Received: 10 March 2020 | Revised: 1 July 2020 | Accepted: 14 August 2020

DOI: 10.1002/cpe.5996

WILEY

SPECIAL ISSUE PAPER

Performance analysis of parallel high-resolution image restoration algorithms on Intel supercomputer

Ivan Lirkov[1] | Stanislav Harizanov[1,2] | Marcin Paprzycki[3] | Maria Ganzha[3,4]

[1]Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Sofia, Bulgaria

[2]Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Sofia, Bulgaria

[3]Department of Intelligent Systems, Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland

[4]Department of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland

Correspondence
Ivan Lirkov, Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Sofia 1113, Bulgaria.
Email: ivan@parallel.bas.bg

Funding information
Bulgarian National Science Fund, Grant/Award Number: KP-06-N27/6; Science and Education for Smart Growth Operational Program (2014-2020), Grant/Award Number: BG05M2OP001-1.001-0003; Polish Academy of Sciences; Bulgarian Academy of Sciences; European Union, Grant/Award Number: BG05M2OP001-1.001-0003

Summary

In this article, we present an experimental performance study of a parallel implementation of two Poissonian image restoration algorithms. Hybrid parallelization, based on MPI and OpenMP standards, is investigated. The implementation is tested for high-resolution radiographic images, on a supercomputer based on Intel Xeon processors, combined with Intel Xeon Phi coprocessors. The experimental results show an essential improvement in the execution times, when running experiments for a variety of problem sizes, and number of threads.

KEYWORDS

Anscombe transform, image restoration, parallel algorithm, Intel Xeon Phi coprocessor

1 | INTRODUCTION

Today, digital image processing is a very active interdisciplinary scientific field, combining research results, among others, from mathematics, computer science, and physics[1]. Technological progress and continuous improvement of product quality (here, the quality of the input image) are the two driving forces behind continuous stream of developments in this field. It is also worthy noting that the scientific results, obtained in this area, are applicable to a broad class of daily life activities, such as: medicine, engineering, national security, photography, material sciences, production quality control, nondestructive testing, archaeology, architecture, and many others. While, broadly understood, scientific progress in image processing is usually associated with artificial intelligence/machine learning/data analytics, all such developed methods are mainly applicable to images of highest possible quality. However, this might often not be the case and, due to specifics of the data acquisition device or natural limitations of the data acquisition process, the generated images could possess high noise levels or other artifacts that make the above mentioned approaches unreliable. Therefore, our work focuses on image preprocessing and more precisely—on nonlinear image denoising.

Let us observe that, without any doubt, the quality, and the practical value, of a given digital image, is directly related to its resolution. Increasing the resolution of the data acquisition device (thus, increasing the image resolution as well) is a prime goal not only in the area of mobile technologies, where the producers try to make their market products (camera, cell phone, and so forth) as attractive, to the customer, as possible and superior to those of the competition. Here, see, for instance, recent introduction of 108 Mega pixel cameras into multiple models of smart phones, and the realistic promise of 160 Mega pixel cell phone cameras. The same trend can be observed in medicine and nanotechnology, where the correct diagnosis of the patient, or accurate extraction of (macro) characteristics of the material, from its microstructure, strongly depends on the ability of visualizing "tiniest details" of the scanned object. Here, recent scientific progress is substantial, and there does not seem to be an end in sight. Every year, improved data acquisition devices appear on the market and, nowadays, there are detectors with 4000×4000 pixel image matrices.

Increasing the resolution of modern equipment naturally leads to the generation of large-scale images. In turn, this results in image (pre)processing, which requires substantial (and growing) computer resources (related to both time and memory). On the other hand, let us note that the time of (pre)processing of medical images, for example, cannot continue growing with the image size, because this could be fatal for the patient (awaiting results). Moreover, in the case of mobile technologies, increasing time of image processing will soon reach the "threshold of pain", beyond which customers would no longer be willing to buy "slowing down devices". This also means that the algorithms that were considered efficient "until yesterday", may (soon) not be able to cope with these new challenges.

In this context, our work is focused on accurate three-dimensional (3D) computed tomography (CT) reconstruction, which is crucial for most of the real-life applications. The two main areas of interest are: industrial material science and medicine, where CT imaging is used on daily basis. Here, it is easy to realize that obtaining 3D images is a demanding task, due to the presence of noise, in each of the generated two-dimensional (2D; radiographic) frames, which are "combined" to create the final 3D volume reconstruction. Specifically, typical industrial CT data consist of thousands of frames, since the voxel size of the 3D reconstruction is inversely proportional to the angle difference between two consecutive scans of the rotated object of interest. When such an object is inanimate (case of material science), and cannot be harmed by the X-ray radiation, the angle difference could be a fraction of the degree (here, one can take an "unlimited" number of images). This would result in a relative "ease" of reaching high quality digitized data. Nevertheless, here the demand for image quality may be extremely high, depending on the particular applications, as details on micro- and even nano-scales are often considered. On the other hand, when dealing with medical CT data, the number of scans is severely limited, due to health issues (one has to consider the level of radiation that a patient is exposed to). While not as demanding on the level of detail (usually only macrosize elements are considered in the diagnostic process), the overall quality of the 3D image is often too low for processing. Nevertheless, in both cases, feature extraction is vital, since the image edges and singularities contain the most important information about the structure and the properties of scanned objects. Those are exactly these features that are crucial for "correctly reading the image", and thus finding critical aspects of the material, or formulating the right diagnosis. Overall, regardless of the specific application area, processing 2D CT images that are to be turned into 3D volume reconstructions of the scanned object is a very challenging task, and an active research field. Here, research reported, indicates that every 2D CT radiographic frame is heavily polluted by a Poisson-dominated noise, which is nonadditive and data-depending. As far as we know, there are no simultaneously reliable and computationally efficient filters for dealing with such noise characteristics. This observation can be seen as an initial motivation for the detailed performance analysis of parallel high-resolution algorithms, presented in this article.

Poisson noise has been shown to be the dominant component in the noise characteristics of both industrial and medical CT radiographic data. It was also found that this noise is nonadditive, and exhibits mean/variance relationship, thus is data-dependent. Therefore, nonlinear filters have to be applied for successful noise removal, and this has to be done independently for each frame. In this article, we compare the performance of two different 3D CT image restoration algorithms, and their applicability for improved 3D volume reconstruction of high-resolution CT data, generated by an industrial Nikon XTH 225 scanner. We focus on industrial image processing, as the generated data is of large-size and the necessity of numerical efficiency (level of detail) is bigger than in the case of medical image processing. Nevertheless, results presented below can be immediately and directly applied to the latter case.

When considering the problem at hand, it has been established that when applying variance-stabilizing transformations (VST), such as the Anscombe transform, the Poisson noise can be approximated by a Gaussian one, for which classical denoising filters can be used. This being the case, both considered algorithms solve an Anscombe-transformed total variation (TV) constrained optimization problem that aims at generating noise-free versions of 2D CT radiographic images. The first algorithm was proposed in Reference 2. It allows for complete splitting of pixel data, and for their independent treatment within each iteration. For this algorithm, it was experimentally observed that the convergence rate of the algorithm heavily depends on both the image size, and the choice of the two free parameters σ and ρ, making the sequential realization of the algorithm impractical for large-scale industrial images. Therefore, in Reference 3 we have developed a hybrid parallel code based on joint application of MPI and OpenMP-based parallelization.[4-8] This approach was motivated by the need to maximize the parallel efficiency of the implementation of the proposed algorithm, in an attempt to make it practical. Note that the proposed approach was selected to match the target (super)computer architecture. The second considered algorithm solves a modified version of the same optimization problem directly in the Anscombe transformed domain, which may lead to a less reliable output, but at the same time, to a significant acceleration of the execution, due to performing smaller number of operations and dealing with less variables in the implementation.

