GeoInformatica (2024) 28:145–175
https://doi.org/10.1007/s10707-023-00500-w

Fast slope algorithm with the use of vectorization and parallelization for multicore architectures

Beata Bylina, Jarosław Bylina, Łukasz Chabudziński, Karol Karpowicz, Michał Klisowski, Piotr Oleszczuk, et al. ***[full author details at the end of the article]***

Received: 28 April 2021 / Revised: 31 July 2022 / Accepted: 17 May 2023 /
Published online: 7 July 2023
© The Author(s) 2023

Abstract
The slope calculation algorithm is one of the most widely used geospatial algorithms employing the 3x3 moving window technique (along with calculation of aspect, curvature and flow direction). This work presents an approach consisting of transforming a slope algorithm from a sequential form into a version that can exploit vector and parallel traits of multicore architectures with vector instructions. This approach allows us to take advantage of the potential of the modern multicore processors. The basic idea for optimizing the 3x3 moving window computation is to split the equation used to calculate the result into parts that operate on data that are known to exist in adjacent memory locations. The research was conducted on two multicore architectures without the change in the code — the older architecture was Sandy Bridge and the newer one was Haswell (with more cores). The efficiency of the developed slope algorithm was verified in practice with the use of DEM files of the same resolution but of different sizes. We showed through the numerical experiments that our approach gives better time performance than the original algorithm (and other tools) — and with no loss of accuracy.

Keywords HPC · Scalability · Parallel computing · OpenMP · DEM · Slope

1 Introduction

The evolution of technologies (sensor systems, automated geocoding, social media platforms) entails the need to store more and more spatial data [1–3], which in turn necessitates the development of more efficient algorithms to process it. Additional factors forcing the development of more efficient algorithms are: the need of fast spatial search, simulation (e.g. real-time simulation of landslides where the slope is constantly changing), modeling the changes of spatial data over time [1, 2, 4].

✉ Michał Klisowski
michal.klisowski@umcs.lublin.pl

Extended author information available on the last page of the article

Most geospatial analyses are very time-consuming, especially for a large scale datasets, and does not scale well [5]. That may become an issue in a practical implementation, especially when a quick response is needed, e.g. in web applications such as terrain visualization [6, 7]. Most algorithms are not designed for multicore architectures with vector instructions.

The challenge is to adopt simple algorithms capable of adapting to the dynamic nature of the problems while still keeping good properties of memory locality and computational load balance [8, 9]. Some of such algorithms are the ones extracting topographic parameters from DEM. These parameters are used in spatial analyses in order to establish relations between components of the natural environment. They are important because they allow a quantitative description of changing terrain properties — what poses a valuable note in analyses of hydrologic, geomorphological and ecological processes [10]. The slope is one of the basic primary parameters [8, 10–12] and it is used e.g. for computing flow velocity for both overland [10, 11] and channelized flow [13]. On the basis of the slope, some new parameters (like soil erosion and deposition [14], soil moisture content [7], flow velocity [15]) are calculated. The slope plays a fundamental role in more advanced models, although it is only one of the input elements for advanced computational algorithms (like modeling rates of snowmelt [16] and evapotranspiration [17]). Therefore, the high performance of the slope algorithm in such complex geospatial analyses is the key to speed them up.

The main computing advantages of contemporary general purpose processors (CPUs) are multicore and vector processing capabilities. To utilize the full computing potential of CPUs, both should be used. This is critical to the performance and, to a limited extent, the compiler can do the task. However, due to complexity of the algorithm or spurious data dependencies, it cannot always be done automatically by optimizing compilers. It is necessary to modify the existing algorithm and manual transformations are used for this purpose — it is a code tuning technique that is widely used [18–20]. The aim of the paper is to show how to perform such transformations manually. In the presentation of our method, as an example, we concentrate on accelerating the slope algorithm. In our case, the loop restructuring is associated with: improved cache utilization, effective vectorization and the use of parallel processing.

This article focuses on accelerating the slope algorithm through vector optimization for each core and parallel optimization for multicore processors. The proposed solutions have been evaluated for different sizes of DEM in terms of: execution time, scalability and programming effort needed for parallelization of the program. The main contributions of this article are following:

- The slope algorithm was modified by transforming the loop, making it easy to vectorize and parallelize.
- The parallelized and vectorized implementations of the algorithm were developed, using appropriate data storage in memory to effectively use cache for shared memory multicore computers.
- The vector-parallel implementations have been tested and evaluated on multicore processors of two different hardware configurations. All the tests were carried out on data sets of three different sizes.
- An attempt in time comparison between our best implementation of the algorithm and existing solutions (both commercial and non-commercial) was made.

Solutions proposed in the paper have been evaluated for different sizes of DEMs in terms of: execution time, scalability and programming effort needed for parallelization of the program.

The techniques used here for parallelization and vectorization can also be applied to other algorithms for neighborhood operations such as aspect, curvature or focal flow.

This article is organized as follows. In Section 2, we discuss: the basic and modified version of slope algorithms, the vectorization and parallelization mechanisms used in our modifications, as well as selected details of our implementations. Section 3 is dedicated to the description of the data used to test implemented algorithms. Section 4 focuses on the details of the testing procedure and on the analysis of the obtained results. We also verify the correctness of the results and compare our implementation with existing one. In Section 5, we present the conclusions of the experiments and mention possible directions for further research.

2 Algorithms and their implementations

2.1 Sequential algorithm

Using data stored in the form of a grid of cells — such as DEM — we can compute the slope for each of them (except for the edge ones), without additional input and knowledge about the area being analyzed. The slope determines the maximum rate of elevation changes from the cell to its neighbors. A high value of the slope between the cell and its neighbors indicates a steep descent from the cell. The lower the slope, the flatter the terrain; the higher the slope, the steeper the terrain.

In research and analysis of the terrain surface, which is carried out using GIS, the slope is usually expressed in degrees, radians or percentages. In our research, the slope is computed in degrees.

The numerical calculation of the slope (S) consists in determining the angle of the gradient vector in the x and y directions:

$$S = \arctan \sqrt{p^2 + q^2}, \tag{1}$$

where p and q are the components of the gradient vector.

There are many algorithms for computing the slope. They differ in the way they approximate the elevation changes rate in perpendicular directions x and y [21–24]. The algorithm for determining the slope is a focal operation. That means that the value for a given cell is calculated from values in its neighborhood [25, 26]. The choice of the algorithm may result in different slope values, even if the computations are based on the same data [27, 28].

For the numerical determination of the slope, the algorithms most often use a moving 3×3 window [22]. One of such algorithms, widely used in GIS software, is the Horn algorithm (Third-order Finite Difference Weighted by Reciprocal of Squared Distance) [29].

In the context of a single cell with the coordinates `(i, j)`, we denote the elevation in this cell by z_{00}. Now, the elevation of the adjacent west and east grid cells will be denoted z_{0-} and z_{0+}, respectively, while the elevation of the south and north cells will be denoted z_{+0} and z_{-0}, respectively. Figure 1 shows the focal neighborhood of the cell z_{00} with the background of the whole DEM.

The components of the gradient vector are the slope p in the x (west-to-east) direction and the slope q in the y (south-to-north) direction. To compute the values of p and q for each cell, we use the elevation values of 8 points that lie in its immediate neighborhood.

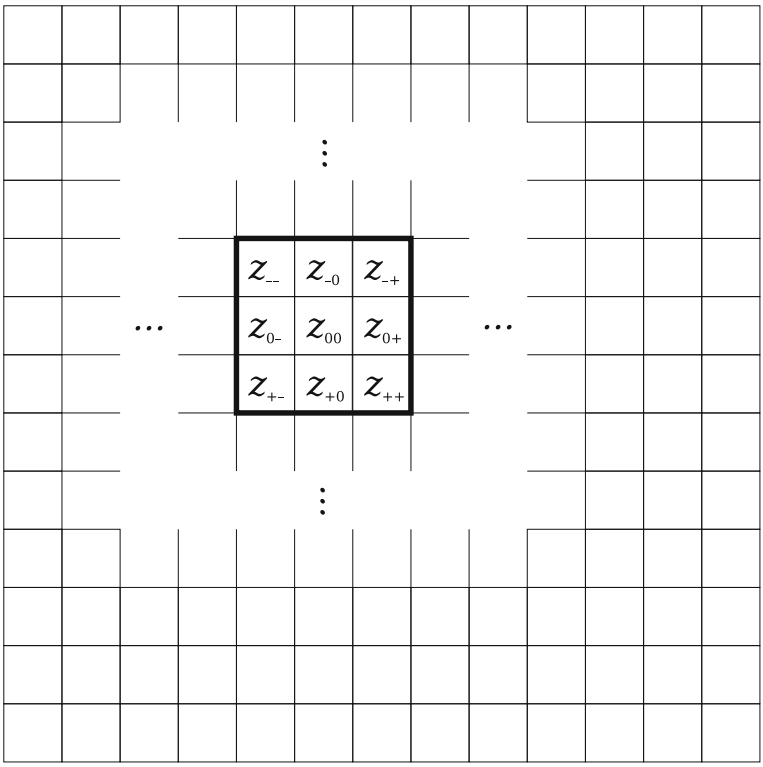


Fig. 1 Focal neighborhood of the cell z_{00}

The estimated value for the component p with the use of the weighted average of three central differences is given by:

$$p = \frac{(z_{-+} + 2z_{0+} + z_{++}) - (z_{--} + 2z_{0-} + z_{+-})}{8\Delta x}, \tag{2}$$

where Δx is the grid interval from west to east, expressed in the same units as the elevation.

Similarly, an estimate for the component q can be expressed by:

$$q = \frac{(z_{+-} + 2z_{+0} + z_{++}) - (z_{--} + 2z_{-0} + z_{-+})}{8\Delta y}, \tag{3}$$

where Δy is the grid interval from south to north, expressed in the same units as the elevation.

The method of determining the slope using (2) and (3) is presented by Algorithm 1. It is a sequential algorithm that receives a DEM of $n \times m$ cells and of the raster cell size $\Delta x \times \Delta y$. It returns as a result a raster of the same sizes with the slope values (however, without calculating the boundary values).

The algorithm requires two arrays — rasters: *dem* stores elevation values, and *slope* — computed slope values in each cell. We also need information about cell size (Δx, Δy) and the number of rows and columns (n, m). The implementation of this algorithm is based directly on (2) and (3) — for each cell in each row the components p in the x direction and q in the y direction are computed, and then the slope value is written to the corresponding cell of the result raster *slope* according to (1).

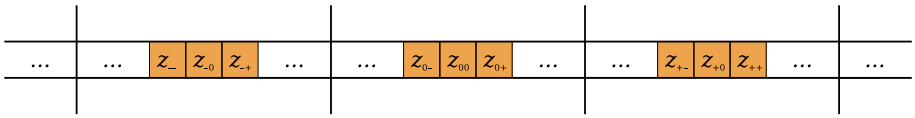


Fig. 2 Memory access in Algorithm 1

This approach determines specific memory access. Figure 2 shows how the data used in one loop iteration are stored in memory. Vertical lines separate the arrangement of subsequent raster rows in memory, and each row contains m cells. To determine both p and q, we need all the items marked in orange, so it can be seen that the pieces of data will be far apart. The amount of data needed means that they will not be able to be retrieved in a single memory reading — this means that in one iteration of the loop the cache memory will be used inefficiently and may be reloaded many times.

