Received June 10, 2021, accepted June 23, 2021. Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2021.3094840

Highly Efficient and Scalable Framework for High-Speed Super-Resolution Microscopy

QUAN DO[1], SEBASTIAN ACUÑA[2], JON IVAR KRISTIANSEN[1], KRISHNA AGARWAL[2], AND PHUONG HOAI HA[1]
[1]Department of Computer Science, UiT The Arctic University of Norway, 9037 Tromsø, Norway
[2]Department of Physics and Technology, UiT The Arctic University of Norway, 9037 Tromsø, Norway

Corresponding author: Quan Do (quan.do@uit.no)

This work was supported in part by the UiT The Arctic University of Norway through the NanoFrame Project under Grant 302535, in part by the European Commission (EU) Horizon 2020's MSCA-IF under Project 749666, in part by the Research Council of Norway's FRIPRO Young Project under Grant 288082, in part by the Experimental Infrastructure for Exploration of Exascale Computing (eX3) Project under Grant 270053, and in part by the UNINETT Sigma2-the National Infrastructure for High-Performance Computing and Data Storage in Norway under Grant NN9342K.

ABSTRACT The multiple signal classification algorithm (MUSICAL) is a statistical super-resolution technique for wide-field fluorescence microscopy. Although MUSICAL has several advantages, such as its high resolution, its low computational performance has limited its exploitation. This paper aims to analyze the performance and scalability of MUSICAL for improving its low computational performance. We first optimize MUSICAL for performance analysis by using the latest high-performance computing libraries and parallel programming techniques. Thereafter, we provide insights into MUSICAL's performance bottlenecks. Based on the insights, we develop a new parallel MUSICAL in C++ using Intel Threading Building Blocks and the Intel Math Kernel Library. Our experimental results show that our new parallel MUSICAL achieves a speed-up of up to 30.36x on a commodity machine with 32 cores with an efficiency of 94.88%. The experimental results also show that our new parallel MUSICAL outperforms the previous versions of MUSICAL in Matlab, Java, and Python by 30.43x, 2.63x, and 1.69x, respectively, on commodity machines.

INDEX TERMS Computational nanoscopy, parallel programming, optimization, super-resolution imaging, image enhancement.

I. INTRODUCTION

In biomedical imaging, there is a limitation in resolving details smaller than the Abbe diffraction limit, which itself is the ratio of the fluorescence emission wavelength to twice the numerical aperture of the microscopy. There are several techniques to overcome the resolution limit, such as single-molecule localization microscopy (SMLM) [1]–[6], including stochastic optical reconstruction microscopy (STORM) [7], photo-activated localization microscopy (PALM) [8]–[9], multiple signal classification (MUSIC) [10]–[12], structured illumination microscopy (SIM) [13], and fluorescence fluctuations based super-resolution microscopy (FF-SRM) [14]. The multiple signal classification algorithm (MUSICAL) [15] belongs to the family of fluctuations-based super-resolution microscopy and is the primary focus of this work.

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

In the original research article of MUSICAL [15], the authors showed several advantages of MUSICAL, such as providing resolution down to 50 nm, low requirements on the number of frames and excitation power, working at high fluorophore densities, and working with any fluorophore that exhibits blinking on the time scale of the recording. Agarwal and Macháň [15] focused on the conceptual physics and statistics-rooted development of MUSICAL, rather than optimizing the computational performance. Their article nonetheless included a comment on the potential of parallelizability, which was not been investigated further.

Input to the algorithm is a video of a fluorescently labeled sample acquired using a high-speed camera. The algorithm considers a sub-video per pixel by using a small neighborhood kernel around it and generates a local super-resolved sub-image. The sub-images are then superimposed at the original location of the pixels to obtain the full field of view of the super-resolved output image. Since the sub-video for each pixel is independently processed, MUSICAL has the potential for parallelization. The first implementation of MUSICAL was conducted in Matlab 2012b. The program used a graphical user interface (GUI), which is convenient for users. However, parallelization was not implemented.

To improve the running time and usability of MUSICAL compared with the first implementation in Matlab, open-source MusiJ version 0.94 [16] was implemented using Java with ImageJ, open library ND4J version 1.0.0-beta6. In open library ND4J, researchers adopted a multi-threading technique and OpenBLAS for solving large matrix multiplication. Currently, another protected improvement of MUSICAL using Python has outperformed the MusiJ-based version [17]. The Python-based version has used NumPy in Python for numerical and linear algebra functions.

In [15]–[17], no auto-thresholding schemes were employed, and the user had to choose a threshold manually by visual inspection of a singular value graph. This is inconvenient, heuristic, and time-consuming. To make MUSICAL more user-friendly, soft-thresholding schemes derived from a new generalized framework for indicator function design were proposed [18] which included auto-thresholding based on the range of the second singular value. That research also showed that those schemes significantly alleviated the subjectivity and sensitivity of hard thresholding while retaining the super-resolution ability. Therefore, computationally efficient auto-thresholding techniques may be quite helpful.

However, a MUSICAL implementation which can perform high-throughput super-resolution microscopy through better efficiency in terms of computational performance is desirable. The previous versions of MUSICAL in Matlab, Java, and Python have several limitations in libraries and techniques for performing high-throughput microscopy; for example, no parallelization technique was used in MUSICAL in Matlab, there were no pointers to optimize memory and caches in all the programs, and OpenBLAS, which does not have the best performance for large matrix multiplication, was adopted in ND4J for MusiJ in Java. Through our new MUSICAL implementation in C++, we resolved all the listed limitations of the previous versions of MUSICAL by using Intel Threading Building Blocks (TBB) and the Intel Math Kernel Library (MKL), and by running our program on Intel Xeon central processing units (CPUs). To achieve this goal, we used several approaches for a new parallel MUSICAL in C++, such as using C++ pointers to outperform the other programming languages mentioned, choosing the best multi-threading technique, optimizing the algorithm and functions through performance analysis, choosing the best function to perform high-throughput super-resolution microscopy, and proposing a fast auto-threshold algorithm.

The contributions of this paper are summarized as follows.

- We optimize MUSICAL for a performance analysis. To analyze the correct performance of individual components in MUSICAL, we optimize and implement MUSICAL in C++ to remove the overhead of high-level programming frameworks such as Java, Python, and Matlab.
- We provide insights into MUSICAL's performance. We have discovered that MUSICAL's performance mainly depends on large matrix multiplication algorithms, disproving the hypothesis that MUSICAL's performance is dominated by singular value decomposition (SVD) [17].
- We develop a new parallel MUSICAL in C++, evaluate the performance of MUSICAL methods in Matlab, Java, Python, and C++, and analyze the scalability of our new parallel MUSICAL on commodity machines. Our experimental results show that our new parallel MUSICAL achieves a speed-up of up to 30.36x on a commodity machine with 32 cores.

II. OVERVIEW OF MUSICAL ALGORITHM

Various MUSICAL flowcharts have been presented in Matlab [15], MusiJ in Java [16], and Python [17]. To simplify matters for other researchers who want to re-implement and understand how we improve the algorithm in a C++ implementation, we present another flowchart of MUSICAL in Fig. 1. In this flowchart, one block is like a function. In this manner, we were able to perform a more systematic performance analysis and simplify our program optimization.

As illustrated by Fig. 1, we calculate or load a point spread function (PSF) matrix in step 2. To optimize our program, we put all mask computations, such as a diagonal Gaussian mask in step 3, mapping a charge coupled device (CCD) mask in step 4, and making a three-dimensional (3D) stack image in step 5, outside loops for multi-threading and scanning windows. To make MUSICAL more user-friendly and easier to use, we propose an auto-threshold algorithm in step 6. Steps 7 and 8 are multi-threading and scanning all pixels, respectively, to obtain windows for super-resolution processing.

An input stack image in 3D (x-y-t) is divided into several partitions. Each partition is scanned for every pixel, where one pixel corresponds to a 3D-cropped window in step 11.

Singular values and singular vectors are calculated for that window in step 12. Then, singular values are compared with the auto-threshold value to determine the row positions in step 13. Those positions divide singular vectors into two parts, i.e., the signal part and the noise part. Afterward, we divide the projection of the PSF onto the signal part by the projections onto the noise part of the input window and use the power factor α to obtain a reconstructed window in step 14.

We next rotate the reconstructed window so that it has the same direction as the input window in step 15 and update that result into the whole reconstructed image in step 16. When we scan all pixels of the input image, we stitch all reconstructed windows together to get the final super- resolution image in step 17.