This article is an extended version of a conference proceedings report, that can be found in Reference 9. Its aim is to present an extensive study of performance characteristics of both algorithms on a (super)computer cluster. While, today, focus of research seems to be on GPU-type architectures, we believe that this work may provide interesting guidelines when the developments in the area of high-performance computing bring back non-GPU approaches (or when GPUs become internalized within CPUs, which still will be clustered to deliver needed computing power). To the effect, we proceed as follows. In Section 2, we start with the description of the two considered algorithms and their parallelization. Next, we introduce the experimental setup in Section 4. Numerical tests that we have conducted are presented and discussed in Section 5. Finally, concluding remarks and future research directions are presented in the last Section 6.

2 | NOTATION AND ALGORITHM DESCRIPTION

Let us start with introducing the notation and detailed description of the two algorithms. In our work, we deal with reconstruction of 2D radiographic (gray-scale) images, generated by a CT scanner. The ground truth that we want to obtain, will be denoted by $\mathbf{w} \in [0, \omega]^{M \times N}$, where ω is the maximal gray-scale intensity, which throughout this article will be either $2^8 - 1 = 255$ for the 8-bit bitmap images, or $2^{16} - 1 = 65535$ for the 16-bit bitmap images. The size of the image will be denoted by $n = MN$. Note that we adopt the notation $\mathbf{w}$ also for the column-wise vector representation of the image ($\mathbf{w} \in [0, \omega]^n$). However, in each case it will be clear from the context, which format we consider. Finally, by $\mathbf{f}$ we denote a single-frame input image, while we use $\overline{\mathbf{w}}$ for the denoised output (resulting from the application of the corresponding algorithm).

If $\mathbf{w}_i \in [0, \omega]$ denotes the exact X-ray amount that should be detected at the pixel i, previous studies suggest that the actual amount of radiation per frame $\mathbf{f}$ is a realization of a Poisson random variable $\mathbf{f}_i = \mathscr{P}(w_i)$

$$Pr(\mathbf{f}_i = k) = e^{-w_i} \frac{w_i^k}{k!},$$

with the expected value w_i. Therefore, in our work we consider the solution of an ill-posed inverse problem of recovering an original 2D image $\mathbf{w}$ from noisy observations

$$\mathbf{f} = \mathscr{P}(\mathbf{w}). \tag{1}$$

For the Poisson distribution (1) it is straightforward that $\mathrm{E}(\mathbf{f}) = \mathrm{Var}(\mathbf{f}) = \mathbf{w}$, which gives rise to data-dependent and nonadditive noise characteristics, meaning that the noise pollution level of each pixel is proportional to the corresponding gray-scale intensity. Thus, unlike white noise removal, nonlinear and less-efficient filters are needed for this setup. As noted above, it is well known, that the Anscombe transform can be applied to the problem in question. It is defined as

$$T : [0, +\infty)^n \to (0, +\infty)^n : \mathbf{v} = (v_i)_{1 \le i \le n} \mapsto 2\left(\sqrt{v_i + \frac{3}{8}}\right)_{1 \le i \le n},$$

and belongs to the class of VST, which remove the above-mentioned mean-variance relation in Poisson distribution. Let us denote by $\mathscr{N}(\mu, \xi)$ the normal distribution with expectation μ and standard deviation ξ. When the noise is purely Poissonian, $\mathrm{T}(\mathbf{f}) \sim \mathscr{N}(\mathrm{T}(\mathbf{w}), 1)$, one can use least-squares techniques for removing the corresponding Gaussian component. Based on this observation, the mathematical model proposed in Reference 2 deals with the constrained convex optimization problem, which can be formulated as follows:

$$\overline{\mathbf{w}} = \underset{\mathbf{w} \in [0, \omega]^n}{\operatorname{argmin}} \| \nabla \mathbf{w} \|_{2,1} \quad \text{subject to} \quad \| T(\mathbf{w}) - T(\mathbf{f}) \|_2^2 \le \tau_A, \tag{2}$$

where $\nabla \in \mathbb{R}^{2n \times n}$ is the discrete gradient operator (forward differences and Neumann boundary conditions are used), and $\| \cdot \|_{2,1}$ denotes the TV seminorm, namely, the sum of the Euclidean lengths of the 2D gradient vectors. The cost function serves as a regularization term and its minimization leads to a smooth output, lacking the typical for noisy images fluctuations in pixel data. Previous investigations provided strong experimental evidence that the choice of TV seminorm as regularizer give rise to significantly better results than the corresponding pure ℓ^2 and ℓ^1 alternatives. The constrained set is an n-dimensional ball in the Anscombe domain, centered at $\mathrm{T}(\mathbf{f})$ with radius $\sqrt{\tau_A}$

$$C_A(\tau_A, \mathbf{f}) := \{ \mathbf{w} \ : \ \| T(\mathbf{w}) - T(\mathbf{f}) \|_2^2 \le \tau_A \}.$$

It serves as a data fidelity term and corresponds to an a priori estimate of $\mathrm{Var}(\mathrm{T}(\mathbf{f})) \approx \tau_A$. Ideally, when Equation (1) holds true, $\tau_A = n$ and the true solution $\mathbf{w}$ lies on the boundary of the constraint, then it becomes a potential candidate for the minimizer (it is a well-known fact that whenever

the cost function does not admit a global minimum in the constraint interior, the minimizer of the convex optimization problem (2) belongs to the constraint boundary).

Based on these definitions, we can now describe in some detail the two algorithms. The first of them has been originally proposed in Reference 2. It belongs to the class of primal-dual hybrid gradient algorithms, with modified dual variable (PDHGMp). It uses pixel-wise epigraphical projections to solve 2. In what follows, we use notation from Reference 2 for the closed half-space V_n and the proximal operator $\text{prox}_{\sigma^{-1}\|\cdot\|_{2,1}}$. Specifically:

$$V_n := \{\zeta \in \mathbb{R}^n : \langle \mathbf{1}_n, \zeta \rangle \leq n\}$$

$$\mathbf{w} = \text{prox}_{\sigma^{-1}\|\cdot\|_{2,1}}(\mathbf{v}), \qquad \text{where} \qquad \mathbf{w}_i = \max\left\{1 - \frac{\sigma^{-1}}{|\mathbf{v}_i|}, 0\right\}\mathbf{v}_i.$$

Thus, following Reference 2, the steps of the first algorithm can be summarized as:

Algorithm: 1

Initialization:

$\mathbf{w}^{(0)}, \boldsymbol{\xi}^{(0)}, (\mathbf{p}_j^{(0)})_{1\leq j\leq 3} = (\overline{\mathbf{p}}_j^{(0)})_{1\leq j\leq 3}, \rho > 0, \sigma > 0, \rho\sigma < 1/9.$

