HW4: Shared-Memory Sample Sort

Course:	ENEE159V/H, Spring 2009
Title:	Shared-Memory Sample Sort
Date Assigned:	March 26th, 2009
Date Due:	April 15th, 2009 **3:30pm**

1 Assignment Goal

The goal of this assignment is to provide a randomized sorting algorithm that runs efficiently on XMT. 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**) $set[i]$, $0 \le i < p$. The sets are $set[0] = \{A[i] \mid A[i] < S'[0]\}$, $set[1] = \{A[i] \mid S'[0] < A[i] < S'[1]\}$, $\ldots$, $set[p-1] = \{A[i] \mid S'[p-1] < A[i]\}$.

Step 2. Consider the input array A divided into p subarrays, $B[0] = A[0,\ldots,N/p-1]$, $B[1] = A[N/p,\ldots,2N/p-1]$ 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:

- $c[i][j]$ - the number of elements from $B[i]$ that belong in partition $set[j]$. The $c[i][j]$ makes up the matrix C as in figure 1.

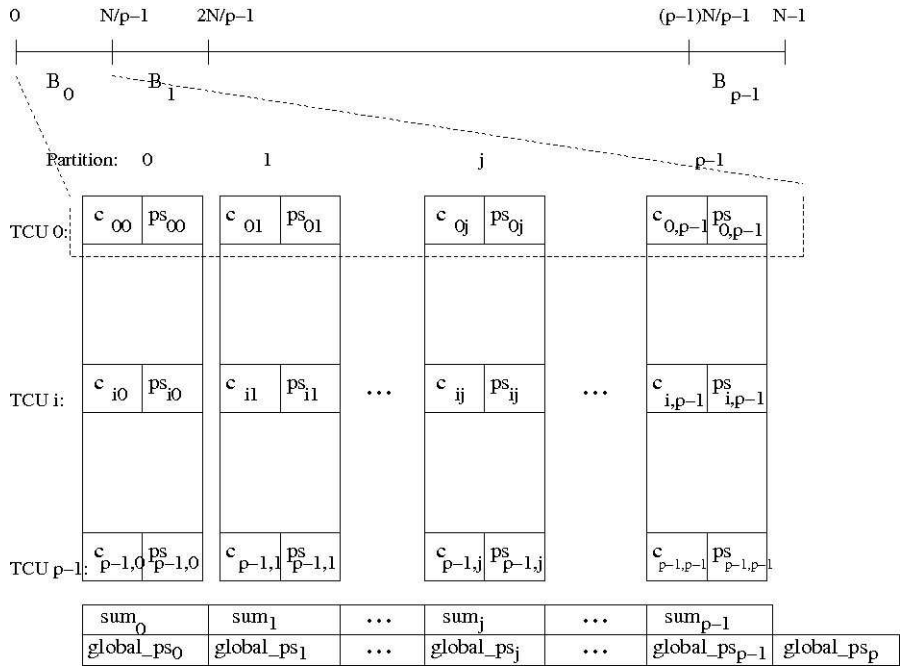

Figure 1: The C matrix built in Step 2.

- $partition[k]$ - 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 3.1 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 $sum[i]$.

Hint: For convenience, you can use a serial prefix-sum algorithm for each column, and start them all in parallel. Note that the ordering of the prefix-sum values is important, and you cannot use the XMT `ps()` or `psm()` instructions.

Step 3.2 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 conveninence.

Hint: You can also use a serial prefix-sum algorithm here. Since the number of input elements is small (equal with the number of processors p), it is not worth using a parallel prefix-sum algorithm, such as the one in the class notes.

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

$$pos[j] = global\_ps[partition[j]] + ps[i][partition[j]] + serial[j]$$

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.

Testing for correctness For the larger data-sets, it is impractical to test the correctness of your algorithm by printing all the elements of the result. Instead, you can add a testing step at the very end of the implementation which simply iterates through all the elements in the `result` array and tests that they are in increasing order. Make sure to remove or comment out this test before you submit your program or collect cycle counts, since it will significantly affect the performance of your program.

