# Parallel Computing of Shared Memory Multiprocessors Based on JOMP

ZHANG Hong
College of Electrical & Information Engineering, Lanzhou University of Technology, Lanzhou 730050, China;
College of Computer & Communication, Lanzhou University of Technology, Lanzhou 730050, China.
zhh_2005@163.COM

CAO Jie
College of Computer & Communication, Lanzhou University of Technology, Lanzhou730050, China.
caoj2976016@qq.com

WANG Xiaoming
College of Electrical & Information Engineering, Lanzhou University of Technology, Lanzhou730050, China.
wangxm@lut.cn

ZHU Changsheng
College of Computer & Communication, Lanzhou University of Technology, Lanzhou730050, China.
zhucs@lut.cn

**Abstract—Aiming at parallel computing of shared memory multiprocessors based on JOMP, this paper probes the writing, compiling and running of a *.jomp file in detail and describes a complete specification and process for parallel programming. It expounds the format of parallel directive sets, main runtime libraries and their functions, making a further illustration of variable attributes in parallel region. Taking an example of computing values of pi on a lenovo Intel(R) Core(TM) i3-2120, it obtained a 1.73 speedup factor and 86.5% efficiency, and proved the feasibility and effectiveness of this system.**

***Keywords- parallel computing; JOMP; compiler; speedup; efficiency;***

## I. INTRODUCTION

JOMP(Java OpenMP) is the OpenMP written in Java. OpenMP is a parallel programming industry standard for shared memory or distributed shared memory, which defines a set of directives and library routines for both FORTRAN and C/C++, and enjoys multiple levels of support from both users and vendors in the high computing field. However, existed OpenMP parallel programming was designed for use with Fortran and C/C++, not with Java[1]. The Hadoop platform, a popular platform for processing big data, is developed with java. In order to improve the performance of the Hadoop platform, it is necessary to research parallel extensions to Java[2]. Thus existing forays into parallel programming middle-ware in Java, such as OpenMP, leave much to be desired[3].

As is known to all, we can realize parallel computing using Java's native threads model, and a directive system, which can perform shared memory parallel computing, has a number of advantages. For example, a directive system is considerably easier to use, less errors to make and allows compatibility. The existent approaches to providing parallel extensions for Java mainly include JavaParty, HPJava, Titanium and SPAR Java. However, these are designed principally for distributed systems, and implementations of Titanium and SPAR are via compilation to C, not Java. At the same time, their designs don't involve genuine language extensions,too. J.M.Bull has defined an OpenMP-like interface for Java which enables a high level approach to shared memory parallel programming[4]. Yet, the system still has many outstanding functions, such as data scope attributes and exception handling and so on. But it is not widely used by users. Java 1.7 introduces the fork-join framework and provides developers efficient computing performance[5]. However, it is not transparent to software developers. Thus on the basis of JOMP, we probes the writing, compiling and running of a *.jomp file in detail and describes a complete specification and process for parallel programming. This paper defines the data scope attributes and extends its application. Experiments have proved that it is feasible and efficient.

## II. THE IMPLEMENTATION OF JOMP

Based on the existed OpenMP standard for C/C++, the implement of JOMP is specified as follows.

### *A. The Format of JOMP Directives*

OpenMP is a collection of compiler directives and library functions, which is primarily used for parallel programming on shared memory computers[6]. Since the Java has no standard form for compiling specific directives, JOMP adopts what is used in the OpenMP Fortran specification and embeds the directives as comments[7], which is shown in Fig .1. Standard parallel mode to execute Java code has one main thread, which executes the serial part of the program until encounters a parallel directive as comments. It will create a new thread team if the Boolean expression in the if clause is true or there are not if clauses at all. Each thread in the new thread team immediately executes the following java code

block in parallel. When all threads in thread team finish their tasks, the main thread continues executing serial java code block.

```
//omp parallel [if(<cond>)]
               [default (shared|none)]
               [shared(<vars>)]
               [private(<vars>)]
               [firstprivate(<vars>)]
               [reduction(<operation>:<vars>)]
<Java code block>
```

Figure 1. The parallel directive format

In Fig .1, the default, shared, private, firstprivate and reduction clauses work in the same way as the C/C++.

