

# Performance modeling of hyper-scale custom machine for the principal steps in block Wiedemann algorithm

Tong Zhou<sup>1</sup> · Jingfei Jiang<sup>1</sup>

Published online: 3 June 2016  
© Springer Science+Business Media New York 2016

**Abstract** Solving large-scale sparse linear systems over GF(2) plays a key role in fluid mechanics, simulation and design of materials, petroleum seismic data processing, numerical weather prediction, computational electromagnetics, and numerical simulation of unclear explosions. Therefore, developing algorithms for this issue is a significant research topic. In this paper, we proposed a hyper-scale custom supercomputer architecture that matches specific data features to process the key procedure of block Wiedemann algorithm and its parallel algorithm on the custom machine. To increase the computation, communication, and storage performance, four optimization strategies are proposed. This paper builds a performance model to evaluate the execution performance and power consumption for our custom machine. The model shows that the optimization strategies result in a considerable speedup, even three times faster than the fastest supercomputer, TH2, while consuming less power.

**Keywords** Architecture · SpMV · Wiedemann algorithm

## 1 Introduction

The large-scale sparse linear systems solver is one of the most challenging questions in scientific and engineering computing, such as fluid mechanics, simulation, and design of materials, petroleum seismic data processing, numerical weather prediction and numerical simulation of unclear explosions. Therefore, it became a favorable application for building large-scale supercomputer. The block Wiedemann (BW) algorithm [8] is the most effective method for solving large-scale sparse linear systems over GF(p)

---

✉ Jingfei Jiang  
jingfeijiang@126.com

<sup>1</sup> National University of Defense Technology, Changsha, China

[2]. The most time-consuming part in this algorithm is the iteration computation of a large sequence of large-scale matrices, which requires considerable computation capability and storage resources. The general architecture of modern supercomputers has high flexibility for many fields, but it would not match the execution and data characteristics for the key steps of BW. When the matrix size significantly increases, common supercomputer may not afford the time cost.

Thomé et al. [22] work on the BW algorithm itself and describe how the half-gcd algorithm can be adapted to speed up BW algorithm. Schmidt et al. [19] derive an efficient CUDA implementation of this operation. In addition, their GPU cluster implementation of BW algorithm results in a significant speedup over traditional CPU cluster implementations. Güneysu et al. [12] build a massively parallel cluster system based on low-cost FPGAs for cryptanalytical operations, such as BW, and describe a novel architecture targeted for a more versatile and reliable system. Several studies have analyzed the speeding up of sparse matrix–vector multiplication (SpMV), which is the kernel operation of sparse linear systems solvers. Catalyurek and Aykanat [6] built a hypergraph model for decomposing sparse matrices on a multiprocessor platform to minimize the communication data while balancing the computing tasks on each processor. Baskaran and Bordawekar [4], Dou et al. [11] and Dave et al. [9] speeded up the operation or matrix multiplication on specific computing architectures such as GPU and FPGA. Buluç and Gilbert et al. [5], Langr and Tvrdik [15] investigated the compact format for sparse matrices to decrease the data volume involving memory-access and obtained a significant speedup.

Some scholars analyzed the data feature for acceleration, such as [14], who quantified the effect of matrix structure on SpMV performance. Sedaghati et al. [20] presented some insights into the correlation between matrix features and the best choice for sparse matrix representation. Furthermore, several studies on supercomputer platforms, for example, that of Anzt et al. [1], unveiled some energy efficiency and performance frontiers for sparse computations on GPU-based supercomputers. Pichel et al. [17] estimated the influence of data and thread allocation in the SpMV performance on the FinisTerra supercomputer, an SMP-NUMA system with more than 2500 processors. Some scholars contribute to constructing an energy-efficient platform for high-performance computation(HPC). Mont Blanc project tries to build from energy-efficient solutions used in embedded and mobile device [18,21]. Kapre and Moorthy [13], Dordopulo et al. [10] and Awad [3] study on FPGA-based platform for energy-saving or supercomputing.

However, all of these studies were either oriented toward general sparse data features or on general purpose platforms, such as clusters and supercomputers. However, the specific sparse matrix features with customized architecture show a high potential for performance improvement. In this paper, we proposed a hyper-scale custom super-computer architecture matching specific data features to process the key procedure of BW. The major contributions in this paper are as follows:

- We have proposed four optimization strategies according to the algorithm and data characteristics, which show high acceleration effects in terms of communication, computation, and memory access.

- Based on these optimization strategies, we propose the parallel algorithm of the principal steps and a custom machine in an economical and efficient manner.
- The performance model of our custom machine was built according to the algorithm and architecture parameters. The computation complexity was comprehensively analyzed based on the model. The evaluated results indicate that our custom machine works well, even better than the fastest computer in the world on this specific problem.

The remainder of this paper is organized into several sections: Section 2 describes the execution flow and characteristics of principal steps in BW. Section 3 proposes optimization strategies according to the algorithm features. Section 4 proposes the parallel algorithm of MSC and a custom machine to support its parallel execution. Section 5 builds the performance model of MSC and evaluates the performance the custom machine can generate. Finally, Sect. 6 presents the conclusions.

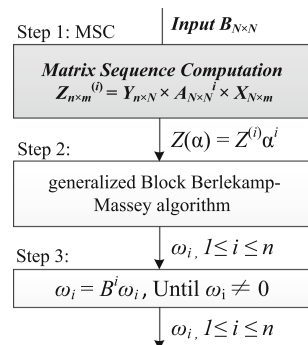
## 2 Execution of block Wiedemann algorithms and characteristics of MSC

### 2.1 Execution framework of block Wiedemann algorithms

The BW algorithm is a blocked iterative algorithm for linear equation  $B\omega = 0$ , which can obtain a solution for the linear equation with high probability. Its procedure includes three steps, as shown in Fig. 1: A sequence of matrices  $Z_{n \times m}^{(i)}$  is calculated in step 1 (MSC). A polynomial  $Z(\alpha)$  is built as the input of step 2 using  $Z^{(i)}$ . The generalized block Berlekamp–Massey algorithm is used in step 2, which generates sequence vector  $\omega_i$ . Then, the sequence of  $\omega_i$  is iteratively multiplied with matrix  $B$  until a zero vector is obtained and  $\omega_i$  is the final result in a high probability. In BW, the MSC step is the most time-consuming stage that is worthy of acceleration. Algorithm 1 shows the serial implementation for MSC. Input  $B$  is a sparse matrix, and  $V$  and  $W$  are randomly generated matrices.

Referring to Algorithm 1, iterative sparse matrix–matrix multiplication is the major operations, which can be regarded as iterative sparse matrix–vector multiplications when  $X^{(i)}$  and  $Y$  are treated as vectors of block width  $m$  or  $n$ .

**Fig. 1** The execution flow of BW algorithm



**Algorithm 1:** Serial MSC algorithm of BW

**Input:**  $B_{N \times N}$ ;  
 Generate matrix  $V_{m \times N}$  and  $W_{N \times n}$  randomly;  
 Define  $X_{N \times m}^{(0)} = V_{m \times N}^T, Y_{n \times N} = W_{N \times n}$   
**Output:**  $Z_{n \times m}^{(i)} = Y_{n \times N} \times B_{N \times N}^i \times X_{N \times m}^{(i)}; 0 \leq i \leq \frac{N}{m} + \frac{N}{n} + O(1)$   
 1 **for**  $i = 0; i \leq \frac{N}{m} + \frac{N}{n} + O(1); i++$  **do**  
 2      $X_{N \times m}^{(i+1)} = B_{N \times N} \times X_{N \times m}^{(i)}$       $Z_{n \times m}^{(i)} = Y_{n \times N} \times X_{N \times m}^{(i+1)}$   
 3 **end**

**Table 1** The data characteristics of MSC

Matrix	Scale	Characteristic	Memory capacity
$B$	$N \times N$	Sparse matrix, with low density $d$ and small-value elements like 1, $-1$ and other small numbers	Compressed format with $s_b$ bytes indicates an element: $N \times d \times s_b$
$X$	$N \times m$	Dense matrix, elements are big numbers	$s_x$ bytes indicates an element: $N \times m \times s_x$
$Y$	$n \times N$	Sparse matrix, all elements' values are 1	$s_y$ bytes indicates an element: $n \times N \times s_y$
$Z$	$n \times m$	Dense matrix, elements are big numbers	$s_z$ bytes indicates an element: $n \times m \times s_z$

