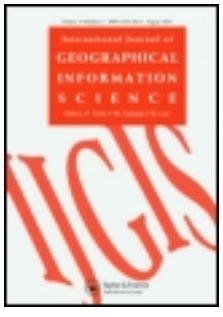

International Journal of Geographical Information Science

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tgis20

A strategy for raster-based geocomputation under different parallel computing platforms

Cheng-Zhi Qin[a], Li-Jun Zhan[a], A-Xing Zhu[ab] & Cheng-Hu Zhou[a]

[a] State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing 100101, PR China

[b] Department of Geography, University of Wisconsin Madison, Madison, WI 53706-1491, USA

Published online: 24 Apr 2014.

To cite this article: Cheng-Zhi Qin, Li-Jun Zhan, A-Xing Zhu & Cheng-Hu Zhou (2014): A strategy for raster-based geocomputation under different parallel computing platforms, International Journal of Geographical Information Science, DOI: 10.1080/13658816.2014.911300

To link to this article: http://dx.doi.org/10.1080/13658816.2014.911300

International Journal of Geographical Information Science, 2014
http://dx.doi.org/10.1080/13658816.2014.911300

A strategy for raster-based geocomputation under different parallel computing platforms

Cheng-Zhi Qin[a]*, Li-Jun Zhan[a], A-Xing Zhu[a,b] and Cheng-Hu Zhou[a]

[a]*State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing 100101, PR China;* [b]*Department of Geography, University of Wisconsin Madison, Madison, WI 53706-1491, USA*

(*Received 9 December 2013; accepted 31 March 2014*)

The demand for parallel geocomputation based on raster data is constantly increasing with the increase of the volume of raster data for applications and the complexity of geocomputation processing. The difficulty of parallel programming and the poor portability of parallel programs between different parallel computing platforms greatly limit the development and application of parallel raster-based geocomputation algorithms. A strategy that hides the parallel details from the developer of raster-based geocomputation algorithms provides a promising way towards solving this problem. However, existing parallel raster-based libraries cannot solve the problem of the poor portability of parallel programs. This paper presents such a strategy to overcome the poor portability, along with a set of parallel raster-based geocomputation operators (PaRGO) designed and implemented under this strategy. The developed operators are compatible with three popular types of parallel computing platforms: graphics processing unit supported by compute unified device architecture, Beowulf cluster supported by message passing interface (MPI), and symmetrical multiprocessing cluster supported by MPI and open multiprocessing, which make the details of the parallel programming and the parallel hardware architecture transparent to users. By using PaRGO in a style similar to sequential program coding, geocomputation developers can quickly develop parallel raster-based geocomputation algorithms compatible with three popular parallel computing platforms. Practical applications in implementing two algorithms for digital terrain analysis show the effectiveness of PaRGO.

Keywords: geocomputation; raster; parallel computing; parallel operator; graphics processing unit (GPU); cluster; compute unified device architecture (CUDA); message passing interface (MPI); open multiprocessing (OpenMP)

1. Introduction

As one of the most common geographic data types, raster is widely used in various geocomputation domains, such as digital terrain analysis (DTA), distributed hydrological modelling and remote sensing image analysis (Duckham *et al.* 2003). Raster-based geocomputation is traditionally coded as sequential algorithms by geocomputation developers. Currently, there is ever-growing demand for parallel raster-based geocomputation because of the rapid growth not only in the volume of raster data for geocomputation but also in the complexity of raster-based geocomputation (Healey *et al.* 1998, Armstrong 2000, Clarke 2003).

*Corresponding author. Email: qincz@lreis.ac.cn

Much research effort has been put into the development of parallel programming of raster-based geocomputation (e.g., Armstrong and Marciano 1996, Huang *et al.* 2011, Tesfa *et al.* 2011, Qin and Zhan 2012) due to the higher availability and the lower operational costs of various parallel computing platforms (Zhang 2010). However, parallel programming has not been an easy task for most geocomputation developers who are familiar with sequential (or serial) programming and are really interested in the development of geocomputation method not in the parallel implementation of these methods. For novices, parallel programming is a long process with a steep learning curve because parallel programming introduces many complicated details (such as communication, synchronization, and load balancing), which are not encountered in sequential programming (Wang and Armstrong 2009). It is still difficult, even for those with good experience in parallel programming, to code an efficient and scalable parallel geocomputation program. Furthermore, it is very often that each parallel geocomputation program is for one type of parallel computing platform (e.g., graphics processing unit (GPU), Beowulf cluster, and symmetrical multiprocessing (SMP) cluster). Different parallel computing platforms often have neither unified parallel hardware model nor a unified parallel programming model. This means that a parallel geocomputation program coded for a specific parallel computing platform often has limited portability to other parallel computing platforms.

All this raises the question as to whether an efficient way can be devised to code a raster-based geocomputation method only once as a sequential program then compile multiple parallel versions (one for each type of parallel computing platform). Researchers have explored this possibility by encapsulating the parallel programming details of common raster-based operations in a parallel raster-based programming library (or, parallel raster-based geocomputation operators (PaRGO)). The geocomputation developers can then use the library to code a parallel program for a specific parallel computing platform in a way that resembles sequential programming.

Parallel raster-based programming libraries of this type currently include Parallel Utilities Library (PUL) (Bruce *et al.* 1995), parallel raster-based Neighbourhood Modelling (NEMO) (Hutchinson *et al*. 1996), Global Arrays (GA) (Nieplocha *et al.* 2006), Parallel Raster Processing Programming Library (pRPL) (Guan and Clarke 2010), and Image Processing Framework (IPF) (Membarth *et al.* 2011). There is a trade-off between transparency and versatility. The more parallel details these libraries hide, the more parallel ability they lose. IPF and NEMO achieve high transparency by hiding the parallel details well. Users without parallel programming knowledge can use them to write parallel codes. However, IPF and NEMO only support local and focal (or neighbourhood) operations, but neither IPF nor NEMO support regional (zonal) or global operations. PUL and GA hide only some parallel details, but this strategy exposes a lot of complex parallel programming library interfaces. In such a way, PUL and GA can be used for parallelization of all kinds of raster-based operations (as long as an operation is parallelizable) by users who have some parallel raster-based programming experience. Compared with the other existing parallel raster-based programming libraries, pRPL achieves a good balance between transparency and versatility by hiding most of the parallel details, leaving only some basic parallel details (such as determining the master node, etc.) to be assigned by users. Thus, pRPL can support not only local and focal operations, but also some regional and global raster operations.

