Mathematical Modelling and Analysis
Volume 10 Number 2, 2005, pages 155–172
© 2005 Technika ISSN 1392-6292

PARALLEL ALGORITHMS FOR SOLUTION OF NONLINEAR DIFFUSION PROBLEMS IN IMAGE SMOOTHING [1]

R. ČIEGIS, A. JAKUŠEV, A. KRYLOVAS and O. SUBOČ

Vilnius Gediminas Technical University

Saulėtekio al. 11, LT-10223 Vilnius, Lithuania

E-mail: `{rc, Aleksandr.Jakushev, akr, os}@fm.vtu.lt`

Received March 2 2005; revised May 5 2005

Abstract. In this work we consider parallel algorithms for solution of nonlinear parabolic PDEs. First mathematical models describing nonlinear diffusion filters are presented. The finite–volume method is used to approximate differential equations. Parallel algorithms are based on the domain decomposition method. The algorithms are implemented by using *ParSol* parallelization tool and a brief description of this tool is also presented. The efficiency of proposed parallel algorithms is investigated and results of the scalability analysis are given. Theoretical predictions are compared with results of computational experiments. Application of nonlinear diffusion filters for analysis of computer tomography images is discussed in the last section of the paper.

Key words: Nonlinear diffusion filters, parallel algorithms, finite–volume method

1. Introduction

The aim of this paper is to present parallel numerical methods and computational results for processing two–dimensional images. The selected mathematical models for image fi ltration are based on partial differential equations approach. Some very popular image fi lters are obtained by convulsion with Gaussian function G_σ of increasing variance. Application of the Gaussian fi lter is equivalent to solving a linear parabolic problem [21]

$$\begin{cases} \dfrac{\partial u(X,t)}{\partial t} = \sum\limits_{i=1}^{2} \dfrac{\partial^2 u(X,t)}{\partial x_i^2}, \quad X = (x_1, x_2) \in Q, \\ u(X,0) = u_0(X)\,, \end{cases} \tag{1.1}$$

[1] This work has been supported in part by Lithuanian State Science and Studies Foundation grant V-04049 and Eureka grant CTBSTROKE E!2981.

where u_0 is an initial image and $t^{1/2}$ defines the the variance of the Gaussian function.

An important improvement of the edge detection theory has been proposed in the pioneering work of Perona and Malik [17]. They introduced a nonlinear diffusion equation of the porous medium type:

$$\frac{\partial u(X,t)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g(|\nabla u(X,t)| \, \frac{\partial u(X,t)}{\partial x_i}\Big), \quad X = (x_1, x_2) \in Q, \qquad (1.2)$$

where g is a smooth nonincreasing function, which is tending to zero at infinity. Here we use a notation

$$\nabla u(X,t) = \Big(\frac{\partial u(X,t)}{\partial x_1}, \; \frac{\partial u(X,t)}{\partial x_2} \Big).$$

The equation selectively diffuses the image in regions where u is smooth and stops the diffusion where $\nabla u(X,t)$ is large. The diffusion is strongly slowed down on edges of the image for large gradients. Outside of edges this nonlinear filter behaviuor is similar to the linear case of equation. Examples of function g were given in [17]

$$g(s) = \frac{1}{1+s^2}, \quad g(s) = e^{-s^2} \, .$$

A direct application of this equation is restricted due to two difficulties:

- It can not be used for noisy signals.
- The equation itself can be unstable for some g. For example if $sg(s)$ is decreasing at some s then we get the inverse heat equation, which is ill posed (see [2]).

It was proposed by Catte, Lions, Morel and Coll [17] to define a modified nonlinear initial–boundary value parabolic problem in $(X,t) \in Q \times (0,T]$:

$$\begin{cases} \dfrac{\partial u(X,t)}{\partial t} = \sum\limits_{i=1}^{2} \dfrac{\partial}{\partial x_i}\Big(g\big(\big|\nabla G_\sigma * u(X,t)\big|\big), \dfrac{\partial u(X,t)}{\partial x_i}\Big) + f(u_0 - u), \\ \partial_\nu u = 0, \quad (X,t) \in \partial Q \times (0,T], \\ u(X,0) = u_0(X)\,, \quad X \in Q\,. \end{cases} \qquad (1.3)$$

Diffusion filters are extensively considered in the book of Weickert [20], a good review on image processing with partial differential equations is presented by Mikula [13].

The rest of the paper is organized as follows. In Section 2, we formulate nonlinear diffusion problems, which add new nonlinearities and enable slow and fast diffusion effects in image processing. In Section 3, we present finite–difference schemes for solving numerically the given initial–boundary value problems. Parallel versions of two algorithms are presented in Section 4. The domain decomposition is used to construct parallel algorithms and the are implemented by by using the parallel array object tool *ParSol*. This tool is described briefly at the beginning of this section. Then a parallel version of the code follows semi-automatically from the serial one. We also investigate the efficiency of parallel algorithms. In Section 5, we discuss numerical experiments with real and artificial images.

2. Nonlinear Diffusion Filters

In some applications it is important to make the image multiscale analysis locally dependent not only on values of the intensity function u but also on the position in the image X. For example we want to apply a different speed of diffusion process in different parts of the image or for different ranges of the intensity function. In such situations the following nonlinear diffusion problems can be used [11]

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g\big(|\nabla G_\sigma * \beta(X,u)|\big), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u), \qquad (2.1)$$

or

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g\big(|\nabla G_\sigma * b(X,u)|\big), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u). \qquad (2.2)$$

In the points, where the derivative β'_u is small (b'_u is large) the diffusion process is slowed down, while where β'_u is large (b'_u is small) this process is fasted up. Interesting examples of application of such nonlinear diffusion problems are given in [11, 13].