The default clause is used to allow users to control the shared attribute of a variable in the parallel region. When the symbol of share is used, it is a default situation which indicates that the same named variable passed into a parallel region will be treated as a shared variable and will not produce a private copy of a thread. When none symbol is used, the variable used in a thread must be shared or private.

The share clause is used to declare a variable as a shared variable.

The private clause is used to declare a private variable, in which each thread has its own private copy and other threads can not access the private copy.

The firstprivate clause is used to inherit the values of originally shared variables, hence private variable can not inherit the values of the same named variables.

TABLE I. OPERATORS AND INITIAL VALUES OF RELATED PRIVATE COPY

| Operator | Initialization Value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

The reduction clause is used to specify an operator for parameters. Each thread creates a private copy of the parameter. At the end of the parallel region, the value of the private copy will be operated by the specified operator and original values of the parameters will be updated by the values of calculated results.Operators and the initial values of related private copy are shown in table 1.

### *B. Main Classes of Runtime Library*

Runtime library is used to provide the necessary functionality supporting parallelism in terms of Java's native threads model[8]. Runtime library is organized into the package jomp.runtime and contains a library of classes and routines used by compiler-generated code.The core of the library is the OMP class, which contains the routines used by the compiler to implement parallelism in terms of Java's native threads model. In Table 2, the commonly static members of class jomp.runtime.OMP are described. The other main classes and their functions are shown in Table 3.

TABLE II. MAIN STATIC METHOD MEMBERS OF CLASS JOMP.RUNTIME.OMP

| Class Name | Functions |
|---|---|
| BusyThread | manage threads |
| BusyTask | |
| Barrier | implements a barrier and manage internal thread to avoid the overhead of a system call |
| Orderer | facilitate implementation of the ordered construct of variables reductions |
| Reduction | implement the reduction clause |
| Ticketer | facilitate dynamic allocation of work to different threads |
| LoopData | store information about a loop or a chunk of a loop |

TABLE III. OTHER CLASSES AND FUNCTIONS

| Method Name | Functions |
|---|---|
| getNumThreads() | return the number of threads executing the current parallel region |
| setNumThreads(n) | set the number of threads to execute parallel regions to n |
| getMaxThreads() | return the maximum number of threads used to execute a parallel region |
| getThreadNum() | return the number of the calling thread |
| getNumProcs() | returns the maximum number of processors that could be assigned to the program |

### C. *the JOMP Compiler for Parallel Directives*

The JOMP Compiler is built based on a Java 1.1 parser provided as an example with the JavaCC. OMPVisitor class is used to implement the bulk of the compiling works[9].

When encountering a parallel directive, such as Fig .2, the compiler creates a new inner class automatically[10], showed in Fig .3. In this inner class, main variables are described as follows. It contains a field of same type signature and name for shared variables, a field of same type signature and named __omp_fptemp_ <varname> for firstprivate variables, a field of same type signature and named __omp_lptemp_<varname> for variables with a reduction operation specially. The single method of this inner class is the go( ) method, which declares a local variable with the same name and type signature for private or firstprivate variables.

In place of the parallel construct itself, the code is inserted to the parallel region to declare a new instance of the inner class and to initiate the fields within it from the appropriate local variables, showed in Fig .4.

```
...
//omp parallel shared(n) private(tid,i,k) reduction(+:s)
        {
          The code to be executed in parallel
        }
...
```

Figure 2. Code with JOMP parallel directives

```
__omp_Class0  __omp_Object0 = new  __omp_Class0();
__omp_Object0.n = n;// shared variables
__omp_Object0.args = args; // shared variables // shared variables
  try {
     jomp.runtime.OMP.doParallel(__omp_Object0);
  } catch(Throwable  __omp_exception) {
     System.err.println("OMP Warning: Illegal thread exception ignored!");
     System.err.println(__omp_exception);
  }
  s +=__omp_Object0._rd_s;   // reduction variables
  n =  __omp_Object0.n;   // shared variables
  args = __omp_Object0.args;   // shared variables
```