Algorithm 1 BaseSlope (B): Basic sequential algorithm.

Input: *dem* — input DEM
Δx — west-to-east cell size
Δy — south-to-north cell size
n — number of rows
m — number of columns
Output: *slope* — output raster of the same size

```
1  for r ← 1...(n − 2) do
2      for c ← 1...(m − 2) do
3          p ← ((dem[r − 1][c + 1] + 2dem[r][c + 1]
4                + dem[r + 1][c + 1])
5                − (dem[r − 1][c − 1] + 2dem[r][c − 1]
6                + dem[r + 1][c − 1]))
7                /(8Δx)
8          q ← ((dem[r + 1][c − 1] + 2dem[r + 1][c]
9                + dem[r + 1][c + 1])
10               − (dem[r − 1][c − 1] + 2dem[r − 1][c]
11               + dem[r − 1][c + 1]))
12               /(8Δy)
13         slope[r][c] ← arctan(√(p² + q²))
14 return slope
```

2.2 Algorithm transformation

An algorithm transformation is needed to utilize vector computing and many cores. Algorithm 2 describes for slope computation.

It is a modification of Algorithm 1, where the inner loop is changed in order to work on vectors, that is, to deal with couple of elements at once. The inner (column) loop (with the control variable `c`) was divided into 6 loops.

In Algorithm 2, we first determine the values of p for the entire row in three steps — each step is one loop. These three consecutive loops must be sequential because there is a data dependency. Then we compute q for the entire row in two separate loops, which must also be sequential because there is a data dependency. In the last (the sixth) loop, which can be

Algorithm 2 TransformedSlope (T): Transformed sequential algorithm.

Input: dem — input DEM
Δx — west-to-east cell size
Δy — south-to-north cell size
n — number of rows
m — number of columns
Output: $slope$ — output raster of the same size

for $r \leftarrow 1 \ldots (n-2)$ **do**
  `/* calculating p */`
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $p[c] \leftarrow dem[r-1][c+1] - dem[r-1][c-1]$
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $p[c] \leftarrow p[c]$
      $+ 2(dem[r][c+1] - dem[r][c-1])$
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $p[c] \leftarrow (p[c] + (dem[r+1][c+1]$
      $- dem[r+1][c-1]))/(8\Delta x)$
  `/* calculating q */`
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $q[c] \leftarrow dem[r+1][c-1]$
      $+ 2dem[r+1][c] + dem[r+1][c+1]$
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $q[c] \leftarrow (q[c] - (dem[r-1][c-1]$
      $+ 2dem[r-1][c]$
      $+ dem[r-1][c+1]))/(8\Delta y)$
  `/* calculating slope */`
  for $c \leftarrow 1 \ldots (m-2)$ **do**
    $slope[r][c] \leftarrow \arctan(\sqrt{p[c]^2 + q[c]^2})$
return $slope$

done only after the end of the previous five loops, we calculate the slope for each element from the row with the index r based on the calculated values p and q.

This division of calculations into 6 loops allows for more efficient access to memory. Figure 3 shows how the data used in one iteration of the outer loop of Algorithm 2 are arranged in memory (vertical lines separate the arrangement of subsequent raster rows in memory). Subsequent subfigures indicate access to memory in subsequent internal loops and correspond to the 5 loops computing p and q. The elements marked with colors are the cells needed for calculations in subsequent internal loops. We can see that to complete each of these loops we need data that is in one row very close together. Thanks to this, they can be loaded into cache in one reading. Then the inner loops each time calculate the entire row. This shows that memory access is much more cache friendly. This means that we reach for slower (RAM) memory less often and the amount of communication between subsequent memory levels is smaller, which usually speeds up calculations.

2.3 Complexity

The complexity of the algorithm (and all its implementations) is linear (that is, $O(N)$, where $N = nm$ is the size of the data) [30]. It is quite obvious if we consider the following:

– the outer r-loop has $O(n)$ iterations;

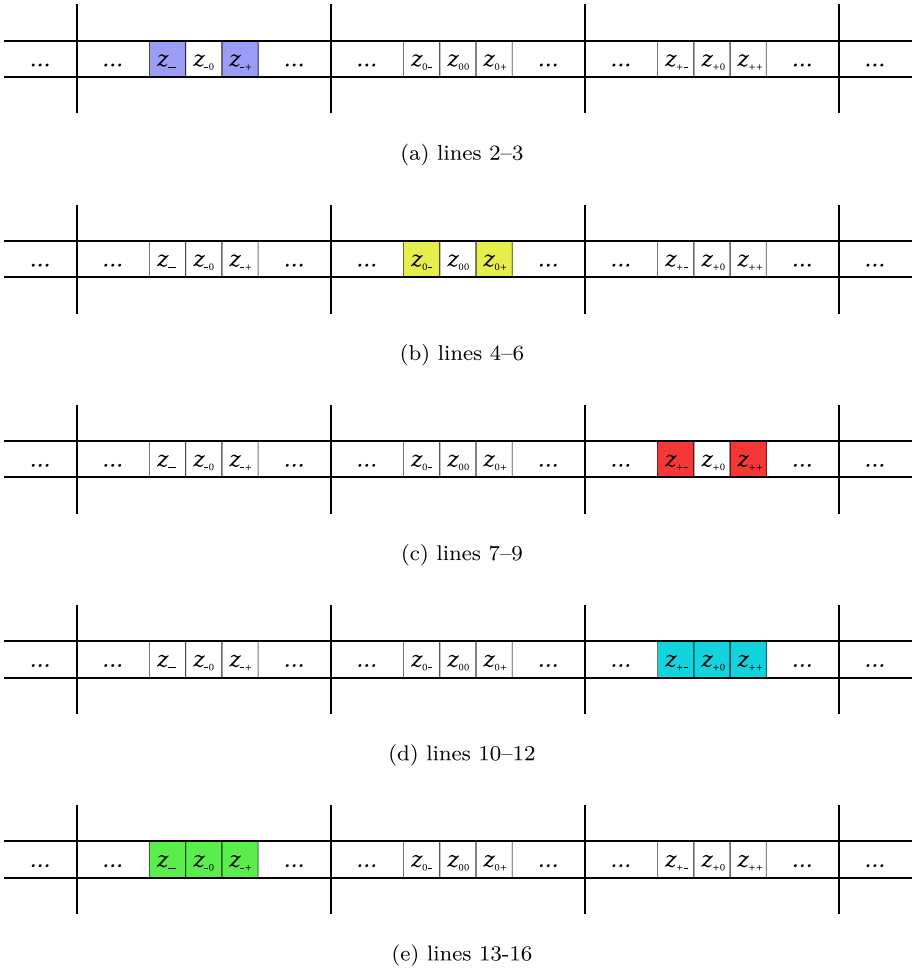


Fig. 3 Memory access in Algorithm 2

– each iteration consists of one (Algorithm 1) or six (Algorithm 2) inner c-loops of $O(m)$ iterations each;
– each of the inner iterations are of constant time.

Thus, together we have the complexity of $O(n)O(m) = O(nm) = O(N)$, that is, linear.

2.4 Implementation details

The first implementation of Algorithm 1 is strictly sequential and does not use the processor's vector properties. It runs with the use of only one thread, performing the slope computation based on the formulas (2), (3) and (1). Later in this paper, we refer to it as `'base'` (B).

However, there is a possibility to parallelize this algorithm using OpenMP to fully utilize parallel machines with shared memory. We have nested loops, so we choose the outer loop to parallelize. The implementation of Algorithm 1 with the use of OpenMP pragmas can be

run on various multicore processors. We refer to this implementation as `'parallelized base'` (PB).

The pure implementation of Algorithm 2 is later referred to as `'transformed'` (T). It is still sequential code, executed with only one thread.

Vector processing involves executing the same instruction on multiple arguments simultaneously. The easiest way to use the capabilities of vector units is to use optimizing compilers. Such vectorization is automatic and does not require any changes in implementation by the programmer. Algorithm 2 (i.e. the transformation of Algorithm 1) allows the compiler to apply vectorization automatically, but may find some expressions too complicated, and then manual selection of these places allows using vectorization fully. The most common method of manual vectorization is to use the SIMD directives of the OpenMP standard [31, 32]. This gives the compiler more information than it can extract from the code by itself, enabling better optimization. This can be done by placing `#pragma omp simd` before each `c`-variable loop (i.e. each inner loop) in Algorithm 2. The sequential implementation of Algorithm 2, vectorized using SIMD pragmas, is referred to as `'transformed, with SIMD'` (TS).

Algorithm 2 can also be easily parallelized using the OpenMP standard. The outer loop can be annotated with `#pragma omp parallel for` and thus it can be executed in parallel. The internal loops are simultaneously vectorized using SIMD extensions (as above). We call this implementation `'parallelized transformed, with SIMD'` (PTS).

3 Test data

The area selected for our tests covers a part of south-eastern Poland (Fig. 4) that varies in terms of its elevation, morphometry, and hydrology. According to the physical-geographical division of Poland [33], it is situated in three macro-regions: Lublin Upland, Roztocze and Sandomierz Basin (Fig. 5).

The input data were the data provided by the Polish Head Office of Geodesy and Cartography (GUGiK) in the *ASCII XYZ GRID* format. They comprise text files containing point coordinates (X, Y, Z) in a regular grid with a 1-meter mesh. The points were interpolated based on a point cloud from Airborne Laser Scanning (ALS) or from measurements conducted on aerial photographs as part of the update necessary for creating the orthophotomap. The average error of ALS elevation data has a value of up to 0.2 m, while the average error of aerial photograph elevation data is in the range of 0.8–2.0 m. Individual files correspond to the range of map sheets in the flat rectangular PL-1992 coordinate system at a scale of 1:5 000 (1/4 sheet 1:10 000). The files do not have any 'no-data' cells [34]. The PL-KRON86-NH system is a system of normal heights.

Table 1 Selected spatial characteristics of test regions (values are given in the rectangular flat coordinate system PL-1992)

Name	Range				Area [km^2]
	x_{min}	x_{max}	y_{min}	y_{max}	
10R	740000	790000	310000	350000	2000
5R	740000	790000	310000	330000	1000
R_a	740000	760000	310000	320000	200
R_b	750000	770000	320000	330000	200
R_c	770000	790000	340000	350000	200

Table 2 Characteristics of given test regions' files

Name	Rows	Columns	Cells	Size [GB]
10R	50000	40000	2000000000	7.45
5R	50000	20000	1000000000	3.73
R_a	20000	10000	200000000	0.74
R_b	20000	10000	200000000	0.74
R_c	20000	10000	200000000	0.74

434 sheets were selected for the tests. The sheets were combined into one file in TIFF format with a resolution of 1 × 1 m. The tool `Mosaic to new raster` from `ArcGIS for Desktop 10.6` was used to merge the data. Within its range, 5 test regions have been designated (Fig. 5). Three areas of 200 km^2 were labeled: R_a, R_b, R_c. These three regions have the same size (`R`) of 200 km^2, but vary in elevation. The next region `5R` is 1000 km^2, which is 5 times the size of `R`. The largest is the region `10R` with an area of 2000 km^2 (10 times larger than the regions `R`) and it covers the above mentioned 4 regions. By using multiples of the size (`R`, `5R`, `10R`), we can investigate the behavior (especially, execution time) of the given algorithm for an increased amount of data. These regions were extracted into separate files in TIFF format using the tool `Clip` from `ArcGIS for Desktop 10.6`. Detailed characteristics of the regions can be found in Table 1.

