HW5: Merge Sort

Course:	ENEE159S
Title:	Merge Sort
Date Assigned:	April 18th, 2009
Date Due:	May 8th, 2009 5:00pm

1 Assignment Goal

The final goal of this assignment is to implement the parallel *merge-sort* algorithm that is introduced in the Section 4.2 of the class notes in XMTC and run it on the XMT simulator. Your parallel algorithm should be as fast as possible.

As the first step of this assignment you will be asked to implement a serial and a parallel solution to the merging problem. The second step will be implementing the merge-sort algorithm using the merge subroutines from the first step as the building blocks. Finally, the third step will be fine-tuning your merge-sort implementation for performance.

2 Description of the Merging Problem

You are given two integer arrays of equal size, $X = X(0),...,X(k-1)$ and $Y = Y(0),...,Y(k-1)$. The arrays are monotonically non-decreasing. The objective is to map each of these elements into an array $Z = Z(0),...,Z(2k-1)$ which is also monotonically non-decreasing. $\log k$ is an integer. You are to implement a serial and a parallel solution to the merging problem based on Section 4.1 of the class notes. For the parallel solution use the least number of `spawn` commands possible (see Exercise 9 in the same section).

Section 4.1 presents two parallel solutions to the merging problem: the "surplus-log" parallel algorithm and the parallel algorithm for ranking. Both algorithms run in $O(logn)$ time on an ideal PRAM. On the other hand, the "surplus-log" algorithm takes a total of $O(nlogn)$ work as opposed to the $O(n)$ work of the parallel algorithm. Remember that, the XMT computer is not an ideal PRAM and it features a limited number of cores (64). This means that work measure will reflect on the total execution time and you should choose the algorithm that performs best not only with respect to the time measure but also the work measure.

Note that the description of the problem here is more general than the definition in the class notes. Specifically, $k/\log k$ may not be an integer. Elements of X and Y are still defined to be unique and pairwise distinct.

3 Description of the Merge-Sort Algorithm

You are given an integer array of n elements $A = A(0),...,A(n-1)$ where $n = 2^l$ for some integer $l \geqslant 0$. Elements of A are unique. The objective is to reorder A into an array $B = B(0) \leqslant B(1) \leqslant ... \leqslant B(n-1)$ via the *merge-sort* algorithm given in Section 4.2 of the class notes.

The merge-sort algorithm is presented recursively in the class notes, however, it can also be implemented non-recursively. In fact, writing parallel recursion with the current XMT compiler is not very convenient[1]. An alternative is to use the "balanced binary tree" form in Algorithm 1. In the pseudocode notation of Algorithm 1, $C_{x,y}$ is a sub-array of an array C, such that $C_{x,y} = C(x \cdot (y-1)), ..., C(x \cdot y - 1)$ and $A \leftrightarrow B$ is a simple pointer swap operation in the C-language[2]. Pointer swap is used to interchange the elements of the arrays A and B for the next iteration of the outer loop. Figure 1 visually demonstrates one iteration of this algorithm for $h = 2$ and $i = 1$ on an example[3] that sorts an input array of $A = (3,7,4,5,2,8,1,6)$.

Algorithm 1 Non-recursive merge-sort algorithm.

1: **for** $h := 1$ **to** $\log n$ **do**
2: **for** $i := 1$ **to** $n/2^h$ **do**
3: $B_{2^h,i} \leftarrow MERGE(A_{2^{h-1},2i-1}, A_{2^{h-1},2i})$
4: **end for**
5: $A \leftrightarrow B$
6: **end for**
7: $A \leftrightarrow B$

