Design and Implementation of Multi-core Parallel Compiler Based on OpenMP

Xiaojian Liu*, Hao Lv, Shuo Liu
Computing Technique Research Institute, Aviation Industry Corporation, Xi’an, PR, China

*Corresponding author: lxjiansky@163.com

Abstract. OpenMP supports parallel incremental development, and has become a mainstream parallel programming standard for shared memory systems. A parallel compiler running on multi-core DSP is designed for OpenMP programs. The main achievement includes translator and runtime. The translator converts OpenMP instructions in source files into function calls in runtime where the specific implementation is provided. The core problem of parallel compiler is how to design parallel strategy to allocate computing tasks to each core, which corresponds to the concept of parallel domain in OpenMP standard. The compiler realizes the master-slave core by transforming parallel instructions and supporting runtime, which has guiding significance for the design of parallel compiler.

Keywords: OpenMP, parallel compiler, multi-core DSP, parallel execution

1. Introduction
In the face of the increasing demand for computing, the improvement of clock frequency of single-core processor has reached the bottleneck. Processor manufacturers compensate for the performance impact of reducing clock frequency to reduce power consumption by increasing the number of processor cores, which makes multi-core processor show obvious computing advantages compared with single-core processor. With more applications needed to be developed on multi-core platform, parallel programming mode is gradually favored by programmers. The concept of parallelism has been embodied in the way of threads in operating systems. However, the thread library represented by Pthread is more inclined to task parallelism, and requires programmers to spend more energy on the underlying operations related to threads. In order to make programmers focus more on business process logic, some easy-to-use parallel programming specifications have been proposed at present, such as OpenMP based on shared memory, MPI based on message passing mode and so on [1]. In order to improve the efficiency of parallel programming on multi-core DSP, this paper introduces in detail the compiler's scheduling and task allocation for multi-core DSP in parallel domain, and explains how to implement parallel computing in fork-join mode on multi-core DSP platform.
2. OpenMP standard and parallel programming model

2.1. Introduction to OpenMP Standard
OpenMP standard provides a set of compiling instructions and application programming interface based on shared memory model, which is independent of hardware platform. It can be added to the program written in C/C++ and FORTRAN to show the parallel mode of a serial program. It uses multithreading technology and uses fork-join mode to realize parallelism. Because of its good portability, easy to use and supporting parallel incremental development, it has become the mainstream standard in shared memory models [2].

OpenMP includes four types of instructions and clauses: parallel domain creation, task sharing, synchronization and mutex, data environment control. The application program interface is used to set or return the relevant information in the parallel domain [3]. The parallel domain creates an instruction and declares a parallel domain. The statements in the code block will be completed in parallel by multiple threads. The task sharing instruction indicates that multiple cores jointly complete tasks in the parallel domain, which can be iterative task for loop or non-iterative task blocks. Synchronization and mutex instructions are used to coordinate the timing constraints of multi-core in the execution process, and ensure the correct reading and writing of shared resources. The data environment control clause is used to indicate the shared attributes of variables in the parallel domain.

2.2. OpenMP parallel programming model
The following is a parallel program written by adding OpenMP instructions on the basis of the standard C language. The purpose is to assign an array. The 100 loops in the for statement are allocated to be executed by different threads. The statement block under #pragma omp parallel instruction is a parallel domain and is executed by multiple threads. OpenMP standard API was called to set the number of threads to 8. #pragma omp for instruction means that eight threads work together to complete the subsequent for loop. Array a is shared by threads, so the assignment of array members by each thread is visible to each other. Outside the parallel domain, the whole program is executed by a single thread [4].

```c
void main(){
    int i, a[100];
    #pragma omp parallel
    { 
        omp_set_num_threads(8);
        #pragma omp for
        for(i=0;i<100;i++)
            a[i]=i;
    }
}
```

OpenMP implements multithreading parallelism based on fork-join model. Fig.1 shows the model diagram of fork-join. After encountering the parallel domain creation instruction in OpenMP standard, the main thread will generate multiple slave threads (i.e. fork operation) before the code execution in the parallel domain. The main and slave threads are allocated by the system on different processors to execute the code in the parallel domain in parallel, and execute the join operation at the end of the parallel domain. When a parallel domain is generated again, repeat the above steps.