For $k = 0, 1, \ldots$ repeat until a stopping criterion is reached

1. $\mathbf{w}^{(k+1)} = \max\{\min\{(\mathbf{w}^{(k)} - \sigma\rho(\overline{\mathbf{p}}_1^{(k)} + \nabla^* \overline{\mathbf{p}}_2^{(k)})), \omega\mathbf{1}_n\}, \mathbf{0}\}$
2. $\boldsymbol{\xi}^{(k+1)} = P_{V_n}(\boldsymbol{\xi}^{(k)} - \sigma\rho\overline{\mathbf{p}}_3^{(k)})$
3. $(v_{1,i}, \eta_i) = P_{\text{epi}\varphi_i}(p_{1,i}^{(k)} + (\mathbf{w}^{(k+1)})_i + 3/8, p_{3,i}^{(k)} + \zeta_i^{(k+1)}), \quad i = 1, \ldots, n$
4. $\mathbf{v}_2 = \mathbf{p}_2^{(k)} + \nabla\mathbf{w}^{(k+1)}$
5. $\mathbf{p}_1^{(k+1)} = \mathbf{p}_1^{(k)} + \mathbf{w}^{(k+1)} + 3/8 - \mathbf{v}_1$
6. $\mathbf{p}_2^{(k+1)} = \mathbf{v}_2 - \text{prox}_{\sigma^{-1}\|\cdot\|_{2,1}}(\mathbf{v}_2)$
7. $\mathbf{p}_3^{(k+1)} = \mathbf{p}_3^{(k)} + \boldsymbol{\xi}^{(k+1)} - \boldsymbol{\eta}$
8. $\overline{\mathbf{p}}_j^{(k+1)} = \mathbf{p}_j^{(k+1)} + (\mathbf{p}_j^{(k+1)} - \mathbf{p}_j^{(k)}), \quad j = 1,2,3.$

Here, recall that the epigraph of a function $h : \mathbb{R}^m \to \mathbb{R}$ is defined as the set

$$\text{epi}\, h := \{(x, \zeta) \in \mathbb{R}^m \times \mathbb{R} : h(x) \leq \zeta\},$$

that is, the set that consists of all points in $\mathbb{R}^{m+1}$, lying above the graph of the function h. When h is a convex function, its epigraph is a convex set, and we can define an *epigraphical projection* of a point $y \in \mathbb{R}^{m+1}$ onto epi h as the closest point $z \in$ epi h of y with respect to the standard Euclidean distance. Note that such point exists and is unique, due to the convexity of the epigraph. In our particular case, we need to compute the orthogonal projection onto the epigraph of the Anscombe-related convex function $\varphi \in \Gamma_0(\mathbb{R})$ defined as:

$$\varphi(s) := \begin{cases} (2\sqrt{s} - z)^2 & \text{if } s \geq 0, \\ +\infty & \text{otherwise,} \end{cases} \tag{3}$$

where $z > 0$ is a parameter. According to [2 proposition 1], it is given by:

$$P_{\text{epi}\varphi}(x, \zeta) = \begin{cases} (\max\{x, 0\}, \zeta) & \text{if } \varphi(\max\{x, 0\}) \leq \zeta, \\ \left(\left(\frac{t_+ + z}{2}\right)^2, t_+^2\right) & \text{if } \varphi(\max\{x, 0\}) > \zeta \text{ and } 4x \geq z^2, \\ \left(\left(\frac{t_- + z}{2}\right)^2, t_-^2\right) & \text{if } \varphi(\max\{x, 0\}) > \zeta \text{ and } 4x < z^2, \end{cases}$$

where t_+, resp. t_- is the unique root, in $[0, +\infty)$, resp. in $(-z, 0)$, of the cubic polynomial

$$p : t \mapsto 17t^3 + 3zt^2 + (3z^2 - 16\zeta - 4x)t + z(z^2 - 4x). \tag{4}$$

In practice, we solve step 3 from Algorithm 1 via applying pixel-wise the Newton method for (4) with $z = 3/8$ and $t_0 = T(p_{1,i}^{(k)} + (\mathbf{w}^{(k+1)})_i + 3/8))$. We observe that for $k > 10$ only one Newton iteration per pixel suffices for approximating $v_{1,i}$ with accuracy 10^{-6}.

There are other classes of algorithms that can be used for solving (2). The main advantage of Algorithm 1 is that it can be straightforwardly modified to deal with more general setups, such as mixed Poisson–Gaussian noise characteristics, introduction of additional Gaussian blur artifact to the input data, or multiframe image denoising. The cost for universality is that such approach for obtaining the solution doubles the dimensionality of the problem, as to each individual pixel value we relate a 2D point from epiφ, instead of a scalar. As what concerns practical aspects of the implementation of the resulting numerical algorithm, the auxiliary variables ζ, η (primal), p_3 and $\overline{p}_3$ (dual) are introduced in Steps 2–3 and 7–8, respectively. Obviously, their appearance increases the memory usage and the total number of operations. Moreover, it slows down the speed of convergence of the algorithm, thus making it less efficient, as shown in the analysis of numerical experiments, reported in Reference 2. The main reason of slow convergence is that the epigraphical projections in Step 3 are quite restrictive on the changes of $\mathbf{w}^{(k)}$ within an iteration, which prevents the latter to quickly reach the desired steady state.

A potential alternative approach is to slightly modify the original problem as in Reference 10, by replacing the regularization term $\| \nabla w\|_{2,1}$ by $\| \nabla T(w)\|_{2,1}$. Since the Anscombe transform is strictly monotonically increasing, such replacement makes sense. It allows one to work solely in the transformed domain, where the projection onto the constrained set is straightforward. In other words, we reformulate the problem into:

$$\underset{T(w)\in[T(0),T(\omega)]^n}{\operatorname{argmin}} \| \nabla T(\mathbf{w})\|_{2,1} \quad \text{subject to} \quad \|T(\mathbf{w}) - T(\mathbf{f})\|_2^2 \le \tau_A, \tag{5}$$

and apply the following PDHGMp algorithm:

Algorithm 2:

Input: $\mathbf{f}$, σ, ρ—free parameters (accelerators) with $\sigma\rho < 1/9$.

I. Preprocessing: $\mathbf{f}_A = T(\mathbf{f})$ and $\tau_A = n$.

II. Initialization: $\mathbf{w}_A^{(0)} = 0$, $\overline{\mathbf{p}}_1^{(0)} = \mathbf{p}_1^{(0)} = 0$, and $\overline{\mathbf{p}}_2^{(0)} = \mathbf{p}_2^{(0)} = 0$.