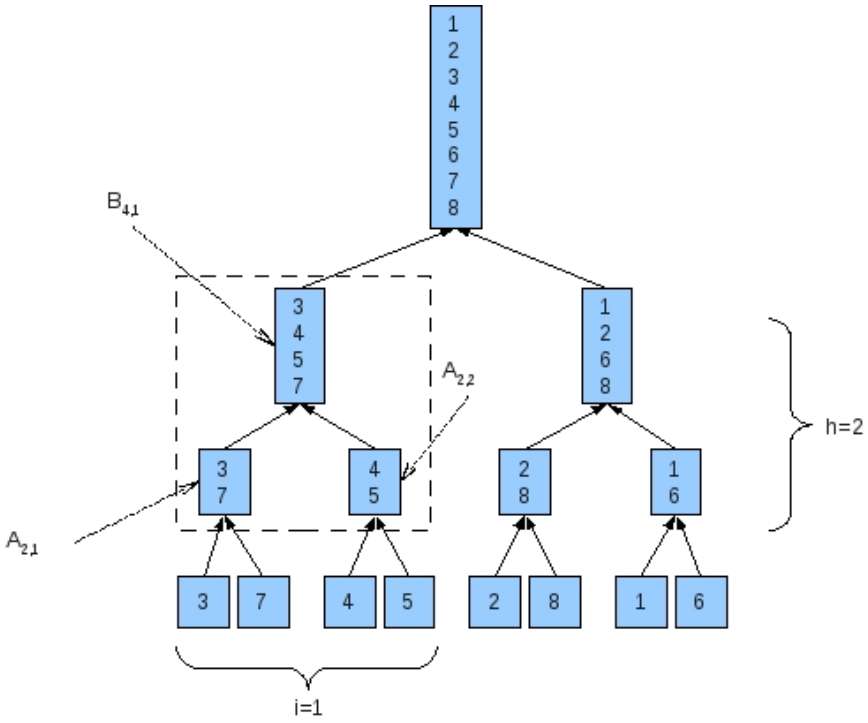


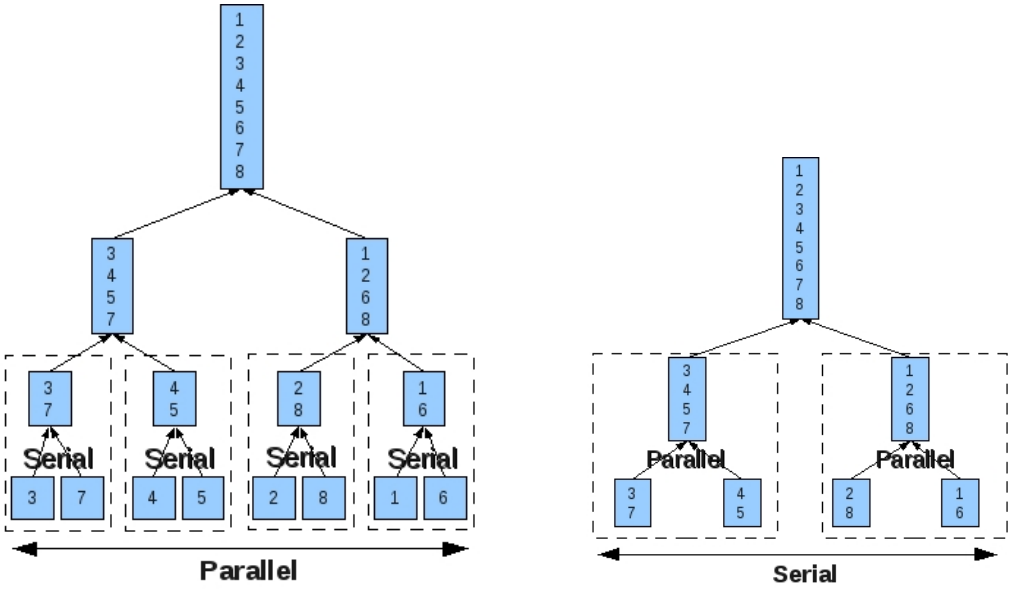
Figure 1: Example of the merge-sort algorithm, non-recursive form.

