# A Parallel Compiler for Multi-core Microcontrollers

W. Pornsoongsong
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, 10330, Thailand
wpornsoo@gmail.com

P. Chongstitvatana
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, 10330, Thailand
prabhas.c@chula.ac.th

***Abstract*—Due to resource limitation of multi-core 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; parallel programming*

## I. Introduction

Microcontrollers have become the backbone of many appliances. They are used in embedded systems such as robots and cars. New development of microcontrollers occurs at very fast pace. Currently, multi-core microcontrollers become available. This type of microcontrollers is powerful. They are used to serve multi-function control of machines and the applications with intensive computation. However, programming this new type of controllers is difficult due to the nature of multi-processing. Using sequential constructs of conventional programming languages does not help to facilitate this task.

Many programming models exist for multi-core processors on desktops and servers. For example, OpenMP [1], Data parallel [2], Cilk [3], including Intel Threading Building Block [4]. They are not suitable to be used on multi-core microcontrollers due to the limit of resource of the target architecture. We propose a compiler for this type of microcontrollers where some semi-automatic parallelization can be performed by the compiler to facilitate the task of programming for multi-core microcontrollers. There are many good reasons for such development.

1) Many "patterns" of multi-processing, such as for-loop, can be recognized and the concurrent tasks can be distributed to multi-core processors semi-automatically. This makes programming much easier and it also reduces many errors involved in manually assigned the tasks to several cores.

2) There are fewer tools on the development of this new type of microcontrollers compared to the tools available for desktop systems. It is undoubtly beneficial to prog ram in a high level language as opposed to programming at a lower level, such as assembly languages. A compiler for this new type of microcontrollers will help to reduce the development time.

In particular, we are interested in one cost-effective multi-core microcontroller from Parallax Inc., an eight-core processor, called "Propeller"[5]. The chip architecture was designed to have eight parallel independent cores which named Cog by Parallax Inc (Fig. 1). There are 2KB internal memory or registers in each core. The share memory of 32KB RAM and 32KB ROM are accessible through memory hub which grants access to 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 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.

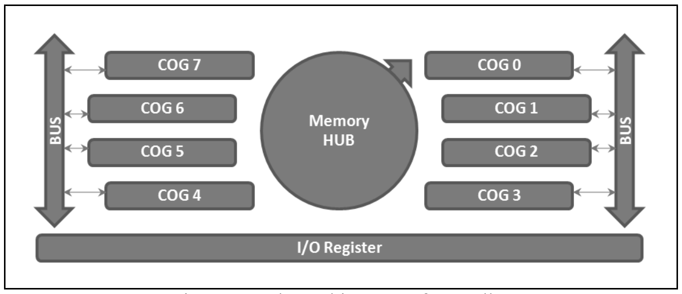

Figure 1. The architecture of Propeller

The flow of the article is as follows. The next section discusses the target language, Spin [6], 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.

## II. 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 follows by 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 a customizable output port 16. There is one global variable named "wait_delay" which is initialized as "user delay". It is 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.

```
CON
   OUTPIN = 16
   USER_DELAY = 1_000_000
VAR
   LONG wait_delay
PUB main | i
   wait_delay := USER_DELAY
   repeat i from 1 to 3
      OUTA [OUTPIN+i] := 1
      waitcnt (CNT+wait_delay)
      wait_delay += wait_delay
      OUTA := 0
```

Figure 2. The sample LED toggling program in Spin language

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 a programmer needs 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.

```
PUB main
   coginit (1, swap (16), @_stack[0])
   coginit (2, swap (17), @_stack[8])
   coginit (3, swap (18), @_stack[16])

PUB swap (port)
   repeat
      OUTA[port] = !OUTA[port]
      waitcnt (CNT+1_000_000)
```

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

## III. Compiler For Parallel Programs

The base-language for our compiler is a simplified C called RZ [7]. 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 [8]. 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).

```
#pragma parallel for
   for (i = 0; i < 10; i = i + 1)
      a[i] = b[i] * c[i];
```

Figure 4. The example of the pragma compiler directive

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).