The main shortcoming of existing parallel raster-based programming libraries is their poor portability, that is, each of them suits only one specific parallel computing platform. pRPL, PUL, NEMO, and GA encapsulate a message passing library, and thus are suitable

for Beowulf cluster. They are not available for GPU and also might not achieve high efficiency on SMP cluster. IPF, which is based on CUDA, applies only to GPU. Furthermore, existing parallel raster-based programming libraries often support only a very few file formats of raster data. This situation also limits the practical applicability of these libraries.

This paper presents a strategy to address the portability problem in existing parallel raster-based programming libraries. This strategy is then illustrated through the design and implementation of a set of PaRGO, which are compatible with different parallel computing platforms. Specifically, PaRGO has the following characteristics: (1) it hides the parallel details as much as possible; (2) it supports local, focal, and global raster operations; (3) it is compatible with common parallel computing platforms (i.e., Beowulf cluster, SMP cluster, and GPU); and (4) it supports a variety of commonly used raster data formats.

2. The strategy and the development of PaRGO

2.1. Basic idea and overall design

In order to design PaRGO, firstly the sequential raster-based geocomputation algorithm was compared with the corresponding parallel algorithm. Most sequential raster geocomputation algorithms include the following five steps (Figure 1a):

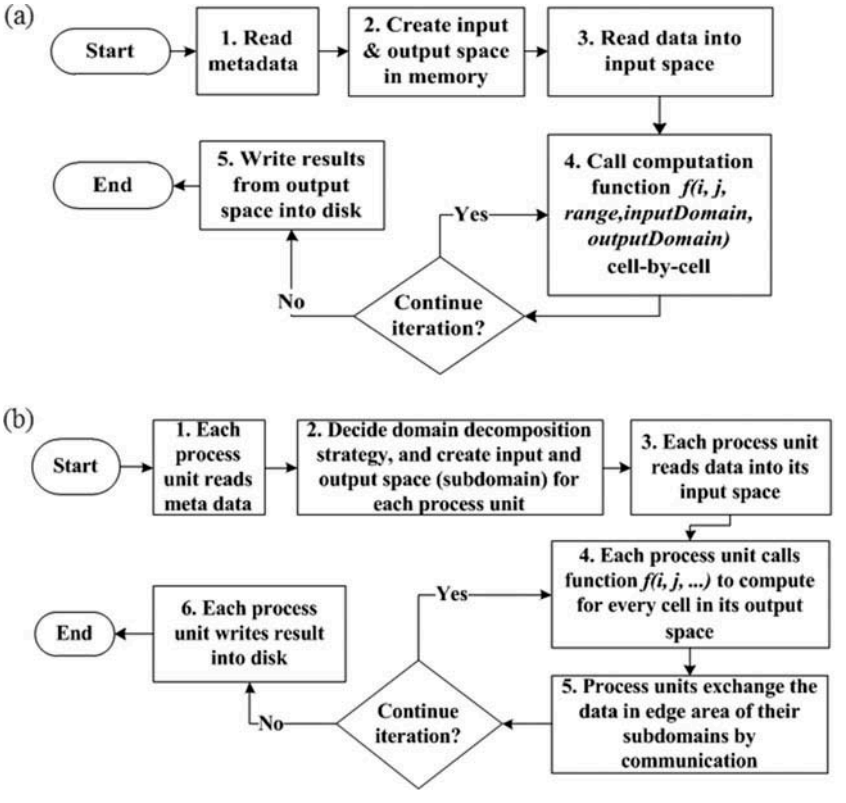

Figure 1. Flowchart of (a) general sequential algorithm and (b) general parallel algorithm of raster-based geocomputation.

(1) Read metadata (e.g., spatial extent, and projection) from the raster file.
(2) Create input and output domains in memory according to the spatial extent information.
(3) Read raster data from the external memory to input domain.
(4) Compute in one round or more iterative rounds of processing. In each iteration, the computation function *f*(*i*, *j*, *range*, *inputDomain*, *outputDomain*) is performed for each cell. The parameter *range* refers to the scope of raster needed for the computation of a cell. Based on *range*, computation function *f*(*i*, *j*, *range*, *inputDomain*, *outputDomain*) reads data from *inputDomain* for computation of each cell and then puts the computation result into *outputDomain*.
(5) Write the result *outputDomain* from memory to a raster file in the external memory.

Compared with the general sequential algorithm of raster-based geocomputation, a general parallel algorithm of raster-based geocomputation (Figure 1b) involves a large number of parallel programming details. These can be divided into three categories: (1) domain decomposition, (2) communication, and (3) input/output (I/O). If the above three kinds of parallel programming details could be encapsulated and hidden for the user, geocomputation developers need only to implement the computation function *f*(*i*, *j*, *range*, *inputDomain*, *outputDomain*) in sequential programming style to code parallel algorithms. This will relieve the developers of the burden in dealing with the parallel programming details and allow them to focus on the geocomputation itself.

According to above basic idea, a set of PaRGO was designed. PaRGO consists of three core modules to hide three kinds of parallel programming details for users: the domain decomposition module, the communication module, and the I/O module. A highlight that makes PaRGO superior to existing parallel raster-based programming libraries, is that it is designed to include different versions to support the following common parallel computing platforms (Lin and Snyder 2008):

(1) *Shared memory parallel machine*, in which each processor shares the global memory by system bus (Figure 2a). These mainly include symmetric multiprocessor (SMP) and coprocessor (e.g., GPU) devices. A multithreading programming model (Dongarra *et al.* 2005) is normally applied to shared memory parallel machines. The widely used multithreading programming models include open multiprocessing (OpenMP) for SMP parallel computing devices (e.g., multiprocessors in personal computers) and compute unified device architecture (CUDA) for GPU devices.
(2) *Distributed memory parallel machine*, in which each processor has its own memory and interacts with other processors by communication network (Figure 2b), is represented by Beowulf cluster. A message passing parallel programming model is normally used on distributed memory parallel machines. Currently, the most widely used message passing programming model is message passing interface (MPI).
(3) *Hybrid distributed-shared memory parallel machine*, which can be seen as a combination of above two types, has an overall structure of distributed memory and consists of many shared memory components (Figure 2c). SMP cluster is representative of this type of parallel machine. MPI/OpenMP hybrid programming is popular on SMP cluster.