3. Finite–Difference Approximations

Usually a discrete image is given on a structure of pixels with rectangular shape. This fact defines a discrete space mesh

$$\omega_h = \big\{ (x_{1i}, x_{2j}) : \quad x_{\alpha k} = kh,\ k = 0, 1, \ldots, N_\alpha \big\}.$$

We also introduce a uniform time grid

$$\omega_\tau = \big\{ t^n : \quad t^n = n\tau,\ n = 0, 1, \ldots, M \big\}.$$

The following notation for discrete functions and finite-differences is used:

$$U_{ij}^n = U(x_{1i}, x_{2j}, t^n), \quad (x_{1i}, x_{2j}, t^n) \in \omega_h \times \omega_\tau\ ,$$

$$\partial_t U_{ij}^n = \frac{U_{ij}^n - U_{i,j}^{n-1}}{\tau}, \quad \partial_{x_1}^- U_{ij}^n = \frac{U_{ij}^n - U_{i-1,j}^n}{h}, \quad \partial_{x_1}^+ U_{ij}^n = \frac{U_{i+1,j}^n - U_{ij}^n}{h},$$

$$\partial_{x_2}^- U_{ij}^n = \frac{U_{ij}^n - U_{i,j-1}^n}{h}, \quad \partial_{x_2}^+ U_{ij}^n = \frac{U_{i,j+1}^n - U_{ij}^n}{h}.$$

3.1. Explicit approximation

By using the finite volume method for approximation of space derivatives, approximating the time derivative by the forward Euler formula and treating the diffusion and nonlinear terms of equation (1.3) from the previous time step we get the following explicit discrete scheme

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^{+} \big(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^{-} U_{ij}^n\big) + f\big(u_{0,ij} - U_{ij}^n\big)\,, \tag{3.1}$$

where a_α defi nes the discrete approximation of the nonlinear diffusion coeffi cient at the boundary of each control volume, e.g.:[18]

$$a_{1,i+1/2,j} = g\Big(\Big((\partial_{x_1}^{+} V_{ij}^n)^2 + \Big(\frac{\partial_{x_2}^{+} V_{ij}^n + \partial_{x_2}^{-} V_{ij}^n + \partial_{x_2}^{+} V_{i+1,j}^n + \partial_{x_2}^{-} V_{i+1,j}^n}{4}\Big)^2\Big)^{1/2}\Big),$$

where $V_{ij}^n = G_\sigma * U_{ij}^n$. The convergence of this scheme is investigated in [2].

3.2. Semi-implicit approximation

The explicit scheme (3.1) is stable only if $\tau \leqslant ch^2$. In order to get an unconditionally stable scheme (with respect to the linear stability condition) we consider the following semi–implicit discrete approximation:

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^{+} \big(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^{-} U_{ij}^{n+1}\big) + f\big(u_{0,ij} - U_{ij}^n\big)\,. \tag{3.2}$$

The convergence analysis of this scheme is presented in [14]. Some implicit and semi–implicit fi nite–difference schemes for solving nonlinear parabolic problems were proposed and investigated by Čiegis *et al.* [5, 6].

At each iteration the obtained sparse system of linear equations is solved by iterative Conjugate Gradient (CG) algorithm. It is well–known that iterative methods can be parallelized much more simply than the direct methods.

Let us write a linear system for solving one iteration of (3.2) as

$$AV = F\,, \quad V = U^{n+1}\,, \tag{3.3}$$

then the preconditioned CG algorithm can be written in the following form [8]:

```
procedure The serial PCG algorithm
begin
     (1)  $V^0, \quad n = 0, \quad R^0 = AV^0 - F,$
     (2)  $BW^0 = R^0, \quad P^0 = W^0\,.$
     (3)  while $\big((W^n, R^n) > \varepsilon\,(W^0, R^0)\big)$
     (4)       $G^n = AP^n\,,$
     (5)       $\tau_{n+1} = \frac{(W^n, R^n)}{(G^n, P^n)}\,,$
     (6)       $V^{n+1} = V^n - \tau_{n+1} P^n\,,$
     (7)       $R^{n+1} = R^n - \tau_{n+1} G^n\,,$
     (8)       $BW^{n+1} = R^{n+1}\,.$
     (9)       $\beta_n = \frac{(W^{n+1}, R^{n+1})}{(W^n, R^n)}\,,$
     (10)      $P^{n+1} = W^{n+1} + \beta_n P^n\,,$
     (11)      $n := n + 1\,.$
     (12) end while
end
```

Here B is a preconditioning matrix. We restrict ourself to using only diagonal preconditioners, since such preconditioners can be parallelized effi ciently. More effi cient preconditioners are known, e.g. the incomplete IC factorization preconditioning [8], but they are serial in nature. Thus there have been many studies of the use of various ordering techniques to overcome the trade–off between parallelism and convergence in incomplete factorization (see, e.g., [7, 15]). The scalability analysis of parallel PCG algorithms is presented in [4, 10].

