IEEE Access
The journal for rapid open access publishing

IEEE Xplore Digital Library
http://ieeexplore.ieee.org/Xplore

A Parallel Nonlocal Means Algorithm for Remote Sensing Image Denoising on an Intel Xeon Phi Platform

Fang Huang, Bo Lan, Jian Tao, Yinjie Chen, Xicheng Tan, Jie Feng and Yan Ma

***Abstract*—The nonlocal means (NLM) algorithm is one of the best image denoising algorithms because of its superior capability to retain the texture details of an image and is widely used in remote sensing (RS) image preprocessing. However, the time complexity of the algorithm is very high due to its nonlocality when searching for similar pixels. As a result, the NLM algorithm cannot satisfy the near real-time requirements of some specific applications. To resolve this issue, a parallel NLM algorithm based on Intel Xeon Phi hardware with Intel's Many Integrated Cores (MIC) architecture was designed and implemented in this study. The parallel algorithm achieved satisfactory speedup, but the speedup obtained showed a step-like distribution for different image sizes. This result was not expected based on the theoretical analysis, which predicted that the speedup should be independent of input dataset size. To address this problem, the parallel algorithm was further optimized by adding pretreatment approaches and cutting down the number of nested loops in the MIC. Finally, experiments using the standard and optimized versions were carried out using RS images of different sizes. Several conclusions could be drawn from the experimental results: (1) the standard parallel algorithm can obtain better speedup with only one MIC card; (2) the optimized parallel algorithm can completely eliminate the step distribution of the speedup and can also accelerate RS image processing significantly.**

***Index Terms*—Parallel Computing, MIC, Remote Sensing, Image Processing, OpenMP**

I. Introduction

Noise is inevitably produced during image generation and transmission and influences image processing quality, not only compromising the visual effect of the image, but also influencing our ability to recognize and understand information in the image and increasing image transmission costs [1]. As a result, image denoising has become a key technique and is coming to play a more and more important role in image processing [2]. Image denoising is an important step for remote sensing (RS) image preprocessing in related applications and has become a research branch in the field of RS image enhancement [3], [4], [5] [6], [7], [8].

Many studies have been carried out on image denoising, and many image denoising methods have been proposed in the last several years [9] [10]. However, most of those methods are not very satisfactory because much texture detail is often lost in the denoising process. The nonlocal means (NLM) algorithm, which was proposed in 2005, can prevent loss of detail to a certain degree [11]. The NLM algorithm can search for pixels that are similar to the target pixel in the entire image based on the self-similarity principle. As a result, it can make full use of the redundant information that exists everywhere in the image. During the search process, the weights to measure the similarity between pixels are calculated based on the differences between the gray-value vectors of similar pixel slices [12]. A pixel slice is a slice of the image of interest that is centered on the pixel that needs to be measured.

Among classical image denoising algorithms, the NLM algorithm attracted much attention when it was first proposed due to its capability to retain image details [13]. It is currently one of the most widely studied methods in the field of image denoising. Although the NLM algorithm has good denoising effect, it has high computational complexity, meaning that it is numerically expensive to execute [14].

To resolve this issue, many researchers have tried to improve the algorithm to enhance its performance. For instance, Vignesh *et al.* proposed a new fast NLM filtering algorithm in which the search process can be terminated early with a certain probability [15]. In this algorithm, a probabilistic statistical method is used to select similar pixels while filtering out dissimilar ones. Mahmoudi *et al.* proposed an improved algorithm using a filter to eliminate irrelevant neighborhoods during the weighted averaging process [16]. Because the filter is based on the average gray-value gradient, the improved algorithm can enhance performance effectively. Huang *et al.* used a graphic processing unit (GPU) to accelerate NLM algorithm performance and achieved speedups as high as 45X [17]. Hu *et al.* used the NLM algorithm for 3D ultrasonic image denoising and used GPU technology to accelerate real-time performance in this specific area [18]. Most recently, Zhu *et al.* designed and implemented OpenMP-based and OpenCL-based parallel NLM algorithms that can run on different platforms [19]. They focused mainly on a performance comparison of different versions on the CPU, GPU, and Intel

Fang Huang is with the School of Resources & Environment, University of Electronic Science and Technology of China, Chengdu 611731, P. R. China; and Institute of Remote Sensing Big Data, Big Data Research Center, University of Electronic Science and Technology of China, Chengdu 611731, P. R. China

Bo Lan is with the School of Resources & Environment, University of Electronic Science and Technology of China, Chengdu 611731 P.R. China

Jian Tao is with Texas A&M Engineering Experiment Station (TEES), Texas A&M University, TX 77843-3128, USA

Yinjie Chen is with the School of Construction & Management Engineering, Xihua University, Chengdu 610039, P.R. China

Xicheng Tan is with the International School of Software, Wuhan University, Wuhan 430079, P.R. China

Jie Feng is with the School of Resources & Environment, University of Electronic Science and Technology of China, Chengdu 611731, P.R. China

Yan Ma is with the Institute of Remote Sensing and Digital Earth, Chinese Academy of Scineces, P. R. China

Corresponding author: Yan Ma, email: mayan@radi.ac.cn

Digital Object Identifier: 10.1109/ACCESS.2017.2696362

Xeon Phi platforms.

In existing studies aiming to improve the performance of denoising applications, some researchers have focused on improving the algorithm itself, whereas others have taken advantage of emerging computing technologies. In general, the acceleration effect is usually better in the latter case, and the acceleration process is simpler and quicker than changing the algorithm. However, the GPU acceleration approach also has some flaws. The GPU programming mode is usually somewhat complex, and therefore developers must understand this new hardware architecture and programming environment [20]. In November 2012, Intel officially released the Xeon Phi coprocessor, which is based on the Intel Many Integrated Cores (MIC) architecture [21], [22]. The process of developing parallel algorithms is easier in this environment because MIC uses the X86 instruction set, with some new extension sets for parallelism purposes [23], [24]. Compared to the GPU platform, the speedup obtained on MIC may be less than the GPU can achieve. However, the programming style on MIC is more flexible, and it is a very suitable platform for developing parallel image processing algorithms such as the NLM algorithm.

