Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 11, No. 2, 2025

A Method for Accelerating the Loading of TIFF Files Based on Memory-Mapped File Technology

Hang Lv [1, 2, *], Longlong Zhang [3], Qiusheng Cao [2, 3]

[1] Henan Polytechnic University, Jiaozuo Henan, 454000, China
[2] Henan Academy of Sciences, Zhengzhou Henan, 450046, China
[3] Zhongyuan Research Institute of Electronics Technology, Zhengzhou Henan, 450047, China

*** Corresponding author:** Hang Lv (Email: 1213936911@qq.com)

Abstract: TIFF (Tagged Image File Format) is of vital importance in fields such as aerial remote sensing and computer vision. To address the challenge of real-time loading of large, multi-frame TIFF files, this study proposes FastTIFF, an acceleration method based on memory-mapped files and multi-threading technology in Ubuntu. FastTIFF provides access to each frame's image and its description information for other programs by using address pointers for each frame. Experiments show that, compared with LibTIFF, FastTIFF achieves a maximum loading speed improvement of 94.68%, with significant performance in handling large files. Even in the smallest file test, the speed only decreased by 7.26%, demonstrating its robustness in processing small-scale files. This method effectively meets the real-time requirements of image recognition software.

Keywords: TIFF File; LibTIFF; TinyTIFF; Memory-Mapped File.

1. Introduction

TIFF (Tagged Image File Format) holds a significant position in the field of image file storage due to its lossless compression, support for multi-page documents, and rich tag information, especially in applications such as aerial remote sensing image recognition where its advantages are particularly prominent. In these domains, the real-time loading, feature extraction, and target recognition of image data are crucial to the efficiency of the entire system. The efficiency of these processes is not only affected by the speed of algorithmic recognition but is also constrained by the speed of image loading and memory occupation. Current TIFF file loading methods often become performance bottlenecks due to high memory occupation and slow loading speeds, severely limiting the timeliness of data processing and analysis. Therefore, improving the loading speed of TIFF files has become key to enhancing overall processing efficiency.

LibTIFF, as the industry-standard library, provides functions for reading, writing, modifying, compressing, and decompressing TIFF files, May et al. [1] allowing for operations such as cropping and rotating each frame of the file. However, when the task is limited to loading TIFF files into memory, the LibTIFF library appears overly cumbersome due to its complex functionalities. To address this issue, Jan W. Krieger developed the TinyTIFF library, which simplifies functions and retains only the basic reading and writing operations of TIFF files, significantly improving the read and write speed of TIFF files, especially when dealing with uncompressed, single or multi-frame, and 8 to 64-bit image pixel depth TIFF files, offering superior performance compared to LibTIFF.

Although TinyTIFF has achieved certain results in improving the loading speed of TIFF files, analysis of its source code reveals that it still uses a single-threaded approach when handling multi-frame TIFF files, limiting its performance on multi-core processors. In response to this issue, this paper deeply analyzes the TIFF file format and proposes a method that combines file memory mapping with multi-threading technology to achieve simultaneous loading of multiple images in multi-frame TIFF files, thereby significantly increasing the loading speed of TIFF files.

2. Related Works

2.1. TIFF File Format Parsing

The TIFF file format specification is composed of three key structures: the Image File Header (IFH), the Image File Directory (IFD), and the Directory Entry (DE), which together determine the organization and decoding methods of TIFF files. The IFH, which is fixed at 8 bytes, includes the byte order, the TIFF signature, and the offset to the first IFD Fu et al. [2]. Each IFD corresponds to an image, recording its properties; while DEs provide detailed descriptions of specific image attributes, such as dimensions and compression status [3]. In DEs, values exceeding four bytes are indirectly referenced through offsets, requiring the decoder to locate and read data based on these offsets Liu et al. [4]. This study proposes an efficient decoding method for TIFF files based on file memory mapping technology and multi-threading, significantly improving data processing efficiency and loading speed [5].