## A. 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 [9]. We believe these examples show the intended use of the parallel #pragma which can be applicable over a wide range of parallel programs.

## B. 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 2×2 to 3×3, the number of calculations is grown from 8 to 27. It is growing at the rate of $N^3$ where *N* is the matrix size.

```
for i from 1 to N
   for j from 1 to N
      for k from 1 to N
         C[i][j] += A[i][k] * B[k][j]
```

Figure 5. Pseudo code for matrix multiplication

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).

```
for (i = 1; i <= N; i = i+1)
   for (j = 1; j <= N; j = j+1)
      #pragma parallel for
      for (k = 1; k <= N; k = k+1)
         C[i*N+j] += A[i*N+k]*B[k*N+j];
```

Figure 6. The ready-to-compile parallel version of the matrix multiplication

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*l]" is required to allocate a stack space for the core.

## C. 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.

```
repeat i from 1 to N
   repeat j from 1 to N
      repeat k from 1 to N/CORE
         repeat l from 1 to CORE
            stepsize := ((k-1)*CORE)+l
            coginit (l+1,
               par_fun (i, j, stepsize),
               @_stack[32*l])

PUB par_fun (i, j, k)
   C[((i-1)*N)+(j-1)] += A[((i-1)*N)
      +(k-1)]*B[((k-1)*N)+(j-1)]
```

Figure 7. The matrix multiplication compilation output

```
#pragma parallel for reduction
   for (i = 1; i <= N; i = i+1)
      sum = sum + V[i];
```

Figure 8. The parallel code of reduction sum

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 $log_2N$ iterations 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).

```
repeat i from LOG2(N) to 1
   repeat j from 1 to (POW(i-1)/CORE)+1
      repeat k from 1 to CORE
         coginit (k+1,
            par_fun (((j-1)*CORE)+k,
            POW(i-1)), @_stack[8*(k-1)])

PUB par_fun (idx, size)
   V[idx-1] += V[(idx-1)+size]
```

Figure 9. The reduction sum output Spin code

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.

## D. 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.

```
for (i = 1; i <= N; i = i+1)
   #pragma parallel for
   for (j = (i mod 2)+1; j < N; j = j+2)
      if (A[j] > A[j+1])
         swap(A[j], A[j+1]);
```

Figure 10. The parallel code for odd-even sort

Here is the output Spin code (Fig. 11).

```
repeat i from 1 to N
   repeat j from 1 to ((N/2)/CORE)+1
      repeat k
      from (i//2)+1+((j-1)*(2*CORE))
      to (j*(2*CORE)) step 2
         coginit (((k//(2*core))/2)+2,
            par_fun (k-1),
            @_stack[(k//(2*CORE))/2])

PUB par_fun (idx) | tmp
   if D[idx] > D[idx+1]
      tmp := D[idx]
      D[idx] := D[idx+1]
      D[idx+1] := tmp
```

Figure 11. The odd-even sort output Spin code

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 number of 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.

## IV. EXPERIMENTS

For the experiment, the programs which are able to run in both sequential and parallel execution were written to be used to test the compilation of the compiler and the efficiency of the real-hardware execution on the propeller microcontroller. The experimental hardware board is used for the measurement of the effectiveness of the compiled output and the efficiency of the output programs.

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 speedup is calculated by the execution time for sequential (i.e. single core) versus time for multi-core (Ts/Tp). The results are recorded for each specific test configurations related to number of core used.

The following figures 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 a single core, the speedup of 6-core on 48×48 is 1.2. The larger matrix has higher speedup as the overhead is smaller compared to the total time of execution.