The sizes of spatial data used for testing are described in Table 2.

To process raster data, they must be stored in operating memory. Because the data are in the form of a raster — that is a matrix, they are often stored as a two-dimensional array.

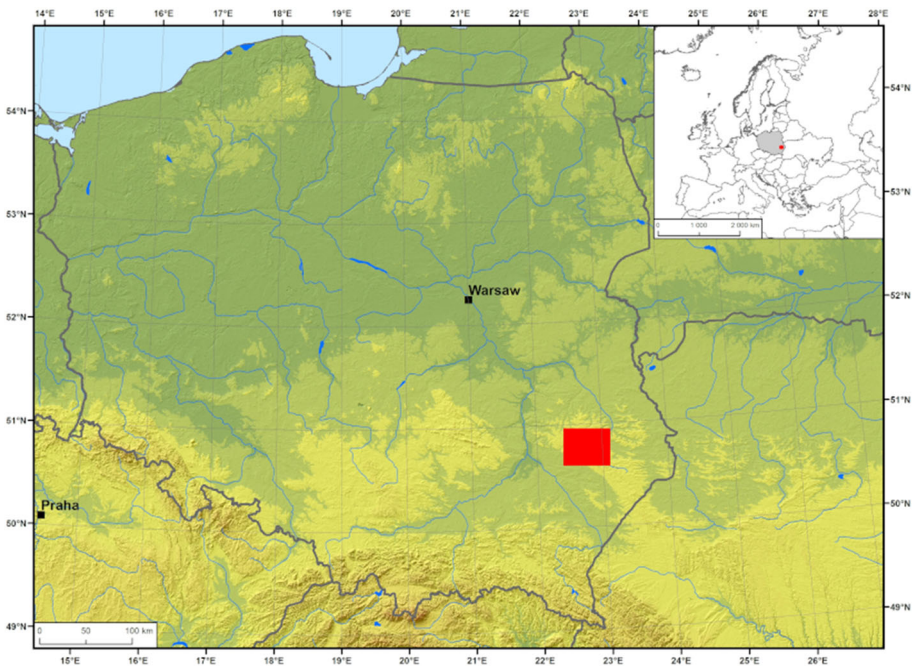


Fig. 4 Study area

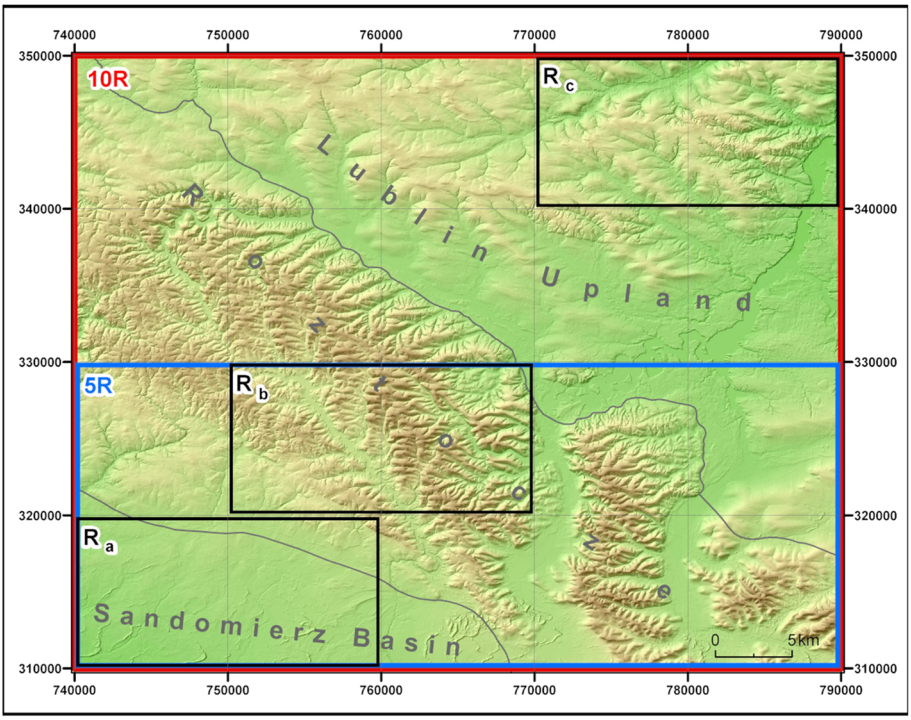


Fig. 5 Test regions R_a, R_b, R_c, 5R, 10R

Another way of representing them in the random access memory is to store the matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(where `m` is the number of rows and `n` is the number of columns) in a one-dimensional array. According to one of the storage schemes for matrices of general form, presented in the LAPACK [35] library and the convention for storing two-dimensional arrays in C/C++ languages, the matrix elements are stored row-by-row (that is, in *row-major order*):

$$A = \left(a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{m1}, \ldots, a_{mn}\right).$$

4 Numerical experiment

In this section, we analyze the execution time of our 5 implementations.

`base (B)` — strictly sequential implementation of Algorithm 1.
`parallelized base (PB)` — parallel implementation of Algorithm 1, that is the `base` version parallelized with OpenMP pragmas using shared memory of many threads.
`transformed (T)` — sequential implementation of Algorithm 2. We expect that this transformation will allow for more efficient memory access and for the use of vector

mechanisms (here, automatically by the compiler options; and in the next implementation, manually by `#pragma omp simd`).

`transformed with SIMD (TS)` — sequential implementation of Algorithm 2, which uses the `#pragma omp simd` construct, which points the compiler to vectorization possibilities.

`parallelized transformed with SIMD (PTS)` — parallel implementation of Algorithm 2. This is highly manually optimized code, where the `transformed` algorithm has `#pragma omp parallel for` so that it can be executed in parallel with many threads. SIMD extensions are also used (`#pragma omp simd`) so that it can be effectively vectorized.

All versions were implemented in C++, while the computations were performed with the use of `float` numeric type.

The tests were carried out using computing platforms equipped with modern multicore processors:

Sandy Bridge

```
processor: 2x Intel Xeon E5-2660 2.20GHz
(2x8 cores with HT)
RAM: 48GB (6x8GB DDR3 1600MHz ECC)
```

Haswell

```
processor: 2x Intel Xeon E5-2670 v3 @ 2.30GHz
(2x12 cores with HT)
RAM: 128GB (8x16GB DDR4 2133MHz ECC)
```

Both units were equipped with the same software:

```
operating system: CentOS 7.6
kernel: Linux 3.10.0
compiler: GCC 8.3.1 + OpenMP 4.5
libraries: GDAL 2.4.0
           GRASS 7.4.1
```

The programs were compiled using the GCC compiler. The Geospatial Data Abstraction Library (GDAL) [36] was used to read and write GeoTIFF files. In each case, data alignment in memory was used as optimization support.

By 'alignment' we mean that the data is (whenever possible) aligned to an integer multiple of the processor word size. This alignment (not difficult to ensure) allows better use of both cache and vector instructions.

As the first experiment, we verified that the run time does not depend of the relative elevation diversity.

The second step was to explore the possibility of automatic vectorization offered by the compiler (Section 4.1) by setting the compiler flags accordingly.

-O0 — a default option. The purpose of the compiler is to reduce the compilation time, flags that improve performance or code size are not enabled.

-O1 — the compiler tries to reduce the execution time and code size without performing optimizations that significantly increase compilation time (basic optimization).

-O2 — the flag -O1 is turned on, but also all possible optimizations that will not cause drastic growth of result code size are included (moderate optimization). The compiler performs optimization based on knowledge of the program, but sometimes even with this option, due to the complexity of individual expressions, the compiler is not able to apply the appropriate optimization mechanisms.

Table 3 Speedup of `parallelized base` and `parallelized transformed with SIMD` implementations for different data sizes and for the number of threads equal to the number of cores on `Sandy Bridge` and `Haswell` architectures relative to performance with one thread

		Sandy Bridge	Haswell
R	parallelized base	7.62	6.99
	parallelized base with SIMD	10.01	8.98
5R	parallelized base	6.38	9.26
	parallelized base with SIMD	8.00	11.56
10R	parallelized base	6.88	10.64
	parallelized base with SIMD	8.36	12.53

-O3 — the flag -O2 is turned on, but also optimizations that can significantly increase the volume of the result code (such as inlining) are included. At applying this optimization the compiler tries automatically to transform and vectorize loops (aggressive optimization).

The impact of the number of threads on program execution time was also examined (section 4.2), i.e. the comparison of code parallelization efficiency by using multithreaded programming on a multicore CPU, using the OpenMP interface. In our tests we never used more threads than available physical cores so that there is a one-to-one relationship between

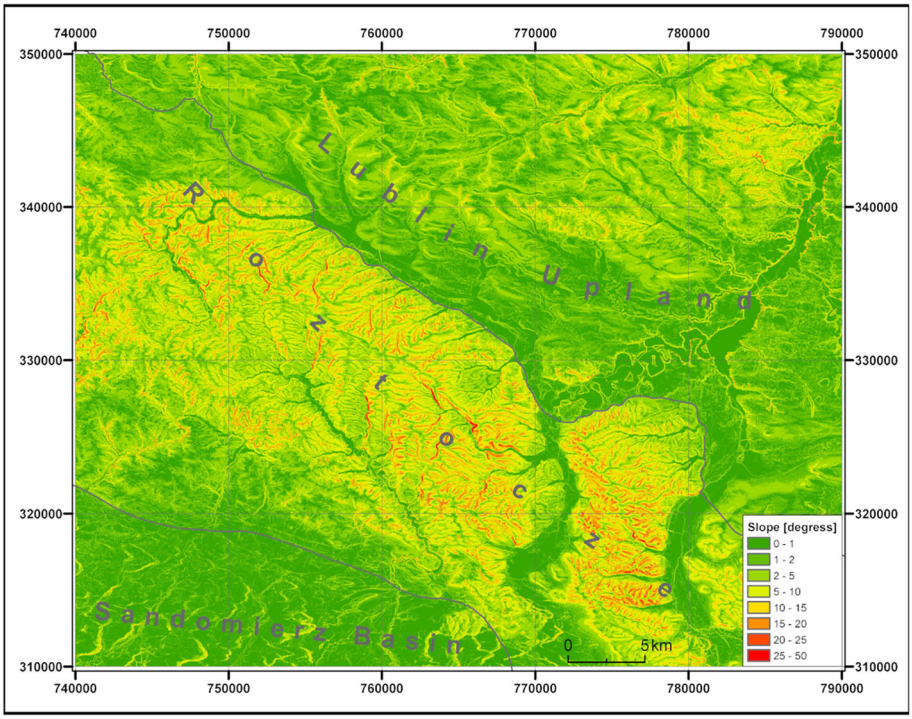


Fig. 6 Slope values visualization in the 10R test region

running threads and used cores. The results of the parallel program were compared for various number of threads (Table 3).

