Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 4486494, 10 pages
https://doi.org/10.1155/2021/4486494

Research Article

SW-LZMA: Parallel Implementation of LZMA Based on SW26010 Many-Core Processor

Bingzheng Li, Jinchen Xu, and Zijing Liu

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

Correspondence should be addressed to Bingzheng Li; francisleeha@163.com

Received 12 August 2021; Revised 2 September 2021; Accepted 7 September 2021; Published 18 September 2021

Academic Editor: Thippa Reddy G

With the development of high-performance computing and big data applications, the scale of data transmitted, stored, and processed by high-performance computing cluster systems is increasing explosively. Efficient compression of large-scale data and reducing the space required for data storage and transmission is one of the keys to improving the performance of high-performance computing cluster systems. In this paper, we present SW-LZMA, a parallel design and optimization of LZMA based on the Sunway 26010 heterogeneous many-core processor. Combined with the characteristics of SW26010 processors, we analyse the storage space requirements, memory access characteristics, and hotspot functions of the LZMA algorithm and implement the thread-level parallelism of the LZMA algorithm based on Athread interface. Furthermore, we make a fine-grained layout of LDM address space to achieve DMA double buffer cyclic sliding window algorithm, which optimizes the performance of SW-LZMA. The experimental results show that compared with the serial baseline implementation of LZMA, the parallel LZMA algorithm obtains a maximum speedup ratio of 4.1 times using the Silesia corpus benchmark, while on the large-scale data set, speedup is 5.3 times.

1. Introduction

With the improvement of high-performance computer performance, its scale is expanding. The large-scale high-performance computing cluster system must maintain long-term and stable uninterrupted operation. The amount of data transmitted, stored, and processed is increasing, and the amount of system log data is also increasing explosively. At present and in the future, it can be predicted that the scale of social computing data and scientific computing data will continue to grow with the improvement of informatization, which brings new challenges to big data processing. Effective compression is necessary to reduce the space required for data storage, make maximum use of the limited communication bandwidth, and make the high-performance computing cluster system give full play to its efficiency. With the increasing amount of data, blockchain applications need a lot of storage space. A fast big data compression algorithm can improve the efficiency of blockchain applications [1].

In the actual application of the Internet of Things, there are obvious shortcomings, such as the limited energy and bandwidth of the sensor nodes, which brings huge challenges to the network data transmission of the Internet of Things devices. The compression algorithm is currently an important technology to reduce the amount of transmitted data. It can appropriately remove the redundancy, reduce the data storage space of the IoT, and improve the speed and success rate of data transmission of the IoT. From the point of view of the server, the rapid development of information technology, especially IoT, has brought about the explosive growth in the amount of data on the server due to the demand for big data processing. This also requires efficient compression algorithms to reduce the amount of data storage and processing of algorithms such as distributed big data processing and machine learning [2, 3]

Lossless compression algorithms have a wide variety of open-source implementations. In the Sunway TaihuLight supercomputer, the existing data compression algorithms include zlib Deflate, XZ, and LZ4. None of the compression algorithms is optimized in parallel, and only a single processor core is used for compression and decompression, while the processing performance does not have much room for improvement. In compression algorithms, there are many problems, such as the contradiction between compression rate and storage space and poor data locality. In order to achieve an effective performance improvement, deep algorithm reconstruction and optimization must be carried out for specific high-performance processors.

Many studies have used multicore processor architecture to parallelize compression algorithms. The parallelization of BWT (burrows Wheeler transform) compression algorithm appeared earlier. Pankratius et al. [4] first proposed a parallel implementation of BWT, which obtained a linear speedup ratio and was applied to Bzip software. Pigz is a parallel version of Gzip compression algorithm, which was proposed by Gristwood et al. [5] and has been widely used, but the compression rate of this parallel algorithm is low. Patel et al. [6] used GPU to parallelize the binary tree search process of the BWT lossless compression algorithm, and the acceleration effect was significant. Wu et al. [7] studied the compression algorithm based on CUDA (compute unified device architecture) and used the block parallel strategy to optimize the LZ77 compression algorithm on the GPU. Pankratius et al. [8] use MPI (message passing interface) programming to realize the distributed MPIBZIP compression algorithm, which is suitable for distributed memory computing. Wright [9] uses MPI and pthreads programming interfaces to implement the bzip2 parallel algorithm in the distributed memory structure and the shared memory structure, respectively. Although BWT-based compression algorithms are easy to parallelize, they are not as good as LZMA (Lempel Ziv-Markov chain algorithm) in terms of compression rate. In the process of multithreading parallelization of open-source compression software such as XZ and 7zip, only the character matching core function in the LZMA algorithm is parallelized. The acceleration effect is not ideal and is limited by the number of processors [10, 11] Leavline and Singh used FPGA to accelerate the LZMA algorithm [12, 13], which can obtain a higher speedup ratio, but the application cost is higher and does not have general applicability.