**2.2 The data and execution characteristics of MSC**

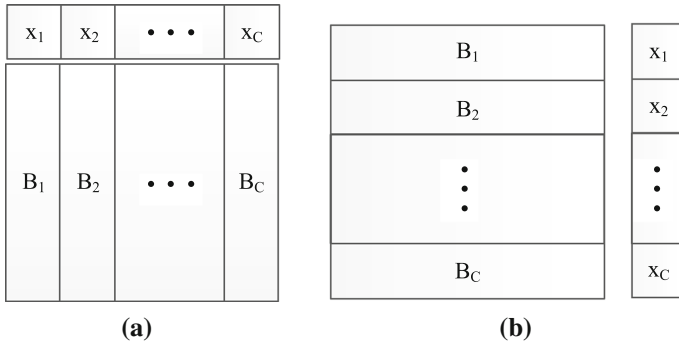
Table 1 shows the data features involved in MSC.  $B$  is an input sparse matrix with a size of  $N \times N$ , whereas  $X$  and  $Y$  are randomly generated matrices with sizes of  $N \times m$  and  $n \times N$ , respectively. MSC yields a sequence of matrices  $Z^{(1)}, Z^{(2)}, \dots, Z^{(i)}, \dots$ . Matrix  $B$ , with large sparsity  $d$ , can be stored in a compact manner using a CSR format. Its nonzeros are uniformly distributed, and most of them are 1 or  $-1$ , and the given  $s_b$  bytes indicate an element in  $B$ .

Matrix  $X$  is a dense matrix, given that each element takes  $s_x$  bytes.

Matrix  $Y$  is a sparse matrix and its nonzeros are 1. Although the given  $s_y$  bytes indicate one element,  $n \times N \times s_y$  bytes are required to store  $Y$ .  $Z$  is the result matrix, also given  $s_z$  bytes to indicate for each element; thus,  $n \times m \times s_z$  bytes are required in total.

**3 Optimization strategies**

As previously introduced, the key operation in MSC is SpMV. In this section, we introduce four optimization strategies to accelerate SpMV according to the execution and data characteristics of MSC. Assuming that  $x^{(i)}$  is a vector (a column) in matrix  $X$ ,  $X_i$  is the  $i$ th vector block of  $x$  after partitioning,  $B_{ij}$  is the matrix block in  $B_i$ , that is, the columns from  $(j - 1) \times \frac{N}{C} + 1$  to  $j \times \frac{N}{C}$  in  $B_i$  where  $C$  is the number of processors. In this section, we primarily consider the iterative multiplication of  $x^{(iter+1)} = Bx^{(iter)}$ .



**Fig. 2** Schemes to project matrix and vector into different processors

### 3.1 Matrix partition strategy

Vastenhouw and Bisseling [23] note that the natural parallel algorithm for sparse matrix–vector multiplication with an arbitrary distribution of the matrix and the vectors consists of the following four phases:

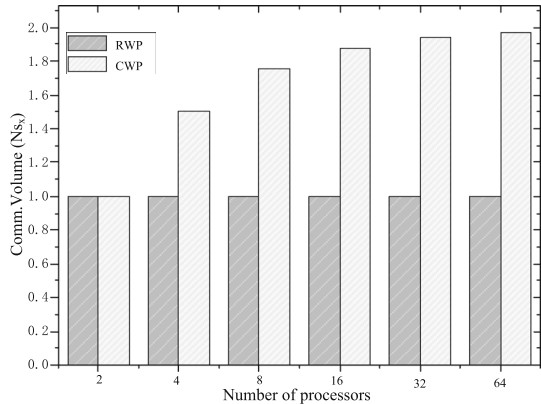
1. Each processor sends its components  $x_j^{(iter)}$  to those processors that possess a nonzero  $b_{ij}$  in column  $j$ .
2. Each processor computes the products  $b_{ij}x_j^{(iter)}$  for its nonzeros  $b_{ij}$  and adds the results for the same row index  $i$ . This operation yields a set of contributions  $x_{is}^{(iter+1)}$ , where  $s$  is the processor identifier,  $1 < s \leq C$ .
3. Each processor sends its nonzero contributions  $x_{is}^{(iter+1)}$  to the processor that possesses  $x_i^{(iter+1)}$ .
4. Each processor adds the contributions received for its components  $x_i^{(iter+1)}$ , giving  $x_i^{(iter+1)} = \sum_{t=0}^{C-1} x_{it}^{(iter+1)}$ .

One-dimensional (1D) and two-dimensional (2D) partition methods may be used. Some scholars proposed more complex and precise means of performing the partition and distribution work, such as Çatalyürek and Aykanat, who presented a 2D hypergraph model. However, random uniformly distributed matrices are expected to gain little for a 2D approach.

For uniformly distributed sparse matrix–vector multiplication, two 1D schemes can be used to map the matrix into different processors, namely row distribution and column distribution. As depicted in Fig. 2, scheme (a), column-wise partition (CWP) partitions matrix  $B$  into  $C$  blocks column-wise, and in scheme (b), row-wise partition (RWP) is performed row-wise.

The advantage of CWP is that it only needs point-to-point communication; however, its communication volume is large. By contrast, the advantage of RWP is that it removes phases 3 and 4, the price to be paid is to distribute the elements of the vector over a large number of processors, and the number of destination processors of  $x_j$  can reach  $C - 1$ . Moreover, based on the sparsity and size of the matrix, we demonstrate that the same vector component  $x_j$  is required by all processors with high

**Fig. 3** Comparison of CWP and RWP on communication data volume with various processors. The trend shows that the CWP’s communication volume will grow into two times of RWP’s with the increasing of processors



probability. Thus, broadcast vector component  $x_j$  will be efficient and minimize the true communication volume. We will quantify the total communication data volume per processor for these two schemes.

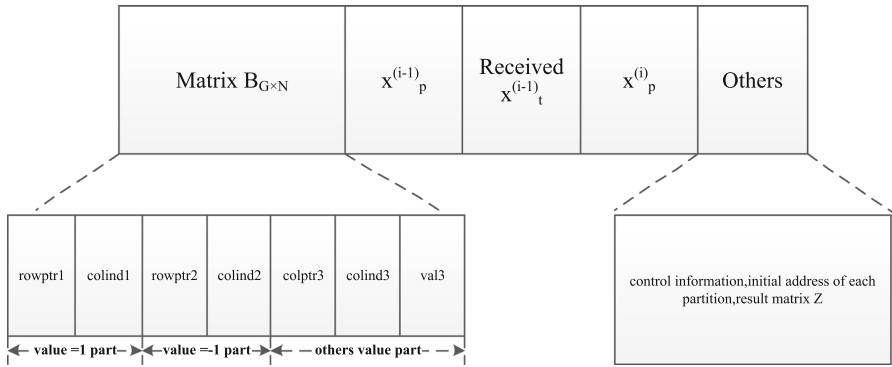
The  $i$ th processor holds  $X_i$  and  $B_i$ ; however, for CWP, the  $i$ th processor computes  $B_i \times X_i$  per iteration and transmits its results to its corresponding processor, such as  $B_{ip} \times X_i$  to processor  $p$  as described in phase 3. It receives its intermediate results transmitted from other processors and merges them as the final outcome of this iteration as described in phase 4. Different from CWP, the  $i$ th processor for RWP broadcasts  $X_i^{iter}$  and receives the  $x^{iter}$  part that it does not hold from all of the other processors when computing  $X_i^{(iter+1)} = B_i x^{iter}$ . The data involving memory access are in the same size because the computation task is uniformly distributed in both schemes.

The transmitting data volume for each processor per iteration is  $\frac{C-1}{C} \times N \times s_x$  and represents the same volume of receiving data when CWP is exploited, where  $C$  is the number of processors.

In contrast to CWP, RWP broadcasts its vector block at one time; thus, the transmitting data volume is  $\frac{1}{C} \times N \times s_x$ . The receiving data volume of RWP is the same as CWP, that is,  $\frac{C-1}{C} \times N \times s_x$ . Hence, the total communication data volumes for one processor per iteration are  $2 \times \frac{C-1}{C} \times N \times s_x$  and  $N \times s_x$  for CWP and RWP, respectively. In Fig. 3, the dark grey bar shows the communication volume of RWP with various processors, whereas the light grey bar shows the communication data volume of CWP. Referring to Fig. 3, we note that the communication volume of CWP will increase to twice that of  $Ns_x$  with the increasing number of processors, whereas RWP will remain at  $Ns_x$ . Thus, RWP is better than CWP in terms of communication data volume.

