Fortran Code Refactoring Based on MapReduce Programming Model

Wenhui Gai,Junfeng Zhao*,Han Wu
College of Computer Science,Inner Mongolia University
Hohhot,China
1084302654@qq.com,cszjf@imu.edu.cn,1500141798@qq.com

***Abstract*—Fortran language has been widely used to solve computation-intensive tasks in science and engineering. Due to the emergence of multi-core architecture, the pursuit of Fortran parallelism has become an important goal in the field of scientific computing. Because of insufficient computing resources and poor scalability of multi-core architecture, the Fortran program after multi-core parallel still cannot adapt to the explosive growth of data. It is a meaningful work to automatically map parallelizable Fortran code to Spark platform. A Fortran code automatic refactoring and unloading scheme for Spark cluster is proposed in this paper, which is an extension of OpenMP unloading model. The refactoring is automatically completed by the compiler during the compilation process, and the unloading work is automatically completed by calling the unloading function library during the program running process. The experimental results show that the scheme can automatically map Fortran code running on the local computer to the Spark cluster, and improve the execution efficiency of the original business.**

Keywords-Fortran; MapReduce; Code refactoring; task unloading

I. INTRODUCTION

There are many researches on parallelization of Fortran language, most of which are based on multi-core CPU. The use of co-processors or accelerators is also a widely studied parallel method. CPU and accelerators constitute a heterogeneous accelerated computing system. Compared with the traditional system based on single-machine multi-core CPU, its key advantage is the high performance power consumption ratio achieved by accelerators, which will gradually replace the previous model in many aspects and become the mainstream of the development of parallel technology in the future.

MapReduce has many efficient implementations[1][2], all of which provide application programming interfaces for developers. Although the specific syntax of different APIs is slightly different, they all require developers to encapsulate the logic of computing tasks into map functions and reduce functions. Developers only need to pay attention to the writing of these two functions. The combination of large data center cluster and MapReduce cloud computing programming model provides an opportunity for the cluster to become an accelerator of local programs. Finally, the program performance can be improved through the powerful parallel computing capability of the cluster.

This paper studies MapReduce refactoring of parallel-cycling code in Fortran programs, and proposes an automatic refactoring and unloading scheme of Fortran code for Spark cluster, which makes the cluster become the accelerator of local Fortran programs. The refactoring work is automatically completed by the compiler during the compilation process, and the unloading work is automatically completed by calling the unloading function library during the program running process.

II. RELATED WORK

Fortran language is the best choice for parallel computing[3]. The pursuit of Fortran parallelism has always been one of the important goals in scientific computing. Early developed Fortran parallel programming interfaces include HPF[4] and CoArray Fortran[5]. In recent years, the widely popular parallel programming interfaces are OpenMP[6] and OpenACC[7] based on instruction programming. Several kernel-based programming interfaces, such as CUDA and OpenCL are also available for Fortran parallelization. In order to further improve the development efficiency of parallelization and reduce the errors easily introduced in the refactoring process of manual parallelization, some researchers have proposed algorithms and tools to realize automatic parallelization refactoring[8]. Tinetti F G proposed a parallelization algorithm to convert legacy Fortran serial code into OpenMP parallel code. The algorithm uses advanced algebraic models to describe code conversion and optimization rules, and uses rewriting rule techniques to automatically apply rules in source code. Advanced algebraic models simplify understanding of legacy programs and their transformations, and can support transformations at different levels of abstraction[9]. Other tools automate parallelization by modifying the output of a compilation system or programming framework. AtzeniS et al. modified the object code generation mechanism of the Flang compilation front end to generate CUDA code for the NVPTX CPU-based back end and automatically unloading the computing tasks to the GPU accelerator. The computing advantage of GPU massively parallel architecture is effectively utilized[10].

