Programmer's Manual for XMTC Language, XMTC Compiler and XMT Simulator

UMIACS-TR 2005-45 (Part 2 of 2)

Aydin O. Balkan and Uzi Vishkin

Contents

Chapter 1

About This Document

Explicit Multi-Threading (XMT) is a computing framework developed at the University of Maryland as part of a PRAM-on-chip vision (**http://www.umiacs.umd.edu/˜vishkin/XMT**). Much in the same way that performance programming of standard computers relies on C language, XMT performance programming is done using an extension of C called XMTC.

The above mentioned web site provides a list of publications for readers interested in XMT Project. Two of these papers summarizes earlier research results and the first generation of the XMTC programming paradigm:

- U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism (Extended Summary and Working Document). Current version of UMIACS TR-98-05. First version: January 1998. (47 pages)

- D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. TOCS 36, 5 pages 521-552, Springer-Verlag, 2003. (26 pages)

This document acts as a tutorial, and demonstrates the basic programming concepts of XMTC language and provides examples and exercises. For more formal definitions as well as requirements and restrictions, we refer the reader to the first part of this technical report UMIACS-TR 2005-45, which is available at `http://www.umiacs.umd.edu/users/vishkin/XMT` web site.

Organization This tutorial consists of 7 chapters. The following is a brief overview of the tutorial.

Chapter 1: About This Document In this introduction chapter we summarize the purpose of this tutorial, and provide an overview to the whole document. We also provide pointers to additional resources.

Chapter 2: From PRAM to XMTC This chapter demonstrates how to write a parallel XMTC code, given a simple PRAM algorithm. We encourage the programmer to *Think in Parallel* and start writing parallel code without writing a serial version first.

Chapter 3: Working with External Inputs Currently, the XMT simulator cannot process arbitrary data files. In this chapter we demonstrate how to use the provided tools to prepare compatible data files.

Chapter 4: Simple Methods for Debugging the XMTC Program   In this chapter we demonstrate the tools that are available for debugging the XMTC program.

Chapter 5: Prefix-Sum Statments   This chapter provides examples regarding the usage of XMTC-specific *Prefix-Sum* statements, and compares the two different *Prefix-Sum* statements in terms of their performance.

Chapter 6: Concurrency in Memory Access   This chapter provides guidelines to handle concurrent writes to a single memory location.

Chapter 7: How to Nest Spawn Statements   Spawn statements can be nested using the `sspawn` statement. This chapter demonstrates this property, and provides examples.

Related Documents   This document is the second part of the UMIACS-TR 2005-45 technical report. The first part focuses on the features of the XMTC language and the usage of the XMT Toolchain. The most up-to-date version of these documents can be accessed at `http://www.umiacs.umd.edu/users/vishkin/XMT` web site.

History   The initial version of this document is completed on February 2005. Based on the improvements in the XMT Toolchain, some sections have been revised on May 2005 and February 2006.

Compatibility   This document lists the features of XMT Tool Chain Version **0.9** as of **Feb-13, 2006**. While later versions are expected to be backwards compatible, there might be small changes. For up-to-date information on such changes, please consult the *XMTC web page*.

Contact Person   Aydin Balkan: balkanay@umd.edu

XMTC Web Page   http://www.umiacs.umd.edu/~balkanay/xmt/

Chapter 2

From PRAM to XMTC

2.1   Spawning Virtual Threads

In this section we will show, how to spawn independent *virtual thread*s using `spawn` statement, and perform simple tasks within each virtual thread.
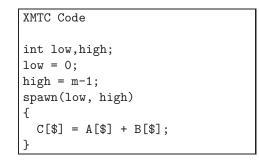
We start with presenting the *Vector Addition* problem, and show the relation between PRAM algorithm and XMTC code. Later we will add more features to this simple problem.

Vector Addition :   You are given two vectors A and B of the same size m. The objective is to add these two vectors into a third vector C.

Example-1   Vector Addition

The PRAM pseudo code and XMTC code for adding two vectors A and B of size m are given below.

```
PRAM Pseudo-Code


for i = 0 to m-1 pardo
{
  C[i] = A[i] + B[i];
}
```

```
XMTC Code


int low,high;
low = 0;
high = m-1;
spawn(low, high)
{
  C[$] = A[$] + B[$];
}
```

In both cases, we create m virtual threads, which can run concurrently. The `for-pardo` structure in the pseudo-code and the `spawn` statement in the XMTC code are responsible from this task. Each thread reads one element from A and B, adds them, and writes the result in C.