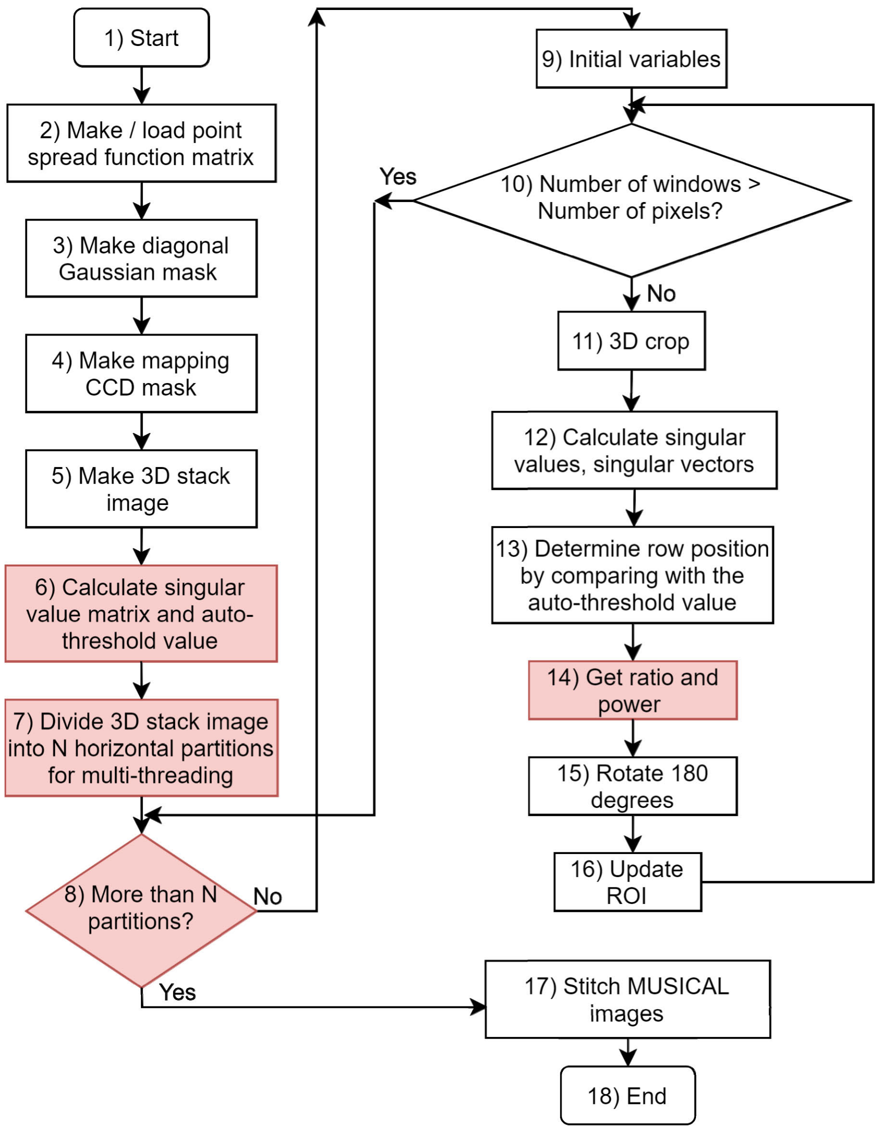

1) Start
2) Make / load point spread function matrix
3) Make diagonal Gaussian mask
4) Make mapping CCD mask
5) Make 3D stack image
6) Calculate singular value matrix and auto-threshold value
7) Divide 3D stack image into N horizontal partitions for multi-threading
8) More than N partitions?
No
Yes
9) Initial variables
10) Number of windows > Number of pixels?
Yes
No
11) 3D crop
12) Calculate singular values, singular vectors
13) Determine row position by comparing with the auto-threshold value
14) Get ratio and power
15) Rotate 180 degrees
16) Update ROI
17) Stitch MUSICAL images
18) End

FIGURE 1. **The flowchart of the multiple signal classification algorithm (MUSICAL), adapted from [15]. Pink shading indicates contributions of this paper. 3D = three-dimensional; CCD = charge-coupled device; ROI = region of interest.**

As highlighted by pink shading in Fig. 1, our contributions are a proposed auto-threshold algorithm (step 6), choosing the best multi-threading technique (steps 7–8), and choosing the best high-performance computing through large matrix multiplication (step 14).

III. NEW PARALLEL MUSICAL IN C++

We collect all the salient properties of the body of work on MUSICAL [15]–[17], such as using floating-point arithmetic (double-precision floating-point arithmetic was used in Matlab in [15]), using a squared matrix to get eigenvalues and eigenvectors, sharing buffers for matrices, and multi-threading. In this paper, through the performance analysis, we improve MUSICAL's running time. We first propose an auto-threshold algorithm to avoid changing a manual threshold frequently. That algorithm is useful for our experiments. We contribute to the improvement of MUSICAL in multi-threading optimization by selecting the best multi-threading technique and optimizing large matrix multiplication by selecting the latest and best large matrix multiplication technique.

A. PROPOSED AUTO-THRESHOLD ALGORITHM

For more user-friendly and easier usage, we propose an auto-threshold algorithm as shown in Fig. 2. The idea of the proposed algorithm comes from how users select a manual threshold in [15]. In that research, the authors sketched singular value matrix curves where the x coordinate indicates the square of the window size, the y coordinate presents the

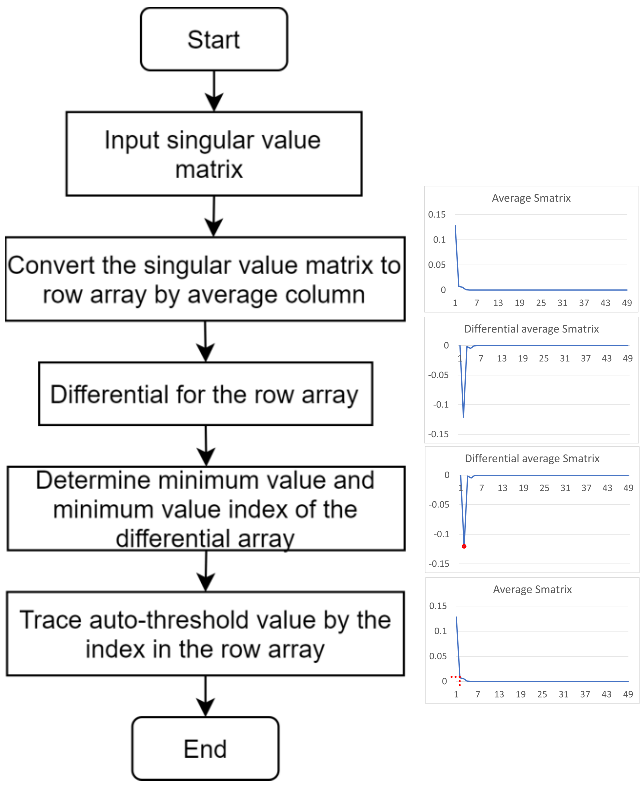

Start
Input singular value matrix
Convert the singular value matrix to row array by average column
Differential for the row array
Determine minimum value and minimum value index of the differential array
Trace auto-threshold value by the index in the row array
End
Average Smatrix
Differential average Smatrix
Differential average Smatrix
Average Smatrix

FIGURE 2. **Proposed auto-threshold algorithm.**

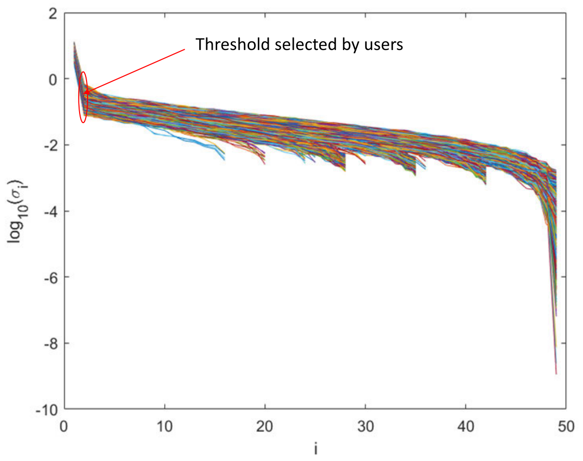


FIGURE 3. **Selected manual threshold on singular value curves in [15]. The *x* coordinate indicates the square of the window size; the *y* coordinate presents the eigenvalues in logarithmic scale.**

eigenvalues in logarithmic scale, and the number of curves equals the number of pixels in a single input frame.

Since singular values are sorted from the biggest value to the smallest value, the singular value curves are monotonically decreasing. Users normally select a threshold in the middle vertical direction of the first interrupted point of singular value curves as presented in Fig. 3. Based on that action, we propose an auto-threshold algorithm as in Fig. 2. The details of the algorithm are provided as follows.

Singular value matrix S has a height corresponding to the square of the window size (N_w) and a width corresponding to the multiplication of width (W) and height (H) of the input stack image, $N_w^2 \times W * H$, which was explained and calculated in [15]. It is the input of the proposed auto-threshold algorithm. We convert the singular value matrix to a row array by taking the mean of all the columns. We take the mean of column c as follows:

$$\overline{S}_c = \frac{\sum_{i=1}^{W\times H} S_{ic}^T}{W \times H}, \tag{1}$$

where $\overline{S}_c$ is the mean of a set of column c, and $\sum_{i=1}^{W\times H} S_{ic}^T$ is the sum of elements from column c of the transposed singular matrix S. By computation in Eq. (1), we get $\overline{S}$:

$$\overline{S} = \left[\overline{S}_1 \quad \overline{S}_2 \quad \dots \quad \overline{S}_M \right], \tag{2}$$

where M equals N_w^2.

Next, we take differences and approximate derivatives for the row array $\overline{S}$ to get the row array D as follows:

$$D = \left[\overline{S}_2 - \overline{S}_1 \quad \overline{S}_3 - \overline{S}_2 \quad \dots \quad \overline{S}_M - \overline{S}_{M-1} \right]. \tag{3}$$

With the differential array D with M-1 length, we find the minimum value and its index.

$$\underset{j\in[1,M-1]}{\arg\min}\ D_j. \tag{4}$$

We trace the minimum index j of the row array D and we finally find the auto-threshold value $\overline{S}_j$. The result of the auto-threshold algorithm is used for determining the row position of a singular value for each window in the next step. The proposed auto-threshold algorithm adopts the same action of users for selecting a threshold. Furthermore, it computes automatically without user interaction in our MUSICAL program. By doing this, the MUSICAL program is more convenient and user-friendly.

B. MULTI-THREADING TECHNIQUES

Multi-threading is the ability of a CPU to execute multiple threads concurrently. In our implementation in Windows, we divide the input stack image into partitions. We choose the best setting for partitions among several settings. The optimal number of partitions is discussed in Section IV. Then, each partition is sent to the core to process simultaneously.

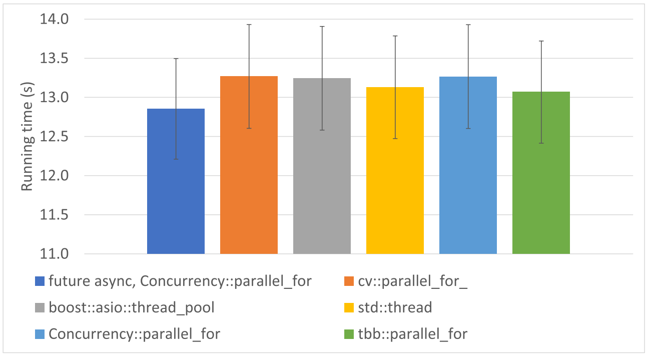
No multi-threading technique was used in the first implementation of MUSICAL in Matlab [15]. For the next implementation of MusiJ in Java, *ThreadPool*, *Executors*, and an implementation of a lambda function were used for multi-threading [16]. Another improvement with *pool* and *async* in Python was implemented [17]. However, by implementing and testing all multi-threading techniques, we choose the best technique and apply it to our MUSICAL in C++.

