Mathematical Modelling and Analysis 2005. Pages 381–388
Proceedings of the 10th International Conference MMA2005&CMAM2, Trakai
© 2005 Technika ISBN 9986-05-924-0

NONLINEAR DIFFUSION PROBLEMS IN IMAGE SMOOTHING

R. ČIEGIS, A. JAKUŠEV and O. SUBOČ

Vilnius Gediminas Technical University

Saulėtekio al. 11, LT-10223 Vilnius, Lithuania

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

Abstract. In this work we consider mathematical models describing nonlinear diffusion filters. 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. Theoretical predictions are compared with results of computational experiments. Application of nonlinear diffusion filters for analysis of computer tomography images is discussed.

Key words: nonlinear diffusion filters, parallel algorithms, finite-volume method

1. 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 [4]

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g\big(\big|\nabla G_\sigma * \beta(X,u)\big|\big), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u), \qquad (1.1)$$

or

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g\big(\big|\nabla G_\sigma * b(X,u)\big|\big), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u). \qquad (1.2)$$

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 [4, 5].

2. Finite–Difference Approximations

2.1. Explicit approximation

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^{+} \big(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^{-} U_{ij}^n\big) + f\big(u_{0,ij} - U_{ij}^n\big)\,, \tag{2.1}$$

where a_α defines the discrete approximation of the nonlinear diffusion coefficient at the boundary of each control volume, e.g.:

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

where $V_{ij}^n = G_\sigma * U_{ij}^n$. The convergence of this scheme is investigated in [1].

2.2. Semi-implicit approximation

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^{+} \big(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^{-} U_{ij}^{n+1}\big) + f\big(u_{0,ij} - U_{ij}^n\big)\,. \tag{2.2}$$

At each iteration the obtained sparse system of linear equations is solved by iterative Conjugate Gradient (CG) algorithm. Let us write a linear system for solving one iteration of (2.2) as

$$AV = F\,, \quad V = U^{n+1}\,, \tag{2.3}$$

then the preconditioned CG algorithm can be written in the following form [2]. Let B be a preconditioning matrix.

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) $\quad G^n = AP^n\,,$
(5) $\quad \tau_{n+1} = \dfrac{(W^n, R^n)}{(G^n, P^n)}\,,$
(6) $\quad V^{n+1} = V^n - \tau_{n+1} P^n\,,$
(7) $\quad R^{n+1} = R^n - \tau_{n+1} G^n\,,$
(8) $\quad BW^{n+1} = R^{n+1}\,.$
(9) $\quad \beta_n = \dfrac{(W^{n+1}, R^{n+1})}{(W^n, R^n)}\,,$
(10) $\quad P^{n+1} = W^{n+1} + \beta_n P^n\,,$
(11) $\quad n := n + 1\,.$
(12) **end while**

end

3. Parallel Algorithms

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 [6]. A possibility to improve this 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++`.

3.1. 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 [?, 8];
- *ParSol* currently is open source library.

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

ParSol structure and usage

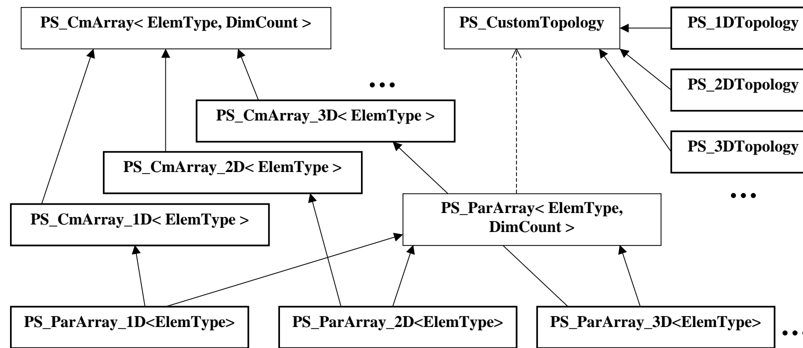

Figure 1. ParSol library class diagram.

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

Parallel array classes.

If parallel arrays are to be used in place of sequential ones, it is natural to make them to be 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 consists of the following steps:

1. Determine the part of sequential array that belongs to the 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.

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. 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.

3.2. 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 (2.1)

We have filtered an artificial image of dimension $N \times N$. First we consider a parallel implementation of the explicit nonlinear algorithm (2.1). 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.

Table 1. The speedup and efficiency for explicit algorithm (2.1) 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

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

Semi–implicit nonlinear algorithm (2.2)

Next we consider a parallel implementation of the semi–implicit nonlinear algorithm (2.2). 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.

Table 2. The speedup and efficiency for explicit algorithm (2.1) 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

Implementation of one CG iteration requires to compute *matrix–vector* multiplication, which is equivalent to application of the explicit difference scheme and *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 [7, 3]. Table 3 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 3. CPU times $T_p(N)$ for implicit nonlinear algorithm (2.2.)

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

4. 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 $2a$, the size of the image is 512X512 pixels.

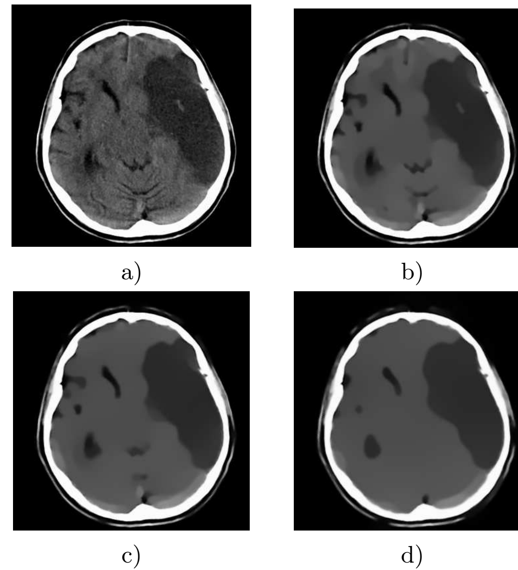

Figure 2. An image of human brain ischemic stroke in computed tomography (ischemic stroke region is denoted by darker color): a) the initial image, b) after 20 iterations, c) after 40 iterations, d) after 100 iterations.

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 filters is to preserve edges of the images. On figures $2b,c,d$ results of CT filtering by non-linear diffusion filters are presented after 20, 40 and 100 iterations.

It visible from results of filtering, contours of the image are not disturbed, thus a localization of CT stroke area is possible by using standard procedures (for example, differential filters). 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 $2c,d$), nonlinear filters are not damaging the image even after large number of unnecessary iterations. This simplifies their usage for automatic recognition of stroke area.

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] 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.

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

[3] 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.

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

[5] K. Mikula. *Image processing with partial differential equations.*

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

[7] R.Čiegis. Analysis of parallel preconditioned conjugate gradient algorithms. *Informatica*, **16**(3), 317 – 332, 2005.

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