*Virtual thread*s are distinguished from each other by their unique *thread ID*. This *thread ID* is represented by `i` in the PRAM code, and by `$` in the XMTC code.

Due to the properties of the XMTC language, the parameters of the `spawn` statement must be kept within variables, which are declared as `register int`. ∎

Exercise-1   Compile and Simulate Your First XMTC Program

We are going to add two vectors with 10 elements each. Vector A contains the elements $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$ and vector B contains the elements $\{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$.

1. Go to http://www.umiacs.umd.edu/~balkanay/xmt/ and download `vecadd.zip`. It contains two files, which are required for compilation and simulation. `xmtData.bin` file contains the contents of A and B vectors, and the header file `vecadd.h` contains the declarations of the vectors. Create a new directory named `vecadd` in your work space, and copy the contents of the zip file there.

2. Write the following code in a text editor, and save it as `vecadd.c`. Make sure that this file is in the same directory as the `vecadd.h` file. (You may omit the comments)

```
#include "vecadd.h"
/*
 * The vecadd.h file declares the variables
 * A, B, C and m. Their values are read from
 * xmtData.bin file during simulation.
 *
 */

int main()
{
  int i;
  int low, high;
  low=0;
  high=m-1;
  spawn(low,high)
  {
    C[$] = A[$] + B[$];
  }

  for(i=0;i<m;i++)
  {
    printf("%d ",C[i]);
  }
  printf("\n\n");
}
```

3. Using the command prompt, go to `vecadd` directory. Type
   > `xmtc vecadd.c`
   This will compile your program, and generate the `vecadd.s` file.

4. Type
   > `xmtsim vecadd.s -binload xmtData.bin`
   to simulate your program using the XMT Simulator.

5. The result will be printed to the terminal.

```
1 5 9 13 17 21 25 29 33 37
End of simulation.
```

6. In order to print the result into a text file, e.g. `outFile.txt`, use the following simulation command:

> `xmtsim vecadd.s -binload xmtData.bin -out outFile.txt`

7. In order to get the number of instructions that are executed during the simulation, use the following command:

> `xmtsim vecadd.s -binload xmtData.bin -count`

The result is:

```
1 5 9 13 17 21 25 29 33 37

End of simulation.
Number of instructions: 479
Number of ALU instructions: 196
Number of SFT instructions: 125
Number of BR instructions: 11
Number of MD instructions: 0
Number of LS instructions: 85
Number of PS instructions: 0
Number of SPAWN instructions: 1
Number of DECODE instructions: 50
```

Here, the total number of instructions is 479. The instructions are classified into the functional units that they are assigned to. In respective order they are: *Arithmetic-Logic Unit*, *Shift Unit*, *Branch Unit*, *Multiply-Divide Unit*, *Load-Store Unit*, *Prefix-Sum Unit*, *Spawn Unit* and finally the remaining instructions, which are not assigned to any of these functional units.

8. In order to get the number of XMT clock cycles that are executed during the simulation, use the following command:

> `xmtsim vecadd.s -binload xmtData.bin -cycle`

The result is:

```
1 5 9 13 17 21 25 29 33 37

Total Execution Time = 2512.0
End of simulation.
```

Here, the total execution time is 2512 cycles.

Please note that you cannot use the `-cycle` and `-count` simulator options together. For details on the simulator please consult the XMTC Manual.

The $ character can be regarded as a **read-only** variable within the spawn block. It can be used in other statements (such as `if`) and mathematical expressions (such as `$+2`).

Exercise-2 Suppose that you only want to add elements with odd index numbers. Below, we show the modified PRAM pseudo-code as well as the spawn block. These changes ensure that the addition is performed only in threads with odd threadID.

```
PRAM Pseudo-Code

for i = 0 to m-1 pardo
{
  if (i is odd) then
    C[i] = A[i] + B[i];
}
```

```
XMTC Code

  int low,high;
  low = 0;
  high = m-1;
  spawn(low,high)
  {
    if($ % 2 == 1)
    {
      C[$] = A[$] + B[$];
    }
  }
```

Please note that the % operator in the above XMTC code is the modulo operator. Verify the correctness of the above code excerpt by modifying your code from Exercise 1, and then compiling and simulating. ∎

Exercise-3 Modify the XMTC program from Exercise 1 such that it implements the following PRAM pseudo-code:

```
PRAM Pseudo-Code

for i = 0 to m-1 pardo
{
if (i is even) then
    C[i] = A[i] + B[m-1-i];
}
```

Compile and simulate your code and verify that the result is $C = \{19, 0, 19, 0, 19, 0, 19, 0, 19, 0\}$. ∎

Chapter 3

Working with External Inputs

3.1   Introduction

Currently, the XMT toolchain does not allow users to read data from input files from within the XMTC code using system calls to file input/output routines. The simulator can read some specially formatted data files and load the contents in one part of the simulated XMT memory. The user can process his/her own external data files to create the memory content files and feed them into the simulator.

