A New Technique To Use A Parallel Compiler for Multi-core Microcontrollers

A. Somdip Dey
Department of Computer Science,
St. Xavier’s College [Autonomous]
Kolkata, India.

Abstract - Now a day multi-core microcontrollers are being used in various fields. Due to resource limitation of microcontrollers, programming them is difficult. This work presents a simple parallel compiler that can exploit multi-core to speed up parallel tasks on a multi-core microcontroller. The parallel constructors are introduced. A scheme to use compiler directives to hint the compiler is discussed. Experiments on the real processors are performed to validate the scheme. The results show that the compiler can exploit multi-core to speed up the computation tasks on the target microcontroller.

Keywords: multi-core microcontroller; parallel compiler; processing; multi tasking;

1 Introduction

Microcontrollers are very interesting and fascinating due to their vast applications nowadays. They are used in many embedded systems from a tiny robot to a car and even an enormous airplane. New development of microcontrollers occurs at very fast pace. Currently multi-core microcontrollers become available. However, programming model for such platform is still rare, especially the parallel compiler. This lack of proper compiler has hindered the use of multi-core microcontrollers.

Many programming models exist for desktops and servers. For example, OpenMP [1], Data parallel [2], including Intel Threading Building Block [3]. They are not suitable to multi-core microcontrollers due to the limit of resource of the target architecture. In particular, we are interested in one cost-effective multi-core microcontroller from Parallax Inc., an eight-core processor called "Propeller"[4]. Its architecture is very unusual. It has a global shared memory with very slow access and fast but small local memory for each core (see Fig. 1). Its instruction set is tailored to some specific way of using individual core. All of these results in a rather difficult programming model for this architecture.

The chip architecture was designed to have the 8 parallel independent cores which named cog by Parallax Inc. There is 2KB internal memory or register in each core. The share memory of 32KB RAM and 32KB ROM are accessible through memory hub which grants access only a single core at a time starting from cog 0 to 7. All I/O pins are connected to every core. Input pins value can be read and output pins value can be written at any time but the output pins value is the logical "OR" value from all 8 cores as the output pins can be driven by those cores.

This work presents a compiler of a parallel language targeted for Propeller. The approach we use in this work is to take a simple imperative language (in this case, a simplified C) and add some decoration to present a parallel constructor for a block of sequential code. The compiler will recognize this decoration and generates proper parallel code for that section. Of course, this approach has the limitation that the type of parallel constructor is limited. We show a number of useful constructors and their applications.

The flow of the article is as follows. The next section discusses the target language, Spin [5], which is embedded with the processor. Section III explains the compiler. Section IV shows examples of the parallel constructors. The conclusion is given in the end.

2 Propeller Spin Language

Spin language is embedded as an interpreter in the microcontroller. This is the solution embedded with the chip. Its intention is to make available a language that is used to control and specify the parallel operations of its multi-core thus simplify how users can program many cores of the chip. The structure of a Spin program is as follows. A global variable is declared before functions then the constant declaration. Object modules, which are a kind of function library or the class in OOP concept, also can be declared for use as well. The main function has all local variables declared immediately after the function declaration. The body of the function then follows with the function's statement.
The Fig. 2 shows a simple program written in Spin to generate frequency specified by user at customizable starting output port 16. There is one global variable named "wait_delay" which will be initialized to be user delay and growing in each iteration. The loop is created by the command "repeat" with a local variable i which will be used to step the output port forward. The value of the pin port A which is the pin 0 to 31 is set by the register name OUTA. The port is specified by the array variable indexing. The counter register CNT is added to the delay time and put as a parameter of the "waitcnt" function to interrupt the hardware to stop.

Special commands are used to start and stop a Propeller core. The command "cognew" initializes a new available core by uploading the corresponding source code to be run (similar to spawning a thread). The other command "coginit" related to the core initialization command is used when programmer need to identify a specific core of its availability. Both commands require the parameter to be used as a stack memory which is used for temporary calls and expression evaluation when the core is starting. A suitable amount of stack space is necessary for a program on a core to properly run. Also to stop the running core, "cogstop" command is used. Fig. 3 shows an example for swapping the output using many cores. The coginit command is used to start the specified core with the swap function at the given pin port.

3 Compiler For Parallel Programs

The base-language for our compiler is a simplified C called RZ [6]. RZ is a small language. Its syntax is very similar to the language C (but without type). It is the language used in a teaching class about compiler. The full set of compiler source code and tools are available in our institution [7]. The new parallel operations are added into the language by using compiler directives scoped over a section of normal code.

"#pragma parallel for" is used to specify parallel operations over a for-loop body. The parameters in the for-loop head: initialization, conditional and increment of loop-index, are parsed and stored. They are used for generating the output parallel code. Here is an example of the use of the pragma (see Fig. 4).

The output of the compiler is the statements in Spin to distribute the work in the body of for-loop over the available cores. In Spin language, the command for iteration is "repeat" and to start a new process, the command "coginit" is used.

As the to-be-run parallel code is issued to many cores, the command to start each core is called. The number of calls is equal to the number of cores. The body of loop is generated as a parameterized function. The parameters of the loop are used as the parameters of the "coginit" command.

The code-generator generates the body of loop as a function with arguments for sharing task and the local variables. This function contains the loop with specified iteration (the output Spin code will be shown with examples in the next section).