In the Sunway TaihuLight supercomputer system [14], the basic unit is a computing node composed of a SW26010 many-core processor, 32 GB of memory, and other control units. The processor architecture is shown in Figure 1. Four core groups (CGs) constitute a SW26010 processor, and there are 64 computing processing elements (CPEs) plus one management processing element (MPE), totally 260 computing units in SW26010. Among them, the CPE adopts a lightweight core design, and its instruction set function is very streamlined, does not support operations such as interrupts, and only runs in user mode. Each CPE contains 16 KB instruction L1 cache and 64 KB LDM (local directive memory, on-chip local data space) and supports 256-bit SIMD operations. The CPE can share memory with the MPE and use DMA (direct memory access) to exchange data between memory and LDM. In the CPE cluster, the CPEs in the same row or column can exchange data through register communication, the maximum amount of data transmitted each time is 256 bits, and the delay is low.

Figure 2 is a memory hierarchy diagram of CPE. The slave core can read data from memory in two ways: direct register access and register LDM access. Since there is no shared cache between CPEs and MPE, the delay of direct register access reaches nearly a hundred clock cycles. One of the ways to solve the problem is to copy data to LDM for memory access through DMA to improve memory access speed. This increases the difficulty of parallel program design and requires the programmers to set up DMA scheduling strategies reasonably, so as to achieve overlap of computing and communication as much as possible and to improve parallel efficiency. Data exchange between CPEs can be carried out by register communication. The parallel program on the SW26010 processor adopts the master-slave parallel programming model. The master thread runs on MPE, and the slave threads run on CPEs. The master thread mainly completes data input, memory copy, result output, and other operations, and the slave threads mainly perform computing tasks. According to the characteristics of the master-slave parallel programming model, Sunway TaihuLight supercomputer system provides the Athread accelerated thread library, which is divided into two parts: the MPE accelerated thread library and the CPE-accelerated thread library.

The main purpose of this paper is to design an LZMA parallel algorithm for Sunway TaihuLight supercomputer system and combine the characteristics of Sunway 26010 many-core processor to reconstruct and optimize the algorithm. We present SW-LZMA that can obtain a maximum speedup ratio of 4.1 times using the Silesia corpus benchmark while on the large-scale data set, speedup is 5.3 times.

2. Analysis of LZMA Algorithm Based on SW26010 Processor

In this section, we mainly analyse the characteristics of the LZMA algorithm that affect the performance of the algorithm such as space requirements, memory access methods, data locality, and hotspot functions. Combined with the analysis of the key technologies of SW26010 Processor, the algorithm can be reconstructed and optimized in a targeted manner.

2.1. LZMA Workflow. The LZMA compression algorithm was proposed by Pavlov in 1998 [15], and its core is based on the improvement of the LZ77 compression algorithm. LZMA uses a sliding window-based dynamic dictionary compression algorithm and interval coding algorithm, which has the advantages of high compression rate, small decompression space requirement, and fast speed. Figure 3 shows the LZMA workflow, including the sliding window algorithm based on LZ77 [16] and interval encoding [17, 18] (range encoding) two-stage compression.

The LZMA supports a dictionary space of 4 KB to hundreds of MBs, which increases the compression rate and also causes its search cache space to become very large. To reduce the time required to match the longest string and quickly search for matching characters, in the implementation of the LZMA algorithm, multiple possible longest matches are stored in the Hash table, and the data structure of the Hash linked list or binary search tree is used to search data. As

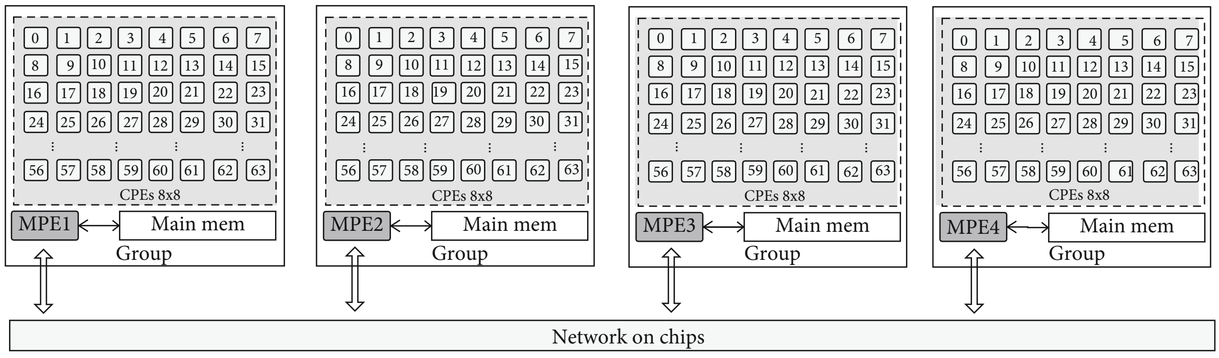

Figure 1: General architecture of SW26010 processor.

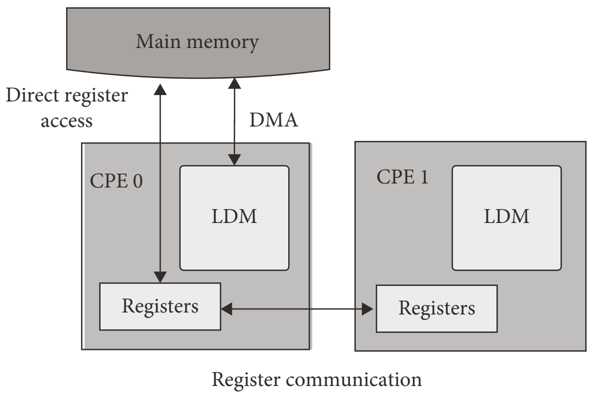

Figure 2: Computing processing element (CPE) memory hierarchy.

shown in Figure 4, in the Hash function, the hash value of the first two bytes of the search cache is used as the index of a hash array, and the hash array stores the starting position of the corresponding matching character group. The size of the hash array is a power of 2 that is half the size of the dictionary. The LZMA encoder sets up different levels of hash functions for 2, 3, and 4 adjacent bytes to achieve efficient positioning corresponding to different dictionary sizes.