### 3.2 Memory partition strategy

The MSC algorithm is principally repeated in executing SpMV; however, the involved data types are countable. Based on this observation, we exploit the strategy of partition main memory to store different data, a new storing style that is derived from the



**Fig. 4** Memory partition of one processor. Matrix block  $B$  is divided into three parts: elements’ value is 1, is  $-1$  and others. Next to it are the vector block result of last iteration, the vector block result of last iteration received from others, the vector block result of intermediate of current iteration and other control information

partition cache scheme. This strategy matches the MSC algorithm (small data types) and maps the index of dense vector  $x$  element into the memory address, which prevents the storage of the elements’ indexes and yields  $s_x = 128$  bytes.

Figure 4 shows the method of memory partition for the  $i$ th processor. Matrix block  $B_i$  is stored into three parts, as will be mentioned in the next optimization strategy. Next to it are the vector block results of the last iteration  $X_i^{iter}$ , the vector block received from the other  $X_*^{iter}$ , the vector block of the current iteration  $X_i^{iter+1}$  and other control information, as well as the result matrix  $Z$ . Notably,  $Z$  is the result and will be only stored in the main memory temporarily before being stored into the next level of the memory system.

### 3.3 Separated value compression

As previously mentioned, most of the elements of matrix  $B$  are 1 or  $-1$ , and this aspect poses the potential for acceleration. We exploit this aspect by dividing  $B$  into three parts for storage:  $B_1$  for elements whose value is 1,  $B_2$  for  $-1$ , and  $B_3$  for other elements. The benefits are derived from the decrease in the demanding memory capacity and the reduction of the computation amount.

The first benefit is the minimizing of  $s_b$ . If we store  $B$  without dividing and in the CSR compression format, each nonzero at least requires two words: one for its value and another for its column index when ignoring the row pointer. Thus,  $s_b = 8$  bytes. However, when we store them apart and combined them with the partition of the main memory, the value item for 95 % of nonzeros can be ignored, and  $s_b$  decreases into nearly 4 bytes, thus gaining a half reduction. Storing them apart doubles the quantity of row pointers, but this quantity is trivial compared with what we have obtained. The reduction of  $s_b$  decreases not only the demanding memory capacity but also the volume of memory access data.

The other benefit is the reduction of the computation amount. The original multiplication accumulation operation will degenerate into simple addition for 1 or subtraction for  $-1$  because of the special position of 1 and  $-1$  in multiplication. For nearly 95 % of nonzeros of matrix  $B$  being 1 or  $-1$ , the decrease in the multiplication operation amount is 95 %, which is considerable.

These two benefits are shown in Fig. 5a, b, respectively. For the demanding memory capacity, we have compared the most popular compress format CSR and COO and our storage method demonstrates a considerable decrease in storage size. Furthermore, the computation amount significantly decreases in our method.

### 3.4 Vector communication strategy

Based on the 1D row distribution scheme and given that  $B_p(:, j)$  is the  $j$ th column of matrix  $B$ 's block in the  $p$ th processor and that at least one nonzero exists in the column, then the component  $x_j$  is required by the  $p$ th processor. Vice versa, when all of the elements in the  $j$ th column of matrix block  $B_p(:, j)$  are zero (all-zero-column), the component  $x_j$  is unnecessary and its probability is

$$p = (1 - d)^G, \quad (1)$$

where  $d$  is the sparsity of matrix  $B$  and  $G = N/C$  is the number of rows of  $B$  in one processor. Thus, the expectation columns whose elements are zeros are derived as follows.

$$E(p) = N \times p \quad (2)$$

The ratio of the unnecessary elements to all of the elements is  $w = E(p)/N = p$ , which is equal to the probability itself. When the problem size varies, the relationship between the probability of the all-zero-column and its expectation value with various processors is presented in Table 2.

Table 2 shows that the probability of the all-zero-column depends on the number of nonzeros per row, that is,  $N \times d$ . When  $N \times d$  is larger than  $10^3$ , even the amount of processors increases to 512 and the probability of the all-zero-column is no more than 0.14. Considering the matrices involved in this study,  $N \times d$  is larger than  $10^3$ , and nearly all of the elements in  $x$  are indispensable for every processor. Based on this observation, we exploit the broadcast method to transfer vector block  $X_i$ , which decreases both the communication volume and the complexity of interconnection structure among the processors as will be introduced below.

## 4 Parallel architecture for MSC

Having improved the performance of SpMV in communication, computation, and memory accessing for MSC, we propose a parallel algorithm and architecture of our custom machine to support its parallel execution in this section.



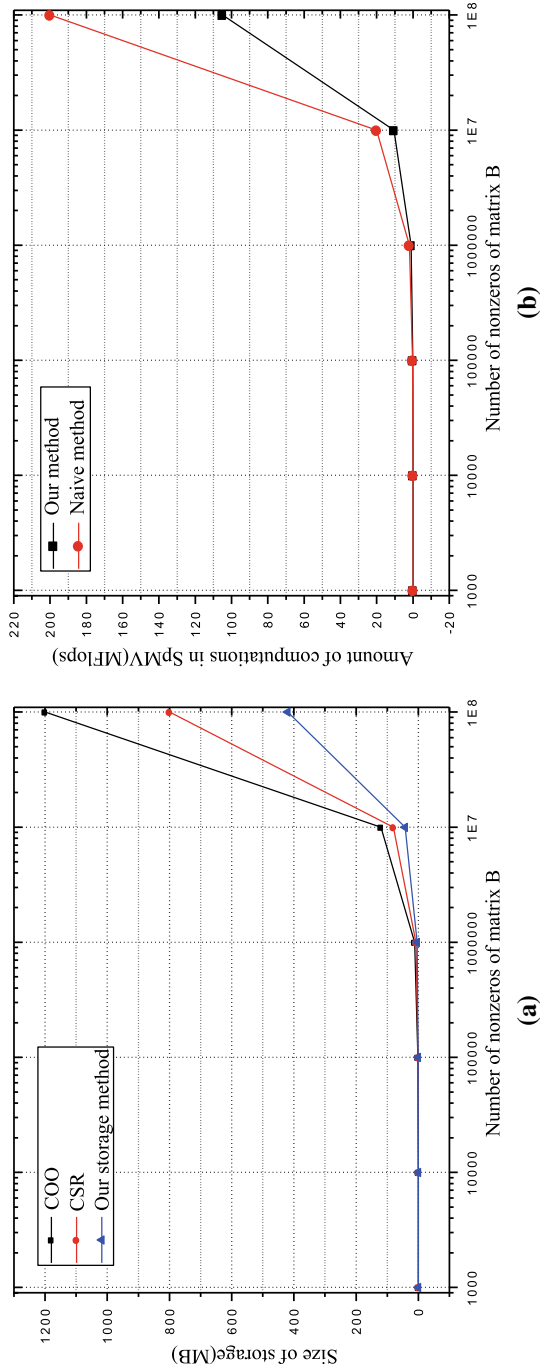


Fig. 5 The comparison of popular methods and our optimization method on size of storage place (a) and amount of computations (b) with various nonzeros of matrix B

**Table 2** Probability of all-zero-column and its expectation value with various processors and problem size

Problem size	Processors	64	128	256	512
$N = 10^6$	$p$	0.21	0.46	0.68	0.82
$d = 10^{-4}$	$E(p)$	$2.1 \times 10^5$	$4.6 \times 10^5$	$6.8 \times 10^5$	$8.2 \times 10^5$
$N = 10^6$	$p$	$1.6 \times 10^{-7}$	$4.0 \times 10^{-4}$	0.02	0.14
$d = 10^{-3}$	$E(p)$	0.16	403	$2.0 \times 10^4$	$1.4 \times 10^5$
$N = 10^7$	$p$	0.21	0.46	0.68	0.82
$d = 10^{-5}$	$E(p)$	$2.1 \times 10^6$	$4.6 \times 10^6$	$6.8 \times 10^6$	$8.2 \times 10^6$
$N = 10^7$	$p$	$1.6 \times 10^{-7}$	$4.0 \times 10^{-4}$	0.02	0.14
$d = 10^{-4}$	$E(p)$	1.64	$4.04 \times 10^3$	$2.0 \times 10^5$	$1.4 \times 10^6$
$N = 10^8$	$p$	0.21	0.46	0.68	0.82
$d = 10^{-6}$	$E(p)$	$2.1 \times 10^7$	$4.6 \times 10^7$	$6.8 \times 10^7$	$8.2 \times 10^7$
$N = 10^8$	$p$	$1.6 \times 10^{-7}$	$4.0 \times 10^{-4}$	0.02	0.14
$d = 10^{-5}$	$E(p)$	16	$4.0 \times 10^4$	$2.0 \times 10^6$	$1.4 \times 10^7$

