XMT-HW2: Shared-Memory Sample Sort

Course:	ENEE759K/CMSC751
Title:	Shared-Memory Sample Sort
Date Assigned:	February 24, 2014
Date Due:	**Part A – ICE**: March 7, 2014, 11:59pm
	Part B – XMTC: March 10, 2014, 11:59pm
Contact:	**ICE**: Fady Ghanim - fghanim@umd.edu
	XMTC: James Edwards - jedward5@umd.edu

1 Assignment Goal

The goal of this assignment is to provide a randomized sorting algorithm that runs efficiently on XMT. While you are allowed some flexibility as to what serial sorting algorithms to use for different steps of the parallel algorithm, you should try to find and select the most efficient one for each case. The Sample Sort algorithm follows a "decomposition first" pattern and is widely used on multiprocessor architectures. Being a randomized algorithm, its running time depends on the output of a random number generator. Sample Sort performs well on very large arrays, with high probability.

In this assignment, we propose implementing a variation of the Sample Sort algorithm that performs well on shared memory parallel architectures such as XMT.

2 Problem Statement

The Shared Memory Sample Sort algorithm is an implementation of Sample Sort for shared memory machines. The idea behind Sample Sort is to find a set of $p-1$ elements from the array, called *splitters*, which partition the n input elements into p groups $set_0 \ldots set_{p-1}$. In particular, every element in set_i is smaller than every element in set_{i+1}. The partitioned sets are then sorted independently.

The input is an unsorted array A. The output is returned in array *Result*. Let p be the number of processors. We will assume, without loss of generality, that N is divisible by p. An overview of the Shared Memory Sample Sort algorithm is as follows:

Step 1. In parallel, a set S of $s \times p$ random elements from the original array A is collected, where p is the number of TCUs available and s is called the oversampling ratio. Sort the array S, using an algorithm that performs well for the size of S. Select a set of $p-1$ evenly spaced elements from it into S': $S' = \{S[s], S[2s], \ldots, S[(p-1) \times s]\}$

These elements are the splitters that are used below to partition the elements of A into p sets (or **partitions**) set_i, $0 \leq i < p$. The sets are $set_0 = \{A[i] \mid A[i] < S'[0]\}$, $set_1 = \{A[i] \mid S'[0] \leq A[i] < S'[1]\}$, $\ldots$, $set_{p-1} = \{A[i] \mid S'[p-2] \leq A[i]\}$.

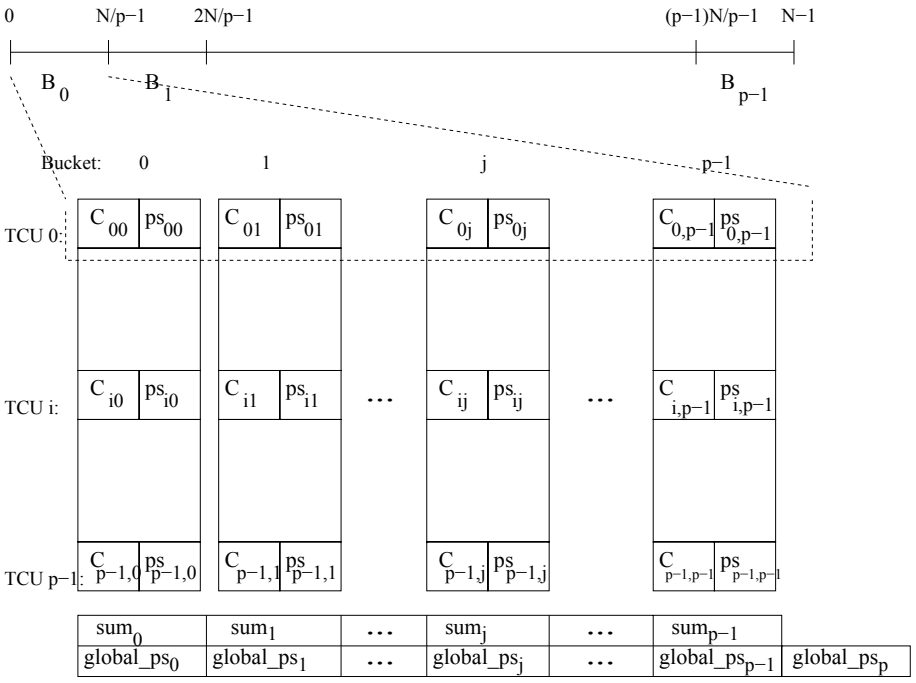

Figure 1: The C matrix built in Step 2.