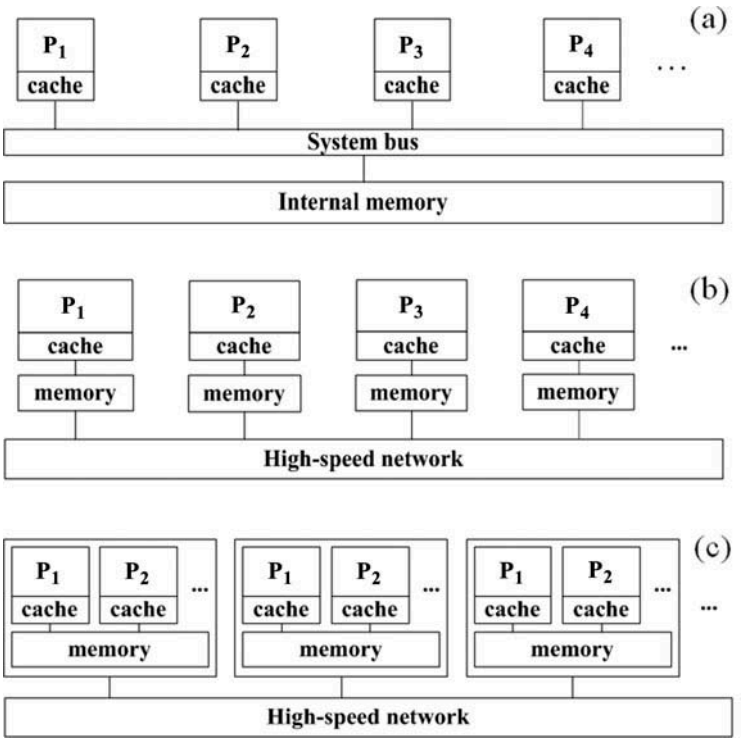

Figure 2. Three types of modern parallel computing machines: (a) shared memory parallel machine; (b) distributed memory parallel machine; and (c) hybrid distributed-shared memory parallel machine.

In this study, PaRGO was designed to provide three versions with a unified user interface, namely, a CUDA version, an MPI version, and an MPI/OpenMP version. In such a way PaRGO can aid the geocomputation developers towards coding once in a sequential programming style and then compiling parallel programs for GPU, Beowulf cluster, and SMP clusters. From the perspective of software architecture, PaRGO serves as a middleware that connects parallel programming libraries specific to different parallel computing platforms with raster-based geocomputation algorithms (Figure 3).

2.2. Design of domain decomposition module

2.2.1. Domain decomposition

Parallel algorithms of raster-based geocomputation commonly decompose the data domain into rectangular subdomains using any of three straightforward strategies of domain decomposition (i.e., row-wise, column-wise, and block-wise). Each subdomain is processed by a process unit when two minimum bounding rectangles (MBRs) are assigned to describe the subdomain for the computation task on this process unit. The first MBR (so-called 'task MBR') is used to indicate the range of the result after the computation task is completed by the corresponding process unit. The second MBR ('data MBR') is used to indicate the scope of data required for the computation task of the process unit.

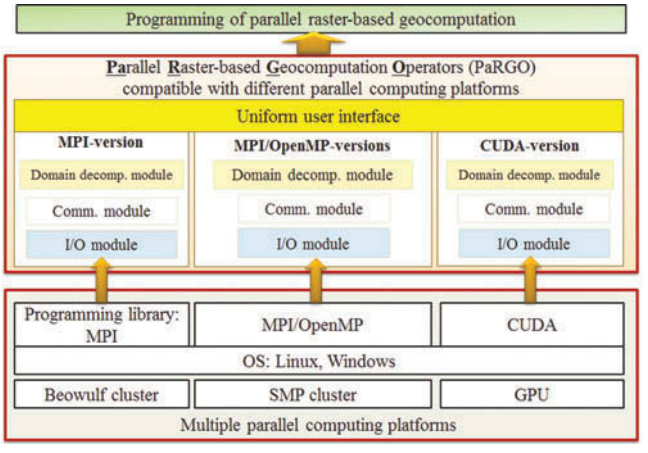

Figure 3. Architecture of parallel raster-based geocomputation operators (PaRGO).

In parallel raster-based geocomputation, the core of domain decomposition is to assign these two MBRs for each process units. Task MBR is determined by the spatial extent of input data, the number of process units, and the domain decomposition strategy. Data MBR is determined by not only the corresponding task MBR but also operations in the raster computation task, which is more complicated. For local operation, data MBR overlaps with the corresponding task MBR. For focal operation in which the computation for a given cell requires the values of its neighbouring cells, data MBR contains not only the task MBR but also the neighbouring area of the task MBR. For global operation, in which the computation for a given cell requires all cells in the input domain, data MBR constitutes the whole input domain. The data MBR for regional operation is hard to predefine, making the parallelization of the geocomputation algorithms involved in regional operation the most difficult, and usually requires the developer to be familiar with parallel details. Because of this, the current design of PaRGO does not support parallelization of regional operation, which is similar to the situation with existing parallel raster-based programming libraries. When the number of process units is assigned by user before the execution of parallel raster-based geocomputation, each process unit will be accordingly assigned its task MBR and data MBR in a static decomposition way, which means that both MBRs of each process unit will be unchanged during the execution.

2.2.2. *Workflow in domain decomposition module*

The domain decomposition modules in the platform-specific versions of PaRGO have different workflows to determine the two MBRs, especially the data MBR.

(1) In the CUDA version, the thread is the basic process unit of CUDA, and all threads share the same address space. In this situation, the master thread first creates a shared array with the same size as the input data domain in memory. Then each thread directly maps to the shared array according to the corresponding data MBR (Figure 4a).

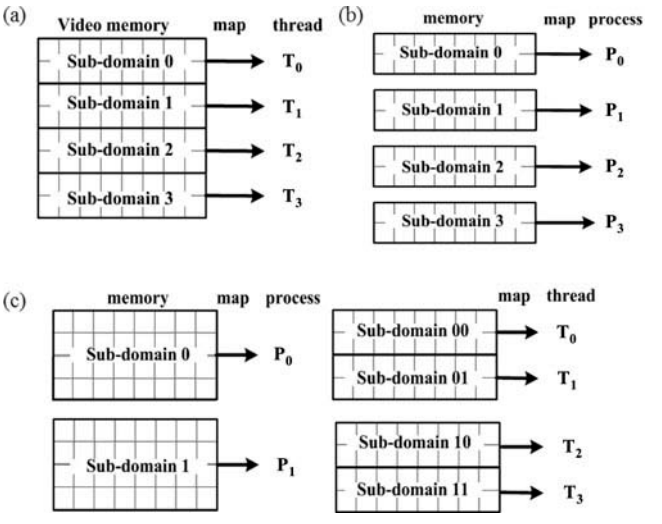

Figure 4. Workflow of (a) CUDA version, (b) MPI version, and (c) MPI/OpenMP version of the domain decomposition module, exemplified by a row-wise domain decomposition case with four process units.