III. For $k = 0, 1, \ldots, K$

1. $\mathbf{w}_A^{(k+1)} = P_{[T(0),T(\omega)]}\left(\underbrace{\mathbf{w}_A^{(k)} - \sigma\rho\left(\overline{\mathbf{p}}_1^{(k)} + \nabla^*\overline{\mathbf{p}}_2^{(k)}\right)}_{x}\right) = \min\left(\max\left(x, T(0)\right), T(\omega)\right)$

2. $v_1^{(k+1)} = P_{B_N(\mathbf{f}_A,\sqrt{\tau_A})}\left(\mathbf{w}_A^{(k+1)} + \mathbf{p}_1^{(k)}\right) = \begin{cases} \mathbf{w}_A^{(k+1)} + \mathbf{p}_1^{(k)}, & \text{if} \|\mathbf{w}_A^{(k+1)} + \mathbf{p}_1^{(k)} - \mathbf{f}_A\|_2 \le \sqrt{\tau_A} \\ \frac{\sqrt{\tau_A}\left(\mathbf{w}_A^{(k+1)} + \mathbf{p}_1^{(k)} - \mathbf{f}_A\right)}{\|\mathbf{w}_A^{(k+1)} + \mathbf{p}_1^{(k)} - \mathbf{f}_A\|_2} + \mathbf{f}_A, & \text{otherwise} \end{cases}$

3. $\mathbf{v}_2 = \mathbf{p}_2^{(k)} + \nabla\mathbf{w}_A^{(k+1)}$

4. $\mathbf{p}_1^{(k+1)} = \mathbf{p}_1^{(k)} + \mathbf{w}_A^{(k+1)} - \mathbf{v}_1^{(k+1)}$

5. $\mathbf{p}_2^{(k+1)} = \mathbf{v}_2 - \text{prox}_{\sigma^{-1}\|\cdot\|_{2,1}}(\mathbf{v}_2)$

6. $\overline{\mathbf{p}}_i^{(k+1)} = \mathbf{p}_i^{(k+1)} + \left(\mathbf{p}_i^{(k+1)} - \mathbf{p}_i^{(k)}\right)$, $i = 1, 2$.

IV. Postprocessing: $\overline{\mathbf{w}} = T^{-1}\left(\mathbf{w}_A^{(K)}\right)$.

Such a reformulation allows us to replace steps 2 and 3 from Algorithm 1 with step 2 from Algorithm 2 and also to omit step 7, that is, instead of computing/approximating pixel-wise epigraphical projections we apply classical ℓ^2-projection onto a ball with center $\mathbf{f}_A$ and radius $\sqrt{\tau_A}$. In the postprocessing step of this algorithm, the inverse Anscombe transform T^{-1} is used. It has the following form:

$$T^{-1} : [0, +\infty)^n \to (0, +\infty)^n : \{v_i\}_{1\le i\le n} \mapsto \left\{\frac{(v_i)^2}{4} - \frac{3}{8}\right\}_{1\le i\le n}.$$

Comparing the two considered algorithms it is clear that Algorithm 2 is more computationally efficient than Algorithm 1, since it does not use any auxiliary variables (lower operation count) and no additional restrictions, such as epigraphical projections, are applied to the data. The main drawback of Algorithm 2 is the explicit use of the, theoretically unstable, inverse Anscombe transform T^{-1}, which Algorithm 1 avoids. However, it is also known that the main problems with T^{-1} are in the neighborhood of zero. However, typical CT data should be penetrable (i.e., $\min\mathbf{f} \ge 1$; otherwise the zero-valued pixels contain no useful information whatsoever and can be disregarded during the image reconstruction process). Thus, Algorithm 2 is a legitimate choice for CT data image reconstruction. In practice, Algorithm 2 uses a projection to the interval $[T(0), T(\omega)]$, so that every pixel in the final approximation $\mathbf{w}_A^{(K)}$ has value bigger than $\sqrt{\frac{3}{2}}$, which is sufficient to avoid potential stability problems.

3 | PARALLELIZATION

3.1 | Multithread-based parallel implementation

In the current parallel implementation, we use OpenMP for the matrix by vector multiplication and all vector operations. We have compiled our code using the OpenMP library and linked it to the multithreaded library for the parallelization on a multicore node of the computer system.

3.2 | Multinode implementation

It is a well-known fact that to achieve an efficient parallel, multinode implementation of any algorithm, one has to minimize the communication between nodes. To this effect, we partition the image into m rectangles so that each rectangle contains approximately MN/m pixels. We map all pixels from a given rectangle into a single computing node. In this way, the Newton's method is performed in parallel for each pixel and does not need any communication. The same is true for the scalar multiplication and vector sums. The computation of the discrete gradient operator requires only M values from the next node (see Reference 3 for details). In Algorithm 1, only the computation of the projection onto V_n requires a single global communication in Step 2. In Algorithm 2, only the computation of the projection onto the ball B_N requires a computation of an Euclidian norm of a vector and again a single global communication is used in Step 2. Here, the MPI function MPI_Allreduce was used to compute the inner product in both algorithms.

4 | EXPERIMENTAL SETUP

Let us now report on the experiments that have been performed with the parallel implementation of the two algorithms. A portable parallel code was implemented in C. As outlined above, the hybrid parallelization is based on joint application of the MPI and the OpenMP standards.

In our experiments, we used standard approach to performance measurement. Thus, times were collected using the MPI provided timer, and we report the average wall-clock time from multiple runs. However, it should be stressed that, across *all* experiments, we have not observed any significant time variance. In what follows, we report the average elapsed time T_p, where p the number of all threads (m MPI processes and every process uses k OpenMP threads). Then, in what follows, we report the parallel speed-up $S_p = T_1/T_p$ and the parallel efficiency $E_p = S_p/p$. Finally, we report performance in terms of Flops. Since this performance measure is not available explicitly in modern supercomputers (while it was explicitly reported in, for instance, Cray Y-MP; see Reference 11), we had to estimate it. Hence, for selected runs, we have calculated the number of completed floating point operations and divided it by the execution time.

We have tested the parallel algorithm on images obtained from the Tomograph XTH 225 Compact industrial CT scanning. The images that we have experimented with had sizes 723×920 and 1446×1840 pixels. In addition, in order to more comprehensively study the performance of the developed algorithm, we applied it to a "transposed image," with size 1840×1446 pixels. In the tables, the size of the image is denoted by $M \times N$.

The parallel code has been tested on the supercomputer Avitohol at IICT-BAS (see http://www.hpc.acad.bg/ and References 3,12 for details). Table 1 resume the important information about hardware and software on the Avitohol.

TABLE 1 Hardware and software on the Avitohol

Number of processors	2
Processor	Intel Xeon E5-2650 v2
Number of cores within one processor	8
Maximal number of threads on one processor	16
Main memory per node	64 GB
Number of coprocessors	2
Coprocessor	Intel Xeon Phi 7120P
Number of cores within one coprocessor	61
Maximal number of threads on one coprocessor	244
Memory of one coprocessor	16 GB
Compiler	Intel C
Compiler options for processors	-O3 -qopenmp
Compiler options for coprocessors	-O3 -qopenmp -mmic
MPI library	Intel MPI

5 | EXPERIMENTAL RESULTS

Let us start from results obtained when using only Intel Xeon processors. Here, Table 2 depicts results obtained on a single node of Avitohol (varying number of threads) while Table 3 present times collected on multiple nodes. As expected, when using only processors available within a single one node, the best results are obtained using (the maximum available) 32 OpenMP threads. We gain from the effect of hyperthreading for all image sizes, used in this set of experiments. However, parallel performance does not improve with problem size. As far as parallel efficiency is concerned, we observe quite interesting results. For Algorithm 1, speed-up for the small problem is almost 10, while for the large problem it is only 8. However, for Algorithm 2, 32-thread speed-up is 6.7 for the small case, and 7.9 in the large one. Finally, matching the estimated number of operations, Algorithm 2 is two times faster, in all experiments.