There are two main parts of external inputs:

- The *Memory Map–Header File*, which contains the names of the variables in the memory

- The *Memory Map–Binary File* and *Memory Map–Text File*, which contain the values of the variables in the memory.

The header file is included in the XMTC program either by using the `#include` directive or the `-include` option of the compiler. The data files are loaded into the simulator by using `-binload` or *-textload* options of the simulator. The contents of the binary and text files are the same, and the user needs to load only one of the two into the simulator. These two options (binary or text format) are provided for the convenience of the user.

3.2   Creating Data Files to use with the Simulator

In this section, we will explain through an exercise, how to create the necessary files for using external data in the XMT simulator.

We use the `memMapCreate` tool to create these files. We refer the reader to the XMTC Manual for features of this program, as well as current bugs and restrictions.

Exercise-4   Consider the vector addition program in Exercise 1. The header and binary files provided with the exercise define and initialize variables A, B, C and m for 10-element vectors. In this exercise we will create corresponding files for 50-element vectors.

1. Go to the directory, where you put the vector addition program (e.g. `vecadd`). From *Downloads* section in http://www.umiacs.umd.edu/~balkanay/xmt/ download and unzip vecadd2.zip into this directory. This will create files `vectorA-50.txt` and `vectorB-50.txt`.

```
**************************************************
**************************************************
*                                                *
*          S E T   F I L E   N A M E S           *
*                                                *
*  1. Set Header Name                            *
*  2. Set Memory Map Name                        *
*  3. Set Text file Name                         *
*  <  Back to previous Menu                      *
*                                                *
**************************************************


 *** > 1
Current Header File :
New Header File     : vecadd2.h


Current Header File : vecadd2.h
```

Figure 3.1: Screenshot after setting header name

These files are called *Content Files*. Each file contains some values, separated by whitespace (space or newline), for the vectors that we are creating. You can generate such files manually using a text editor, or you can write a program that converts some data file into this format.

2. Run `memMapCreate` while you are still in the same directory by typing:

   > `memMapCreate`

3. In the *Main Menu*, select 1. (by pressing 1 and ENTER key)

4. In the *Set File Names* menu select 1. Enter `vecadd2.h` as the header file name. See Figure 3.1.

5. In the same menu, select 2 to enter `vecadd2.bin` as the binary file name and select 3 to enter `vecadd2.txt` as the text file name.

6. Select < to go back to the *Main Menu*. Then select 2 to go to *Read/Write Files Menu*

7. We want to create two integer arrays (as vectors) with 50 elements. Select 2 to create the first one.

8. Enter `A` as the name, `1` as dimension, `50` as the size of dimension 0, and `vectorA-50.txt` as the source. Check your entry and answer `y` to the question, if everything is correct. See Figure 3.2.

9. Repeat this process for array B. Enter `vectorB-50.txt` as the source.

10. Repeat this process for array C. Enter `0` as the source in order to initialize all elements as 0.

11. Now, we need to prepare the scalar variable `m` that contains the number of elements in our program. Press 1 to add a scalar variable.

12. Enter `m` as name, verify the name (by entering `y`), and enter `50` as the value.

13. Press L from *Read/Write Files Menu* and compare the list of variables with Figure 3.3.

9

```
**************************************************
**************************************************
*                                                *
*        R E A D / W R I T E   F I L E S         *
*                                                *
*  1. Add Integer Scalar Variable                *
*  2. Add Integer Array  Variable                *
*  3. Add Double Scalar Variable                 *
*  4. Add Double Array  Variable                 *
*  R. Read Variables from Header File            *
*  L. List Current Variables                     *
*  D. Delete Last Variable                       *
*  H  Create Header File                         *
*  B  Create Text and Binary Files from sources  *
*  <  Back to previous Menu                      *
*                                                *
**************************************************


 *** > 2
Name of array Variable    : A
Number of dimensions (1-2): 1
Size of Dimension 1        : 50
Source (file name OR 0(default) OR R for random numbers ): vectorA-50.txt

Name      : A
Dimension: 1
Size [0] : 50
Source   : vectorA-50.txt
Is this correct (y/n)?
```

