WVM) and tiled virtual memory (TVM) approach

	DEM1 (0.1 GB)	DEM2 (2.3 GB)	DEM3 (4 GB)	DEM4 (6 GB)	DEM5 (8.7 GB)
TauDEM 5.1	✓	✓	×	×	×
PD Algorithm with WVM	✓	✓	✓	×	×
PD Algorithm with TVM	✓	✓	✓	✓	✓

The objectives of the study were to develop a capability to run the PD algorithm for large datasets without relying on the operating system's built-in virtual memory system. We performed an experiment using three approaches for the five DEMs listed in Table 1. We used a machine configured with 4 GB RAM. The last two DEM datasets require memory significantly more than this. DEM4 is 6.05 GB and requires nearly 21 GB of memory for the algorithm, while DEM5 is 8.65 GB and requires 34 GB of memory for the algorithm. We set the target memory capacity for our tiled virtual memory (TVM) approach to 2 GB and used a tile size of 1000 because of its better memory usage efficiency (see Table 2). We also performed a run with the tiled virtual memory disabled so that the Windows 7 op-

erating system's built-in virtual memory manager (WVM) was invoked. The default Windows 7 paging file size was used. The third comparison ran the same problem using TauDEM 5.1 [4] which uses an MPI message-passing approach to implement the PD parallel algorithm. It has no virtual memory management so also relies on the Windows 7 virtual memory manager for swapping where the total memory demanded by all processes is greater than the physical memory capacity. As illustrated in Table 4, we were able to successfully run the PD algorithm with the developed system for all the datasets, the largest of which demanded memory more than five times the physical memory available. By contrast, the TauDEM 5.1 package failed to process DEM3, DEM4 and DEM5, and the PD algorithm with WVM failed to process DEM4 and DEM5.

6. Conclusion and Discussion

The increased availability of high resolution DEMs is driving the need for software to process and analyze these on all systems including desktop PC systems with limited memory. In this paper we introduced a virtual tile memory manager that can be used with hydrologic terrain analysis programs to provide user level memory management and enable the processing of large DEMs on a PC, where operating system virtual memory management and multi-process domain partitioning fail. This was illustrated using a parallel pit removal algorithm for hydrologically conditioning DEM datasets. Elimination of pits is an essential step in hydrologic terrain analysis, required before applying other hydrological algorithms to calculate flow paths, slopes, and delineation of watersheds and subwatersheds. We used a parallel implementation of a modified Planchon and Darboux algorithm [10] for pit removal in shared-memory multithreaded systems, designed to run on desktop computers having limited system memory. system. The system is optimized to retrieve tiles with a high hit ratio. The main conclusions of the system developed are:

1. We are able to run this hydrologic terrain analysis algorithm on limited memory systems for datasets so large that previously available algorithms fail. This overcomes the main obstacle of the memory capacity of the machine, by utilizing a user-level shared virtual memory system.
2. While evaluated with pitremove, there is nothing in the user-level virtual memory system that should preclude it from use with other algorithms which may require modification of the page replacement algorithm based on algorithm-specific data access patterns to maximize system throughput.
3. Efficiencies were achieved using algorithm refinements such as load balancing and the pre-fetching approach used in the page replacement algorithm of the user-level shared virtual memory system that increased hit rate and reduced wait time.

We conclude from the experimental results that the implemented parallel application is beneficial to geoscientists and researchers for computing raster-based computations in very large DEM datasets on a single machine with limited memory. We believe that the implemented system also can be used for other flow algebra algorithms used in TauDEM [7] with slight modification. For other algorithms the pre-fetching approach will need to be modified according to the algorithm-specific data access patterns. Furthermore, it may be beneficial to explore adaptive page replacement algorithms that change the replacement algorithm at run time based on the tile-access pattern of the system.

Acknowledgments

This research was supported by the US Army Engineer Research and Development Center System-Wide Water Resources Program under contract number W912HZ-11-P-0338. This support is gratefully acknowledged.

References

[1] I. D. Moore, R. B. Grayson, A. R. Ladson, Digital terrain modelling: A review of hydrological, geomorphological, and biological applications, Hydrological Processes 5 (1) (1991) 3–30. doi:10.1002/hyp.3360050103.

[2] T. Hengl, H. I. Reuter, Geomorphometry: concepts, software, applications, Vol. 33, Elsevier, 2009.

[3] J. P. Wilson, J. C. Gallant (Eds.), Terrain Analysis: Principles and Applications, Wiley, New York, 2000.

[4] D. G. Tarboton, Terrain Analysis Using Digital Elevation Models (TauDEM), Utah Water Research Laboratory, Utah State University, http://hydrology.usu.edu/taudem/ (2014).

[5] T. K. Tesfa, D. G. Tarboton, D. W. Watson, K. A. Schreuders, M. E. Baker, R. M. Wallace, Extraction of hydrological proximity measures from dems using parallel processing, Environmental Modelling and Software 26 (12) (2011) 1696 – 1709. doi:10.1016/j.envsoft.2011.07.018.

[6] C. Wallis, R. Wallace, D. G. Tarboton, D. W. Watson, K. A. T. Schreuders, T. K. Tesfa, Hydrologic terrain processing using parallel computing, 18th World IMACS / MODSIM Congress, 2009, pp. 2540 – 2545, http://www.mssanz.org.au/modsim09/F13/wallis.pdf.

