Programming Assignment 5

Due Wednesday, December 3
(Note the change in due date)

Bitonic Sort

Bitonic sort is a somewhat unusual sorting algorithm that has the virtue that it's relatively easy to parallelize. At its heart is repeated use of the butterfly structure. The algorithm proceeds by building sequences of keys that form a *bitonic* sequence: a sequence that first increases, and then decreases. It then uses butterfly structures to sort a bitonic sequence into either a decreasing or increasing sequence.

Suppose first that we have a list of n integer keys, and n is a power of 2. Any pair of elements can be turned into either an increasing or a decreasing sequence by a "compare-swap," which is a two-element butterfly. If we have a four-element list, then we can first create a bitonic sequence with a couple of compare swaps, and then use a butterfly to create a sorted list.

As an example, suppose $n = 4$ and our input list is $\{40, 20, 10, 30\}$. Then the first two elements can be turned into an increasing sequence by a compare-swap:

```
if (list[0] > list[1]) Swap(list[0], list[1]);
```

To get a bitonic sequence, we want the last two elements to form a decreasing sequence:

```
if (list[2] < list[3]) Swap(list[2], list[3]);
```

This gives us a list that's a bitonic sequence: $\{20, 40, 30, 10\}$.

Now the bitonic sequence can be turned into a sorted, increasing, list by a four-element butterfly:

	Subscripts			
	0	1	2	3
Start: Element	20	40	30	10
1st stage: Partner	2	3	0	1
1st stage: New Element	20	10	30	40
2nd stage: Partner	1	0	3	2
2nd stage: New Element	10	20	30	40

So in order to sort a four-element list we first do two two-element butterflies (one on the first two elements and one on the last two). Then we do a four-element butterfly on the entire list. The purpose of the first two two-element butterflies is to build a bitonic sequence (a sequence that first increases and then decreases). The purpose of the four-element butterfly is to build an increasing (sorted) sequence.

So if we have an eight-element list we'll need to do

1. Four two-element butterflies. This will build one bitonic sequence from the first four elements, and another bitonic sequence from the last four elements.

2. Two four-element butterflies. This will build an eight-element bitonic sequence: the first four elements will increase, and the last four will decrease.

3. One eight-element butterfly. This will turn the eight-element bitonic sequence into an increasing sequence.

Here's an example that sorts the list $\{15, 77, 83, 86, 35, 85, 92, 93\}$. ("Elt" is an abbreviation of "element.")

	Subscripts							
	0	1	2	3	4	5	6	7
Start: Elt	15	77	83	86	35	85	92	93
Two Elt: Order	Incr		Decr		Incr		Decr	
Two Elt: Partner	1	0	3	2	5	4	7	6
Two Elt: New Elt	15	77	86	83	35	85	93	92
Four Elt: Order	Incr				Decr			
Four Elt 1: Partner	2	3	0	1	6	7	4	5
Four Elt 1: New Elt	15	77	86	83	93	92	35	85
Four Elt 2: Partner	0	1	3	2	5	4	7	6
Four Elt 2: New Elt	15	77	83	86	93	92	85	35
Eight Elt: Order	Incr							
Eight Elt 1: Partner	4	5	6	7	0	1	2	3
Eight Elt 1: New Elt	15	77	83	35	93	92	85	86
Eight Elt 2: Partner	2	3	0	1	6	7	4	5
Eight Elt 2: New Elt	15	35	83	77	85	86	93	92
Eight Elt 3: Partner	1	0	3	2	5	4	7	6
Eight Elt 3: New Elt	15	35	77	83	85	86	92	93

So the *serial* algorithm for bitonic sort chooses partners using the butterfly structure and alternates between increasing and decreasing sequences until the final butterfly is carried out, and this results in an increasing, sorted sequence.

In parallel bitonic sort, we'll have many more elements than threads. So we use a block partition, and assign n/p elements to each thread. Then each thread sorts its assigned

sublist using a fast sorting algorithm (you should use the C library `qsort` function). After the sublists are sorted, the basic structure of the serial algorithm is implemented, except that instead of compare-swaps, the basic operation is merge-split.