This paper is organized as follows. Section 2 gives a brief introduction to various aspects of developing the parallel NLM algorithm, including the principle of the serial NLM algorithm and an introduction to Intel Xeon Phi. Section 3 concentrates mainly on the design and implementation of the parallel NLM algorithm. Because the speedups obtained did not agree with those expected from theoretical analysis, the section also describes the optimization measures that were taken to overcome these problems. Section 4 describes the experiments with the standard and optimized versions that were carried out with different sizes of input RS images. Finally, Section 5 discusses the results and draws conclusions.

II. Brief Introduction to the Background

A. Principle of the Serial NLM Algorithm

Buades *et al*. [11] presented the NLM algorithm in 2005. Compared to other denoising algorithms, the NLM algorithm shows superior performance in maintaining the image microstructure. The principle of serial NLM is illustrated in (Fig.1).

Assume that the target pixel is i, and that j_n are its neighborhoods. A pixel slice is a slice of the image of interest that is centered on the pixel that needs to be measured. For instance, N_i is the image slice with center pixel i, and $u(N_i)$ is used to represent the gray-value vector of the image slice of N_i. In this way, the similarity between i and j can be measured and calculated based on the differences between the gray-value vectors of similar pixel slices, *i.e.*, $u(N_i)$ and $u(N_j)$. In the process, the gray-value vector of the image slice provides a measurement of the similarity between pixels i and j. The more similar $u(N_i)$ and $u(N_j)$ are, are, the greater is the degree of similarity between i and j.

N_{j1}, N_{j2}, N_{j3} and N_i are the image slices (the white rectangles in Fig. 1) for pixels of j_1, j_2, j_3 and i respectively. Obviously, N_{j1} and N_{j2} have similar structural characteristics

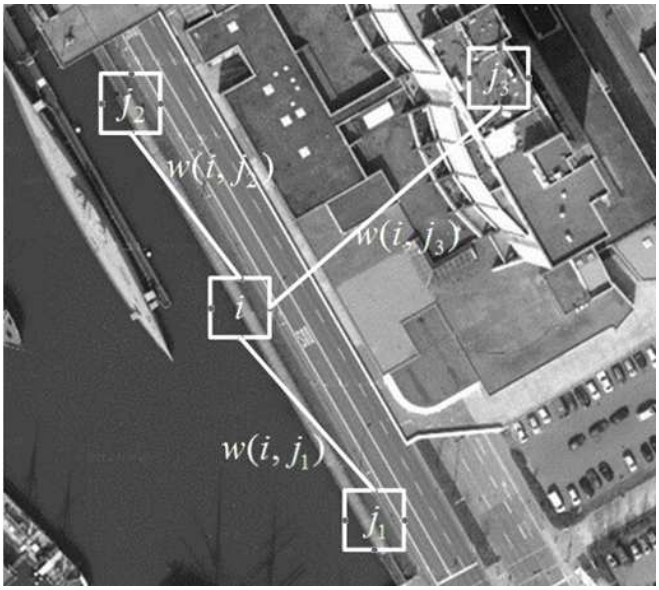


Fig. 1. Principle of the NLM algorithm.

to the image slice of i, here, N_i; whereas the image slice for j_3, N_{j3}, is different from N_i. Hence, the given weights for j_1 and j_2 will be much higher than for j_3. According to this principle, the NLM noise removal algorithm can be simply expressed as Equation (1).

$$NLu(p) = \frac{1}{C(p)} \int F(d^2(B(p,f), B(q,f)))u(q)dq \quad (1)$$

Where p is the target pixel, q is its neighbor, and their image slices and corresponding gray-value vectors are N_p, N_q , and $u(p)$, $u(q)$, respectively. F is a decreasing function, $C(p)$ is a normalization factor, and $d^2(B(p,f), B(q,f))$ represents the square Euclidean distance between N_p and N_q. The formula to calculate $d^2(B(p,f), B(q,f))$ can be expressed as:

$$d^2(B(p,f), B(q,f)) = \frac{1}{(2f+1)^2} \sum_{j \in B(0,f)} (u(p+j) - u(q+j))^2 \quad (2)$$

Note here that the image slice is represented as a circle and that the radius of the image slice can be changed. In Equation (2), $B(p,f)$ denotes the image slice for pixel p, and its corresponding radius is f.

B. Intel Xeon Phi Coprocessor

With increasing demands for computing power, computer hardware manufacturers began to integrate more and more cores into one chip [25]. In a many-core processor, the computation capability of each individual core is far less than that of one normal CPU, but because the processor has more cores, it is more suitable for concurrent computing.

The Intel Xeon Phi is designed for parallel computing and has the following features [26]: (1) it is a standard PCI-E interface board; (2) it supports the X86-64 instruction set and also has the extended 512-bit quantization instructions.

As a result, it is compatible with existing code running on a normal x86 CPU to a very great extent and has optimization methods and procedures similar to a traditional CPU; (3) it is equipped with an independent memory, power supply, and other features; (4) it has more than 50 cores. Every core offers four-way simultaneous multithreading (SMT) and 512-bit-wide SIMD vectors, which correspond to eight double-precision (DP) or sixteen single-precision (SP) floating-point numbers. Note that, compared with hyper threads, SMT has better performance because it has its own hardware thread unit. Hence, the Intel Xeon Phi coprocessor can provide powerful hardware support for high-concurrency scenarios; (5) it has a peak computing capability greater than 1 TFlops of DP; and (6) the card has a Linux-based chip operating system that supports Linux instructions and virtual Ethernet equipment. In this way, the MIC card can be regarded as one of the computing nodes in the network because it has its own IP address.