In our implementation, we experiment with various multi-threading techniques, such as the Parallel Patterns Library's (PPL) *future async* combined with *parallel_for*, Intel TBB

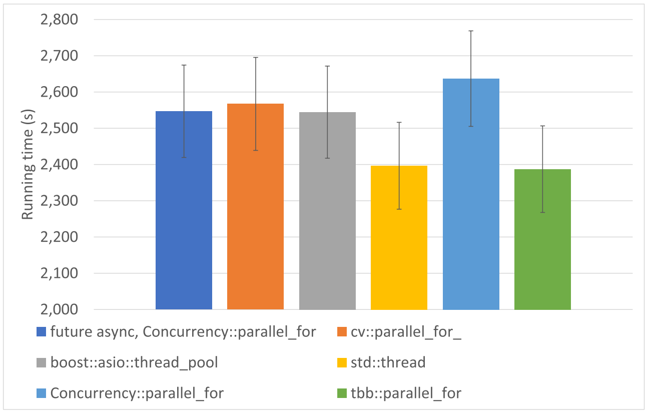
TABLE 1. Running time of our multiple signal classification algorithm in C++ for multi-threading techniques.

Technique	Running time with image size 160×160×49 (s)	Running time with image size 2048×2048×500 (s)
future async, Concurrency::parallel_for	**<u>12.85</u>**	2546.95
tbb::parallel_for	**13.07**	**<u>2387.18</u>**
std::thread	13.13	**2396.57**
boost::asio::thread_pool	13.25	2544.54
*cv::parallel_for_*	13.27	2567.32
Concurrency::parallel_for	13.27	2637.02
ΔT	≤ 3.27%	≤ 10.47%

* The bold underlining represents the best performance.



a) 160×160×49



b) 2048×2048×500

FIGURE 4. Running time graphs for our multiple signal classification algorithm in C++ for multi-threading techniques.

version 2020.2.216's *parallel_for*, C++'s *thread*, Boost version 1.73.0's *async* input/output *thread_pool*, OpenCV version 3.4.10's *parallel_for*, and PPL's *parallel_for*.

We test those multi-threading techniques five times under the same conditions, such as partitions per core = 1, the same optical and MUSICAL parameters, and we present the average running times in Table 1 and the error bars in Fig. 4. Through experiments with two stack images— InVitroSample1.tif of size 160 × 160 × 49 and 30.08.19-14.26.00.tif of size 2048 × 2048 × 500—we select the most suitable multi-threading technique for our program.

In Table 1 and Fig. 4, we see that with stack image 160 × 160 × 49, *future async* combined with PPL's *parallel_for* gives us the best performance (12.85 s), and the second best performance is obtained by using TBB's *parallel_for* (13.07 s). The time difference between the two methods is 1.71%. The time difference between the two techniques is defined as

$$\Delta T = \frac{T_1 - T_2}{T_2} \times 100\%, \tag{5}$$

where T_1 and T_2 are the running time of multi-threading techniques 1 and 2, and ΔT is the time difference between the two techniques.

However, using the previously mentioned partitions per core for the larger stack image of size 2048 × 2048 × 500, the best performance is obtained by TBB's *parallel_for* (2387.18 s), and the lower performance by C++'s *std::thread* (2396.57 s), while *future async* combined with PLL's *parallel_for* uses 2546.95 s. The time difference between the two best methods is 0.39% and that of TBB's *parallel_for* and *future async* combined with PPL's *parallel_for* is 6.69%.

By this performance analysis of various multi-threading techniques and the experimental results in Table 1 and Fig. 4, we conclude that TBB's *parallel_for* gives us the most suitable multi-threading technique for our program. TBB also gave the best multicore programming solution in other research [19]. That is further evidence of our optimal implementation for the multi-threading technique.

C. HIGH-PERFORMANCE COMPUTING TECHNIQUES FOR LARGE MATRIX MULTIPLICATION

In Fig. 1, obtaining the ratio and power block in step 14 includes large matrix multiplication, which is multiplication between the eigenvector matrix and the PSF mapping matrix. The size of the eigenvector matrix is $N_w^2 \times N_w^2$, while the size of the PSF mapping matrix is $N_w^2 \times N_w^2 \times Sub^2$, where Sub is the subpixel per pixel factor to present how much super resolution we want to have. N_w^2 and $N_w^2 \times Sub^2$ are the height and width of the CCD matrix, respectively.

In applications, the window size is less than or equal to 11, $N_w \leq 11$, while $Sub \leq 20$. In our tests, we use $N_w = 7$ and $Sub = 20$ based on the optical characteristics of the stack image, a discussion of which is out of the scope of this paper. The choice implies that we have a matrix multiplication of 49×49 and 49×19600 to get a resulting matrix of 49×19600. The running time of that multiplication is long since it lies inside a loop of all pixels of the input stack image. This means that the number of large matrix multiplications is the same as the number of pixels of the input stack image. Hence, finding the optimal large matrix multiplication is of paramount importance for our MUSICAL in C++. To find the best large matrix multiplication technique, we use stack image InVitroSample1.tif of size 160 × 160 × 49 since it is large enough in terms of the number of pixels and is suitable for the waiting test time. Other input stack images have similar results.

TABLE 2. Running time for large matrix multiplication per window for various techniques with stack image size 160 × 160 × 49.

Technique	Minimum time (ms)	Maximum time (ms)	Average time (ms)
mmul (MusiJ in Java)	1.48	178.98	5.58
cv::gemm (OpenCV)	24.32	311.8	55.71
Eigen	0.15	193.96	11.72
cblas_sgemm (CBLAS / MKL)	0.23	29.26	**2.41**

* The bold underlining represents the best performance.

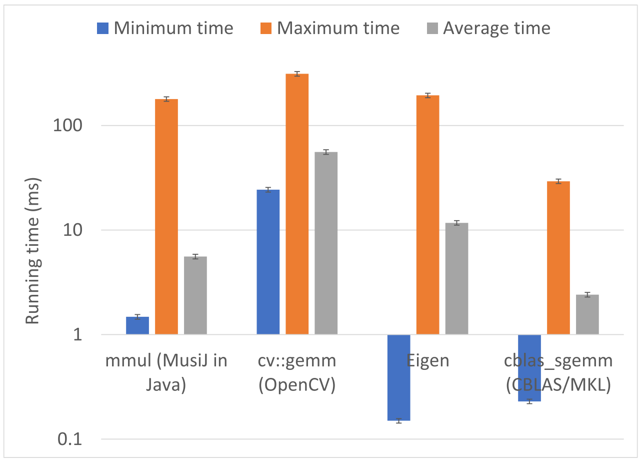


FIGURE 5. Large matrix multiplication per window graph of various techniques for stack image size 160 × 160 × 49.

We know that MusiJ uses open library ND4J where OpenBLAS is adopted. Table 2 and Fig. 5 show that MusiJ's running time for large matrix multiplication per window is 1.48 ms, 178.98 ms, and 5.58 ms for the minimum, maximum, and average time, respectively. We test the latest large matrix multiplication techniques such as OpenCV's *cv::gemm*, Eigen's large matrix multiplication, C Basic Linear Algebra Subprograms' (CBLAS) *cblas_sgemm* which belongs to Intel MKL.

First, we test with OpenCV version 3.4.10's large matrix multiplication. The OpenCV time for large matrix multiplication per window is 24.32 ms, 311.8 ms, and 55.71 ms for the minimum, maximum, and average time per window, respectively. That performance is worse than the result of MusiJ.

Second, we adopt Eigen version 3.3.7's large matrix multiplication. The Eigen time for large matrix multiplication per window is 0.15 ms, 193.96 ms, and 11.72 ms for the minimum, maximum, and average time per window, respectively. The result of Eigen is better than that of OpenCV; however, it is still worse than the result of MusiJ.

Third, we adopt CBLAS's large matrix multiplication technique from Intel MKL version 2020.2.254. CBLAS's running time for large matrix multiplication per window is 0.23 ms, 29.26 ms, and 2.41 ms for the minimum, maximum, and average time, respectively. Finally, with this technique, we achieve the best performance compared with the others, such as the large matrix multiplication techniques in Java, OpenCV, and Eigen. Hence, we choose CBLAS of Intel MKL for our large matrix multiplication.

We run the entire program with various large matrix multiplication techniques (Table 3). Our MUSICAL in C++ with large matrix multiplication using CBLAS / Intel MKL gives us the best performance compared with other techniques, such as using Eigen, OpenCV, or Java where OpenBLAS was adopted. By combining Tables 2 and 3, Fig. 6 shows that MUSICAL's speed-up is proportional to the large matrix multiplication's speed-up.

TABLE 3. Multiple signal classification algorithm's (MUSICAL) running time using various large matrix multiplication techniques with stack image size 160 × 160 × 49.

MUSICAL using different large matrix multiplication techniques	Running time (s)
MusiJ with *mmul* (Java)	25.91
MUSICAL *cv::gemm* (OpenCV)	153.43
MUSICAL using Eigen	42.69
MUSICAL using *cblas_sgemm* (CBLAS / MKL)	**12.85**

* The bold underlining represents the best performance.

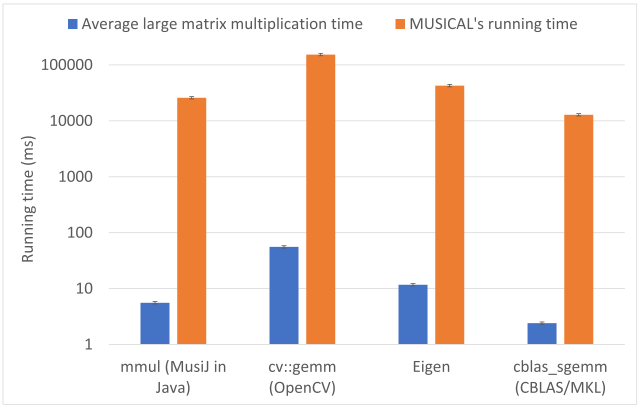


FIGURE 6. Proportionality of multiple signal classification algorithm (MUSICAL) speed-up relative to that of large matrix multiplication techniques for stack image size 160 × 160 × 49.

Through this analysis, this paper provides the insight that MUSICAL's performance mainly depends on the performance of large matrix multiplication algorithms, disproving the myth that MUSICAL's performance is dominated by SVD [17].

In general, matrix multiplication is time-consuming in terms of the order of $O(n^3)$. However, when applying several techniques such as optimizing cache, shared-memory parallelism, etc., the currently smallest running time for matrix multiplication is $O(n^{2.3728596})$ [20] and it is applied in CBLAS. That is further evidence that CBLAS / Intel MKL has the best performance compared with others.

D. PERFORMANCE ANALYSIS

We have referred to an open-source MusiJ implemented in Java [16]. We implement our MUSICAL in C++ and

TABLE 4. Performance analysis and optimization for our multiple signal classification algorithm in C++ with stack image size 160 × 160 × 49. 3D = three-dimensional; ROI = region of interest.

