The ICE Language and Compiler: Manual and Tutorial

By: Fady Ghanim, Uzi Vishkin, and Rajeev Barua

Contents

1. Introduction

1.1 The purpose of this manual

Immediate Concurrent Execution (ICE) is a new parallel programming model developed at the University of Maryland as part of a PRAM-on-chip vision. The ICE model is intended to facilitate easier programming of parallel systems, and PRAM-like algorithms while making minimal compromises on performance. The ICE model in its current form is an extension of the C language

The ICE model relies on the concept of lock-step execution inside parallel regions, as opposed to threaded model in other parallel programming languages and extensions. In lock-step execution, an instruction in a parallel section will not be executed on any parallel context until all contexts have finished executing the previous one. This makes parallel programming easier since the programmer does not need to worry about issues related to synchronization. This is due to the fact that the compiler will take care of it.

This document will provide instructions on how to operate the current version of the ICE compiler, a description of the new keywords and utilities, and their usage. It will also provide examples in relevant sections on how to use the new extensions.

To be able to use this language effectively, the reader needs to have basic understanding of the PRAM and WD-Model, and basic command of the C/C++ language.

2. The ICE Language

2.1 New ICE Statements

The ICE language is a superset of the C programming language. Ideally, a serial program written in pure C would compile and run through the tool chain without any problems. This, however, is not the case due to limitations in the tool chain explained in this document and in the XMT Toolchain Manual. This section summarizes the single new statement of ICE language that allow parallel programming of PRAM algorithms as is.

2.1.1 The Pardo Statement

2.1.1.1 Usage

```
pardo (int CID = LB; UB; STEP) {
        //Body
        Stm1;
        Stm2;
}
```

Where:
CID : an identifier to denote the parallel context ID
LB: the Lower bound; The ID of the first parallel context
UB: the Upper Bound; The ID of the last parallel context
STEP: the step between one context ID and the next

This statement will create as many as $\left\lceil \frac{(UB - LB + 1)}{Step} \right\rceil$ *concurrent* parallel contexts, with the first having ID of ***LB***, the last having an ID of ***UB***, with a difference of ***STEP*** between the IDs of one parallel context and the next.

The first part of the pardo statement consists of the variable name ***CID*** and ***LB.*** The variable name ***CID*** is an identifier for the parallel context ID that can be defined either as part of the pardo statement, or earlier, and can be any sequence of alphanumeric characters aside from keywords. ***CID*** denotes the ID of the current parallel context wherever it is used inside a pardo block. ***LB*** is the initializer of ***CID*** and can be any legal expression, identifier or constant integer, or can be absent if ***CID*** was initialized before the pardo.

The second and third part of a pardo statement are ***LB*** and ***STEP*** consecutively. Both ***UB*** and ***STEP*** can be an expression, a predefined identifier or an unsigned constant number. ALL three parts of a pardo statement must be integers. A pardo requires all three parts to work, and will issue a syntax error if any one of those three is missing.

Statements within a pardo block are executed in a lock-step fashion. Namely, every parallel context starting from context ***LB*** up to context ***UB*** will execute statement 1 first, before any context can start executing statement 2. Once all contexts finished executing statement 1, they will start executing statement 2.

Variables declared inside of a pardo block are local to the parallel context, meaning that each context will have its own local copy of that local variable.

2.1.1.2 Usage Examples

Example 1: example of legal pardo usage

In the following code we create N parallel contexts with IDs ranging from 0 to N − 1. Then, we create an integer variable temp. Each virtual thread has its own copy of this variable. All N parallel contexts read the array C using an expression containing the identifier “i” simultaneously. Based on this value, we copy an element from array B to array A either as it is and then increment it, or the negative of it. This copying is done simultaneously at the same time for all parallel contexts.

```
Int main() {
   ….
   Int N;
   ….
  pardo (int i = 0; N-1; 1) {
         int temp = C[i*2];
         if (temp > 0) {
            A[i] = B[i];
            A[i]++;
         } else {
            A[i] = -1 * B[i];
         }
  }
```