3.3. Kačur's scheme

For completeness of the material we also present special approximations for nonlinear parabolic problems (2.1) and (2.2) with slow and fast diffusion [11]. In the case when $b(x,s)$ is nondecreasing Lipschitz continuous in s, $b(x,0)=0$ and $\beta(x,s)\equiv s$ the solution of (2.1) is approximated by the solution θ^{n+1} of the regular discrete elliptic problem

$$\lambda^{n+1}\frac{\theta^{n+1}-U^n}{\tau}=\sum_{\alpha=1}^{2}\partial^{+}_{x_\alpha}\big(a_\alpha(U^n_{ij})\,\partial^{-}_{x_\alpha}\theta^{n+1}_{ij}\big)+f\big(u_{0,ij}-U^n_{ij}\big)\,, \qquad (3.4)$$

where λ^{n+1} is the relaxation function, which should satisfy the condition

$$\frac{1}{2}\tau^d\leqslant\lambda^{n+1}\leqslant\min\Big\{\frac{B\big(X,U^n+\alpha(\theta^{n+1}-U^n)\big)-B(X,U^n)}{\theta^{n+1}-U^n},\,K\Big\}\,.$$

Here $\alpha\in(0,1)$, $K>0$, $d\in(0,1)$ are parameters of the method, α is close to 1, and K is large. Function B is defi ned as

$$B(X,s)=b(X,s)+\tau^d s\,.$$

Then a new solution U^{n+1} is obtained by solving the algebraic equation

$$B(X,U^{n+1})=B(X,U^n)+\lambda^{n+1}\big(\theta^{n+1}-U^n\big)\,.$$

Similarly equation (2.2) is approximated by the fi nite–difference scheme

$$\lambda^{n+1}\frac{\theta^{n+1}-U^n}{\tau}=\sum_{\alpha=1}^{2}\partial^{+}_{x_\alpha}\big(a_\alpha\big(B(U^n_{ij})\big)\,\partial^{-}_{x_\alpha}\theta^{n+1}_{ij}\big)+f\big(u_{0,ij}-U^n_{ij}\big)\,, \qquad (3.5)$$

The other interesting for applications case is obtained when $\beta(x,s)$ is nondecreasing Lipschitz continuous in s, $\beta(x,0)=0$ and $b(x,s)\equiv s$. The solution of (2.2) is approximated by the solution $\theta^{n+1}\approx\beta(x,U^{n+1})$ of the following regular discrete elliptic problem

$$\mu^{n+1}\frac{\theta^{n+1}-\beta(U^n)}{\tau}=\sum_{\alpha=1}^{2}\partial^{+}_{x_\alpha}\big(a_\alpha(U^n_{ij})\,\partial^{-}_{x_\alpha}\theta^{n+1}_{ij}\big)+f\big(u_{0,ij}-U^n_{ij}\big)\,, \qquad (3.6)$$

where μ^{n+1} is the relaxation function, which should satisfy the condition

$$\frac{1}{2}\tau^d \leqslant \mu^{n+1} \leqslant \min\left\{\frac{B^{-1}\big(X, B(U^n) + \alpha\big(\theta^{n+1} - \beta(U^n)\big)\big) - U^n}{\theta^{n+1} - \beta(U^n)},\, K\right\}.$$

Here function B is defined as

$$B(X,s) = \beta(X,s) + \tau^d s\,.$$

The new solution U^{n+1} is obtained from the algebraic correction

$$U^{n+1} = U^n + \mu^{n+1}\big(\theta^{n+1} - \beta(U^n)\big)\,.$$

Similarly equation (2.1) is approximated by the finite–difference scheme

$$\mu^{n+1}\frac{\theta^{n+1} - \beta(U^n)}{\tau} = \sum_{\alpha=1}^{2} \partial^+_{x_\alpha}\left(a_\alpha\big(\beta(U^n_{ij})\big)\,\partial^-_{x_\alpha}\theta^{n+1}_{ij}\right) + f\big(u_{0,ij} - U^n_{ij}\big)\,, \quad (3.7)$$

4. Parallel Algorithms

The power of modern personal computers is increasing constantly, but not enough to fulfill all scientific and engineering computational demands. In such cases, parallel computing may be the answer. Parallel computing not only gives access to increasing computational resources, but it also may be economically feasible – lots of computers are connected to various networks now, and a large portion of them spend most of their time idle.

The major difficulty in using parallel computers, however, is that writing a parallel program (or parallelizing existing sequential codes), requires the knowledge of special methods and tools, which is not trivial to be mastered [16]. Hence the main obstacle in the spreading parallel computing is the lack of specialists who may create parallel software.

One of the ways to improve the situation is the creation of tools to simplify the parallelization of algorithms. We have developed a new tool, which can be used for semi–automatic parallelization of data parallel algorithms, that are implemented in `C++`.

4.1. Parallel programming tools and standards

In this section we describe very briefly the main parallel programming standards and tools used for parallelization of serial codes which can be parallelized using the data decomposition method.

MPI (*Message Passing Interface*) is a standard for a `C/C++` or Fortran libraries [19]. It is wide spread, has lots of implementations on different platforms, both commercial and free. However, it is quite complicated, and parallelizing programs using MPI is a tedious process [9].

HPF (*High Performance Fortran*) is an extension of Fortran language standard. HPF is well suitable for algorithms with parallel data. If program is written in standard Fortran following some simple rules (considering the usage of Fortran arrays), then it may be parallelized just by adding several directives, describing such things as processor topology [16]. The drawbacks of HPF are diminishing popularity of Fortran language and the need to develop separate HPF compiler.

OpenMP (*Open Multi Programming*) is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism [3]. It provides a standard among a variety of shared memory architectures/platforms. OpenMP establishes a simple and limited set of directives for programming shared memory machines. Signifi cant parallelism can be implemented by using just 3 or 4 directives.