### 4.1 Parallel algorithm of MSC

Considering that the computation for each column of  $X$  can be parallelly executed and other matrices take narrow space, we use  $m$  nodes in the parallel architecture. In the architecture, each node holds one column of  $X$  and stores other matrices duplicate. Thus, each node is responsible for the one column of  $X$  for the iterative SpMV; the computation will be executed independently without any communication. Algorithm 2 describes the parallel version of the MSC algorithm with  $m$  nodes and  $C$  processors in each node. Both vector  $x$  and matrix  $B$  are divided by row-oriented and stored into each processor, whereas  $Y$  being column-oriented is divided. Thus, each processor stores  $G = \frac{N}{C}$  rows of matrix  $B$ ,  $G$  elements of  $x$ , and  $G$  columns of  $Y$ .

In Algorithm 2,  $x_p^{(i)}$  is the intermediate result vector and the subscript indicates its owner processor, whereas its superscript  $i$  indicates identifiers of iteration. All  $m$  nodes need to execute  $\frac{N}{m} + \frac{N}{n} + O(1)$  iterations, and each iteration needs to compute  $x^{(i)} = B \times x^{(i-1)}$  (11–24) and  $Z = Y \times x^{(i)}$  (25–33). To compute  $x^{(i)} = B \times x^{(i-1)}$ , each processor must have complete vector  $x^{(i-1)}$ ; thus, it involves two interleaving operations, namely transferring  $x^{(i-1)}$  block vector and multiplication accumulation. A token exists to designate the processor that occupies the data bus to broadcast its own  $x^{(i-1)}$  block vector, and the others receive this broadcast until all of the processors have broadcasted and computed. Computing  $Z = Y \times x^{(i)}$  also involves two operations, namely computing local  $Z$  and merging all of the processors' local  $Z$  in processor 1. The “+” used for matrix in the algorithm signifies expanded addition for the sparse matrix. The execution flow of lines 11–33 in Algorithm 2 is also depicted in Fig. 6.

**Algorithm 2:** Parallel implementation algorithm for MSC

---

**Input:**  $B_{N \times N}, X_{N \times m}, Y_{n \times N}$ ;  
**Output:**  $Z_{n \times m}^{(i)}$ ;  $0 \leq i \leq \frac{N}{m} + \frac{N}{n} + O(1)$

- 1 %Use  $m$  nodes with  $C$  processors in each node to solve the problem;
- 2  $G = \frac{N}{C}$ ;
- 3 **forall the node**  $j$ ,  $1 \leq j \leq m$ , **processor**  $p$ ,  $1 \leq p \leq C$  **do**
- 4  $x_p^{(0)} = X(p * G - G + 1 : p * G, j)$ ;
- 5  $B_p = B(p * G - G + 1 : p * G, :)$ ;
- 6  $Y_g = Y(:, p * G - G + 1 : p * G)$ ;
- 7 **end**
- 8 %Store  $x_p^{(0)}, B_p$  and  $Y_p$  in processor  $p$ , node  $j$ ; Each node calculates a column of  $X$  by  $C$  processors, each processor calculates  $G$  elements;
- 9 **forall the node**  $j$ ,  $1 \leq j \leq m$  **do**
- 10 **for**  $i = 1, i \leq \frac{N}{m} + \frac{N}{n} + O(1), i++$  **do**
- 11 **forall the processor**  $p$ ,  $1 \leq p \leq C$  **do**
- 12  $x_p^{(i)} = 0$ ;
- 13 **for**  $k = 1; 1 \leq k \leq C; k++$  **do**
- 14 **if**  $token == me$  **then**
- 15 Broadcast  $x_p^{(i-1)}$ ;
- 16  $x_p^{(i)} += B_{pp} \times x_p^{(i-1)}$ ;
- 17 **end**
- 18 **else**
- 19 Receive  $x_t^{(i-1)}$ ; % A block of X from processor  $t$ ;
- 20  $x_p^{(i)} += B_{pt} \times x_t^{(i-1)}$ ;
- 21 **end**
- 22 **end**
- 23 **end**
- 24 **end**
- 25 **forall the processor**  $p, 2 \leq g \leq C$  **do**
- 26  $Z_p^{(i)} = Y_p \times x_p^{(i)}$ ;
- 27 Send  $Z_p^{(i)}$  to processor 1;
- 28 **end**
- 29 **for processor 1 do**
- 30  $Z_p^{(i)} = Y_p \times x_p^{(i)}$ ;
- 31 Receive  $Z_p^{(i)}$  from other  $C - 1$  processors;
- 32  $Z^{(i)} += Z_p^{(i)}$ ;
- 33 **end**
- 34  $Z_j^{(i)} = Z^{(i)}$ ;
- 35 **end**

---

## 4.2 Custom machine for MSC

### 4.2.1 Interconnection structure

All of the processors in one node only involve the transfer of  $x$  vector block when ignoring the transfer of  $Z$ . In addition, we have evaluated that the broadcast method will satisfy the demands; thus, this paper takes the bus structure as the interconnect

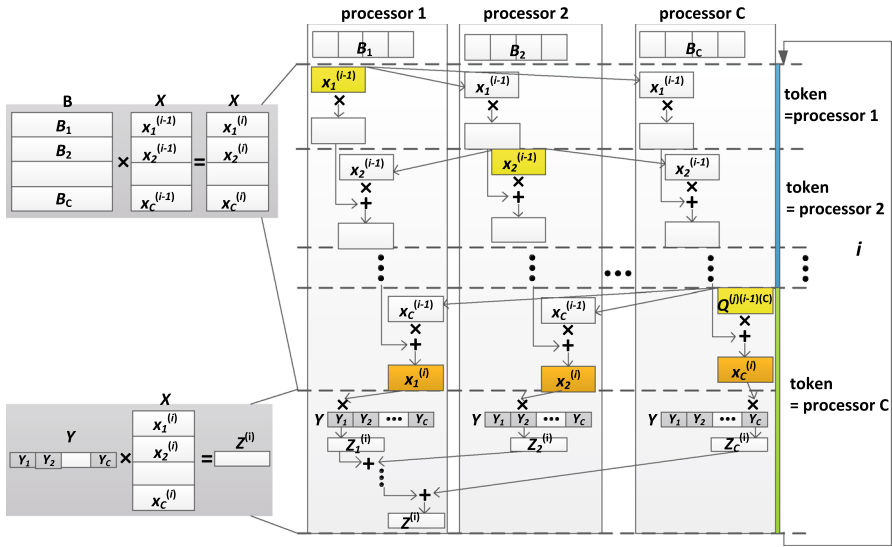


Fig. 6 Execution flow of lines 11–33 in Algorithm 2

structure among processors in one node. The data demands can be satisfied when all of the processors occupy the data bus and broadcast their own  $x$  vector block individually. Compared with the crossbars or other interconnected schemes, the bus structure costs less resources with low complexity.

### 4.2.2 Memory structure

Nearly all of the data accessed from the main memory are concentrated on lines 14–20 in Algorithm 2 and their volume is at the TB level, whereas others stay on the MB level when no cache memory exists.  $x_p^{(i-1)}$  involves twice the memory accessed for broadcast and computation. The broadcast will access the entire  $x_p^{(i-1)}$  at once, whereas the computation needs to access the elements of  $x_p^{(i-1)}$  for  $nnz(B_{pp})$  times because each non-zero element of  $B_{pp}$  will access one element in  $x_p^{(i-1)}$  from memory when disregarding the cache memory.  $B_{pp}$  involves a one-time load from memory, and  $x_p^{(i-1)}$  involves a one-time load and one-time storage. Hence, when ignoring the other operation, the total data volume accessing from the main memory for one processor per iteration is  $Dv = volumn(x_p^{(i-1)}) + nnz(B_{pp}) \times s_x + volumn(B_{pp}) + volumn(x_p^{(i-1)}) = G \times s_x + d \times G^2 \times s_x + d \times G^2 \times s_b + G \times s_x = 2 \times G \times s_x + d \times G^2 \times (s_x + s_b)$ .