Remember that execution is done based on a lock-step model. So assuming C has the values [1, 3, -5, -4, 6, -2, -7, 1] and N = 4, this is how the execution of the pardo block will proceed:

Parallel context ID	i = 0	i = 1	i = 2	i = 3
(T)ime step = 1	$temp_0$ = c[0]	$temp_1$ = c[2]	$temp_2$ = c[4]	$temp_3$ = c[6]
T = 2	If ($temp_0$ > 0)	If ($temp_1$ > 0)	If ($temp_2$ > 0)	If ($temp_3$ > 0)
T = 3	A[0] = B[0]	A[1] = -1 * B[1]	A[2] = B[2]	A[3] = -1 * B[3]
T = 4	A[0]++	-	A[2]++	-
T = 5	*The pardo block terminates for all parallel contexts*			

Notice that when contexts i = 0 and i = 2 are executing A[i]++, contexts i = 1 and i = 3 were waiting for the other two contexts to complete.

Example 2: WD summation

In this example, we find the sum of the elements in array ***A***. First we copy in parallel the array ***A*** into level 0 of array ***B***. Then, we add every two adjacent elements, in parallel. We repeat that for all levels of ***B*** until in the end we have the summation in ***B[logN+1][1]***

```
#define N 8
#define logN 3

unsigned exp (int expon);          //calculates and return 2^exp
int A[N];
int B[logN+1][N];

int main() {

  pardo (int i=0; N-1; 1) {
          B[0][i] = A[i]
  }

  for (int h = 1; h < LOGN; h++) {
          pardo (int i = 0;  (N/exp(h)) - 1; 1) {
               B[h][i] = B[h-1][2i+1] + B[h-1][2i];
          }
  }

  Printf ("sum is %d", B[logN+1][1];
}
```

2.1.1.3 Limitations

Currently the ICE compiler is still in its initial stage and as such, there are certain limitations that will be addressed as the compiler matures. It is important for the reader to be aware of these limitations since that will save them time and effort while writing their programs. These limitations are:

1- Function calls inside a parallel section are not supported. You will not be expected to use such function calls and if you do it would be at your own risk.
2- Structures and pointers are not supported. Again: use at your own risk.
3- Nested parallel sections are not supported.
4- Nested loops inside a parallel sections are not supported. However, you can safely use as many none nested loops as you want.
5- *switch case* statements inside Parallel sections are not supported. Use multiple *if* - *else if* - *else* statements instead.
6- If your step and/or upper bound are expressions (i.e. a+b*c), you should first perform the operation and store the result in a temporary variable and then use that in the pardo command. For example:

```
tmp = a+b*c;     //use tmp in pardo for step/UB

pardo (...;...;tmp){}

or

pardo (...;tmp;...){}
```

7- The compiler does not support a loop block nested under an *if-else* statement inside a pardo (i.e. the following is **not** supported

```
pardo (...;...;...) {
    if (....) {
        while (...) { ///not supported
            ...
        }
    }
}
```

However, inside a loop you can nest if-else statements freely.

```
pardo (...;...;...) {
        while (...) {             // supported
                if (....) {
                                ...
                }
        }
}
```

8- Don't use any of the following identifiers (or any variances of them) for variables or functions names:

   ICE__PARDO__BARRIER
   ICE__Pardo__START
   ICE__Pardo__END
   OUTLINE__PARDO
   Ice__Depth
   Ice__Work
   ___Total_finished
   gk

9- Currently the ICE compiler does not have a built-in way to implement arbitrary concurrent writes. Instead, we created the specialized utility described in section 2.2.2 to perform concurrent writes to scalars and single memory locations. Please refer to section 2.2.2 in this document for more details.

10- Currently, the ICE compiler have a problem with using memory efficiently. This may cause problems with large datasets to give wrong answers upon execution, or give no answers at all. The dataset size that will work correctly depends on many factors, and as such varies from one problem to the next. When doing your projects, start with the smallest dataset that will be provided to you. If that works move on to larger data sets.

11- All limitations mentioned in the XMTC compiler documentation apply here as well, since, as part of the compilation process, the ICE compiler uses the XMTC compiler.