Existing research has built converters for several SQL declarative languages and integrated MapReduce to support these languages, include Pig Latin/Pig[11-12], SCOPE[13-14], HadoopDB[15], Hive[16], YSart[17], and Jqal[18]. At present, some scholars have proposed some methods and tools for

DOI reference number : 10.18293/SEKE2023-072

refactoring programming language into MapReduce code. Li B proposed a tool J2M (Java-to-MapReduce) that translates Java into MapReduce[19]. This tool is similar to the implementation of an editor, but only compiles target loops with special identification. The object code is generated by extracting some of the necessary information from the source code and combining it with a pre-defined MapReduce template, leaving the rest of the source code unchanged. In order to realize code refactoring for memory cloud computing platform, Li B proposed a translator J2S(Java-to-Spark) that generates MapReduce jobs for Spark platform, which can translate three types of Java source code: for-loop, task, mixed loop and task[20]. Ahmad M B S proposed a conversion tool Casper, which can convert serial Java programs into Spark-MapReduce jobs[21]. In recent years, there are also researches on Fortran language refactoring of MapReduce. Wottrich R proposed OpenMR, a programming model that refactors Fortran, C and C++ source code into MapReduce code[22]. This programming model maps loop iterations to working nodes in a Hadoop cluster based on OpenMP parallel compilation instructions customized in the source code. The compiler can generate map functions and reduce functions required by Hadoop at compile time. While this approach is supported by a set of proof-of-concept, code generation is done manually and there is no comprehensive assessment of the performance overhead of the programming model.

Through the comparison and analysis of relevant research status, it can be found that there are still some problems in the current research. First of all, most existing researches on MapReduce programming model refactoring focus on the transformation from SQL-like queries to MapReduce. The research of refactoring from high-level programming language to MapReduce, most of which focus on object-oriented language such as Java. Secondly, existing research projects require manual participation, and there are still deficiencies in automation. In addition, although the research methods of multi-core parallel refactoring of Fortran programs have been relatively mature in the academia and industry, the refactoring methods of MapReduce programming model are still insufficient, and Fortran programs cannot effectively utilize the powerful parallel computing capability of clusters. The research methods of multi-core parallel refactoring of Fortran programs have been relatively mature, but the refactoring methods of MapReduce programming model are still insufficient. Fortran programs cannot effectively utilize the powerful parallel computing capability of clusters. Fortran, an important language in the field of parallelism, a comprehensive refactoring method is needed to effectively utilize the computing power of clusters. Therefore, this paper studies the refactoring and unloading of Fortran program to MapReduce model.

III. Design Of Refactoring And Unloading Scheme

In this paper, a refactoring and unloading scheme of Fortran code is designed based on OpenMP unloading model of LLVM compilation system. It is oriented to Fortran source code embedded with OpenMP offload instruction and OpenMP parallel loop instruction, aiming to realize circular parallelization of Fortran code in distributed environment. In this scheme, Spark cluster is selected as the target device because it features fast execution, low computing latency, and high data interaction. According to the analysis of OpenMP unloading model, the data flow diagram of this scheme is shown in the figure 1.