When considering cache memory, given that the cache can store  $f$  elements of  $x$ , that is, the size of the cache is  $s_x \times f$  bytes. Algorithm 3 shows the implementation algorithm of  $x_p^{(i)} + = B_{pp} \times x_p^{(i-1)}$ , which corresponds to line 16 or line 20 in Algorithm 2. The algorithm contains  $Blocks = \lceil \frac{G}{f} \rceil$  sub-steps. Each sub-step contains one load and multiplication accumulation. Each load operation will access  $f$

---

**Algorithm 3:** Parallel implementation algorithm for  $x_p^{(i)} + = B_{pp} \times x_p^{(i-1)}$  with Cache

---

**Input:**  $B_{pp}, x_p^{(i-1)}$ ;  
**Output:**  $x_p^{(i)}$

- 1 Divide  $B_{pp}$  based on the size of cache  $f$ ;
- 2  $Blocks = \lceil \frac{G}{f} \rceil$ ; **for**  $q = 1, q \leq Blocks, q++$  **do**
- 3      $Load(x_p^{(i-1)}((q-1) \times f + 1 : q \times f))$ ;
- 4      $x_p^{(i)}((p-1) \times f + 1 : p \times f) + = B_{pp}(:, (q-1) \times f + 1 : q \times f) \times x_p^{(i-1)}((q-1) \times f + 1 : q \times f)$ ;
- 5 **end**

---

elements of  $x$  from the main memory to cache. Multiplication accumulation will load and store  $G$  elements of  $x_p^{(i)}$  and load  $B_{G \times f}$  itself. Thus, the data volume for each sub-step that involves memory access is  $f \times s_x + 2 \times G \times s_x + d \times G \times f \times s_b$ . The total data volume for each processor per iteration is subsequently  $Dvc = Blocks \times (f \times s_x + 2 \times G \times s_x + d \times G \times f \times s_b) = G \times s_x + 2 \times \frac{G}{f} \times G \times s_x + d \times G^2 \times s_b$ .

Table 3 compares the memory access data volume with different sizes of cache and without cache (when cache size equals to zero) when the processors, matrix size, and sparsity ratio vary. Table 3 shows that when the matrix size increases to  $10^7$  and the row weight (number of nonzeros per row) is less than  $10^3$ , the scheme without cache is better than the one with cache because of the less main memory access volume and resources.

### 4.2.3 Computation architecture

The multiplication accumulation should be executed in three independent parts,  $B_1, B_2$ , and  $B_3$ , because of the manner of storing matrix  $B$ . The computation architecture for this operation is shown in Fig. 7. The blocks in this picture, except for MUX for multiplexing, are organized according to the FIFO (first input first output) scheme for storing data, and the shaded ones indicate that the data stored are directly accessed from the main memory.

The data flow is primarily controlled by the state of all of the FIFOs. First, we load the indexes of the three parts of matrix  $B$ , intermediate result  $x_p^{(i)}$ , and matrix elements into their own FIFO. We subsequently load the corresponding  $x_p^{(i-1)}$  elements based on the column index of matrix nonzeros. We store the vector elements indexed by the  $B_1$  part into FIFO x1, the  $B_2$  part into FIFO x2, and the  $B_3$  part into x3. For the first two parts, the multiplication accumulation will degenerate into addition or subtraction accumulation because the matrix values are 1 or  $-1$ . Only the third part of the matrix will maintain multiplication accumulation. The intermediate accumulation result will be stored into Q1, Q2, and Q3. Subsequently, these results will be merged into  $x_p^{(i)}$  based on their row indexes. Finally, the merged result  $x_p^{(i)}$  will be written into its corresponding main memory partition part.

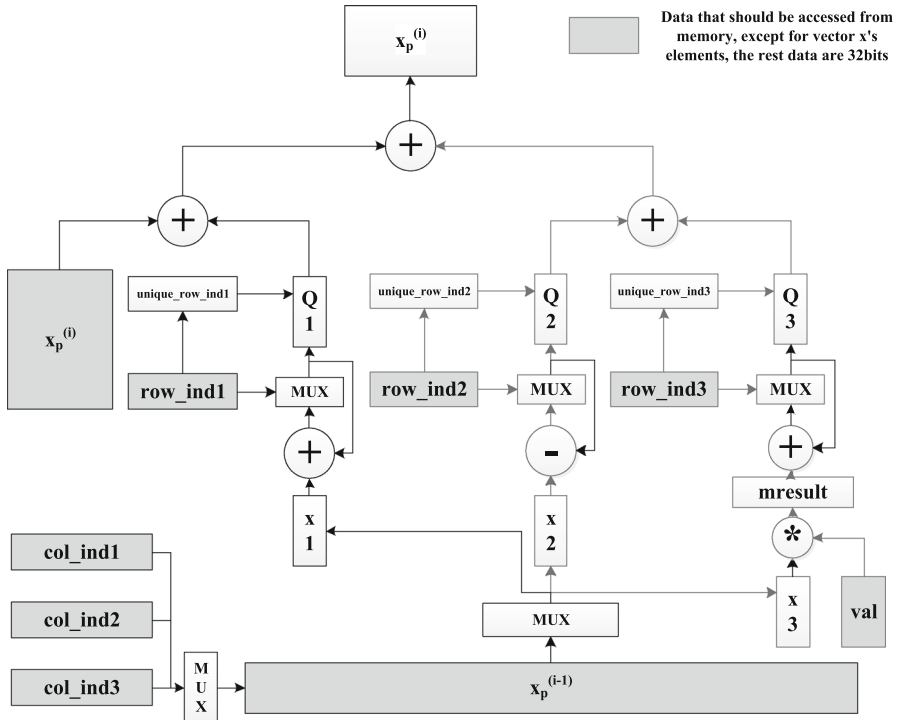
**Table 3** Data volume involving memory access with various size of cache and varies processors (GB)

Problem size	Cache size	$C = 64$	$C = 128$	$C = 256$	$C = 512$
$N = 10^6$ $d = 10^{-4}$	0 MB ( $f = 0$ )	0.0072	0.0028	0.0012	0.0006
	8 MB ( $f = 0.0625$ M)	0.0031	0.0013	0.0006	0.0003
	16 MB ( $f = 0.125$ M)	0.0026	0.0011	0.0005	0.0003
	32 MB ( $f = 0.25$ M)	0.0023	0.0011	0.0005	0.0003
$N = 10^6$ $d = 10^{-3}$	0 MB ( $f = 0$ )	0.0362	0.0010	0.0030	0.0010
	8 MB ( $f = 0.0625$ M)	0.0040	0.0015	0.0006	0.0003
	16 MB ( $f = 0.125$ M)	0.0035	0.0014	0.0006	0.0003
	32 MB ( $f = 0.25$ M)	0.0032	0.0013	0.0006	0.0003
$N = 10^7$ $d = 10^{-5}$	0 MB ( $f = 0$ )	0.0722	0.0281	0.0120	0.0055
	8 MB ( $f = 0.0625$ M)	0.1210	0.0352	0.0113	0.0041
	16 MB ( $f = 0.125$ M)	0.0710	0.0227	0.0082	0.0033
	32 MB ( $f = 0.25$ M)	0.0460	0.0165	0.0066	0.0029
$N = 10^7$ $d = 10^{-4}$	0 MB ( $f = 0$ )	0.3623	0.1006	0.0301	0.0100
	8 MB ( $f = 0.0625$ M)	0.1298	0.0374	0.0119	0.0042
	16 MB ( $f = 0.125$ M)	0.0798	0.0249	0.0087	0.0034
	32 MB ( $f = 0.25$ M)	0.0548	0.0187	0.0072	0.0030
$N = 10^8$ $d = 10^{-6}$	0 MB ( $f = 0$ )	0.7223	0.2806	0.1201	0.0550
	8 MB ( $f = 0.0625$ M)	10.2098	2.6024	0.6756	0.1814
	16 MB ( $f = 0.125$ M)	5.2098	1.3524	0.3631	0.1033
	32 MB ( $f = 0.25$ M)	2.7098	0.7274	0.2069	0.0642
$N = 10^8$ $d = 10^{-5}$	0 MB ( $f = 0$ )	3.6227	1.0057	0.3014	0.1004
	8 MB ( $f = 0.0625$ M)	10.2977	2.6244	0.6811	0.1828
	16 MB ( $f = 0.125$ M)	5.2977	1.3744	0.3686	0.1047
	32 MB ( $f = 0.25$ M)	2.7977	0.7494	0.2124	0.0656

The execution for computing line 26 or line 30 in Algorithm 2 is the same as that for the  $B_1$  part of matrix, for the sake of  $Y$ 's elements being 1. Thus, introducing its detailed data flow is unnecessary.