TABLE 2 Time for 40,000 iterations of both algorithms using shared memory parallelism on one node

		k					
M	*N*	1	2	4	8	16	32
Algorithm 1							
723	920	2179.60	1717.66	903.49	505.26	333.20	232.29
1446	1840	8611.48	7183.28	3711.57	2088.34	1408.84	1182.71
1840	1446	8614.31	7194.07	3745.31	2121.20	1435.31	1111.03
Algorithm 2							
723	920	972.14	928.62	487.99	282.38	194.56	143.16
1446	1840	4689.79	4541.24	2369.99	1326.96	928.02	729.60
1840	1446	4556.96	4472.26	2315.80	1319.88	927.88	579.12

TABLE 3 Time for 40,000 iterations of both algorithms using distributed memory parallelism on many nodes

		Nodes					
M	*N*	2	3	4	5	6	8
Algorithm 1							
$k = 16$							
723	920	153.00	**92.46**	**60.44**	**46.92**	**39.44**	**30.95**
1446	1840	700.69	**462.50**	**342.79**	**276.55**	**218.02**	**157.52**
1840	1446	712.73	**468.08**	**346.28**	**275.77**	**219.76**	**158.58**
$k = 32$							
723	920	**137.67**	**103.58**	**93.06**	**93.47**	**87.75**	**85.84**
1446	1840	**617.40**	**465.74**	**390.46**	**343.30**	**312.82**	**261.77**
1840	1446	**676.37**	**475.63**	**391.69**	**351.24**	**288.26**	**249.03**
Algorithm 2							
$k = 16$							
723	920	88.08	**48.09**	**33.46**	**26.99**	**23.13**	**18.62**
1446	1840	448.36	**289.90**	**212.83**	**174.52**	**135.15**	**97.10**
1840	1446	447.42	**288.40**	**211.35**	**173.05**	**134.34**	**95.35**
$k = 32$							
723	920	**76.90**	**64.38**	**59.02**	**57.99**	**61.50**	**64.61**
1446	1840	**418.17**	**298.22**	**248.49**	**211.75**	**181.62**	**160.36**
1840	1446	**444.15**	**300.20**	**234.66**	**220.40**	**187.18**	**159.63**

TABLE 4 The used memory (in KB) by both algorithms

		Nodes						
M	*N*	1	2	3	4	5	6	8
Algorithm 1								
723	920	184,500	132,884	133,584	128,092	130,504	123,372	124,216
1446	1840	457,068	273,888	223,412	199,748	180,936	173,724	170,736
1840	1446	456,812	283,660	220,220	206,072	185,188	174,132	168,728
Algorithm 2								
723	920	160,652	120,508	119,596	111,560	117,440	114,920	121,604
1446	1840	368,364	239,348	188,160	172,052	165,944	154,684	146,684
1840	1446	376,264	235,260	188,500	172,028	169,060	159,872	144,564

TABLE 5 Speed-up using only processors

		Cores									
M	*N*	2	4	8	16	32	48	64	80	96	128
Algorithm 1											
723	920	1.27	2.41	4.31	9.38	15.76	23.57	36.06	46.45	55.26	70.36
1446	1840	1.20	2.32	4.12	7.51	13.95	18.62	25.12	31.11	39.50	54.67
1840	1446	1.20	2.30	4.06	7.99	12.68	19.28	24.88	31.21	39.20	54.32
Algorithm 2											
723	920	1.05	1.99	3.51	7.11	12.64	20.21	29.05	36.02	42.03	52.20
1446	1840	1.03	1.98	3.54	6.44	11.24	16.18	22.04	26.87	34.70	48.30
1840	1446	1.02	1.97	3.45	7.87	10.26	15.80	21.56	26.33	33.92	47.79

Table 4 shows the memory used by both algorithms on the number of nodes up to 8.

The execution time on 2 and up to eight nodes (using only processors) is presented for 16 and 32 threads. Here, a slightly different results can be observed. We used bold numbers to mark the better performance, when varying the number of OpenMP threads, for the same number of nodes. The first observation is that on two nodes the effect of hyperthreading improve the performance for all image sizes (for both algorithms). In other words, when two nodes are used, then using 32 threads leads to best execution times. When the number of nodes is larger than 2, the better performance is obtained when using 16 OpenMP threads per node. However, when looking carefully into the data, one can easily note that, for 32 threads, for Algorithm 1, speed-up between two and eight nodes is 1.6 for the small images, while reaching 2.7 for the large ones. For Algorithm 2, this is even more pronounced. For the small problem speed-up is 1.2, while for the large images, speed-up increases to 2.7. This indicates that, in some way, the image sizes that we experimented with are "too small" to use *both* more than two nodes and 32 threads. Finally, considering 32 thread performance on eight nodes, for the small image size, Algorithm 2 is 1.3 times faster, while for the large images, it becomes 1.6 times faster. This would, again, indicate that both algorithms are constrained by the image size and as image size increases their inherent expected two times operation count difference reveals itself.

To provide somewhat more detailed insight into the performance of the parallel versions of both algorithms, using only processors of the Avitohol, the measurement of the strong scalability of the algorithms is reported in Table 5. This time the speed-up is captured from the point of view of the number of cores.

Here, we observe that the speed-up decreases with image size; from 70 to 54 for Algorithm 1, and from 52 to 47 for Algorithm 2. This is caused by the better usage of cache memories. The best parallel efficiency, of 54% can be observed for the small image, when Algorithm 1 is applied. This result, however, needs to be confronted with the fact that Algorithm 1 takes (almost twice) more time to execute (see Table 3).

Let us now move from processor-only performance to the situation when the coprocessors are used. Here, times using only coprocessors are presented in Tables 6 and 7.

TABLE 6 Time for 40,000 iterations of both algorithms using shared memory parallelism on one coprocessor

		k					
M	***N***	**1**	**8**	**60**	**120**	**240**	**244**
Algorithm 1							
723	920	18,376.11	5183.35	741.75	389.94	222.15	222.21
1446	1840	83,409.50	27,946.75	3862.43	2011.66	1101.20	1088.28
1840	1446	83,552.88	27,884.25	3860.97	2008.20	1102.52	1089.72
Algorithm 2							
723	920	9523.57	2925.11	396.19	239.44	151.26	132.07
1446	1840	47,076.40	17,566.23	2458.70	1298.40	708.24	695.74
1840	1446	47,138.40	17,651.65	2442.48	1284.65	704.40	696.54

TABLE 7 Time for 40,000 iterations of both algorithms using only coprocessors

		Nodes						
M	***N***	**1**	**2**	**3**	**4**	**5**	**6**	**8**
Algorithm 1								
$k = 200$								
723	920	148.03	94.21	77.51	66.39	63.65	59.40	53.95
1446	1840	643.77	310.39	222.66	182.21	159.11	143.54	123.10
1840	1446	639.40	303.77	214.85	173.98	149.33	138.72	114.50
$k = 240$								
723	920	132.89	**85.41**	**69.96**	**62.63**	**59.85**	**55.49**	**52.07**
1446	1840	555.25	277.51	200.58	165.41	145.47	131.83	113.85
1840	1446	551.06	267.97	193.34	156.14	135.18	127.49	103.76
$k = 244$								
723	920	**130.92**	**85.68**	**70.14**	**62.86**	**60.26**	**55.97**	**52.03**
1446	1840	**550.13**	**273.74**	**198.74**	**163.64**	**144.14**	**130.72**	**113.42**
1840	1446	**541.38**	**265.48**	**190.88**	**155.05**	**134.37**	**125.94**	**103.52**
Algorithm 2								
$k = 200$								
723	920	95.34	63.60	54.76	48.03	47.67	45.84	42.32
1446	1840	407.30	201.23	152.69	127.05	114.05	105.86	92.06
1840	1446	413.97	195.12	143.57	119.81	104.90	97.61	84.50
$k = 240$								
723	920	85.42	58.76	50.32	**46.76**	**45.94**	**43.35**	**41.80**
1446	1840	366.21	182.57	136.76	117.77	105.44	97.24	86.46
1840	1446	351.43	173.44	131.59	108.55	97.03	88.71	77.14
$k = 244$								
723	920	**84.73**	**58.37**	**50.01**	**46.90**	**45.94**	**43.10**	**41.58**
1446	1840	**352.89**	**180.59**	**135.71**	**116.18**	**104.72**	**95.78**	**86.28**
1840	1446	**348.52**	**172.22**	**130.78**	**107.81**	**95.66**	**87.48**	**77.07**