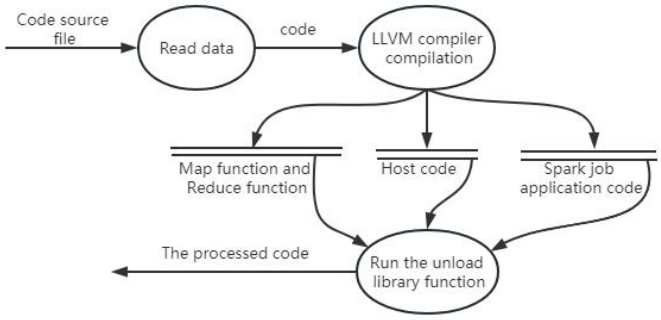


Figure 1. Data flow diagram

First, the OpenMP target instruction and parallel do instruction are inserted into the Fortran serial source code to guide refactoring and unloading. The source code needs to be correct and only handles Fortran programs that run properly on the CPU, not other types of code. The program embedded with the OpenMP instruction is then read by Flang[23], the Fortran front-end compiler of the LLVM compilation system, which constructs an Abstract Syntax Tree (AST) based on the syntax information of the program. After the whole abstract syntax tree is constructed, the static analysis method of traversing abstract syntax tree is selected for semantic analysis. The AST Consumer and RecursiveASTVistor interfaces provided by Flang are used to access the abstract syntax tree, and depth-first traversal is performed on the abstract syntax tree. During the traversal, semantic analysis is completed, including the address stored in the variable, the scope of the variable, the name of the variable, the type of the variable and information about the loop. The OpenMP target node of the abstract syntax tree contains all information about the unloading data and the loop to be refactored. The information in the OpenMP target node is saved in the defined data structure to provide information support for the subsequent automatic refactoring and unloading module.

A. Refactoring Scheme Design

For map function refactoring, firstly build the map function's parameters. The first argument is "index", which indicates the lower index limit of the loop in the map function. The second argument is "bound", which indicates the upper bound of the loop index in the map function; The third and subsequent parameters are variables associated with the map instruction, including the map to clause and map from clause. Parameter values are assigned by the Spark driver node. The body of the map function is then built. First the scheme need to extract the loop body in parallel do exactly as it is, and then body link the loop into a new loop that is a child of the original loop with an index upper bound on the first parameter "index" and a lower bound on the second parameter "bound".

For the refactoring of reduce function, first build the parameters of reduce function. The function parameters are the variables in the two reduction clauses, namely a0 and a1. The parameter values are the partial calculation results of the

variable a returned by the map function, passed by the driver node. Then build the body of the function. The body performs the operator specified in the reduction clause for both arguments and assigns the result to the second argument.

This paper uses a Fortran compiler for code refactoring, so the resulting map and reduce functions are machine code for the Fortran language. To enable Fortran functions to be executed in the Spark cluster environment, this paper uses Java Native Interface (JNI) to encapsulate map functions and reduce functions. This needs to be converted to JNI native functions according to JNI naming conventions. The Spark cluster also needs to generate a Spark application that describes cluster jobs to trigger cluster jobs. Based on the analysis of Spark execution mode, this paper designs the Spark job application template. During refactoring, the specified information is inserted into the template to generate Spark job applications for computing tasks.

B. Unloading Scheme Design

The automatic unloading module relies on the flexible implementation of the OpenMP unloading model.

The data processing part mainly deals with the associated variables of map instruction. The OpenMP unloading model is used to process data mapping of variables associated with map instruction. Meanwhile, in order to reduce the overhead of moving data across the Internet, this paper extends the use of map instruction clause and implements distributed data partitioning. Based on the Spark cluster architecture, this paper designs an unloading function library that can directly interact with the Spark cluster and has the unloading function in the function interface provided by the OpenMP unloading model. Relying on the OpenMP unloading model, the compiler replaces the area of code associated with the target directive with a call to the unloading library. Then run the program to realize the automatic unloading of calculation tasks.

IV. IMPLEMENTATION OF REFACTORING AND UNLOADING SCHEME

Based on the detailed design in Chapter III, this chapter details the implementation of the refactoring and unloading solution to support the refactoring and unloading process. This paper is implemented in the LLVM compilation system. The following introduces the overall implementation method.

To achieve the extensibility of the OpenMP unloading model, the LLVM compilation system breaks the implementation of the model into different components, including a compiler that generates object code, an unloading wrapper library that is independent of the target device, and an unloading plugin that is specific to the target device. This paper extends the LLVM compiler to generate object code for the Spark cluster and the libomptarget runtime unloading library, where the unloading plugin is implemented to unloading computing tasks into the cluster. The implementation name of the defined scheme is FCloud (Fortran-Cloud), which is composed of LLVM compiler, unloading wrapper library, and cluster plugin. The overall implementation is shown in the figure 2.

