Parallelization Strategies to Speed-Up Computations for Terrain Analysis on Multi-Core Processors

Steffen Schiele[1], Holger Blaar[1], Detlef Thürkow[2], Markus Möller[2], Matthias Müller-Hanneman[1]

[1]University of Halle-Wittenberg
Institute of Computer Science
Von-Seckendorff-Platz 1
06120 Halle (Saale)

[2]University of Halle-Wittenberg
Institute of Geosciences
Von-Seckendorff-Platz 4
06120 Halle (Saale)

steffen.schiele@informatik.uni-halle.de

Abstract: Efficient computation of regional land-surface parameters for large-scale digital elevation models becomes more and more important, in particular for web-based applications. This paper studies the possibilities of decreasing computing time for such tasks by parallel processing using multi-threads on multi-core processors. As an example of calculations of regional land-surface parameters we investigate the computation of flow directions and propose a modified D8 algorithm using an extended neighborhood. In this paper, we discuss two parallelization strategies, one based on a spatial decomposition, the other based on a two-phase approach. Three datasets of high resolution digital elevation models with different geomorphological types of landscapes are used in our evaluation. While local surface parameters allow for an almost ideal speed-up, the situation is different for the calculation of non-local parameters due to data dependencies. Nevertheless, still a significant decrease of computation time has been achieved. A task pool-based strategy turns out to be more efficient for calculations on datasets with many data dependencies.

1 Introduction

A large variety of methods for space-oriented analyses of the earth's surface have been developed in the past couple of years. Methods based on photogrammetry and laserscanning are able to produce digital elevation models (DEMs) with a geometric resolution within centimeters and a high quality, as well as a high quantity of data. Algorithms have to be developed enabling computational efficient processing of large datasets in reasonable time [Woo09]. Therefore, new strategies for efficient implementation and parallelization of such computations are needed.

The attribute 'flow direction' is the basis for the calculation of the most popular hydrological parameters like 'specific catchment area' or 'topographic wetness index' [ZKLY07, GP09]. The typical workflow for the computation for these parameters is sketched in Figure 1. First, the raw data of the DEM is preprocessed to remove artefacts, systematic errors, and to reduce noise. It is also important to eliminate spurious sinks by filling [RHGS09]. Afterwards, the flow directions are determined. The computation of catchment area and

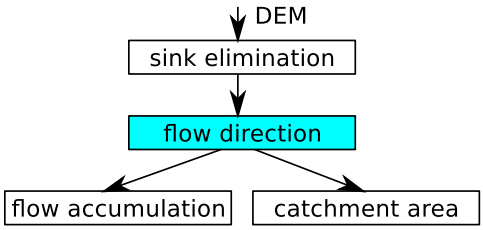


Figure 1: Sketch of the typical workflow for hydrological parameter calculations.

flow accumulation base on the flow direction. Computationally, these steps are less intensive.

Although the sequential running time for test sites with 10^8 grid cells is in the range of a few minutes on standard desktop machines, further reductions of processing times are still of crucial importance for the acceptance of web-based hydrological applications [ASMB03, GTDS10]. Our intention is to investigate how parallel processing using multi-threads can help to decrease computing time of flow direction's calculation. For this purpose, we have developed two parallelization strategies and tested them on a modified single-flow algorithm (Section 2.1). In this paper, we restrict our discussion of parallelization strategies to this single-flow algorithm, although we have implemented parallel algorithms for the remaining steps of the workflow, too [Sch10].

A classical and most basic algorithm to determine flow directions is the so-called D8-algorithm [OM84]. From each grid cell, all flow is passed to the neighbor in direction of the steepest descent. Except for cells at the boundary of the DEM, a cell can be viewed as the center cell of a 3×3 subgrid. The 8 non-central cells of such a subgrid are considered as neighbors, hence the name D8. The crucial point in this method is how ambiguous flow directions are resolved, when the same minimum down-slope gradient is found for several neighbor cells. We would like to emphasize that such ambiguities are not an academic consideration, they occur quite often in practice. To resolve such ambiguities, we have developed an extended neighborhood approach. Extended neighborhoods pose a particular challenge for parallelization due to non-local memory access patterns.