As an example, we'll sort the same eight-element list using four threads:

Thread	0		1		2		3	
Subscript	0	1	2	3	4	5	6	7
Start: Elt	15	77	83	86	35	85	92	93
After Local Sort: Elt	15	77	83	86	35	85	92	93
Two Thread: Order	Incr				Decr			
Two Thread: Partner	1		0		3		2	
Two Thread: New Elt	15	77	83	86	92	93	35	85
Four Thread: Order	Incr							
Four Thread 1: Partner	2		3		0		1	
Four Thread 1: New Elt	15	77	35	83	92	93	85	86
Four Thread 2: Partner	1		0		3		2	
Four Thread 2: New Elt	15	35	77	83	85	86	92	93

Programming Assignment

For programming assignment 5, you should implement parallel bitonic sort using Pthreads and a logical block partition of the list. The command line will have the form

```
$ ./pth_bitonic <thread count> <number of elements> [g] [o]
```

If the character `'g'` is on the command line, your program should use the C library functions `srandom` and `random` to generate the list. The main function should generate the list before starting any threads, and it should seed the random number generator with the int 1. The random values should be in the range 0 to 999,999.

If the `'g'` isn't on the command line, your main function should read the list from `stdin`.

If the character `'o'` is on the command line, your output should include the original unsorted list and the final sorted list.

Regardless of command line options, your program should print the total elapsed, wall-clock time for the sort. This should include the time it takes to start and terminate the threads, but it should not include the time for I/O — unless there's `DEBUG` output. (See below.) To get the timings you can use the header file `timer.h` on the class website.

You should assume that the number of threads is a power of 2, and the number of elements in the list is evenly divisible by the number of threads.

Some Details

You should store the list that's being sorted in shared memory. It may also be useful to have a shared, temporary array where the updated list is stored during a merge-split. After

completing the merge-splits in a stage of the sort, you can copy the temporary list into the original list. However, for full credit, you should swap pointers to the temporary list and the original list.

In distributed memory programs the implementation of merge-split requires that the two processes first exchange their sublists before the merges take place. In a shared memory program, the two threads can work directly with each others' sublists: no communication is necessary *until* either the new list is to be copied into the old list or pointers to the two lists are swapped. Either the copy or the swap will create a race condition. The simplest way to avoid this is to include a barrier before the copy or swap.

If you do use one or more barriers in your program, you should not use Pthreads barriers: there are a number of systems that haven't implemented these. Rather use a condition variable and a mutex as discussed in class.

In addition to the output described in the preceding section, your program should include a `DEBUG` flag that will turn on printing of the updated list after each stage of the sorting algorithm. This should include information on which butterfly is being executed (e.g., the first 2-element butterfly, the second 4-element butterfly, etc.) and which stage of the butterfly has been completed.

Finally, it may be useful to define a bitmask that can be used for determining whether the thread should be forming an increasing or a decreasing sequence in the butterfly that's about to be started. For example, if `and_bit` starts at 2 (or 010 in binary), and we have 4 threads, then in the different butterflies the threads can form increasing or decreasing sublists as follows

1. 2-element butterfly, `and_bit = 2 = 010`:
   (a) 0 = 000 and `000 & 010 = 000`. So Thread 0 is part of an increasing sublist.
   (b) 1 = 001 and `001 & 010 = 000`. So Thread 1 is part of an increasing sublist.
   (c) 2 = 010 and `010 & 010 = 010 != 0`. So Thread 2 is part of a decreasing sublist.
   (d) 3 = 011 and `011 & 010 = 010 != 0`. So Thread 3 is part of a decreasing sublist.
2. 4-element butterfly, left shift `and_bit` by one bit. So `and_bit = 100`.
   (a) 0 = 000 and `000 & 100 = 000`. So Thread 0 is part of an increasing sublist.
   (b) 1 = 001 and `001 & 100 = 000`. So Thread 1 is part of an increasing sublist.
   (c) 2 = 010 and `010 & 100 = 000`. So Thread 2 is part of an increasing sublist.
   (d) 3 = 011 and `011 & 100 = 000`. So Thread 3 is part of an increasing sublist.

Submission

As usual, your final electronic copy should be committed to your CS SVN repository by 5:30 pm on Wednesday, December 3. You don't need to turn in hardcopy.

Grading

1. Correctness will be 65% of your grade. Does your program use the C library `qsort` function? Does it use a logical block partition of the input list? Does it correctly implement parallel bitonic sort? Are intermediate lists correct? Is the final list correct? Has the timing been taken correctly? Are the details correct: the merge-split, the butterfly structure, etc.

2. Documentation will be 10% of your grade. Does your header documentation include the author's name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments and its return value?

3. Source format will be 10% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?

4. Quality of solution will be 15% of your grade. Are any of your functions more than 40 lines (not including blank lines, curly braces, or comments)? Are there multipurpose functions? Is your solution too clever – i.e., has the solution been condensed to the point where it's incomprehensible?

   Quality of the solution includes allocation of minimal size arrays for the bitonic sort and swapping of arrays in the merge-splits.

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's source code. (This includes source code that you might find on the internet.) You also cannot show your source code to anyone else.