2.2. FastTIFF Method Design

FastTIFF is an acceleration method for loading TIFF files based on memory-mapped files and multi-threading technology. It is proposed in this study, based on the research results of LibTIFF and TinyTIFF, aiming to solve the problem of slow loading speeds for large TIFF files. FastTIFF maps file content directly into the virtual memory address space through memory mapping technology, avoiding traditional file I/O operations, thereby improving data access efficiency. At the same time, it uses multi-threading technology to process multiple frames of images in parallel, significantly increasing the loading speed of TIFF files.

2.2.1. Memory Management Strategy

To ensure the economy and efficiency of memory usage, this study adheres to a strict principle of data type matching.

For example, when processing IFH, we use two uint8_t variables to store byte order (II or MM), a uint16_t type variable to store the flag bits, and a uint32_t type variable to store the offset to the first IFD. This precise data type matching helps to reduce memory occupation and improve data access speed.

Referencing the research on C++ efficient memory pools by YAN et al. [6] and the research on software memory management methods by Zhao et al. [7], this study designs five core classes to store and manage file data: ReaderFile, ReaderIFD, ReaderFrame, threadArgs, and ThreadPool, as shown in Fig.1:

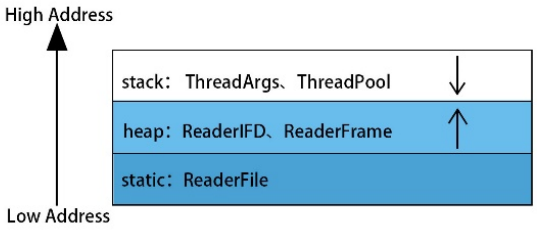


Fig 1. Memory Management

1. ReaderFile Class: This is a static class located in the static storage area, responsible for storing the program's exception information, exception flags, system and file byte order, file size, IFD offset, and the pointer array to frame information frames. Since frames contain addresses in the heap area, these spaces need to be manually released before the program ends to avoid memory leaks.
2. ReaderIFD Class: This class dynamically allocates space in the heap area to store tag, value_type, count information, and the address pointer of value in IFD. After the current IFD data is read, these spaces are released in time to optimize memory usage.
3. ReaderFrame Class: Uses calloc to allocate space in the heap area according to the number of frames contained in the file, used to store relevant information for each frame, such as image width and height, strip offset, strip count, bit depth, pixel padding order, and image address. This information is crucial for subsequent image processing, and this space is released before releasing frames to ensure efficient memory management.
4. ThreadArgs Class: This class is located in the stack area and serves as the parameter object for the thread function, being automatically released as the program runs out, reducing the complexity of memory management.
5. ThreadPool Class: Also located in the stack area and automatically managed and released by the program. This object contains a ReaderFile object pointer tiff, a thread array threads, a thread argument array args, and the maximum number of threads threadsMaxNumber. Through the management of the thread pool, it achieves efficient use of multi-threading resources and dynamic memory allocation.

Through the above memory management strategies, this study not only improves the loading speed of TIFF files but also ensures the efficiency of memory use and the stability of the program. The application of these strategies provides a new solution for large-scale image data processing, which has important academic value and practical significance.

2.2.2. Memory Mapping

In this study, we delve into the application of memory-mapped file technology in improving the efficiency of processing large TIFF files. Based on the research by Huang et al. [8] on memory-mapped files in caching systems and the research by Yang et al. [9] on memory-mapped files in the fast access of large data files, we recognize that memory-mapped technology can effectively reduce data copying, eliminate inter-process communication consumption, and avoid read-write operation locking, thereby significantly improving cache read performance.

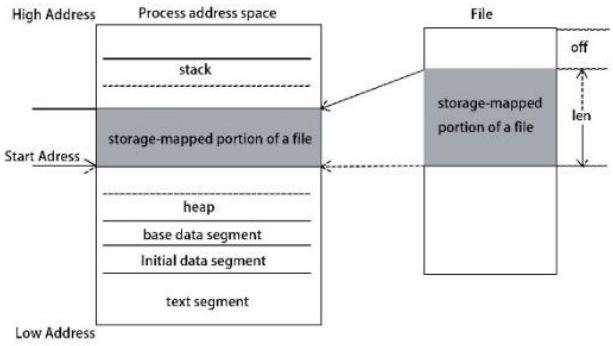


Fig 2. Memory Mapping Mechanism

Memory mapping technology allows a process to directly map file content into the virtual memory address space, thus avoiding traditional file I/O operations. This mapping enables file content to be operated through memory access, improving data access efficiency. The mmap library function provided by the Ubuntu system can map a file into the process's address space, where a segment of the file's memory address corresponds one-to-one with the disk address, achieving direct interaction between the file and memory.