Contents	Language / Function	Minimum time (ms)	Maximum time (ms)	Average time (ms)
9) Initial variables per thread	Java	228	250	242.25
	C++	7.52	17.3	**<u>10.92</u>**
11) 3D crop per window	Java	0	143	0.49
	C++	0.0028	8.99	**<u>0.33</u>**
12) Calculates singular values, singular vectors per window	Java	0.69	12.93	1.19
	C++ using Eigen	0.39	48.37	0.81
	C++ using OpenCV	0.33	33.81	0.66
	C++ using CBLAS / Intel MKL	0.23	15.45	**<u>0.62</u>**
13) Determines row position per window	Java	0.003	0.19	0.0014
	C++	0	0.0093	**<u>0.0013</u>**
14) Gets ratio and power per window	Java	1.48	178.98	5.58
	C++ using OpenCV	24.32	311.8	55.71
	C++ using Eigen	0.15	193.96	11.72
	C++ using CBLAS / Intel MKL	0.23	29.26	**<u>2.41</u>**
15) Rotates 180 degrees per window	Java	0.026	21.44	0.14
	C++	0.006	5.38	**<u>0.018</u>**
16) Updates ROI per window	Java	0.2	201.87	0.79
	C++	0.046	19.33	**<u>0.081</u>**

* The bold underlining represents the best performance.

optimize our program based on the performance analysis. To analyze the true performance of individual components in MUSICAL, we implement our MUSICAL in C++ to remove the overhead of high-level programming frameworks such as Java, Python, and Matlab. That counts as our contribution.

Most of the running time of our MUSICAL is used for the blocks on the right-hand side in Fig. 1. Hence, we should focus on those blocks for performance optimization. Table 4 shows the results of the performance analysis and the program optimization for our MUSICAL in C++. Since there are two loops there, we show the minimum, maximum, and average time per thread or window. Table 4 and Fig. 7 compares the average time per window of each step's inside the loop of our MUSICAL in C++ and MusiJ in Java.

We improve the current implementation of MUSICAL to analyze the true impact of individual components on MUSICAL's performance. For example, we move CCD mask mapping outside the loop. In this way, the implementation time for step 9) initial variables per thread is minimized to 10.92 ms compared with a time in Java of 242.25 ms. Our new MUSICAL also outperforms MusiJ in Java in step 11) 3D crop, 13) determine row position, 15) rotate 180 degrees, and 16) update region of interest (ROI) as 0.33 ms, 0.0013 ms, 0.018 ms, and 0.081 ms compared with 0.49 ms, 0.0014 ms, 0.14 ms, and 0.79 ms, respectively.

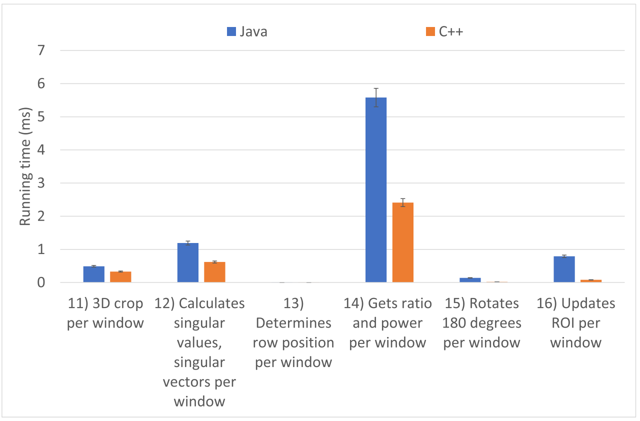


FIGURE 7. Comparing average time per window of each step's inside loop of our multiple signal classification algorithm in C++ and MusiJ in Java. 3D = three-dimensional; ROI = region of interest.

As mentioned, the best performances in calculating singular values and singular vectors in step 12, and obtaining the ratio and power in step 14, belong to C++ using CBLAS / Intel MKL compared with those using Java, Eigen, and OpenCV. For calculating singular values and singular vectors per window, CBLAS / Intel MKL uses 0.62 ms on average while Java, Eigen, and OpenCV consume 1.19 ms and 0.81 ms, and 0.66 ms, respectively.

Similar to obtaining ratio and power, the average running time per window of CBLAS / Intel MKL is 2.41 ms while Java, OpenCV, and Eigen use 5.58 ms, 55.71 ms, and 11.72 ms, respectively. As a result, we use our new C++ version of MUSICAL with MKL to analyze the actual performance of individual components in MUSICAL.

From Table 4, we compute the time percentage of each component of our MUSICAL as shown in Fig. 8. We can count three blocks using the most running time, such as 14) obtaining ratio and power (69.65%), 12) calculating singular values and singular vectors (17.92%), and 11) 3D cropping (9.54%). This means that MUSICAL's performance mainly depends on the performance of large matrix multiplication in step 14) obtaining ratio and power.

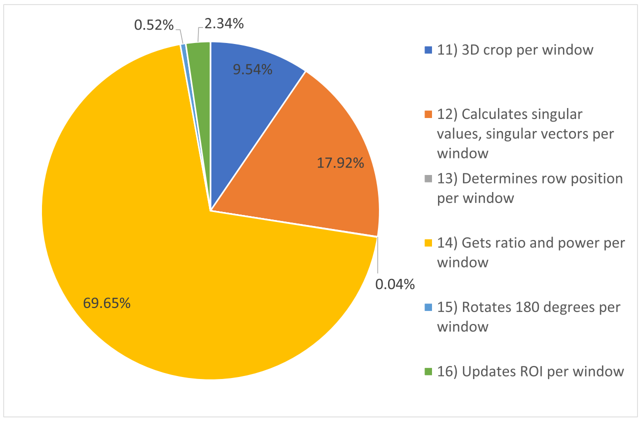


FIGURE 8. Time percentage of each component in the main multiple signal classification algorithm execution. 3D = three-dimensional; ROI = region of interest.

We apply our proposed auto-threshold algorithm, the multi-threading technique of TBB, and the high-performance computing technique for large matrix multiplication of CBLAS / Intel MKL to our MUSICAL in C++. In addition, we optimize our implementation. Fig. 9 shows the performance analysis of our MUSICAL in C++ on stack image 160 × 160 × 49.

On the left-hand side of this figure, the program runs through each block only once. However, on the right-hand side of this figure, the program runs through each block, where the number of times is equal to the number of pixels of the input stack image. For this reason, there are two loops on the right-hand side of this figure, indicating the multi-threading loop and the scanning window loop for all pixels.

Since we use GUI for convenience and a buffer to keep the current variable values, we can minimize the running time for the PSF mapping matrix in step 2 for the second time running. In addition, we can save that matrix to a file. If users do not change the input parameters, the running time for this step is reduced to the loading time of the file or the loading time of data from the buffer.

Making a mapping CCD mask in step 4 is also time-consuming since it has large matrix multiplication where we apply CBLAS of Intel MKL. Since it occurs only once, the running time for this step is not significant.

Our proposed auto-threshold algorithm in step 6 takes 3.89 ms, once. That running time is small; hence, our proposed algorithm is simplified. For calculating a singular value matrix, we can reduce that time by reducing the number of scanning windows. For example, we can use a gating technique in which the input stack image is scanned 16 times, for example, in the vertical direction. That minimizes the running of this step. However, that step occurs once, and it is not the intended scope of this paper.

On the right-hand side of this figure, there are two loops. Hence, adopting high-performance and parallel programming inside these loops is important. We use the best multi-threading technique, i.e., TBB, to save a maximum of 10.47% of time consumed as mentioned in Section III.B. Next, the squaring matrix for calculating singular values and singular vectors also saves time as presented in [17]. In particular, applying the large matrix multiplication of CBLAS / Intel MKL in step 14 gives us 2.41 ms per window, which reduces by half the time of that step compared with MusiJ, as shown in Table 4. That is a significant improvement in MUSICAL's performance.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

Experiments for this paper were undertaken on an HP Z4 G4 workstation with Intel Xeon W-2123 3.6 GHz CPU, 32 GB RAM, 8 logical processors, 4 cores, and a 64-bit system. The number of threads used by our MUSICAL program was 8.

To check the improvement of our MUSICAL in C++ compared with other recent research, we chose an input stack image that was large enough and was suitable for the waiting test time. Then, we chose four input stack images: LiveCellMicrotubulesSample1.tif (144 × 144 × 49), InVitroSample1.tif (160 × 160 × 49), InVitroSample2.tif (360 × 208 × 49), and 30.08.19-14.26.00.tif (2048 × 2048 × 500). To simplify, we call these 144 × 144×49, 160 × 160 × 49, 360 × 208 × 49, and 2048 × 2048 × 500 for later presentation of results. We undertook experiments five times with these images, recorded the average time results (Tables 5-6), and indicate the error bars in Figures 10-11.

We used the same input parameters for programs in Matlab 2019b, Java, Python, and C++. The optical parameters of 144 × 144 × 49, 160 × 160 × 49, and 360 × 208 × 49 were emission wavelength 510 nm, numerical aperture 1.49, magnification 100, pixel size 6500 nm, and window size 7. In addition, the optical parameters of 2048 × 2048 × 500 were emission wavelength 640 nm, numerical aperture 1.2, magnification 1, pixel size 108 nm, and window size 7. All input stack images used the same MUSICAL parameters, such as the activated auto-threshold option, alpha factor 4, subpixels per pixel 20.