TABLE 8 Speed-up using only coprocessors

M	*N*	*p*					
Algorithm 1							
		8	60	120	240	244	488
723	920	3.55	24.77	47.13	82.72	82.70	140.37
1446	1840	2.98	21.60	41.46	75.74	76.64	151.62
1840	1446	3.00	21.64	41.61	75.78	76.67	154.33
		976	1464	1952	2440	2928	3904
723	920	214.47	262.00	292.33	304.95	328.34	353.19
1446	1840	304.71	419.70	509.71	578.67	638.09	735.37
1840	1446	314.73	437.72	538.87	621.79	663.44	807.09
Algorithm 2							
		8	60	120	240	244	488
723	920	3.26	24.06	39.81	63.02	71.94	112.43
1446	1840	2.68	19.15	36.26	66.47	67.55	132.30
1840	1446	2.67	19.30	36.76	66.94	67.58	135.46
		976	1464	1952	2440	2928	3904
723	920	163.38	190.53	203.93	207.58	221.37	229.42
1446	1840	260.36	346.92	404.66	449.35	491.65	546.02
1840	1446	274.19	360.33	436.82	492.35	539.25	611.38

As expected, in general, the best performance on one coprocessor (Table 6) is obtained using the maximal available number of threads ($k = 244$). For Algorithm 1, speed-up using 244 threads is 83 for the small image, while it drops to 77 for large images. For Algorithm 2, speed-up drops from 72 for small images to 68 for large ones. Again, Algorithm 2 is almost two times faster across all results.

Let us now consider results presented in Table 7. Here, we depict performance obtained on one to eight nodes, for 200, 240, and 244 threads. In bold we have presented best results for each combination of number of nodes and number of threads, for Algorithms 1 and 2, separately. As it can be seen, similarly to the results presented in Table 6, for big images, the best performance is obtained for 244 threads. Here, Algorithm 1 is 1.3 times slower than Algorithm 2 (for the largest number of nodes and threads).

For small images there are substantial differences between the considered algorithms. For two to six nodes, the execution of the Algorithm 1 is faster when using 240 OpenMP threads. For Algorithm 2, the only exception is the case of four nodes.

Comparing the results presented in Tables 2, 3, and 7 one can observe that for big images, both considered algorithms run faster using only coprocessors, on the same number of nodes. For small images again the behavior of the algorithms is slightly different. Here, Algorithm 1 runs faster using coprocessors on up to three nodes, while Algorithm 2 runs faster using processors except the execution on one and two nodes.

Finally, the strong scalability obtained on coprocessors is presented in Table 8. Here, one can see that the obtained parallel efficiency is better for the "transposed" images.

Results in Table 8 are presented in terms of number of threads. The best speed-up was obtained for Algorithm 1, for the large transposed image (807). However, again, this has to be considered together with the fact that Algorithm 1 takes more time to complete (see, Table 7). Interestingly, speed-up with the transposed large image is substantially larger than for the standard one. This is caused by the approach used for the parallelization of both algorithms for distributed memory computers. In the current version of the parallel implementation, the image is divided into vertical strips. Thus, using distributed memory the performance of both algorithms is better for the transposed images.

Let us now consider the situation when we use a hybrid approach and run the codes jointly on processors and coprocessors. The average time using processors as well as coprocessors is shown in Table 9, for the same setup as before.

Comparing results in Tables 2, 3, and 9 it can be seen that, for the small images, there is an improvement in the performance only when using one to four nodes. For the large images the considered algorithms have from two to three times better performance when using both processors and coprocessors, compared with the performance using only processors, on up to eight nodes.

TABLE 9 Time for 40,000 iterations of both algorithms on processors and coprocessors

		nodes						
M	*N*	1	2	3	4	5	6	8
Algorithm 1								
723	920	105.88	69.52	57.33	51.08	49.78	47.84	46.71
1446	1840	431.84	224.16	158.54	126.98	109.73	103.94	86.55
1840	1446	425.61	225.80	168.47	125.12	102.26	102.59	80.97
Algorithm 2								
723	920	62.57	48.85	42.33	39.49	40.23	38.98	38.72
1446	1840	277.76	146.13	108.56	89.23	81.40	74.51	65.93
1840	1446	271.22	141.57	110.37	83.27	74.37	69.93	59.25

TABLE 10 The number MPI processes and the number of threads for all processes used for the experiments depicted in Table 9, where $m_c \times k_c + m_\varphi \times k_\varphi$ means m_c processes on processors, k_c threads on processors, m_φ processes on coprocessors, k_φ threads on coprocessors

		Nodes						
M	*N*	1	2	3	4	5	6	8
Algorithm 1								
723	920	1×16+	2×16+	6×8+	8×8+	10×8+	12×8+	16×8+
		2×240	4×240	6×244	8×240	10×240	12×240	16×240
1446	1840	1×32+	2×16+	3×16+	4×16+	5×16+	6×16+	16×8+
		2×244	4×240	6×240	8×244	10×244	12×244	16×244
1840	1446	1×32+	2×16+	3×16+	4×16+	5×16+	6×16+	16×8+
		2×240	4×244	6×240	8×240	10×244	12×240	16×244
Algorithm 2								
723	920	1×16+	2×16+	6×8+	8×8+	10×8+	12×8+	16×8+
		2×240	4×240	6×240	8×240	10×244	12×240	16×240
1446	1840	1×32+	2×16+	3×16+	4×16+	5×16+	12×8+	16×8+
		2×244	4×244	6×244	8×244	10×244	12×244	16×244
1840	1446	1×32+	2×16+	3×16+	4×16+	5×16+	12×8+	16×8+
		2×244	4×244	6×240	8×244	10×244	12×244	16×244

Here, again, we see that the "transposed" image takes less time to be processed that the standard one. Moreover, Algorithm 1 is 1.3 slower for both large images. Finally, the speed-up when going from one to eight nodes is 5 for Algorithm 1 and 4.2 for Algorithm 2.

To better understand presented results, Table 10 summarizes the number of MPI processes and the number of threads for all processes used for the experiments depicted in Table 9.

Here it can be seen that for all image sizes, and number of nodes, the best results are obtained using 240 or 244 threads on coprocessors. At the same time, for all images, the best performance is observed using 16 OpenMP threads on two to eight processors. Considering the numbers of threads leading to best performance, there is no substantial difference between Algorithms 1 and 2.