The running time of the entire program consists of: input data reading time, calculation time and time to save the results to disk. When examining the performance of the algorithm, the most interesting is the reduction of computation time, so during work we focused on this aspect. Both loading data and saving results largely depends on the hardware configuration (the way components are connected, the distribution of data on disks, space for saving results, environment options), so we did not consider these issues.

The scalability of parallel implementations relative to data size was also analyzed, as shown in Section 4.3.

The slope values resulting from our implementations have been verified for compliance.

An attempt was also made to compare our algorithms with existing solutions (Section 4.5).

The verification of the correctness of the resulting data (Fig. 6) obtained as a result of the parallelized implementations consisted in comparing them with the result file generated by the sequential program. The comparison was made using the `gdalcompare` tool from the GDAL library [36]. This tool is used to find differences in two files in the geotiff format: both differences in individual raster cells, as well as metadata saved in the file.

For all analyses, the results of the parallel programs with multithreading were identical to the results of the sequential program.

4.1 Impact of compiler options

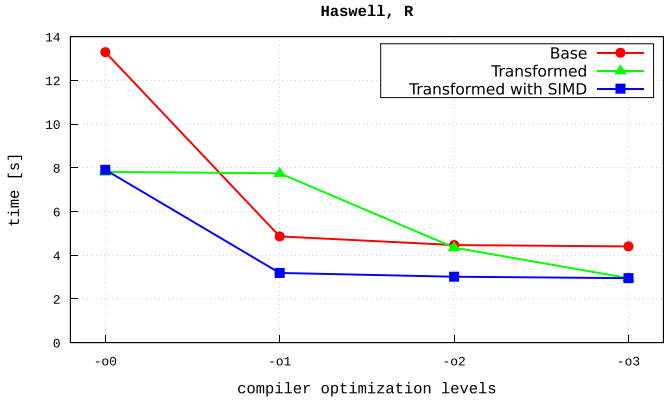
The graphs in Fig. 7 show the impact of compiler optimization levels on single threaded performance of sequential implementations (`base`, `transformed`, `transformed with SIMD`) on `Haswell` architecture for different data sizes: (a) — `R`, (b) — `5R`, (c) — `10R`.

One can see that the most basic compiler optimization level, i.e. `-O1`, optimizes the basic code (base) quite efficiently. Application of further compiler optimization levels, i.e. `-O2`, `-O3`, does not shorten the execution time, regardless of the data size. Only at these options levels, further optimizations, including data dependencies, are used and might produce a vectorized code. In this case, the execution time does not change, the compiler is not able to vectorize the loop without additional information.

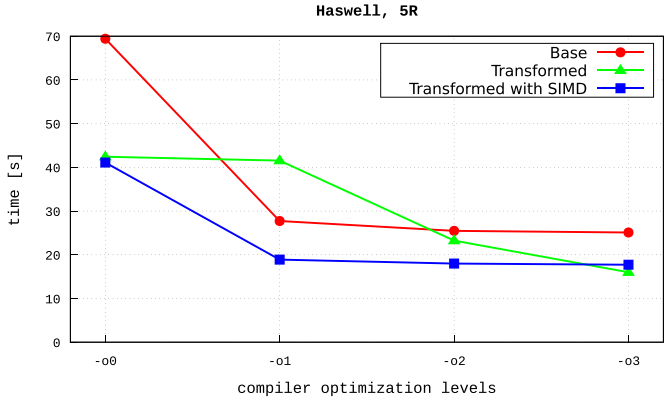
Notice that, regardless of the data size, our transformed algorithm, without applying any compiler optimization options, works better than basic version. This is a consequence of better memory access.

The result of modifications applied to the `transformed` version make the basic compiler optimization level (`-O1`) unable to shorten the execution time. But for each successive compiler option (`-O2`, `-O3`) algorithm is performed faster. It shows that only the elimination of data dependencies in the code of the algorithm, gives the compiler the possibility to apply automatic vectorization.

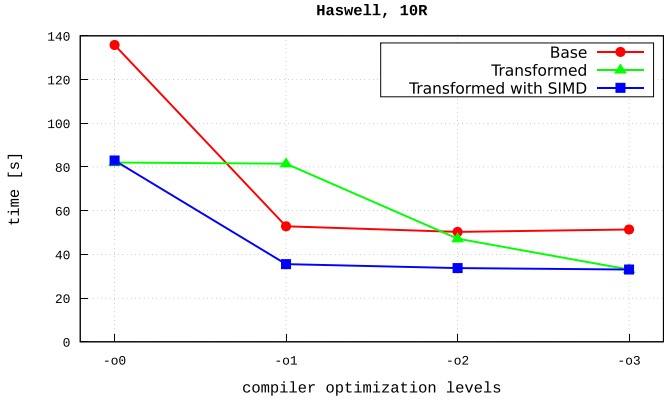
The `transformed with SIMD` algorithm, with no compilation optimization options enabled (`-O0`), also runs shorter than the `base` version. Next, the `-O1` option reduces execution time and is the fastest for this version. Indicating by `#pragma omp simd` the fragments of code for vectorization causes the `transformed with SIMD` version to achieve the same execution time in every situation whenever the compiler optimization options, i.e. `-O1` , `-O2` and `-O3`, are enabled. Therefore, based on the conclusions from Fig. 7, the `-O3` optimization with `#pragma omp simd` was adopted for further testing.



(a) Region R

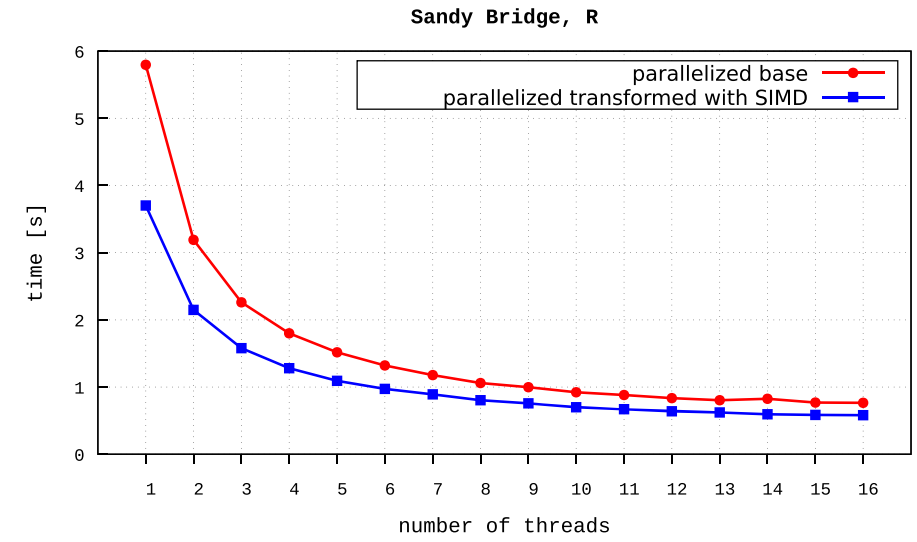


(b) Region 5R

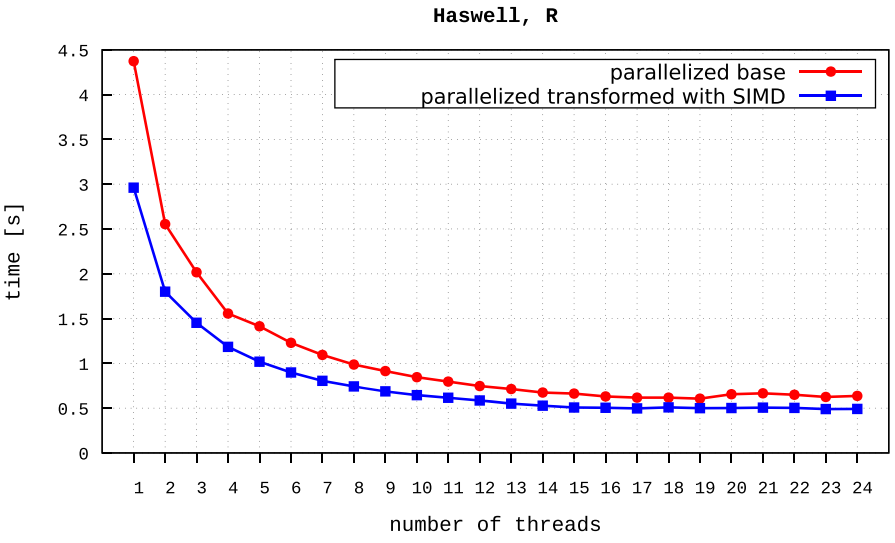


(c) Region 10R

Fig. 7 Impact of compiler optimization options on the execution time of implementations running with one thread for different data sizes



(a) Sandy Bridge architecture

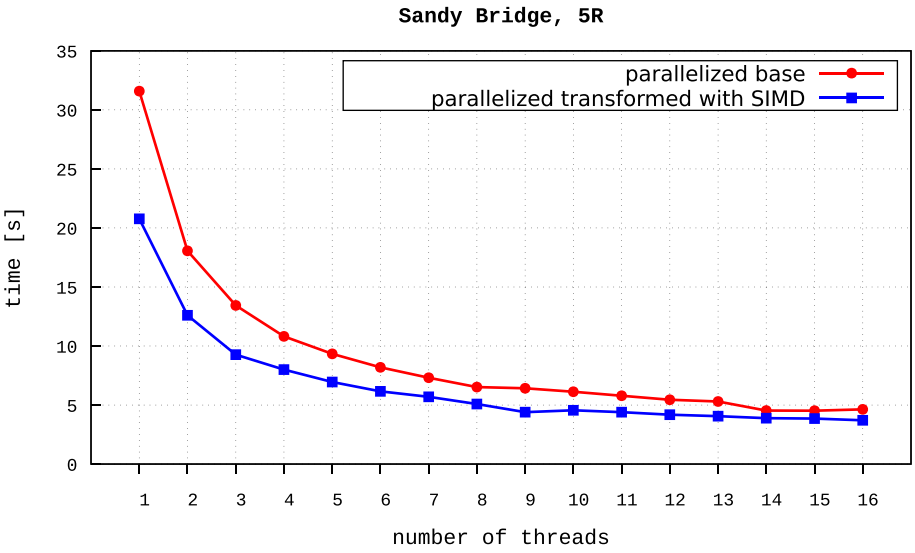


(b) Haswell architecture

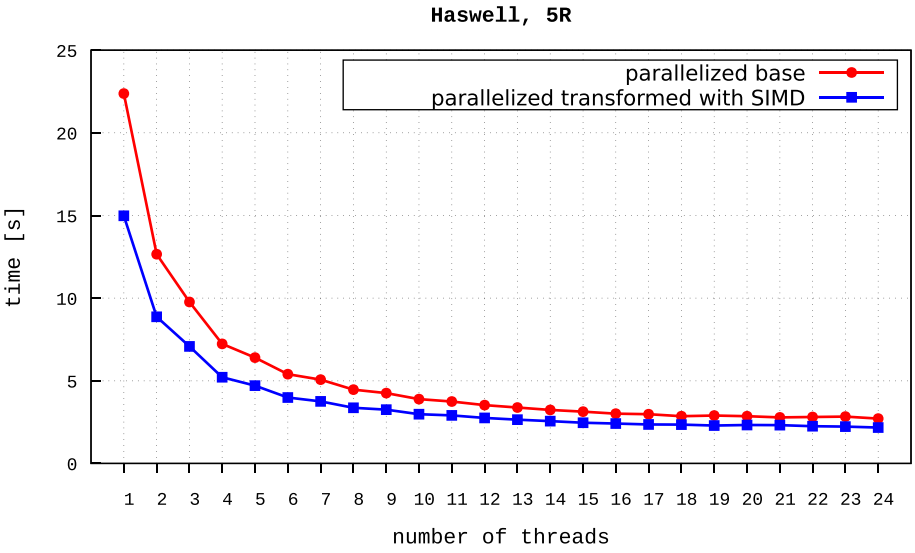
Fig. 8 Parallel implementations' execution times for data size `R` running for different number of threads