Related work Based on the model of SIMD (single instruction stream, multiple data stream) computers, Mower [Mow94] discusses data-parallel procedures for drainage basin analysis. More recent work on multi-core machines uses the OpenMP library or MPI. For example, Neal et al. [NFT09] describe and report experience with parallelization of procedures for flood inundation models using the OpenMP interface. Building on the message passing interface MPI, Tesfa et al. [TTW+11] developed parallel approaches for the extraction of hydrological proximity measures. In contrast to the single flow direction method studied in this paper, they use the multiple flow direction model D-infinity. Instead of using multi-core processors, another interesting approach for parallelization is the usage of GPUs. Ortega and Rueda [OR10] have studied the applicability of this approach for parallel computation of drainage networks using the CUDA framework.

To cope with large-scale high-resolution datasets, Mølhave et al. [MAAR10] developed I/O-efficient external memory algorithms. The focus of our paper, however, is on algorithms which can be handled within internal memory.

Overview In Section 2, we first sketch our extended D8 algorithm for flow computations and then explain our two parallelization strategies. We also describe the sites used in our experimental study. Computational results for both parallelization strategies are given in Section 3. Finally, we summarize and discuss our observations in Section 4.

2 Methods

2.1 Extended D8 algorithm for flow computations

To avoid ambiguous flow directions, we introduce a modified algorithm D8*e* which recursively extends the neighborhood in such cases until a unique single flow direction is found. For a given cell c and its neighborhood $N(c)$ let $S(c) \subseteq N(c)$ be the subset of neighbors which realize the steepest descent. Thus, $S(c)$ forms the candidate set for the flow direction of c. If $|S(c)| = 1$, the flow direction is unique and we are done. Otherwise, we determine the extended neighborhood $EN(S(c))$ as the set of all cells which are connected to some cell $\bar{c} \in S(c)$ by a path of cells with the same altitude as $\bar{c}$. Then we compute recursively the flow direction of all cells within $EN(S(c))$. Among all considered cells, we take again the steepest descent. If this value is unique, we can now assign the flow direction of cell c as the one which leads along a path of assigned flow directions to the cell of steepest descent. Otherwise, the procedure has to be continued in the same manner until the ambiguities are resolved or no further neighborhood extension is possible. In the latter case, an arbitrary decision for cell c is made.

The neighborhood extension is the most time-consuming part of the computation of flow directions. The average size of the extended neighborhood varies widely depending on the terrain. An example of the effect of ambiguous flow directions is given in Figure 2. The figure shows two catchment areas, one computed with the D8 and one with the D8*e* algorithm. The first one misses a significant part of the catchment area.

2.2 Parallelization strategies

We investigate two main strategies for parallelization. The first approach divides the DEM into squares and the second one divides the computation of the flow directions into two phases. All threads access the same DEM stored in shared memory. Therefore, there is no need to transfer data but the data access has to be synchronized among the threads.

Dividing the DEM into squares If the grid domain of the DEM is partitioned into a number of disjoint squares and the flow directions are computed concurrently by different threads complications often arise. Namely, the extended neighborhood computation can include cells of other squares. This means that the same flow direction of a single cell might be calculated several times and the number of such cells could be prohibitively high.

To avoid such problems a pre-computation is executed. At first, the algorithm chooses squares which are slightly overlapping at common boundary cells. Then the flow direction of all boundary cells and their neighbors is computed in a sequential step. In doing so, no thread will cross its square boundary in the upcoming parallelized computations. Figure 3 exemplarily illustrates two cases where the algorithm extends the neighborhood of the cell with altitude 7 and precalculates the flow directions of all cells of the extended neighborhood (cells with altitude 5). Finally, the results of the sequential pre-computation are accessible by each thread. Afterwards, each thread computes the flow direction of the remaining cells.