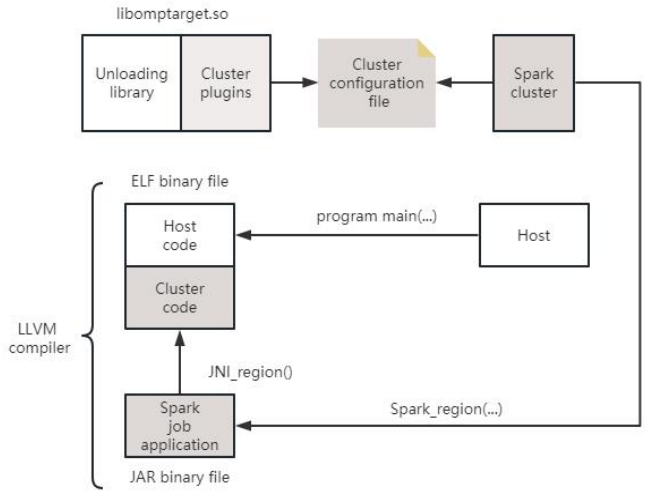


Figure 2. Overall realization of scheme

The LLVM compiler implements the analysis and compilation of the input source program, during which three types of code are generated. The first is the map and reduce functions running on the cluster, which are contained in JNI_region. The second is the code that runs on the mainframe, which is contained in program main, embedded in the same binary in ELF format. The third code is the Spark job application code compiled into a JAR file. When a job is submitted to the cluster, the driver node runs the Spark job application and distributes cyclic tasks among working nodes. Then the working node runs the map and reduce functions written in Fortran language through the JNI interface.

The unloading wrapper library component was designed by Jacob, in the LLVM compilation system to achieve the scalability of OpenMP unloading.The main tasks were to detect the available target devices, create the device data environment, perform the correct unloading function according to the device type, etc. When an unloading area "target" is synthesized, the LLVM compiler generates a set of calls to the unloading wrapper library, regardless of the target device.

Cluster unloading plugin is a component specific to the target device in the libomptarget runtime unloading library. Currently, there are GPUs for GPU and DSPs for DSP. The cluster plugin implemented in this paper can directly interact with the cluster according to the architecture of the cluster, and provide services such as data transmission and triggering the cluster to execute jobs. Because the cluster device is not set up on the local computer, it cannot be automatically detected. Users need to provide identification or authentication information to allow the current application to connect to the cloud service implementation for unloading. The cluster plugin reads the configuration file at runtime to set up the cluster device correctly. Therefore, you need to deploy the cluster and set up the configuration file before running the application. The implementation of LLVM compiler component corresponding information collection module and code refactoring module, unloading packaging library and cluster plugin corresponding unloading module.

V. EXPERIMENTS

A. Experimental Environment and Test Procedure

The local computer is a laptop with 8 GB of memory. The target cluster consists of a private Spark cluster with a driver node and 16 working nodes. Use the libssh API to implement SSH/SFTP communication between the local computer and the target cluster. Table I describes the configuration parameters of each node in the cluster.

TABLE I. SPARK CLUSTER CONFIGURATION

Configuration name	Configuration description
Operating system	Ubuntu14.04
CPU processor	AMD Opteron(TM) Processor 2376 CPU @ 2.3GHz
Memory	16GB
Spark version	2.3.1
JDK version	1.8.0_171
Kernel number	8

In this paper, PolyBench/Fortran suite[24] is used as the benchmark test assembly, from which programs that support the OpenMP target structure and use the typical do loop implementation are selected: SYRK, Mat-mul, 2MM, 3MM, Collinear-list. All data sets used in the experiment are composed of randomly generated single precision numbers, and the matrix used is expanded to 16000 × 16000 (about 1GB). Taking the Mat-mul serial program as an example, the refactoring and unloading process based on FCloud is as follows: First, write the Spark cluster configuration file, and then insert instructions in the Mat-mul serial source code according to the standard description mode defined by OPenMP. During code compilation, FCloud completes Spark-MapReduce refactoring of parallel cyclic code and generates calls to unloading library functions. By running logs to learn about the unloading process and Spark job execution. After the running is complete, the log displays the execution time and other information.