We also mention *Unified Parallel C* (UPC), which is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. *UPC* uses a *Single Program Multiple Data* (SPMD) model of computation in which the amount of parallelism is fi xed at program startup time, typically with a single thread of execution per processor [1].

4.2. Parallel array objects

The aim of *ParSol* is to bring HPF parallelization simplicity to C++language, using popular parallelization standards. Hence, the current *ParSol* library features are:

- Created for C++ programming language;
- Based on HPF ideology;
- The library heavily uses such C++ features as OOP and templates;
- Only standard C/C++ features are used;
- Currently, MPI 1.1 standard is used to implement parallelization [9, 19];
- *ParSol* currently is open source library.

Due to compliance to general standards, *ParSol* is expected to be used on wide variety of platforms. Currently *ParSol* has been tested on the following confi gurations:

- MS-Windows OS, MS Visual C++ compiler, MPICH MPI library;
- Linux OS, gcc compiler, LAM MPI library (`http://vilkas.vtu.lt`);
- IBM SP4 supercomputer, VisualAge C++ compiler, IBM MPI library (`http://www.cineca.it`).

At present, *ParSol* may be used for parallelization of data–parallel or domain–decomposition algorithms.

ParSol structure and usage

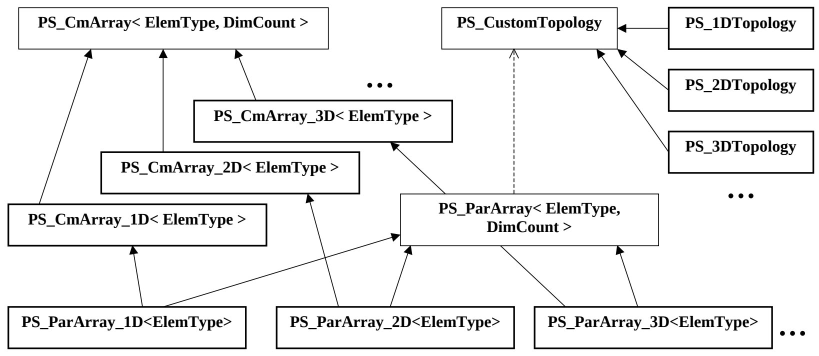
PS_CmArray< ElemType, DimCount >
PS_CustomTopology
PS_1DTopology
PS_2DTopology
PS_3DTopology
PS_CmArray_3D< ElemType >
PS_CmArray_2D< ElemType >
PS_CmArray_1D< ElemType >
PS_ParArray< ElemType, DimCount >
PS_ParArray_1D<ElemType>
PS_ParArray_2D<ElemType>
PS_ParArray_3D<ElemType>
...

Figure 1. ParSol library class diagram.

ParSol class diagram is shown in Fig. 1. The main elements of the library are:

Sequential array classes

These are the classes to be used instead of native C/C++ arrays. No MPI or other libraries, except *ParSol* itself, is necessary to use them. Comparing to native C/C++ arrays, *ParSol* sequential arrays have a number of advantages for programming mathematical algorithms, such as virtual indexes, built-in array operations, automated management of dynamically allocated memory, periodical boundary conditions.

The main functionality resides in template class `PS_CmArray`. However, general functionality requires interface complexity. So children are derived for special cases (1D, 2D, 3D arrays), that provide intuitive and user-friendly interface. It is recommended for end-user to use those classes whenever possible.

Parallel array classes

If parallel arrays are to be used in place of sequential ones, it is natural to make them the descendants of appropriate sequential arrays, adding parallelization code to the sequential array functionality. However, parallelization is similar for different kinds of arrays. So parallelization code is localized in class `PS_ParArray`, and is used in parallel array classes by multiple inheritance.

Parallelization

A general schema for construction of data parallel algorithms is illustrated on Fig. 2. It consists of the following steps:

1. Determine the part of sequential array that belongs to given process;
2. Determine the neighbour processes that will participate in information exchange;
3. Determine the amount of data to be exchanged with every neighbour process;
4. Exchange information with neighbours, when required.

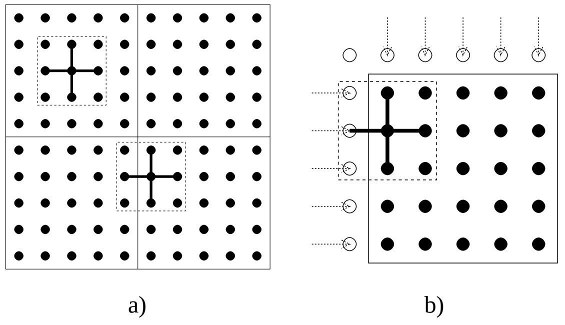

Figure 2. Transition from sequential to parallel array: a) serial array and an example of the stencil, b) local part of one processor with the appropriately extended set of nodes.

Topology classes

The purpose of these classes is to ensure that all processes are in proper order for parallel array functionality. In HPF, this functionality is performed by special directives. All general code resides in `PS_CustomTopology` class. As with sequential array classes, there are also descendants `PS_{1,2,3}DTopology`, which provide end–user with more friendly interface for one-, two- and three-dimensional cases.

Stencil classes

A stencil is determined depending on requirements of the computational scheme. Based on stencil, different amount of information needs to be exchanged among neighbours (see Fig. 2*b*). This part of data is required for parallel arrays to operate properly.

To use *ParSol*, a programmer must develop his/her sequential application in the same way as without *ParSol*, only using *ParSol* arrays wherever computational data is stored. The other requirements are to specify the stencil, make algorithm independent on the order in which array points are processed and use global array operations provided by *ParSol* wherever possible. The last one may also be called an advantage, because it frees programmer from implementation of simple tasks, allowing to concentrate on problem solving, and makes code cleaner.