To bring about more complete picture of potential efficiency of different approaches to parallelization, in Figure 1, we summarize the execution times of both algorithms using only shared memory parallelism provided by the OpenMP standard. Here, we have not run experiments on processors with more than 32 threads, as it is clear that potential for further time reduction has been exhausted. Overall, as it was expected, Algorithm 2 is faster because it has lower computational complexity. The remaining observations, stated above, apply also to the results obtained for shared memory OpenMP parallelization. We have compared the performance of the parallel algorithms using only processors, only coprocessors, and using both processors and coprocessors. The average time obtained on up to eight nodes for the three image sizes is shown in Figure 2. Again,

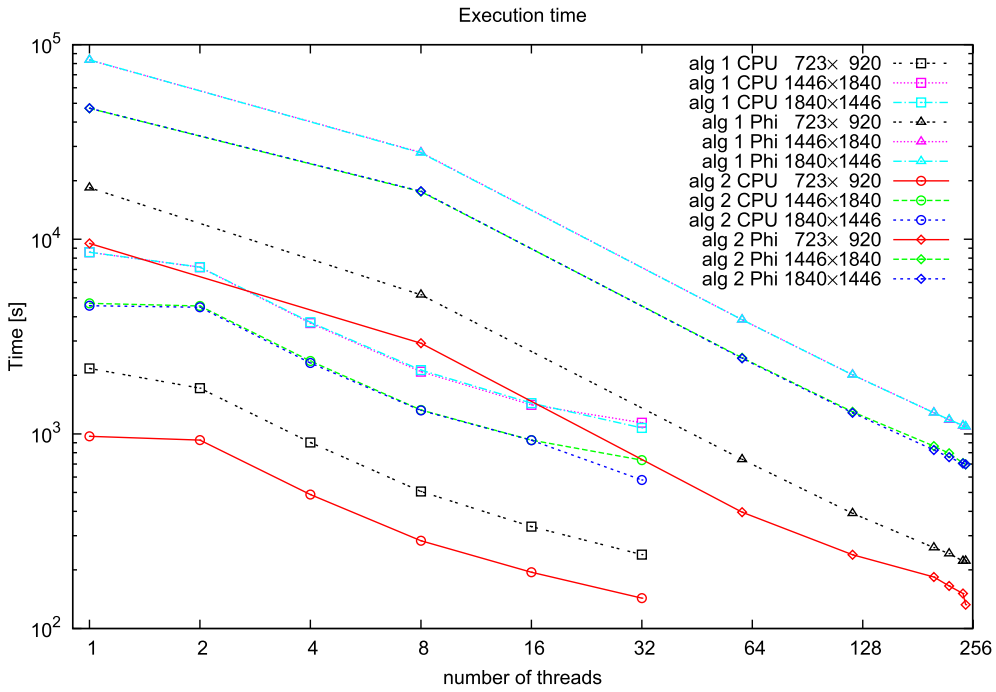

FIGURE 1 Execution time using one MPI process for various image sizes

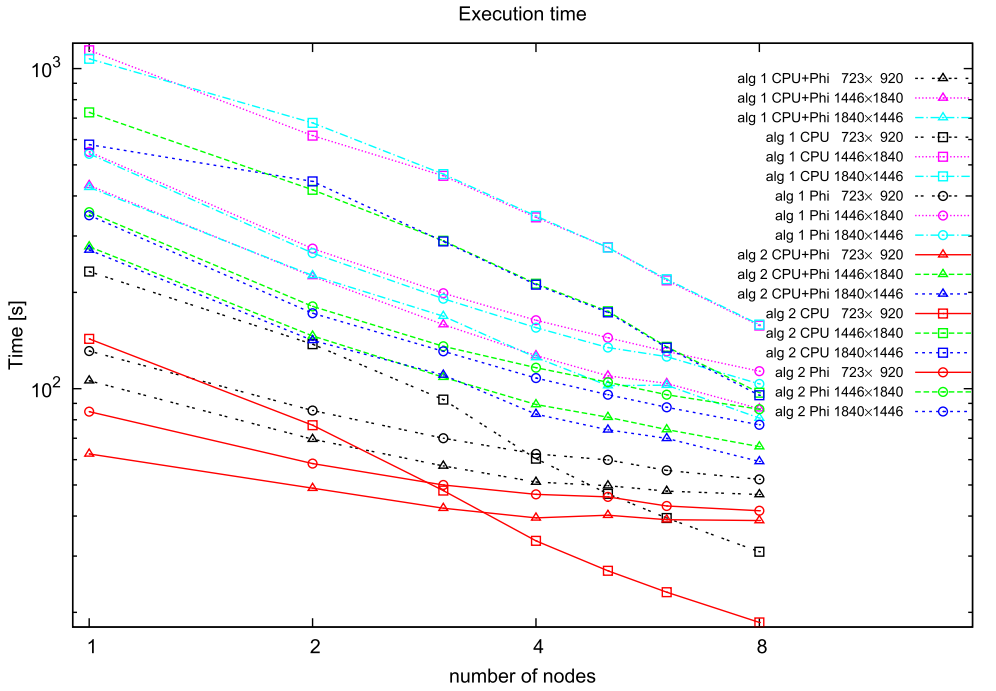

FIGURE 2 Execution time for various image sizes

from Figure 2 it can be seen that Algorithm 2 is much faster. Comparing the execution times one can see that for large images there is an essential improvement when running both algorithms using processors as well as coprocessors. There exists one more performance measure that can be used to evaluate efficiency of algorithms. This is their "raw speed," measured in terms of number of floating point operations per second. Such performance measure complements the results presented above. Hence, in Figure 3 we present the Flop execution rates of both algorithms for one of the input images.