B. Feasibility Analysis

This section verifies whether FCloud can complete the refactoring and unloading by comparing the running results of FCloud and the original serial program. First, use FCloud to test five applications. Table II shows the number of loops contained in the program, the number of nested loops in the program, and the corresponding completion results.

TABLE II. EVALUATION PROGRAMS AND RESULTS

Program	Loop nesting level	Number of cycles	Whether it can be compiled and run normally
SYRK	2	1	Yes
Mat-mul	3	1	Yes
2MM	3	2	Yes
3MM	3	3	Yes
Collinear-list	4	3	Yes

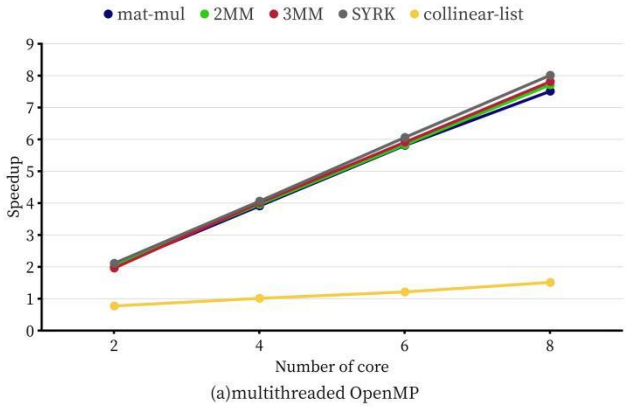
This paper compares the execution results of this scheme and the original serial program in the same data set to verify whether the external behavior of the program has changed, and then to verify whether this scheme has effectively completed the refactoring and unloading. It is mainly to check whether the serial program execution results are consistent with the program execution results of this scheme through traversal, and the verification results are consistent.

C. Performance Analysis

To test the performance of this scheme, this paper compares the clustered distributed parallelism of FCloud with the parallelization method popular in Fortran, OpenMP single-machine multithreaded parallelism. The main way is to compare the run speed of the two programs with that of the serial program. Run speed is the most important indicator to test the performance of the parallel scheme.

Since a single machine only has a maximum of 8 cores, single-machine multithreaded OpenMP experiments were only tested on 2, 4, 6 and 8 threads. As shown in the running results of OpenMP in Figure 3 (a), the acceleration of other programs is close to linear except that Collinear-list program accelerates 1.5 times on 8 inner cores. Figure 3 (b) shows that the overall speed-up of all tested programs on FCloud tends to increase with the number of cores, with 3MM programs getting up to 58 times speed-up at 128 cores. At the same time, it can be seen that even the Collinear-list program, which performs the worst in single-machine multithreaded OpenMP, can achieve 12.5 times speed-up with 128 cores.

Through the above analysis, it can be concluded that this scheme can effectively improve the operation efficiency of the program. Compared with multithreaded OpenMP technology whose parallel capability is limited by the number of single cores, although the acceleration achieved in this scheme is not linear, it will increase with the increase of the number of cores. As the number of cores increases, the 3MM program with the maximum computational complexity gains the largest execution acceleration compared with other tasks with lower computational complexity, indicating that the program with more complex computation can gain greater acceleration through this scheme as the number of cores increases.



