2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

An efficient hybrid MPI/OpenMP parallelization of the asynchronous ADMM algorithm

1st Qinnan Qiu
Shanghai University
School of Computer Engineering and Science
Shanghai, China
qqn1996@shu.edu.cn

2nd Yongmei Lei
Shanghai University
School of Computer Engineering and Science
Shanghai, China
lei@shu.edu.cn

3rd Dongxia Wang
Shanghai University
School of Computer Engineering and Science
Shanghai, China
wangdongxia1983@126.com

4th Guozheng Wang
Shanghai University
School of Computer Engineering and Science
Shanghai, China
gzh.wang@outlook.com

***Abstract*—Alternating direction method of multipliers (ADMM) is an efficient algorithm to solve large-scale machine learning problems in a distributed environment. To make full use of the hierarchical memory model in modern high-performance computing systems, this paper implements a hybrid MPI/OpenMP parallelization of the asynchronous ADMM algorithm (AH-ADMM). The AH-ADMM algorithm updates local variables in parallel by OpenMP threads and exchanges information between MPI processes, which relieves memory and communication pressure by replacing multi-processing with multi-threading. Furthermore, for the SVM problem, the AH-ADMM algorithm speeds up the calculation of sub-problems through an efficient parallel optimization strategy. This paper effectively combines the features of both algorithm design and programming model. Experiments on the Ziqiang4000 high-performance cluster demonstrate that the AH-ADMM algorithm scales better and run faster than the existing distributed ADMM algorithms implemented by pure MPI. The AH-ADMM can reduce the communication overhead by up to 91.8% and increase the convergence rate by up to 36x. For large datasets, the AH-ADMM can scale well on the cluster which over 129 cores.**

Keywords—distributed ADMM algorithm, asynchronous communication, hybrid parallel programming model, MPI, OpenMP

I. INTRODUCTION

The immense growth of data has made it necessary to solve large-scale machine learning problems in distributed environments. Therefore, how to take advantage of modern high-performance computing (HPC) resources to implement efficient distributed machine learning algorithms is an important problem. In general, many distributed machine learning problems can be expressed as the following global consensus optimization problem:

$$\min_{x_1,\dots,x_N,z} \sum_{i=1}^{N} f_i(x_i) + g(z) \;\; s.t. x_i = z, i = 1,\dots,N. \qquad (1)$$

Where $x \in \mathbb{R}^d$ represents the model parameter, $x_i \in \mathbb{R}^d$ is the local primal variable which is a local copy of the model parameter x on each node, $z \in \mathbb{R}^d$ is the global consensus variable, d is the number of the features. The objective function $f(x)$ is divided into N parts, where each $f_i: \mathbb{R}^d \to \mathbb{R}$ is the local objective function. $g: \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is the regularization function.

The alternating direction method of multipliers (ADMM) algorithm can derive the large global problem into smaller sub-problems and is suitable for a parallel solution in distributed environments [1]. The ADMM algorithm has been applied to solve a variety of machine learning problems, such as Linear regression [2], Support vector machine (SVM) [3], and many others. References [4][5] have been proved that there are advantages in solving SVM problems using the ADMM algorithm in a distributed environment. In this paper, we use the ADMM algorithm mainly to solve the optimization problem of SVM. For the parallelization and distributed implementation of the ADMM algorithm, one of the main approaches uses Message Passing Interface (MPI), such as [6]-[10]. However, the communication overhead and memory footprint will easily become limitations on the scalability of the ADMM algorithm when one tries to improve the time to solution by using a large number of cores. In addition, the lack of fine-grain thread parallelism cannot effectively utilize shared memory of multi-core computing resources.

Multiple computing cores are becoming ubiquitous. Most HPC architectures comprise clusters of multi-core CPU nodes interconnected via a high-speed network supporting a hierarchical memory model—shared memory within a single node and distributed memory across the nodes [11]. It is meaningful to study how to utilize higher core counts of modern multi-core CPUs and the characteristics of the heterogeneous memory model. Rabenseifner et al. [12] proposed a hybrid MPI/OpenMP parallel programming model. OpenMP is a shared memory programming model which can provide automatic guidance for multithreaded parallel strategies. It makes the implementation of the parallel applications relatively easy but only run on the shared memory [11]. The hybrid MPI/OpenMP parallel programming model combines distributed memory parallelization on the node interconnect with shared memory parallelization inside each node. Experiments verify that the hybrid version outperforms the pure MPI version for several machine learning algorithms [13][14]. Mironov et al. [15] analyzed and proved the effectiveness of the hybrid MPI/OpenMP parallel programming model in reducing the overall memory footprint of the Hartree-Fock method. However, for the ADMM algorithms, very few efforts have focused on the efficient implementation of hybrid parallelization so far. The difficulty in implementing applications by hybrid MPI/OpenMP approach is how to combines the characteristics of the algorithm and programming models.

978-1-6654-3574-1/21/$31.00 ©2021 IEEE

DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00083

In this paper, in order to make full use of the computing resources of multicore nodes and the advantages of the heterogeneous memory model in modern high-performance computing systems, we propose an asynchronous ADMM algorithm based on a hybrid MPI/OpenMP programming model (AH-ADMM). The AH-ADMM algorithm combines algorithm design and programming models. In general, the algorithm is logically divided into two-layer models. The fine-grained thread parallelism within nodes is implemented in the local updating layer, and the process parallelism between nodes is implemented in the global updating layer. The main contributions of this paper include:

(1) Our proposed model uses multi-threading instead of multi-processing to calculate the sub-problem, which reduces the number of worker processes on a node. Therefore, the memory footprint and communication cost of the ADMM algorithm are saved.

(2) Since the calculation of sub-problems in the ADMM algorithm is time consuming [16], we also design an efficient multithreading optimization strategy to speed up the calculation of the sub-problems.

(3) The asynchronous communication strategy can reduce the waiting overhead in global communication.

Overall, the hybrid parallel approach improves the convergence rate of the ADMM algorithm and the computing capacity of a single node, thereby can scale well on the large-scale HPC systems with multicore nodes. Experiments on the Ziqiang 4000 high-performance cluster of Shanghai University have proved, compared with AD-ADMM [6][7] and HAD-ADMM [8], the AH-ADMM algorithm has better performance and higher scalability.