## 5 Performance evaluation and verification

The entire computation as previously analyzed is SpMV operation, and the computation time can be overlapped by memory accessing time because of its low computation memory ratio. Thus, the total time is determined by the time of memory access and data communication among all of the processors. This section evaluates the performance of the parallel architecture previously introduced on the basis of the data volume involved in memory accessing and communication. The performance model will neglect these steps because the data involving lines 24–34 in Algorithm 2 are trivial.



**Fig. 7** Computation architecture. The *blocks* in this picture, except for MUX for multiplexing, are FIFO structure using for store data and the *shaded ones* indicate the data it stored are accessed from memory directly. Adders in picture operate 1024bits operators and multiplier for operator of 32 bits multiplies 1024 bits operator

### 5.1 Performance evaluation

Matrix  $B$  is split into  $C$  processors uniformly; hence, it takes  $d \times N^2 \times s_b$  bytes/ $C$  memory space in each processor. Vector  $x$  is also split into  $C$  processors uniformly, but two times more space should be spared, one for the received vector block and the other for the intermediate result. Thus, it takes  $3 \times N \times s_x$  bytes/ $C$ .

One processor will broadcast and receive a complete vector  $x$  per iteration; hence, the communication data volume is  $N \times s_x$  bytes. While  $x$  is stored in the main memory, the communication will also cause the same volume of memory access, which overlaps with communication. No communication occurs for matrix  $B$  because every processor holds its necessary matrix block  $B$ .

The memory access pattern in one iteration generally consists of regular access patterns over the matrix two times that of the intermediate vector  $x$  access: one for loading and the other for storage. This aspect results in  $d \times N^2 \times s_b$  bytes/ $C$  of  $B$  and  $2 \times N \times s_x$  bytes of  $x$  for memory access. Each nonzero of  $B$  will index a corresponding elements of  $x$  from memory in an irregular manner. This aspect results in  $d \times (N^2/C) \times s_x$  bytes of irregular memory access. The analyzed results are presented in Table 4.

**Table 4** Data volume involving memory store, access, and communication in one iteration

Data	Memory size	Memory access volume	Commun. volume
Matrix B	$d \times N \times N \times s_b B/C$	$d \times N \times N \times s_b B/C$	0
Matrix X ( $Q^{(i-1)}, Q^{(i)}$ )	$3 \times N \times s_x B/C$	Continuously $2 \times N \times s_x B$ Incontinuously $d \times (N^2/C) \times s_x B$ Continuously $N \times s_x B$ (overlapped with commun.)	$N \times s_x B$

We take  $Bandc$  and  $Bandm$  as the peak communication bandwidth and memory bandwidth with the efficient of  $Ec$  and  $Em$ , respectively. Provided that the regular memory access efficient is  $\alpha$  times than the irregular one, the total time required for solving this problem is as follows.

$$\begin{aligned}
 T &= \text{iterations} \times (\text{Commu.time} + \text{Memory time}) \\
 &= \left(\frac{N}{n} + \frac{N}{m}\right) \times ((N \times s_x B)/\min(Bandc \times Ec, Bandm \times Em) \\
 &\quad + (d \times N^2 \times s_b B/C + \alpha \times d \times (N^2/C) \times s_x B)/(Bandm \times Em)) \quad (3)
 \end{aligned}$$

Moreover, the entire memory capacity demanded for each processor is

$$M = d \times N^2 \times s_b B/C + 3 \times N \times s_x B/C \quad (4)$$

The memory data width is 512 bits for each processor; in our custom machine, each processor exploits eight memory access channels with 64 bits per channel and 8 burst length. The  $x$  element is 1024 bits; hence, the irregular memory access of  $x$  will use 2 burst length in 8, and the access efficient  $\alpha$  equals 4.

The system exploits DDR4 as its memory with 268 Gbps bandwidth per channel. The  $Bandm$  equals 268 GBps because eight channels exist on each processor. The interconnection exploits Intel QPI, which can obtain 400 Gbps with 16-way, that is,  $B$  and  $c = 400$  Gbps. The efficient of communication bandwidth is generally significantly high, given  $Ec = 1$  in this case.

$Em$  is determined by the feature of the algorithm itself and the manner of storing data. To ascertain its value in MSC, this paper uses the DRAMSim simulator to simulate the execution of MSC and obtain the trace of memory access. Based on the memory access trace,  $Em = 64\%$ , after ascertaining the major parameters of our architecture, we consider one specific problem as an example. For a specific problem size with  $n = m = 1000$ , Eq. 3 can calculate its performance, and the result is shown in Table 5 with various processors and  $s_b = 4$  bytes. Table 6 shows the performance evaluation when  $N = 10^8$ ,  $d = 10^{-4}$  with various processors and  $s_b$ . Observing that the communication time is independent of  $s_b$  and processor amount, the *Commu.time* is not shown in Table 6.

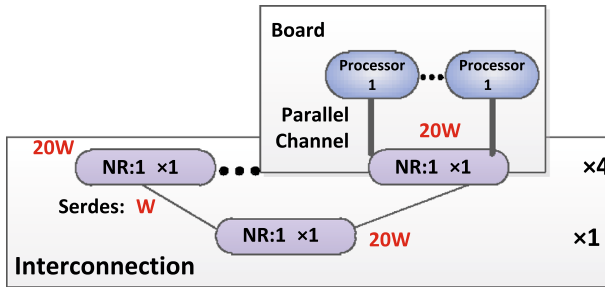


**Table 5** Evaluation performance with various processors but constant  $s_b$ 

Problem size	Processors	64	128	256	512
$N = 10^6$ $d = 10^{-4}$	Commu.time (s)	0.0026	0.0026	0.0026	0.0026
	Memory time (s)	0.0047	0.0024	0.0012	0.0006
	Exec.time (s)	0.0073	0.0049	0.0037	0.0031
	A vec.time (days)	0.0002	0.0001	0.00009	0.00007
$N = 10^6$ $d = 10^{-3}$	Commu.time (s)	0.0026	0.0026	0.0026	0.0026
	Memory time (s)	0.0470	0.0235	0.0118	0.0059
	Exec.time (s)	0.0496	0.0261	0.0143	0.0084
	A vec.time (days)	0.0011	0.0006	0.0003	0.0002
$N = 10^7$ $d = 10^{-5}$	Commu.time (s)	0.0256	0.0256	0.0256	0.0256
	Memory time (s)	0.0470	0.0235	0.0118	0.0059
	Exec.time (s)	0.0726	0.0491	0.0374	0.0315
	A vec.time (days)	0.0168	0.0114	0.0086	0.0073
$N = 10^7$ $d = 10^{-4}$	Commu.time (s)	0.0256	0.0256	0.0256	0.0256
	Memory time (s)	0.4701	0.2350	0.1175	0.0588
	Exec.time (s)	0.4957	0.2606	0.1431	0.0844
	A vec.time (days)	0.1147	0.0603	0.0331	0.0195
$N = 10^8$ $d = 10^{-6}$	Commu.time (s)	0.2560	0.2560	0.2560	0.2560
	Memory time (s)	0.4701	0.2350	0.1175	0.0588
	Exec.time (s)	0.7261	0.4910	0.3735	0.3148
	A vec.time (days)	1.6807	1.1366	0.8646	0.7286
$N = 10^8$ $d = 10^{-5}$	Commu.time (s)	0.2560	0.2560	0.2560	0.2560
	Memory time (s)	4.7006	2.3503	1.1752	0.5876
	Exec.time (s)	4.9566	2.6063	1.4312	0.8436
	A vec.time (days)	11.4737	6.0331	3.3129	1.9527

**Table 6** Evaluation performance with various processors and  $s_b$ 

Problem type	Processors	64	128	256	512
$s_b = 2$ bytes	Memory time (s)	46.8240	23.4120	11.7060	5.8530
	Exec.time (s)	47.0800	23.6680	11.9620	6.1090
	A vec.time (days)	108.9814	54.7870	27.6898	14.1412
$s_b = 4$ bytes	Memory time (s)	47.0062	23.5031	11.7515	5.8758
	Exec.time (s)	47.2622	23.7591	12.0075	6.1318
	A vec.time (days)	109.4032	54.9979	27.7952	14.1939
$s_b = 8$ bytes	Memory time (s)	47.3706	23.6853	11.8426	5.9213
	Exec.time (s)	47.6266	23.9413	12.0986	6.1773
	A vec.time (days)	110.2467	55.4196	28.0061	14.2994



**Fig. 8** The interconnection structure of the node and its power consumption

- *Commu.time* is the communication time per iteration. Given that the communicated data volume is constant, the time required will not change with various processors and is 0.256 s.
- *Memorytime* is the memory access time per iteration, and *Exec.time* is the sum of the time of the first two events.
- *A vec.time* is the time executed for one vector of matrix  $X$ . When using 1000 nodes, each node will calculate one vector, and the *A vec.time* is also the *Totaltime* for solving this entire problem.

