# HW2: Integer Sort & Radix Sort

| | |
|---|---|
| **Course:** | CMSC751/ENEE759, Spring 2009 |
| **Title:** | Integer Sort and Radix Sort |
| **Date Assigned:** | February 23th, 2009 |
| **Date Due:** | Tuesday March 10th, 2009 |
| **Contact:** | Alex Tzannes - tzannes@cs.umd.edu |

## 1 Assignment Goal

The goal of this assignment is to implement the *Integer Sorting* and *Radix Sort* algorithms presented in the class notes (Section 6) in XMTC and run it on the XMT FPGA. The input is an array A[1..n] of integers in the range $[0..r-1]$.

For *Integer Sort* we will assume that $n$ is divisible by $r$ and $r = \sqrt{n}$. The goal is to rank the elements of the input array from smallest to largest.

For *Radix Sort* we will assume that $r = n^2$. The goal is to sort the elements of the input array from smallest to largest.

## 2 Description of Integer Sort

Please be aware that this description is slightly different from that in the class notes.
**Step 1.** Partition A into n/r subarrays: $B_0 = A[0..r-1]$, $B_1 = A[r..2r-1]$,..., $B_{n/r-1} = A[n-r..n-1]$. Compute the following two sets of values:

1. $number(v,s)$: the number of elements of subarray $B_s$ that have value $v$
2. $serial(i)$: the number of elements $A(j)$ such that $A(j) = A(i)$ and precede element $i$ in its subarray $B_s$ (i.e., $serial(i)$ counts only $j < i$, where $\lfloor j/r \rfloor = \lfloor i/r \rfloor = s$), for $1 \le i \le n$.

NOTE: This step is slightly different than what is presented in the class notes since it doesn't require you to sort the subarrays.

Example: $B_1 = (2,3,2,2)$ (r=4). Then, $number(2,1) = 3$, and $serial(3) = 1$.

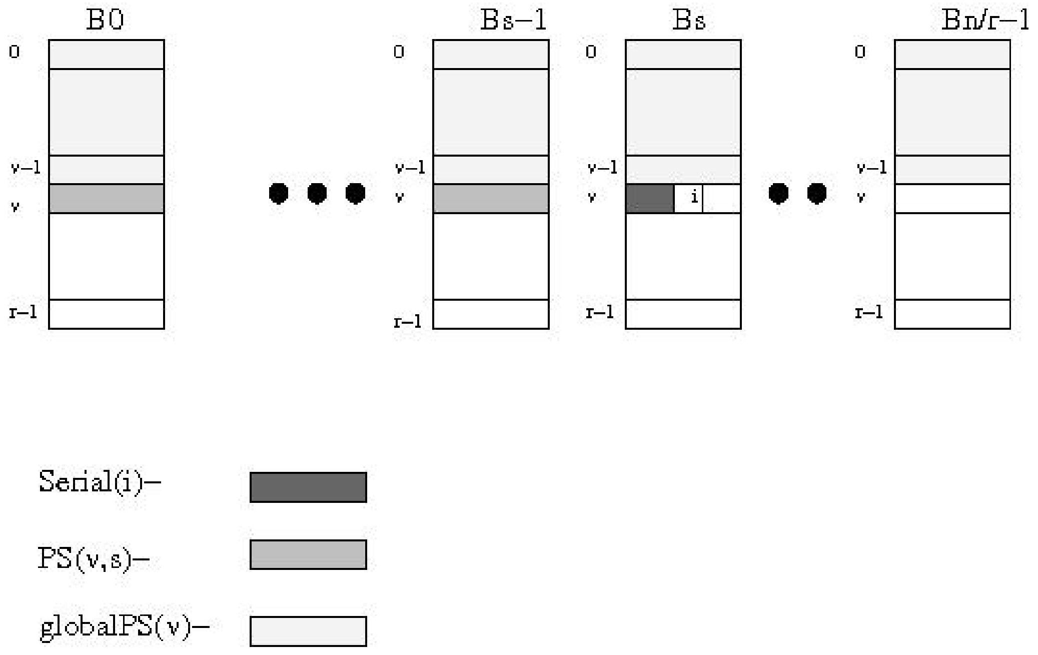
**Step 2.** Separately (and in parallel) for each value $0 \le v \le r-1$ compute the prefix-sums of $number(v,0)$, $number(v,1)$ .. $number(v,n/r-1)$ into $ps(v,0)$, $ps(v,1) \ldots ps(v,n/r-1)$, and their sum (the number of elements whose value is v) into $cardinality(v)$. For this algorithm, the definition of prefix sum at position $x$ is the sum of elements $[0..x-1]$, i.e., $ps[x] = \sum_{i=0}^{x-1} A[i]$, and $ps[0] = 0$. Therefore $ps(v,0) = 0$ and $globalPS(0) = 0$ in step 3. Also note that $ps$ is an XMTC keyword, so use a different name for your array when you code this step.