The structure of this paper is as follows. Section 2 introduces the relevant background. Section 3 and 4 describe and analyze AH-ADMM. The results of the experiments are explained in Section 5. Finally, Section 6 concludes this paper and gives our future work.

II. Background

The ADMM fits well with the distributed system because of its natural parallel characteristic. How to implement efficient parallelization of the ADMM algorithm in modern HPC systems is our concern. With the development of multi-core CPUs, the hybrid parallel programming model has received more attention.

A. Distributed ADMM algorithm

Boyd et al. [1] start from constructing the augmented Lagrangian L_ρ to solve problem (1) by the ADMM algorithm. Then, $L_\rho(\{x_i\}, z)$ is minimized by updating x and z alternately. The expression of L_ρ is shown in (2) and the iterative formulas of the ADMM algorithm are shown in (3)-(5):

$$L_\rho(\{x_i\}, z, \{y_i\}) = \sum_{i=1}^{N}(f_i(x_i) + \frac{\rho}{2}\left\|x_i + \frac{y_i}{\rho} - z\right\|_2^2) + g(z). \quad (2)$$

$$x_i^{k+1} = \underset{x_i}{\operatorname{argmin}}(f_i(x_i) + x_i^T y_i^k + \frac{\rho}{2}\|x_i - z^k\|_2^2). \quad (3)$$

$$z^{k+1} = \underset{z}{\operatorname{argmin}}(g(z) + \frac{\rho}{2}\sum_{i=1}^{N}\left\|x_i^{k+1} + \frac{y_i^k}{\rho} - z\right\|_2^2). \quad (4)$$

$$y_i^{k+1} = y_i^k + \rho(x_i^{k+1} - z^{k+1}). \quad (5)$$

Where $y_i \in R^d$ is the local dual variable of x_i, $\rho > 0$ is the penalty parameter. For the original synchronous ADMM algorithm [1], a process called master needs to wait for all the worker processes finishing their sub-problem calculation before it can proceed. Due to the difference in computing speed and network delays between workers, the synchronization overhead becomes the bottleneck of algorithm efficiency. Therefore, the asynchronous distributed ADMM (AD-ADMM) algorithm [6][7] becomes a new research hotspot. In the AD-ADMM algorithm, two conditions to control the asynchrony: partial barrier $0< S \leq N$ and bounded delay $\tau > 0$. Through these two conditions, the waiting time of each iteration is reduced, and the convergence of the algorithm is guaranteed.

While the implementation of AD-ADMM has been remarkably successful, it has the bottlenecks of memory footprint and communication cost. Recent ADMM parallelization efforts focused on the optimization of communication. Wang et al. [8] proposed an asynchronous distributed ADMM based on a hierarchical communication structure (HAD-ADMM). The HAD-ADMM algorithm groups the processes and sets a master process in each node called submaster. The master process is set up to communicate with each submaster process, which basically achieves load balancing. Xie et al. [10] implemented a hierarchical ring-allreduce communication architecture to reduce the inter-node communication cost. Wang et al. [9] analyzed the features of high-dimensional sparse datasets and designed a strategy to reduce the information transmitted between nodes. These algorithms are all implemented by pure MPI. The major issues are additional data replication and communication overhead between processes within a node. Our work starts with the approach of parallelization implementation, aiming at improving the computing capacity of a single node and the scalability of the ADMM algorithm.

B. Hybrid MPI/OpenMP programming model

The hybrid MPI/OpenMP parallel programming model fits well with the hierarchical memory system. Jin et al. [11] have proved that the hybrid approach can help to reduce the demand for resources (such as memory and network), which is very important for running jobs of large-scale datasets. However, the parallelization strategy between threads will affect the performance of the algorithm implemented by the hybrid approach and the hierarchical programming model may lead to complex application codes. These are problems we need to solve. Rabenseifner et al. [12] pointed out, the hybrid model can be applied well to a certain class of applications with easily exploitable multi-level parallelism. The characteristic of the ADMM algorithm makes itself very suitable to be implemented by the hybrid model. The specifics will be discussed in Sections 3 and 4.

III. Hierarchical Asynchronous Distributed ADMM Algorithm

The ADMM algorithm can divide the original problem into multiple sub-problems, and each sub-problem can be solved in parallel on a separate node. Because of the independence of sub-problems and the characteristics of multi-level updating, we logically divide the algorithm updating procedure into two layers, which are the local updating layer and the global updating layer. This hierarchical

updating also makes the algorithm more adaptable to the hybrid MPI/OpenMP parallel programming model.

Given a dataset $A \in R^{l\times d}$, l is the number of samples. We let $A_i \in R^{l_i\times d}$ be the i-th partition of all data, where $\sum_{i=1}^{N} l_i = l$. We consider the L2-regularized L2-Loss SVM problem. The global consensus problem we need to solve can be described as formula (6):

$$\min_{x_i,\dots x_{N,z}} f(x) = \frac{1}{2}x_i^T x_i + C\sum_{i=1}^{N}\sum_{j\in A_i}\max(1 - b_j x^T a_j, 0)^2 + \sum_{i=1}^{N}\frac{\rho}{2}\|x_i - z\|_2^2. \quad s.t.\ \ x_i = z\ \ i = 1,\dots,N \tag{6}$$

Where C is a hyperparameter, $a_j \in R^d$ is the feature vectors of the j-th sample of A_i, $b_j \in \{-1, +1\}$ is the label of a_j.