Figure 3. The inner class

```
private static class __omp_Class0 extends jomp.runtime.BusyTask {
    double n;   // shared variables
    String [ ] args;   // shared variables
    double _rd_s;   // variables to hold results of reduction
    public void go(int __omp_me) throws Throwable {
    int tid;     // private variables
    int i;   // private variables
    double k;   // private variables
    double s = 0;     // reduction variables, init to default
            {
                The code to be executed in parallel
            }
    s = (double) jomp.runtime.OMP.doPlusReduce(__omp_me, s);     // call reducer
    if (jomp.runtime.OMP.getThreadNum(__omp_me) == 0) {
      _rd_s = s;   }
  }
}
```

Figure 4. Instance of inner class and initialization

## III. Application Mode and Examples

### A. *Application Setting*

Before applying JOMP Compiler and runtime library and running java parallel programs, it is a must to configure the system environment variable. The JOMP jar package , which is mainly composed by JOMP Compiler and runtime library, is under the JDK lib path. Then

append JOMP jar package to CLASSPATH, a system environment variable, such as %JAVA_HOME%\lib\jomp.jar[11].

Next, we can begin to write java programs executed in parallel, for example TestJavaOMP.jomp. The extension name for this java program must be jomp. So JOMP Compiler is called and *.jomp file is compiled into *.java file. With the example of TestJavaOMP.jomp, the compiling directive is in the form of java jomp.compiler.Jomp TestJavaOMP. After that, a java file will be obtained, where the compiling and running is not to repeat. The process is shown in Fig .5.

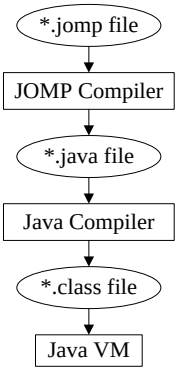


Figure 5. The process of jomp file compiling

### B. An Example

JOMP has been applied to compute the value of Pi, of which the formula is described in formula (1).

$$\pi = 4 * \sum_{n=1}^{\infty} (-1)^{n+1} * \frac{1}{2 * n - 1} \quad (1)$$

In above formula, $\pi$ is the value of pi and n is the natural number. The running result is showed in table 4. Fig .6 shows the changes of running time with the numbers of threads.

In Fig .6, it can be seen that the performance has been greatly improved after using parallel computing based on JOMP. When tested on a lenovo Intel(R) Core(TM) i3-2120, the original serial code took 348.6 milliseconds, the parallel code with two threads on two processors took 215.8 milliseconds, and the parallel code with four threads on two processors only took 201.6 milliseconds. This represents a speedup factor of 1.73, and an efficiency of 86.5%. But it is not always the case. When six threads was used to test, it took 257 milliseconds and when eight threads, average time is also about 250 milliseconds. With increasing of threads, additional cost will increase too. More threads means that more cost need to access the critical section. So it is the key issue to determine a large speedup factor by the experiments.

TABLE IV. RESULT OF COMPUTING PI ON INTEL(R) CORE(TM) I3-2120

| No. Of Thread | No. Of Running(ms) | | | | | | | | Average Time(ms) | Pi Value |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | |
| 1 | 351 | 348 | 350 | 348 | 351 | 347 | 347 | 347 | 348.6 | 3.1415926335902506 |
| 2 | 224 | 220 | 206 | 220 | 222 | 212 | 201 | 210 | 215.8 | 3.1415926335840005 |
| 4 | 205 | 202 | 203 | 200 | 204 | 201 | 199 | 199 | 201.6 | 3.141592633596291 |
| 6 | 262 | 261 | 260 | 256 | 254 | 255 | 254 | 254 | 257 | 3.141592633593593 |