Step 2. Consider the input array A divided into p subarrays, $B_0 = A[0,\ldots,(N/p)-1]$, $B_1 = A[N/p,\ldots,2(N/p)-1]$ etc. The ith TCU iterates through subarray B_i and for each element executes a binary search on the array of splitters S', for a total of N/p binary searches per TCU. The following quantities are computed:

- c_{ij} - the number of elements from B_i that belong in partition set_j. The c_{ij} makes up the matrix C as in figure 1.
- $partition_k$ - the partition (i.e. the set_i) in which element $A[k]$ belongs. Each element is tagged with such an index.
- $serial_k$ - the number of elements in B_i that belong in $set_{partition_k}$ but are located before $A[k]$ in B_i.

For example, if $B_0 = [105,101,99,205,75,14]$ and we have $S' = [100,150,\ldots]$ as splitters, we will have $c_{0,0} = 3$, $c_{0,1} = 2$ etc., $partition_0 = 1$, $partition_2 = 0$ etc. and $serial_0 = 0$, $serial_1 = 1$, $serial_5 = 2$.

Step 3. Compute prefix-sums $ps_{i,j}$ for each **column** of the matrix C. For example, $ps_{0,j}$,$ps_{1,j}$,...,$ps_{p-1,j}$ are the prefix-sums of $c_{0,j}$,$c_{1,j}$,...,$c_{p-1,j}$.

Also compute the sum of column i, which is stored in sum_i. Compute the prefix sums of the $sum_1,\ldots,sum_p$ into $global\_ps_{0,\ldots,p-1}$ and the total sum of sum_i in $global\_ps_p$. This definition of $global_ps$ turns out to be a programming convenience.

Step 4. Each TCU i computes: for each element $A[j]$ in segment B_i, $i \cdot \frac{N}{p} \leq j < (i+1)\frac{N}{p}$:

$$pos_j = global\_ps_{partition_j} + ps_{i,partition_j} + serial_j$$

Copy $Result[pos_j] = A[j]$.

Step 5. TCU i executes a (serial) sorting algorithm on the elements of set_i, which are now stored in $Result[global\_ps_i, \ldots, global\_ps_{i+1} - 1]$.

At the end of Step 5, the elements of A are stored in sorted order in $Result$.

3 Hints and Remarks

Sorting algorithms The Sample Sort algorithm uses two other sorting algorithms as building blocks:

- Sorting the array S of size $s \times p$. Any serial or parallel sorting algorithm can be used. Note that for the "interesting" values of N (i.e. $N \gg p$), the size of S is much smaller than the size of the original problem. An algorithm with best overall performance is expected.
- Serially sorting partitions of $Result$ by each TCU. Any serial sorting algorithm can be used. Remember to follow the restrictions imposed on spawn blocks, such as not allowing function calls, and avoid concurrent reads or writes to memory.

Oversampling ratio The oversampling ratio s influences the quality of the partitioning process. When s is large, the partitioning is more balanced with high probability, and the algorithm performs better. However, this means more time is spent in sampling and sorting S. The optimum value for s depends on the size of the problem. We will use a default value of $s = 8$ for the inputs provided.

Random numbers for sampling Step 1 requires using a random number generator. Such a library function is not yet implemented on XMT. We have provided you with a pre-generated sequence of random numbers as an array in the input. The number of random values in the sequence is provided as part of the input. The numbers are positive integers in the range 0..1,000,000. You need to normalize these values to the range that you need in your program. Use a global index into this array and increment it (avoiding concurrent reads or writes) each time a random number is requested, possibly wrapping around if you run out of random numbers.