2.2. Memory Space Demand. In the SW26010 processor, each CPE is equipped with a 64 KB LDM. In order to ensure that the CPR can obtain higher acceleration performance, it is necessary to copy the calculation data to the CPE's LDM space for memory access, which requires precise control of the use of the CPE's LDM variable memory space. Table 1 shows the usage of the local variables of the hotspot function of the LZMA algorithm, which mainly includes the local array size that takes up a large space, and the local scalar space takes up a small space and is negligible.

In the string-matching function based on the hash table, due to the large dictionary space, the hash table *hash_buf* reserves a larger hash space. This far exceeds the 64 KB LDM space of CPE and needs to be optimized to compress the use of local space. The range of the dictionary search can be reduced as much as possible within the allowable range of the compression loss, thereby reducing the size of the hash space of the hash table lookup function.

2.3. Memory Access Characteristics. In the LZMA algorithm, the data structure of the hash linked list is used to quickly find matching characters. Due to its relatively large search cache, its hash look-up table space has increased, with random access to memory in the range of 100 KBs to 10 MBs. At the same time, the LZ77 algorithm is based on sliding window streaming compression, because the uncoded data is continuously input, the coded data is discarded after reaching the upper limit of the search buffer space, and its data locality is poor.

Since it is impossible to prejudge the length and position of the repeated character string in the uncoded data, nor can it predict the distance of the matching character string, it is difficult to prefetch the data in the LZMA algorithm. During the compression process, the size of the dictionary gradually increases as the number of matching strings increases. Compressing the current data block depends on the dictionary obtained from the previous compression process. The LZMA algorithm has the characteristics of random access to memory and data dependence, which is a memory access-intensive algorithm. The key to its performance optimization is to combine the storage structure of the SW26010 processor to reconstruct the data structure and memory access of the algorithm to reduce memory access overhead and maximize the acceleration performance of CPEs.

2.4. Hotspot Functions. The main time-consuming functions of the LZMA compression algorithm are concentrated in the LZ77 string matching core function. The core function pattern matching process is shown in Algorithm 1. Among them, the time-consuming operations are mainly hash table lookup and character matching and hash table update.

In the hotspot function, the character matching process access to memory has a certain continuity, that is, starting from the current byte position, matching, and searching the same longest string in the cache. And each matched character is given by the input data. The position where the longest matching character may appear is stored in the hash table, and its look-up table access has certain randomness. In view of the characteristics of hotspot functions, there are mainly two optimization ideas. The first is to finely

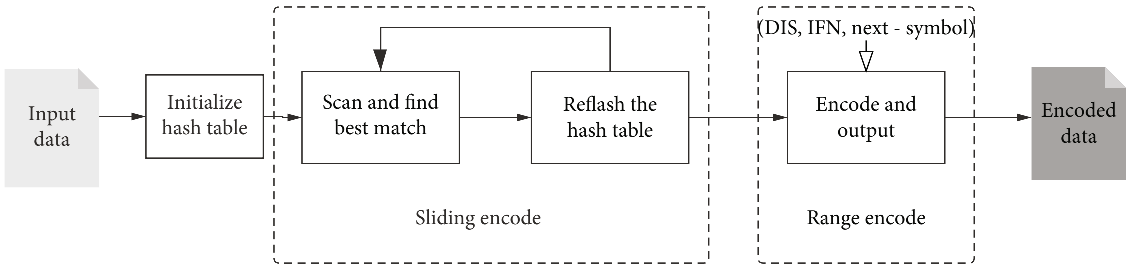

FIGURE 3: Flow diagrams of LZMA.