Power consumption is also the machine's characteristic of user attention, and this paper evaluates the power consumption of the custom machine based on the current situation of products and TH2. Power consumption chiefly consists of three parts, namely processor chip power consumption, memory power consumption, and interconnection power consumption. We evaluate the power consumption of the system of 1000 nodes and each node with 64 processors below.

- *Processor chip power consumption:* Processor chip logic includes three parts: memory controller, network controller (NC), and computation logic. Each part will consume no more than 2 W based on our engineering experience; thus, we calculate processor chip consumption as 6 W.
- *Memory power consumption* The storage system exploits DDR4, and each processor is equipped with eight channels with 32 SDRAM chips in total. Each SDRAM chip, taking Micron's MT40A512M16 as an example, consumes 0.3 W, and the entire storage system per processor with  $Em = 64\%$  will consume 6.2 W.
- *Interconnection power consumption* Building the data bus interconnection in one node requires five  $16 \times 16$  cross bar chips (NR) as shown in Fig. 8, and each cross bar chip takes 20 W. These cross bar chips are interconnected by SerDes interfaces, and each SerDes interface takes approximately 8 W. Thus, the interconnection module power consumption is 132 W per node.

Therefore, the total power consumption of the custom machine in a typical solving state is  $64,000 \times (6 + 6.2) + 1000 \times 132 = 0.92$  MW.

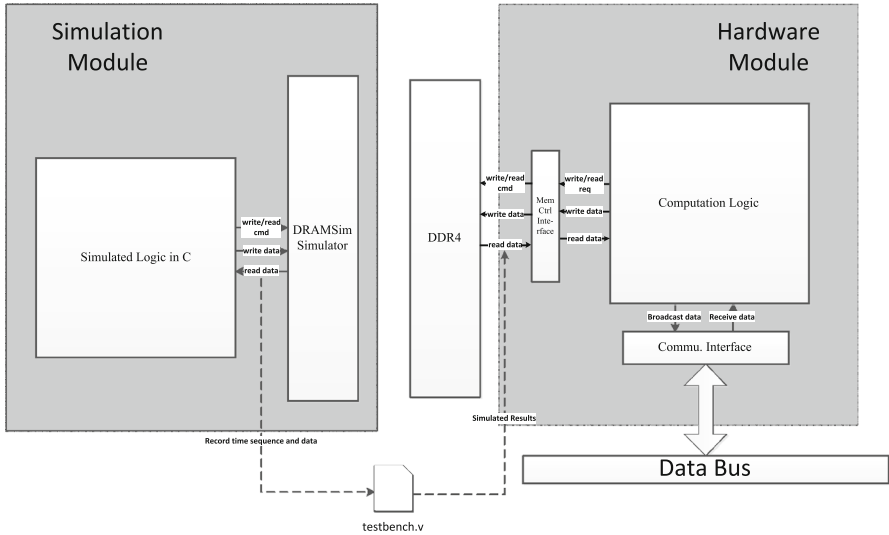


Fig. 9 Verification method

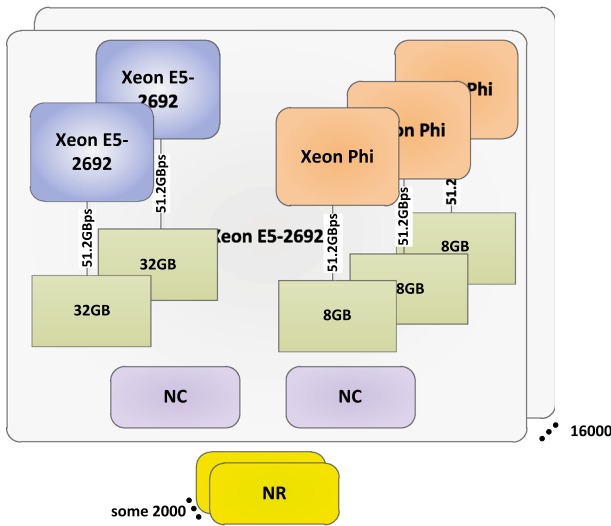
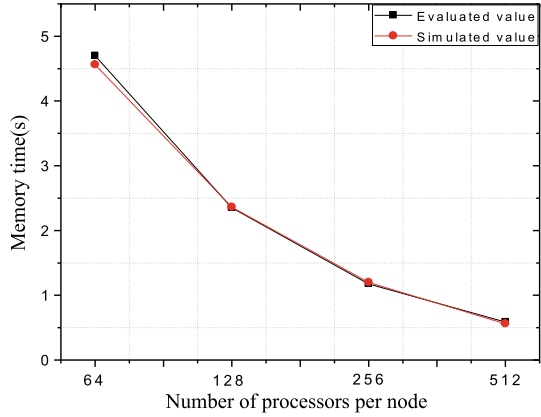
### 5.2 Performance verification

The preceding result is verified through a hardware module combined with a software module, as shown in Fig. 9. In the software module part, we use DRAMSim to simulate DDR4 and use C language to implement the simulation of a computation module. We subsequently record the data and clock information returned from DRAMSim into a file, named testbench.v. The hardware module is simulated by ModelSim, the computation logic is written by verilog code, and the interface of memory and interconnection logics is simulated by the IP processor from Xilinx Inc. In the consideration of its low simulating speed, we simulate a sequence of block matrix multiplications and deduce its performance, which is reasonable because the nonzeros are uniformly distributed. The testbench.v functions as an input file for the hardware module part, and ModelSim runs at 50 MHz.

Figure 10 shows the simulation results after the calibration into 2.1 GHz, which is the maximum working frequency of DDR4 and the comparison of our model evaluations with the problem size of  $N = 10^8, d = 10^{-5}$ . The evaluated value is consistent with our simulated value, thereby verifying that our performance model of MSC is correct and precise.

We also evaluate the performance on the fastest computer TH2 based on its configuration parameters depicted in Fig. 11 [7]. Referring to Fig. 11, we note that TH2 comprises 16,000 nodes, and each consists of two Intel Xeon E5 2692 and three Intel Xeon Phi processors; five processors are used in each node. The memory bandwidth is 51.2 GBps per processor. The system with some 2000 NR interconnection chips ensures that the communication bandwidth is higher than the memory bandwidth. To ensure that the system fits into our performance model, we take 16 nodes in TH2 as

**Fig. 10** The comparison of simulation and evaluation results



**Fig. 11** The configuration of TH2 which is consisted in 16,000 nodes with two Intel Xeon E5 2692 and three Intel Xeon Phi each

one node in our model; thus,  $16,000/16 = 1000$  nodes exist, each with  $16 \times 5 = 80$  processors.

Based on Eq. 3, Fig. 12 compares the execution time of our custom machine with  $n$  node and each node with various processors and TH2 on some specific problems, including  $N = 10^6-10^9$  and  $d = 10^{-3}-10^{-8}$  when  $n = m = 1000$ . The evaluation performances are commonly higher than TH2 and can obtain average  $3 \times$  speedup with 1000 nodes, each with 64 processors. The power consumption of TH2 is 17.8 MW, whereas our custom machine consumes merely 0.92 MW, which is far below that of TH2.