Register spills There is currently an issue on XMT which occurs when the body of a spawn block exceeds a certain complexity. Please refer to Appendix A for more information on how to deal with this problem, if you encounter it while solving this assignment.

4 Assignment

1. **Parallel Sort:** 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 of your choice. 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. This implementation should be as fast as possible.

4.1 Setting up the environment

The header files and the binary files can be downloaded from the web using the following commands:

```
$ wget http://terpconnect.umd.edu/~jspeiser/ssort.tgz
$ tar xzvf ssort.tgz
```

This will create the directory *ssort* with following folders: *data, src*, and *doc*. Data files are available in data directory. Put your *c* files to *src*, and *txt* files to *doc*.

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

Run all your programs (serial and parallel) using the data files given in the following table. You can directly include the header file into your XMTC code with *#include* or you can include the header file with the compile option *-include*. To run the compiled program you will need to specify the binary data with *–data-file* option.

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

5 Output

The array has to be sorted in **increasing** order. The array **result** should hold the array of sorted values *for both the serial and parallel solutions*.

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			

5.1 Submission

NOTE: When performing the archiving, do not include the entire ssort directory, just archive the src and doc folders. Run the following commands to submit the assignment:

```
$ tar czvf selection.tgz doc/ src/
```

A Avoiding register spills in XMTC

The following restriction applies when programming in XMTC at this time.

Currently the only local storage available to threads is in the TCU registers. Therefore, when programming in XMTC, special care has to be taken not to overflow the capacity of this storage. Registers are used to store local variables and temporary values. The compiler does a series of optimizations to fit everything into registers, but in some cases when a parallel section is long and complex, it fails to do so and additional storage is required.

At the present time, if the compiler detects such a situation, compilation will fail with the error message: `"Register spill detected in spawn block. Aborting compilation."`

The solution is to split the spawn block into shorter, simpler parallel sections for which the registers provide enough storage. At the present time, if you get an error message from the compiler regarding register spills, you will have to change the code by splitting the spawn sections yourself. There is no general recipe for this, you will have to use your knowledge of the application to chose how to change the code.

Here is a simple example. In the code in the left column below, the value x is used at the beginning and the end of the tread, but not in the middle. However, this usually requires a register to be allocated to x and reserved throughout the whole parallel section. This increases the *register pressure* and might lead to a register spill, if the `code1` and `code2` sections are complex and require using local registers as well.

An **immediate possible solution** is presented in the righthand column below: the parallel section is split into two, and x is re-assigned closer to the end, thus reducing the register pressure and possibly avoiding a register spill.

```
Initial code
High register pressure

spawn(low, high) {
    int i, x = A[$];
    for (i=0; i<5; i++) {
        B[$+i] = x;
        // .. code 1 .. //
    }
    for (i=0;i<5;i++) {
        // .. code 2 .. //
    }
    C[$] = x;
}
```

```
Transformed code
Register pressure is lower

spawn(low, high) {
    int i, x = A[$];
    for (i=0; i<5; i++) {
        B[$+i] = x;
        // .. code 1 .. //
    }
} // join

spawn(low,high) {
    int i, x;
    for (i=0;i<5;i++) {
        // .. code 2 .. //
    }
    x = A[$];
    C[$] = x;
} // join
```

A **medium-term solution**, which is currently under development, is to use a parallel stack, stored in shared memory. However, there is a performance issue with this solution: storing and retrieving values from shared memory is much slower than the registers, and can significantly affect running time of the parallel section (for example if the memory access occurs in a loop).

The **long term ideal solution** will include the following ingredients:

- increasing the number of registers available

- adding some type of local memory to the TCUs (e.g. cluster buffers or scratch-pads) and retargetting register spills to them (instead of shared memory)
- have the compiler perform spawn block splitting (as showed above) to minimize using the stack and generate the optimal code without the programmer's assistance
- use data prefetching mechanisms to reduce the penalty of a register spill to memory.