![Figure 1. OpenMP execution model](image-url)
3. Parallel compiler design and translation transformation

3.1. Introduction to Hardware Platform
The parallel compiler implemented in this paper is based on TI's TMS320C6678 DSP (C6678) hardware platform. Based on keystone I architecture, this chip integrates eight processing cores with the same structure. Each core can read and execute instructions independently, with the main frequency up to 1.25GHz. Each core has 32KB L1P cache, 32KB L1D cache and 512KB local L2 cache. In addition, all cores can share 4MB multi-core shared memory which can store program or data [5].

3.2. Multi-Core Parallel Scheme
In order to implement the fork-join model on multi-core platform, the master-slave core running scheme is designed as shown in Fig.2. Once the program starts running, both the master and slave cores start, and execute different processes. The master core executes the serial code in OpenMP program, and the slave core waits for the master core to send tasks. When the master core enters the parallel domain, it generates multiple task threads similar to fork operation, and then sends the task to each slave core one by one. Slave cores start to execute after receiving the task and once the task is completed, they need to give feedback to the master core immediately. After the tasks are assigned to the slave cores, the master core starts to execute its own task, and then waits for the messages from the slave cores. It does not exit the parallel domain (equivalent to join operation) and continue to execute the serial code until all slave cores have completed the task. The slave core enters the wait state again after the task is completed. If it needs to enter the parallel domain again, the above process is repeated.

3.3. Compilation Process and Transformation Operation
The whole compiling system is composed of translator, runtime and local compiler. The main work of this paper is the design of translator and runtime. The processing flow of OpenMP program is shown in Fig.3. The input is OpenMP program (source 1), which is a .c file containing OpenMP instructions. The interpreter translates the OpenMP instructions in the source program into library functions implemented in the runtime, and generates a .c file containing the runtime interface (source 2). The local compiler uses the compiler cl6x provided by Ti to compile source 2, and finally generates the executable file running on c6678 DSP combined with the runtime link.

![Figure 2. Parallel mode of master-slave cores](image-url)
The translator includes two steps: analysis and transformation. Source 1 generates the intermediate code in the form of abstract syntax tree (AST) after lexical, grammatical and semantic analysis, and then transforms AST to output source 2. In this process, we use Lex and Yacc tools to complete lexical and grammatical analysis, and complete the construction of AST in the process of executing semantic actions.

Fig. 4 shows the AST corresponding to a simple OpenMP program. Each box represents an AST node and different types of AST nodes are designed according to different types of statements in the program (such as variable declaration, function definition, expression, etc.). The information recorded in each node is different. For example, in Fig. 4, the AST node of function definition records the function name (decl member) and function body (body member). Function call belongs to the class of expression, and the corresponding AST node records the left value (member left) and right value (right member) of expression. OpenMP instruction corresponds to a type of node, which will be dealt with in the transformation step. These nodes are connected and organized into a tree, so the whole program can be represented by an AST node at the top.

After the AST is established according to source 1, the transformation is started. In the process of traversing the whole AST the transformation is completed, but most of the AST nodes corresponding to the standard C language statements do not need to be transformed. Only the OpenMP nodes need to be processed. The specific operation is to crop the corresponding subtree of OpenMP node and replace it with the AST subtree of the expected changed code. The corresponding Fig. 4 shows that the OpenMP node in the dashed box is removed from the original AST and replaced with a new subtree. At the same time, the new subtree must be completely embedded without changing other structures in the original AST. After the new AST is constructed, the code program in the form of output characters is traversed again, i.e., generating source 2.
This section will introduce the code form of OpenMP instruction in source 1 after translator transformation. Firstly, #pragma omp parallel instruction in source 2 is replaced by a parallel domain to create function interface. Secondly, the code block (parallel statement) under the instruction is extracted and packaged into new task functions. Finally, the data environment is constructed according to the shared attributes of variables in the parallel domain declared in source 1.