(2) In the MPI version of the domain decomposition module, the process is the basic process unit of MPI, and each process has a separate address space. Thus, each process needs to create its own subdomain in memory according the size of its data MBR (Figure 4b).
(3) In MPI/OpenMP version of the domain decomposition module, the workflow consists of two steps. The first step is the same as the workflow of the MPI version. In the second step, each thread directly maps to the shared array according to the corresponding data MBR, which is similar to the workflow of CUDA version (Figure 4c).

2.3. *Design of communication module*

2.3.1. *Communication modes*

There are two basic modes of communication in parallel programming, that is, implicit communication used for multithreading programming models (e.g., CUDA and OpenMP), and explicit communication used for the message passing parallel programming model (e.g., MPI). In the implicit communication mode, when a thread modifies the value of cell in its task MBR, other threads whose data MBR contains this cell can simultaneously obtain the modified value on this cell because all threads share a global array. Therefore, the communication within CUDA is transparent to users.

In explicit communication mode, because each process of MPI has a separate address space, modification on a cell by one process cannot affect other processes whose data MBR contains these cells before explicitly message passing codes are executed to allow these processes to communicate with each other. MPI/OpenMP hybrid programming involves both communication modes. In this situation, the communication within the MPI part is explicit when the communication within OpenMP part is implicit and transparent to users.

2.3.2. *MPI-based communication workflow in communication module*

The communication module of PaRGO needs to encapsulate the explicit communication process related to MPI and so to make it transparent to users. For this purpose, the communication module is designed to work in three steps:

Step 1. Specify the sending–receiving relationship between processes. At the beginning of the communication process with MPI, in the sending process it must be specified which are the receiving processes. Each receiving process must also specify the sending process(es) from which they will receive messages. Inside the communication module this is determined by the domain decomposition strategy used. For example, Figure 5a shows under the row-wise (or column-wise) domain decomposition strategy with n processes, the processes P_0 and P_{n-1} communicate, respectively, with the processes P_1 and P_{n-2} when the process P_i communicates with the two processes (P_{i-1} and P_{i+1}), which use the neighbouring subdomains of the P_i's subdomain. The situation for a block-wise domain decomposition strategy is similar (Figure 5b).

Step 2. Determine the data-sending range and the data-receiving range for each process. The data-sending range of a sending process to one of its receiving processes comprises the cells within both the task MBR of the sending process and the data MBR of the receiving process. This data-sending range is exactly the data-receiving range of the receiving process from this sending process. Note that for local operation both the data-sending range and the data-receiving range for each process are empty.

Step 3. Communicate among processes at the end of each iterative round of processing. MPI provides different types of sending and receiving routines, such as standard type, synchronous type, ready type, buffered type, and so on. Each type is for a different purpose. For raster-based geocomputation, the message size needed for communication is often greater than the threshold of the system buffer. This may cause a bottleneck because the send process must wait when there is insufficient buffer for the receive process. In order to avoid this problem, the communication module of PaRGO adopts buffered communication that automatically sets the buffer size according to the size of the message.

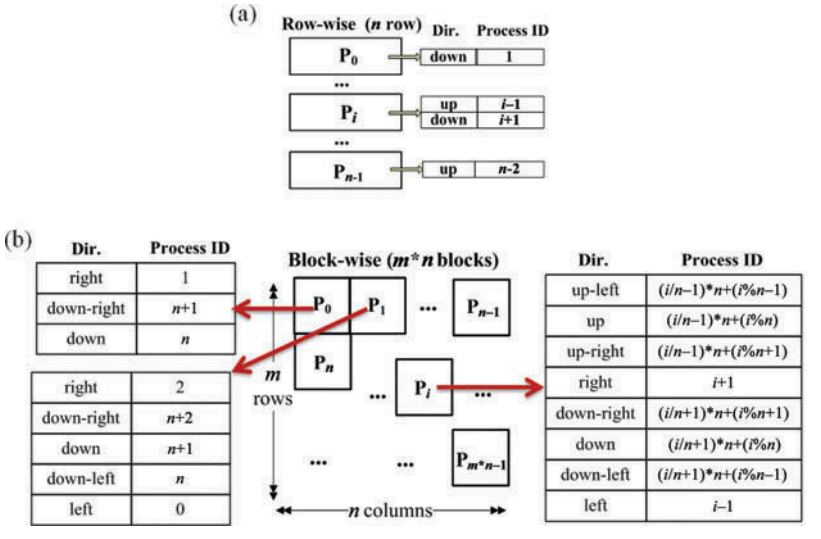

Figure 5. The receiving process IDs of each sending process: (a) for the row-wise domain decomposition strategy; (b) for the block-wise domain decomposition strategy.

2.4. Design of I/O module

I/O module of PaRGO is based on the open-source geospatial data abstraction library (GDAL[1]), which provides a single abstract data model to read and write a variety of spatial raster data formats. GDAL has been widely used to support a variety of commonly used geospatial raster data formats in sequential geocomputation (Warmerdam 2008). In the I/O module of PaRGO, the parallel I/O mode of using GDAL is explored to improve the I/O efficiency of parallel raster geocomputation.

Because the MPI version and the MPI/OpenMP version of a parallel algorithm both have the same I/O, the MPI version supports the I/O needs of both versions of PaRGO. The CUDA-version I/O module also involves data-transfer between graphics memory in GPU and internal memory. In this section, the designs of the MPI-version and CUDA-version I/O modules of PaRGO are described.

2.4.1. MPI-version I/O module

The function of the MPI-version I/O module is to load raster data stored in external memory into internal memory for each process according to its data MBR, and later to write the raster data of the computation result within the task MBR from internal memory to external memory for each process.

A sequential I/O mode using GDAL to access raster data has been applied to parallel raster-based geocomputation in some recent studies (e.g., Wang *et al.* 2012). In the sequential I/O mode, a master process takes charge of the whole I/O process between internal and external memory when other processes access their data subdomains through communication with the master process. This mode has not only the single bottleneck problem when the size of raster data file exceeds the memory of the master node, but also the overheads of data distribution between the master process and the work processes.

The MPI-version I/O module of PaRGO adopts a parallel I/O mode of using GDAL proposed by Qin *et al.* (2013) (Figure 6). Note that the direct application of GDAL with the parallel I/O mode is highly inefficient and does not even achieve correct output for column-wise or block-wise domain decomposition (Qin *et al.* 2013). Qin *et al.* (2013) analysed the reasons for this problem and proposed an MPI-version data redistribution module solution based on a two-phase I/O strategy. In the parallel I/O mode with the aid of data redistribution module, each process uses GDAL to directly read its data subdomain stored in external memory. After computation, each process uses GDAL to open the shared output raster file in external memory and to write the result data in it (Figure 6). Thus the shortcomings of the sequential I/O mode are avoided and the I/O efficiency of parallel raster geocomputation is improved. The specifics of both the problem and its solution are not included here due to length limitations; interested readers are referred to Qin *et al.* (2013) for details. By this means, the MPI-version I/O module of PaRGO hides the parallel I/O details for users when the parallel I/O can be supported for a variety of commonly used geospatial raster data formats.