3.1 Parallel Constructors:

To illustrate the idea of using #pragma in parallel programs, two constructors are discussed. Example of parallel programs and the output code from the compiler are shown. These examples are: matrix multiplication, reduction and odd-even sort. We believe these examples show the intended use of the parallel #pragma which can be applicable over a wide range of parallel programs.

3.2 Matrix Multiplication:

Fig. 5 shows the pseudo code for matrix multiplication (NxN). Initially C is zero. The number of calculation is growing rapidly when the size of matrix is increasing. When the matrix size grows from 2x2 to 3x3, the number of calculations is grown from 8 to 27. It is growing at the rate of N3 where N is the matrix size (NxN).

There are three nested loops. Each element of C[i][j] is the inner product of the row i of A and the column j of B. The parallelization is made at the deepest inner loop. By distributing the calculation to each core, it will reduce the execution time in proportion to the number of core used in the calculation. Here is how to write a parallel version of the matrix multiplication (see Fig. 6).

We distribute the different "C += A * B" over the different core. To put a large amount of work to the limited number of cores, two loops (the innermost) are required. The first loop is used to issue work to several cores and the second loop is used to strip the vector properly for each core. Here is the output of the compiler (see Fig. 7).

The innest loop is stripped over many cores (a constant CORE) and "coginit" is called for each core. The function "par_fun" is the body of the for-loop. The "@stack[32*1]" is required to allocate a stack space for the core.

3.3 Reduction Sum:

Fig. 8 shows the code of reduction sum. We use "#pragma parallel for reduction" to indicate the type of parallel constructor. Initially sum is zero. The vector V (of size N) is reduced to a scalar value by summation. Reduction is done by the divide and conquer method. The vector is divided into two halves. The operation is applied on a pair with one element from the first half and another element from the second half. The result is stored "in-place"
at the first half of the array. Each iteration reduces the vector by half if there is enough processors to perform the operation concurrently and there is no data dependency. It takes log2N to reduce a vector of size N to a scalar. Each pair of numbers can be processed in each core in Propeller. Using two cores, a vector of size 1024 can be reduced to a scalar value with 10 iterations. Here is the output Spin code for reduction sum (see Fig. 9).

The logarithm function is used to calculate the number of the iteration. Also the power function, it is used to find the size of each level of the tree to correctly distribute the work to the available cores.

3.4 Odd-Even Sort:

This is an algorithms used for sorting on parallel systems. The comparison operation can be done in parallel. The algorithm compares the adjacent elements. Assume the first round is an even-round. The comparison starts at the 0th element. The next odd-round starts at the 1st element. If there are N cores, in N-1 iterations the sorting is done. Fig. 10 shows the parallel code for odd-even sort. Assuming the index starts at 1.

The output Spin Code is shown in Fig. 11.

The first loop iterates over all items. The second loop iterates over half of the items because each iteration is dealing with odd-only or even-only. The iterations needed in each round is N/2. The third loop distributes the work over many cores starting from the odd or even index. The index is calculated using modulo function which can be seen as the double-slash symbol.

Each of the algorithms is compiled and the result contained the parallel code section. All codes have a dedicated function for "coginit" command to be called when starting and initializing a core to run in parallel mode.

4 List of Figures

Fig. 1. The architecture of Propeller

Fig. 2. The sample LED toggling program in Spin language

Fig. 3. An example of a parallel program written in Spin

Fig. 4. The example of the pragma compiler directive

Fig. 5. Pseudo code for matrix multiplication

Fig. 6. The ready-to-compile parallel version of the matrix multiplication
5 Experiment

Three algorithms: matrix multiplication, reduction sum and odd-even sort, which can exploit the parallelism are used. Each program is compiled and then run in the Propeller microcontroller chip. The results are recorded for each specific test configurations related to number of core used.

The following figures listed, show the comparison of the execution time of each program varies by the number of core used. For the matrix multiplication, Fig. 12 shows that the speedup increases when more cores are used. Compare to single core, the speedup of the 6-core on 48x48 is 1.2. The larger matrix size have higher speedup as the overhead is smaller compare to the total time of execution.

The result of the reduction sum program is shown in Fig. 13. The result of the small data size indicates that the 6-core is slower than the 3-core. When increase data size, the 6-core speed up is better than the 3-core. The reason of anomaly, which the 6-core is slower on small data size is that there is larger overhead in core initialization process. The result of the odd-even sorting is shown in Fig. 14. For the 3-core, it seems that the speedup is almost independent from the size of data.
6 Conclusion

This work presents a parallel compiler for a particular multi-core microcontroller. The compiler directive "#pragma" is used to hint the compiler to generate a proper code for parallel section. Two parallel constructors are introduced. The experiments are performed to measure the speed up of the execution time while varying the number of cores used. The results show that the compiler generates correct output code that can exploit multi-core to speedup the computation.

7 Acknowledgment

Somdip Dey expresses his gratitude to all his fellow students and faculty members of the Computer Science Department of St. Xavier’s College [Autonomous], Kolkata, India, for their support and enthusiasm. He also thanks Dr. Asoke Nath, professor and founder of Computer Science Department of St. Xavier’s College (Autonomous), Kolkata, for his constant support and helping out with the preparation of this paper.

8 References