Dividing the computation into two phases The main idea of this approach is to compute the flow direction using the original D8 algorithm during a first phase. Instead of extending the neighborhood of cells with ambiguous flow directions (see Section 2.1) such cells are only marked. During the second phase, our algorithm extends the neighborhood of each marked cell and computes their flow directions.

The first phase can be parallelized by dividing the DEM into squares. Only using the 3×3-neighborhood of a cell for computing the flow direction, no pre-computation is needed. In the second phase, the marked cells are assigned to different groups in such a way that redundant calculations are avoided. At first, a marked cell is assigned to an empty group. Afterwards, a cell will be assigned to this group, if

- the marked cell is adjacent to one cell of the group, or
- there is at most one unmarked cell between the marked cell and a cell of the group.

The group assignment is done with breadth-first search.

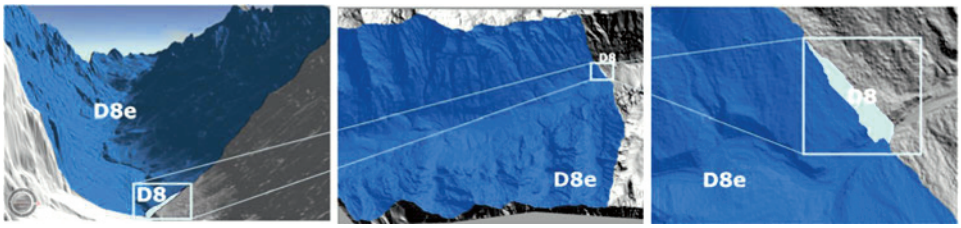
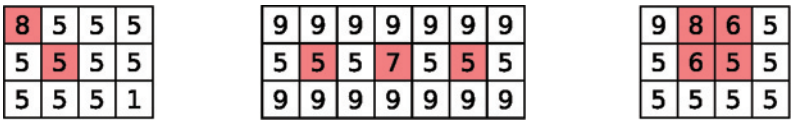


Figure 2: Comparison of catchment areas based on the flow directions computed using the D8 and D8*e* algorithm. This example shows the dramatic difference between the traditional D8 algorithm (which leads to a way too small catchment area) and our new D8*e* algorithm resulting in a catchment area confirmed by experts.

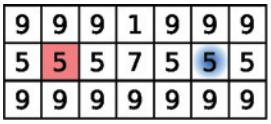
(a) (b)

Figure 3: Excerpts of a DEM where two squares overlap in the gray-shaded boundary cells. Numbers in each cell of the DEMs correspond to the elevation. In both examples, the D8*e* algorithm extends the neighborhood of the cell with altitude 7 and precalculates the flow directions of all cells with altitude 5.

Figure 4(a) shows marked cells for which the computation requires a neighborhood extension. In each of the three examples the algorithm puts the marked cells into the same group. The differently marked cells in the example of Figure 4(b) belong to two different groups. The extended neighborhoods of both cells are disjoint. One thread is responsible for assigning cells to groups and manages these groups by a task pool [RR10]. Pseudocode is given in Algorithm 1. Each of the other threads takes a group out of the task pool and computes the flow direction of the marked cells of this group. In doing so, this approach ensures that two threads will never compute the flow direction of the same cell and that the computed extended neighborhoods will never overlap.



(a) In each example, cells with ambiguous flow direction are marked. Because of the neighborhood extension, the marked cells will be computed by the same thread. Therefore, our algorithm puts them into the same group.



(b) In this example, differently marked cells are far enough from each other so that their extended neighborhoods are disjoint. Thus, our algorithm puts them into different groups.

Figure 4: Parallelization of the D8*e* algorithm by dividing the computation into two phases. Numbers in each cell correspond to the elevation in the DEM.