Figure 3.2: Screenshot after entering the details for Vector A

14. Press H to create the header file.

15. Go to *Read/Write Files Menu* and press B to create the binary and text files.

16. Change the first line of the program given in Exercise 1 to:

    ```
    #include "vecadd2.h"
    ```

17. Compile and simulate the modified vector addition program. The result should be an array of odd numbers from 1 to 49 (twice) and then back to 1.

                                                               ■

Using the `memMapCreator` program, you can create integer and double scalars and (currently) one or two dimensional arrays. As you create arrays, this program creates an additional scalar variable for each dimension of the array, and puts the size of the dimension in the binary file. You can refer to this variable from your XMTC code. For example, if the name of a two dimensional array is `myArray`, then the variables `myArray_dim0_size` and `myArray_dim1_size` contain the sizes of respective dimensions.

```
 *** > l
Name      : A
Type      : int
Dimension : 1
Size [0]  : 50
Source    : vectorA-50.txt

Name      : B
Type      : int
Dimension : 1
Size [0]  : 50
Source    : vectorB-50.txt

Name      : C
Type      : int
Dimension : 1
Size [0]  : 50
Source    : 0

Name      : m
Type      : int
Dimension : 1
Size [0]  : 1
Source    : 50
```

Figure 3.3: List of variables

Exercise-5   Consider the program from Exercise 4. Instead of using variable m, use A_dim0_size in the code. Do not change anything else. Compile and simulate. Verify that the result is the same. You could also have used B_dim0_size or C_dim0_size, since they are automatically generated as well, and all three arrays have the same size. ∎

Chapter 4

Simple Methods for Debugging the XMTC Program

Serial C programs can be debugged using conventional tools such as *gdb* or *ddd* after compiling with *gcc*. Currently for XMTC, such tools are not available. Instead, we provide the programmer a *Serializer* tool, which converts a parallel XMTC code into a serial C code. Then, the programmer can use the above mentioned debugging tools. In addition, we provide the user with a *Memory Reader* tool, which partially visualizes the content of the XMT memory.

This chapter guides the user through the *Serializer* and *Memory Reader* tools, and provides some hints for debugging a parallel XMTC program.

4.1 Serializer

The *Serializer* converts the parallel XMTC program into a serial program that can be compiled easily using GNU gcc or Microsoft's Visual Studio. It also creates a header file, that properly initializes the program data that you prepared using the `memMapCreate` program.

Exercise-6 **Problem:** We have an array A of 50 elements. We also have an initially empty array C. We want to copy elements of A to C following these rules:

- if $A[i] < 10$ then $C[i] \leftarrow A[i] - 5$
- if $A[i] = 10$ then $C[i] \leftarrow A[i]$
- if $A[i] > 10$ then $C[i] \leftarrow A[i] + 5$

Suppose that the code in Figure 4.1 is written for this purpose. The program displays the array C as $\{-4, -3, -2, -1, 0, 1, 2, 3, 4, 15, 16, 17, 18, 19...29, 30, 29, 28, 27...16, 15, 4, 3, 2, ...-5\}$, which is wrong because it does not contain the number 10.

Follow the below steps to debug this program:

1. Go to the download section of http://www.umiacs.umd.edu/~balkanay/xmt/and download buggy.zip. Extract the files into an empty directory.

2. Type `serXMTC buggy.c -memload buggy` and hit enter. This will generate the following files:

12

```
#include "buggy.h"

int main() {
   register int start, end;

   int i;

   start=0;
   end=A_dim0_size-1;

   spawn(start,end) {
      if(A[$] < 10) {
        C[$] = A[$]-5;
      } else {
        C[$] = A[$]+5;
      }
   }
   for(i=0;i<50;i++){
      printf("%d   ",C[i]);
   }
   printf("\n\n\n");
}
```

Figure 4.1: A simple program with a bug.

- serial_buggy.c
- serial_buggy.h
- serializing_header.h

3. Compile the serialized code using gcc by typing `gcc serial_buggy.c`. As you run, you will see the incorrect result on the screen.