The MIC has extremely flexible programming modes. In programming design, the MIC card can be seen as a coprocessor, but also can be regarded as a separate computing node. In this way, its parallel application mode can be divided into three types [27]: (1) the CPU is the host, the MIC is the computing end; (2) the MIC card is the host, the CPU is the computing end; and (3) the CPU and the MIC are equivalent in the $offload$ mode. The most commonly used mode is the first one.

Furthermore, programming on the MIC platform offers several more optimization approaches. For example, performance can be further optimized by selecting the right parallel programming mode, or by selecting the compatible number of threads, or by selecting suitable running devices, $i.e.$, CPU or MIC, pointing to different pieces of code. The CPU and the MIC can use an asynchronous mechanism either between threads or within one thread. In this way, computing resources can be fully used. In particular, maintenance costs can be greatly reduced, $e.g.$, the code running in a normal CPU can be run on an MIC platform without much modification.

III. Parallel NLM Algorithm Design and Implementation

The sequential NLM algorithm has good denoising performance, but very high time complexity. Assuming that the input image size (height and width) is N x N, the size of the matching window is W x W, and the size of the search window is K x K, then the time complexity of the algorithm is $O(N^2K^2W^2)$. Such a high time complexity has made it difficult and challenging to apply the algorithm directly, especially in time-critical applications. According to the above analysis, parallelizing this algorithm on an MIC platform would be a wise choice. On the one hand, this algorithm is suitable to be parallelized; on the other hand, using an MIC will be easier and faster than using other coprocessors.

The process of designing and implementing the parallel algorithm involves the following steps: (1) find the hotspot of the serial algorithm using a performance analysis tool such as Intel Vtune Amplifier, and (2) plan suitable parallelization strategies that match the algorithm principle with the MIC hardware characteristics.

```
…
For y=0 To Height                          // Traversing each column
    …
    For x=0 To Width                       // Traversing each row
        // Calculate the searching window
        // Calculate the weight
        // Normalized the weights
        // Assignment
        ...
    End For
End For
// Subsequent processing
...
```

Fig. 2. Pseudocode of the NLM algorithm.

A. Sequential NLM Algorithm Hotspot Detection and Analysis

The NLM algorithm is a typical spatial-domain denoising algorithm. According to Equation (1), the NLM algorithm can be expressed using the pseudocode shown in Fig.2 [28].

During processing, every image pixel goes through the same sequence of operations: (1) determining the search box; (2) calculating the weight; (3) normalizing the weight; and (4) assigning the value. The most time-consuming part of the program is the 2D loop. The performance analysis tool, Intel VTune Amplifier, determined that this loop takes about 99% of the total time of the NLM algorithm. Because the operations on each pixel in the loop portion of the NLM algorithm are independent and no data dependency exists before or after the loop, the parallelization process is straightforward.

B. Parallel NLM Algorithm Design and Implementation

The number of cores in CPUs currently available on the market is 4 to 12, but the MIC coprocessor has approximately 60 cores, and each core provides four-way SMT. Hundreds of threads could be used for parallelization to help improve the efficiency of certain specific algorithms. However, parallel program development is closely related both to the parallelization strategy and the development tools used. Generally speaking, the first step should be to design and develop a parallel algorithm on the multicore platform and then to transplant the code to the MIC platform in a special "offload" mode. To make the results comparable, the elapsed time durations and speedups obtained with MIC in the rest of this paper were all measured under this programming mode.

How to allocate these many cores reasonably and use them efficiently is of great significance for the success of a parallel NLM algorithm. Based on the principle of the NLM algorithm and the characteristics of the MIC, the parallelization approach shown in (Fig.3) was proposed.

There are many cores in the MIC platform, and these cores can provide corresponding threads. For instance, the maximum number of threads offered by the MIC platform used in this research is 240. The computing operations for each pixel are divided into many threads and are then mapped to the corresponding cores. If the platform can provide sufficient

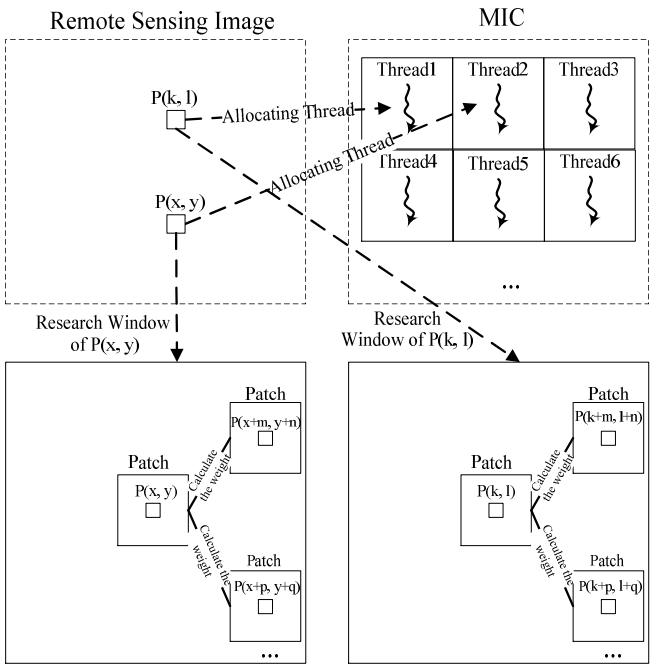


Fig. 3. Parallel NLM algorithm design.