2.3 Efficiency measurement

For measuring the runtime of a parallel implementation, the *real time* (wall clock time) can be used, but it can be influenced by other applications running on the system. *User* and *system CPU time* of a parallel application is the *accumulated* user and system CPU time on all processors. Because of the disruptive effect of other processes running on the system the number of cores used by one job cannot easily be determined and can also vary during program execution. We used an almost unloaded system for real-time measurements.

Algorithm 1 The procedure determines groups of cells with ambiguous flow direction (see Figure 4(a)) and puts each group of such cells into the task pool. If all cells with ambiguous flow direction are grouped a signal is send to all threads.

```
1:  procedure DETERMINE GROUPS OF CELLS
2:      for each cell ∈ DEM ∧ cell.flowdirection == 0 do
3:          create newGroup
4:          newGroup ← {cell}
5:          for each nCell in 4 × 4 neighborhood of a cell ∈ newGroup do
6:              if nCell.flowdirection == 0 then
7:                  newGroup ← newGroup ∪ {nCell}
8:              end if
9:          end for
10:         put newGroup into the taskpool
11:     end for
12:     send signal termination to all threads
13: end procedure
```

2.4 Study sites and datasets

The computational experiments are executed on three different DEMs with high spatial resolutions (Table 1). All DEMs are based on airborne laser-scanning which are cleaned from vegetation and artificial objects like buildings. The datasets represent three geomorphological types of landscapes in Central Europe: high mountains (the Alps - Reintal; see Figure 2), low mountain ranges (the Ore Mountains - Saidenbachtal) and floodplains of the lowlands (Floodplain of the River Mulde). In all DEMs sinks were removed by filling within a preprocessing step (see [RHGS09]). Table 1 shows the meta data of the used DEMs.

Table 1: Meta data parameters of the DEM datasets

DEM dataset	**Cell number**	**Columns and rows**	**Spatial resolution** $[m^2]$	**Filesize**[a] [Mb]
Reintal	37,734,557	10717×3521	1×1	290
Saidenbachtal	35,000,000	7000×5000	2×2	240
Mulde	116,674,076	6661×17516	1×1	880

[a] ascii-grid format (*.asc)

3 Results

The following runtime measurements were done on a symmetric multiprocessing computer (two Intel(R) Xeon(R) CPUs with four cores and 2.93 GHz each, and with 47 GB main memory). Additional runtime measurements were executed on different hardware, like a Linux Server with four AMD Opteron™ processors 852 (1000 MHz) and 16 GB main memory, or a symmetric multiprocessing cluster (18 computation nodes with 16 CPU cores each and with at least 32 GB main memory each). In all cases the computations require up to 2 GB main memory. Computations on the symmetric multiprocessing cluster were executed only on one node. On the available architectures speed-up and efficiency did not show significant differences. The algorithms are implemented in C++ and the Pthread library is used for parallelization, using compiler `g++` in version 4.4.3.

3.1 Dividing the DEM into squares

The sequential runtimes for the three DEMs are 3.4s for DEM 1 (Reintal), 6.7s for DEM 2 (Saidenbachtal) and 122s for DEM 3 (Mulde). The obtained speed-ups for up to four threads are shown in Table 2. The number of threads is displayed in the form of „$a \cdot b$“ where a is the number of rowwise partitions, and b the number of columnwise partitions. In Figure 5 the runtime (real time) is compared to the ideal runtime. The stacked bars show

Table 2: Speed-ups of the parallel algorithm which divides the DEM into squares.