In this study, we designed an optimization strategy aimed at reducing the time consumption in file processing, especially the overhead caused by frequent file pointer movement and thread locking mechanisms. First, by reading the file for the first time, we accurately calculate the total number of IFDs (denoted as IFD_COUNT) and determine the starting position of each IFD segment (marked as thread_begin). Subsequently, using memory mapping technology, the file is mapped to the memory space, a mapped file is generated, and the IFH part is precisely cut off. The remaining part is copied out into IFD_COUNT equal parts, each corresponding to the processing needs of an IFD segment.

This step lays the data foundation for subsequent parallel processing. Finally, these units are allocated to a thread pool, where multiple threads execute the next processing tasks in parallel. Through the above method, we effectively avoid the time waste caused by the single file pointer repeatedly moving inside the file and also reduce the efficiency loss caused by thread locking mechanisms, thereby significantly improving the overall efficiency and performance of file processing.

2.2.3. Thread Pool Design

In this study, aiming to optimize the decoding processing efficiency of large TIFF files, we designed a thread pool mechanism to ensure that even in the face of a large number of IFDs and large image sizes, efficient processing capabilities can be maintained. The core mechanisms of this design include task privatization strategy, lock-free

concurrency control, load balancing allocation, and task stealing algorithm, which are detailed below:

1. Task Privatization Strategy: The task privatization strategy is achieved by creating and maintaining an exclusive task queue for each thread. When the thread pool activates a certain thread object, the thread only retrieves and executes tasks from its exclusive task queue, reducing thread interaction overhead. This strategy helps to reduce lock competition, improve the continuity of thread execution, and thereby enhance the overall parallel processing efficiency.
2. Lock-Free Concurrency Control: Since each thread focuses on processing tasks in its private task queue without accessing shared resources, the performance loss caused by traditional lock mechanisms' lock and unlock operations.
3. Load Balancing Allocation: To achieve an even distribution of tasks, the thread pool employs an intelligent allocation strategy that uniformly distributes new tasks to the private task queues of individual thread objects. If there are remaining tasks, they are directly assigned to the thread pool queue that initiated the allocation, ensuring that the workload of all threads remains relatively balanced, preventing resource idleness or overload. This load balancing strategy helps optimize resource utilization and improve the system's throughput.
4. Task Stealing Algorithm: To further enhance resource utilization and load balancing, the thread pool introduces a task stealing mechanism. This mechanism maintains a thread index system, allowing threads that have completed their current task queues to access neighboring thread objects (or the thread pool's global task queue). If the neighboring thread's task queue is not empty, it steals half of the tasks to continue execution in its own queue; if the neighboring thread's queue is also empty, it attempts to steal tasks from the thread pool's global task queue; if there are no tasks available in the global queue, the thread enters a suspended state, waiting for new task assignments. This dynamic adjustment of task allocation effectively improves the overall resource utilization rate of the system and reduces thread idle time.