4.2 Scalability relative to the number of threads

The charts in Figs. 8, 9 and 10 show the execution time of parallel versions (that is `parallelized base` and `parallelized transformed with SIMD`) for individual data sizes, running on `Sandy Bridge` and `Haswell` architectures for a different number of threads without using Hyper-Threading (HT) technology.

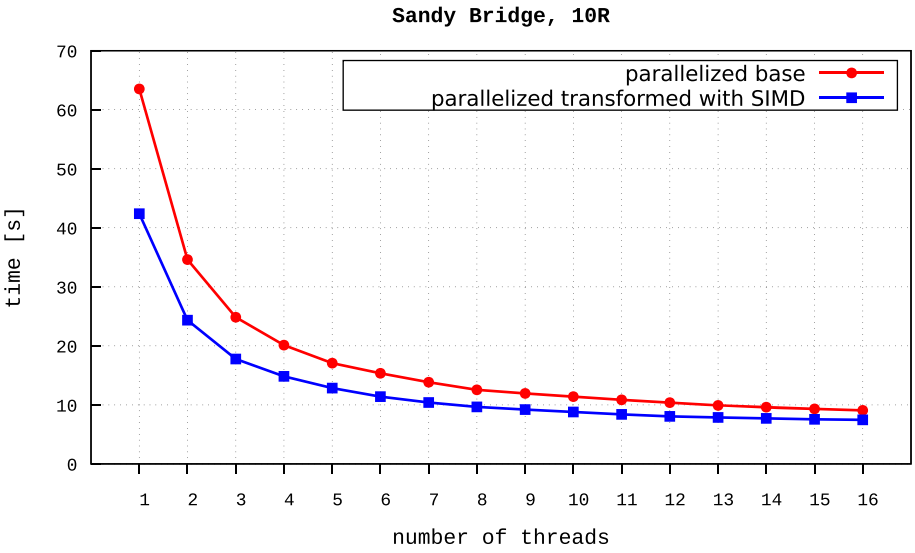


(a) Sandy Bridge architecture

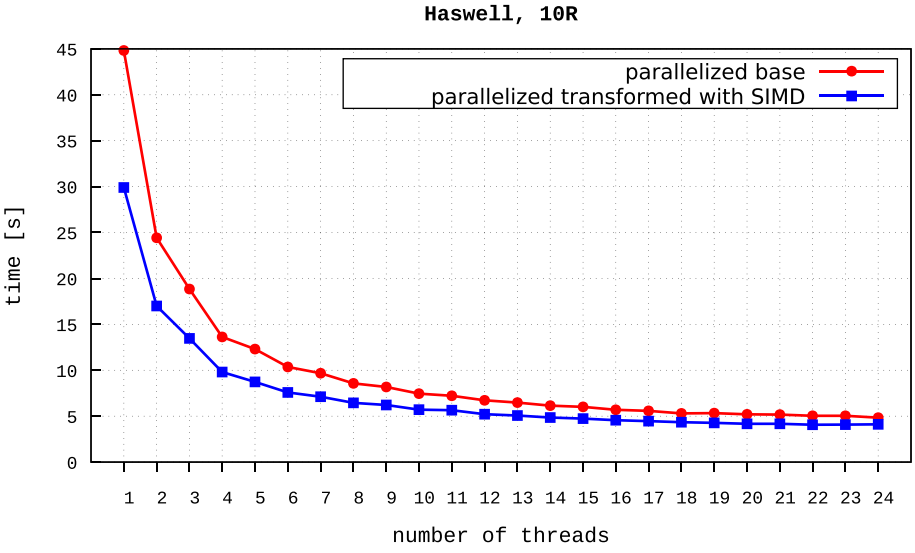


(b) Haswell architecture

Fig. 9 Parallel implementations' execution times for data size `5R` running for different number of threads

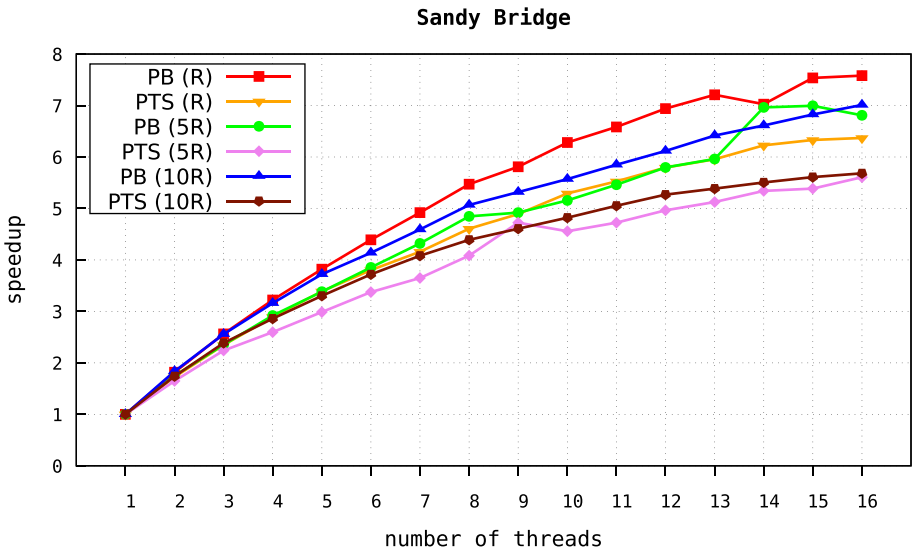


(a) Sandy Bridge architecture

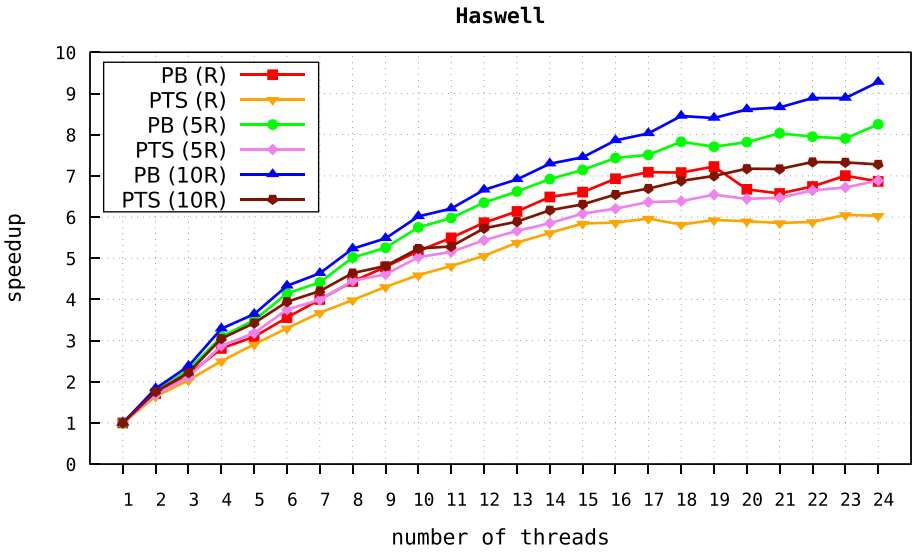


(b) Haswell architecture

Fig. 10 Parallel implementations' execution times for data size `10R` running for different number of threads

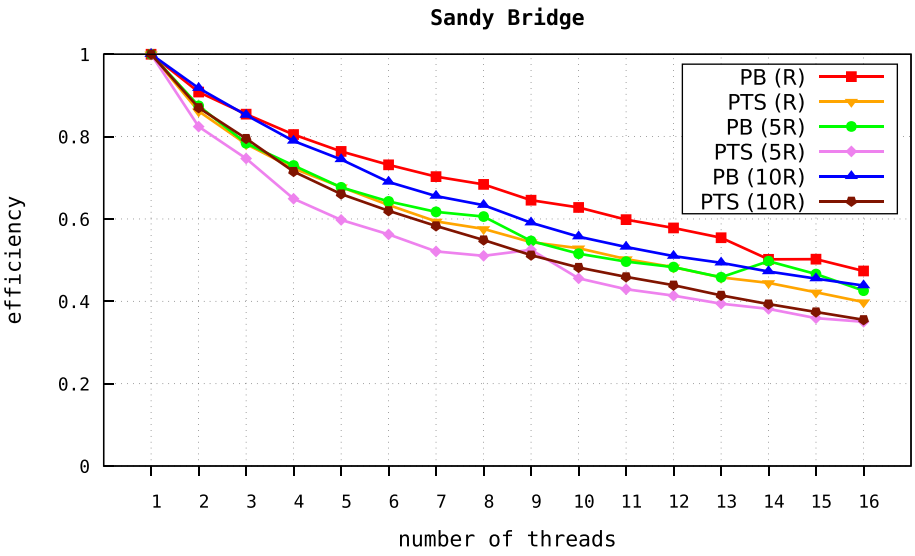


(a) Sandy Bridge architecture

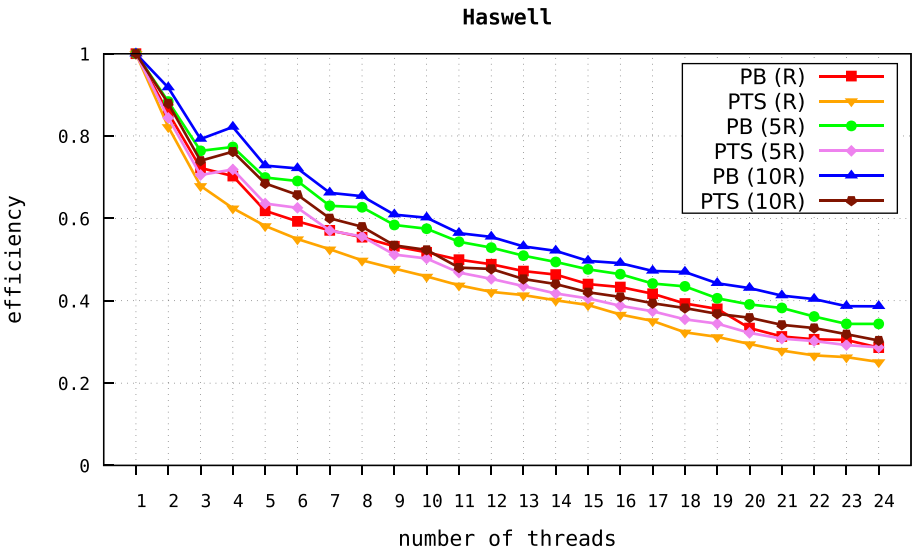


(b) Haswell architecture

Fig. 11 Speedup of parallel implementations for different data sizes



(a) Sandy Bridge architecture



(b) Haswell architecture

Fig. 12 Efficiency of parallel implementations for different data sizes

In Fig. 11, we present the speedup and in Fig. 12 — efficiency of our parallel implementations. These metrics are well-known [37]. The speedup S_n for n threads describes the gain from using many threads and it is defined as