A. PERFORMANCE EVALUATION

We compared the implementations in Matlab, Java, Python, and our C++ program in terms of their running time. This provided critical insight to show how our program

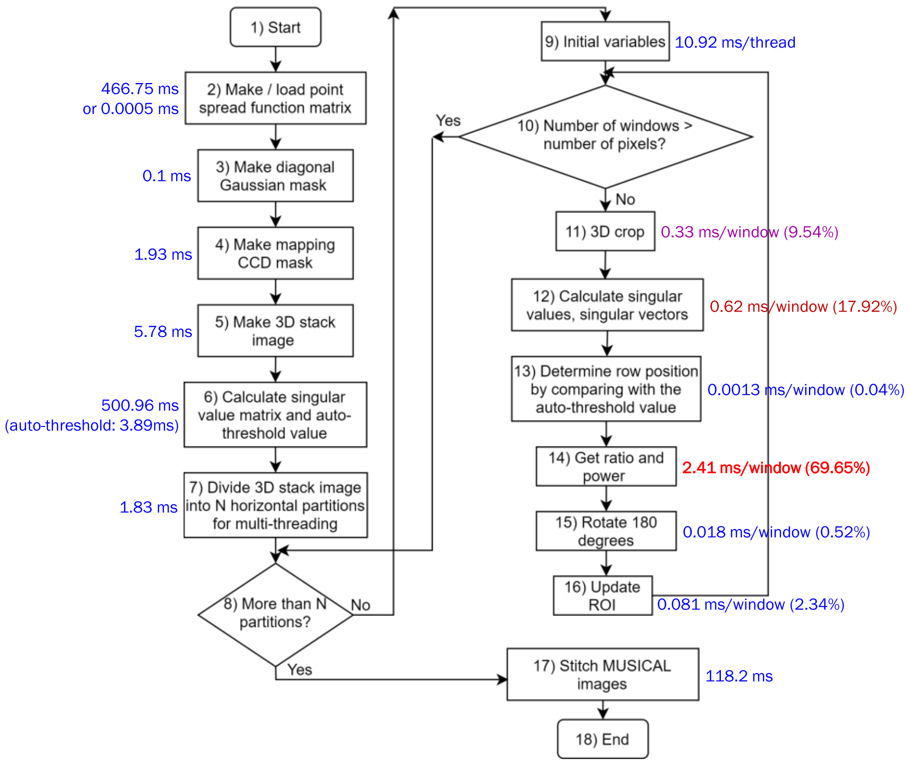

1) Start
2) Make / load point spread function matrix
466.75 ms or 0.0005 ms
3) Make diagonal Gaussian mask
0.1 ms
4) Make mapping CCD mask
1.93 ms
5) Make 3D stack image
5.78 ms
6) Calculate singular value matrix and auto-threshold value
500.96 ms (auto-threshold: 3.89ms)
7) Divide 3D stack image into N horizontal partitions for multi-threading
1.83 ms
8) More than N partitions?
No
Yes
9) Initial variables
10.92 ms/thread
10) Number of windows > number of pixels?
Yes
No
11) 3D crop
0.33 ms/window (9.54%)
12) Calculate singular values, singular vectors
0.62 ms/window (17.92%)
13) Determine row position by comparing with the auto-threshold value
0.0013 ms/window (0.04%)
14) Get ratio and power
2.41 ms/window (69.65%)
15) Rotate 180 degrees
0.018 ms/window (0.52%)
16) Update ROI
0.081 ms/window (2.34%)
17) Stitch MUSICAL images
118.2 ms
18) End

FIGURE 9. Performance analysis of our multiple signal classification algorithm (MUSICAL) in C++ with stack image 160 × 160 × 49. 3D = three-dimensional; CCD = charge-coupled device; ROI = region of interest.

TABLE 5. Our multiple signal classification algorithm's running times compared with other methods. S_{MC++}, S_{JC++}, and S_{PC++} represent the increasing time of Matlab, Java, and Python compared with C++, respectively. NA = not applicable.

Stack image size	Threads	Matlab [15] (s)	Java [16] (s)	Python [17] (s)	Ours in C++ (s)	S_{MC++}	S_{JC++}	S_{PC++}
144×144×49	1	277.87	50.27	30.17	26.20	10.61	1.92	1.15
	4	277.87	26.51	17.38	12.37	22.47	2.14	1.41
	8	277.87	23.43	16.15	**10.48**	**26.51**	**2.24**	**1.54**
160×160×49	1	391.12	66.2	36.2	32.81	11.92	2.02	1.1
	4	391.12	29.14	21.02	14.84	26.36	1.96	1.42
	8	391.12	25.91	20.18	**12.85**	**30.43**	**2.02**	**1.57**
360×208×49	1	1038.76	238.68	105.85	104.28	9.96	2.29	1.01
	4	1038.76	122.33	57.75	46.58	22.30	**2.63**	1.24
	8	1038.76	76.40	52.56	**39.67**	**26.18**	1.93	**1.32**
2048×2048×500	1	NA	NA	6854.14	5887.02	NA	NA	1.16
	4			4879.42	2885.59			**1.69**
	8			3068.06	**2387.18**			1.29

* The bold underlining represents the best performance.

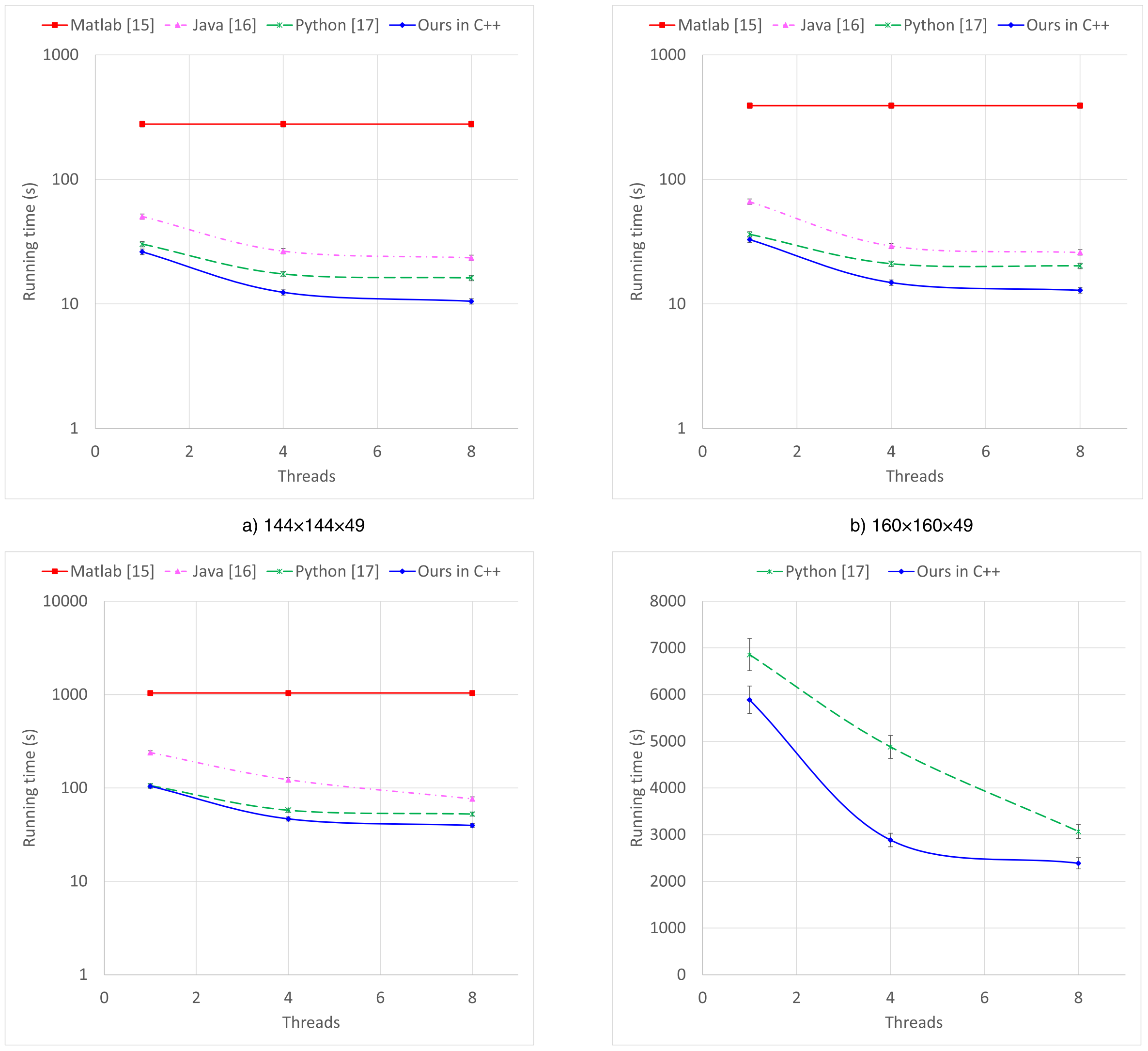
was optimized. Using the whole possible hardware resource was also important. In these experiments, we tested with 1, 4, and 8 threads; then we determined what number of threads provided the best performance for multi-threading.

In Table 5, we define a speed-up factor, S_{XY}, of our program compared with the other reference programs implemented in Matlab, Java, and Python as follows:

$$S_{XY} = \frac{T_X}{T_Y}, \qquad (6)$$

where T_X and T_Y are the running times of implementations X and Y. S_{XY} indicates the factor with which the implementation Y performs faster than the implementation X. In detail, S_{MC++}, S_{JC++}, and S_{PC++} represent the comparative advantage of C++ implementation over Matlab, Java, and Python implementations, respectively.

Table 5 shows us that our MUSICAL in C++ used the optimal TBB's multi-threading technique and high-performance computing in CBLAS / Intel MKL's large



a) 144×144×49 b) 160×160×49 c) 360×208×49 d) 2048×2048×500

FIGURE 10. Our multiple signal classification algorithm's running time curves compared with other methods.

matrix multiplication. The optimized program gave us the best results compared with other programs using Matlab, Java, and Python in all thread numbers. Among different numbers of threads, using 8 threads provided us the best results.

With stack image 144 × 144 × 49, our program used a minimized time in 8 threads of 10.48 s and Matlab, Java, and Python had a running time of more than 26.51x, 2.24x, and 1.54x, respectively. To guarantee that our C++ program showed improvement compared with other programs in Matlab, Java, and Python, we tested with other sizes. With our MUSICAL in C++ and 8 threads with stack image 160 × 160 × 49, our program provided a minimized running time of 12.85 s. Matlab, Java, and Python used a running time of more than 30.43x, 2.02x, and 1.57x, respectively. In experiments with 360 × 208 × 49 we obtained a minimized running time for our program of 39.67s in 8 threads, while Matlab, Java, and Python had a running time of more than 26.18x, 1.93x, 1.32x, respectively; using 4 threads, our program used shorter time than Java by 2.63x.

We tested for another input stack image of size 2048 × 2048 × 500, the largest input stack image we had. We undertook similar experiments for various numbers of threads and all the other implementations such as Matlab, Java, and Python. However, there were limitations for libraries and programs in Matlab and Java—we could not get their results for those two programs for the large input stack image. Hence, in Table 5, we only show the results of Python and C++.