Further explanation is given combined with the translation result of an OpenMP program. An array variable z is defined outside the parallel domain in the following source 1 program, and the value of the element whose subscript is the thread number of the executing thread is output in the parallel domain. According to the OpenMP standard, z is shared data in a parallel domain. The main function in source 1 corresponds to two parts in source 2 program after translation: the original function and the task function in the parallel domain. The main function of source 2 includes the serial statement in source 1, the information encapsulation of shared variables and the function interface of parallel domain creation. The newly generated task function _thrFunc0_ includes parallel statements and data environment preparation before parallel statements. The parallel statements in this example are relatively simple. If there are other OpenMP instructions in the parallel domain code in source 1, the translator will replace them with corresponding functions in the runtime in source 2.

```c
/******* Source 1 program *******
int main()
{
    int z[5] = {1,2,3,4,5};
    printf("serial\n");
    #pragma omp parallel num_threads(5)
    {
        int j =omp_get_thread_num();
        printf("%d",z[j]);
    }
}
/******* Source 1 program *******
/******* Source 2 program *******
int main()
{
    int z[5] = {1,2,3,4,5};
    printf("serial\n");
    /*Record the information of shared variables in shvars*/
    struct_shvt{
        int(*z)[5]
    }shvars = {&z};
    execute_parallel(5,_thrFunc0_,shvars);
}
void _thrFunc0_(void* me)
{
    /*Record the information of shared variables in shvars*/
    struct_shvt{
        int(*z)[5];
    }shvars;
    shvars = get_shared_vars(me);
    int(*z)[5] = shvars->z;
    {
        int j =omp_get_thread_num();
        printf(" %d\n",(*z)[j]);
    }
    return;
}
/******* Source 2 program *******
```
Parallel domain creation function execute_parallel is only called by the master core, and its specific implementation is provided by the runtime. The function has three parameters, which are the number of threads participating in the execution in the parallel domain (in this case, In this case, it is declared by num_threads), task function name (in this case, it is _thrFunc0_) and the structure that records shared variable information (in this case, it is shvars). Since the shared variables in the parallel domain are uncertain with the program definition, the translator encapsulates all the shared variables in a parallel domain into a structure to facilitate the transfer in the form of parameters.

A parallel domain corresponds to a task function, which is called by every core that enters the parallel domain. In the task function, the data environment declared in source 1 needs to be reconstructed before the parallel statement in the task function. It is realized by redefining and initializing the shared variables with the same name and private ones. As can be seen from the above example, the structure used to record the shared variable information before calling the execute_parallel function is exactly the same as that used in the task function to build the data environment. In fact, the structure obtained by get_shared_vars function is the parameter variable passed in execute_parallel before.

The reason why we need to create shared variables in parallel domain and transfer the information of shared variables between task functions is that the execution context of each core is stored in its own local memory. If a variable is declared to be a private variable in the parallel domain, no additional processing is needed. However, for shared variables, in order to make all cores in the parallel domain access symmetrically, the variable should be declared in the shared area and its address should be recorded by the master core and sent to the slave cores. It can be accessed indirectly through the address in the parallel statements of task functions.

4. Runtime implementation
This chapter introduces the runtime support for the above parallel scheme, including the key data structure, the implementation of the parallel domain creation function generated in source 2 after the parallel instruction transformation and the execution flow of the slave cores in the parallel domain.

4.1. Data Structure of Simulating Parallel Domain In Runtime
The processing of parallel domain in runtime revolves around two data structures which are the execution structure used to record the state of each core in the parallel domain and the thread structure that the master core distributes tasks to the slave cores.

- **Executive structure**
  In the runtime, each core will create its own independent executive body after entering the parallel domain, which records its own information in the parallel domain such as its own number in the parallel domain, the total number of cores entering the same parallel domain and so on. The API in OpenMP standard can return the total number of threads in the current parallel domain, the thread number of the current thread, etc., all of which are implemented by querying the values of related members in the executive body of each core.

  The executive structure maintains the parent-child relationship between the master and slave cores in the same parallel domain. After the master core creates the parallel domain, its executive body will become the parent node of slave cores’ executive node in the parallel domain. The parent executive node records the shared information in the parallel domain, such as task functions, shared variables, etc. which can be obtained by the child node through its parent node.

- **Thread structure**
  Thread is the actual bearing structure that the master core sending tasks to the slave cores, which encapsulates the pointer pointing to the parent executive body recording the information of the parallel domain and their respective thread numbers. The relationship between the executive body and the thread is shown in Fig.5. Threads 1, 2 and 3 are the tasks sent from the master core to the three slave cores, and the pointers in the threads in the same parallel domain point to the same parent executive.
Therefore, the slave cores receiving the threads will access the same task function and the threads are distinguished by numbers.

The life cycle of a thread only exists in the parallel domain. In order to avoid the computational overhead caused by frequently creating and destroying threads, a group of threads are created in the form of a linked list in runtime to form a thread pool. Each time a parallel domain is created, the master core applies for a corresponding number of threads from the thread pool and sends them to the slave cores. After the slave cores complete the task, the threads received from the master core are put back into the thread pool for use in the next parallel domain.