**Step 3.** Compute the prefix sums of $cardinality(0)$, $cardinality(1) \ldots cardinality(r-1)$ into $globalPS(0)$, $globalPS(1) \ldots globalPS(r-1)$.

**Step 4.** In parallel for every element i ($0 \le i \le n-1$), compute its rank:

$$rank(i) = serial(i) + ps(v,s) + globalPS(v)$$

where $v = A(i)$ and $B_s$ the subarray of element $i$ ($s = \lfloor i/r \rfloor$)



## 3 Description of Radix Sort

Radix sort uses integer sort as its base so it is advisable to tackle it after you finish integer sort. As mentioned in the introduction the range $r$ of values for radix-sort is larger than the number of elements to sort $n$ and for that reason using a parallel integer sort or a serial bucket sort is inefficient. Radix sort overcomes the issue of a larger range by sorting $k$ times, each time taking into account only a portion of the actual value of the input elements.

Imagine having the binary representation (since we are working on computers) of a value $v$ of the input ($v \in [0..r-1]$). It is $b = \log_2 r$ bits long. Now break it up in $k$ segments, each $l = b/k$ bits long (assume that $b$ is divisible by $k$). Radix sort works in $k$ rounds sorting the input elements not using their value, but the value of their $i^{th}$ segment, starting from the one that contains the least significant bit, and moving up towards the most significant bit.

As an example we present the case where k=2:

**Step 1.** Apply the integer sorting algorithm to sort the input array $A$ using
$A(1)(mod\sqrt{r}), A(2)(mod\sqrt{r}), \ldots, A(n)(mod\sqrt{r})$ as keys.
If the computed rank of element $i$ is $j$ then $B(j) := A(i)$.

**Step 2.** Apply the integer sorting algorithm again, now on array $B$ using
$\lfloor B(1)/\sqrt{r} \rfloor$, $\lfloor B(2)/\sqrt{r} \rfloor$, $\ldots$, $\lfloor B(n)/\sqrt{r} \rfloor$ as keys.

## 4 Assignment

1. Serial Sort: Implement a stable (order preserving) serial sort in XMTC (a template program `isort.s.c` is provided). While you can choose any algorithm you want, you probably want to implement an order preserving bucket sort, since it will be used as part of the parallel sort. The serial implementation will be used to calculate the speedup achieved by the parallel implementation.

2. Parallel Integer Sort: Implement the parallel integer sorting algorithm described above (and in your class notes) in XMTC (a template program `isort.p.c` is provided).

3. Serial Radix Sort: Implement serial radix sort in XMTC with $k = 2$. Save your program as `rsort.s.c`

4. Parallel Radix Sort: Implement parallel radix sort in XMTC. Start with $k = 2$, see why that choice of $k$, while adequate for the serial version, is inadequate for the parallel version, and implement it for $k = 4$ as well. Save your program as `rsort.p.c`

## 4.1 Setting up the environment

The header files and the binary files can be downloaded from */opt/xmt/class/xmtdata/.* To get the data files, log in to your account in the class server and copy the *isort.tgz* file using the following commands:

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

This will create the directory *isort* with following folders: *data, src*, and *doc*. Data files are available in data directory. Edit the *c* files in *src*, and the *txt* file in *doc*.

## 4.2 Input Format for Integer Sort

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

| | |
|---|---|
| #define N | The number of elements to sort. |
| #define R | The number different values. Values will be in $[0..R-1]$. |
| #define NbyR | The value of $N/R$. Also the number of sub-arrays in Step 1. |
| int A[N] | The array to sort. |
| int rank[N] | To store the resulting ranks. |