Number of TCUs Although the number of TCUs on a given architecture is fixed (e.g. 1024 or 64), for the purpose of this assignment we can scale down this number to allow easier testing and debugging. The number of available TCUs will be provided as part of the input for each dataset.

Measuring Work and Depth to get the most meaningful measurement of work and depth you will need to measure them for every step of the algorithm. That doesn't mean that you need to measure them for every statement that you write in your program, but rather in a place representative of the work and depth of that step in the algorithm. For example, step 4 above will have an $O(N/NTCU)$ time, and $O(N)$ work. so the result should be something of the same performance bounds.

4 Assignment

1. **Parallel Sort:** You are required to submit the solution in both ICE and XMTC.
   - Write a parallel ICE program ssort.ice.cpp that Implements the described algorithm. Also provide a WD analysis of the Algorithm described and measure the work and depth of your implementation. Compare the results to your analysis.

- Write a parallel XMTC program ssort.p.c that implements the Shared Memory Sample Sort algorithm. This implementation should be as fast as possible.

2. **Serial Sort:** Write a serial XMTC program ssort.s.c that implements a serial sorting algorithm. This implementation will be used to for speedup comparison. You can use one of the serial sorting algorithms implemented as part of sample sort, or you can write a different sorting algorithm.

No template files are provided with the homework distribution. You will create these files under the *src* directory.

4.1 Setting up the environment

Log in to your account in the class server, copy the *ssort.tgz* file from */opt/xmt/class/xmtdata/* directory and extract it using the following commands:

```
$ cp /opt/xmt/class/xmtdata/ssort.tgz ~
$ tar xzvf ssort.tgz
```

This will create the directory *ssort* with *src* and *doc* folders. Put your *c* files in *src*, and *txt* files to *doc*.

Data files are located at a common location in the server (*/opt/xmt/class/xmtdata/ssort*). If you use the Makefile system explained in Section 4.4, you will not need to explicitly refer to this location. The provided Makefile utilizes command line options to pass the paths to the header and data files to the compiler.

4.2 Input Format

The input is provided as an array of integers *A*.

#define N	The number of elements to sort.
int A[N]	The array to sort.
int s	The oversampling ratio.
#define NTCU	The number of TCUs to be used for sorting.
#define NRAND	The number of random values in the RANDOM array.
int RANDOM[NRAND]	An array with pregenerated random integers.
int result[N]	To store the result of the sorting.

You can declare any number of global arrays and variables in your program as needed. The number of elements in the arrays (*n*) is declared as a constant in each dataset, and you can use it to declare auxiliary arrays. For example, this is valid XMTC code:

```
#define N 16384

int temp1[16384];
int temp2[2*N];
int pointer;

int main() {
 //...
}
```

4.3 Data sets

The following datasets are provided for you to use with both your ICE and XMTC programs:

Dataset	N	NTCU	Header File	Binary file
d1	256	8	d1/ssort.h	d1/ssort.xbo
d2	4096	8	d2/ssort.h	d2/ssort.xbo
d3	128k	64	d3/ssort.h	d3/ssort.xbo

The paths are given with respect to */opt/xmt/class/xmtdata/ssort*, however you will not need the explicit path unless you do not use the Makefile system. A data set can be chosen by passing a DATA argument to the Makefile. See Section 4.4 for examples.
You will still need to provide the entire path when using them with ICE.

4.4 Compiling and Executing

For your convenience, a Makefile is provided with the homework distribution. You can use the provided makefile system to compile and run your XMTC programs. To run the parallel SSORT on the d1 data set, use the following command in the src directory:

```
> make run INPUT=ssort.p.c DATA=d1
```

This command will compile and run the ssort.p.c program with the d1 data set. For other programs and data sets, change the the name of the input file and the data set.