(a)multithreaded OpenMP

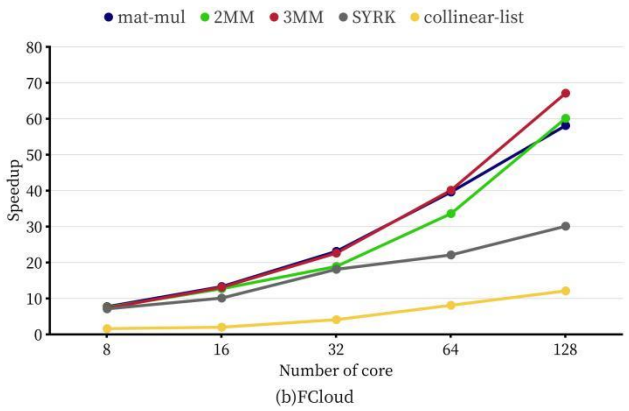
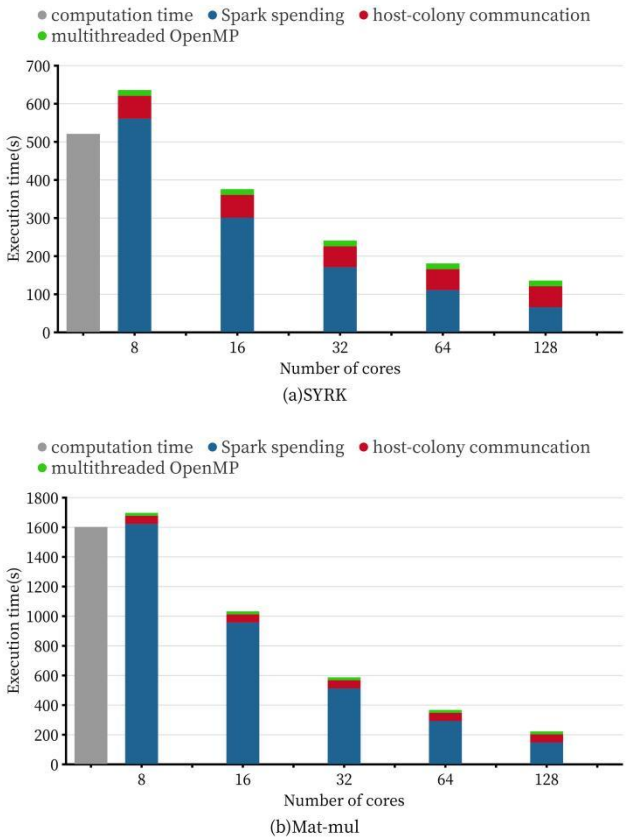


Figure 3. Execution acceleration of multithreaded OpenMP and FCloud

D. *Performance Overhead Analysis*

The results of the experiment show that the acceleration of FCloud does not increase linearly with the increase of the number of cores, and in the case of 8 cores, the running acceleration is lower than that of single-machine multithreaded OpenMP technology, which is caused by the performance overhead generated by FCloud when running the program. This section mainly analysis its performance overhead to verify whether the scheme has a good performance power consumption ratio.

Figure 4 shows the running time of FCloud and multithreaded OpenMP on a single node.