The task completion status of the slave cores in the parallel domain is monitored by the master core through the executive body and thread structure. As shown in Fig.5, a key variable is recorded in the parent executive body - "the number of slave cores that have not completed tasks in the parallel domain". When the master core creates a parallel domain, the initial value of this variable will be set to the number of threads in the parallel domain. When the master core sends the thread to the slave core and the slave core completes the task, it will access the variable through the parent executive body and subtract 1 from its current value.

Therefore, the variable is shared by multiple cores and its value will change with the execution of the slave cores. The master core can judge whether there are slave cores that have not completed the task the current parallel domain by the value of the variable.

![Diagram of the relationship between executive body and threads](image)

**Figure 5.** Diagram of the relationship between executive body and threads

### 4.2. Implementation of Parallel Domain Creation Function

The parallel domain creation function is called by the master core to distribute tasks to the slave cores, execute tasks by itself and synchronize before the end of the parallel domain. Its implementation process is as follows.

1) According to the number of cores in the parallel domain specified in the source program, apply for the same number of threads from the thread pool of the runtime. If the number of threads available is not enough, it will be terminated and a prompt message will be returned.

2) Set the information of the whole parallel domain and fill in the parameters in the parent executive body node representing the parallel domain, including task function, shared variables, cores entering the parallel domain and so on.

3) Send task to the slave cores: fill in the information to be sent in the application threads, mainly including the pointer to the parent executive body node and the thread number shown in Fig.5. After setting, as shown in Fig.6, the threads will be sent to the corresponding slave cores in turn.
4) Execute the task function to complete the tasks that belong to in the parallel domain.
5) Synchronous exit, wait for all the slave cores in the parallel domain to complete the tasks and then exit the parallel domain. The specific implementation is that after the main core enters the loop, it does not jump out of the loop until the value of the variable "the number of slave cores in parallel domain that has not completed tasks" in the executive body is 0.

4.3. Execution Process of Slave Core in Parallel Domain
In the parallel domain creation function, the master core sends tasks to the slave cores in the thread structure and after receiving the task, the slave cores execute three steps: entering the parallel domain, calling the task function and exiting the parallel domain. These three steps are all related to the parent executive body encapsulated in the received thread.

1) Enter the parallel domain: Establish the parent-child relationship with the master core executive body node and fill in the information in the executive body structure according to the parent executive body.
2) Call task function: By accessing the task function pointer of the parallel domain recorded in the parent executive body, the task function generated by the translator is executed.
3) Exit the parallel domain: Put the thread sent by the master core back to the thread pool and subtract the value of variable "the number of slave cores that have not completed tasks in the parallel domain" in the parent executive body by 1, which is equivalent to feedback the message that the tasks have been completed to the master core.

5. Conclusion
This paper designs and implements a parallel compiler for OpenMP programs running on multi-core DSP platform. By translating the compilation guidance instructions in the source program into the runtime interface, the local compiler and runtime are used to compile and link the translated C program. Running on multi-core DSP, the generated executable file can achieve the effect of master-slave cores executing code in parallel domain in parallel. While making full use of the computing resources of multi-core DSP, it improves the programming efficiency of programmers in developing parallel programs. In the following work, the following work can focus on the performance optimization of compiler, such as quantitative analysis of the computing tasks assigned by each core in the parallel domain. A more reasonable task allocation strategy is added to achieve load balance of computing cores.

References
[1] Q.M.Luo, Z.Ming, G.Liu. The Principle and Implementation of OpenMP [M]. Beijing: tsinghua university press, 2012.
[2] X.J.Gong, L.M.Zou, Y.X.Hu. Research on Parallel Program Design Method of Multi-core System Based on OpenMP[J].Journal of Nanhua University (Natural Science Edition), 2013, 27 (1): pp.64-68.
[3] Taxas Instruments. OpenMP Programming for Keystone Multicore Processors [EB/OL]. http://www.ti.com, 2012.

[4] Z.Q.Wang. Research and Application of Multi-Core DSP Parallel Optimization Method Based On OpenMP [D]. Hunan: National University of Defense Technology, 2014.

[5] Taxas Instruments Inc. KeyStone Architecture Multicore Navigator UserGuide [EB/OL]. http://www.ti.com/lit/ug/sprugr9d/sprugr9d.pdf, 2011-12.