If you need to just compile the input file (no run):

```
> make compile INPUT=ssort.p.c DATA=d1
```

You can get help on available commands with

```
> make help
```

Note that, you can still use the `xmtcc` and `xmtfpga` commands as in the earlier assignments. You can run with the makefile system first to see the commands and copy them to command line to run manually. In case of the example we used above, the commands will look like:

```
>  xmtcc -include ${DPTH}/d1/ssort.h ${DPTH}/d1/ssort.xbo ssort.p.c -o ssort.p
```

where $*DPTH* is defined as */opt/xmt/class/xmtdata/ssort*. If the program compiles correctly a file called `ssort.p.b` will be created. This is the binary executable you will run on the FPGA using the following command:

```
>  xmtfpga ssort.p.b
```

Please use the *ICE.py* script to compile your ICE program. So in case of the example used above, the command will look like this:

```
>  ICE.py -include ${DPTH}/d1/ssort.h -XBO ${DPTH}/d1/ssort.xbo -i ssort.ice.cpp -o ssort.p
```

If successful the binary executable `ssort.p.b` will be created. You can run that on the FPGA using the same command above.

5 Output

The array has to be sorted in **increasing** order. The array **result** should hold the array of sorted values.

Prepare and fill the following table: Create a text file named table.txt in doc. **Remove any *printf* statements from your code while taking these measurements.** Printf statements increase the clock count. Therefore the measurements with printf statements may not reflect the actual time and work done.

Dataset	**d1**	**d2**	**d3**
Parallel sort clock cycles			
Serial sort clock cycles			

Note that a part of your grading criteria is the performance of your parallel implementation on the largest dataset (d3). Therefore you should try to obtain the fastest running parallel program. As a guideline, for the larger dataset (d3) our Serial Sort runs in 45526102 cycles, and our Parallel Sample runs in 9047152 cycles (speedup $\sim$5x) on the FPGA computer.

Finally, please create a text file named analysis.txt in doc. This file should include the WD analysis of the algorithm, and the WD measurements result that you did in the ICE program. Please show how the measurements you made relate to the WD analysis of the algorithm for the different Datasets.

5.1 Submission

The use of the make utility for submission *make submit* is required. Make sure that you have the correct files at correct locations (*src* and *doc* directories) using the make submitcheck command. Run following commands to submit the assignment:

```
$ make submitcheck
$ make submit
```

5.2 Discussion about Serial Sorting Algorithms

In this assignment you need a serial sorting algorithm in three different places. First when you implement the serial sorting itself to compare against your implementation, but also within the sample sort algorithm, first to sort the array of samples S and later to sort in parallel the p segments. So choosing the right serial sorting algorithm is very important. The discussion below should guide you and limit your search space when looking for the best serial algorithms to use with sample sort.

In Table 1 the performance of four serial sorting algorithms is compared as well as the performance of sample sort using some combinations of these algorithms. The serial algorithms are *quicksort* (QS), *heapsort* (HS), *bubble sort*(BS) and *bubble sort with termination check*(BS+check)[1]. The notation *"Sample Sort(XX/YY)"* indicates the parallel sample sort algorithm using the serial sorting algorithm *XX* in Step 1 to sort array S and the serial sorting algorithm *YY* in Step 5.

The Table shows that the fastest serial algorithm of the ones compared is quicksort, heapsort comes second, and bubblesort is too slow to get a cycle count for the largest dataset. Quicksort however is a recursive algorithm, naturally implemented using recursive function calls. For that reason it was not used for Step 5 (the QS/QS configuration was not implemented) since function calls are currently not supported in parallel code. Students have been able to implement a non-recursive version of quicksort to use in Step 5 which gave improved performance.

[1] The algorithm checks after each of the N passes of the input array $A[N]$ if there were any swaps. If not it terminates earlier.

Table 1: Table of cycle counts for different serial sorting algorithms and sample sort using different sorting algorithms

Dataset	d1	d2	d3
Serial(QS)	50302	1002756	45526102
Serial(HS)	64376	1562200	103129058
Serial(BS)	327350	96523199	timeout
Serial(BS+check)	340158	100349982	timeout
Sample Sort (QS/HS)	59011	1501593	9047152
Sample Sort (HS/HS)	59819	1502359	9101561
Sample Sort (QS/BS)	150381	83490620	timeout