2.4.2. CUDA-version I/O module

Compared with the MPI-version I/O module, the CUDA-version I/O module of PaRGO takes charge of not only transferring data between internal and external memory by GDAL, but also transferring data between the internal memory and the graphics memory in GPU. The read operation implemented in the CUDA-version I/O module is in two steps

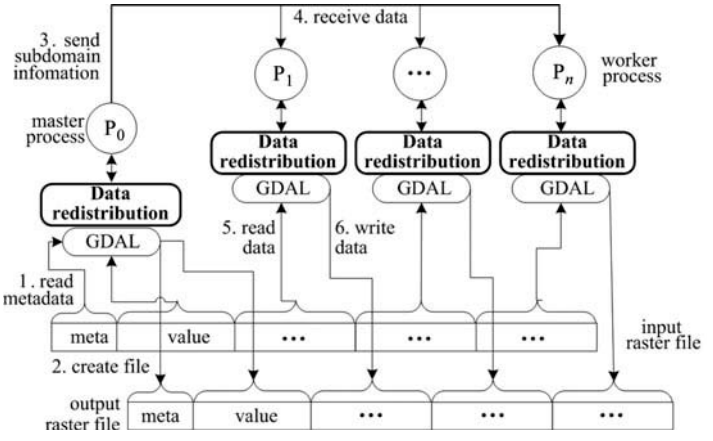

Figure 6. Parallel raster I/O mode of using GDAL with a data redistribution module for parallel geospatial processing (Qin *et al.* 2013).

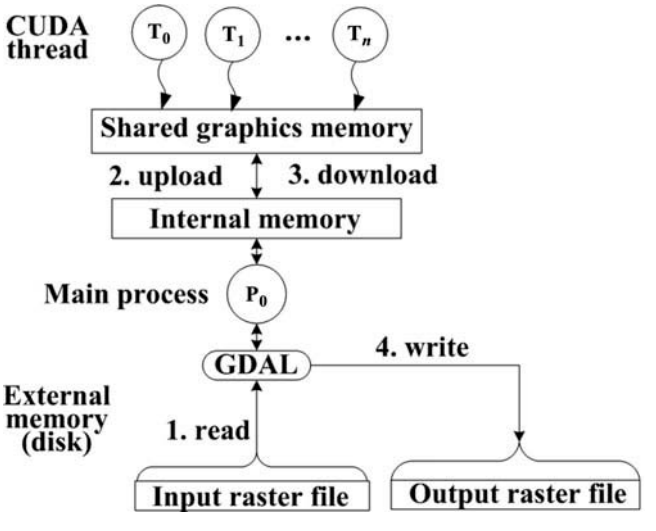

Figure 7. Workflow in CUDA-version I/O module of PaRGO.

(Figure 7): (1) load data from external memory into internal memory; and (2) upload data from internal memory to graphics memory. The write operation implemented in the CUDA-version I/O module is also in two steps (Figure 7): (1) transfer data from graphics memory to internal memory; and (2) write data from internal memory to external memory.

3. Implementation of PaRGO

The MPI, MPI/OpenMP, and CUDA versions of PaRGO were developed in *C++* programming language and are implemented as a template programming library. In this way PaRGO supports all commonly used types of cell attribute values in raster-based geocomputation. Further, PaRGO is implemented as an object-oriented programming library that comprises kernel classes and user classes (Figure 8).

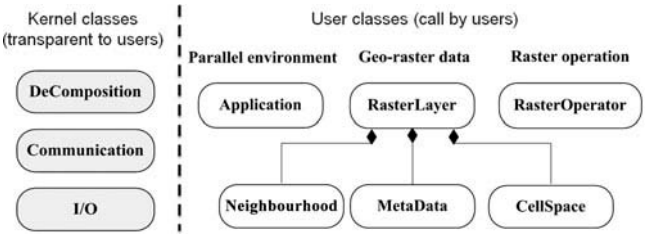

Figure 8. Kernel classes and user classes in PaRGO.

3.1. *Kernel classes*

The kernel classes of PaRGO include *DeComposition*, *Communication*, and *I/O* classes (Figure 8), which implement the modules for domain decomposition, communication, and I/O, respectively. The design of PaRGO encapsulates these three categories of parallel programming detail. The *DeComposition* class supports the three straightforward domain decomposition strategies, that is, row-wise, column-wise, and block-wise. The *Communication* class provides a buffered communication mode. The MPI-version *I/O* class provides parallel I/O for commonly used geospatial raster data formats by using GDAL with a data redistribution module under the parallel I/O mode. Kernel classes support user classes and are transparent to users.

3.2. *User classes*

User classes, with a unified user interface for the three versions of PaRGO, are called by geocomputation developers. This enables the developers to code in a sequential object-oriented programming style but achieve the parallel algorithms compatible with the three commonly used parallel computing platforms.

User classes include the parallel environment class, the geo-raster data classes, and the raster operation class (Figure 8).

3.2.1. *Parallel environments class*

The parallel environments class (*Application*) is mainly used to initialize the parallel environment at the start of a parallel program and to release parallel resources at the end of the parallel program.

3.2.2. *Geo-raster data classes*

The geo-raster data classes comprise the *Neighborhood*, *MetaData*, *CellSpace*, and *RasterLayer* classes. The *Neighborhood* class defines the range used for a raster operation on a cell. Arbitrary neighbourhood configuration is supported. The *MetaData* class records two types of metadata of geo-raster data: (1) the global metadata, such as the original extent of raster data, NoData, projection, etc.; and (2) task MBR and data MBR. The *CellSpace* class stores subdomain values for each process unit. The *RasterLayer* class is called by users to access a raster layer defined by the *Neighborhood*, *MetaData*, and *CellSpace* classes.

3.2.3. Raster operation class

PaRGO provides a base class, *RasterOperator*, for users. By writing customized *RasterOperator* classes derived from this base class, users can implement their own raster geocomputation operation, as they do in sequential object-oriented programming.

4. Application

4.1. Coding parallel raster-based geocomputation algorithm using PaRGO

This section describes the application of PaRGO to the parallel programming of two algorithm cases in DTA based on the gridded digital elevation model (DEM), which is a typical domain of raster-based geocomputation, to evaluate the effectiveness of the proposed PaRGO. The algorithms used in this study are (1) a slope gradient algorithm, representing a simple algorithm, and (2) a DEM-preprocessing algorithm, representing a more complex DTA algorithm.

