HW5: Shared-Memory Sample Sort

Course:	Informal Parallel Programming Course for High School Students, Fall 2007
Title:	Shared-Memory Sample Sort
Date Assigned:	October 30, 2007
Date Due:	November 13, 2007

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, $0 \leq i < p$. 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_{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.

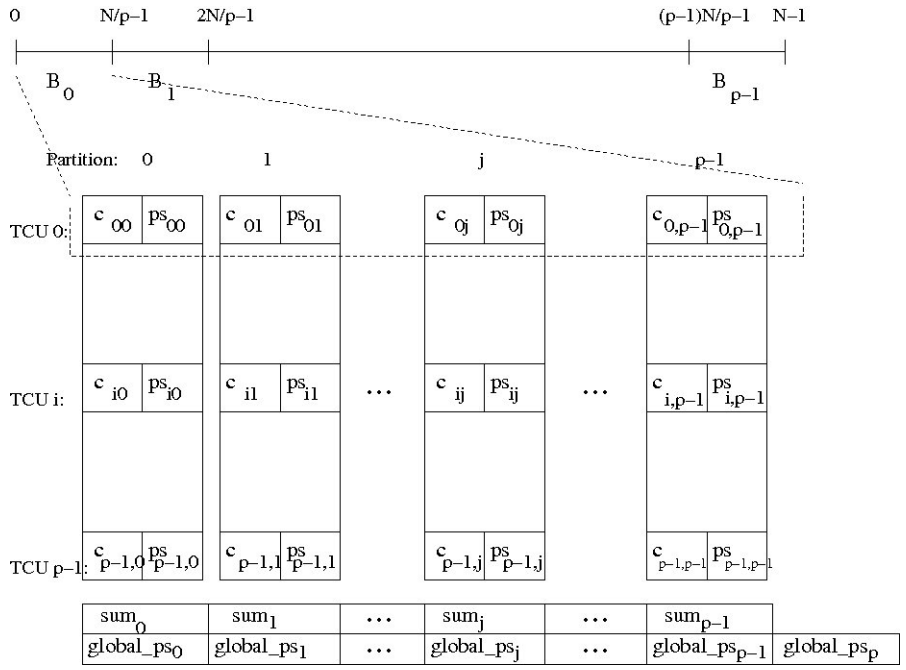

Figure 1: The C matrix built in Step 2.

- $partition_k$ - 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. 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 conveninence.

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 partioning 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.

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 the Shared Memory Sample Sort 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.

4.1 Setting up the environment

The header files and the binary files can be downloaded from $\sim$ *swatson/xmtdata*. To get the data files, log in to your account in the class server and copy the *ssort.tgz* file from directory using the following commands:

```
$ cp ~swatson/xmtdata/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.32b
d2	1024	8	data/d2/ssort.h	data/d2/ssort.32b
d3	4096	8	data/d3/ssort.h	data/d3/ssort.32b
d4	4096	64	data/d4/ssort.h	data/d4/ssort.32b
d5	16k	64	data/d5/ssort.h	data/d5/ssort.32b

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**	**d4**	**d5**
Parallel sort clock cycles					
Serial sort clock cycles					

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
```

If you have any questions, please send an e-mail to Scott Watson, *swatson@umd.edu*