The parallelization of such a sequential program takes the following steps:

1. Replace includes of sequential headers with parallel ones, for example `PS_CommonArray.h` to `PS_ParallelArray.h`;
2. Replace sequential classes with their parallel analogy in variable declarations only;
3. Add MPI initialization code (one line at the beginning of the program);
4. Add topology initialization code (in its simplest case, one line at the beginning of the program);
5. Specify when array neighbours should exchange data.

Finally, MPI library should be linked during a building process.

4.3. Computational experiments

In this section we present some results of computational experiments. Computations were performed on PC cluster "Vilkas" of Vilnius Gediminas technical university and IBM SP4 computer at CINECA, Bologna.

Explicit nonlinear algorithm (3.1)

We have filtered an artificial image of dimension $N \times N$. First we consider a parallel implementation of the explicit nonlinear algorithm (3.1). Table 1 presents experimental speedup $S_p(N)$ and efficiency $E_p(N)$ values for solving problems of different size on PC cluster "Vilkas". Here p is the number of processors,

$$S_p(N) = \frac{T_1(N)}{T_p(N)}, \quad E_p(N) = \frac{S_p(N)}{p}$$

and $T_p(N)$ is CPU time required to solve the problem with p processors. A two–dimensional data decomposition $p_1 \times p_2$ was used with p_1, p_2 values as close to each other as possible. If p is a prime number then we get one–dimensional data decomposition. The image processing was done till time moment $T(N)$ and the following CPU times $T_1(N)$ (in *s*) were obtained for the sequential algorithm

$$T(160) = 0.1, \; T_1(160) = 213.3, \quad T(240) = 0.03, \; T_1(240) = 332.8,$$
$$T(320) = 0.01, \; T_1(320) = 361.6.$$

Table 1. The speedup and efficiency for explicit algorithm (3.1) on PC cluster.

p	$S_p(160)$	$E_p(160)$	$S_p(240)$	$E_p(240)$	$S_p(320)$	$E_p(320)$
2	1.56	0.780	1.76	0.880	1.87	0.934
4	2.36	0.590	3.00	0.750	3.45	0.862
6	2.78	0.463	3.93	0.655	4.77	0.795
8	2.95	0.369	4.69	0.585	5.88	0.735
9	3.16	0.351	5.04	0.560	6.28	0.698
11	3.33	0.303	5.50	0.500	7.09	0.644
12	3.35	0.279	5.64	0.470	7.47	0.623
15	3.39	0.226	6.38	0.425	8.56	0.571

Table 2 presents experimental speedup $S_p(N)$ and efficiency $E_p(N)$ values for solving the same problem on SP4 computer. The following CPU times $T_1(N)$ (in *s*) were obtained for the sequential algorithm

$$T_1(80) = 57.24, \quad T_1(160) = 471.2, \quad T_1(320) = 770.4.$$

Table 2. The speedup and efficiency for explicit algorithm (3.1) on SP4.

p	$S_p(80)$	$E_p(80)$	$S_p(160)$	$E_p(160)$	$S_p(320)$	$E_p(320)$
2	1.975	0.988	1.984	0.992	2.004	1.002
3	2.794	0.931	2.950	0.985	2.970	0.990
4	3.741	0.935	3.928	0.982	3.986	0.996
6	5.168	0.861	5.463	0.910	5.916	0.986
8	6.766	0.846	7.293	0.911	7.831	0.979
9	6.784	0.754	7.604	0.845	8.467	0.941
12	8.701	0.725	10.19	0.849	11.216	0.934
16	10.84	0.677	12.75	0.797	15.041	0.940
24	14.18	0.591	18.24	0.760	21.961	0.915

Scalability Analysis

In this section we will estimate the complexity of the parallel implementation of discrete scheme (3.1). The complexity of the serial algorithm is given by

$$W = \gamma M N^2 ,$$

where M is the number of iterations, $N \times N$ is the dimension of the image, and γ estimates the CPU time required to implement one basic operation of the algorithm.

The parallel algorithm is based on the domain decomposition method. This data decomposition is implemented automatically by parallel array objects. The nodes of 2D image can be partitioned among the processors by using 1D and 2D Cartesian mappings.

Let $p_1 \times p_2$ be a topology of processors. Then the computational complexity of parallel algorithm (3.1) depends on the size of largest local grid part, given to one processor. It is equal to

$$T_{p,comp} = \gamma M \left\lceil \frac{N}{p_1} \right\rceil \times \left\lceil \frac{N}{p_2} \right\rceil ,$$

where $\lceil x \rceil$ denotes a smallest integer number larger or equal to x. The effi ciency of the parallel algorithm depends on the disbalance of sizes of the distributed local subgrids. We defi ne the computational overhead of the parallel algorithm

$$H_{comp}(p_1, p_2) = p\, T_{p,comp} - W \, .$$

It is always important to reduce this disbalance by selecting a proper topology of processor distribution. We illustrate this statement by two simple cases. First let us assume that $N = 84$ and $p = 16$. Considering three possible distributions of processors we obtain the following computational overheads of the parallel algorithm

$$H_{comp}(1,16) = 504 M\gamma, \quad H_{comp}(2,8) = 336 M\gamma, \quad H_{comp}(4,4) = 0 \, .$$

Let us assume that no additional overhead is introduced by the parallel algorithm. Then the effi ciency $E_p(N)$ can be written as

$$E_{p_1 \times p_2}(N) = \frac{W}{W + H_{comp}(p_1,p_2)} = \frac{1}{1 + H_{comp}(p_1,p_2)/W} \,.$$