Dimitrios Meintanis [16] in 2009 also designed a hardware architecture for MSC on FPGA. The implementation of 1024 Virtex-5 chips processes MSC for a  $N = 4 \times 10^7$

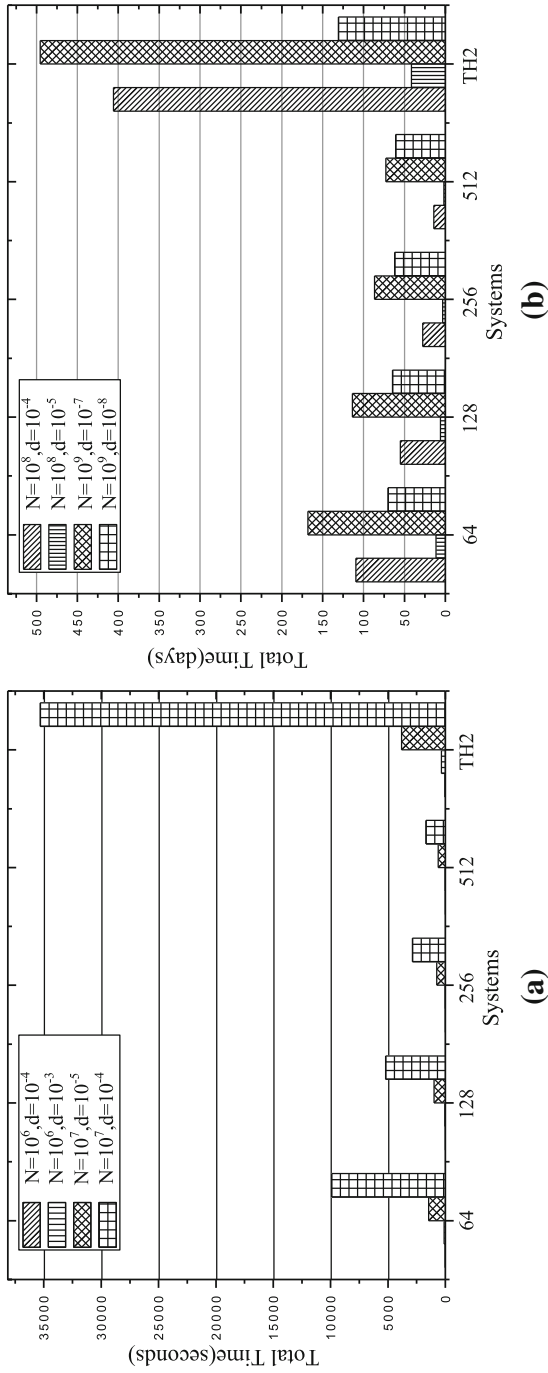


Fig. 12 The comparison of our custom machine with various processors and TH2

and  $d = 2.5 \times 10^{-6}$  matrix takes about 2.72 days, while the same problem may take one node with 64 processors of our proposed machine 0.2689 days, that is ,about  $10 \times$  speedup.

## 6 Conclusion

This paper proposes a custom machine for iterative large sparse matrix–vector multiplication in block Wiedemann algorithm. To maximize the full potential performance of SpMV in MSC, four optimization strategies are suggested in the aspects of computation, communication, and storage. Based on these strategies, we proposed the parallel algorithm of MSC and a custom machine includes an interconnection structure, a memory structure, and a computation architecture. Subsequently, a performance model is built to evaluate the execution of MSC, which shows a  $3 \times$  speed up and reduced power consumption for our custom machine when compared with TH2.

**Acknowledgments** This work was funded by National Natural Science Foundation of China (number 61303070). We acknowledge TH-1A supercomputing system service to support our simulation. We would like to thank the reviewers for their helpful comments.

## References

1. Anzt H, Tomov S, Dongarra J (2015) Energy efficiency and performance frontiers for sparse computations on GPU supercomputers. In: Proceedings of the sixth international workshop on programming models and applications for multicores and manycores, pp 1–10. ACM
2. Aoki K, Shimoyama T, Ueda H (2007) Experiments on the linear algebra step in the number field sieve. In: Atsuko M, Hiroaki K, Kai R (eds) Advances in information and computer security, pp 58–73. Springer, Berlin
3. Awad M (2009) FPGA supercomputing platforms: a survey. In: International conference on field programmable logic and applications, 2009. FPL 2009, pp 564–568. IEEE
4. Baskaran MM, Bordawekar R (2008) Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. IBM Reserach Report, RC24704 (W0812-047)
5. Buluç A, Gilbert JR (2008) On the representation and multiplication of hypersparse matrices. In: IEEE international symposium on parallel and distributed processing, 2008. IPDPS 2008, pp 1–11. IEEE
6. Çatalyürek UV, Aykanat C (2001) A fine-grain hypergraph model for 2D decomposition of sparse matrices. In: Parallel and distributed processing symposium. Proceedings 15th international, pp 1199–1204. IEEE
7. Chen C, Du Y, Jiang H, Zuo K, Yang C (2014) HPCG: preliminary evaluation and optimization on Tianhe-2 CPU-only nodes. In: 2014 IEEE 26th international symposium on computer architecture and high performance computing (SBAC-PAD), pp 41–48. IEEE
8. Coppersmith D (1994) Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math Comput* 62(205):333–350
9. Dave N, Fleming K, King M, Pellauer M, Vijayaraghavan M (2007) Hardware acceleration of matrix multiplication on a xilinx FPGA. In: 5th IEEE/ACM international conference on formal methods and models for codesign, 2007. MEMOCODE 2007, pp 97–100. IEEE
10. Dordopulo AI, Levin II, Doronchenko YI, Raskladkin MK (2015) High-performance reconfigurable computer systems based on virtex FPGAs. In: Victor M (ed) Parallel computing technologies, pp 349–362. Springer, Berlin
11. Dou Y, Vassiliadis S, Kuzmanov G, Gaydadjiev G (2005) 64-bit floating-point FPGA matrix multiplication. In: FPGA, pp 86–95. ACM, New York
12. Güneysu T, Paar C, Pfeiffer G, Schimmler M (2008) Enhancing copacobana for advanced applications in cryptography and cryptanalysis. In: International conference on field programmable logic and applications, 2008. FPL 2008, pp 675–678. IEEE

13. Kapre N, Moorthy P (2015) A case for embedded FPGA-based SoCs in energy-efficient acceleration of graph problems. *Supercomput Front Innov* 2(3):76–86
14. Kimball D, Michel E, Keltcher P, Wolf MM (2014) Quantifying the effect of matrix structure on multithreaded performance of the SPMV kernel. In: High performance extreme computing conference (HPEC), 2014 IEEE, pp 1–6. IEEE
15. Langr D, Tvrđik P (2015) Evaluation criteria for sparse matrix storage formats. *IEEE Trans Parallel Distrib Syst* 27(2):428–440
16. Meintanis D, Papaefstathiou I (2009) A module-based partial reconfiguration design for solving sparse linear systems over GF (2). In: International conference on field-programmable technology, 2009. FPT 2009, pp 335–338. IEEE
17. Pichel JC, Lorenzo JA, Heras DB, Cabaleiro JC (2009) Evaluating sparse matrix-vector product on the finiserrae supercomputer. In: 9th international conference on computational and mathematical methods in science and engineering, pp 831–842
18. Rajovic N, Carpenter PM, Gelado I, Puzovic N, Ramirez A, Valero M (2013) Supercomputing with commodity CPUs: are mobile SoCs ready for HPC? In: 2013 international conference for high performance computing, networking, storage and analysis (SC), pp 1–12. IEEE
19. Schmidt B, Aribowo H, Dang HV (2013) Iterative sparse matrix-vector multiplication for accelerating the block Wiedemann algorithm over GF (2) on multi-graphics processing unit systems. *Concurr Comput Pract Exp* 25(4):586–603
20. Sedaghati N, Ashari A, Pouchet LN, Parthasarathy S, Sadayappan P (2015) Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs. In: Proceedings of the 2nd workshop on parallel programming for analytics applications, pp 17–24. ACM
21. Stanisic L, Videau B, Cronsioe J, Degomme A, Marangozova-Martin V, Legrand A, Méhaut JF (2013) Performance analysis of HPC applications on low-power embedded platforms. In: Proceedings of the conference on design, automation and test in Europe, pp 475–480. EDA Consortium
22. Thomé E (2001) Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In: Proceedings of the 2001 international symposium on symbolic and algebraic computation, pp 323–331. ACM
23. Vastenhouw B, Bisseling RH (2005) A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev* 47(1):67–95