4.1.1. Algorithm case 1: coding a parallel slope gradient algorithm using PaRGO

Slope gradient calculation is a typical neighbourhood computation algorithm. Without loss of generality, a third-order finite difference method (Horn 1981) commonly used to calculate slope gradient (Skidmore 1989) was selected for this case study.

Because parallel programming details have been encapsulated in PaRGO and are transparent to the user, the slope gradient algorithm can be parallelized in a simple sequential object-oriented programming way using PaRGO. The codes of the parallel slope gradient algorithm with PaRGO are in two parts: the first part is to implement customized class *SlopeOperator* derived from the base class *RasterOperator* by overriding the computation function *Operator* (Code 1); the second part is a main function that simply calls the methods of the user classes in PaRGO (Code 2).

Code 1. The customized class *SlopeOperator* of the parallel slope gradient algorithm using PaRGO.

```
class SlopeOperator : public RasterOperator<double>
{
    public:
       SlopeOperator()
          :RasterOperator<double>(),
           inputLayer (NULL), outputLayer (NULL) {}
       ~SlopeOperator() {}
       void demlayerLoad(RasterLayer<double> &layer); //load input raster layer
       void slopelayerLoad(RasterLayer<double> &layer); //load output raster layer
       virtual bool Operator(int i, int j); //operator function for geocomputation
    protected:
       RasterLayer<double> * inputLayer;   // input raster layer
       RasterLayer<double> * outputLayer; // output raster layer
};
// load input gridded DEM
void SlopeOperator::demlayerLoad(RasterLayer<double> &layer)
{
    inputLayer = &layer;
}
```

```
//load slope gradient raster
void SlopeOperator::slopelayerLoad(RasterLayer<double> &layer)
{
    outputLayer = &layer;
}
//operator function for computing slope gradient
bool SlopeOperator::Operator(int i, int j)
{
        CellSpace<double> &dem = *(inputLayer->cellSpace());//input DEM data
        CellSpace<double> &slope = *( outputLayer->cellSpace());//output raster data
        Neighborhood<double>& nb = *( inputLayer->neighborhood()); //assign the
neighborhood window
        MetaData &meta = *(inputLayer->metaData());    //metadata of raster layer
        int cellSize = meta.cellSize; //cell size of DEM
        double* d = new double[nb.size()];    //record the elevation values in the
neighborhood window
        for(int k = 0; k < nb.size(); k++)
        {
            int iRow = i + nb[k].row();
            int iCol = j + nb[k].col();
            d[k++] = dem[iRow][iCol];
        }
        //calculate slope gradient by the third-order finite difference method
        double dx = (d[8] + 2*d[5] + d[2] - d[6] -2*d[3] - d[0])/(8.0* cellSize);
        double dy = (d[2] + 2*d[1] + d[0] - d[6] -2*d[7] - d[8])/(8.0* cellSize);
        slope[i][j] = sqrt(dx*dx + dy*dy); //write results in the CellSpace of output raster
        delete []d;
        return true;
}
```

4.1.2. *Algorithm case 2: coding a parallel DEM-preprocessing algorithm using PaRGO*

DEM-preprocessing is used to remove depressions and flat areas in an original DEM, which is necessary to reasonably calculate flow accumulation and other important regional topographic attributes on DEM (Hengl and Reuter 2009). The DEM-preprocessing algorithm proposed by Planchon and Darboux (2001) (or P&D algorithm) takes an iterative approach to revising the elevation of cells in depressions and flat areas in the original DEM with a user-specified, very small step value of elevation-revision. The P&D algorithm is thought to be suitable for calculating flow accumulation at a finer scale (Qin *et al.* 2006). In this study, we selected the P&D algorithm as the second algorithm case for parallelization by PaRGO.

The P&D algorithm proceeds in two steps: the first step, the water-covering step, is a local operation that floods the entire original DEM to a sufficiently high water level, except at the boundary of the DEM; the second step is the water-removal step, which iteratively drains the excess water off to ensure that for each cell there is a path leading to the boundary (Planchon and Darboux 2001). In the water-removal step, each iteration is a

Code 2. Main function of the parallel slope gradient algorithm using PaRGO.

```
int main(int argc, char *argv[])
{
/* enum ProgramType{MPI_Type = 0, MPI_OpenMP_Type, CUDA_Type, Serial_Type};*/
    Application::START(MPI_Type, argc, argv); //assign parallel environment to be MPI-based, and initialize the parallel environment
    RasterLayer<double> inputLayer;    //create input raster layer
    RasterLayer<double> outputLayer ; //create output raster layer
    inputLayer.readNeighborhood(neighborfile);  //assign the neighborhood window by reading a config file (3×3 window for this case of slope gradient calculation. Arbitrary size of neighborhood window can be assigned by the config file.)
    inputLayer.readFile(inputfilename);          //open input raster data file, and automatically make domain decomposition
    outputLayer.copyLayerInfo(inputLayer);       //automatically make domain decomposition for output raster data
    SlopeOperator SlpOper;   //create an object of the operator of slope gradient calculation
    SlpOper.demlayerLoad(demLayer); //load the input DEM data
    SlpOper.slopelayerLoad (slopeLayer); //load the output layer of slope gradient
    SlpOper.Run(); //execute Operator-method for every cell in task MBR of each process unit
    outputLayer.writeFile(outputfilename); //write the results in output raster file
    Application::END(); //clean the parallel computing resources
    return 0;
}
```

focal operation. This iterative procedure is compute-intensive and needs to be parallelized (Qin and Zhan 2012).

When using PaRGO, the multistep and iterative P&D algorithm is coded as parallel algorithms in a sequential object-oriented programming style. The codes of the parallel P&D algorithm in PaRGO comprise three parts. The first two parts are the customized classes derived from the base class *RasterOperator* for the water-covering step and the water-removal step. The third part is the main function. Due to length limitations, the code file of the P&D algorithm using PaRGO is provided as supplementary materials of this paper.

4.2. *Experimental design*

In order to evaluate the portability and efficiency of PaRGO, the two parallel algorithms using PaRGO were compiled into MPI-version programs (called MPI_Slope and MPI_DEMPreprocessing) and MPI/OpenMP-version programs (called MPIOMP_Slope and MPIOMP_DEMPreprocessing) on an SMP cluster, and were also compiled into CUDA-version programs (called CUDA_Slope and CUDA_DEMPreprocessing) on a GPU server. For comparison, the corresponding sequential algorithms were compiled into the sequential programs (called Serial_Slope and Serial_DEMPreprocessing).