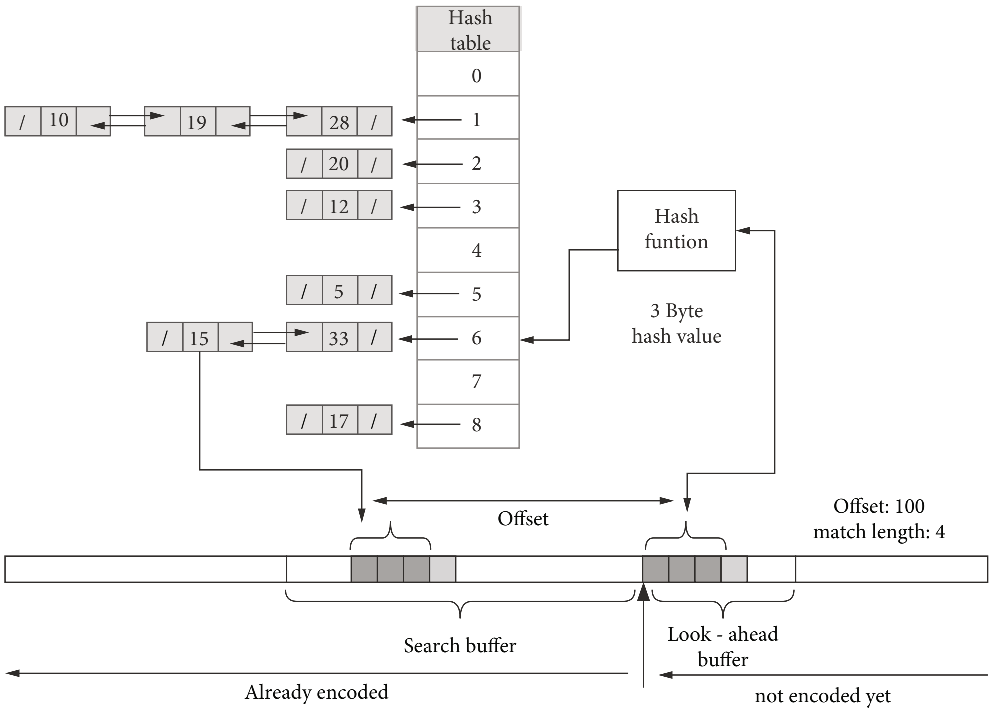

FIGURE 4: LZMA sliding window algorithm based on hash table.

TABLE 1: Local variable size of hotspot function.

Variable	Types	Local memory size	Scope (hot spot function)
hash_buf	Int	558 KB	Hc3Zip_MatchFinder_GetMatches Hc3Zip_MatchFinder_Skip
data_stream	Char	128 KB	GetOptimum
CRangeEnc_buf	Char	48 KB	LitEnc_Encode LenEnc_Encode
Match	Int	24 KB	ReadMatchDistances GetOptimum
isRepG0-G7	Int	4 KB∗8	LzmaEnc_CodeOneBlock GetOptimum
litProbs	Typedef struct	16 KB	LitEnc_Matched_GetPrice ReadMatchDistances FillAlignPrices
g_FastPos	Int	10 KB	FillDistancesPrices

divide the space usage and focus the memory-intensive operations on LDM to the greatest extent; the second is to properly reconstruct the algorithm and replace the three-byte hash function with a two-byte hash function to further compress the use of LDM space. Both of those ways can reduce the memory access delay. At the same time, attention

```
        Require:
        Hash_table: Hash table for fast search entry
        cur_pos: Pointer on first byte of the uncompressed data
        Process:
1.        Dictionary initialization
2.        While(there are still having uncompressed data in cur_pos)
3.            Calculate the hash_value of the first batch
4.            If(the hash_value can be found in hash_table){
5.                Update the value to the hash_table
6.                Encode the maximum string as (offset, len, cur) from current position
7.            }else{
8.                Encode the value as (0, 0, cur) according to the current position
9.            }
10.       End while
        Output: (offset, len, cur)
```

Algorithm 1: LZ77 compression algorithm based on sliding window.

should be paid to the LDM cache space size and the load balance of data transmission to achieve maximum hiding of computing communication.

3. Design and Implementation of SW-LZMA

3.1. Parallel Design of SW-LZMA. First, we designed the SW-LZMA multithreaded parallel algorithm on the SW26010 processor. The data to be compressed is evenly distributed to 64 CPEs cores. The CPE directly accesses the main memory to read the data to be compressed, and after adding header information to the compressed data, it directly outputs them to the main memory, and finally, the MPE writes the data blocks into the file in order. We adopted master-slave asynchronous parallelism and handed over the core computing tasks of the LZ77 compression algorithm and interval coding in the LZMA algorithm to the CPEs cluster. The MPE is only responsible for data partitioning and I/O operations. The steps of the thread parallel algorithm are as follows.

Step 1. Data segmentation. According to the number of CPEs, the data to be compressed are divided into several subblocks. We divide them according to the integer multiple of the memory page size. Since the amount of calculation in the compression algorithm is approximately proportional to the amount of input data, the parallel task load balance can be achieved only if the size of the divided data block is equal.

Step 2. Two-stage compression. Each data block is independently compressed by the CPE, including two-step compression. In the LZ77 algorithm, first, initialize the compression dictionary. As the sliding window advances, the data to be compressed continues to be input, and the dictionary size increases accordingly. Subsequently, the data structure compressed by the LZ77 algorithm is further compressed and output as the input data of interval coding.

Step 3. Data consolidation. After the CPEs complete the compression, the MPE is responsible for merging the compressed data. First, the MPE writes a 5 Byte header, and the content of which is compression parameter information such as dictionary size and maximum matching length. Then, each compressed block is output after adding 4 Byte header information *block_size* in the order of arrangement. The content of *block_size* is the size of the compressed data block.

3.2. Implementation of DMA Double-Buffers. Through the parallel method in Section 3.1, the core computing part of the serial LZMA compression algorithm can be transplanted to the CPEs cluster. However, when the CPE directly accesses the main memory, its memory access overhead will seriously reduce the performance of the parallel algorithm, and its acceleration effect is not enough to compensate for the performance loss caused by the memory access delay. In addition, in the serial version of the LZMA compression algorithm, the data to be compressed is stored in a dynamically allocated memory space, and the current compressed sliding window is determined by the address pointer. Due to the large scale of compressed data and the limited space of the CPE's LDM, even if it is divided into blocks according to thread tasks, its size is far greater than the 64 KB maximum capacity of LDM, and the data blocks to be compressed cannot be loaded all at once. Therefore, the algorithm needs to be reconstructed and optimized to compress the use of LDM space.