3. Methods

3.1. Processing Flow

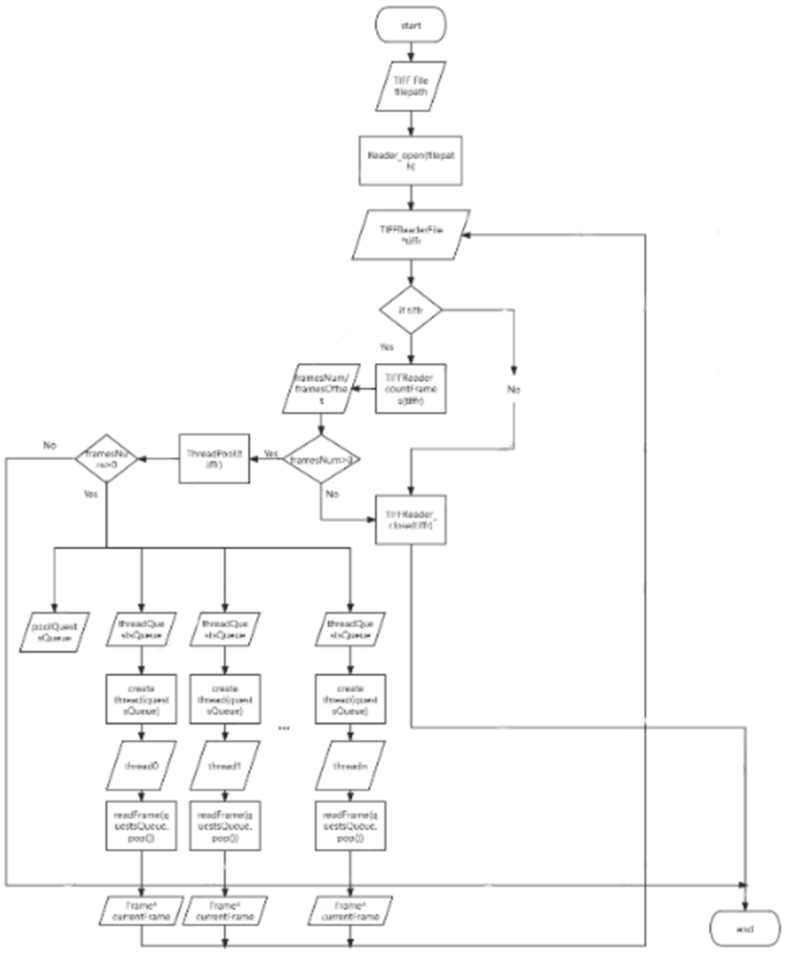


Fig 3. Program Flowchart

In this study, we propose the FastTIFF processing flow as depicted in Fig. 3, which is designed to efficiently handle large TIFF files. This process not only optimizes the file reading and decoding procedures but also significantly enhances overall performance through parallel processing techniques. The following is a detailed description of the FastTIFF processing flow:

1. Initialization and File Reading: The FastTIFF processing flow initially accepts the path of a TIFF file as a parameter and creates a TIFFReaderFile object, denoted as tiffr. It utilizes the stat system call to retrieve the file size and opens the file in read-only binary mode. Subsequently, it obtains crucial information such as system byte order, file byte order, file flag bits, and the offset of the first IFD, storing this information within the

tiffr object. Concurrently, it sets the offset for the first frame and returns the tiffr object, providing a foundational data structure for subsequent processing.

2. Frame Information Acquisition and Task Distribution: By reading the entire file for the first time, we ascertain the number of frames and their respective starting positions within the file. This information is encapsulated as tasks and passed to the thread pool. The thread pool assigns memory-mapped file segments to each thread object and evenly distributes tasks across the task queues of each thread, ensuring all threads operate in parallel, thus fully leveraging the computational resources of multi-core processors.
3. Sub-thread Processing: Each sub-thread creates a ReaderFrame object, referred to as currentFrame, for storing the current frame data. The sub-thread retrieves the number of DEs (Directory Entries) and iteratively calls relevant functions to obtain all DE data within the current IFD, storing it in a TIFFReader_IFD object and returning a pointer. After transferring the data from this object to currentFrame and releasing it to minimize memory usage, it allocates width*height* bitspersample/ 8 bytes of space for currentFrame->image to store the image data. Finally, it places the address of currentFrame in tiffr->frames, completing the processing of a single frame.
4. Task Stealing and Thread Synchronization: When a thread is idle, it attempts to acquire new tasks from the task queue of the neighboring indexed thread or the thread pool's global task queue. This task stealing mechanism aids in dynamically adjusting task distribution and improving resource utilization. All threads enter a suspended state after completing their assigned tasks; when all threads are suspended, it indicates that the file loading is complete. At this point, the TIFFReaderFile object containing all vital data from the TIFF file is returned as shown in Fig. 4.