The MPI version, the MPI/OpenMP version, and the above sequential programs were tested on an IBM SMP cluster with 134 compute nodes. Each node consists of two Intel Xeon (E5650 2.0 GHz) 6-core CPUs and 24 GB DDRIII memory. Compute nodes share the disk of the I/O node through Infiniband networks. CUDA-version programs were tested on a NVIDIA Corporation Tesla M2075 (GPU) server that consists of 448 cores and 6 GB graphics memory. The software environment included RedHat Enterprise Linux Server 6.2 as the operating system, *g++* 4.4.6, OpenMP 3.0, MPICH2 (version 1.3.1), CUDA 4.2, and GDAL 1.9.1. The test raster data was a gridded DEM with a dimension of 11,130 × 9320 cells, which is in GeoTiff format.

Efficiency was evaluated by the runtimes and speedup ratios of parallel programs. Here the runtime of each tested program was the execution time of the tested program, not including the time needed for I/O between internal and external memory. For CUDA-version programs, the runtime of the computation part also counted the time needed for transferring data between GPU and CPU. MPI-version programs were tested with one compute node, and with 16 compute nodes (1, 2, 4, and 8 processes for each compute node), respectively. Correspondingly, the MPI/OpenMP-version programs were tested with one compute node, and with 16 compute nodes (1 process × 1/2/4/8 threads for each compute node), respectively.

Note that the P&D algorithm needs a user-specified parameter, that is, the step value of the elevation-revision. In this test, an extremely small value (0.00005 m) was used as this step value in all versions of the tested P&D programs to simulate an extreme case of compute-intensive geocomputation. A larger step value would reduce the runtimes of every tested version of the P&D algorithm; however, the speedup ratios of each tested P&D program should not show an obvious change.

4.3. *Experimental results*

Runtimes of tested programs for slope gradient calculation (Table 1) show that the runtime of CUDA_Slope was dramatically faster than for the sequential slope gradient calculation program (Serial_Slope). MPI_Slope and MPIOMP_Slope executed with only one process/thread need almost same runtime as Serial_Slope. When more computing resources were used and the count of processes/threads increased, the runtimes of MPI_Slope and MPIOMP_Slope continued to shorten. In this experiment, MPI_Slope and MPIOMP_Slope using 64 or more processes/threads were faster than CUDA_Slope. Runtimes of tested programs of the P&D algorithm (Table 2) similarly show a good

Table 1. Runtimes (unit: s) of tested programs for slope gradient calculation.

		Count of process units used with SMP cluster							
		With 1 compute node				With 16 compute nodes			
	GPU server	1	2	4	8	16	32	64	128
Serial_Slope	–	5.43	–	–	–	–	–	–	–
CUDA_Slope	0.15	–	–	–	–	–	–	–	–
MPI_Slope	–	5.44	2.76	1.43	0.75	0.39	0.21	0.10	0.05
MPIOMP_Slope*	–	5.44	2.76	1.38	0.72	0.38	0.21	0.11	0.06

Note: *MPIOMP_Slope was tested with one, and with 16 compute nodes (1 process × 1/2/4/8 threads for each compute node).

Table 2. Runtimes (unit: s) of tested programs for P&D DEM-preprocessing.

	GPU server	Count of process units used with SMP cluster							
		With 1 compute node				With 16 compute nodes			
		1	2	4	8	16	32	64	128
Serial_DEMPreprocessing	–	40,031	–	–	–	–	–	–	–
CUDA_DEMPreprocessing	408	–	–	–	–	–	–	–	–
MPI_DEMPreprocessing	–	40,118	20,101	10,181	5666	2857	1440	744	393
MPIOMP_DEMPreprocessing*	–	40,114	20,738	10,321	6079	2659	1421	820	491

Note: *MPIOMP_DEMPreprocessing was tested with one, and with 16 compute nodes (1 process × 1/2/4/8 threads for each compute node).

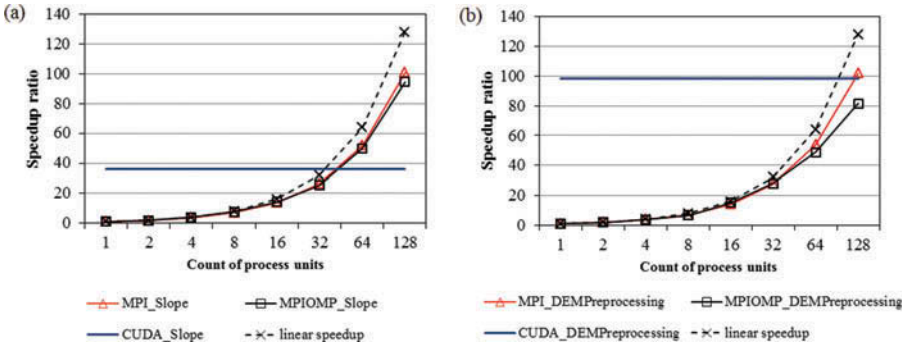

Figure 9. Speedup ratios of parallel programs of (a) the slope gradient algorithm and (b) the P&D algorithm.

performance, which is due to the effectiveness of PaRGO. In this experiment, CUDA_DEMPreprocessing also achieved high performance, which was only slower than MPI_DEMPreprocessing using 128 processes. This shows the advantage of GPU device for parallelization for raster-based geocomputation algorithms.

As shown in Figure 9, the speedup ratios of the tested MPI version and MPI/OpenMP version parallel programs using PaRGO indicated almost linear speedup. In this test, the CUDA-version programs with PaRGO also achieved a high speedup ratio, especially for the compute-intensive P&D algorithm of DEM-preprocessing (with a speedup ratio near 100). The experimental results show the effectiveness of PaRGO.

5. Conclusion and future work

This paper presents a strategy to address the portability problem in existing parallel raster-based programming libraries. This strategy is illustrated through the design and implementation of a set of PaRGO compatible with three popular types of parallel computing platforms (i.e., GPU supported by CUDA, Beowulf cluster supported by MPI, and SMP cluster supported by MPI/OpenMP). PaRGO encapsulates three types of parallel programming details (i.e., domain decomposition, communication, and parallel I/O) in a form that is transparent to users. In the I/O module of PaRGO, a parallel I/O mode using GDAL is

implemented to support a variety of commonly used geospatial raster data formats with a high I/O efficiency.

By using PaRGO in a sequential object-oriented programming style, geocomputation algorithm developers who lack knowledge and experience of parallel programming can quickly develop parallel raster-based geocomputation algorithms based on local, focal, and global raster operations. In addition, parallel algorithms with PaRGO can be compiled into MPI-version, MPI/OpenMP-version, or CUDA-version programs, so as to be compatible with three popular types of parallel computing platforms. Practical applications of PaRGO in coding parallel algorithms of slope gradient and DEM-preprocessing showed the effectiveness of PaRGO.