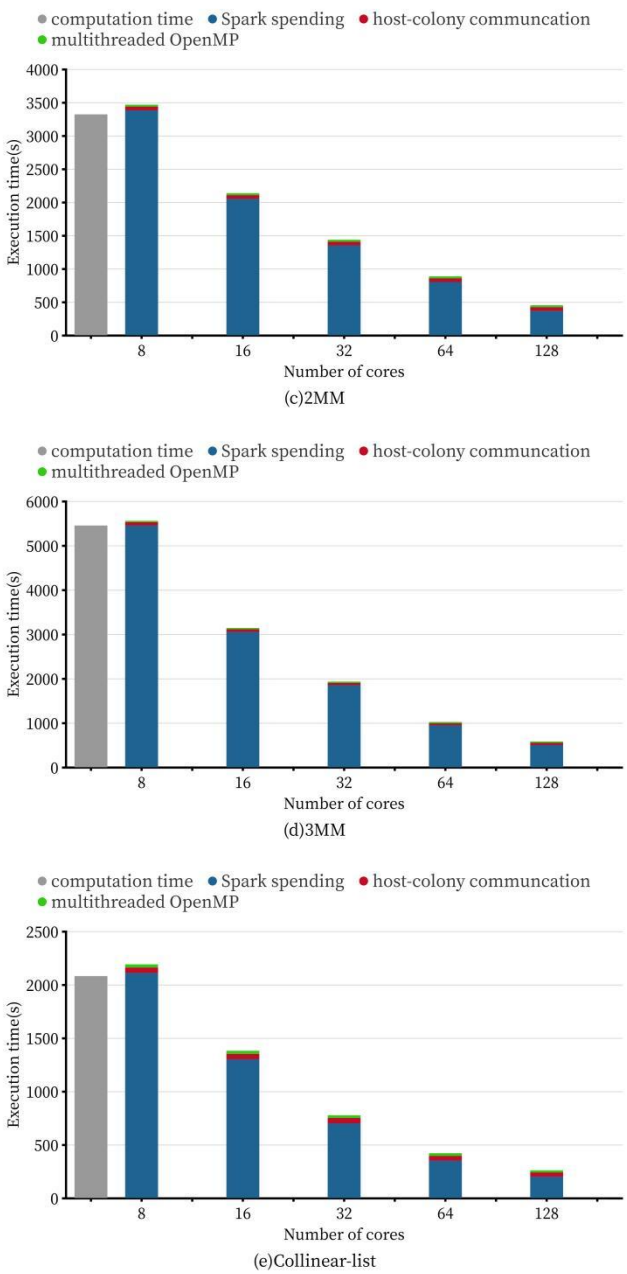


Figure 4. The running time of FCloud and multithreaded OpenMP

As the number of cores increases, the computation time of FCloud decreases, while the communication time and Spark spending remain roughly the same. Comparing the run time of FCloud on 8 cores to the run time of single-machine multithreaded OpenMP also shows a smaller performance overhead. When computing time alone is considered, FCloud has 4.3% more overhead than multithreaded OpenMP, which proves how efficient JNI is at running native functions. When the overhead of Spark clusters is added, FCloud is 6.7% more expensive than multithreaded OpenMP, demonstrating the Spark platform's superior parallelism performance even in the driver node-worker node execution architecture. When the communication overhead between the local computer and the cluster is added, the total running time of the FCloud is 7.6%

higher than that of the single multithreaded OpenMP, which indicates that the overhead of transferring data between the local computer and the Spark cluster is limited in the small number of clusters, even without great computing power.

Based on the above analysis, it can be concluded that the overhead of data transmission and Spark scheduling is stable and limited. Compared with the significantly reduced computing time, the overhead caused by the performance is small, and the scheme has a good performance-power ratio, which further proves the effectiveness of the scheme.

Experimental results show that this scheme can complete the parallel refactoring and unloading of Fortran programs, and has good performance when dealing with large data sets and complex computing applications. Meanwhile, the overhead of scheduling and data transmission within Spark is limited, and JNI runs local functions efficiently, which proves that this solution has a good performance power ratio.

VI. CONCLUSION

In this paper, a Fortran code automatic refactoring and unloading scheme for Spark cluster is proposed. A parallel refactoring method is proposed based on the parallel mode of MapReduce programming model. The OpenMP unloading model is used to automatically uninstall the computing tasks on the local computer to the Spark cluster. The experimental results show that the scheme is feasible, effective and has good performance and power consumption ratio. The application of the refactoring and unloading scheme proposed in this paper can help to improve the efficiency of specific Fortran programs. For future work, it would be interesting to cover more Fortran code and reduce performance costs.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China (61962039).

REFERENCES

[1] Carbone P, Katsifodimos A, Ewen S, et al. Apache flink: Stream and batch processing in a sing le engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4).

[2] Spark A. Apache spark[J]. Retrieved January, 2018, 17(1): 2018.

[3] Rashid Z N, Zebari S R M, Sharif K H, et al. Distributed cloud computing and distributed paral lel computing: A review[C]. 2018 International Conference on Advanced Science and Engineer ing (ICOASE). IEEE, 2018: 167- 172.