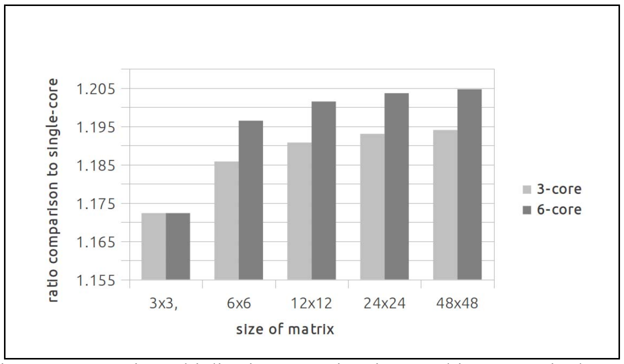

Figure 12. Matrix multiplication execution time: multi-core vs. single-core

The result of the reduction sum program is shown in Fig. 13. The result of the small data size indicates that 6-core is slower than 3-core. When increase data size, 6-core speed up is better than 3-core. The reason of anomaly which 6-core is slower on small data size is that there is large overhead in core initialization process.

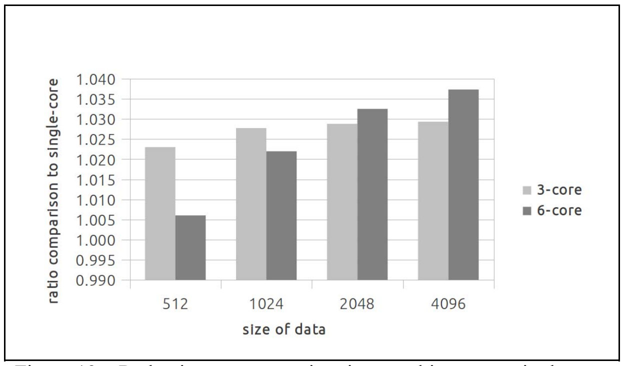

Figure 13. Reduction sum execution time: multi-core vs. single-core

The result of the odd-even sorting is shown in Fig. 14. For 3-core, it seems that the speedup is almost independent from the size of data.

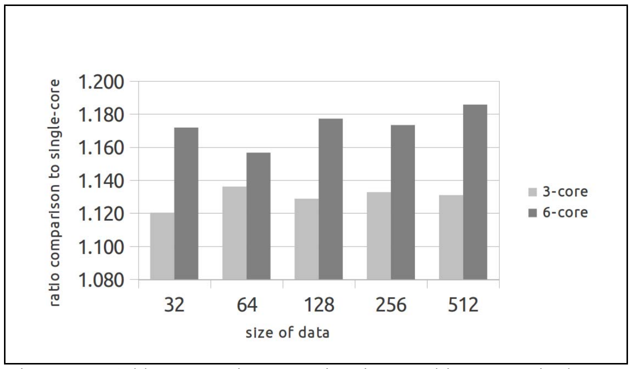

Figure 14. Odd-Even sorting execution time: multi-core vs. single-core

## V. 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 speed up the computation.

Because the compiler directives introduced in this work are based on for-loop, there are limitations of their applications. Many concurrent tasks are not expressed as for-loop, for example, a general multi-thread task or task synchronization. The pragmas illustrated here are not applicable. However, the concept of having a hint to the compiler about the pattern in the program that can be exploited by multi-core processors can be extended to cope with variety of parallelism expressible in a program.

## REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: An Industry Standard Api for Shared Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46-55, Jan-Mar 1998.

[2] W. Daniel Hillis and Guy L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM - Special issue on parallelism*, IEEE Magazine, vol 29, no. 12, Dec 1986.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. "Cilk: An efficient multithreaded runtime system," *In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 207-216, Santa Barbara, California, July 1995.

[4] N. Popovici and T. Willhalm. "Putting Intel Threading Building Blocks to work*," Proc. of the International Workshop on Multicore Software Engineering (IWMSE 2008)*, Leipzig, Germany, May 11 2008.

[5] Parallax Propeller, http://www.parallax.com/tabid/407/Default.aspx, 2012.

[6] J. Martin and S. Lindsay, "Parallax Propeller Manual," 2006.

[7] RZ language and its compiler, http://www.cp.eng.chula.ac.th/faculty/pjw/project/rz3/index-rz3.htm, 2012.

[8] RZ compiler tools kit, http://www.cp.eng.chula.ac.th/faculty/pjw /project/rz/rz2compiler.htm, 2012

[9] B. Wilkinson and M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers," *Pearson Prentice Hall*, 2005.