Thus we get

$$E_{1\times 16}(84) = 0.933, \quad E_{2\times 8}(84) = 0.955, \quad E_{4\times 4}(84) = 1 \,.$$

The second example illustrates the statement that in some cases a parallel algorithm can be faster if the number of processors is reduced. Let assume that $N = 80$, then we compare the speed-up of the parallel algorithm with 12 and 13 processors:

$$S_{3\times 4}(80) = 11.85, \quad S_{1\times 13}(80) = 11.43 \,.$$

Additional overheads are introduced by any parallel algorithm when one processor sends data to other processors. Let assume that two processors are exchanging n elements. This can be done in $\alpha + \beta n$ time by using e.g. the *odd–even* data exchange algorithm [12]. Here α is the message startup time and β is the time required to send one element of data.

Data Exchange Protocols

Three different data exchange protocols are implemented in *ParSol* tool and users can select each of them by specifying simple directives.

All-At-Once

In the first part of this scheme, every process initializes all sends and receives with all possible neighbours at once, using `MPI_Start`. In the second part, the process simply waits for all started communications to complete (using cycle with `MPI_Test`). In this scheme, it is up to MPI implementation to determine the best way to manage sends and receptions. It is expected that the developers of MPI library will optimize this part of the library for each type of parallel computer.

This protocol leads to a non-blocked data communication between processors and some computations may be performed between the first and second parts of data communication algorithm.

Pair-By-Pair

In this scheme, every process communicates with all possible neighbours in strict order by using the *odd-even* communication algorithm separately for each dimension. Communication with the next neighbour will not start until the communication with previous neighbour is finished. When communicating with the given neighbour, both send and receive are initialized at the same time, using `MPI_Start`, then waiting till completion of both operations (using `MPI_Wait`). We note that all processors communicate in parallel.

Pair-Ordered

This scheme is similar to *Pair-By-Pair* scheme, and differs only in the way communication between two specific neighbours takes place. In this scheme, it is strictly determined which neighbour sends data first. Data is send and received using `MPI_Start`–`MPI_Wait` combination, which is analogous to `MPI_Send` or `MPI_Recv`.

4.4. Analysis of communication costs

In this section we investigate how data communication costs depend on the number of processors and on the selected topology of their distribution. We restrict our analysis to the *odd-even* algorithm and will assume that the communication network of the parallel computer is a 2D torus.

Let the size of image is $N \times N$ and the number of iterations is M. First we use one-dimensional topology $p \times 1$ of processors. Then data communication time is given by

$$K_{p\times 1}(N) = (2\alpha + 2\beta N)M\,.$$

In the case of $\frac{p}{2} \times 2$ topology this additional overhead of the algorithm is equal to

$$K_{p/2\times 2}(N) = \Big(3\alpha + \frac{p+2}{p}\beta N\Big)M\,.$$

If processors are distributed as $\sqrt{p} \times \sqrt{p}$ torus, then data communication time is given by

$$K_{\sqrt{p}\times\sqrt{p}}(N) = \Big(4\alpha + \frac{4N}{\sqrt{p}}\beta\Big)M\,.$$

Thus for the torus topology communication costs are decreased when a number of processors is increased.

Now we will investigate results of computational experiments in order to estimate communication costs for PC cluster "Vilkas" and IBM SP4 supercomputer. First, we note that computational speed of one PC node is two times faster than SP4 node. Second, it is known that the network of the given PC cluster has comparatively very large latency, i.e. $\alpha \gg \beta$. This explains why the effi ciency of parallel algorithm is better for IBM SP4 computer and why the speedup S_p saturates more fastly for PC cluster.

In Table 3 we present overheads of the parallel algorithm implemented on the PC cluster. As was stated above they coincide with communication costs. Since the number of iterations is different for different values of N, we have scaled the results with respect to $N = 160$.

Table 3. The communication costs for explicit algorithm (3.1) on PC cluster.

p	$K_p(160)$	$K_p(240)$	$K_p(320)$
2	29.8	34.2	32.1
4	37.0	41.8	36.0
8	41.3	44.0	40.8
12	45.8	46.9	45.7

It follows from results presented in Table 3 that communication costs depend only slightly on N and thus the latency error component α dominates the total communication error. Using this information we can predict that no practical speed-up can be expected for solving a smaller problem with the image size 80×80. The computation time of the serial algorithm is $T_1(80) = 26.03$ (iterations are done till $T = 0.2$). Applying our theoretical model we predict the following complexity of the parallel algorithm:

$$T_{2,pred}(80) = \frac{26.03}{2} + \frac{29.8}{2} = 27.9, \quad T_{4,pred}(80) = \frac{26.03}{4} + \frac{37}{2} = 25.0,$$
$$T_{8,pred}(80) = \frac{26.03}{8} + \frac{41.3}{2} = 23.9.$$

Computational experiments have confirmed these theoretical predictions:

$$T_2(80) = 27.35, \quad T_4(80) = 24.51, \quad T_8(80) = 24.32.$$

Scalability analysis

The scalability analysis of any parallel algorithm enables us to find the rate at which the size of problem W needs to grow with respect to the number of processors p in order to maintain a fixed efficiency of the algorithm. Then the isoefficiency function $W = g(p, E)$ is defined by the implicit equation [12]

$$W = \frac{E}{1-E} H(p, W). \tag{4.1}$$

For simplicity of notation we take $E = 0.5$.