number of threads:	$\mathbf{1 \cdot 1}$	$\mathbf{1 \cdot 2}$	$\mathbf{2 \cdot 1}$	$\mathbf{1 \cdot 3}$	$\mathbf{3 \cdot 1}$	$\mathbf{2 \cdot 2}$
speed-up (DEM 1 “Reintal”):	0.99	1.80	1.78	2.33	2.23	3.23
speed-up (DEM 2 “Saidenbachtal”):	0.97	1.59	1.63	2.05	2.02	2.40
speed-up (DEM 3 “Mulde”):	0.98	1.53	1.55	1.46	1.81	1.82

the sequential computing part and the cumulative runtime of threads. Added together they are a measure for the cost of computation. The cost of the computation is also directly obtained by measuring the CPU time used. Both clarify that rising the number of threads tends to result in an increase of used CPU time. But the effects differ between different runs with the same number of threads. The more threads we use, the more likely it is that the pre-computation steps are computationally more intensive. Note that by chance the boundary cells of the squares may have (almost) with unambiguous flow directions. If so, the pre-computation step is computationally less intensive as we can see in the case 2×2 in Figure 5. The load balance depends in particular on the topology and on the partition of the DEM. In most but not all cases we have a suboptimal load balance.

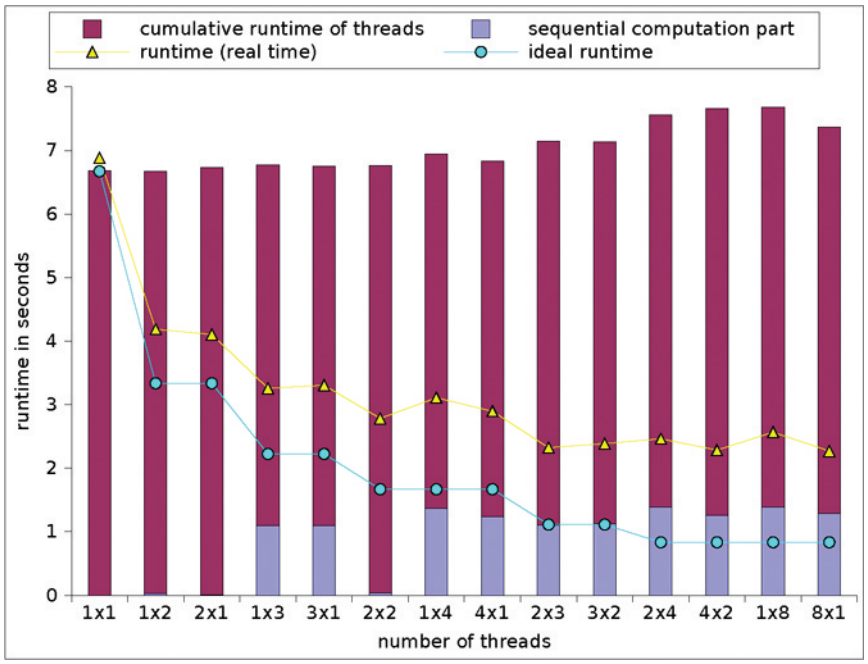


Figure 5: Runtime diagrams of the parallel algorithm which divides the DEM into squares. The number of threads on the x–axis is displayed in the form of „$a \cdot b$“ where a is the number of rowwise partitions, and b the number of columnwise partitions.

3.2 Dividing the computation into two phases

In Table 3, the speed-ups for up to four threads are presented. The runtime (real time) compared to the ideal runtime is shown in Figure 6. The stacked bars show the cumulative calculation time of threads of both phases and the runtime which is needed to group the cells. The calculation time is the period in which the threads perform computations without synchronization and communication. The maximum waiting time becomes significant when using more than two threads (“Reintal” and “Mulde”) or more than three threads (“Saidenbachtal”). For instance, using four threads for computing the flow directions of “Saidenbachtal”, one of the four threads had to wait up to 63% of the whole parallel runtime. In case of less than four threads, a thread has to wait for a task up to one percent of the whole runtime. Regarding the dataset “Mulde” a thread has to wait up to 23% (four threads) or 28% (eight threads), respectively, of the whole parallel runtime. The average waiting time of a thread is 14% (using four threads) or 27% (using eight threads), respectively, of its calculation time.