A. Local Updating Layer

In this layer, the AH-ADMM updates the local primal variable and the local dual variable by worker processes. The AH-ADMM allocates a process to each node and makes one of the nodes as the master. Each worker is responsible for updating the local primal variable and local dual variable in parallel as formula (7)-(8):

$$x_i^{k_i+1} = \underset{x_i}{\operatorname{argmin}}(C\sum_{j\in A_i}\max(1 - b_j x_i^T a_j, 0)^2 + x_i^T y_i^{k_i} + \frac{\rho}{2}\|x_i - \hat{z}\|_2^2). \tag{7}$$

$$y_i^{k_i+1} = y_i^{k_i} + \rho(x_i^{k_i+1} - \hat{z}). \tag{8}$$

Where k is the clock the master keeps, which starts from zero and is incremented by 1 after each z updating. Similarly, each worker has itself clock k_i. $\hat{z}$ is the newest global variable received from the master. To reduce the communication volume, the local primal variables and local dual variables can be aggregated as formula (9) in advance before they are sent to the master:

$$s_i^{k_i+1} = \rho x_i^{k_i+1} + y_i^{k_i}. \tag{9}$$

B. Global Updating Layer

In this layer, the AH-ADMM updates the global variable by the master. According to the two conditions (partial barrier: $0< S \le N$ and bounded delay: $\tau > 0$) of AD-ADMM [], the master process asynchronously receives the $\widehat{s_i}$ sent by workers i ($i \in S_k'$), stores them in s_i^{k+1} and reset 1 to τ_i simultaneously. The S_k' is the set of workers that have arrived when the clock of the master is k. The τ_i records the clock of worker i last arrival which is stored in ϕ_k. If a worker whose clock is greater than τ, the master should wait. When $S_k' \ge S$, the master can update the global variable as the formula (10). For the workers $i \notin S_k'$, the master uses the last variable s_i^k to participate in the calculation and add 1 to τ_i. Then workers $i \in S_k'$ receive the z^{k+1}.

$$z^{k+1} = \frac{1}{N\rho+1}\sum_{i=1}^{N}\widehat{s_i}. \tag{10}$$

The specific procedure is provided in Algorithm 1.

Algorithm 1: AH-ADMM: the asynchronous distributed ADMM based on the hybrid parallel programming model

AH-ADMM – local updating layer:

1: **initialize:** $k_i = 0, x_i^0 = 0, y_i^0 = 0, z^0 = 0.$
2: **repeat**
3: **wait** until receiving $\hat{z}$ from the master;
4: **Parallel update**
5: $x_i^{k_i+1}$ by (7);
6: $y_i^{k_i+1}$ by (8);
7: **update** $s_i^{k_i+1}$ by (9);
8: **send** $s_i^{k_i+1}$ to the master;
9: **set** $k_i \leftarrow k_i + 1$;
10: **until** the stopping criterion is satisfied.

AH-ADMM – global updating layer:

1: **initialize**: $k = 0, s_i^0 = 0, i = 1,\dots N,$ $\{\tau_1, \tau_2, \dots, \tau_N\} = 0.$
2: **repeat**
3: **wait** until receiving $\widehat{s_i}$ from workers $i, i \in S_k'$ such that $S_k' \ge S$ and $\max(\tau_1, \tau_2, \dots, \tau_N) \le \tau$;
4: **for** worker $i \in S_k'$ **do**
5: $\tau_i \leftarrow 1$;
6: $s_i^{k+1} \leftarrow \widehat{s_i}$;
7: **end for**
8: **for** worker $i \notin S_k'$ **do**
9: $\tau_i \leftarrow \tau_i + 1$;
10: $s_i^{k+1} \leftarrow s_i^k$;
11: **end for**
12: **update** z^{k+1} by (10);
13: **broadcast** z^{k+1} to all the workers in the set S_k';
14: **set** $k \leftarrow k + 1$
15: **until** the stopping criterion is satisfied;

IV. Design and Implementation of the Hybrid MPI/OpenMP Parallelization of the Distributed Asynchronous ADMM Algorithm

The AH-ADMM algorithm is divided into two layers, which fits the hybrid MPI/OpenMP parallel programming model well. The details of the algorithm are described in Section 3. In this section, we mainly discuss the hybrid MPI/OpenMP parallelization implementation of the AH-ADMM algorithm. By the way, our hybrid parallel programming model is implemented on the original asynchronous ADMM algorithm framework in this paper, but can also be extended to ADMM algorithms based on other distributed frameworks, such as parameter servers, etc.

Our main idea is to implement hybrid parallelism through a multi-level parallelism mechanism. This hybrid parallel programming model is adapted to the hierarchical memory model of HPC systems. We divide the hybrid model into node-level parallelism and cluster-level parallelism. For different updating layers, the different levels of programming models are applied by analyzing the characteristics of the distributed AH-ADMM algorithm.

A. The implementation of the node-level parallelism by OpenMP:

As mentioned earlier, the updating of the local primal variable x_i is an independent sub-problem to each worker and the calculation of sub-problems in the distributed ADMM algorithm is time consuming. Therefore, on the local updating layer, we assign one MPI process on each worker node to update the local variables (x_i, y_i). Meanwhile, the multi-threading is used to update the local variables in parallel on the shared memory by OpenMP. It should be noted that the local update layer has two parallel levels: node-level and

cluster-level.

There are a master thread and T threads in the OpenMP parallel region of each MPI process. OpenMP is used to control T threads to calculate the sub-problem(x_i) and y_i in parallel, the master thread is responsible for waking up parallel threads and obtains the sum computed by the threads. The updating of y_i is easy, but the updating of sub-problem by different optimization algorithms usually lead to complex codes. Therefore, we only implement the parallelization of partial operations which are the bottleneck of sub-problem calculation. We will introduce the parallel optimization strategy for the sub-problem in detail in Section 4.2.

B. *The implementation of the cluster-level parallelism by MPI:*

For the updating of the global variable, the AH-ADMM algorithm needs to receive all local variables and send the global variable to workers. Therefore, cluster-level parallelism is implemented for the global updating layer.

The MPI is used to transfer the variables across distributed memory nodes and the communication mode is changed to asynchronization. MPI is used for communication between processes. On each worker node, the MPI calls are performed by the master thread of the MPI processes which sends the master s_i. On the master node, a master process is set. The model parameters s_i is received and the global variable z is sent by the master process. The blocking MPI calls and wait calls are for fine control of asynchronous message passing.