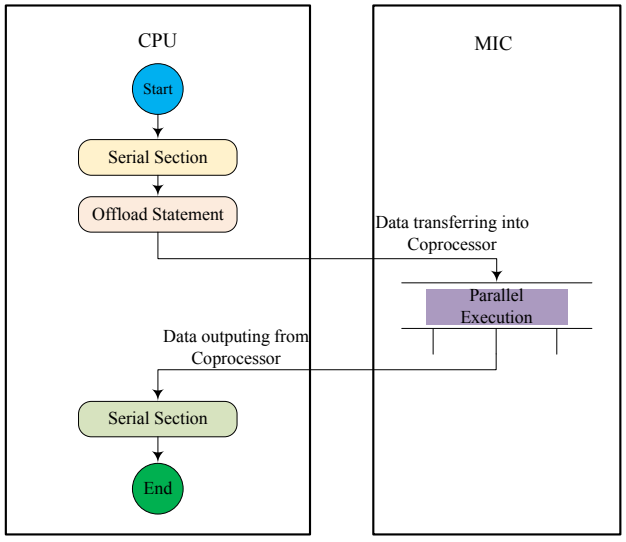


Fig. 4. Flowchart of parallel program execution.

computing power, optimal performance is obtained when the number of threads is equal to the number of pixels. Under this circumstance, each thread is responsible for one pixel. As shown in Fig.3, the loop processing time can be reduced to the time to process a single pixel. In fact, the platform used cannot meet this requirement. In other words, several pixels must be mapped into one thread. The parallel NLM algorithm therefore starts its execution on a CPU. The computation is then sent to the MIC to calculate the main loop. The threads are mapped to the MIC cores in batches. Each core starts one to four threads simultaneously. When the parallel part finishes, control is returned to the CPU, which outputs the results and

```
…
// Codes for other operations
…
// Start OpenMP command lines
Omp_set_num_threads (NUM_THREADS);
#pragma omp parallel for …
For y=0 To Height
…
        For x=0 To Width
                // Calculate the searching window
                // Calculate the weight
                // Normalized the weights
                // Assignment
                ...
        End For
End For

// Subsequent processing
```

(a) With OpenMP

```
// Codes for other operations
…
__attribute__((target(mic)))void pixdiff();
__attribute__((target(mic)))float exp();
…
// Start MIC command lines
#pragma offload target(mic:0)\
inout(ptSum(length(ap))alloc_if(1)free(1)\
...
inout(ptO(length(ap))alloc_if(1)free(1)
…
// Auto-vectorization definitions
int A[n] __attribute__((aligned(64)));
...
// Start OpenMP command lines
Omp_set_num_threads (NUM_THREADS);
#pragma omp parallel for …
For y=0 To Height
…
        For x=0 To Width
                // Calculate the searching window
                // Calculate the weight
                // Normalized the weights
                // Assignment
                ...
        End For
End For
// Subsequent processing
```

(b) With MIC

Fig. 5. Pseudocode of parallel NLM algorithm for multicore/MIC platform.

terminates the program (Fig.4).

To develop a parallel algorithm on an MIC platform, the best practice is first to implement the parallel program using OpenMP, which can run on a multicore platform. The OpenMP program implementation should consider the hotspots of the serial NLM algorithm. Subsequently, the input and output variables should be determined, and the corresponding $offload$ statement should be inserted accordingly.

By analyzing the hotspots of the algorithm, it became apparent that there is no data dependency in the two loop levels, $i.e.$, the x and y loops in Fig.2. This means that the algorithm can be parallelized as the top-level loop, the bottom-level loop, and both loops. According to established parallelization principles [29], it is strongly suggested to use the coarse-grained approach on most occasions, $i.e.$, it is better to perform parallelization starting from the upper layer. The coarse-grained programming approach reduces the time required for thread creation, destruction, and scheduling. In this study, the parallelized version was created using OpenMP as the development tool. The pseudocode is shown in Fig.5 in the red box.

This version of the parallel NLM algorithm can run on a multicore computing platform directly. In Fig.5 (a), the statement "omp_set_num_threads (NUM_THREADS)" indicates the number of threads that will be set. In addition, some other keywords like $offload$, in, $inout$, $alloc\_if$ and $free\_if$ are used to transplant the codes to the MIC computing platform (Fig.5 (b)).

In the blue box in Fig.5 (b), the keyword $offload$ divides the computing devices into the host end and the device end and it clearly defines the computing scopes for each end. Within the scope of $offload$, the code is executed on the host end; once outside the scope, the code is executed on the device end. According to the definition of the MIC parallelization mode, the host can be either the CPU or the MIC. In this research, the CPU was the host end by default, and the MIC

was the device end. In addition, certain other keywords are needed to control the corresponding memory spaces in both host and device ends for communication and data transfer. These keywords are in and $inout$. After memory allocation, the data on the host end are copied into the device end. At this point, the code is running in truly parallel computation mode. After the computation on the MIC is done, there are two possibilities for dealing with the computing results when they leave the device end. One is to copy the results and move them to the host end; the other is to do nothing. The $inout$ command is responsible for the first situation and the in command for the second. In this algorithm implementation, four memory spaces must be defined: $ptI, ptLu, ptO$ and $ptSum$. The first two use the in keyword, whereas the other two need $inout$ to control the memory copy operation. In addition, the memory space in the device end must be turned on and off. The keywords $alloc\_if$ and $free\_if$ are responsible for these actions. A test expression should be used as the input parameter for these operators. In this study, the expression was set to $true$, which means the program needs to allocate some memory when it enters the device end and free up the space used when leaving.