4. You can insert `printf` statements in this `serial_buggy.c` and/or use debugger programs such as gdb or ddd to find out that the equality condition ($A[i] = 10$) is not handled properly in the `if` statement.

■

4.2   Memory Map Reader

Another tool that can be used for debugging purposes, is the Memory Map Reader. This tool reads an XMT binary data file and an XMTC header file, and creates a C header file that initializes arrays that are declared in the XMTC header file with the data in the XMT binary file. You can then browse the generated file to see the contents, or write a small C program that analyzes the data for you.

Exercise-7   Consider the "buggy.c" program of Exercise 6, in Figure 4.1.

1. Compile this code using XMTC Compiler, and simulate it using XMT Simulator.

13

2. In order to see the initial data, you can execute the following command:

```
memReader buggy.h xmtDataInitial.h xmtData.bin
```

If you browse the `xmtDataInitial.h` file, you will see that the arrays are initialized with the values from `xmtData.bin` file.

3. In order to see the data in the arrays after the simulation, you first need to simulate your XMTC program using `-bindump <filename>` option. Suppose that you used a file called `xmtDataResult.bin` in this option. Then you can execute the following command to read the contents of this memory file:

```
memReader buggy.h xmtDataResult.h xmtDataResult.bin
```

If you browse the `xmtDataFinal.h` file, you will see that the arrays are initialized with values, which are modified during the simulation.

4. Browse this file to see that the C array does not contain the value 10 due to the bug in the program.

5. For larger data, you can include the header file `xmtDataResult.h` into a C program that analyzes the contents of the arrays.

■

Chapter 5

Prefix Sum Statements

5.1 Prefix Sum

Prefix-Sum is an atomic operation with two operands. The usage is:

```
ps(int local_integer, psBaseReg ps_base);
```

The operand `local_integer` must be declared as an `int`, which is local to the current spawn block, and it must either have the value 0 or 1.

As this statement is executed, the value of the `local_integer` is added to `ps_base`, and the old value of `ps_base` is copied to `local_integer`.

Exercise-8 The task is to perform the following operation in parallel:

- Find all elements of an array, which are greater than 5 and less than 12
- Copy them into a new array
- Count these copied elements

1. Write the program in Figure 5.1.
2. Get the necessary files from `count.zip` file from download section in http://www.umiacs.umd.edu/~balkanay/xmt/.
3. Compile and Simulate the program, verify the correct result.

Counting in this program is performed by executing a `ps` operation on a common counter base `psb0`.

Note that in the end result, C array contains 0s (initial values) between the copied elements. In his exercise we did not try to group the copied elements together. Each *Virtual Thread* preserves the index value of the copied element. *Array Compaction* problem asks you to group the values toward the front of the array. ∎

You can initialize the *prefix-sum base* variable `psb0` with any value. Consider the changes in Figure 5.2 to the program in Exercise 8. Now, the counter starts at 100, and as a second change, we write the value of the counter to the new array instead of the value itself.

The most important WARNING of this chapter: The results show that the value of `psb0` increases with the index of the array. This happens only because the XMT simulator simulates the virtual threads consecutively, starting with the lowest virtual thread ID. In general, the XMT Programmer **MUST NOT** assume that the `ps` operation will be executed by the virtual threads in the order of the increasing thread ID. The actual execution order may depend on other factors such as physical distances within the processor chip.

```c
#include "count.h"

psBaseReg psb0;

int main()
{
  int i;
  int j=m;
  int low, high;
  low=0;
  high=j-1;
  psb0=0;
  spawn(low,high)
  {
     int temp;
     if(A[$] > 5 && A[$] < 12){
        C[$]=A[$];
        temp=1;
     } else {
        temp=0;
     }
     ps(temp,psb0);
  }

  for(i=0;i<j;i++)
  {
    printf("%d ",A[i]);
  }
  printf("\n\n");
  for(i=0;i<j;i++)
  {
    printf("%d ",C[i]);
  }
  printf("\n\n");
  printf("%d ",psb0);
  printf("\n\n");
  return 0;
}
```

Figure 5.1: Program for Exercise 8

16

```
...
  psb0=100;
  spawn(low,high)
  {
     int temp;
     if(A[$] > 5 && A[$] < 12){
        temp=1;
        ps(temp,psb0);
        C[$]=temp;
     } else {
        temp=0;
     }
  }
...
```

Figure 5.2: Modifications to the program of Exercise 8

5.2   Prefix Sum to Memory

Prefix Sum to Memory is an atomic operation with two operands. The usage is:

```
psm(int local_integer, int* psm_base);
```

The operand `local_integer` must be declared as an `int`, which is local to the current spawn block. Unlike the `ps` statements, the value of this operand is not limited to 0 or 1, i.e. it can assume any legal integer value.