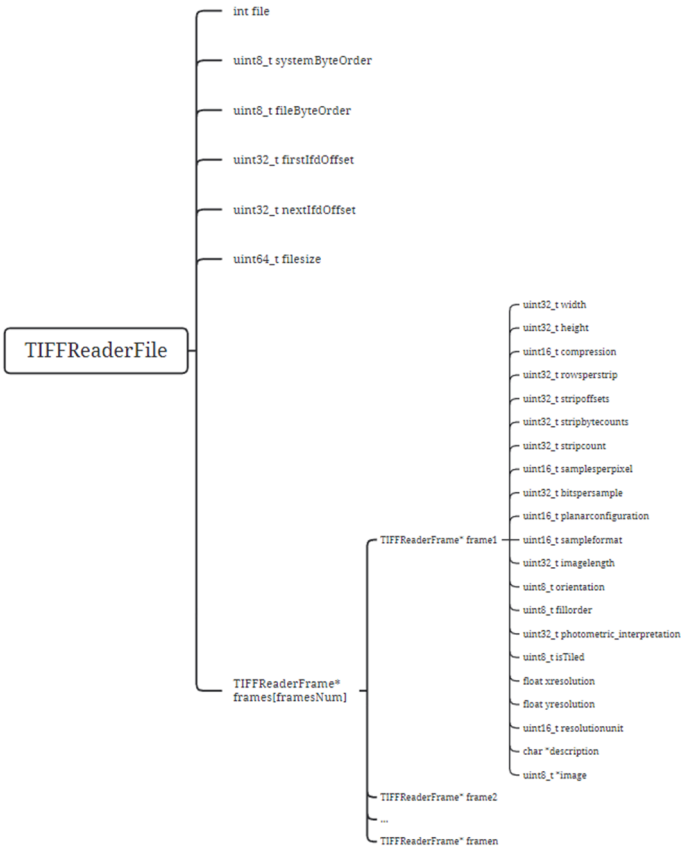


Fig 4. Program Return Object

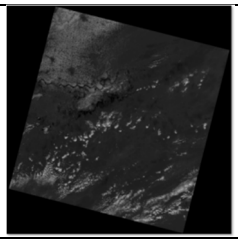
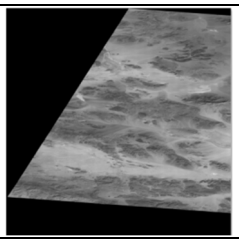
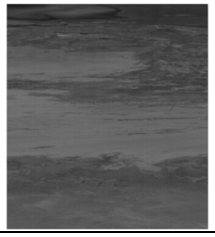
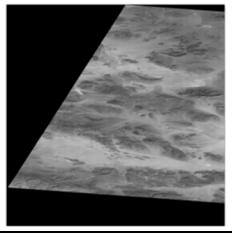
3.2. Experimental Testing

In this study, to comprehensively evaluate the performance of the FastTIFF method, especially in resource-constrained computational environments, we selected the Ubuntu 22.04 operating system as our testing platform. Ubuntu 22.04 is one of the widely used Ubuntu distributions, and its stability and popularity make it an ideal experimental environment. Regarding hardware configuration, we chose the Intel Core i7-7700 processor, a four-core, eight-thread CPU released in 2017. Although its performance is somewhat lacking compared to current mainstream CPUs, it still represents the actual hardware level of a portion of users. Additionally, the system is equipped with 16GB of memory, providing ample running space for the tests. By conducting tests on this configuration, we aim to verify that the FastTIFF method can exert its advantages even on computers with lower performance.

3.2.1. Test Cases

To further verify the effectiveness of the FastTIFF method, we used four representative TIFF files as test data. These data were sourced from the Landsat satellite data of the United States Geological Survey (USGS). The Landsat data are widely recognized in the academic community for their extensive applications and scientific value. Therefore, testing with these data ensures the reliability and practical value of the results. By applying the FastTIFF method to this specific dataset, we can accurately assess its performance in processing large-scale TIFF files. The file information is shown in Table 1.