[4] Kennedy K, Koelbel C, Zima H. The rise and fall of high performance fortran[J]. Communications of the ACM, 2011, 54(11): 74-82.

[5] Numrich R W, Reid J. Co-Array Fortran for parallel programming[C]. ACM Sigplan Fortran F orum. New York, NY, USA: ACM, 2018, 17(2): 1-31.

[6] Chandra R, Dagum L, Kohr D, et al. Parallel programming in OpenMP[M]. Morgan kaufmann, 2001:12.

[7] OpenACC-Standard.org, "The OpenACC application programming interface version 2.6,"[Online].https://www.openacc.org/sites/default/fifiles/inline-fifiles/OpenACC.2.6.fifinal.pdf. November 2017.

[8] Doroshenko A, Zhereb K. Parallelizing legacy Fortran programs using rewriting rules technique and algebraic program models[C]. International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications. Springer, Berlin, Heidelber g, 2012: 39-59.

[9] Tinetti F G, Méndez M. Fortran legacy software: source code update and possible parallelisation issues[C]. ACM SIGPLAN fortran forum. New York, NY, USA: ACM, 2012, 31(1): 5-22.

[10] Özen G, Atzeni S, Wolfe M, et al. OpenMP GPU offload in Flang and LLVM[C]. 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). IEEE, 2018: 1-9.

[11] Olston C, Reed B, Srivastava U, et al. Pig latin: a not-so-foreign language for data processing [C]. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008: 1099-1110.

[12] Gates A F, Natkovich O, Chopra S, et al. Building a high-level dataflow system on top of Map-Reduce: the Pig experience[J]. Proceedings of the VLDB Endowment, 2009, 2(2): 1414- 1425.

[13] Chaiken R, Jenkins B, Larson P Å, et al. Scope: easy and efficient parallel processing of massive data sets[J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1265- 1276.

[14] Zhou J, Bruno N, Wu M C, et al. SCOPE: parallel databases meet MapReduce[J]. The VLDB Journal, 2012, 21(5): 611-636.

[15] Abouzeid A, Bajda-Pawlikowski K, Abadi D, et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 922-933.

[16] Thusoo A, Sarma J S, Jain N, etc. Hive: A Warehousing Solution Over a Map-Reduce Framework[J]. Proceedings of the VLDB Endowment, 2009, 2(2): 1626- 1629.

[17] Lee R, Luo T, Huai Y, et al. Ysmart: Yet another sql-to-mapreduce translator[C]. 2011 31st International Conference on Distributed Computing Systems. IEEE, 2011: 25-36.

[18] Beyer K S, Ercegovac V, Gemulla R, et al. Jaql: A scripting language for large scale semistructured data analysis[J]. Proceedings of the VLDB Endowment, 2011, 4(12): 1272- 1283

[19] Li B, Zhang J, Yu N, et al. J2M: a Java to MapReduce translator for cloud computing[J]. The J ournal of Supercomputing, 2016, 72(5): 1928-1945.

[20] Li B, Xiao X, Pan Y. Automatic translation from Java to Spark[J]. Concurrency and Computati on: Practice and Experience, 2018, 30(20): e4459.

[21] Ahmad M B S, Cheung A. Automatically leveraging mapreduce frameworks for data-intensive applications[C]. Proceedings of the 2018 International Conference on Management of Data, 2018: 1205- 1220.

[22] Wottrich R, Azevedo R, Araujo G. Cloud-based OpenMP parallelization using a MapReduce runtime[C]. 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing. IEEE, 2014: 334-341.

[23] N. PGI, "Flang compiler," https://github.com/flang-compiler. 2021.

[24] Louis-Noel Pouchet. PolyBench: The polyhedral benchmark suite. http://web.cse.ohio-state.edu /~pouchet.2/ software/polybench/. 2015