[7] D. G. Tarboton, K. A. T. Schreuders, D. W. Watson, M. E. Baker, Generalized terrain-based flow analysis of digital elevation models, in: Proceedings of the 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Cairns, Australia, 2009, pp. 2000–2006.

[8] D. Engler, S. Gupta, M. Kaashoek, Avm: application-level virtual memory, in: Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on, 1995, pp. 72–77. doi:10.1109/HOTOS.1995.513458.

[9] A. Marowka, Back to thin-core massively parallel processors, Computer 44 (12) (2011) 49 –54. doi:10.1109/MC.2011.133.

[10] O. Planchon, F. Darboux, A fast, simple and versatile algorithm to fill the depressions of digital elevation models, CATENA 46 (23) (2002) 159 – 176. doi:10.1016/S0341-8162(01)00164-3.

[11] GDAL Development Team, GDAL - Geospatial Data Abstraction Library, Open Source Geospatial Foundation, http://www.gdal.org (2014).

[12] S. K. Jenson, J. O. Dominque, Extracting topographic structure from digital elevation data for geographic information system analysis, Photogrammetric Engineering and Remote Sensing 54 (11) (1988) 1593–1600.

[13] D. G. Tarboton, R. L. Bras, I. Rodriguez-Iturbe, On the extraction of channel networks from digital elevation data, Hydrological Processes 5 (1) (1991) 81–100. doi:10.1002/hyp.3360050107.

[14] L. Wang, H. Liu, An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling, International Journal of Geographical Information Science 20 (2) (2006) 193–213. doi:10.1080/13658810500433453.

[15] R. Barnes, C. Lehman, D. Mulla, Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models, Comput. Geosci. 62 (2014) 117–127. doi:10.1016/j.cageo.2013.04.024.

[16] X. J.-W. LIU Yong-He, ZHANG Wan-Chang, Another fast and simple dem depression-filling algorithm based on priority queue structure, Atmospheric and Oceanic Science Letters 2 (4) (2009) 214.

[17] H.-T. Do, S. Limet, E. Melin, Parallel computing flow accumulation in large digital elevation models, Procedia Computer Science 4 (0) (2011) 2277 – 2286, proceedings of the International Conference on Computational Science, ICCS 2011. doi:10.1016/j.procs.2011.04.248.

[18] R. Xu, X. Huang, L. Luo, S. Li, A new grid-associated algorithm in the distributed hydrological model simulations, SCIENCE CHINA Technological Sciences 53 (2010) 235–241, 10.1007/s11431-009-0426-4.

[19] H. Wang, Y. Zhou, X. Fu, J. Gao, G. Wang, Maximum speedup ratio curve (msc) in parallel computing of the binary-tree-based drainage network, Computers and; Geosciences 38 (1) (2012) 127 – 135. doi:10.1016/j.cageo.2011.05.015.

[20] L. Ortega, A. Rueda, Parallel drainage network computation on cuda, Computers and Geosciences 36 (2) (2010) 171 – 178. doi:10.1016/j.cageo.2009.07.005.

[21] J. F. O'Callaghan, D. M. Mark, The extraction of drainage networks from digital elevation data, Computer Vision, Graphics, and Image Processing 28 (3) (1984) 323 – 344. doi:http://dx.doi.org/10.1016/S0734-189X(84)80011-0.

[22] S. K. Reinhardt, J. R. Larus, D. A. Wood, Tempest and typhoon: User-level shared memory, SIGARCH Comput. Archit. News 22 (2) (1994) 325–336. doi:10.1145/192007.192062.
URL http://doi.acm.org/10.1145/192007.192062

[23] A. Basu, J. Gandhi, J. Chang, M. D. Hill, M. M. Swift, Efficient virtual memory for big memory servers, SIGARCH Comput. Archit. News 41 (3) (2013) 237–248. doi:10.1145/2508148.2485943.
URL http://doi.acm.org/10.1145/2508148.2485943

[24] H. Huang, P. Pillai, K. G. Shin, Design and implementation of power-aware virtual memory, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '03, USENIX Association, Berkeley, CA, USA, 2003, pp. 5–5.
URL http://dl.acm.org/citation.cfm?id=1247340.1247345

[25] J. Pardy, K. Hartling, Efficient Data Management and Analysis with Geoprocessing, ESRI, http://proceedings.esri.com/library/userconf/devsummit13/papers/devsummit-086.pdf (2014).

[26] A. S. Tanenbaum, A. S. Woodhull, A. S. Tanenbaum, A. S. Tanenbaum, Operating systems: design and implementation, Vol. 2, Prentice-Hall Englewood Cliffs, NJ, 1987.

[27] K. Li, Ivy: A shared virtual memory system for parallel computing., ICPP (2) 88 (1988) 94.

[28] A. J. Smith, Sequential program prefetching in memory hierarchies, Computer 11 (12) (1978) 7–21.

[29] M. Bhandarkar, L. Kal, E. de Sturler, J. Hoeflinger, Adaptive load balancing for mpi programs, in: V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, C. Tan (Eds.), Computational Science - ICCS 2001, Vol. 2074 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 108–117. doi:10.1007/3-540-45718-6_13.
URL http://dx.doi.org/10.1007/3-540-45718-6_13

[30] T. Li, D. Baumberger, D. A. Koufaty, S. Hahn, Efficient operating system scheduling for performance-asymmetric multi-core architectures, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07, ACM, New York, NY, USA, 2007, pp. 53:1–53:11. doi:10.1145/1362622.1362694.