Working on the data set “Saidenbachtal” and “Reintal”, further investigations show that all threads have nearly equal computation time. The effort of calculation is well balanced, but in some periods there were no tasks for the threads. Thus, one or more threads had to wait. The maximum difference between computation time and the cumulated time of all

threads were determined, too. The more threads were used the more threads had to wait to get new tasks, but the threads had all nearly the same computation time.

Using the data set “Mulde”, we get a maximum difference between the calculation time of each thread of 10% (two threads are used) up to 18% (eight threads are used).

Table 3: Speed-ups of the parallel algorithm which divides the computation into two phases.

number of threads:	$\mathbf{1 \cdot 1}$	$\mathbf{1 \cdot 2}$	$\mathbf{2 \cdot 1}$	$\mathbf{1 \cdot 3}$	$\mathbf{3 \cdot 1}$	$\mathbf{2 \cdot 2}$
speed-up (DEM 1 “Reintal”):	0.90	1.67	1.73	2.23	2.26	2.47
speed-up (DEM 2 “Saidenbachtal”):	0.94	1.82	1.84	2.38	2.40	2.58
speed-up (DEM 3 “Mulde”):	0.85	1.76	1.77	2.31	2.29	2.08

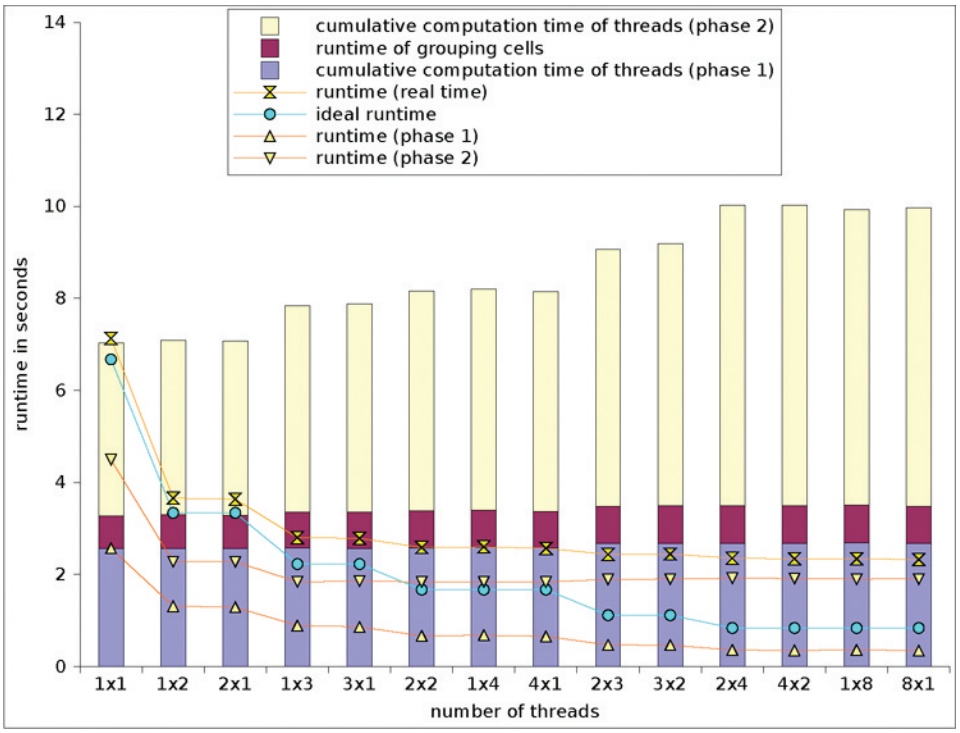


Figure 6: Runtime diagrams of the parallel algorithm which divides the computation into two phases. The number of threads on the x–axis is displayed in the form of „$a \cdot b$“ where a is the number of rowwise partitions, and b the number of columnwise partitions. Theses partitions only apply to the first phase.