The operand `psm_base` must be a pointer to an integer value in the memory.

As this statement is executed, the value of the `local_integer` is added to the value pointed by `psm_base`, and the old value of the memory location pointed by `psm_base` is copied to `local_integer`.

In the following exercises you will observe the differences between `ps` and `psm` statements. In the exercises the difference in *XMT Clock Cycles* have been highlighted as the primary metric for performance. In all three exercises we recommend to perform additional simulations using `-count` simulator option, in order to observe the number of simulated instructions as an additional metric for performance evaluation.

Exercise-9   Review ps and compare with psm with single prefix-sum base

In this exercise you are given an array $A[n]$ of n integers. You will find the number of elements of this array, whose last (decimal) digit is 0.

1. The first part can be done by the program shown in Figure 5.3.

2. The `count2.zip` file in download section of http://www.umiacs.umd.edu/~balkanay/xmt/ contains the XMTC code and the necessary header and data files.

3. Compile and simulate the `count2.c` file as shown below, and verify that there are 381 elements of array A, with their last digits equal to 0.

   > `xmtc count2.c -include count2.h`

   > `xmtsim count2.s -binload count2.bin`

4. Simulate the same file with `-cycle` option and see the number of clock cycles it takes to execute.

17

```
psBaseReg psb0;

int main(){

  int low, high;
  int i=0;

  low=0;
  high=n-1;
  psb0=0;

  spawn(low,high){
    int temp;
    int mod=10;
    temp=1;

    if(A[$] % mod == 0){
    ps(temp,psb0);
    }
  }

  printf("Total: %d\n",psb0);

}
```

Figure 5.3: The XMTC code for the Exercise 9

5. Replace the `ps` statement with `psm(temp,&i);` and modify the `printf` statement to print the value of `i`. Compile and simulate using `-cycle` option and observe that the performance is worse compared to the original implementation.

∎

The previous exercise demonstrated that `ps` statement yields better performance compared to `psm` statement, when:

• both are operated on a single base, and

• the value to-be-added is equal to 1.

The `ps` statement works with **registers at the processor core** and the `psm` statement works with **memory locations** as prefix-sum base. Performing an operation involving memory locations involves the overhead of memory access. Furthermore, the XMT architecture has specialized hardware that can efficiently handle multiple `ps` operations. Therefore, under these listed conditions `ps` statement should be preferred over `psm` statement.

Exercise-10   How to use ps with multiple prefix-sum bases

In this exercise you will use the same data arrays as in the previous exercise (Ex 9). This time you will count the number of elements in array A with all possible values of their last digits. In other words, you will count the number of elements in A that end with $0, 1, 2, ..., 9$.

18

```
psBaseReg psb0;
psBaseReg psb1;

int main(){
  int low, high;
  int i=0;

  low=0;
  high=n-1;

  while(i<10) {
    psb0=0;
    psb1=0;
    spawn(low,high){
      int temp;
      int mod=10;
      int rem0=i;
      int rem1=i+1;
      temp=1;

      if(A[$] % mod == rem0){
      ps(temp,psb0);
      }

      if(A[$] % mod == rem1){
      ps(temp,psb1);
      }
    }
    printf("Total: %d\t%d\n",psb0,psb1);
    i+=2;
  }
}
```

Figure 5.4: XMTC code with while loop

XMT architecture cannot accommodate arbitrarily many `psBaseReg` values to use with the `ps` statement, due to limited resources. For this exercise only, assume that only two such `paBaseReg` are available. In reality more resources are available; however, here we will demonstrate how to work efficiently with limited resources.

Make the following modifications to the `count2.c` XMTC code of the previous exercise:

1. Add a second `psBaseReg` variable called `psb1`

2. Modify the *Spawn Block* to handle two counters

3. Enclose the *Spawn Block* within a while loop such that it repeats 5 times

4. The final code should look as in Figure 5.4

5. Verify the output:

19

```
Total: 381      412
Total: 419      393
Total: 398      394
Total: 382      384
Total: 374      463
End of simulation.
```

6. Simulate the same code with `-cycle` option to see that the cycle count is more than 5 times of the cycle count of Exercise 9.