In Table 5, our MUSICAL in C++ shows better performance compared with that in Python for all numbers of threads for stack image 2048 × 2048 × 500. Our MUSICAL in C++ used 2387.18s with 8 threads, while the running time in Python was 1.29x greater, and it was 1.69x greater for 4 threads. The speed-up performance of our MUSICAL in

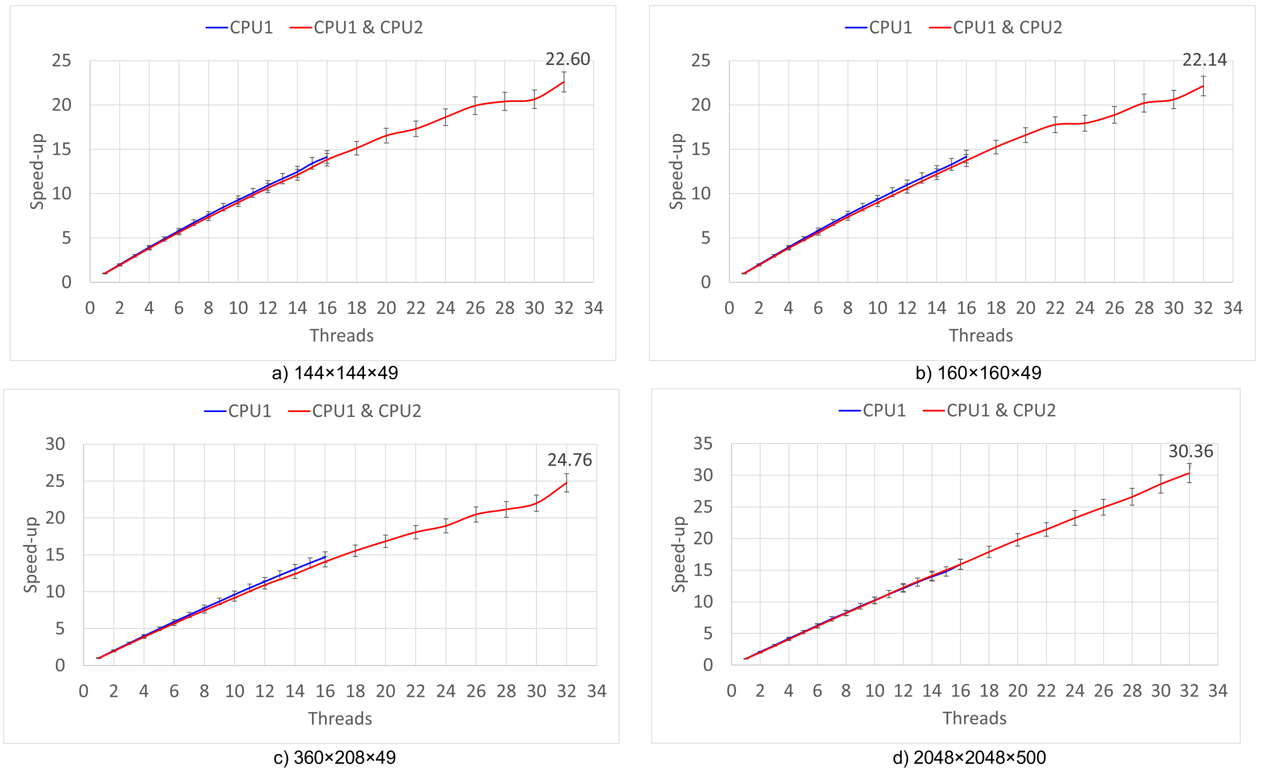


FIGURE 11. **Our multiple signal classification algorithm's scalability on our Linux server.**

C++ with 8 threads was not good compared with that of 4 threads for two reasons:

- The CPU cache per core was 2.06 MB, while the cache requirement for subpixel = 20 was 3.66MB. Because of a lack of high-speed memory, the program used lower-speed memory, such as RAM.
- The CPU speed was reduced when using more than 2 cores in W-2123 CPU frequency behavior since the AVX512's instructions for multiplications of floating-point numbers created a lot of heat, forcing the cores to clock down.

However, the performance of our MUSICAL in C++ was better than in the other programs. This was due to the best multi-threading technique of Intel TBB and the best large matrix multiplication in CBLAS / Intel MKL. On the other hand, MUSICAL in Python uses classic *pool* and *async* for the multi-threading technique and NumPy for large matrix multiplication.

To demonstrate the improvement obtained by using our method, we present the running time of MUSICAL programs in Matlab, Java, Python, and C++ in Fig. 10. The blue curve at the bottom of Fig. 10 shows that our MUSICAL in C++ outperformed that in Matlab, Java, and Python in terms of the running time, with the same input conditions and the same system platform.

Through the experimental results of four input stack images for programs in Matlab, Java, Python, and C++, our MUSICAL in C++ provided us the best results compared with the others. It outperformed when the number of threads was 1, 4, or 8, especially 8. Hence, we conclude that our MUSICAL in C++ provided us the best performance in terms of the running time, with the same input conditions and the same system platform.

B. SCALABILITY EVALUATION

To test the scalability of MUSICAL in an upgrading system, we transferred the MUSICAL program to our Linux server. We used a Red Hat Linux server with Intel Xeon Gold 6130 processor, CPU 2.1 GHz, 32 cores, 2 sockets, 16 cores per socket, CPU minimum speed 1 GHz and CPU maximum speed 3.7 GHz, and a 64-bit system. The software and libraries used were g++ 10.2.1 and OpenCV 3.4.14, respectively, where MKL and TBB should be activated ON (Intel TBB and Intel MKL version 2020.4.304).

We experimented with our Linux server as in Visual Studio C++ 2019, Windows 10, with the four stack images. We ran the experiments five times, obtained the average running time and the speed-up factor as defined in Eq. (6), and present the error bars in Fig. 11. To improve scalability and parallelism, we took the following steps:

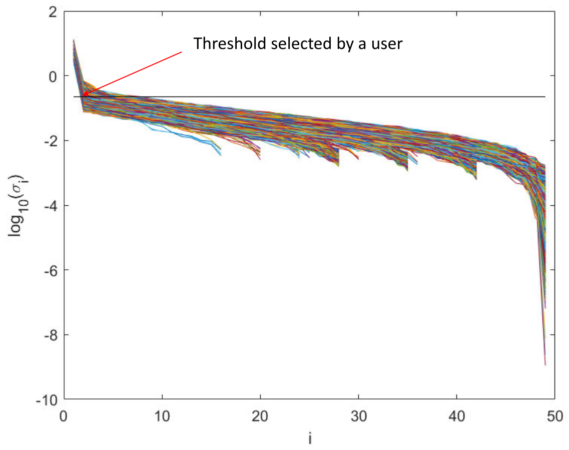
- ***Big enough number of partitions:*** Since scalability depends on the computational load and the number of partitions to enable parallelism, the number of partitions should be 300–500 times the number of cores, i.e., 10368 partitions for stack image $144 \times 144 \times 49$, 12800 partitions for stack image 160 × 160 × 49, 18720 partitions for stack image 360 × 208 × 49, and 16384 partitions for stack image 2048 × 2048 × 500.

- ***Last-Level Cache (LLC) misses less than 5%:*** We used the *perf stat -r 5 -e task-clock, cycles, instructions, LLC-loads, LLC-load-misses* function to monitor utilized cores and LLC misses. Through our experiments, to improve the scalability we also guaranteed LLC misses of less than 5%, which did not hurt overall performance.
- ***Cache balance:*** Our Linux server used an Intel Xeon Gold 6130 processor, which has a 22 MB cache for 16 cores, i.e., a 1.375 MB cache for each core. However, MUSICAL needs a 3.66 MB, 2.06 MB, and 0.92 MB cache for the CCD mask of subpixel factors 20, 15, and 10, respectively. When we used a high subpixel factor, for example 20 or 15, the efficiency of cores was high, but LLC misses were more than 5%. When we used a small subpixel factor, for example 10, the efficiency of cores was not very high, but we guaranteed that LLC misses were less than 5%. To balance the scalability, CPU cache, and cache requirement of MUSICAL, we used a subpixel factor of 10.
- ***Using the AVX2 instructions:*** We chose the AVX2 instructions when we built OpenCV on our Linux server. With that option, we used 256-bit vectors instead of 512-bit vectors, which made overheat and clock down.
- ***Optimal CPU speed:*** The Intel Xeon Gold 6130 processor reduced speed when we increased the number of cores from 3.5 GHz to 1.9 GHz [21], since the AVX2's or AVX512's instructions for multiplications of floating-point numbers created a lot of heat, forcing the cores to clock down. To assess the scalability while discounting this effect, we clamped CPU speed such that all cores should use the same speed as 1.9 GHz by using function: *sudo cpupower -c all frequency-set -u 1900MHz*. Finally, our new parallel MUSICAL achieved a speed-up of up to 30.36x on our Linux server with 32 cores (efficiency 94.88%) as shown in Fig. 11 and a minimized running time of stack image 2048 × 2048 × 500 in 99.59 s. Other images shared in [15] have similar scalability results.

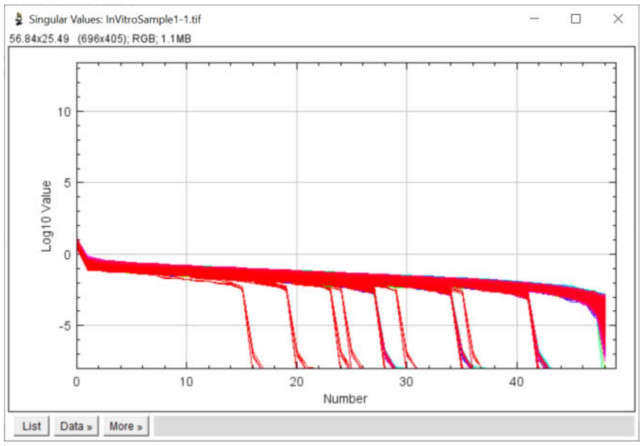
Through the results in Table 5 and Figs. 10 and 11, we conclude that if we upgrade the system platform, i.e., increase the number of cores together with their caches, our MUSICAL's running time will decrease in linearity.

C. AUTO-THRESHOLD EVALUATION

In [15]–[17], there was no auto-threshold algorithm. Users chose a manual threshold based on a singular value matrix graph [15] and [16], as shown in Fig. 12, or directly put a threshold value in a text file based on their experience [17]. It is inconvenient, heuristic, and time-consuming to take at least several seconds to choose a threshold compared with an



a) Singular value matrix graph to choose a threshold in [15].


Singular Values: InVitroSample1-1.tif
56.84x25.49 (696x405); RGB; 1.1MB
Log10 Value
Number
10
5
0
-5
0
10
20
30
40
List
Data »
More »

b) Singular value matrix graph to choose a threshold in [16].