There are two potential sources of parallelism in the merge-sort algorithm: the *for* loop at the line 2 of the algorithm above, which is merging pairs within a level of the binary tree, and the merging itself (line 3). Even though these types can be applied concurrently, in this assignment you are restricted to using only one type for each level (i.e. for a fixed value of h). On the other hand, you are free to switch the type of parallelism you are using for different levels. Consider the first iteration of the outer loop

[1] Remember that currently the XMT compiler does not support nested spawns or function calls within a parallel section. Nesting is still possible with the `sspawn` construct, which you are not allowed to use in this assignment.

[2] Depending on the implementation one array copy operation might be required, which can be accomplished in $O(n)$ work and $O(1)$ time.

[3] Same example is given in the Figure 10 of the class notes.



(a) (b)

Figure 2: (a) For $h = 1$, serial merge operations done in parallel, (b) For $h = 2$, parallel merge operations done serially.

($h = 1$) for the example above, given in Figure 2a. The merge operations (denoted by the dashed boxes) can be performed in parallel, but then each merge operation has to be done serially. Conversely, for the second iteration of the outer loop (Figure 2b), *parallel-merge* can be used to merge the arrays however the merge operations have to be performed one at a time.

Performance considerations are the reason for switching between the two types of parallism. Closer to the leaves of the binary tree, the sub-arrays are relatively smaller in size, but there are many of them to be merged. Therefore it will be more efficient to use a serial merge algorithm and assign each merge to a parallel thread. On the other hand, towards the root of the binary tree, size of the sub-arrays grows and the number of nodes to be processed diminishes. In this case, one should consider switching to using parallel-merge but handle the tree nodes serially. The cross-over point usually depends on factors such as the parameters of the underlying architecture, how you structure your code, etc. and can be found empirically through a number of trials. It is a part of your assignment to find the point that gives the best performance.

4 Assignment

For the first three items below, provide a source file under the *src* directory. A template file for each item is already provided at the same location. The last item, performance graph, should be placed under the root of the assignment package (parent of the *src* directory).

1. Serial merge – `merge.s.c`: Implement a serial routine for the merge problem.
2. Parallel merge – `merge.p.c`: Implement a parallel routine for the merge problem.
3. Merge-sort – `mergesort.c`: Implement the merge-sort algorithm based on the description in Section 3. Your implementation should choose between the two types of parallelism depending on the level of the binary tree it is operating on (i.e. the value of h in Algorithm 1) and use the serial and parallel merge as building blocks.
4. Fine-tuning merge-sort: You should find the optimal cross-over point for your algorithm via exhaustive search. Cross-over point is defined as the smallest value of h (in Algorithm 1) for which

you use the parallel merge subroutine. The objective function for your search is the minimum execution time. You are supposed to report this value in `mergesort.c`. For instructions see the comments in the file.

5. Cross-over performance graph – `crossover.pdf`: Run `mergesort.c` over the largest data set (dms3) for various cross-over values (you can use the `make crossover` command, see Section 4.6). Record the cycle counts reported by the simulator for every run. Plot cross-over value vs. cycle counts in `crossover.pdf`. Mark the global minimum on the graph. The graph should justify your choice of the cross-over value in the previous item. `crossover.pdf` should be placed under the root of the assignment package (parent of the *src* directory).

Note that XMTC compiler does not allow function calls in parallel sections. This means you cannot call the serial merge sub-routine from inside the merge-sort algorithm. You will have to manually copy-paste the code instead.

4.1 Setting up the environment

The header files and the binary files are packed in *mergesort.tgz*. Extract this file using the following command:

```
$ tar xzvf mergesort.tgz
```

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

4.2 Data format for Serial and Parallel Merge Algorithms

The input and output are defined as arrays of integers, all less than $2^{31} - 1$.

#define k	The number of elements in X and Y arrays.
#define logk	Logarithm of k in base 2.
int X[k]	The first input array.
int Y[k]	The second input array.
int R[2k]	The output array.