In summary, the main body of the parallel program first executes on the CPU, but transfers the computations to the MIC card when the program reaches the first "offload" statement. At the same time, certain variables, such as pointers needed by the MIC, are constructed with the help of keywords like in and out. The standard parallel NLM algorithm is illustrated by the pseudocode shown in Fig.5 (b).

Note that the functions and variables that execute on the MIC end must first be modified and labeled with $\_\_attribute\_\_(target(MIC))$ as indicated by the green box in Fig.5 (b)). Only in this way will the invocation and execution be correct. For example, the function $pixDist$ should be modified to $\_\_attribute\_\_((MIC(target)))pixDist(...)$ if invoked by the MIC end. In addition, to make full use of the MIC's computation potential, auto-vectorization is used in this standard parallel version (as shown by the purple box in Fig.5 (b)). With these definitions, the vectorization operation can be activated using Intel compiler options.

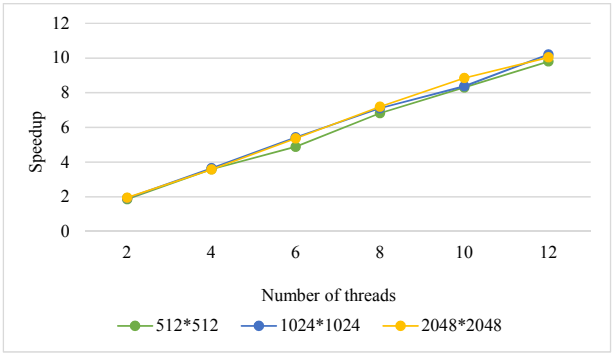
C. Performance Analysis with the Standard Parallel Version

Speedup is the most common used indicator to evaluate a parallel algorithm's performance. Speedup is the ratio of the elapsed time of a particular task running on a single-processor system to that on a parallel system, which can be calculated by Equation (3) [30]:

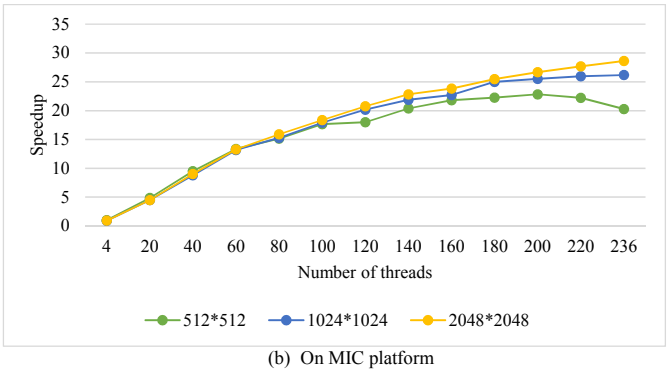
$$S_p = \frac{T_1}{T_p} \tag{3}$$

Where S_p is the speedup achieved, T_1 is the elapsed time for a single processor to process the task, and T_p is the running time of the parallel system, $e.g.$, on a multicore processor, to process the same computing task.

Using Equation (3), one can deduce the estimated speedup that the parallel NLM algorithm can achieve. As is well known, the time complexity of the serial NLM algorithm is $O(N^2K^2W^2)$. Because the values of K and W are fixed,



(a) On Multi-cores platform



(b) On MIC platform

Fig. 6. Speedup changes with number of threads.

the elapsed time increases with the number of pixels, $i.e.$, the product of the height and width (N) of the image. If there are sufficient cores (assuming n cores) to process the entire image at the same time, the expected speedup is equal to nc and can be expressed by Equation (4). However, this is an ideal condition. In real applications, the expected speedup is far smaller than this. However, it has been demonstrated that the speedup obtained is not influenced by image size, $i.e.$, the speedup achieved with different image sizes remains the same:

$$\lim_{nc \to N^2}(S_p) = \frac{O(N^2K^2W^2)}{O(\frac{N^2}{nc}K^2W^2)} = \frac{O(N^2K^2W^2)}{\frac{1}{nc}O(N^2K^2W^2)} = nc \tag{4}$$

In these experiments, two platforms were selected to determine the performance of the standard version. The first was a multicore platform and the second a MIC coprocessor platform. The details of these platforms are listed in Table I.

The experiments used three groups of datasets of different sizes, but extracted from the same original Landsat-7 satellite image (GeoTiff format) with sizes of 512*512, 1024*1024, and 2048*2048. The noise parameter σ of the datasets was set to 50. On the multicore platform, the maximum number of open threads was set to 12 so that one core was responsible for one thread. The maximum number of threads on the MIC can be set to as many as 236 (because there are 60 cores in total, and every core has 4 SMT, but one core is reserved to manage and control communications between the host and device ends). The calculated speedups are shown in Fig.6.

Fig.6 (a), shows that the actual speedup increases linearly

TABLE I
DEVELOPMENT PLATFORMS AND THEIR CONFIGURATIONS.

Platform	Configuration	Soft Wares Installed
Multi-cores Platform	Intel Xeon CPU E5-2697 v2, 12 cores, 2.70GHz 64G RAM	icpc compiler, Version 15.0.0.090, Red Hat, Enterprise Linux Server release 6.4
MIC Platform	Xeon Phi Co-processor, 60 cores 1.05 GHz, 64G RAM	icpc compiler, Version 15.0.0.090, Red Hat, Enterprise Linux Server release 6.4

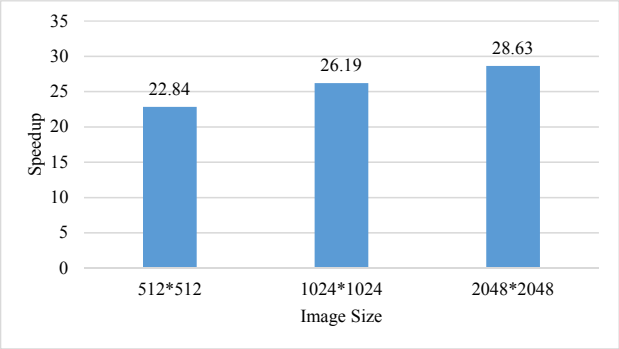


Fig. 7. Speedup changes with image size.

with the number of threads (cores) and agrees well with the expected speedup. However, Fig.6 (b) shows that the speedup increases linearly while the number of threads remains below 60. At this point, there is only one thread running in each core. After this point, the acceleration ratio curve begins to flatten out. This phenomenon has also appeared in other research studies [31]. Some researchers believe that this results from cache competition among concurrent threads [31].