2.2 The ICE Library

The ICE language features a library of utilities and functions commonly used by parallel programmers and parallel programming learners alike. In this section, we will discuss those libraries.

Note: To be able to use any of the following ICE utilities, make sure to include the header file “ice.h”, and then initialize them using ***INIT***

2.2.1 Work and Depth Counters

Since one of the goals of the ICE language is to be an educational tool targeted at PRAM and the Work Depth Algorithmic Models (WD-Model), we decided that learners would benefit from the ability to experiment and see the effects of their experimentation in terms of work and depth.

We define three utilities for the accounting of work-depth for the various ICE programs, these utilities are:

```
WORKDEPTH
WORK(unsigned &);
DEPTH(unsigned &);
```

2.2.1.1 Usage

To use these utilities the programmer needs to insert them at **every step** they would like to account the work and/or depth for in their program.

Both of ***WORK()*** and ***DEPTH(***) are useful in situations where the user would like to see the contribution of a certain step to the whole work/depth complexity of the entire program. In both cases, the user needs to provide the variable s/he wants to be used for collecting the statistic. The DEPTH utility will increment the variable by 1. The WORK utility will increment the variable by the number of parallel contexts executing that step. To get the result, the programmer needs to either print it out to STDOUT, or access the data in any other way they like[1].

WORKDEPTH provides a simple way for collecting **both** the work and depth statistics for the **whole** program. Hidden work and depth counters will be incremented each time the step is executed. At the end of the execution of the program, the work and depth will be printed to STDOUT. Note that this will happen without any further instructions from the user.

[1] Look-up the different ways to look at the contents of memory location discussed in section 10.6 in the XMTC Toolchain Manual

2.2.1.2 Usage example

Example 3: For this example, we will use example 2 from earlier. Find the total work and depth, and find the contribution of each pardo to the total work and depth statistic.

```
#include “ice.h” //must include this to be able to use ICE utilities
INIT; // must run before using any of the ICE utilities
#define N 8
#define LOGN 3

unsigned exp (int expn);            //calculates and return 2^expn
int A[N];
int B[LOGN+1][N];

int main() {
  int CopyDepth1 = 0; // the depth for copying A
  int CopyWork1 = 0; // the work for copying A
  int SumDepth2 = 0; // the depth for copying A
  int SumWork2 = 0; // the work for copying A

  pardo (int i=0; N-1; 1) {
          B[0][i] = A[i]
          DEPTH(CopyDepth1);
          WORK(CopyWork1);
          WORKDEPTH;
  }

  for (int h = 1; h < LOGN; h++) {
          pardo (int i = 0;  (N/exp(h)) - 1; 1) {
               B[h][i] = B[h-1][2i+1] + B[h-1][2i];
               WORK(SumWork2);
               DEPTH(SumDepth2);
              WORKDEPTH;
          }
  }

  printf (“sum is %d”, B[LOGN+1][1];
  WORKDEPTH;

 printf(“the depth for copying is %d , and the work is %d”
            , CopyDepth1, CopyWork1 ); // depth will be 1, work will be 8
 printf(“the depth for summation step is %d , and the work is %d”
            , SumDepth1, SumWork1 ); // depth will be 3, work will be 7
}
/// total depth will be (logN + 2) = 5, and total work will be 2N = 16
```

Example 4: finding work and depth in situations of an unbalanced depth of different parallel context

In this example, find A[i] =$\sum_{j=0}^{i} j$, and find the total depth and work, and the depth of the summation process. (Note: this example "does not make sense" as an efficient parallel algorithm; its only purpose is as an example of unbalanced depth.)

```
#include "ice.h"
INIT

#define N 1024
unsigned A[N];

int main() {
        unsigned D = 0;

        pardo (int i = 0; N-1; 1) {
                unsigned sum =0;
                WORKDEPTH;
                for (int j = 0; j < i + 1; j++) {
                        sum += j;
                        DEPTH(D);
                        WORKDEPTH;
                }
                A[i] = sum;
                WORKDEPTH;
        }

        printf("D= %d\n", D); /// prints 1024
        return 0;
}
/// total depth will be (N + 2) = 1026
/// and total work will be N×(N+1)/2 + 2N = 525824
```

Notice that the ***for*** loop in this example will have different number of iterations per different parallel context. (i.e. context 0 will have 1 iteration, while context 1023 will have 1024 iterations). In this case, notice how the depth is counted along the longest path.