FIGURE 12. Manual threshold selection in the latest researches.

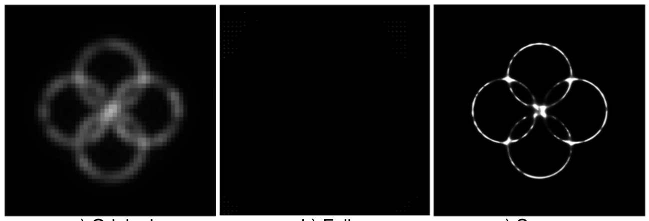


FIGURE 13. Evaluation example for the proposed auto-threshold algorithm.

auto-threshold time of 3.89 ms as shown in Fig. 9. If users choose a threshold without a singular value matrix graph, it can sometimes be wrong threshold and then the object cannot be reconstructed in the MUSICAL results, for example in Fig. 13b.

Since a threshold is a singular value to divide a singular value matrix into two parts, i.e., a signal part and a noise part, we can choose a number, for example 0.1. Then, it may work, although not perfectly for all cases. To guarantee that

TABLE 6. Auto-threshold results. MUSICAL = multiple signal classification algorithm.

No.	Stack image name	Size	Auto-threshold value	Evaluation	MUSICAL running time [s]
1	SynEx1SNR128.tiff	7×7×49	0.00152015	Success	0.0050
2	SynEx1.tiff	7×7×49	0.00151358	Success	0.0051
3	SynPairDelX208Angstrom.tif	25×25×49	0.00288495	Success	0.51
4	SynPairDelX417Angstrom.tif	25×25×49	0.0026055	Success	0.52
5	SynPairDelX217Angstrom.tif	25×25×49	0.00309682	Success	0.52
6	SynPairDelX200Angstrom.tif	25×25×49	0.00275534	Success	0.53
7	SynPairDelX222Angstrom.tif	25×25×49	0.0024801	Success	0.53
8	SynPairDelX250Angstrom.tif	25×25×49	0.00244621	Success	0.54
9	SynPairDelX238Angstrom.tif	25×25×49	0.00322375	Success	0.54
10	SynPairDelX455Angstrom.tif	25×25×49	0.00251692	Success	0.54
11	SynPairDelX333Angstrom.tif	25×25×49	0.0029704	Success	0.54
12	SynPairDelX714Angstrom.tif	25×25×49	0.00370721	Success	0.55
13	SynPairDelX263Angstrom.tif	25×25×49	0.00337054	Success	0.55
14	SynPairDelX556Angstrom.tif	25×25×49	0.00296404	Success	0.55
15	SynPairDelX313Angstrom.tif	25×25×49	0.00334463	Success	0.55
16	SynPairDelX625Angstrom.tif	25×25×49	0.00372205	Success	0.55
17	SynPairDelX385Angstrom.tif	25×25×49	0.00339588	Success	0.55
18	SynPairDelX286Angstrom.tif	25×25×49	0.0026371	Success	0.55
19	SynPairDelX833Angstrom.tif	25×25×49	0.00380135	Success	0.56
20	SynPairDelX233Angstrom.tif	25×25×49	0.00264593	Success	0.56
21	SynPairDelX204Angstrom.tif	25×25×49	0.00238913	Success	0.56
22	SynPairDelX294Angstrom.tif	25×25×49	0.00242346	Success	0.56
23	SynPairDelX357Angstrom.tif	25×25×49	0.00272888	Success	0.56
24	SynPairDelX303Angstrom.tif	25×25×49	0.00253208	Success	0.56
25	SynPairDelX213Angstrom.tif	25×25×49	0.00320794	Success	0.56
26	SynPairDelX500Angstrom.tif	25×25×49	0.00342988	Success	0.57
27	SynPairDelX227Angstrom.tif	25×25×49	0.00282699	Success	0.57
28	SynPairDelX256Angstrom.tif	25×25×49	0.00319412	Success	0.57
29	SynPairDelX1000Angstrom.tif	25×25×49	0.00458454	Success	0.57
30	SynPairDelX244Angstrom.tif	25×25×49	0.00306076	Success	0.58
31	SynPairDelX278Angstrom.tif	25×25×49	0.00326493	Success	0.58
32	SynPairDelX270Angstrom.tif	25×25×49	0.00274132	Success	0.58
33	SynPairDelX323Angstrom.tif	25×25×49	0.00270819	Success	0.58
34	SynEx3_10SBR40.tiff	27×27×49	0.0259865	Success	0.61
35	SynEx3_10SBR18.tiff	27×27×49	0.110823	Success	0.61
36	SynEx3.tiff	27×27×49	0.00044154	Success	0.61
37	SynEx3_10SBR20.tiff	27×27×49	0.0836184	Success	0.62
38	SynEx3_10SBR30.tiff	27×27×49	0.039332	Success	0.62
39	SynEx3_10SBR13.tiff	27×27×49	0.167393	Success	0.64
40	SynEx3_10SBR70.tiff	27×27×49	0.00993311	Success	0.65
41	SynEx3_10SBR60.tiff	27×27×49	0.0124621	Success	0.65
42	SynEx3_10SBR14.tiff	27×27×49	0.158246	Success	0.65
43	SynFork2.tif	27×27×49	0.163701	Success	0.66
44	SynEx3_10SBR15.tiff	27×27×49	0.149957	Success	0.67
45	SynFork1.tif	27×27×49	0.122017	Success	0.67
46	SynEx3_10SBR50.tiff	27×27×49	0.0179873	Success	0.67
47	SynPeriod.tiff	27×27×49	0.0714005	Success	0.67
48	SynEx3_10SBR19.tiff	27×27×49	0.090201	Success	0.68
49	SynEx3_10SBR90.tiff	27×27×49	0.00637689	Success	0.69
50	SynEx3_10SBR16.tiff	27×27×49	0.125823	Success	0.70
51	SynEx3_10SBR80.tiff	27×27×49	0.00800208	Success	0.71
52	SynEx3_10SBR17.tiff	27×27×49	0.0971499	Success	0.71
53	SynEx3_10SBR11.tiff	27×27×49	0.197111	Success	0.73
54	SynEx3_10SBR100.tiff	27×27×49	0.00566331	Success	0.73
55	SynEx3_10SBR12.tiff	27×27×49	0.184795	Success	0.74
56	SynEx2.tiff	51×51×49	0.0791619	Success	1.05
57	SynEx2SNR8.tiff	51×51×49	0.0921795	Success	1.07
58	SynEx2SBR3.tiff	51×51×49	0.0940053	Success	1.17
59	CardioMyoblast_Mitochondria.tif	45×30×49	0.0527213	Success	1.38
60	SynEx4Bck.tiff	51×51×49	0.0567022	Success	1.67
61	014_c00000002-650_775.tif	70×70×49	0.0406796	Success	2.45
62	SynSTORMshortdark.tif	100×100×49	0.0359407	Success	4.96
63	SynSTORMlongdark.tif	100×100×49	0.0316614	Success	5.10
64	InVitroSample3_10ExcitationPower9.tif	108×108×49	0.0832781	Success	6.18
65	InVitroSample3_10ExcitationPower103.tif	108×108×49	0.0593801	Success	6.37
66	InVitroSample3_10ExcitationPower2056.tif	108×108×49	0.0339386	Success	6.47
67	LiveCellFActin.tif	124×124×49	0.0911582	Success	7.95
68	LiveCellMicrotubulesSample2.tif	144×144×49	0.0960844	Success	11.17
69	LiveCellMicrotubulesSample1.tif	144×144×49	0.0521597	Success	10.48
70	InVitroSample1.tif	160×160×49	0.0299559	Success	12.85
71	InVitroSample2_10ExcitationPower402.tif	360×208×49	0.00233917	Success	39.67
72	InVitroSample2_10ExcitationPower103.tif	360×208×49	0.00480433	Success	41.31
73	InVitroSample2_10ExcitationPower2056.tif	360×212×49	0.00543295	Success	41.35
74	30.08.19-14.26.00.tif	2048×2048×500	0.0278712	Success	2387.18

the auto-threshold algorithm may return a bad value, we have "Manual Threshold" option.

To test the accuracy of the proposed auto-threshold algorithm, we set the "Auto-Threshold" option in the GUI of our MUSICAL in C++. For convenience, we selected two numbers of frames, such as 49 frames and 500 frames. The three outputs were auto-threshold value, auto-threshold evaluation, and MUSICAL's running time. Through extensive simulations, we obtained perfect auto-threshold accuracy as shown in Table 6.

To evaluate our proposed auto-threshold algorithm, we tested each case and observed the results. If the contents of objects had super resolution as is the purpose of MUSICAL, it was a successful auto-threshold case. Otherwise, it was a failed case. Fig. 13 demonstrates a successful case and a failed case as an example of auto-threshold algorithm evaluation. If we had used a random threshold, the MUSICAL results may have been the same as the failed auto-threshold results.

Using the 74 stack images in the shared research of [15], we get 100% successful auto-threshold results. Through the experimental results in Table 6, we see that our proposed auto-threshold algorithm is reliable. Hence, we conclude that our proposed auto-threshold is more user-friendly, since it can be used automatically without user interaction and is suitable for real-life applications (auto-threshold time 3.89 ms and 100% successful auto-threshold results).

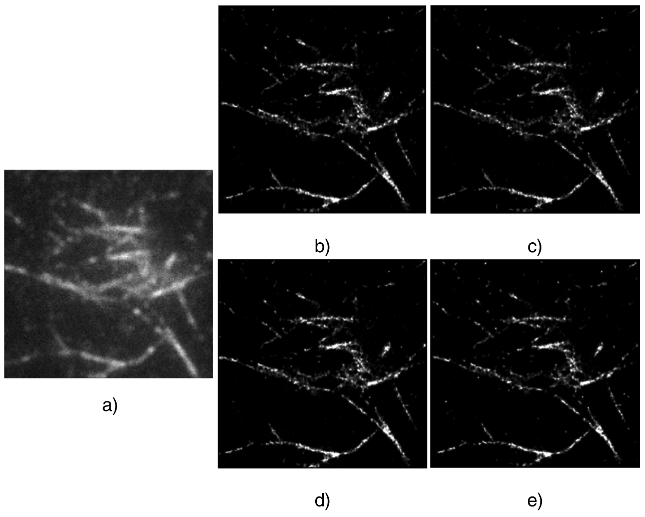


FIGURE 14. InVitroSample1.tif image quality for a) original, b) result in Matlab [15], c) result in Java [16], d) result in Python [17], and e) our result in C++.