In particular, the acceleration of the test cases with image sizes 1024*1024 and 2048*2048 reached a maximum when the number of threads was 236, whereas the case with image size 512*512 reached a maximum when 200 threads were involved. The maximum speedups obtained were different, meaning that speedup was dependent on input dataset size. Fig.7 shows the maximum speedups that could be achieved with these three images.

Figure 7 also shows that the parallel NLM algorithm can greatly reduce running time and can achieve good speedup. The smallest speedup, 22.84, was obtained with an image size of 512*512, and the greatest speedup, 28.63, was achieved with an image size of 2014*2014. This performance was satisfactory, but the acceleration effect was inconsistent for different image sizes. The speedup showed a strange stepped distribution with increasing image size. The speedup obtained with the 2048*2048 image size was greater than in the other cases. According to the theoretical analysis, this phenomenon should not occur, but it provided the motivation for further optimization of the parallel NLM algorithm.

IV. OPTIMIZATION OF THE PARALLEL NLM ALGORITHM

The speedups obtained considered only the main body of the NLM algorithm, i.e., the part with the two-layered loop. The reference time of the sequential algorithm, T_1 in Equation (1), was the running time for the algorithm that used only a single core on a common multicore CPU platform. Note that (1) the speedup increased almost linearly when the number of threads was less than 60, but after that, its growth slowed down; and (2) the speedup obtained changed with input image size and showed a step-like distribution. After scrutiny of the performance measurement data, it was concluded that the step-like distribution of the speedup was due to (1) the cost for MIC initialization and (2) the lack of a satisfactory coarse-grained strategy and adequate load balancing among threads. The following sections will address these issues.

A. Optimization Measure 1: MIC Pretreatment

In parallel mode, as described above, the parallel program is started by the CPU and first performs some serial operations; when it first encounters the "offload" statement (the blue box in Fig.5), it switches the next computing operation from the CPU to the MIC. At this point, the MIC card carries out certain initialization operations and then starts parallel processing of the "for" loop. After this, the main part of the NLM algorithm is finished. Finally, the program at the CPU end regains control and terminates the program. The present research has considered only the speedup obtained on the MIC, $i.e.$, the time taken for the computing task to run from start to finish on the MIC. Initializing the MIC requires some time (about 0.1 s or more), which reduces the speedup obtained to some extent. Because this initialization time is more or less constant, its influence on the speedup obtained is different for different input image sizes. For smaller images, the parallel part takes proportionally less time, and therefore the influence of initialization is proportionally greater. The opposite is true for larger files. As a result, the MIC initialization time contributed to the ladder-like speedup distribution to a certain extent. The algorithm was therefore optimized by moving MIC initialization from the parallel part to the CPU end. In other words, the statement in the blue box in Fig.5 was moved to the beginning of the program.

B. Optimization Measure 2: Improve the Coarse-Grained Loop Level

As described in Section 3, the coarse-grained method was removed from the outer loop layer to make the algorithm parallel using OpenMP. This method has the advantage of being easy to implement. However, it has some performance problems. For instance, taking a 512*512 image as an example,

TABLE II
ELAPSED TIME AND IMPROVEMENT RATIO COMPARISON FOR THE TWO OPTIMIZATION VERSIONS.

Image Size (M)	Time Duration of Standard Version (s)	Time Duration of Opt1 (s)	Opt1 Improvement (%)	Time Duration of Opt2 (s)	Opt2 Improvement (%)
512*512	9.63	8.45	12.25	6.12	27.55
1024*1024	33.34	32.25	3.25	27.6	14.42
2048*2048	125.33	124.20	0.90	113.37	8.72

```
...
whf=Width*Height
For cell=0 To whf
        ...
        x=cell%Width;
        y=cell/Height;
        ...
        // Calculate the searching window
        // Calculate the weight
        // Normalized the weights
        // Assignment
        ...
End For
// Subsequent processing
...
```
(a)

```
...
For cell=0 To whf                    // The outer loop
        ...
        For ai=0 To 4                // The inner loop
                x=(cell*4+ai)%Width;
                y=(cell*4+ai)/Height;
                ...
                // Calculate the searching window
                // Calculate the weight
                // Normalized the weights
                // Assignment
                ...
        End For
End For
// Subsequent processing
...
```
(b)

Fig. 8. Pseudocode for the optimized version.

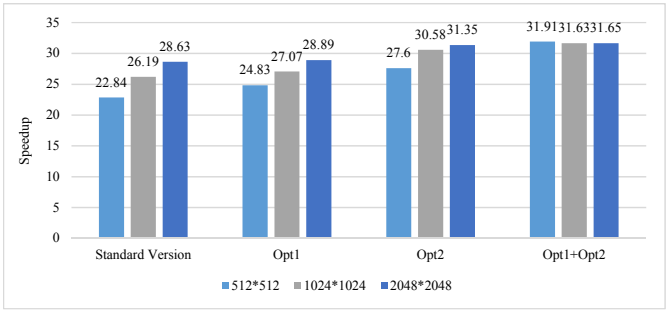


Fig. 9. Speedup changes with number of threads.

the outer loop (processing by height) is in parallel mode, but the inner loop is still in sequential mode (processing by width, 512 pixels in a line). Hence, 512 lines are distributed among 236 threads, meaning that every thread must process more than two (512/236=2.17) lines. Under this circumstance, there are 40 threads (512-236*2) that need to process three lines, at which point the other 196 threads are in a waiting state. This could also occur with images of different sizes. For instance, with 1024*1024 and 2048*2048 images, there are 156 and 116 threads respectively in this idle state. This explains the better performance in the 1024*1024 and 2048*2048 cases and contributes to the step-like speedup function obtained.