Table 1. Test Case

				
File name	merged_output1.tif	merged_output2.tif	merged_output3.tif	merged_output4.tif
File size	164.6MB	588.4MB	1.6G	2.9G
Frames count	7	10	10	50
Max frame size	7791*7921	7611*7731	12000*13400	7611*7731

3.2.2. Test Results

The TestFile.tif was decoded and loaded 15 times using FastTIFF, LibTIFF, and TinyTIFF, respectively, with the time taken for each trial recorded and the average duration calculated, as shown in Table 2:

Table 2. Test Results

	merged_output1.tif			merged_output2.tif			merged_output3.tif			merged_output4.tif		
Function	LibTIFF	TinyTIFF	FastTIFF	LibTIFF	TinyTIFF	FastTIFF	LibTIFF	TinyTIFF	FastTIFF	LibTIFF	TinyTIFF	FastTIFF
1	71.427992	86.484534	75.930764	1516.797959	297.225525	95.851077	3867.833763	733.534524	271.097573	8480.676632	1355.654654	437.949213
2	69.71493	87.057204	69.978454	1489.936576	277.948646	96.663622	4026.708479	751.510659	263.393038	8080.375172	1362.12467	444.738988
3	73.55045	88.849141	77.454358	1489.186708	284.108342	108.789711	3865.144893	750.533697	264.588787	8367.506832	1361.066773	474.065056
4	71.649546	80.028899	77.6065	1492.154162	283.664793	94.202312	3837.911635	756.577176	244.971734	8080.782126	1348.135549	440.745061
5	69.680839	89.360645	76.06686	1491.587076	282.241813	102.006733	3921.796196	742.059363	259.419358	8065.462331	1516.075147	476.049998
6	71.188122	89.604142	75.804589	1495.273456	279.458518	97.197847	3942.858153	731.481288	258.028762	8233.438695	1362.284791	431.531202
7	71.608716	89.162744	76.971536	1470.086666	295.542409	97.866864	3888.955185	740.994094	256.686574	8143.007721	1348.734396	440.747216
8	79.154032	86.807607	75.083205	1515.044871	279.29352	100.893773	3904.555386	741.121162	247.812735	8328.422572	1336.058145	445.922861
9	71.473566	81.997844	74.445413	1506.266322	280.756057	101.243595	3885.772009	744.956373	253.641809	8462.439908	1332.158971	441.396248
10	70.135992	83.048198	71.308936	1525.086273	279.32101	133.5219	3837.026788	747.307745	249.867855	8460.447178	1380.768892	440.381709
11	75.736406	86.409789	77.753852	1495.706036	281.328409	92.800082	3843.845875	735.071653	248.611785	8641.435052	1375.198203	442.560894
12	70.099869	86.073803	93.126837	1517.774613	281.410532	98.220131	3890.269388	734.969199	254.354904	8581.607446	1371.624022	442.298068
13	74.204272	86.557452	81.493564	1511.956359	281.162693	101.639146	3874.647159	743.319802	253.446573	8427.734432	1366.595987	436.304606
14	69.037467	81.916632	76.971893	1538.916813	280.461158	107.812369	3870.314311	733.341159	259.921933	8517.761229	1358.654797	431.82303
15	69.956529	80.003144	76.911638	1483.693836	282.037258	95.409625	3953.996027	742.737582	250.607999	8512.726083	1362.493924	443.427745
Mean	71.9079152	85.5574519	77.1272266	1502.63118	283.064046	101.607919	3894.10902	741.967698	255.763428	8358.92156	1369.17526	444.662793
		18.98%â	7.26%â		81.16%á	93.24%á		80.95%á	93.43%á		83.62%á	94.68%á

The results demonstrate that FastTIFF exhibits significant performance advantages in processing large-scale TIFF files.

Specifically, for the file merged_output1.tif (164.6 MB, 7 frames, with the largest frame size of 7791×7921), TinyTIFF showed an 18.98% decrease in read speed compared to LibTIFF, while FastTIFF only decreased by 7.26%. This indicates the robustness of FastTIFF in handling smaller-scale files. For the file merged_output2.tif (588.4 MB, 10 frames, with the largest frame size of 7611×7731), TinyTIFF achieved an 81.16% increase in read speed compared to LibTIFF, whereas FastTIFF achieved a higher increase of 93.24%. This result highlights FastTIFF's superior efficiency in processing medium-scale files.