Let us consider $p \times 1$ topology of processors. The total overhead of the parallel algorithm for discrete scheme (3.1) is given by data communication costs, thus $H(p, W) = (2\alpha + 2\beta N)M$. Due to the stability requirement of the explicit scheme $M = cN^2$, thus we get the following equation

$$\gamma c N^4 = 2\alpha c N^2 p + 2c\beta N^3 p,$$

or

$$W = \frac{2\alpha\sqrt{c}}{\sqrt{\gamma}} pW^{1/2} + \frac{2\beta c^{1/4}}{\gamma^{3/4}} pW^{3/4}.$$

In this case it is possible to get the isoefficiency function in a closed form as a function of p, but it is more convenient to analyze the influence of each individual term. The component that requires the problem size to grow at the fastest rate determines the overall asymptotic isoefficiency function. For 1D topology of processors the overall asymptotic isoefficiency function is determined by the second term of the total overhead function

$$W = \frac{16c\beta^4}{\gamma^3} p^4 = \mathcal{O}(p^4).$$

It requires a linear growth of N with respect to p to maintain a certain efficiency E.

Let us consider $\sqrt{p} \times \sqrt{p}$ topology of processors. Then the isoefficiency function is defined by the following equation

$$W = \frac{4\alpha\sqrt{c}}{\sqrt{\gamma}} pW^{1/2} + \frac{2\beta c^{1/4}}{\gamma^{3/4}} \sqrt{p}\, W^{3/4}.$$

In this case both components require the problem size to grow at the same rate and the overall asymptotic isoeffi ciency function is given by

$$W = \mathcal{O}(p^2)\,.$$

It requires only a linear growth of the image size $N \times N$ with respect to p to maintain a certain effi ciency E.

Semi–implicit nonlinear algorithm (3.2)

Next we consider a parallel implementation of the semi–implicit nonlinear algorithm (3.2). The main computational steps are the following:

- Pre-smoothing of the image. A few steps of the explicit linear scheme are done.
- Solution of a system of linear equations by the CG iterative method.

Implementation of one CG iteration requires to compute

- Matrix–vector multiplication, which is equivalent to application of the explicit difference scheme;
- Global reduction operation, when inner–product of two vectors is computed. Such operation is implemented as a built-in method of parallel array objects of *ParSol* tool.

Scalability analysis of parallel preconditioned CG algorithms is done in [4, 10]. Table 4 presents CPU times $T_p(N)$ (in *s*) required to solve the given image processing problem on SP4 computer for different sizes of images.

Table 4. CPU times $T_p(N)$ for implicit nonlinear algorithm (3.2.)

p	$T_p(160)$	$T_p(320)$	$T_p(480)$
1	64.97	241.4	281.9
2	26.82	86.71	118.5
4	12.89	40.37	63.94
6	9.24	26.91	42.84
8	7.59	21.37	32.44
16	4.83	10.30	16.44

5. Processing of CT Images

One of the topical problems in computed tomography (CT) is reliable allocation of ischemic stroke area. A precise solution of this problem allows us to evaluate the volume of stroke and helps the medics to select the tactic of treatment properly. Possibility to solve this problem quickly enables automatic processing of CT images.

The aim our research is to develop a specialized software and to implement it as a tool. High rates of calculations can be achieved by using parallel computing, which allows to use personal computers of small hospital.

Stroke region in CT images can be of various size and form, but in all cases sorption susceptibility of touched area is 1.5-2 times larger. The example of CT image is given in fi gure 3*a*, the size of the image is 512X512 pixels.

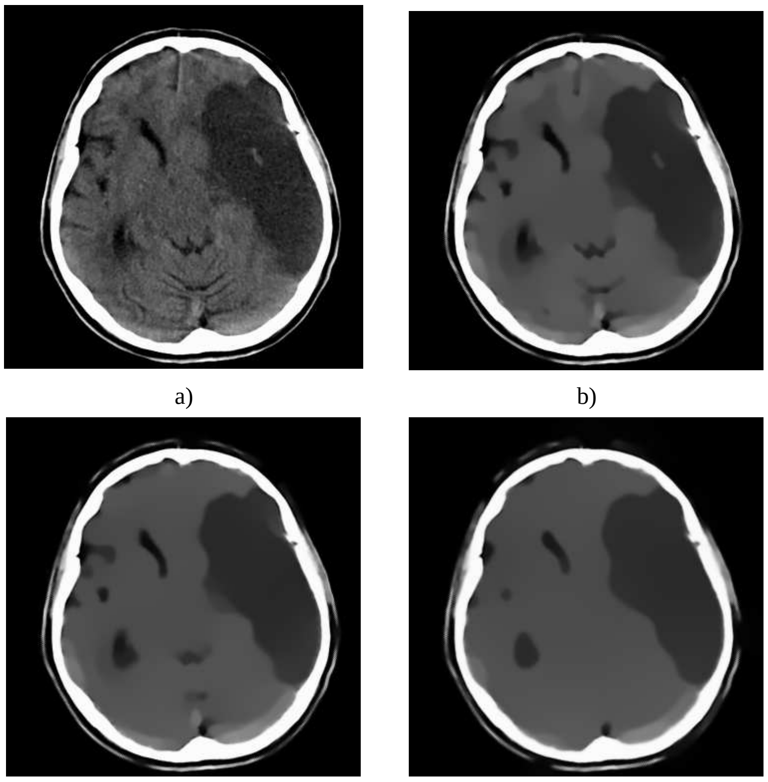

Figure 3. An image of human brain ischemic stroke in computed tomography (ischemic stroke region is denoted by darker color): a) the initial image, b) after 20 iterations, c) after 40 iterations, d) after 100 iterations.