Fig. 1 shows the sequence of important events that may occur during the iteration of the AH-ADMM algorithm. The sub-problems are calculated in parallel by multi-threading on each worker node. Since $S = 2$, the master only needs to wait for two updated parameters x_i of worker $i \in S'_k$ arriving to update z, and then send z to worker $i \in S'_k$. When the master receives the updating of worker 2, it cannot update immediately. This is because $\tau_1 > \tau$ and the master needs to wait for the updating of worker 1 reaching to update z.

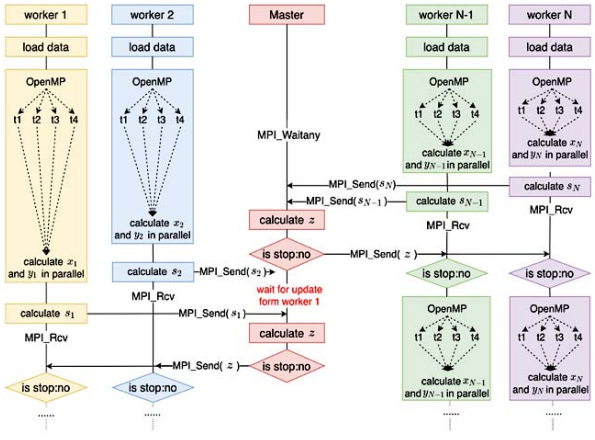

Fig. 1. The sequence of important events in an example run of AH-ADMM where S=2,τ=1. The dashed arrow represents the OpenMP is used, and the solid arrow represents the MPI is used.

C. *Parallel Optimization Strategy for the Sub-problem*

1) Parallel dual coordinate descent algorithm: The ADMM algorithm provides the freedom to propose efficient methods for solving the sub-problems in distributed machines. In this section, we use the parallel dual coordinate descent algorithm to solve the L2-regularized L2-Loss SVM problem. The dual form of (7) can be written as the formula (11):

$$\min_{\alpha} f(\alpha) = \frac{1}{2\rho}\alpha^T \bar{Q}\alpha - d^T\alpha \quad s.t.\, \alpha_j \geq 0, \quad \forall j. \tag{11}$$

Where $\bar{Q} = Q + D$, $Q_{ij} = b_i b_j a_i^T a_j$, D is a diagonal matrix, $D_{jj} = \rho/(2C)$, $d = [1 - b_1 v^T a_1, \dots, 1 - b_s v^T a_s]^T$, $\{a_1 \dots a_s\}$ denotes the data in A_i and $v = z^k - \frac{y_i^k}{\rho}$.

The dual coordinate descent algorithm [17] optimizes one variable in α at a time and then circularly moves to the next variable and so on. In other words, for any j, we can optimize α_j while other variables are fixed. Let $\mathrm{G}_j^{(t)}$ be the partial derivative of $f(\alpha)$ with respect to α_j at the t-th iteration, then we have:

$$\mathrm{G}_j^{(t)} = b_j \boldsymbol{w}^{(t)^T} a_j - 1 + D_{jj}\alpha_j^{(t)}. \tag{12}$$

Where $\boldsymbol{w}^{(t)} = \sum_{j=1}^{s} b_j \alpha_j^{(t)} a_j + v$. Thus, the optimal α_j will be the root of $\mathrm{G}_j^{(t)}$ projected on $[0, \infty)$. We can update α_j as:

$$\alpha_j^{(t+1)} = \max\left(\alpha_j^{(t)} - \frac{\mathrm{G}_j^{(t)}}{\bar{Q}_{jj}}, 0\right). \tag{13}$$

The main calculation task of each iteration of the dual coordinate descent method is (12), so parallel optimization is carried out for this part. We apply and improve the parallel dual coordinate descent method in [18] to make it suitable for parallel optimization of the sub-problems in the distributed ADMM algorithm. Firstly, select a set B and split all data $\{a_1, \dots, a_s\}$ to blocks. Then calculate the $\mathrm{G}_j^{(t)}$ with each block in parallel. The α_j can be parallelized by using OpenMP:

1: **for all** $j \in B$ **do in parallel**
2: $\quad \mathrm{G}_j^{(t)} \leftarrow b_j \boldsymbol{w}^{(t)^T} a_j - 1 + D_{jj}\alpha_j^{(t)}$
3: $\quad \alpha_j^{(t+1)} \leftarrow \max\left(\alpha_j^{(t)} - \frac{\mathrm{G}_j^{(t)}}{\bar{Q}_{jj}}, 0\right)$
4: $\boldsymbol{w}^{(t+1)} = \sum_{j=1}^{s} b_j \alpha_j^{(t+1)} a_j + v$

However, we can see that the calculation of $\boldsymbol{w}^{(t)}$ becomes the bottleneck because it is much more time-consuming than the updating of $\alpha_j^{(t)}$ and the calculation is repeated for each iteration. Calculating $\boldsymbol{w}^{(t)}$ in parallel is much more difficult than calculating $\mathrm{G}_j^{(t)}$ in parallel because two threads may want to update the same component of $\boldsymbol{w}^{(t)}$ simultaneously. Instead, we design the algorithm so that the $\boldsymbol{w}^{(t)}$ update only accounts for a small part of the total calculation. Algorithm 2 shows the details:

Algorithm 2: A parallel dual coordinate descent algorithm for solving the sub-problems:

1: **Initialize** $\alpha^{(0)}, w^{(0)} = \sum_{j=1}^{s} b_j \alpha_j^{(0)} a_j + v$
$\quad$ and $\varepsilon, 0 < \bar{\varepsilon} \ll \varepsilon$.
2: **while** true **do**
3: $\quad G_{MAX} \leftarrow -\infty$;
4: $\quad$ Split $\{1, \dots, s\}$ to $\bar{B}, \dots, \overline{B_T}$;
5: $\quad t \leftarrow 0$;
6: $\quad$!$omp parallel shared ($\alpha$, $\boldsymbol{w}$, G) & private(j)
7: $\quad ithread \leftarrow$omp_get_thread_num();