We have implemented several modifications of the algorithm to improve the runtime. One modification eliminates recursion. Another one implements a heuristic which tries to change the processing order in such a way that large groups of cells are handled with priority. To avoid an overwhelming effort for the data access, a modification – which

merges small groups of cells together and puts those into the task pool – has also been implemented. However, all these modifications did not show significant runtime differences.

As shown in Figure 6, for all test sites the runtime decreases but the accumulated computation time increases with the number of threads. There is no explicit synchronization during the calculations because of the disjoint groups of cells. An increased number of threads causes an increase of the overhead of thread administration and an increase of random access which can induce for instance false sharing (see [HS08]). The cumulated waiting time also increases.

4 Discussion and conclusion

In this study, we investigated the efficiency of parallelization techniques on the example of the non-local "extended neighborhood" of raster cells. An extended neighborhood is used within the D8*e* algorithm to make the flow direction unique when ambiguous flow directions occur. The ordinary D8 neighborhood has ambiguous flow directions. Two parallelization approaches were tested regarding their efficiency (Section 2.2):

1. The advantage of parallelization by dividing the DEM into squares is the low cost of synchronization. A disadvantage is the inefficient load balancing. On the one hand, if we have a well-adjusted load balance the speed-ups would be improved. On the other hand, speed-ups near to the best possible speed-up in all cases could not be achieved because of the sequential part (pre-computation step). Synchronization costs are insignificant in our implementations.

2. The advantage of parallelization by dividing the computation into two phases is the well-adjusted load balance. Disadvantages are the increasing cost of synchronization and data access. The sequentially grouping of cells causes increased computation expenditure and is independent from the number of threads. This parallelization alternative is more independent of the composition of the DEM because of the dynamic distribution of the cost-intensive calculations to the threads. Possible reasons for the non-ideal speed-ups have been examined. Unbalanced distribution of calculations, synchronization time and conflicts between threads because of a shared data structure can be excluded as being mainly responsible for these observations. Possible explanations are data access and the effect of false sharing caused by the high number of write and read accesses.

Neither parallelization strategy enabled speed-ups that are equal to the number of threads. This is in contrast to the parallelized calculation of local surface parameters like slope and aspect where speed-ups near to the number of threads have been achieved for all three study areas. The speed-up comparisons also revealed landscape-related dependencies. While the speed-ups for the high mountain dataset computation are higher by running the first strategy (DEM "Reintal"), the speed-ups of the second strategy have proved to be more efficient for the calculation on the low mountain and the floodplain datasets (DEM 2

"Saidenbachtal" and DEM 3 "Mulde"). The second case shows the effect of flat areas (e.g. dams, filled sinks and floodplains) on the parallelization efficiency where a fixed DEM division into threads was carried out. This implies that the distribution of the extended neighborhoods is fixed, too. This could lead to a sub-optimal load balancing. The second parallelization strategy is more appropriate to such DEMs because of the dynamic consideration of the extended neighborhood. As mentioned in the abstract, this task pool-based strategy is more efficient for calculations on datasets with many data dependencies. However, the second parallelization strategy is more computationally intensive. Thus, in the case of almost optimal load balances (e.g. dataset "Reintal") the speed-ups of the second parallelization strategy are lower than the speed-ups of the first one.

In all datasets there are many small and few huge extended neighborhoods. As a consequence, one thread could work on just one huge extended neighborhood while the other threads have already finished.

Further runtime measurements based on different spatial resolutions (area of DEM 1 in $2 \times 2\,\mathrm{m}^2$ and $5 \times 5\,\mathrm{m}^2$ resolution, area of DEM 2 in $5 \times 5\,\mathrm{m}^2$ resolution, and area of DEM 3 in 2×2 resolution) were executed, too. Because of the decreased runtime of the calculations caused by the lower number of cells, the overhead of administration of the threads becomes more significant. The speed-ups were a bit lower than the speed-ups presented above. Runtime measurements based on the original DEMs (the datasets previous to the sink filling) were also executed. Because of significantly fewer flat areas, the speed-ups were significantly higher than the presented speed-ups.