$$S_n = T_1/T_n,$$

where T_n is the run time of the implementation using n threads. The efficiency E_n for n threads shows how well the additional threads are utilized and it is defined with the speedup as

$$[E_n = S_n/n.$$

On the basis of the tests carried out, it can be seen that, regardless of the architecture, the algorithm is scalable with respect to the number of threads, i.e. for the same size of input data, the computation time decreases as the number of threads increases. Comparison of the results on both architectures shows that the best ones are obtained for the number of threads equal to the number of available cores. Table 3 shows the speedup obtained for the number of threads equal to the number of cores on a given architecture in relation to the execution with one thread. On `Sandy Bridge`, the `parallelized base` implementation achieves speedup from 6.38 (for `5R`) to 7.64 (for `R`), while on `Haswell` from 6.99 (for `R`) to 10.64 (for `10R`) — here we can see that the more data we have, the more speed we gain. For the `parallelized transformed with SIMD` implementation on `Sandy Bridge`, we achieve speedup from 8 (for `5R`) to 10.01 (for `R`), while on `Haswell` from 8.98 (`R`) to 12.53 (`10R`) — here we can also see that the more data we have, the bigger speedup is. It is also clear that regardless of the architecture, `parallelized transformed with SIMD` works faster.

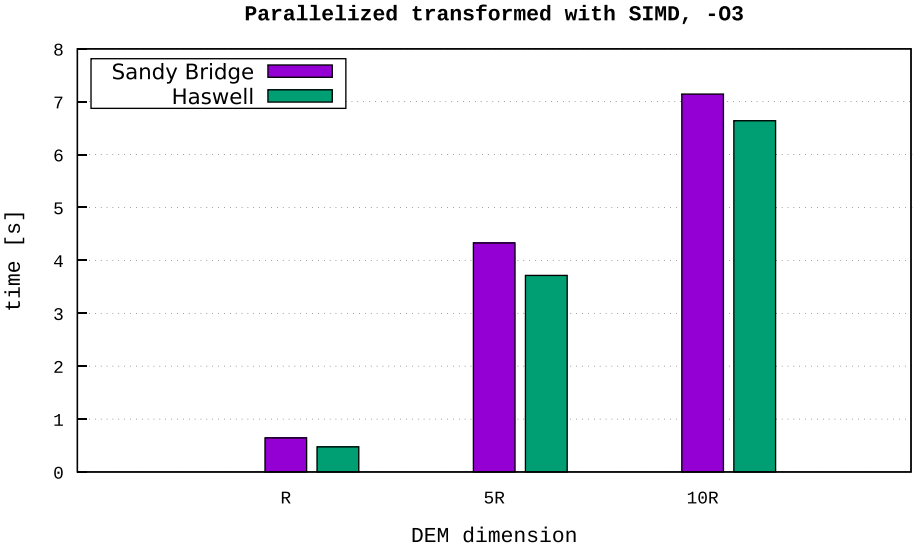


Fig. 13 Execution time of `parallelized transformed with SIMD` for different data sizes and for the maximum number of threads on `Sandy Bridge` and `Haswell`

4.3 Scalability relative to the data size

Figure 13 presents the execution time of `parallelized transformed with SIMD` version for different data sizes on both architectures (`Sandy Bridge` and `Haswell`) for the number of threads equal to their respective number of available physical cores, i.e. 16 and 24, respectively. This chart shows a clear increase in computation time with increasing data size, i.e. for `Sandy Bridge` it is 0.6 s for `R`, 3.7 s for `5R`, 7.5 s for `10R`, and for `Haswell` it is 0.5 s for `R`, 2.2 s for `5R` and 4.1 s for `10R`. Moreover, we can also see that the execution time increases proportionally, which allows us to predict the running time for larger data sizes, e.g. we can assume that the slope computation time on `Haswell` for 24 threads and the size `20R` will be about 10 s (i.e. about 20 times longer than the time for the size `R` of 0.5 s).

In addition, we can note that changing the architecture to a newer one (`Haswell`) and with more cores shortens the execution time of the same algorithm without the need of the program code modification .

4.4 Weak and strong scalability

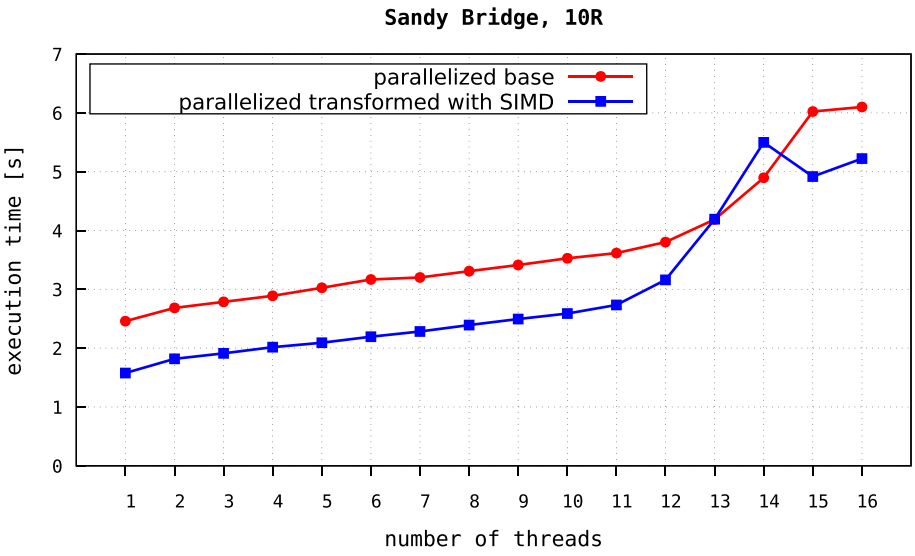
Figure 14 shows the weak scalability of the algorithms. The tests here are being conducted for different numbers of processors, but the amount of work is proportional to the number of threads used. To achieve a nice weak scalability [38], we expect the plot (execution time as a function of the number of threads used) be a horizontal straight line. However, our plots are not that favorable. The weak scalability is hard to achieve because for our algorithms, small data sizes are not representative to the real applications for which the implementations are adjusted.

Figure 15 shows the strong scalability of the algorithms. This time tests were run for a different number of processors, but the amount of work was always the same (size `10R`). The plot (runtime vs. number of employed processors) was performed on a logarithmic-logarithmic [38] scale. In such a plot good strong scalability should result in a straight line with a slope of -1. Here, the plots are closer (than the plots of weak scalability) to straight lines. It is caused by the fact that here all the tests are done with maximal data size and our implementations deals better with such input. Thus, the strong scalability is (maybe contrary to expectations) somewhat fulfilled better than the weak scalability.

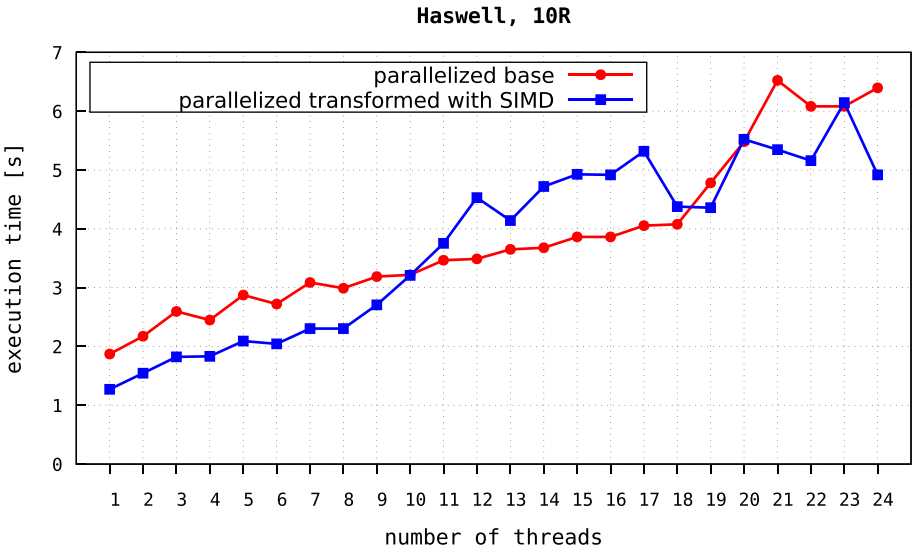
4.5 Comparison with existing solutions

In this section, we compare our most effective implementation (`parallelized transformed with SIMD`) with implementations of the slope algorithm in most popular open-source packages (GDAL/QGIS and GRASS GIS) as well as commercial software, (ArcGIS for Desktop).

We verified the correctness of our implementation by comparing its results with the results of the Horn algorithm implemented in above mentioned GIS software. All four implementations (the three reference programs and our implementation) give slightly different results. However, the differences are negligible.

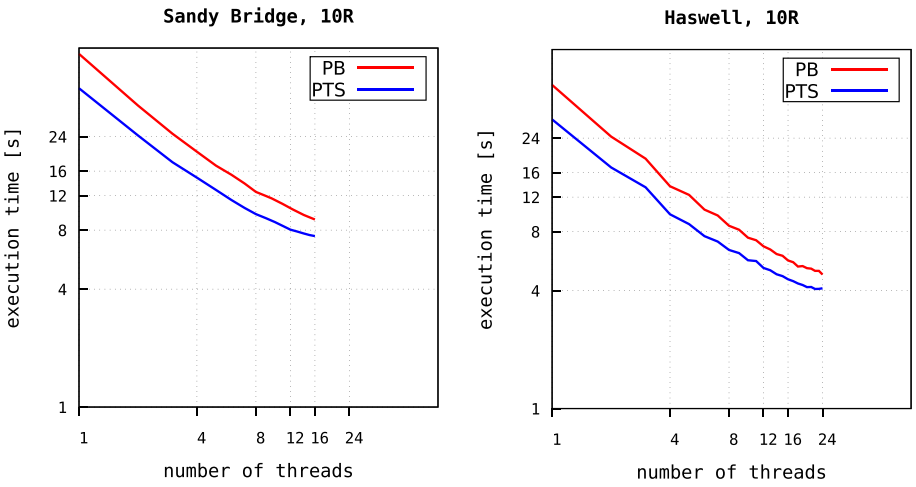


(a) Sandy Bridge architecture



(b) Haswell architecture

Fig. 14 Weak scalability (data size is proportional to the number of threads; maximum is `10R`)



(a) Sandy Bridge architecture (b) Haswell architecture

Fig. 15 Strong scalability for data size `10R`

GDAL (also used in QGIS)

GDAL [36] is a library used by many packages mainly for reading and writing data in various raster formats. GDAL also provides functions and utility programs for performing various operations on spatial data, including operations on DEM — like `gdaldem` program.