You can declare any number of global arrays and variables in your program as needed. The number of elements in the arrays (*N*), the number of values (*R*) and their quotient (*NbyR*) are declared as constants in each dataset, and you can use them to declare auxiliary arrays. For example, this is valid XMTC code:

```
int serial[N];
int prefixSum[R][NbyR];

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

## 4.3 Input Format for Radix Sort

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

| | |
|---|---|
| #define N | The number of elements to sort. |
| #define R | The number different values. Values will be in $[0..R-1]$. |
| int A[N] | The array to sort. |
| int result[N] | To store the sorted elements. |

### 4.4 Data sets for Integer Sort

Both the serial and parallel versions of your program will be 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 compiler option *-include*.

| Dataset | N | R | Header File | Binary file |
|---|---|---|---|---|
| d1 | 256 | 16 | data/d1/isort.h | data/d1/isort.xbo |
| d2 | 4096 | 64 | data/d2/isort.h | data/d2/isort.xbo |
| d3 | 64k | 256 | data/d3/isort.h | data/d3/isort.xbo |

### 4.5 Data sets for Radix Sort

Both the serial and parallel versions of your program will be 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*.

| Dataset | N | R | Header File | Binary file |
|---|---|---|---|---|
| d1 | 256 | 64K | data/d1/rsort.h | data/d1/rsort.xbo |
| d2 | 4096 | 16M | data/d2/rsort.h | data/d2/rsort.xbo |
| d3 | 64k | 4G | data/d3/rsort.h | data/d3/rsort.xbo |

Note that the large dataset has values from $[0..4G-1] = [0..2^{32}-1]$ but since an integer only has 32 bits and it is signed the values are really only in $[0..2^{31}-1]$.

### 4.6 Compiling and Executing

You can compile the parallel program using the following command line for the small dataset (d1):

```
>  xmtcc -include ../data/d1/isort.h ../data/d1/isort.xbo isort.p.c -o isort.p
```

If the program compiles correctly a file called `isort.p.b` will be created. This is the binary executable you will run on the FPGA using the following command:

```
>  xmtfpga isort.p.b
```

If you wish to have the rank printed on the screen by the program compile using the `-D PRINT_RESULT` flag. For larger datasets we will provide a textual memory dump of the rank array. To compare your results against them use the following command when running the program:

```
> xmtfpga isort.p.b --memdump dump --dumpvar rank
```

After the execution, a file called `dump` will be left in the directory and you can compare it using *diff* to the provided correct solution.

Adapt the above commands to compile the radix sort programs as well.

### 4.7 Debugging

In order to test you integer sort implementation you need to write a procedure that runs two simple tests on the output of the sorting program, the `rank` array. The tests are:

1. Check that the ranks in the rank array appear exactly once and are in the range of $[0..N-1]$.
2. Check that ranking produces an array that is indeed sorted in ascending order.

Your testing routines should be derived from the most efficient PRAM algorithms you can design. Please report in *table.txt* (see Section 4.8) the parallel complexity of these algorithms.

Provide pseudo-code for your checks at the end of the `table.txt` (see Section 4.8 below) as well as their work and time complexity for full credit.

## 4.8 Output

For integer sort, the input array $A$ has to be ranked in *increasing* order, as described in Section 2. The ranking should be order-preserving (i.e. if $A[i] = A[j]$, and $i < j$ then $rank[i] < rank[j]$) and it should be stored in array *rank*.

For razix sort, the input array $A$ has to be sorted in *increasing* order, into array *result*. Note that the output of the two algorithms has slightly different format: for the first we ask the ranks, while for the second the actual sorted array.

Fill-in the text file called `table.txt` in the `doc` directory. **Remember to remove any `printf` statements from your code before taking measurements, as well as any checking code.** `Printf` statements and checking code (see Section 4.7) increase the clock count. Therefore the measurements with printf statements may not reflect the actual time and work done.

Note that a part of your grading criteria is the performance of your parallel implementation on the largest dataset (d3) for both integer-sort and radix-sort. Therefore you should try to obtain the fastest running parallel program. As a guideline, for the larger dataset (d3) our Serial Integer Sort runs in 6877346 cycles, and our Parallel Integer Sort runs in 777748 cycles (speedup ~8.8x) on the FPGA computer. Also our Serial Radix-Sort with $k = 2$ runs in 64659107 cycles and our Parallel Radix-Sort with $k = 4$ runs in 3563886 cycles (a speedup of $\sim 18x$).

| **Dataset** | **d1** | **d2** | **d3** |
|---|---|---|---|
| Parallel isort clock cycles | | | |
| Serial isort clock cycles | | | |

| **Dataset** | **d1** | **d2** | **d3** |
|---|---|---|---|
| Parallel rsort clock cycles | | | |
| Serial rsort clock cycles | | | |

| **Check** | **Time Complexity** | **Work Complexity** |
|---|---|---|
| Each rank $i \in [0..N-1]$ appears exactly once | | |
| The resulting array is indeed sorted | | |

## 4.9 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
```