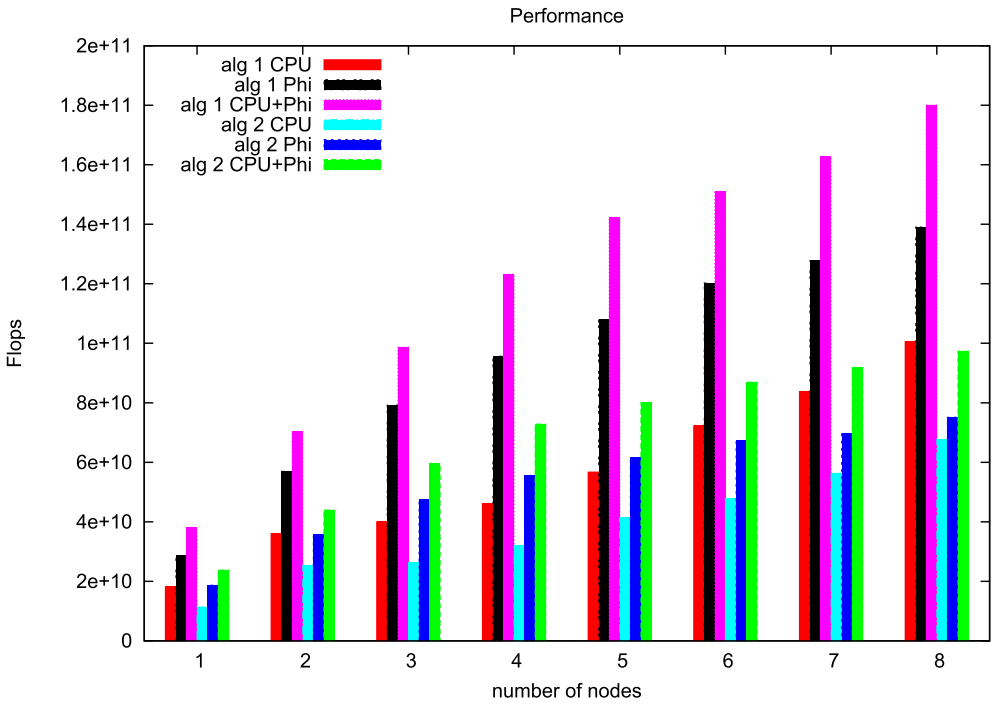

FIGURE 3 Performance of both algorithms for one image

6 | CONCLUDING REMARKS AND FUTURE WORK

We have studied the efficiency of the parallel implementation of image restoration algorithms, based on least squares techniques. Numerical experiments were conducted on a supercomputer, using Intel Xeon processors, as well as Intel Xeon Phi coprocessors. As it was expected, for big images, using only processors on one and two nodes of the system, the best results are obtained using the maximal available number of threads. In addition, for both sizes of the images, and for both algorithms, the best results on coprocessors were obtained using the maximal (or almost maximal) number of available threads. Finally, the experimental results show an essential improvement when running codes using *both* processors and coprocessors, for a variety of image sizes and number of threads.

In the current version of our parallel implementation of the considered algorithms, the image is divided into strips. The sizes of all strips are almost the same. During the experiments it was seen that, for small images on more than three nodes, our parallel algorithms run faster using only processors, compared with the results using only coprocessors.

Separately, let us add that we are aware of the fact that, in 2020, use of Intel high-performance (co)processors may seem like a step in the wrong direction. However, even today, not all high-performance computing is done using (NVIDIA) GPUs. For instance, the fastest TOP500 computer in the world of June 2020 is based on ARM processors[13]. Moreover, high-performance computing is dynamically exploring various directions (including, for instance, heterogeneous hardware designs, like the GVirtuS,[14] or domain-specific hardware accelerators[15]). Overall, since the beginning of parallel computing (see, for instance, https://ieeexplore.ieee.org/abstract/document/1675993), the ultimate goal was (and is) to match the problem, with the algorithm(s) than can be used to solve it, with the characteristics (e.g., it size) of the data that is to be processed, with the architecture of available computer systems, and with user requirements. This methodological approach provided context to our work and brought about questions that we have tried to address. However, recognizing the importance of GPU-based computing, we plan to redesign our approach to parallelization, and port both algorithms to GPUs.

ACKNOWLEDGMENTS

The partial support by grant KP-06-N27/6 from the Bulgarian NSF is acknowledged. This work has been accomplished with the partial support by the Grant No BG05M2OP001-1.001-0003, financed by the Science and Education for Smart Growth Operational Program (2014–2020) and cofinanced by the European Union through the European structural and Investment funds. This research was part of the collaboration agreement between Bulgarian Academy of Sciences and Polish Academy of Sciences "Practical aspects of scientific computing."

ORCID

Ivan Lirkov https://orcid.org/0000-0002-5870-2588

REFERENCES

1. Harizanov S. *Extracting Trustful Structural Information from High-Resolution CT Data via Pattern Recognition*. Bochum, Germany: Ruhr University Bochum; 2016.
2. Harizanov S, Pesquet JC, Steidl G. Epigraphical projection for solving least squares Anscombe transformed constrained optimization problems. In: Kuijper A, Bredies K, Pock T, Bischof H, eds. *Scale-Space and Variational Methods in Computer Vision. 7893 of Lecture Notes in Computer Science*. Lecture Notes in Computer Science7893. Berlin, Heidelberg: Springer; 2013:125-136. https://doi.org/10.1007/978-3-642-38267-3_11.
3. Harizanov S, Lirkov I, Georgiev K, Paprzycki M, Ganzha M. Performance analysis of a parallel algorithm for restoring large-scale CT images. *J Comput Appl Math*. 2017;310:104-114. https://doi.org/10.1016/j.cam.2016.07.001.
4. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, Mc Donald J. *Parallel Programming in OpenMP*. San Francisco, CA : Morgan Kaufmann Publishers; 2000. https://www.amazon.com/Parallel-Programming-OpenMP-Rohit-Chandra/dp/1558606718.
5. Chapman B, Jost G, Van Der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming. 10 of Scientific and Engineering Computation Series*. Cambridge, England: The MIT Press; 2008. https://www.amazon.com/Using-OpenMP-Programming-Engineering-Computation/dp/0262533022.
6. Gropp W, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, Massachusetts London, England: The MIT Press; 2014. https://www.amazon.com/Using-MPI-Programming-Message-Passing-Engineering/dp/0262527391/ref=pd_lpo_14_t_0/143-6398474-8604444.
7. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. *MPI: The Complete Reference. Scientific and Engineering Computation Series*. 2nd ed. Cambridge, MA: The MIT Press; 1997.
8. Walker D, Dongarra J. MPI: a standard message passing interface. *Supercomputer*. 1996;63:56-68.
9. Lirkov Ivan. Performance Analysis of a parallel denoising algorithm on Intel Xeon computer system. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K, eds. *13th International Conference on Parallel Processing and Applied Mathematics, PPAM 2019, Part II. 12044 of Lecture Notes in Computer Science*. Lecture notes in computer science 12044. Cham, Switzerland: Springer; 2020:93-100. https://doi.org/10.1007/978-3-030-43222-5_8.
10. Harizanov S. Reconstructing 2D radiographic data. Technical Report, University of Szeged; Szeged; 2017.
11. Paprzycki M, Cyphers C. Using Strassen's matrix multiplication in high performance solution of linear systems. *J Comput Math Appl*. 1996;31(4/5):55-61.
12. Ganzha M, Lirkov I, Paprzycki M. Performance analysis of hybrid parallel solver for 3D Stokes equation on Intel Xeon computer system. In: Todorov M, ed. *Applications of Mathematics in Technical and Natural Sciences, AMiTaNS 2019. 2164 of AIP Conference Proceedings*. Melville, NY: AIP; 2019:120003-1-120003-8. http://dx.doi.org/10.1063/1.5130863.
13. Cutress I. New #1 Supercomputer: Fugaku in Japan, with A64FX, take Arm to the Top with 415 PetaFLOPs. https://www.anandtech.com/show/15869/new-1-supercomputer-fujitsus-fugaku-and-a64fx-take-arm-to-the-top-with-415-petaflops; 2020.
14. Laccetti G, Montella R, Palmieri C, Pelliccia V. The high performance internet of things: using GVirtuS to share high-end GPUs with ARM based cluster computing nodes. Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J, *Parallel Processing and Applied Mathematics. PPAM 2013*. Lecture Notes in Computer Science8384. Berlin, Heidelberg: In: Springer; 2013:734-744. https://doi.org/10.1007/978-3-642-55224-3_69.
15. Dally WJ, Turakhia Y, Han S. Domain-specific hardware accelerators. *Commun ACM*. 2020;63(7):48-57.

How to cite this article: Lirkov I, Harizanov S, Paprzycki M, Ganzha M. Performance analysis of parallel high-resolution image restoration algorithms on Intel supercomputer. *Concurrency Computat Pract Exper*. 2020;e5996. https://doi.org/10.1002/cpe.5996