Furthermore, for the files merged_output3.tif (1.6 GB, 10 frames, with the largest frame size of 12000×13400) and merged_output4.tif (2.9 GB, 50 frames, with the largest frame size of 7611×7731), FastTIFF achieved read speed increases of 93.43% and 94.68% compared to LibTIFF, respectively, while TinyTIFF's increases were 80.95% and 83.62%. These findings suggest that as the file size increases, FastTIFF's performance advantage becomes more pronounced.

4. Conclusion

In this study, we proposed FastTIFF, a rapid loading method for large TIFF files, aiming to enhance the decoding efficiency of TIFF files through memory-mapped technology and multi-threaded parallel processing. Analysis of the test results leads us to the following conclusions:

1. Performance Improvement Verification: FastTIFF consistently outperforms LibTIFF and TinyTIFF in processing TIFF files of varying scales, particularly in handling large-scale files. Its significant performance improvements provide a new technical approach for the field of TIFF file processing and lay a solid foundation for future optimization and applications.

2. Feasibility of the Method: The design of FastTIFF incorporates key technologies such as memory optimization management, lock-free concurrent control, load balancing distribution, and task stealing algorithms. The combination of these technologies enables FastTIFF to effectively reduce inter-thread overhead, avoid lock contention, achieve load balancing, and dynamically adjust task allocation. The integration and optimization of these technologies are key factors in the performance enhancement of FastTIFF.

3. Considerations for Stability and Universality: Although the test results are encouraging, we recognize that the singularity of the testing environment may limit the universality of the results. Therefore, FastTIFF requires further testing for stability and performance under various scenarios and environments, including different operating systems, hardware configurations, file sizes, complexities, and levels of concurrency. Through these tests, we can more comprehensively assess the performance of FastTIFF and verify its stability and reliability in practical applications.

4. Future Work: Building on the current research, future work will focus on further optimization and expansion of FastTIFF. This may include improvements in the management of memory-mapped files and enhancements to thread pool management strategies to increase resource utilization.

In summary, the development of FastTIFF not only provides a new technical solution for the rapid loading of TIFF files but also offers new research considerations for the fields of parallel computing and high-performance computing.

Its application prospects in image processing, computer vision, and remote sensing image analysis are broad, and it holds significant academic value and practical significance for scientific research and industrial applications that require handling large-scale image datasets. Future research will be dedicated to further validating its stability and exploring its applications in a wider range of fields.

Declarations

Ethical and informed consent for data used Written informed consent for publication of this paper was obtained from all authors.

Competing interests the authors declare no competing interests.

References

[1] May P, Davies K. Practical Analysis of TIFF File Size Reductions Achievable Through Compression[C]//iPRES. 2016.

[2] Fu Xichun,Li Kechen, Dong Yi, et al. Multiple image format file generation methods of VSP interpretation results [J]. Information Systems Engineering,2021, (02):151-152. (in Chinese).

[3] Hei Se Tang Zha. (2021). TIFF Image File Format Analysis. CSDN. https://blog.csdn.net/ weixin_ 45847421/ article/ details/ 115189496.

[4] Liu Honghao, Ma Xibao, Zhang Yuanyuan, et al. Research on Tiff image Digitization File [J]. Automation Technology and Application, 2014,33(04):20-24. (in Chinese).

[5] vstion. (2013). TIFF Image File Format Explained. CNBlogs. https://www.cnblogs.com/gywei/p/3393816.html.

[6] YAN Tao, Yu Xi, Liu Yonghong, et al. Design and implementation of efficient memory pool based on C++ [J]. Journal of Chengdu Univer- sity (Natural Science Edition), 2017, 36 (03): 25 7-261. (in Chinese).

[7] Zhao Changyu, Wang Xilong. Embedded software memory management method research [J]. Science and technology and innovation, 2024, (16): 129-131. The DOI: 10.15913 / j.carol carroll nki kjycx. 2024.16.036.

[8] Huang Xiangping, Peng Mingtian, Yang Yongkai. Application of Electronic Technique, 2019,46(07):113-117+126. (in Chinese) DOI: 10.16157/ j. issn.0258-7998.191043.

[9] Yang Ningxue, Zhu Changqian, Nie Aili. Application Research of Computers,2004, (08):187-188. Memory mapped file and its application to fast access of large data files [J].