Note: Keep in mind that the sum is not written to ***A[i]*** until all contexts finish their loop iterations.

2.2.2 Concurrent write

The ICE language is meant to provide an arbitrary Concurrent Read Concurrent Write (CRCW) PRAM model. However currently the ICE compiler does not have a built-in way to implement that. We define this utility instead to allow programs to use that feature:

void CWrite(int value, int &var);

2.2.2.1 Usage

This utility is handy for whenever there are multiple values to be written to a certain variable. Simply use it to perform the write by supplying a ***value*** to be written and the ***var*** to be written to.

2.2.2.2 Usage example

Example 4: assume that your program requires that in a certain step, there are going to be multiple writes to variable X

```
#include "ice.h" //must include this to be able to use ICE utilities
INIT; // must run before using any of the ICE utilities

int main() {
    Int x;
    ….
    pardo (int I = 0; 100; 1) {
            ….
            CWRITE (I,x); // legal
            ….
    }
    …
    Printf("value of x is %d", x);  // prints the ID of the context that wrote to x successfully
}
```

Please remember that in arbitrary CRCW, you cannot make assumptions on which context managed to do the write successfully.

3. Compiling and Executing ICE programs

3.1 The ICE Compiler

The ICE compiler translates the program written in the ICE language into its equivalent implementation in XMTCC. As such many of the procedures used in compiling and executing the ICE program is very similar to the process discussed in the XMTCC manual and tutorial.

For the programmers' convenience, ICE.py is a one stop script that will facilitate the procedure of compiling the ICE program and the resultant XMTC program, and produce XMT binary that can be run on either the XMT FPGA, or the XMT simulator.

3.1.1 Usage

We start first by examining the Script Command Line

```
ICE.py [options] -i <inputfile> -o <outputfile>
-h --help                  Display this help message.
-w                         Terminate compilation on warnings
-g                         Produce a serial debug-able version of the original ICE program
-I --Include               Specify file or path to include
-V --Verbose               Verbose mode – Display all commands executed
-D --Dirty                 Keep all Intermediary files
--XBO                      specify Data file
--SIM #                    Produce a binary to be used with the simulator with # TCUs
```

This script requires the user to supply both an input source, and the name of the output file. The inputs are files with the extension .cpp. The outputs are files <filename>.sim and <filename>.b. When specifying the output all the user needs is to supply <filename> without any extensions.

Therefore, if you have a file called example2.cpp, the command to compile this would be:

```
ICE.py –i example2.cpp –o ex2
```

This will produce two files ex2.b and ex2.sim, which you can use to execute them using the xmtfpga. It is worth noting that the script tells xmtc to dump a map of the global variables in GlobalMap.txt[2].

3.1.2 Including header and .xbo files

To include header files, use the ***--Include*** option along with the path to the files to include. To link an XMT binary object (.xbo), use the ***--XBO*** flag and specify the files to include.

```
ICE.py –i example2.cpp –o ex2 --Include ICE.h --XBO example2.xbo
```

3.1.3 Compiling for use with the XMT Simulator

The ICE.py script will compile the ICE program for the FPGA by default. To compile an ICE source code into an XMT binary that can work with the simulator, you have to issue the --SIM option and then specify the number of TCUs you want to simulate.

```
ICE.py –i example2.cpp –o ex2 --SIM 1024
```

[2] Refer to the in section 10.6 in the XMTC toochain manual

3.1.4 Debugging your ICE program

To debug your ICE program, use the –g option. This option will produce a serial version of your program compiled using GCC with –g flag. This file can be executed on your machine and can be debugged using GDB (or its graphical frontend DDD). In this mode, the output filename that you provide will be the name of the executable produced by GCC. It simply goes like this:

```
ICE.py –i example2.cpp –o ex2 -g
```

You can also use ***printf(“custom messages”)*** to debug your code, if you are not comfortable with GDB.