In future work we will try to improve the computation by parallelizing the extended neighborhood computation. But this seems to be challenging because of data dependencies. We will also work on much larger datasets (about 10^9 cells). Because of runtimes greater than some minutes we will focus on applications besides web-based implementations, too. In anticipation of a growing number of cores per processor, it will be worth studying other parallelization strategies for shared memory machines.

References

[ASMB03] W. Al-Sabhan, M. Mulligan, and G.A. Blackburn. A real-time hydrological model for flood prediction using GIS and the WWW. *Computers, Environment and Urban Systems*, 27(1):9–32, 2003.

[GP09] S. Gruber and S. Peckham. Land-surface parameters and objects in hydrology. In T. Hengl and H.I. Reuter, editors, *Geomorphometry - Concepts, Software, Applications*, volume 33 of *Developments in Soil Science*, pages 171–194. Elsevier, Amsterdam, The Netherlands, 2009.

[GTDS10] C. Gläßer, D. Thürkow, Ch. Dette, and S. Scheuer. The development of an integrated technical-methodical approach to visualise hydrological processes in an exemplary post-mining area in Central Germany. *ISPRS Journal of Photogrammetry and Remote Sensing*, 65(3):275–281, 2010. Theme issue "Visualization and exploration of geospatial data".

[HS08] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Elsevier, Amsterdam, The Netherlands, 2008.

[MAAR10] T. Mølhave, P. K. Agarwal, L. Arge, and M. Revsbæk. Scalable algorithms for large high-resolution terrain data. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application*, COM.Geo '10, pages 20:1–20:7. ACM, 2010.

[Mow94] J. E. Mower. Data-parallel procedures for drainage basin analysis. *Computers & Geosciences*, 20:1365–1378, November 1994.

[NFT09] J.C. Neal, T. Fewtrell, and M. Trigg. Parallelisation of storage cell flood models using OpenMP. *Environmental Modelling & Software*, 24:872–877, 2009.

[OM84] J.F. O'Callaghan and D.M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing*, 28(3):323–344, 1984.

[OR10] L. Ortega and A. Rueda. Parallel drainage network computation on CUDA. *Computer & Geosciences*, 36:171–178, 2010.

[RHGS09] H.I. Reuter, P. Hengl, P. Gessler, and P. Soille. Preparation of DEMs for Geomorphometric Analysis. In T. Hengl and H. I. Reuter, editors, *Geomorphometry - Concepts, Software, Applications*, volume 33 of *Developments in Soil Science*, pages 87–120. Elsevier, Amsterdam, The Netherlands, 2009.

[RR10] T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, Berlin, Heidelberg, 2010.

[Sch10] S. Schiele. Effiziente parallele Simulation von Niederschlagsabflüssen in digitalen Geländemodellen. Master's thesis, Martin-Luther-Universität Halle-Wittenberg, 2010.

[TTW+11] T.K. Tesfa, D.G. Tarboton, D.W. Watson, K.A.T. Schreuders, M.E. Baker, and R.M. Wallace. Extraction of hydrological proximity measures from DEMs using parallel processing. *Environmental Modelling & Software*, 26(12):1696–1709, 2011.

[Woo09] J. Wood. Overview of Software Packages Used in Geomorphometry. In T. Hengl and H. I. Reuter, editors, *Geomorphometry - Concepts, Software, Applications*, volume 33 of *Developments in Soil Science*, pages 257–267. Elsevier, Amsterdam, The Netherlands, 2009.

[ZKLY07] L. Zhang, Z. Kang, J. Li, and L. Yang. Comparison of the performance of flow-routing algorithms used in GIS-based hydrologic analysis. *Hydrological processes*, 21:1026–1044, 2007.