In an effort to optimize the method, the two-layered loop was redesigned into a one-level loop (illustrated in Fig.8 (a)). However, this method encountered stability issues, and sometimes the result was not correct. In the one-level loop method, every core has four SMT, meaning that the 236 threads are created by the 59 cores. Sometimes sequential pixels end up being processed on different cores. To avoid this, the restriction should be imposed that each group of four pixels is assigned to one core. The problem was solved by rearranging the loop from a one-level loop (Fig.8 (a)) to a new two-level loop (Fig.8 (b)). By using this method, the number of threads in a waiting state was reduced to a minimal level.

In the optimized parallel version, the outer-layer loop is expanded to its maximum extent. This helps reduce the ratio of idle threads to all threads and can make full use of computing resources. Note that this measure can in principle also be replaced by the "collapse" optimization technique in MIC programming. This is a very simple and effective way to solve multiple for loops. However, it encounters the same problem as the methods shown in Fig.8 (a) in practice.

C. Experiments on Optimizing Parallel Algorithms

Based on the standard parallel NLM algorithm and using the optimization methods presented above, two optimized parallel algorithms were implemented: Opt1-pretreatment and Opt2-coarse-grained. The average running times of these two optimized versions were obtained, the corresponding improvements (Tab. II) were calculated, and their speedups were compared (Fig. 9). In the experimental results presented below, the input data were identical to those used in testing the standard version.

Table II shows that: (1) both kinds of optimization method are effective, and the running time can be greatly shortened, especially for smaller input images; and (2) the second optimization method has a stronger performance enhancement effect than the first.

Fig 9 compares the speedups achieved with the standard method, Opt1, Opt2, and a combined optimization. The figure shows that for different input data sizes, (1) the speedup of the standard version presents a sharp ladder-like distribution with obvious differences for different input sizes; (2) both Opt1 and Opt2 showed increased speedup, but the trend slowed down with larger images; (3) with the combined optimization, the speedups obtained for different image sizes remained constant at the maximum value of 31X. Furthermore, the step distribution disappeared, in accordance with the theoretical analysis.

V. CONCLUSIONS AND FUTURE WORK

This study has introduced the principle of the NLM algorithm and has presented a design and several implementations of the corresponding parallel algorithm on an Intel MIC architecture. The standard parallel version using "offload" mode can achieve a speedup of 20-28X. Experiments found

that the speedups achieved showed a step-like distribution for different RS image sizes. Hence, the algorithm was further optimized using two approaches. One involved considering MIC initialization time, and the other involved improving the coarse-grained loop level. Comparison of the basic parallel and optimized versions revealed that the optimized parallel NLM algorithm can satisfactorily eliminate the ladder-distribution phenomenon.

In summary, the parallel algorithm can be used for real-time image denoising and large-scale image processing applications on an Intel MIC platform. The parallel algorithm presented here still has room for further optimization in the following directions: (1) many optimization measures have not been considered in this research, for instance, optimization methods involving memory management, cache use, and data transfer [32]; (2) when the MIC starts computing, the CPU becomes idle, meaning that the available computing resources are not all efficiently used. The authors are working on a dynamic load-balancing parallel NLM algorithm that can make full use of the computing capacity of both the MIC and the CPU; (3) unfortunately, there was only one MIC card in the experimental setup used here. Actually, modern computing platforms often have several MIC devices. A suitable scheduling algorithm should therefore be designed to enable the code developed in this research to make full use of computing resources under this circumstance.

VI. Acknowledgements

This study was supported by Key Laboratory of Spatial Data Mining & Information Sharing of Ministry of Education, Fuzhou University (Grant No. 2017LSDMIS03); Fundamental Research Funds for the Central Universities (Grant No. ZYGX2015J111); the National Natural Science Foundation of China (Grant Nos. 41001221, 51277167, 41471368, and 41571413); and also the National Science Foundation of the United States (Grant Nos. IIS-1723292, and 1251095). This study used computing resources provided by Tsinghua University and we wish to thank the support staff for their hard work and support. The serial NLM algorithm implementation is based on the source code of http://www.ipol.im/pub/art/2011/bcm_nlm/srcdoc/libdenoising_8cpp_source.html. Here we would like to thank the provider.

References

[1] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity." *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[2] M. Elad and M. Aharon, "Image denoising via sparse and redundant representations over learned dictionaries," *IEEE Transactions on Image Processing*, vol. 15, no. 12, pp. 3736–45, 2007.

[3] J. Wei, Y. Huang, K. Lu, and L. Wang, "Nonlocal low-rank-based compressed sensing for remote sensing image reconstruction," *IEEE Geosci. Remote Sensing Lett.*, vol. 13, no. 10, pp. 1557–1561, 2016.

[4] L. Wang, H. Geng, P. Liu, K. Lu, J. Kolodziej, R. Ranjan, and A. Y. Zomaya, "Particle swarm optimization based dictionary learning for remote sensing big data," *Knowl.-Based Syst.*, vol. 79, pp. 43–50, 2015.