When measuring the GDAL slope algorithm running time we used `gdaldem` with the following call:

```
gdaldem slope dem.tif slope.tif
```

The slope algorithm in GDAL is implemented in the following manner: after opening the data file each row of the raster is loaded into memory along with two adjacent rows. Then for each cell of the row its entire neighborhood is read into a nine-element array and the slope of the cell is calculated. The calculation result is immediately stored in the appropriate cell of the resulting raster file. Due to this approach, it is not possible to separate the calculation time, so in this case we can only measure the total time for loading data into memory, performing calculations and saving the result.

Moreover, GDAL uses only one core — so nothing is parallelized.

GRASS GIS

GRASS GIS [39] is a general purpose open-source GIS software suite with raster and vector processing capabilities.

GRASS GIS uses its own data format. All data needed in computations must be collected in one directory (`GRASS GIS database`).

GRASS GIS implementation of the slope algorithm is very simillar to the GDAL one. Therefore, also in this case, it is not possible to separate the computation time from the reading and writing time.

During the tests the following command was used:

```
grass74 database --exec r.slope.aspect elevation=dem
slope=slope
```

GRASS GIS also utilizes only one core, without parallelization.

ArcGIS for desktop

ArcGIS for Desktop is a geographic information systems (GIS) software consisting of a collection of products which can be deployed on mobile devices, laptop, desktop computers and servers. It is released by Esri and serve for building many practical GIS applications in different areas [40]. For the past few releases, Esri has been increasing the number of tools that can take advantage of parallel processing. This allows some tools to distribute their work across multiple cores. ArcGIS Desktop 10.6 has approximately 30 such tools, but a direct slope tool is not supported. This software is closed-source, thus, the details of implementation are hidden.

The slope algorithm in ArcGIS for Desktop was tested on a machine with hardware specifications just like Haswell, but with other software:

```
operating system: MS Windows Server 2012 R2 Standard
ArcGIS version:   10.6 for Desktop
```

Comparison of the running time

It is not possible to accurately compare the execution time of our algorithm with existing solutions. Problems that arise are: different ways of storing input data, no built-in tools for measuring time, different ways of organizing data processing — which entails the inability to separate the time of reading data, computations and saving the resulting file — as it did during testing our algorithms.

The same slope computation method was used in all packages, namely, approximate partial derivatives by the Horn's formula [29]. Although the same algorithm is the basis in every tool, its implementations are different.

GDAL and GRASS GIS were compiled in our Haswell environment. When calling `./configure` for GDAL, an option `–with-threads` was added, among others; and for GRASS GIS — options `–with-openmp` and `–with-pthread` to make these programs use multithreading if they can.

We tried to standardize tests by performing the same operation for each package: create a slope file in GeoTIFF format based on a DEM file in GeoTIFF format. However, the GRASS GIS requirement of all the data being in its internal format and collected in the internal database would cause two additional steps: importing DEM from the GeoTIFF file and exporting the result to the GeoTIFF file. These steps do not occur in the case of other tested software and the time of their execution does not provide any meaningful information for the comparison. Therefore, we decided not to use the GeoTIFF format in the case of GRASS GIS.

Similarly to the procedure of measuring the time of our implementations, for each program (GDAL, GRASS GIS, ArcGIS) five launches for each measurement were made and all given times are the average times. Between subsequent starts, the cache and hard disk buffers in RAM were not cleared, so the input data was read directly from RAM. So, the first launch (treated as a warm-up) was not taken into account when calculating the average running time.

The obtained execution times (in seconds) are presented in Table 4. Wherever it was possible, time was measured in such a way as to get as close as possible to the time of

Table 4 Execution time comparison (in seconds) on `Haswell` of our `parallelized transformed with SIMD` (PTS) implementation with 1and 24 threads (1thr and 24thr, respectively) with exisiting solutions

		1R	5R	10R
GDAL	**total**	**14.85**	**78.48**	**158.99**
GRASS GIS	**total**	**39.31**	**219.93**	**432.18**
ArcGIS	**total**	**25.01**	**114**	**230**
PTS 1thr	reading	0.93	4.27	8.50
	computations	2.96	14.98	29.90
	writing	3.22E-04	2.87E-04	3.43E-04
	total	**3.89**	**19.25**	**38.40**
PTS 24thr	reading	0.98	4.25	8.70
	computations	0.49	2.18	4.11
	writing	5.65E-05	6.63E-05	5.24E-05
	total	**1.47**	**6.43**	**12.81**

the computations alone. This way the execution time is given also for our `parallelized transformed with SIMD` (PTS) implementation with 1 and 24 threads. Here, 'reading' means reading the input data and allocating them in the main memory, 'computations' are the time to perform computations until the resulting raster is created in the main memory, and the 'writing' is the time to generate the result file. In the case of GDAL, GRASS GIS and ArcGIS, due to the manner of work, it is not possible to separate its individual stages. On the basis of the results obtained, it can be clearly seen that the solution we propose significantly speeds up the computation of the slope for each data size, even with one thread, and the implementation with 24 threads embiggens this difference.

5 Summary

The development of parallel computer architectures provides efficient solutions to improve the processing speed of the geospatial algorithms. In this paper we present a methodology that allows acceleration of calculations performed on raster data and based on neighborhood operations. The proposed methodology is based on such a transformation of a known and widely used algorithm (here: for slope computation), which enables the use of multicore computers with shared memory.

During conducted tests, we showed that the slope algorithm transformation based on splitting loops can be efficiently implemented on modern multicore architecture with shared memory using high-level directive-based OpenMP standard.

The performance tests show that the parallel implementations of the slope algorithm are effective to enhance the computation efficiency on the multicore architectures. The numerical experiments carried out showed that algorithm transformations are needed to show the compiler pieces of code that can be vectorized. The methodology used here enables a better utilization of vector units available in modern processors. It causes speeding up the slope algorithm even with one thread.

Parallelization of the transformed algorithm significantly reduces the running time of the program. On the basis of the tests carried out, we can see that our transformed algorithm is scalable with respect to the number of threads, i.e. for the same size of the problem, the

computation time is reduced as the number of threads increases. The algorithm is also scalable with respect to input data size. In addition, the use of a newer architecture with more cores gives shorter computation time without changing the source code. Our research shows that on multicore processors it is always worth using OpenMP and the number of threads equal to the number of physical cores for this type of calculation.

The effectiveness of the developed methodology was also verified for the correctness of the results, as well as the calculation time in comparison with other implementations of commonly available geospatial platforms such as GDAL, GRASS GIS and ArcGIS. Our parallel-vector implementation significantly outperforms others.

The proposed approach can be used to create high-performance, scalable code suitable for computer systems based on multicore processors with shared memory. The proposed methodology should benefit the programmers who optimize the complex geospatial computations which use neighborhood operations and inspire them for further efforts in this field.

In future works, we plan to extend the study to other architectures, especially GPU and CPU-GPU hybrids — to see how the described methodology operates on those architectures. We also intend to investigate another geospatial algorithms based on DEM, for example, focal hydrological analyses (e.g., flow direction). Another challenge is the implementation of the parallel slope algorithm on a hybrid cluster with multicore processors and GPUs.

Author Contributions B. Bylina: supervision; data analysis and interpretation. J. Bylina: visualisation; writing — review and editing. Ł. Chabudziński: data curation; visualization; writing — review and editing. K. Karpowicz: investigation; resources; data analysis and interpretation. M. Klisowski: software; data analysis and interpretation; writing — review and editing. P. Oleszczuk: visualization; writing — review and editing. J. Potiopa: writing — original draft preparation; writing — editing; data analysis and interpretation. P. Stpiczyński: conceptualization; methodology; software.

Availability of data and material The DEM data (10R region) is available at http://matrix.umcs.pl/k/10R.tif

Code Availability The source code of all implementations discussed is available in the public repository at https://github.com/gis-umcs/fast-slope-article/.

Declarations

Conflict of interest Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Hohl A, Delmelle E, Tang W (2015) Spatiotemporal domain decomposition for massive parallel computation of space-time kernel density. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences II-4/W2:7–11. https://doi.org/10.5194/isprsannals-II-4-W2-7-2015
2. Hohl A, Saule E, Delmelle E, Tang W (2020) Spatiotemporal domain decomposition for high performance computing: a flexible splits heuristic to minimize redundancy. In: Tang W, Wang S (eds) High Performance