8: **for** $\bar{B}$ in $\bar{B}, \ldots, \overline{B_T}$ **do**
9: !$omp do schedule(static)
10: **for all** $j \in \bar{B}$ **do in parallel**
11: $G_j^{(t)}(:, ithread) \leftarrow b_j \boldsymbol{w}^{(t)^T} a_j - 1 + D_{jj}\alpha_j^{(t)}$;
12: !$omp master
13: $G_{MAX} \leftarrow \max(G_{MAX}, \max_{j \in \bar{B}} |G_j^{(t)}|)$;
14: $B \leftarrow \left\{ j | j \in \bar{B}, |G_j^{(t)}| \geq \delta\varepsilon \right\}$;
15: **for all** $j \in B$ **do**
16: $d_j^{(t)} \leftarrow \max\left(\alpha_j^{(t)} - \frac{G_j^{(t)}}{\bar{Q}_{jj}}, 0\right) - \alpha_j^{(t)}$;
17: **if** $|d| \geq \bar{\varepsilon}$ **then**
18: $\alpha_j^{(t+1)} \leftarrow \alpha_j^{(t)} + d_j^{(t)}$;
19: $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + d_j^{(t)} a_j b_j$;
20: $t \leftarrow t + 1$;
21: !$omp end master
22: !$omp barrier
23: !$omp end parallel
24: **if** $G_{MAX} \leq \varepsilon$ **or** $t = 0$ **then**
25: break;

Where ε is the stopping tolerance which typically larger than 0.001 and $\delta \in (0,1)$ can be chosen not too small. Each time we calculate $G_j^{(t)}$ in parallel of elements in a block $\bar{B}$ and then the master thread selects a subset $B \in \bar{B}$ for $d_j^{(t)}$ updating. Note that if the change of $\alpha_j^{(t)}$ is too small, $\boldsymbol{w}^{(t)}$ doesn't need to be updated. This operation is protected by implicit and explicit barriers. Fig. 2 shows a timeline for an example of a sub-problem solved on a worker node.

2) Hot start optimization: We found that in each iteration of the ADMM algorithm, $w_i^{k_i}$ may not change much. Therefore, in the local updating layer, $w_i^{k_i-1}$ can be used as the starting point for updating the $w_i^{k_i}$. The master thread saves the previous α in the shared memory and uses the previous α for updating $w_i^{k_i-1}$. The previous α is reused as α_0. When starting a new iteration, the master thread can read α_0 directly from the shared memory without reinitializing them.

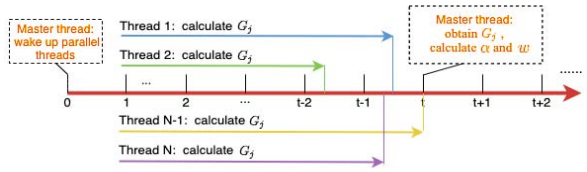

Fig. 2. The timeline for an example of a sub-problem solved on a worker node.

D. Performance Analysis

The implementation of hybrid MPI/OpenMP which shares the model parameters between the OpenMP threads via shared memory. It saves the extra memory footprint caused by the model parameters replication between processes. Furthermore, the sub-problem is calculated by threads instead of processes, which reduces the number of worker processes in the system. Since threads within nodes share the memory, data collection and broadcasting can be done through replication rather than message passing. Therefore, the communication overhead is saved not only between nodes but also within nodes.

We specifically analyze the memory footprint on a worker node and the communication volume in an iteration in the two implementations, as shown in Table 2. For simplicity, the symbols for the variables that are frequently used in the rest of the paper are shown in Table 1.

TABLE I. NOTATIONS

Variable	Description	Variable	Description
l	Number of samples	d	Number of features
N	Number of worker processes	G	Number of nodes
N_{C_i}	Number of cores used by each worker node	N_C	Number of cores used by all nodes
P	Number of processes of each node	T	Number of threads of each node

TABLE II. QUANTITATIVE ANALYSIS OF MEMORY FOOTPRINT AND COMMUNICATION VOLUME IN AN ITERATION OF THE TWO IMPLEMENTATIONS

Implementation	*Memory footprint*	*Communication volume*
Hybrid MPI/OpenMP	$M_H = \frac{m_A}{G-1} + 3m_p d$	$C_H = 2m_p dS$
Pure MPI	$M_P = \frac{m_A}{G-1} + 3m_p dN_{C_i}$	$C_P = 3m_p dSN_{C_i}$

Where m_A denotes the memory footprint of the dataset, m_p denotes the number of bytes of a model parameter, S is the number of updated variables the master needs to wait. The updating of sub-problems takes $O(|B|d)$. We can calculate the difference between the communication volume and memory footprint of the two implementations as $C_P - C_H = (3N_{C_i} - 2)m_p dS$ and $M_P - M_H = (3N_{C_i} - 3)m_p d$. We can see that when $N_{C_i} > 1$, the communication volume and memory footprint of the hybrid parallel implementation will be less than the implementation of pure MPI. In general, the hybrid parallel implementation of the distributed ADMM algorithm can effectively reduce the communication cost and memory footprint, which achieves the reduction of the convergence time and improving the scalability of the distributed ADMM algorithm.

V. EXPERIMENTS

In this section, the corresponding experimental comparisons are performed to evaluate the effectiveness of the AH-ADMM algorithm. Our algorithm is compared with two algorithms implemented by pure MPI, which are the asynchronous ADMM algorithm (AD-ADMM) [6] and the ADMM algorithm based on the hierarchical communication structure (HAD-ADMM) [8]. These two algorithms have been introduced in Section 2. All three algorithms use the dual coordinate descent method to solve the sub-problems.

A. Experimental Environment and Implementation

The algorithms are tested on the cluster supercomputer "Ziqiang 4000" of Shanghai University. We used 9 nodes where each node has two intel E5-2690 CPU (2.9GHZ/8-core) and 64GB RAM. One of the nodes is set as master, and the rest are workers. All algorithms are implemented in C++ using MPICH v3.9.5 and OpenMP v3.0. We consider two large datasets: rcv1 and url. The specific information of the datasets is shown in Table 3. Each dataset is divided into a training dataset and a test dataset according to the ratio of 8:2. Besides,

we randomly shuffle the data to different nodes to ensure that class labels in the data are balanced.

TABLE III. SUMMARY OF THE DATASET.