The current version of PaRGO cannot directly support parallelization of regional operation in raster-based geocomputation. Recently, some new parallelization strategies for DTA algorithms with typical regional operation (e.g., flow accumulation computation) have been proposed (Qin and Zhan 2012, Schiele *et al.* 2012). It is possible that parallelization of regional operation could be supported by combining PaRGO with the new parallelization strategies for DTA algorithms with typical regional operation. More work is needed to confirm this utility of PaRGO.

Currently, PaRGO includes three versions with a unified user interface, that is, MPI-version, MPI/OpenMP-version, and CUDA-version, which is compatible with Beowulf cluster, SMP clusters, and GPU, respectively. With the rapid development of parallel programming techniques, cross-platform parallel programming library and standard (e.g., OpenCL) has been released recently. Even handheld/embedded devices can be used for parallel computing. Under the strategy proposed in this paper, it still needs additional work to develop new version of PaRGO based on new-released parallel programming library to provide enhanced performance and functionality for parallel raster-based geocomputation.

Funding

This study was supported by the National High-Tech Research and Development Program of China [grant number 2011AA120302], the National Natural Science Foundation of China, and the Institute of Geographic Sciences and Natural Resources Research, CAS [grant number 2011RC203].

Note

1. GDAL: http://www.gdal.org/, with current version of 1.9.2.

Supplemental data

Supplemental data for this article can be accessed at http://www.researchgate.net/publication/261552707_DEM-preproc-PD_alg_code-PaRGO

References

Armstrong, M.P., 2000. Geography and computational science. *Annals of the Association of American Geographers*, 90 (1), 146–156. doi:10.1111/0004-5608.00190.

Armstrong, M.P. and Marciano, R.J., 1996. Local interpolation using a distributed parallel supercomputer. *International Journal of Geographical Information Systems*, 10 (6), 713–729. doi:10.1080/02693799608902106.

Bruce, R.A., *et al.*, 1995. CHIMP and PUL: support for portable parallel computing. *Future Generation Computer Systems*, 11 (2), 211–219. doi:10.1016/0167-739X(94)00063-K.

Clarke, K.C., 2003. Geocomputation's future at the extremes: high performance computing and nanoclients. *Parallel Computing*, 29 (10), 1281–1295. doi:10.1016/j.parco.2003.03.001.

Dongarra, J., *et al.*, 2005. *Sourcebook of parallel computing*. San Francisco, CA: Elsevier.

Duckham, M., Goodchild, M.F., and Worboys, M.F., 2003. *Foundations of geographic information science*. New York: Taylor & Francis.

Guan, Q.F. and Clarke, K.C., 2010. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. *International Journal of Geographical Information Science*, 24 (5), 695–722. doi:10.1080/13658810902984228.

Healey, R., *et al.*, 1998. *Parallel processing algorithms for GIS*. Bristol, PA: Taylor & Francis.

Hengl, T. and Reuter, H.I., eds., 2009. *Geomorphometry: concepts, software, applications*. Amsterdam: Elsevier.

Horn, B.K.P., 1981. Hill shading and the reflectance map. *Proceedings of the IEEE*, 69 (1), 14–47. doi:10.1109/PROC.1981.11918.

Huang, F., *et al.*, 2011. Preliminary study of a cluster-based open-source parallel GIS based on the GRASS GIS. *International Journal of Digital Earth*, 4 (5), 402–420. doi:10.1080/17538947.2010.543954.

Hutchinson, D., *et al.*, 1996. Parallel neighbourhood modelling. *In*: *Proceedings of the 4th ACM international workshop on advances in geographic information systems*, 15–16 November, Rockville, MD.

Lin, C. and Snyder, L., 2008. *Principles of parallel programming*. Boston, MA: Addison-Wesley.

Membarth, R., Lokhmotov, A., and Teich, J., 2011. Generating GPU code from a high-level representation for image processing kernels. *In*: *Proceedings of the 5th workshop on highly parallel processing on a chip*, 30 August, Bordeaux.

Nieplocha, J., *et al*., 2006. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20 (2), 203–231.

Planchon, O. and Darboux, F., 2001. A fast, simple and versatile algorithm to fill the depressions of digital elevation models. *Catena*, 42, 159–176.

Qin, C.-Z. and Zhan, L.-J., 2012. Parallelizing flow-accumulation calculations on graphics processing units – from iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. *Computers & Geosciences*, 43, 7–16. doi:10.1016/j.cageo.2012.02.022.

Qin, C.-Z., Zhan, L.-J., and Zhu, A.-X., 2013. How to apply the geospatial data abstraction library (GDAL) properly to parallel geospatial raster I/O? *Transactions in GIS*. doi:10.1111/tgis.12068.

Qin, C.-Z., *et al.*, 2006. Review of multiple flow direction algorithms based on gridded digital elevation models. *Earth Science Frontiers*, 13, 91–98. (in Chinese with English abstract).

Schiele, S., *et al.*, 2012. Parallelization strategies to deal with non-localities in the calculation of regional land-surface parameters. *Computers & Geosciences*, 44, 1–9. doi:10.1016/j.cageo.2012.02.023.

Skidmore, A.K., 1989. A comparison of techniques for calculating gradient and aspect from a gridded digital elevation model. *International Journal of Geographical Information Systems*, 3, 323–334. doi:10.1080/02693798908941519.

Tesfa, T.K., *et al.*, 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. *Environmental Modelling & Software*, 26 (12), 1696–1709. doi:10.1016/j.envsoft.2011.07.018.

Wang, S. and Armstrong, M.P., 2009. A theoretical approach to the use of cyberinfrastructure in geographical analysis. *International Journal of Geographical Information Science*, 23 (2), 169–193. doi:10.1080/13658810801918509.

Wang, X., Li, Z., and Gao, S., 2012. Parallel remote sensing image processing: taking image classification as an example. *In*: Z. Li, *et al*., eds. *Proceedings of the 6th international symposium on intelligence computation and applications*, 27–28 October, Wuhan. Berlin: Springer-Verlag, 159–169.

Warmerdam, F., 2008. The geospatial data abstraction library. *In*: H. Brent and L.G. Michael, eds. *Open source approaches in spatial data handling*. Berlin: Springer, 87–104.

Zhang, J., 2010. Towards personal high-performance geospatial computing (HPC-G): perspectives and a case study. *In*: *Proceedings of the ACM SIGSPATIAL international workshop on high performance and distributed geographic information systems*, 2 November, San Jose, CA.