∎

Although the above implementation seems to work correctly, due to resource limitations, the overall task is serialized and it suffers from performance degradation due to overheads. The *Prefix-sum to memory* statement provides a more efficient and more generalized alternative to this implementation. The next exercise will demonstrate this point.

Exercise-11 Multiple prefix-sum bases with psm

In this exercise you will first perform the same task as in Exercise 10, but you will use the `psm` statement instead of `ps`. Then, you will generalize this code from 10 groups to any number of groups.

Suppose that the array C is initialized to 0, and it is big enough to hold the 10 counters that we need.

1. Start with the code in Figure 5.3

2. Modify the *Spawn Block* such that an integer `group` is calculated as $group = A[\$] \bmod 10$, and replace the `ps` statement by `psm(temp, &C[group]);` statement.

3. Enclose the `printf` statement in a loop such that it prints all of the values for $C[0], C[1], ...C[9]$.

4. Verify the output:
```
Total: 381
Total: 412
Total: 419
Total: 393
Total: 398
Total: 394
Total: 382
Total: 384
Total: 374
Total: 463
End of simulation.
```

5. Run the simulator using `-cycle` option to verify that this implementation is indeed more efficient than the implementation of Exercise 10.

6. Instead of hardcoding the number 10, use a C macro called `N_GROUPS`. Add these lines at the beginning of your XMTC code file, and modify the rest of your code accordingly:
```
#ifndef N_GROUPS
#define N_GROUPS 10
#endif
```

```
#ifndef N_GROUPS
#define N_GROUPS 10
#endif

int main(){

  int low, high;
  int i=0;

  low=0;
  high=n-1;

  spawn(low,high){
    int temp;
    int group;
    int mod=N_GROUPS;
    temp=1;
    group=A[$] % mod;
    psm(temp, &C[group]);
  }

  for(i=0;i<N_GROUPS;i++){
    printf("Total: %d\n",C[i]);
  }
}
```

Figure 5.5: Final version of the XMTC code for Exercise 11

7. The final code should look like Figure 5.5. Compile, simulate and verify that the result is the same.

8. Now, compile using the following command:
   > xmtc -count2.c -include count2.h -D N_GROUPS=1000

9. Simulate with -cycle option to see the results and the performance.

■

5.3   Comparison

Although both ps and psm statements are performing essentially the same operation, previous sections demonstrated the differences in their performance.

The XMTC programmer shall prefer the ps statement if both of the following conditions are satisfied:

- The number of simultaneously used prefix-sum bases is small (within the limits of the architecture resources)

- The value-to-be-added is always 1

21

On the other hand the programmer would need to use the `psm` statement if any of the following conditions is true:

- The number of simultaneously used prefix-sum bases cannot be limited during implementation, or if such a limitation would heavily serialize the execution

- The value-to-be-added is not restricted to 1.

Chapter 6

Concurrency in Memory Access

The purpose of XMTC is to provide PRAM-like programming. XMT uses a model between *Arbitrary Common-Read Common-Write (CRCW)* and *Queued-Read Queued-Write (QRQW)* programming model. For simplicity we focus on the Arbitrary CRCW programming model. In this chapter, we demonstrate the guidelines for correct XMTC coding, following the Arbitrary CRCW model.

In case of concurrent writes, the final contents of the memory may change depending on which thread executed the final *write* operation. On the other hand, concurrent reads do not change the contents of the memory, in other words, the end result of the program does not depend on the order in which the threads read a common value. Concurrent reads may create situations that cause lower performance. Currently, these issues are being addressed at lower levels (ISA and hardware) of XMT architecture. The reader shall expect an update on this document regarding performance issues during concurrent read operations. Finally, if a memory location is subject to mixed read and write operations, the programmer shall separate all write operations and all read operations from each other. This can be easily done by placing such operations in separate *spawn block*s.

6.1   Arbitrary CRCW

In the *Arbitrary CRCW* model, an arbitrary processor will write the value to the common memory location. The user chooses a thread to write according to the order of reaching a point in the program. A *gatekeeper* structure will ensure that only one of the threads is allowed to access the common memory location.

Exercise-12   In this exercise you are going to perform a common write to a memory location following *Arbitrary CRCW* model. You will use a *gatekeeper* to ensure proper operation.

1. Suppose that you have a statement in the *spawn block* of your XMTC code that may cause multiple threads to write into a common memory location. For example:

   ```
   C[0] = A[$];
   ```

2. You will assign an order to the threads based on their arrival time to a specific point in the program. Then you will let one of the threads write to the common location C[0].