Computing for Geospatial Applications, Springer International Publishing, Cham, pp 27–50, https://doi.org/10.1007/978-3-030-47998-5_3
3. Goodchild M (2007) Citizens as sensors: the world of volunteered geography. GeoJournal 69:211–221. https://doi.org/10.1007/s10708-007-9111-y
4. Kwan MP, Neutens T (2014) Space-time research in giscience. Int J Geogr Inf Sci 28(5):851–854. https://doi.org/10.1080/13658816.2014.889300
5. Song W, Wu C (2021) Introduction to advancements of GIS in the new IT era. Ann GIS 27(1):1–4. https://doi.org/10.1080/19475683.2021.1890920
6. Zhu AX, Zhao FH, Liang P, Qin CZ (2021) Next generation of GIS: must be easy. Ann GIS 27(1):71–86. https://doi.org/10.1080/19475683.2020.1766563
7. Miaomiao S, Li W, Zhou B, Lei T (2016) Spatiotemporal data representation and its effect on the performance of spatial analysis in a cyberinfrastructure environment - a case study with raster zonal analysis. Comput Geosci 87:11–21. https://doi.org/10.1016/j.cageo.2015.11.005
8. Stojanovic N, Stojanovic D (2013) High-performance computing in GIS: Techniques and applications. Int J Reason-based Intell Syst 5:42–49. https://doi.org/10.1504/IJRIS.2013.055126
9. Yildirim AA, Watson DW, Tarboton DG, Wallace RM (2015) A virtual tile approach to raster-based calculations of large digital elevation models in a shared-memory system. Comput Geosci 82:78–88. https://doi.org/10.1016/j.cageo.2015.05.014
10. Wilson JP (2018) Environmental applications of digital terrain modelling. John Wiley & Sons. https://doi.org/10.1002/9781118938188
11. Wilson JP (2012) Digital terrain modeling. Geomorphology 137(1):107–121, https://doi.org/10.1016/j.geomorph.2011.03.012. https://www.sciencedirect.com/science/article/pii/S0169555X11001449, geospatial Technologies and Geomorphological Mapping Proceedings of the 41st Annual Binghamton Geomorphology Symposium
12. Xuejun L, Lu B (2008) Accuracy assessment of DEM slope algorithms related to spatial autocorrelation of DEM errors. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 307–322. https://doi.org/10.1007/978-3-540-77800-4_16
13. Nilsson H, Pilesjö P, Hasan A, Persson A (2021) Dynamic spatio-temporal flow modeling with raster DEMS. Transactions in GIS 1572–1588. https://doi.org/10.1111/tgis.12870
14. Lin J, Guan Q, Tian J, Wang Q, Tan Z, Li ZJ, Wang N (2020) Assessing temporal trends of soil erosion and sediment redistribution in the hexi corridor region using the integrated RUSLE-TLSD model. CATENA 195:104756
15. Abudu S, Ping Sheng Z, Liang Cui C, Saydi M, Sabzi HZ, King JP, (2016) Integration of aspect and slope in snowmelt runoff modeling in a mountain watershed. Water Science and Engineering 9(4):265–273. https://doi.org/10.1016/j.wse.2016.07.002. https://www.sciencedirect.com/science/article/pii/S1674237016300436
16. Launiainen S, Guan M, Salmivaara A, Kieloaho AJ (2019) Modeling boreal forest evapotranspiration and water balance at stand and catchment scales: a spatial approach. Hydrol Earth Syst Sci 23:3457–3480. https://doi.org/10.5194/hess-23-3457-2019
17. Svetlitchnyi A, Plotnitskiy S, Stepovaya O (2003) Spatial distribution of soil moisture content within catchments and its modelling on the basis of topographic data. J Hydrol 277(1):50–60. https://doi.org/10.1016/S0022-1694(03)00083-0. https://www.sciencedirect.com/science/article/pii/S0022169403000830
18. Bacon DF, Graham SL, Sharp OJ (1994) Compiler transformations for high-performance computing. ACM Comput Surv 26(4):345–420. https://doi.org/10.1145/197405.197406
19. Kennedy K, McKinley KS (1994) Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Banerjee U, Gelernter D, Nicolau A, Padua D (eds) Languages and Compilers for Parallel Computing. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 301–320
20. Wolf ME, Lam MS (1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distrib Syst 2(4):452–471. https://doi.org/10.1109/71.97902
21. Florinsky I (1998) Accuracy of local topographic variables derived from digital elevation models. Int J Geogr Inf Sci 12:47–61. https://doi.org/10.1080/136588198242003
22. Jones K (1998) A comparison of algorithms used to compute hill slope as a property of the DEM. Comput Geosci 24:315–323. https://doi.org/10.1016/S0098-3004(98)00032-6
23. Warren S, Hohmann M, Auerswald K, Mitasova H (2004) An evaluation of methods to determine slope using digital elevation data. Catena pp 215–233. https://doi.org/10.1016/j.catena.2004.05.001
24. Zhou Q, Liu X (2004) Analysis of errors of derived slope and aspect related to DEM data properties. Comput Geosci 30:369–378. https://doi.org/10.1016/j.cageo.2003.07.005
25. Olaya V (2009) Chapter 6 Basic Land-Surface Parameters. In: Hengl T, Reuter HI (eds) Geomorphometry, Developments in Soil Science, vol 33, Elsevier, pp 141 – 169. https://doi.org/10.1016/S0166-2481(08)00006-8. http://www.sciencedirect.com/science/article/pii/S0166248108000068

26. Tomlin CD (1990) Geographic information systems and cartographic modeling. Englewood Cliffs, N.J. : Prentice Hall
27. Skidmore A (1989) A comparison of techniques for calculating gradient and aspect from a gridded digital elevation model. Int J Geogr Inf Syst 3:323–334. https://doi.org/10.1080/02693798908941519
28. Tang J, Pilesjö P, Persson A (2013) Estimating slope from raster data - a test of eight algorithms at different resolutions in flat and steep terrain. Geod Cartogr 39(2):41–52. https://doi.org/10.3846/20296991.2013.806702
29. Horn BKP (1981) Hill shading and the reflectance map. Proc IEEE 69(1):14–47. https://doi.org/10.1109/PROC.1981.11918
30. Bylina B, Potiopa J, Klisowski M, Bylina J (2021) The impact of vectorization and parallelization of the slope algorithm on performance and energy efficiency on multi-core architecture. In: Ganzha M, Maciaszek LA, Paprzycki M, Slezak D (eds) Proceedings of the 16th Conference on Computer Science and Intelligence Systems, Online, September 2-5, 2021, Annals of Computer Science and Information Systems, vol 25, pp 283–290. https://doi.org/10.15439/2021F68
31. Jeffers J, Reinders J, Sodani A (2016) Intel Xeon Phi Processor High Performance Programming. Morgan Kaufmann
32. Supalov A, Semin A, Dahnken C, Klemm M (2014) Optimizing HPC applications with intel cluster tools. Apress
33. Solon J, et al (2018) Physico-geographical mesoregions of Poland: Verification and adjustment of boundaries on the basis of contemporary spatial data. online http://rcin.org.pl/igipz/Content/65112/WA51_84317_r2018-t91-no2_G-Polonica-Solon.pdf
34. GUGiK (2020) Główny Urząd Geodezji i Kartografii. http://www.gugik.gov.pl/. http://www.gugik.gov.pl/pzgik/zamow-dane/numeryczny-model-terenu
35. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammerling S, McKenney A, et al (1999) LAPACK Users' guide: third edition. Society for Industrial and Applied Mathematics. https://books.google.pl/books?id=AZlvEnr9gCgC
36. GDAL/OGR contributors (2020) GDAL/OGR Geospatial Data Abstraction software Library. Open Source Geospatial Foundation. https://gdal.org
37. Wilkinson B, Allen M (1998) Parallel programming - techniques and applications using networked workstations and parallel computers. Pearson Education
38. Heath MT (2015) A tale of two laws. Int J High Perform Comput Appl 29(3):320–330. https://doi.org/10.1177/1094342015572031
39. GRASS Development Team (2020) Geographic Resources Analysis Support System (GRASS GIS) Software, Version 7.4. Open Source Geospatial Foundation. http://grass.osgeo.org
40. Maguire D (2008) ArcGIS: General purpose GIS software system. In: Shekhar S, Xiong H (eds) Encyclopedia of GIS, Springer, Boston, MA. https://doi.org/10.1007/978-0-387-35973-1_68

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Beata Bylina is a mathematics graduate (1998) and obtained her PhD degree in computer science on Algorithms to solve linear systems with the WZ factorization for Markovian models of computer networks in 2005 at The Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences in Gliwice (Poland) as well as DSc in computer science (Computer aspects of the matrix computations—algorithms and applications, 2020). She has been working in Marie Curie-Skłodowska University in Lublin (Poland) since 1998, now as an assistant professor. Her research interests include numerical methods for linear equation systems and for ordinary differential equations, scientific and parallel computing.

Jarosław Bylina is a mathematics graduate (1998) and a PhD in computer science (Distributed methods to solve Markovian models of computer networks, done at The Silesian University of Technology in Gliwice (Poland) in 2006) as well as DSc in computer science (Techniques for adjusting computational algorithms to contemporary hardware architectures, 2020). He has been working in Marie Curie-Skłodowska University in Lublin (Poland) since 1998, now as an assistant professor. He is interested in numerical methods for Markov chains, modelling of teleinformatic systems, programming languages, as well as parallel and distributed computing and processing.

Łukasz Chabudziński received the MSc and Ph.D. in geography from Maria Curie-Skłodowska University in 2007 and 2016, He worked as a GIS Specialist at ESRI. He started working in the GIS Laboratory at Maria Curie-Sklodowska University in 2008. He is currently an assistant professor and he deals with spatial analysis, field research, and teaching. His research interests include geographic information systems, geographical modelling, especially efficient spatial data processing (both vector and raster) and creating new GIS tools. He is one of the initiators of creating a new field of study in Poland — Geoinformatics. He is also the originator and organizer of the Academic Geoinformatics Championships — GIS Challenge.

Karol Karpowicz received the B.Eng. in Technical Physics from the Faculty of Mathematics, Physics and Computer Science of the Maria Curie Skłodowska University in 2016. Now he works as an administrator of computing clusters at the same Faculty. His interests include electronics and computer systems architectures.

Michał Klisowski is a mathematics graduate (1998) and a PhD in Computer Science (2015). He is a teaching and research assistant at Institute of Computer Science, Maria Curie-Skłodowska University in Lublin (Poland). His current research interests include high-performance computing and geospatial algorithms.

Piotr Oleszczuk is an assistant professor at the Department of Quantitative Methods in Management of the Faculty of Management, Lublin University of Technology. He holds a Ph.D. in Mathematics from the Faculty of Mathematics, Physics and Computer Science of the Maria Curie Skłodowska University. His current research interests include inter-disciplinary applications of mathematics, data science, high-performance computing and cloud computing.

Joanna Potiopa is a mathematic graduate (2003) and received PhD degree in mathematics from Maria Curie-Sklodowska University in 2015 (Algorithms for solving selected types of narrow banded linear systems on computers with multi-core processors). She has been working in Marie Curie-Skłodowska University in Lublin (Poland) since 2003, now as an assistant professor. Her research interests include numerical methods for linear equation systems, databases, parallel computing, development and performance evaluation of algorithms for modern parallel architectures and spatial data processing.

Przemysław Stpiczyński is an associate professor at Institute of Computer Science, Maria Curie-Skłodowska University in Lublin, Poland. He received his Ph.D. degree in Computational Mathematics in 1995 and DSc degree in Computer Science in 2010. He has published over 70 research papers. His main interests are parallel and distributed computing, parallel programming, numerical analysis. He is a program committee member of several scientific conferences like Parallel Processing and Applied Mathematics, Computer Aspects of Numerical Algorithms and High Performance Computing and Simulations.

Authors and Affiliations

Beata Bylina[1] · Jarosław Bylina[1] · Łukasz Chabudziński[1] · Karol Karpowicz[1] · Michał Klisowski[1] · Piotr Oleszczuk[2] · Joanna Potiopa[1] · Przemysław Stpiczyński[1]

Beata Bylina
beata.bylina@umcs.pl

Jarosław Bylina
jaroslaw.bylina@umcs.pl

Łukasz Chabudziński
lchabudzinski@poczta.umcs.lublin.pl

Karol Karpowicz
karol.karpowicz@poczta.umcs.lublin.pl

Piotr Oleszczuk
p.oleszczuk@pollub.pl

Joanna Potiopa
joanna.potiopa@mail.umcs.pl

Przemysław Stpiczyński
przemyslaw.stpiczynski@umcs.lublin.pl

1 Maria Curie-Sklodowska University, Lublin, Poland

2 Lublin University of Technology, Lublin, Poland

GeoInformatica is a copyright of Springer, 2024. All Rights Reserved.