D. IMAGE QUALITY

This research focuses on optimizing the running time of MUSICAL while keeping a similar quality of reconstructed MUSICAL output image compared with other previous methods. Hence, we compared the output of our MUSICAL in C++ with that in other recent research, such as programs in Matlab, Java, and Python.

Fig. 14 uses InVitroSample1.tif of size 160 × 160 × 49 as an example of an image quality comparison of our MUSICAL in C++ with other methods. Through running MUSICAL programs in Matlab, Java, Python, and C++, we obtained a super-resolution output image of size 3200 × 3200. We did the same for the other input stack images. The results in Fig. 14 show that our MUSICAL in C++ retained a similar image quality compared with the other MUSICAL programs in Matlab, Java, and Python.

E. FUTURE WORK

Through the scalability analysis, in future work, we will extend our research to a cluster of computers and use graphics processing unit (GPU) for our program. The efficiency of the cluster and GPU approach was also proved in [22]. In addition, a review of computational approaches from the past to the future to obtain super-resolution microscopy is presented in [23].

For a cluster of computers, new programming codes are needed to coordinate the work of all computers together. One candidate is open-source UPC++ developed by the Lawrence Berkeley National Laboratory. Since we can divide the input stack image into partitions, i.e., the maximum number of partitions is the same as the number of pixels, and MUSICAL can enable parallelism with those partitions, a larger number of cores in a cluster can be utilized to speed up MUSICAL. By increasing the number of cores, we may get better results.

Since we need results for real-life applications with large input stack images, i.e., 2048 × 2048 × 500, another choice is using GPU. GPU has its own high-performance parallel programming. In addition, one normal GPU has thousands of cores. These conditions are suitable for MUSICAL's high scalability and parallelism.

V. CONCLUSION

In this paper, the optimization of MUSICAL has been achieved through performance analysis. By using C++ with the latest high-performance computing and parallel programming techniques, our new parallel MUSICAL in C++ outperformed the other MUSICAL versions in Matlab, Java, and Python. With the same input conditions and the same system platform, the optimized MUSICAL in C++ has a running time 30.43x, 2.63x, and 1.69x shorter compared with that in Matlab, Java, and Python, respectively, while the image quality of the MUSICAL output is similar to that in the other programs. In particular, our new parallel MUSICAL achieves a speed-up of up to 30.36x on our Linux 32-core server (efficiency 94.88%). Moreover, we have provided insights into MUSICAL's performance and scalability to determine further performance improvement possibilities, for example exploiting GPUs and high-performance computing clusters. Finally, we have proposed a simplified auto-threshold algorithm to improve MUSICAL's usability. We expect that significantly faster and scalable MUSICAL implementation will enable MUSICAL to perform high-throughput super-resolution microscopy in which the computation time is the main roadblock.

AUTHOR CONTRIBUTIONS

PHH is a principal investigator in the NanoFrame project 302535. QD implemented MUSICAL in C++ in Windows and our Linux server, and wrote several drafts of the manuscript. For more than one year, KA, PHH, and QD had bi-weekly meetings for the NanoFrame 302535 project to create this paper. SA transferred all achievements of MusiJ in Java and MUSICAL in Python. JIK set up software and libraries, and gave comments and advice on MUSICAL in C++ on our Linux server. KA, PHH, and QD revised the manuscript. KA, PHH, and SA commented on the manuscript.

REFERENCES

[1] A. Small and S. Stahlheber, "Fluorophore localization algorithms for super-resolution microscopy," *Nature Methods*, vol. 11, no. 3, pp. 267–279, Mar. 2014.

[2] T. Dertinger, R. Colyer, G. Iyer, S. Weiss, and J. Enderlein, "Fast, background-free, 3D super-resolution optical fluctuation imaging (SOFI)," *Proc. Nat. Acad. Sci. USA*, vol. 106, no. 52, pp. 22287–22292, Dec. 2009.

[3] S. Cox, E. Rosten, J. Monypenny, T. Jovanovic-Talisman, D. T. Burnette, J. Lippincott-Schwartz, G. E. Jones, and R. Heintzmann, "Bayesian localization microscopy reveals nanoscale podosome dynamics," *Nature Methods*, vol. 9, no. 2, pp. 195–200, Feb. 2012.

[4] I. Yahiatene, S. Hennig, M. Müller, and T. Huser, "Entropy-based super-resolution imaging (ESI): From disorder to fine detail," *ACS Photon.*, vol. 2, no. 8, pp. 1049–1056, Aug. 2015.

[5] S. Geissbuehler, A. Sharipov, A. Godinat, N. L. Bocchio, P. A. Sandoz, A. Huss, N. A. Jensen, S. Jakobs, J. Enderlein, F. Gisou Van Der Goot, E. A. Dubikovskaya, T. Lasser, and M. Leutenegger, "Live-cell multiplane three-dimensional super-resolution optical fluctuation imaging," *Nature Commun.*, vol. 5, no. 1, pp. 1–7, Dec. 2014.

[6] L. Zhao, C. Han, Y. Shu, M. Lv, Y. Liu, T. Zhou, Z. Yan, and X. Liu, "Improved imaging performance in super-resolution localization microscopy by YALL1 method," *IEEE Access*, vol. 6, pp. 5438–5446, 2018.

[7] E. A. Mukamel, H. Babcock, and X. Zhuang, "Statistical deconvolution for super-resolution fluorescence microscopy," *Biophys. J.*, vol. 102, pp. 2391–2400, May 2012.

[8] E. Betzig, G. H. Patterson, R. Sougrat, O. W. Lindwasser, S. Olenych, J. S. Bonifacino, M. W. Davidson, J. Lippincott-Schwartz, and H. F. Hess, "Imaging intracellular fluorescent proteins at nanometer resolution," *Science*, vol. 313, no. 5793, pp. 1642–1645, Sep. 2006.

[9] S. T. Hess, T. P. K. Girirajan, and M. D. Mason, "Ultra-high resolution imaging by fluorescence photoactivation localization microscopy," *Biophys. J.*, vol. 91, no. 11, pp. 4258–4272, Dec. 2006.

[10] F. K. Gruber, E. A. Marengo, and A. J. Devaney, "Time-reversal imaging with multiple signal classification considering multiple scattering between the targets," *J. Acoust. Soc. Amer.*, vol. 115, no. 6, pp. 3042–3047, Jun. 2004.

[11] R. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE Trans. Antennas Propag.*, vol. AP-34, no. 3, pp. 276–280, Mar. 1986.

[12] X. Chen and K. Agarwal, "MUSIC algorithm for two-dimensional inverse problems with special characteristics of cylinders," *IEEE Trans. Antennas Propag.*, vol. 56, no. 6, pp. 1808–1812, Jun. 2008.

[13] M. G. L. Gustafsson, "Surpassing the lateral resolution limit by a factor of two using structured illumination microscopy," *J. Microsc.*, vol. 198, no. 2, pp. 82–87, May 2000.

[14] I. S. Opstad, S. Acuña, L. E. V. Hernandez, J. Cauzzo, N. Škalko-Basnet, B. S. Ahluwalia, and K. Agarwal, "Fluorescence fluctuations-based super-resolution microscopy techniques: An experimental comparative study," 2020, *arXiv:2008.09195*. [Online]. Available: http://arxiv.org/abs/2008.09195

[15] K. Agarwal and R. Macháň, "Multiple signal classification algorithm for super-resolution fluorescence microscopy," *Nature Commun.*, vol. 7, no. 1, Dec. 2016, Art. no. 13752.

[16] S. Acuña, F. Ströhl, I. S. Opstad, B. S. Ahluwalia, and K. Agarwal, "MusiJ: An ImageJ plugin for video nanoscopy," *Biomed. Opt. Exp.*, vol. 11, no. 5, pp. 2548–2559, 2020.

[17] S. A. Acuña-Maldonado, "Multiple signal classification algorithm: Computational time reduction and pattern recognition applications," M.S. thesis, Dept. Phys. Technol., UiT Arctic Univ. Norway, Tromsø, Norway, Mar. 2019.

[18] S. Acu?a, I. S. Opstad, F. Godtliebsen, B. S. Ahluwalia, and K. Agarwal, "Soft thresholding schemes for multiple signal classification algorithm," *Opt. Exp.*, vol. 28, no. 23, pp. 34434–34449, 2020.

[19] S. Nanz, S. West, K. S. D. Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2013, pp. 183–192.

[20] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," in *Proc. 32nd Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2021, pp. 522–539.

[21] *Xeon Gold 6130–Intel–WikiChip*. Accessed: Jul. 5, 2021. [Online]. Available: https://en.wikichip.org/wiki/intel/xeon_gold/6130

[22] M. U. Ashraf, F. A. Eassa, A. A. Albeshri, and A. Algarni, ''Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems,'' *IEEE Access*, vol. 6, pp. 23095–23107, 2018.

[23] S. S. Kaderuppan, E. W. L. Wong, A. Sharma, and W. L. Woo, ''Smart nanoscopy: A review of computational approaches to achieve super-resolved optical microscopy,'' *IEEE Access*, vol. 8, pp. 214801–214831, 2020.

QUAN DO received the Ph.D. degree from the Gwangju Institute of Science and Technology, South Korea. He is currently a Postdoctoral Research Fellow with the Department of Computer Science, UiT The Arctic University of Norway. His current research interests include parallel programming, parallel and distributed computing systems, image analysis, and bio-medical image computing.

SEBASTIAN ACUÑA received the M.S. degree from the UiT The Arctic University of Norway. He is currently a Ph.D. Research Fellow with the Department of Physics and Technology, UiT The Arctic University of Norway. His current research interests include computational nanoscopy and super-resolution imaging.

JON IVAR KRISTIANSEN is currently a Senior Engineer with the Department of Computer Science, UiT The Arctic University of Norway.

KRISHNA AGARWAL received the Ph.D. degree from the National University of Singapore. She is currently an Associate Professor with the Department of Physics and Technology, UiT The Arctic University of Norway. Her current research interests include computational nanoscopy, super-resolution imaging, and inverse problems. https://sites.google.com/site/uthkrishth

PHUONG HOAI HA received the Ph.D. degree from the Chalmers University of Technology, Sweden. He is currently a Professor with the Department of Computer Science, UiT The Arctic University of Norway. His current research interests include energy-efficient computing, parallel programming, and distributed computing systems. www.cs.uit.no/∼phuong