In order to improve the locality of data, we use the non-blocking DMA-based memory access double buffer technology based on the characteristics of the LZ77 sliding window algorithm and LDM space resources. As shown in Figure 5, the CPE does not directly access the main memory to read compressed data. Instead, the data in the current compression window and the data before and after it are transferred to the LDM buffer as a compression unit through the DMA method to achieve fast memory access. At the same time, the next compression unit has initiated DMA transfer to perform data prefetching. After the task of the current compression unit is completed, the compression calculation can be performed directly to achieve calculation and memory

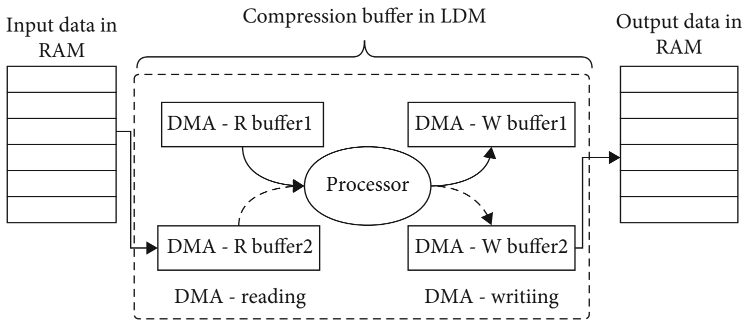

Figure 5: DMA double-buffers.

```
    Require:
    Hash_table: Hash Table for fast search entry
    cur_pos: Pointer on first byte of the uncompressed data
    Src: Pointer on input data (in main memory)
    Process:
1.      DMA_get(buffer_base, src, buffer_size)
2.      While(src data is not empty){
3.          If(buffer block compression is finished){
4.              DMA_iget(buffer_base, src, buffer_size)
5.              DMA_iput(dest, compressed _data, compressed_size)
6.          }else{
7.              LZ77_Compress(hash_Table, cur_pos, buffer_base)
8.              Range_Encode(compressed _data)
9.          }
10.     }end while
    Output:
    Dest: Compressed _data output (in main memory)
    Compressed _data: Output buffer (in LDM)
```

Algorithm 2: LZMA Athread on CPEs.

access overlap, which further reduces memory access overhead. At the same time, the output data is also buffered and copied to the memory through DMA. Algorithm 2 is an example of LZMA algorithm multithreaded parallel implementation using Athread interface.

3.3. LDM Space Layout Optimization. In the serial version of the LZMA algorithm, a pointer is used to directly point to the memory space of the data to be compressed, and a sliding window-based dictionary compression algorithm is implemented in the form of displacement. In the SW-LZMA algorithm, the compressed data needs to be copied to the LDM buffer area for memory access. In order to achieve DMA double buffering and make full use of the LDM space, we use manual methods for fine-grained management and allocation of the LDM address space and reconstruct the sliding window algorithm. We set up continuous double buffer space, and the pointer *buffer_base* points to the starting address of the address space, that is, the starting position of the first buffer. The pointer *buffer_middle* points to the middle position of the buffer space, that is, the starting position of the second buffer. The pointers *pos_start* and *pos_end* point to the start and end positions of the current sliding window, respectively.

At the beginning of the algorithm, as shown in Figure 6, the CPE initiates a blocking DMA request to read the data block to buffer 1, then calls the sliding window compression function, and initiates a nonblocking DMA request to read the next data block to buffer 2. When the sliding window pointer *pos_end* moves to the *buffer_middle* position, check that the nonblocking DMA request is completed, and then the compression can continue. Later, when the sliding window pointer *pos_start* moves to the *buffer_middle* position, a nonblocking DMA request is initiated to read the next data block to buffer 1. When the pointer *pos_start* and pointer *pos_end* move to the end position of the buffer, they move to the start position in a loop and continue to compress until all the data is compressed.

4. Evaluation

We mainly test and analyse the compression rate and compression time of the SW-LZMA algorithm. The benchmark performance is the compression ratio and compression time of the serial LZMA algorithm running on the main core. The timing method of the test is to use the Athread timing interface to count the number of CPU beats that the algorithm has run and calculate the operation time cost.

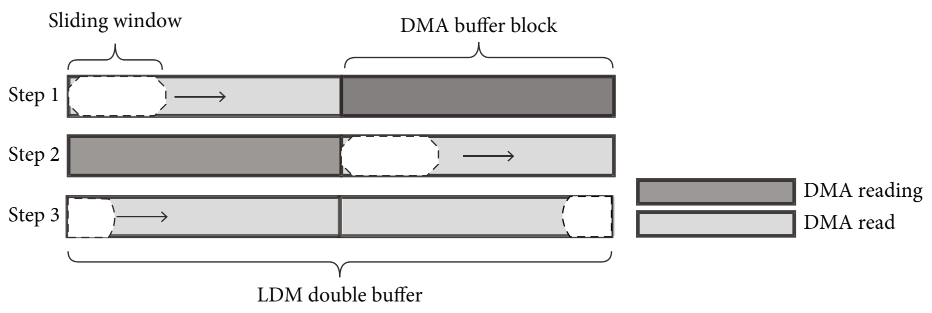

FIGURE 6: LDM address space partition of sliding window encoding based on DMA double buffer.