[5] F. Aires, W. B. Rossow, N. A. Scott, and A. Chedin, "Remote sensing from the infrared atmospheric sounding interferometer instrument 1. compression, denoising, and first-guess retrieval algorithms," *Journal of Geophysical Research Atmospheres*, vol. 107, no. D22, pp. ACH 6–1–ACH 6–15, 2002.

[6] P. Liu, F. Huang, G. Li, and Z. Liu, "Remote-sensing image denoising using partial differential equations and auxiliary images as priors," *IEEE Geoscience & Remote Sensing Letters*, vol. 9, no. 3, pp. 358–362, 2012.

[7] L. Wang, K. Lu, P. Liu, R. Ranjan, and L. Chen, "IK-SVD: dictionary learning for spatial big data via incremental atom update," *Computing in Science and Engineering*, vol. 16, no. 4, pp. 41–52, 2014.

[8] L. Wang, K. Lu, and P. Liu, "Compressed sensing of a remote sensing image based on the priors of the reference image," *IEEE Geosci. Remote Sensing Lett.*, vol. 12, no. 4, pp. 736–740, 2015.

[9] J.-L. Starck, E. J. Cands, and D. L. Donoho, "The curvelet transform for image denoising," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 11, no. 6, p. 670684, 2002. [Online]. Available: http://dx.doi.org/10.1109/TIP.2002.1014998

[10] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image denoising by sparse 3-d transform-domain collaborative filtering," *Image Processing, IEEE Transactions on*, vol. 16, no. 8, pp. 2080–2095, 2007.

[11] A. Buades, B. Coll, and J. M. Morel, "A non-local algorithm for image denoising," in *IEEE Computer Society Conference on Computer Vision & Pattern Recognition*, 2005, pp. 60–65 vol. 2.

[12] Y. C. Li, C. M. Yeh, and C. C. Chang, "Data hiding based on the similarity between neighboring pixels with reversibility," *Digital Signal Processing*, vol. 20, no. 4, pp. 1116–1128, 2010.

[13] J. Wu and C. Tang, "Random-valued impulse noise removal using fuzzy weighted non-local means," *Signal Image & Video Processing*, vol. 8, no. 2, pp. 349–355, 2014.

[14] C. Liu and W. T. Freeman, *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part III*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. A High-Quality Video Denoising Algorithm Based on Reliable Motion Estimation, pp. 706–719. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15558-1_51

[15] R. Vignesh, B. T. Oh, and C. C. J. Kuo, "Fast non-local means (nlm) computation with probabilistic early termination," *IEEE Signal Processing Letters*, vol. 17, no. 3, pp. 277–280, 2010.

[16] M. Mahmoudi and G. Sapiro, "Fast image and video denoising via nonlocal means of similar neighborhoods," *IEEE Signal Processing Letters*, vol. 12, no. 12, pp. 839–842, 2005.

[17] K. Huang, D. Zhang, and K. Wang, "Non-local means denoising algorithm accelerated by gpu," *Proc Spie*, vol. 7497, pp. 749 711–749 711–8, 2009.

[18] S. Hu and W. G. Hou, "Denosing 3d ultrasound images by non-local means accelerated by gpu," in *International Conference on Intelligent Computation and Bio-Medical Instrumentation*, 2011, pp. 43–45.

[19] H. Zhu, Y. Wu, P. Li, D. Wang, W. Shi, P. Zhang, and L. Jiao, "A parallel non-local means denoising algorithm implementation with openmp and opencl on intel xeon phi coprocessor," *Journal of Computational Science*, vol. 17, Part 3, pp. 591–598, 2016, recent Advances in Parallel Techniques for Scientific Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750316301090

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[21] G. Chrysos, "Intel® xeon phi coprocessor-the architecture," *Intel Whitepaper*, 2014.

[22] W. Xue, C. Yang, H. Fu, X. Wang, Y. Xu, J. Liao, L. Gan, Y. Lu, R. Ranjan, and L. Wang, "Ultra-scalable CPU-MIC acceleration of mesoscale atmospheric modeling on tianhe-2," *IEEE Trans. Computers*, vol. 64, no. 8, pp. 2382–2393, 2015.

[23] K. W. Schulz, R. Ulerich, N. Malaya, P. T. Bauman, R. Stogner, C. Simmons, and R. Ulerich, "Early experiences porting scientific applications to the many integrated core (mic) platform," *Tacc*, 2012.

[24] J. Wei, D. Liu, and L. Wang, "A general metric and parallel framework for adaptive image fusion in clusters," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1375–1387, 2014.

[25] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 2011.

[26] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[27] Y. Hu, Q. Li, Z. Cao, and J. Wang, "Parallel simulation of high-dimensional american option pricing based on cpu versus mic," *Concurrency & Computation Practice & Experience*, vol. 27, no. 5, pp. 1110–1121, 2014.

[28] G. Palma, M. Comerci, B. Alfano, S. Cuomo, P. D. Michele, F. Piccialli, and P. Borrelli, "3d non-local means denoising via multi-gpu," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, 2013, pp. 495–498.

[29] Q. He, D. Chen, and D. Jiao, "From layout directly to simulation: A first-principle-guided circuit simulator of linear complexity and its efficient parallelization," *IEEE Transactions on Components Packaging & Manufacturing Technology*, vol. 2, no. 4, pp. 687–699, 2012.
[30] S. Brawer, *Introduction to parallel programming*. Academic Press, 2014.
[31] D. Kou and D. Kong, "Parallel implementation of finite-element mesh integration algorithm on many integrated core," *Computer Science*, vol. 42, no. 11, pp. 56–60, 2005.
[32] E. Wang, B. Zhang, Qing andd Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *High-Performance Computing on the Intel Xeon Phi(TM): How to Fully Exploit MIC Architectures*. Springer Press, 2014.