In CT processing it is important not to disturb contours of the stroke area, since they are used for calculation of the volume of stroke area. This information is important for medics. One of advantages of non-linear fi lters is to preserve edges of the images. On fi gures 3*b,c,d* results of CT fi ltering by non-linear diffusion fi lters are presented after 20, 40 and 100 iterations.

As it is visible from results of fi ltering, contours of the image are not disturbed, thus a localization of CT stroke area is possible by using standard procedures (for example, differential fi lters). For such localization there is no need to perform 100 iterations, since we can see good enough results after 40 or less iterations. We note that apriori estimation of required number of iterations is not a simple task. It is even more important if we try to develop a specialized tool for automatic detection of stroke region. As it follows from results of numerical experiments (see 3*c,d*), nonlinear fi lters are not damaging the image even after large number of unnecessary iterations. This simplifi es their usage for automatic recognition of stroke area.

The goal of our work is to select non-linear diffusion fi lters, specially adjusted for CT processing. It is required to select parameters of fi lters to process the stroke region effi ciently and quickly. The effectiveness is estimated statistically, performing large amount of numerical experiments.

Acknowledgments

R. Čiegis did some part of this work under the Project HPC–EUROPA (RII3–CT-2003-506079), with the support of the European Community – Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme. He gratefully acknowledges the hospitality and excellent working conditions in CINECA, Bologna. In particular he thanks Dr. Giovanni Erbacci for his help.

References

[1] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks and K. Warren. *Introduction to UPC and Language Specification CCS-TR-99-157*. IDA Center for Computing Sciences, 1999.

[2] F. Catte, P.L. Lions, J.M. Morel and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Numer. Anal.*, **29**(1), 182 – 193, 1992.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. Mc Donald and R. Menon. *Parallel programming in OpenMP*. Academic Press, San Diego, 2001.

[4] R. Čiegis. Analysis of parallel preconditioned conjugate gradient algorithms. *Informatica*, **16**, 2005. (accepted for publication)

[5] R. Čiegis and M. Meil¯unas. On the difference scheme for a nonlinear diffusion – reaction type problem. *Liet. matem. rink.*, **33**(1), 16 – 29, 1993.

[6] Raim. Čiegis, Rem. Čiegis and M. Meil¯unas. On the convergence in the L_2 norm of difference schemes for systems of parabolic partial differential equations. *Liet. matem. rink.*, **33**(3), 269 – 279, 1993.

[7] S. Doi and T. Washio. Ordering strategies and related techniques to overcome the trade–off between parallelism and convergence in incomplete factorizations. *Parallel Computing*, **25**, 1995 – 2014, 1999.

[8] G.H. Golub and Ch. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[9] W. Gropp, E. Lusk and A. Skjellum. *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1999.

[10] A. Gupta, V. Kumar and A. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, **6**(5), 455 – 469, 1997.

[11] J. Kačur and K. Mikula. Slow and fast diffusion effects in image processing. *Comput. Visual. Sci.*, **3**, 185 – 195, 2001.

[12] V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. The Benjamin/Cummings Publishing Company, Inc., New York, Amsterdam, Sydney, Tokyo, 1994.

[13] K. Mikula. Image processing with partial differential equations. In: *Modern methods in scientific computing and Applications*, volume 75 of *NATO Science II*, Kluwer Academic, Dodrecht, 263 – 322, 2002.

[14] K. Mikula and N. Ramorosy. Semi–implicit finite volume scheme for solving nonlinear diffusion equations in image processing. *Numer. Math.*, **89**, 561 – 590, 2001.

[15] M. Monga-Made and H. A. van der Vorst. A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. *Future Generation Computer Systems*, **17**, 925 – 932, 2001.

[16] P. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, 1997.

[17] P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. In: *Proc. IEEE Computer Society Workshop on Computer Vision*, 1987.

[18] A.A. Samarskii. *The theory of difference schemes*. Marcel Dekker, Inc., New-York, Basel, 2001.

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra. *MPI: the complete reference*, **1**, The MIT Press, 1998.

[20] J. Weickert. *Anisotropic Diffusion in Computer Vision*. Teubner, Stuttgart, 1998.

[21] A.P. Witkin. Scale–space filtering. In: *Proc. Eight Internat. Conference on Artificial Inteligence*, volume 2, 1019 – 1022, 1983.

Apie lygiagrečiuosius algoritmus, skirtus netiesinių difuzijos lygčių sprendimui ir jų naudojimą vaizdų filtravimui

R. Čiegis, A. Jakušev, A. Krylovas, O. Subоč

Šiame darbe nagrinėjami lygiagretieji algoritmai, kurie skirti netiesinių nestacionarių difuzijos lygčių sprendimui. Pirmiausia yra suformuluoti netiesinių filtrų matematiniai modeliai. Šie uždaviniai aproksimuoti baigtinių t¯urių schemomis.

Lygiagretieji algoritmai konstruojami duomenų lygiagretumo metodu. Jie realizuoti autorių sukurtu *ParSol* programavimo įrankiu. Pateiktas trumpas šio įrankio aprašymas. Ištirtas lygiagrečiųjų algoritmų efektyvumas ir pateikti algoritmų išplečiamumo analizės rezultatai. Teorinės išvados palygintos su skaičiavimo rezultatais. Netiesiniai difuziniai filtrai pritaikyti galvos kompiuterinių tomogramų filtravimui.