TABLE 2: Silesia corpus test contents.

Filename	Description	Type	Raw size (byte)
dickens	Collected works of Charles Dickens	English text	10192446
mozilla	Tarred executables of Mozilla 1.0 (Tru64 UNIX edition)	Exe	51220480
mr	Medical magnetic resonance image	Picture	9970564
nci	Chemical database of structures	Database	33553445
ooffice	A dll from Open Office.org 1.01	Exe	6152192
osdb	Sample database in MySQL format from open source database benchmark	Database	10085684
reymont	Text of the book by Władysław Reymont	Polish pdf	6627202
samba	Tarred source code of Samba 2-2.3	Src	21606400
sao	The SAO star catalog	Bin data	7251944
webster	The 1913 Webster Unabridged Dictionary	HTML	41458703
xml	Collected XML files	HTML	5345280
x-ray	X-ray medical picture	Hospital image	8474240

4.1. Benchmark Corpus and Experiment Platform. The Silesia compressed test corpus was proposed by Sebastian Deorowicz in 2003 [15], providing a file data set covering typical data types currently in use. The files' sizes are between 6 MB and 51 MB. The corpus is proposed to solve the problem of the lack of large files and single file types in the traditional Canterbury corpus. Table 2 shows the test example of the benchmark test set.

The experimental platform is the Sunway Taihulight supercomputing system, and its parameters are shown in Table 3 [18]. The compression algorithm benchmark test set used in the experiment is the Silesia corpus benchmark test set. At the same time, in order to test the compression performance of a large amount of data, we copied and packaged the Silesia corpus test set files to form GB-level data for compression testing.

TABLE 3: Experiment environment.

Item	Parameters
MPE	1.45GHz, 32 KB L1 D-cache, 256 KB L2 cache
CPE	1.45GHz, 64 KB LDM
CG	1 MPE + 64 CPE
Single node	1 CPU (4 CGs) +4∗8 GB DDR3 memory

4.2. Performance Evaluation. In order to test the acceleration effect of the SW-LZMA parallel algorithm on SW26010 processor, the serial version of the MPE LZMA algorithm was selected as a benchmark to compare the performance of different optimization schemes. The compression speed and compression rate of SW-LZMA in the Silesia corpus benchmark test are shown in Figure 7. Due to the large memory access bottleneck, the 64-thread parallel version that reads data directly from the main memory only obtains an average speedup of 2 times and even spends more time than the serial compression in some cases. In contrary, the optimized version of communication overlaps using DMA double buffering obtained an average speedup ratio of 3.7 times and a maximum speedup ratio of 4.1 times, indicating that the parallel performance of using DMA double buffering has been greatly improved. In terms of compression ratio, the compression ratios of parallel and serial versions are basically the same.

Further analysis, we discussed the impact of the choice of single buffer size *buffer_size* on the number of message transfers and compression rate in the DMA double buffer design. As shown in Figure 8, when the *buffer_size* is less than 20 KB, due to the small amount of data copied by a single DMA, calculation and communication cannot be fully overlapped. At the same time, the number of DMA increases, the corresponding DMA overhead increases, and the compression speed decreases slightly. When the *buffer_size* is greater than 25 KB, the compression speed does not change much with the buffer size. Theoretically, the setting of the buffer should enable the DMA communication delay

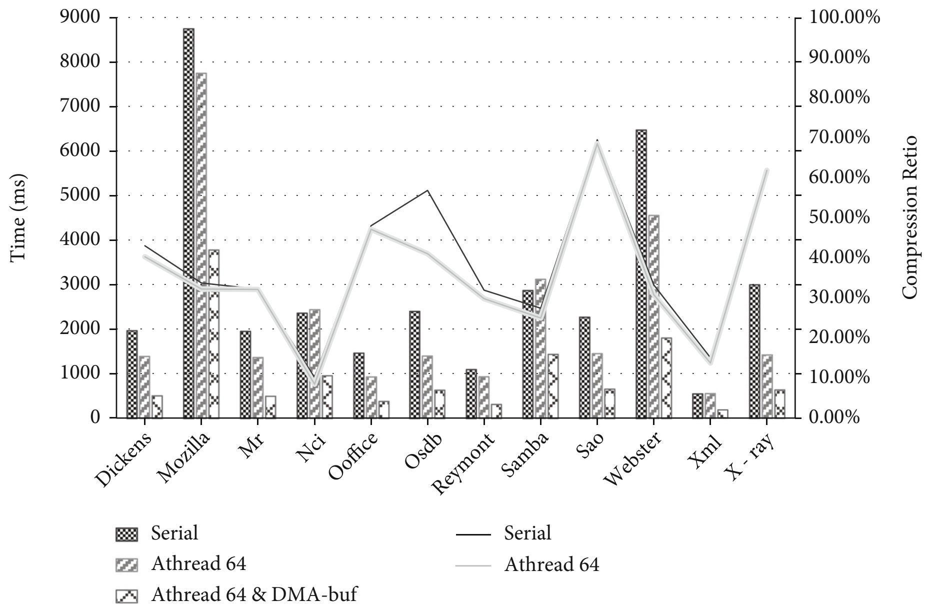

FIGURE 7: Test performance results of SW-LZMA.