Dataset	*l*	*d*	*Density*[a]
rcv1	677,399	47,236	0.155%
url	2,396,130	3,231,961	0.004%

[a] Density is the average ratio of nonzero features per sample

We set the penalty parameter $\rho = 1$, the partial barrier $S = 4$, and the bounded delay $\tau = 4$. We use the dual residual r^k and the primal residual s^k as the stop criteria of these three Algorithms. The definitions of r and s are shown in (14). If (15) and (16) are established, the algorithm stops [1].

$$r_i^{k+1} = x_i^{k+1} - z^{k+1}, s^{k+1} = \rho(z^{k+1} - z^k). \tag{14}$$

$$\frac{\left\|r_i^{k+1}\right\|_2}{N} \le ABS\sqrt{d} + REL \times \max\left\{\frac{1}{N}\sum_{i=1}^{N}\left\|x_i^{k+1}\right\|_2, \left\|z^{k+1}\right\|_2\right\}. \tag{15}$$

$$\left\|s^{k+1}\right\|_2 \le ABS \times \sqrt{d} + REL \times \frac{1}{N}\sum_{i=1}^{N}\left\|\rho y_i^{k+1}\right\|_2. \tag{16}$$

Both the absolute error ABS and the relative error REL are set to 0.001.

B. Convergence Test

The training time vs. relative error is used to measure the convergence rate of these three algorithms. The definition of relative error f_{rerr} is shown in (17).

$$f_{rerr} = (f - f_{best})/f_{best}. \tag{17}$$

Where f represents the value of the loss function in the current state and f_{best} represents the minimum value of the loss function obtained from all algorithms.

We tested the three algorithms with $N_C = 65$, $N_C = 33$, and $N_C = 17$. For AD-ADMM and HAD-ADMM, $P = N_{C_i}$ and $T = 1$. For AH-ADMM, $T = N_{C_i}$, and $P = 1$. As shown in Fig. 3, in the case of using the same number of cores, the AH-ADMM algorithm has the fastest convergence rate. This is because the AH-ADMM reduces the number of sub-problems and solves only one sub-problem at each node, which can reduce the synchronization overhead and communication time. Besides, For the dataset of url, the acceleration of the convergence rate is more obvious. Compared with AD-ADMM, the convergence rate of AH-ADMM can increase by 36.36 times when 65 cores are used. However, for the dataset of rcv1, the convergence rate only increases by 1.15-1.48 times. Furthermore, as N_{C_i} increases, the AH-ADMM algorithm often becomes faster. This observation confirms the importance of computing sub-problems in parallel with using the advantages of multi-core.

C. Performance Test

1) Accuracy test: We tested the training time vs. accuracy of the three algorithms to measure the performance. The accuracy AC is defined as the ratio of the number of samples correctly classified by the algorithm to the total number of samples for a given test data set. The specific definition is shown in (18).

$$\mathrm{AC} = \frac{\mathrm{n_{right}}}{\mathrm{n_{total}}}. \tag{18}$$

Where $\mathrm{n_{right}}$ represents the number of samples which are predicted correctly and $\mathrm{n_{total}}$ represents the total number of testing samples.

As can be seen from Fig. 4, the AH-ADMM algorithm is the fastest approach to achieve the best accuracy and will not reduce the accuracy. On the other hand, it can be found that for the AH-ADMM algorithm, the change of $\mathrm{N_{C_i}}$ does not affect the best accuracy.

2) Training time analysis: We define the training time includes computation time and communication time. The computation time includes the optimization time of the sub-problem, the communication time includes the time the master waits to receive s_i, and the time the master sent z to workers. We analyze the training time of the three algorithms when they reach the same accuracy in Fig. 5. We can see that the computation time of the AH-ADMM algorithm is longer than the other two algorithms. This is because the synchronization overhead among threads will slow down the system in updating local variables with OpenMP. For smaller datasets such as rcv1, the effect of this additional overhead is more obvious. On the contrary, the AH-ADMM algorithm has a satisfied performance in the reduction of communication time. For AH-ADMM, the communication cost can be reduced by up to 91.8% when $N_C = 65$ compared with AD-ADMM. In general, the AH-ADMM algorithm can achieve higher accuracy in a shorter time.

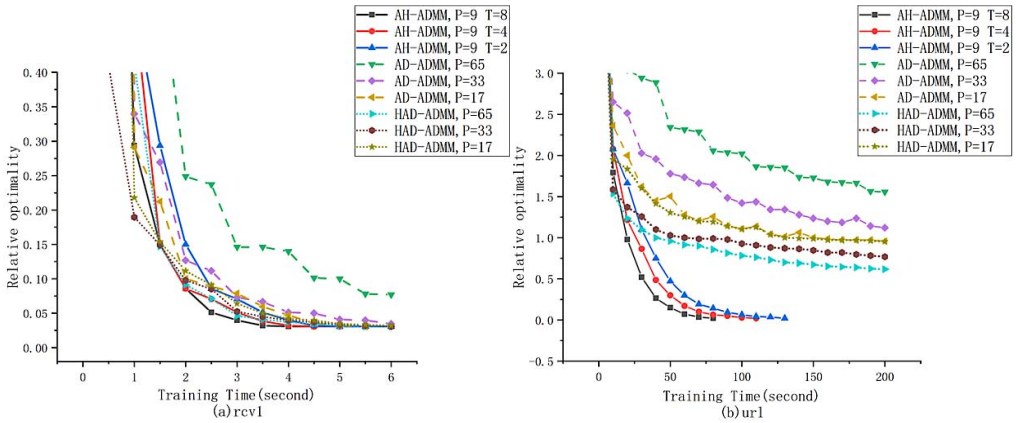

Fig. 3. Convergence comparisons between the three algorithms on the datasets: rcv1(a) and url(b).