3. Figure 6.1 shows a program, where we perform a prefix-sum operation to determine the order in which the threads arrive at the prefix-sum statement. Then we use this order in a conditional statement to make sure that the *first-arriving thread* is allowed to write into

```
#include "concurrent.h"

psBaseReg psb0;

int main() {
    int start, end;
    int i;

    psb0=0;
    start=0;
    end=A_dim0_size-1;

    spawn(start,end) {
        int temp=1;
        ps(temp,psb0);

        if(temp==0) {
          C[0]=A[$];
        }

    }
    for(i=0;i<50;i++){
        printf("%d    ",C[i]);
    }
    printf("\n\n\n");
}
```

Figure 6.1: Example code for Arbitrary CRCW model of programming

the common location only. This structure is the *gatekeeper* that allows only one thread to access the common location.

■

Exercise-13   You can change the order of the thread that you want to grant access. Modify the value 0 in the `if` statement and observe that a different thread writes a different value to C[0].
■

We would like to refer the reader to the **warning** in Section 5.1 regarding the simulation order of the threads. In real hardware, the order of arrival at a common point in the *spawn block* may depend on many parameters including the physical location of the processing unit on the chip, and it will be determined during run-time. The programmer must not assume that the threads will reach such common points in an order based on their *thread ID*.

Chapter 7

How to Nest Spawn Statements

The single-spawn statement `sspawn` is used to spawn a single additional virtual thread from within the *spawn block* of a thread. The usage is:

```
spawn(low, high)
{
  int child_ID;

  ... Some code here ...


  sspawn(child_ID)
  {
    ... Initialization Block: Code for newly spawned thread
  }
  ... Some other code here ...
}
```

Warning: The `sspawn` statement will spawn a thread, which will execute the same spawn block as the currently running thread. Therefore, unless the `sspawn` is enveloped by a control structure such as `if` statement the threads may keep spawning, and eventually the simulator may crash.

Warning:

Exercise-14   Write a program that copies the array A to array C. In this exercise, spawn only $A\_dim0\_size/2$ threads instead of A_dim0_size threads, as you would normally do. Each thread should perform its duty of copying, then it shall spawn a single thread that will copy one other element, so that the whole array will get copied. In addition, each thread from the initial spawn set shall mark the array B with the integer 1, and each later-spawned thread shall mark the array B with integer 2.

1. Write the program shown in Figure 7.1. Compile and execute it.

2. Observe that the later-spawned threads assume the *thread ID* in an incremental way, starting from the highest *thread ID* of the initially spawned threads.

Please note that the variable `child_ID` (`newID` in Exercise 14) is not explicitly modified in the program code. The internal mechanisms of the XMT architecture assigns a value to this variable in runtime. The value of this variable is not known by the user, prior to the execution. Particularly, in Exercise 14, it is incorrect to assume before execution that `newID=$+25`, although the result of the simulation is consistent with this. The only correct assumptions regarding the `newID` values are:

- $newID > high$

- for singly spawned threads, `newID` values are consecutive integers starting with $high + 1$, i.e. if a total of k threads are spawned during the execution of all threads of a *Spawn Block* using `sspawn` statement, then for each of these threads $high < newID < high + k$

where $high$ is the highest value of the *Thread ID*s of the initially spawned set using the `spawn` statement (please see the usage box at the beginning of this chapter).

```
#include "sspawn.h"

// sspawn.h has :
// volatile int locks[50];
// for synchronization
// all elements of locks are initialized to 0

int main() {
   int start, end;
   int i;

   start=0;
   end=(A_dim0_size/2)-1;

   spawn(start,end) {
      int newID;

      if($ >= 25) {
        while (locks[$]==0); // spin wait
      }

      if($ < 25) {
        sspawn(newID)
        {
          B[$]=1;
          B[newID]=2;
          locks[newID]=1; // child can start now
        }
      }

      C[$]=A[$];
   }

   for(i=0;i<50;i++){
      printf("%d  ",A[i]);
   }
   printf("\n\n\n");

   for(i=0;i<50;i++) {
      printf("%d  ",B[i]);
   }
   printf("\n\n\n");

   for(i=0;i<50;i++) {
      printf("%d  ",C[i]);
   }
   printf("\n\n\n");
}
```

Figure 7.1: Program with sspawn statement for Exercise 14