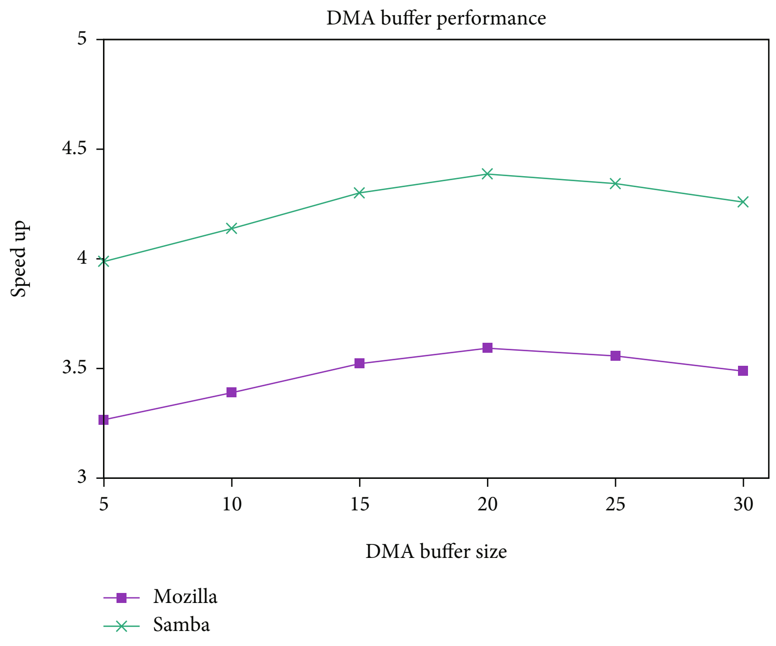

FIGURE 8: DMA buffer size influence on SW-LZMA performance.

and calculation to achieve load balance, but because the LDM space is limited and needs to be reserved for other local variables, the buffer cannot be expanded indefinitely.

In Section 2.2, we mainly discuss the memory space demand of the LZMA algorithm and try to satisfy it within the 64 KB LDM space of each CPE. We designed DMA double-buffers, and LDM address space partition of sliding window in Section 3 to make full use of LDM space. According to the experimental results, SW-LZMA parallel algorithm has reached the maximum utilization of the local memory space of CPEs and cannot be expanded to reach the maximum bandwidth utilization and frequency of the SW26010 processor mainly due to the LDM space limitation and memory access latency. Therefore, we take the buffer size with the best performance currently as the optimal parameter to maximize the overlap gain of computing communication optimization.

Most of the compressed test corpus data are small in scale, and no GB-level test cases are provided. We use Linux tar tool to package multiple copies of Silesia corpus to generate several large file test sets. We test the compression performance of the SW-LZMA parallel algorithm on big data based on the large file test sets. Table 4 shows the

Table 4: Large-scale data compression test results.

File size (MB)	Serial		Athread		Speedup
	Time (s)	Compression ratio	Time (s)	Compression ratio	
202.1387	34.01	32.96%	15.88	30.24%	2.142
404.2676	66.93	32.96%	27.74	30.22%	2.412
808.5254	268.55	32.96%	51.48	30.21%	5.216
1212.7832	409.78	33.23%	77.11	30.21%	5.314

Table 5: Performance test comparison between Intel x86 and Sunway 26010.

Hardware architecture	Intel E5-1650 v2 3.5 GHz	Sunway 26010	
Software environment	Linux 3.13, gcc 4.8.4 at -O2 level optimization, single thread	sw5CC -O2 level optimization, athread multithreads	
Compression corpus	Canterbury corpus	Silesia corpus	
Thread number	Single thread	Single thread	Athread
Compression ratio	28.96%	34.12%	32.33%
Compression speed (MB/s)	4.40	3.01	15.71
Speedup	1	0.68	3.57

performance comparison between the serial LZMA algorithm and the SW-LZMA parallel algorithm in large-scale data compression. It can be seen from the data table that when the data volume exceeds 500 MB, the parallel compression rate has a significant increase, with a maximum speedup of 5.3 times. The parallel compression algorithm has better adaptability in the compression of large-scale data.

4.3. Related Work. Alakuijala et al. used the Canterbury compression test corpus to perform performance testing and comparative analysis on compression algorithms such as Bzip2 and LZMA [19]. Their experiment platform is Intel E5-1650 v2 3.5 GHz processor. We compare it with the test results of the SW-LZMA parallel algorithm. Since the compressed data sets are different, we only compare the average compression ratio and average speedup ratio. The results are shown in Table 5. Because the CPU frequency gap is obvious, the performance of the LZMA serial algorithm on SW26010 is inferior to that on Intel CPU, and the parallel version of the LZMA has a more obvious acceleration effect than that of the Intel CPU, indicating that the SW-LZMA has better performance advantages.