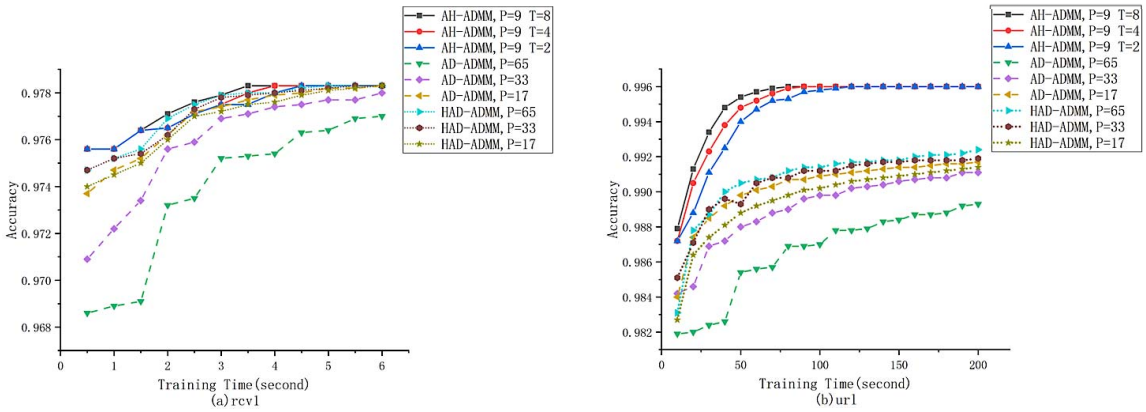

Fig. 4. Performance comparisons between the three algorithms on the datasets: rcv1(a) and url(b).

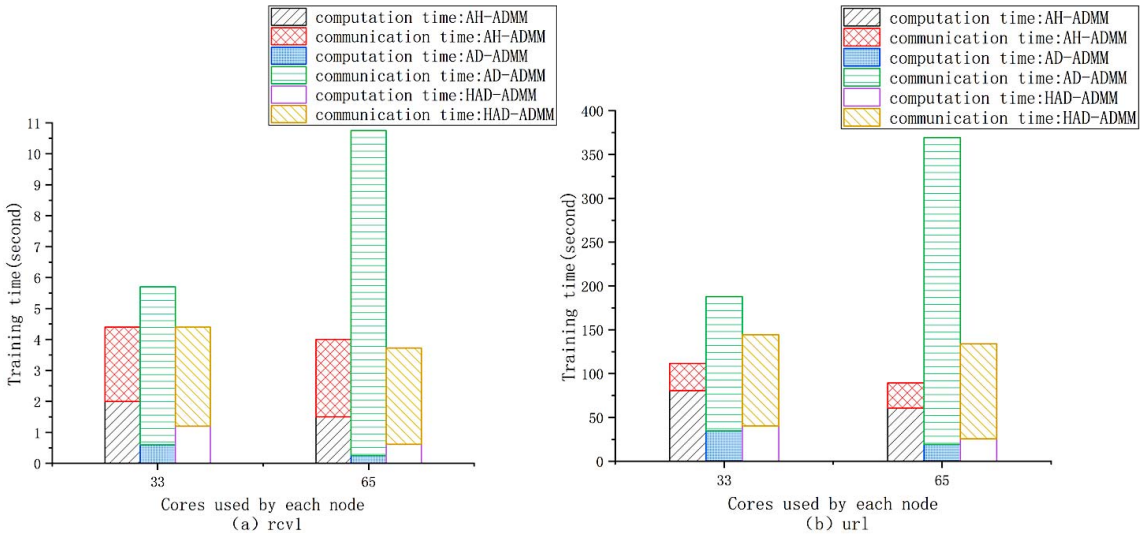

Fig. 5. The training time of the three algorithms on the datasets: rcv1(a) and url(b).

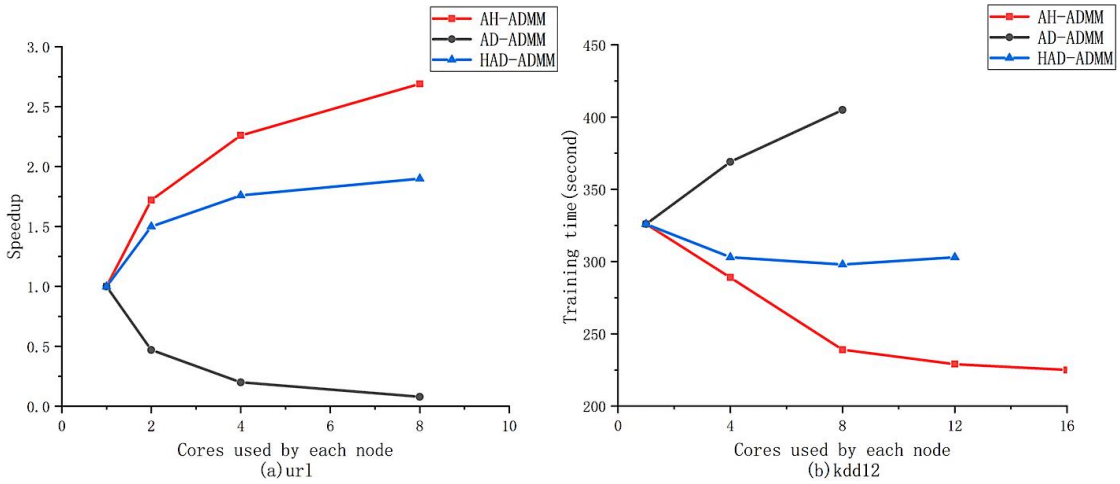

Fig. 6. Scalability of the three algorithms on the dataset: url(a) and kdd12(b).

D. Scalability Test

1) Speedup test: We tested the speedup of the three algorithms to measure the scalability shown in Fig. 6(a). The speedup SP is defined as (19).

$$SP = \frac{t(1)}{t(N_{C_i})}. \tag{19}$$

Where $t(1)$ represents the training time when one core is used by each node, $t(N_{C_i})$ represents the training time when N_{C_i}

cores are used by each node. The training time refers to the system time for the algorithm to reach the best accuracy.

It can be seen from Fig. 6(a) that the speedup of the other two algorithms has increased as N_{C_i} increases in addition to the AD-ADMM. When $N_{C_i} = 8$, the speed of the AH-ADMM can be increased to 2.59 times. However, we also find that the speedup does not increase linearly with N_{C_i} and there is still a large gap from the ideal speedup.

2) The test of the maximum number of cores available on a single node: To verify the reduction in memory overhead and the impact of memory footprint on the scalability, we tested three algorithms for the maximum number of cores available on a node.