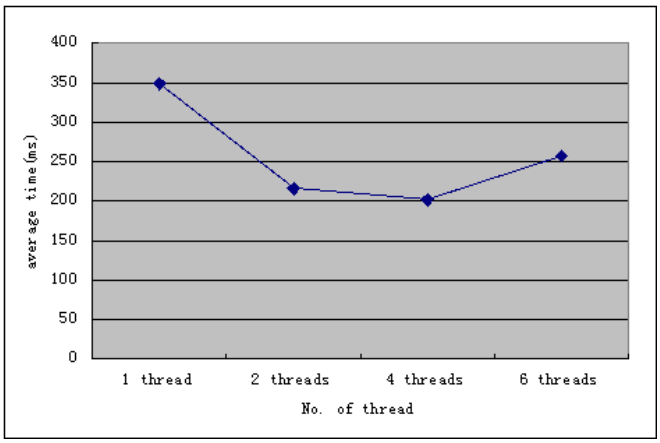


Figure 6. Changes of running time with the numbers of threads

## IV. CONCLUSIONS

Similar to the OpenMP C/C++ specification, JOMP defines an OpenMP programming interface for Java which mainly includes two parts: the compiler and the runtime library. This paper probes the parallel programming based on JOMP, expounds the format of parallel directive sets, and clearly defines the variable attributes. The compiling and running of a *.jomp file is parsed in detail and a complete specification is described. This makes subsequent implement and maintenance of parallel programming more easier, and makes the running performance more better. With the example of computing

values of pi on a lenovo Intel(R) Core(TM) i3-2120, a 1.73 speedup factor and 86.5% efficiency were obtained, which proved the feasibility and effectiveness of the system.with the increasing extend to implement the whole specification of JOMP, it will provide parallel programming in Java a more attractive proposition and support for hybrid-parallel programming on clusters[12].

## ACKNOWLEDGMENT

Project supported by the Natural Science Foundation of Gansu Province (No. 2014GS03435)

## REFERENCES

[1] J. M. Bull, L. A. Smith, L. Pottage, et al. Benchmarking Java against C and Fortran for scientic applications[C]. Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, 2001,97-105.

[2] Chuck Lam. Hadoop in Action[M].POST & TELECOM PRESS,2011.

[3] Laurent Amsaleg,Distributed High-Dimensional Index Creation using Hadoop HDFS and C++,Content-Based Multimedia Indexing (CBMI),2012,253-261.

[4] J. M. Bull,M. E. Kambites.JOMP-an OpenMP-like Interface for Java[C].In ACM 2000 Java Grande Conference,2000,1-10.

[5] N. Giacaman, O. Sinnen. Object-oriented parallelisation of Java desktop programs[J]. IEEE Software, Software for the Multiprocessor Desktop: Applications, Environments, Platforms, 2011,28(1):32-38.

[6] M. Klemm, M. Bezold, R. Veldema, et al. JaMP: an implementation of OpenMP for a Java DSM[J]. Concurrency & Computation:Practice & Experience, 2007,19(18):2333-2352.

[7] M. Klemm, R. Veldema, M. Bezold, et al. A Proposal for OpenMP for Java[C]. In Proceedings of IWOMP. 2006,409-421.

[8] N. Giacaman,O. Sinnen. Pyjama: OpenMP-like implementation for Java, with GUI extensions[C]. International Workshop on Programming Models and Applications for Multicores and Manycores,2013, 43-52.

[9] A. Senghor, K. Konate.A Java hybrid compiler for shared memory parallel Programming[C].13th International Conference on Parallel and Distributed Computing, Applications and Technologies,2012,131-136.

[10] S. Lee, S.-J. Min, R. Eigenmann. OpenMP to GPGPU:a compiler framework for automatic translation and optimization[C]. in Proceedings of 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2009, 101-110.

[11] N. Giacaman, O. Sinnen. Parallel task for Parallelising object-oriented desktop applications[J]. International Journal of Parallel Programming, 2013, 41(5):621-681.

[12] S. Benkner and V. Sipková. Language and Compiler Support for Hybrid-Parallel Programming on SMP Clusters[C]. in Proceedings of the 4th International Symposium on High Performance Computing,2002,11-24.