5. Conclusions

The main work of this paper is to transplant the LZMA compression algorithm to the Sunway Taihulight supercomputer system and to reconstruct and optimize the parallel algorithm according to the characteristics of the Sunwei many-core processor. We use the Athread interface to parallelize the LZMA algorithm with multithreads and blocks and design a DMA-based double buffer mode to achieve overlap of computing communication. In further optimization, we perform fine-grained management and layout optimization on the LDM address space, set the buffer size reasonably, and obtain the best computing communication overlap effect. The test results show that in the Silesia Corpus benchmark test set, the SW-LZMA algorithm achieves a maximum speedup of 4.1 times. In the large file compression test, the SW-LZMA parallel algorithm achieved a maximum speedup of 5.1 times. Compared with mainstream CPU serial algorithms such as x86 CPU, the SW-LZMA algorithm has an obvious acceleration effect on SW26010 many-core processors, greatly reducing algorithm execution time, and has better performance. The SW-LZMA parallel algorithm not only can provide high-speed compression algorithms for applications in the field of high-performance computing but also is well known about its feasibility for more big data applications such as smart grid [20] and cloud computing [21].

In the future, there will be two research directions to further improve the performance of the LZMA algorithm: one is to upgrade the LZMA algorithm to further reduce the use of local space without affecting the compression rate; the other is to design more efficient parallel LZMA algorithm based on the new high-performance computing processors.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (Grant no. 2018YF0804003).

References

[1] N. Deepa, Q. V. Pham, D. C. Nguyen et al., “A survey on blockchain for big data: approaches, opportunities, and future directions,” 2020, https://arxiv.org/abs/2009.00858.

[2] G. T. Reddy, M. P. K. Reddy, K. Lakshmanna et al., “Analysis of dimensionality reduction techniques on big data,” *IEEE Access*, vol. 8, pp. 54776–54788, 2020.

[3] S. Rajadurai, M. Alazab, N. Kumar, and T. R. Gadekallu, “Latency evaluation of SDFGs on heterogeneous processors using timed automata,” *IEEE Access*, vol. 8, pp. 140171–140180, 2020.

[4] V. Pankratius, A. Jannesari, and W. F. Tichy, “Parallelizing Bzip 2: a case study in multicore software engineering,” *IEEE Software*, vol. 26, no. 6, pp. 70–77, 2009.

[5] T. Gristwood, P. C. Fineran, L. Everson, and G. P. Salmond, “PigZ, a TetR/AcrR family repressor, modulates secondary metabolism via the expression of a putative four-component resistance-nodulation-cell-division efflux pump, ZrpADBC, in Serratia sp. ATCC 39006,” *Molecular Microbiology*, vol. 69, no. 2, pp. 418–435, 2008.

[6] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, “Parallel lossless data compression on the GPU,” in *2012 Innovative Parallel Computing (InPar)*, San Jose, CA, USA, 2012.

[7] L. Wu, M. Storus, and D. Cross, *CUDA WUDA SHUDA: CUDA Compression Projects*, Stanford University, 2009.

[8] V. Pankratius, A. Jannesari, and W. F. Tichy, “Parallelizing Bzip2: a case study in multicore software engineering,” *IEEE Software*, vol. 26, no. 6, pp. 70–77, 2009.

[9] C. Wright, “Hybrid programming fun: making bzip2 parallel with MPICH2 & Pthreads on the Cray XD1,” in *Proceedings of the 48th Cray User Group meeting. CUG’06*, pp. 78–84, Lugano, Switzerland, 2006.

[10] S. D. Agostino, “Lempel–Ziv data compression on parallel and distributed systems,” *Algorithms*, vol. 4, no. 3, pp. 183–199, 2011.

[11] X. Wang, L. Gan, J. Xu et al., “PLZMA: a parallel data compression method for cloud computing,” in *Algorithms and architectures for parallel processing: 18th international conference, ICA3PP 2018*, pp. 504–518, Guangzhou, China, 2018.

[12] E. J. Leavline and D. Singh, “Hardware implementation of LZMA data compression algorithm,” *International Journal of Applied Information Systems (IJAIS)*, vol. 5, no. 4, pp. 51–56, 2013.

[13] B. Li, L. Zhang, Z. Shang, and Q. Dong, “Implementation of LZMA compression algorithm on FPGA,” *Electronics Letters*, vol. 50, no. 21, pp. 1522–1524, 2014.

[14] H. Fu, J. Liao, J. Yang et al., “The Sunway TaihuLight supercomputer: system and applications,” *Science China Information Sciences*, vol. 59, no. 7, 2016.

[15] D. Salomon, *Data Compression: The Complete Reference*, Springer-Verlag New York, Inc., 2000.

[16] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[17] G. Martin, “Range encoding: an algorithm for removing redundancy from a digitised message,” in *Proceedings of the Conference on Video and Data Recording*, pp. 24–27, Southampton, 1979.

[18] S. Deorowicz, *Universal Lossless Data Compression Algorithms*, Silesian University of Technology, 2003.

[19] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne, *Comparison of Brotli, Deflate, Zopfli, Lzma, Lzham and Bzip2 Compression Algorithms*, Google Inc, 2015.

[20] M. Alazab, S. Khan, S. S. R. Krishnan, Q. V. Pham, M. P. K. Reddy, and T. R. Gadekallu, “A multidirectional LSTM model for predicting the stability of a smart grid,” *IEEE Access*, vol. 8, pp. 85454–85463, 2020.

[21] S. Senthilkumar, N. Kryvinska, S. Bhattacharya, and G. Reddy Bojja, *SCB-HC-ECC Based Privacy Safeguard Protocol For Secure Cloud Storage Of Smart Card Based Health Care System*, Frontiers in Public Health, 2021.