You can declare any number of global arrays and variables in your program as needed. X and Y are monotonically non-decreasing arrays and $\log k$ is an integer. The output should be stored in R. The values in X and Y need not to be preserved after the execution.

4.3 Data format for Parallel Merge-Sort Algorithm

The input and output are defined as arrays of integers, all less than $2^{31} - 1$. An `INF` macro is defined in the `mergesort.c` file for convenience.

#define n	The number of elements in the input and output arrays.
#define logn	Logarithm of n in base 2.
int A[k]	The input array.
int R[k]	The output array.

You can declare any number of global arrays and variables in your program as needed. $\log k$ is an integer. The output should be stored in R. The values in A need not to be preserved after the execution.

4.4 Data sets for Serial and Parallel Merge Algorithms

You should run your program using the following data sets including the header file with the compiler option *-include*. You can also use the *#include* directive inside the XMTC file, however remove these directives before submitting your code.

Data Set	k	Header File	Binary file
dm1	128	data/dm1/merge.h	data/dms1/merge.xbo
dm2	2048	data/dm2/merge.h	data/dms2/merge.xbo
dm3	64K	data/dm3/merge.h	data/dms3/merge.xbo

Each data directory includes a correctout.txt file that contains the correct output for the matching input.

4.5 Data sets for Parallel Merge-Sort Algorithm

You should run your program using the following data sets including the header file with the compiler option *-include*. You can also use the *#include* directive inside the XMTC file, however remove these directives before submitting your code.

Data Set	n	Header File	Binary file
dms1	256	data/dms1/mergesort.h	data/dms1/mergesort.xbo
dms2	4096	data/dms2/mergesort.h	data/dms2/mergesort.xbo
dms3	128K	data/dms3/mergesort.h	data/dms3/mergesort.xbo

Each data directory includes a correctout.txt file that contains the correct output for the matching input.

4.6 Compiling and executing via the Makefile system

You can use the provided makefile system to compile and run your programs. For the smallest data set (dms1) merge-sort program is run as follows:

```
> make run INPUT=mergesort.c DATA=dms1
```

This command will compile and run the mergesort.c program with the dms1 data set. For other programs and data sets, change the name of the input file and the data set. You can use the `make check` command to compile and run your program and check the result for correctness (remember that each data set comes with a matching correct output).

```
> make check INPUT=mergesort.c DATA=dms1
```

As a result of this command, contents of the output array R will be dumped in a text file, R.txt, which will be automatically compared against the matching correctout.txt file. If you need to just compile the input file (no run):

```
> make compile INPUT=mergesort.c DATA=dms1
```

In order to list the simulation clock cycles for different values of the cross-over parameter (from 2 to 18) you can run the following command:

```
> make crossover
```

This will run `mergesort.c` file with the `dms3` data set. The results will be reported in `cycle.txt` file. You are supposed to graph these values in the `crossover.pdf` file.

You can get help on available commands with

```
> make help
```

Note that, you can still use the `xmtcc` and `xmtsim` commands as in the previous assignments.

4.7 Grading Criteria and Submission

In this assignment, you will be graded on the correctness of your programs (merge.s.c, merge.p.c and mergesort.c) and the performance of your merge-sort implementation in terms of completion time. Performance of your program will be compared against the performance of our reference implementation with the largest data set (dm3), which runs in $5.9M$ clock cycles. Don't forget that you are also supposed to report the cross-over point in `mergesort.c`.

Please remove any printf statements that you may have placed for debugging purposes from your code as they will affect the performance and possibly break the automated grading script. Once you have the three source files under the *src* directory you can check the correctness of your programs:

```
> make testall
```

check the validity of your submission:

```
> make submitcheck
```

and package the assignment into a tgz file for submission:

```
> make submit
```

This command will produce a compressed file (`mergesort_sln.tgz`) in the parent directory of the `src` directory.