We experimented with three algorithms on the big dataset kdd12 of size about 20GB. The number of samples is 149,639,105 and the number of the features is 54,686,452. In this experiment, we tested the training time vs. cores used by each node when reaching the same accuracy. The results are shown in Fig. 6(b). It can be found that when the number of cores used by each node reaches 16, the AD-ADMM and the HAD-ADMM cannot run anymore. This is where memory limitations come into play. Although our hybrid approach can use 16 cores to calculate simultaneously, there is no obvious performance improvement compared to using 12 cores. The synchronization overhead between threads limits the scalability of the hybrid approach also.

E. Summary of Experiments

In this section, we show the test results of convergence rate, performance and scalability of the three algorithms. Compared with the AD-ADMM, the convergence rate of the AH-ADMM can increase by 36.36 times when 65 cores are used. Meanwhile, the AH-ADMM becomes much faster as the number of cores used by each node increases, which proves the effectiveness of using more cores to compute sub-problems by multi-threading. It can be found that the main reason for the decrease in training time is the reduction of communication time which can be up to 91.8%. We measure the scalability of the algorithm by the speedup and the maximum number of cores available on a node. Although the AH-ADMM algorithm has better scalability compared with the other two algorithms, it has not reached our ideal due to the synchronization overhead between threads in solving sub-problems.

VI. CONCLUSION

In this paper, aiming at making full use of modern HPC platforms with multicore nodes, we implement the hybrid MPI/OpenMP parallelization of the distributed asynchronous ADMM algorithm (AH-ADMM). As we know, it is the first attempt at the ADMM algorithm implemented by the hybrid MPI/OpenMP parallel programming model. Experiments show that the AH-ADMM algorithm has the fastest convergence rate and best scalability compared with the AD-ADMM algorithm and the HAD-ADMM algorithm. However, the scalability of the AH-ADMM algorithm still does not achieve ideal efficiency. It may be due to the design of the sub-problem optimization strategy. Therefore, we will continue to study the efficient parallelization approaches of the sub-problem in future work.

ACKNOWLEDGMENT

This research is supported in part by the National Natural Science Foundation of China under grant No.U1811461 and the High-Performance Computing Center of Shanghai University.

REFERENCES

[1] S. Boyd, N. Parikh, E. Chu, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.* vol. 3, no. 1, pp. 1-122, April 2010.

[2] Y. Gu, J. Fan, L. Kong, S. Ma, H. Zou, "ADMM for high-dimensional sparse penalized quantile regression," Technometrics, vol. 60, no. 3, pp. 319-331, 2018.

[3] L. Guan, L. Qiao, D. Li, T. Sun, K. Ge, X. Lu, "An efficient ADMM-based algorithm to nonconvex penalized support vector machines," in IEEE International Conference on Data Mining Workshops, 2018, pp. 1209-1216.

[4] S. Huang, C. Yang, "A hardware-efficient ADMM-based SVM training algorithm for edge computing," unpublished.

[5] L. Guan, T. Sun, L. Qiao, Z. Yang, D. Li, K. Ge, et al, "An efficient parallel and distributed solution to nonconvex penalized linear SVMs," Frontiers of Information Technology & Electronic Engineering, vol. 21, no. 4, pp. 587-603, 2020.

[6] R.Zhang, J.Kwok, "Asynchronous distributed ADMM for consensus optimization," in International conference on machine learning, 2014, pp. 1701-1709.

[7] T. Chang, M. Hong, W. Liao, X. Wang, "Asynchronous distributed ADMM for large-scale optimization—Part I: Algorithm and convergence analysis," IEEE Transactions on Signal Processing, vol. 64, no. 12, pp. 3118-3130, 2016.

[8] S. Wang, Y. Lei, "Fast communication structure for asynchronous distributed ADMM under unbalance process arrival pattern," in International Conference on Artificial Neural Network, 2018, pp. 362-371.

[9] D. Wang, Y. Lei, "Asynchronous Distributed ADMM for Learning with Large-Scale and High-Dimensional Sparse Data Set," in International Conference on Advanced Hybrid Information Processing, 2019, pp. 259-274.

[10] J. Xie, Y. Lei, "ADMMLIB: A library of communication-efficient AD-ADMM for distributed machine learning," in IFIP International Conference on Network and Parallel Computing, 2019, pp. 322-326.

[11] H. Jin, D. Jespersen, P. Mehrotra, R. B, L. H, B. C, "High performance computing using MPI and OpenMP on multi-core parallel systems," Parallel Computing, vol. 37, no. 9, pp. 562-575, 2011.

[12] R. Rabenseifner, G. Hager, G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes" in 17th Euromicro international conference on parallel, distributed and network-based processing, 2009, pp. 427-436.

[13] W. Kwedlo, P. Czochanski, "A Hybrid MPI/OpenMP Parallelization of K-Means Algorithms Accelerated Using the Triangle Inequality," IEEE Access, vol.7, pp. 42280-42297, 2019.

[14] S. Pal, T. Xu, T. Yang, S. Rajasekaran, J. Bi, "Hybrid-DCA: A double asynchronous approach for stochastic dual coordinate ascent," Journal of parallel and distributed computing, vol. 143, pp. 47-66, 2020.

[15] V. Mironov, A. Moskovsky, M. D'Mello, Y.Alexeev, "An efficient MPI/OpenMP parallelization of the Hartree–Fock–Roothaan method for the first generation of Intel® Xeon Phi™ processor architecture," The International Journal of High Performance Computing Applications, vol. 33, no.1, pp. 212-224, 2019.

[16] C. Zhang, H. Lee, K. Shin, "Efficient distributed linear classification algorithms via the alternating direction method of multipliers" in Artificial Intelligence and Statistics, 2012, pp. 1398-1406.

[17] C. Hsieh, K. Chang, C. Lin, S. Sathiya, S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM" in Proceedings of the 25th international conference on Machine learning, 2008, pp. 408-415.

[18] W. Chiang, M. Lee, C. Lin, "Parallel dual coordinate descent method for large-scale linear classification in multi-core environments" in Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 1485-1494.