CMSC 216 Project #6 Fall 2010

Multithreaded Mergesort **Due: December 10, 2010, 6:00PM**

Mergesort is an algorithm that can be easily adapted to use multithreading to speed up the sorting. In this project, you will write two mergesort implementations: one without multithreading and one with the ability to operate with up to 4 threads. You will also write a program to aid in testing and timing your mergesort implementations.

1 Procedure

1.1 Obtain the project files

We have supplied two files for your use in this project: (a) `.submit`, needed for submitting your project; and (b) `mergesort.h`, a header file that contains prototypes for the two functions you must write. These files are compressed and contained in `~/216public/project6/project6.tgz`.

Your code for this project should be contained in your `~/216/project6` subdirectory, which is created by extracting files from the tarball you copied.

1.2 Create a `Makefile`

As with some previous projects, you are asked to create a `Makefile` that we will use to build your programs. §2.1 lists the targets you are required to implement, as well as any other requirements for your `Makefile`.

1.3 Write and test a non-threaded mergesort implementation

First, you must implement a mergesort that does not take advantage of multithreading. You must then write a program to store random numbers in an array, and then time how long it takes your mergesort implementation to sort this array.

1.4 Write and test the multithreaded mergesort

You also must write a multithreaded mergesort function that allows a user to sort an array using one, two, or four threads. This multithreaded mergesort should use the non-threaded mergesort to do much of the actual sorting work.

You must then add to your testing program calls to your multithreaded mergesort to time the sort using different numbers of threads.

1.5 Answer questions about your mergesort's operation

In a plain text file named `REPORT`, we also require you to place the answers to several questions about the performance of your mergesort implementations. The questions are specified in §2.6.

2 Specifications

2.1 `Makefile`

Your `Makefile` should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using the `-ansi`, `-pedantic-errors`, `-Wall`, and `-Werror` flags with `gcc`.

You must have the following targets present in your `Makefile`:

1. `all`: make all executables
2. `clean`: delete all object files and executables
3. `mergesort.o` and `mergetest.o`, the object files for your mergesort code and your testing program
4. `mergetest`: the executable testing program created by linking the two object files

We will use the `Makefile` you provide to build and execute your public tests on the submit server; if there is no `Makefile`, your `mergetest` program will not be built, and you will not receive credit for any of the tests.

2.2 The mergesort algorithm

Mergesort, in its simplest form, is an algorithm that divides its input list into two lists, sorts them, and then merges the two sublists into a sorted version of the input list. The algorithm is shown here:

MERGESORT(A)
  if LENGTH(A) ≤ 1
    then return A
  $B \leftarrow$ MERGESORT(first half of A)
  $C \leftarrow$ MERGESORT(second half of A)
  $A \leftarrow$ MERGE(B, C)
  return A

The implementation of the MERGESORT and MERGE functions are left to you as part of the project.

2.3 Comparison functions

Your mergesort implementation should be able to sort any type of data, including `int`s, `double`s, strings, and `struct`s. Because the sorting for different data types requires different comparison code for each type (e.g. strings must use `strcmp()`, while `int`s can be compared using relational operators), you will be using a function pointer to do the work for comparisons. The caller of your sorting functions will supply a pointer to a function that the caller will, in general, also write (except possibly in the case of `strcmp()`, which is already available in the C string library). The prototype for the comparison function is in `mergesort.h`, but it must return a value in the same way that `strcmp()` does. That means the comparison function must return an integer greater than, equal to, or less than 0, if the value pointed to by its first argument is greater than, equal to, or less than the value pointed to by its second argument, respectively.

2.4 Functions

Your `mergesort.c` may contain any helper functions you find useful, but it must contain the following functions, which are declared in `mergesort.h`:

1. `void mergesort(void *arr, size_t num_elem, size_t elem_size, Compare_fn f)`
   This function sorts the first `num_elem` elements of the array `arr` using the comparison function `f` to define the ordering of elements. It should use the mergesort algorithm to perform its sorting. If either `arr` or `f` is `NULL`, the function should just return, doing nothing to the input array.

2. `void mt_mergesort(void *arr, size_t num_elem,`
   `size_t elem_size, Compare_fn f, int thread_ct)`
   This function also performs a mergesort in the same manner as the `mergesort()` function, but splits the work to be done among `thread_ct` new worker threads (i.e., there should be `thread_ct + 1` threads, including the main thread). So the main thread does should not do any sorting work while the worker threads are doing their work. If `thread_ct` is not one of $\{1, 2, 4\}$, this function should just return.

   The division of work to the various threads can be done however you like, although we recommend dividing the array into equal sized subarrays at the beginning, and using the `mergesort()` function for each individual subarray within a thread, and then merging the results as appropriate. You may also find writing a `merge()` helper function useful when implementing these functions.

2.5 Testing program

Your program, `mergetest`, should take three integer arguments on its command line. The first argument, n, is the number of integers that must be sorted. The second argument, s, is the number that is used to seed the

random number generator you will use to generate the array of numbers to sort. The third argument, m, is the upper bound on the value of the numbers you will sort. You may assume that the numeric values of all three arguments can be stored in a signed `int` without any loss of data. The `atoi()` function may be of use here.

Your program will need to dynamically allocate an array to hold the n integers. The array should be filled with n random integers; the integers should be obtained via this algorithm:

```
srand(s);
for (i = 0; i < n; i++)
  A[i] = rand() % m;
```

After setting up the array, your program should proceed to sort it using each of the mergesorts implemented (non-threaded, multithreaded with 1, 2, and 4 threads) – making sure to sort a **copy** of the original array **each** time, as sorting already sorted data will invalidate your results. After each sort, you should report to standard output the time spent sorting, in terms of wall clock time, user time, and system time, in the format shown below.

```
0 threads:  0.751018s wall;  0.735888s user;  0.002999s sys
1 threads:  0.774871s wall;  0.779882s user;  0.000000s sys
2 threads:  0.417598s wall;  0.781881s user;  0.003000s sys
4 threads:  0.251508s wall;  0.779882s user;  0.012998s sys
```

For your own testing purposes, you should also check after each sort to ensure that the array truly is in sorted order, and that it is a permutation of the original array. If you wish, you can print out diagnostic data (such as the results of these two tests) to standard error; we will not be checking any data printed to that output stream.

2.6 Report questions

In your `REPORT` file, you should answer the following questions:

1. For each of the 4 versions of your mergesort (non-threaded, and multithreaded with 1, 2, and 4 threads), how long did that version take to sort the numbers, measured by wall clock time, user time, and system time?
2. Measuring wall clock time, what speedup did each of your three multithreaded runs obtain compared to the non-threaded mergesort? **Speedup** is computed as the time for the non-threaded run divided by the time for a multithreaded run.
3. Measuring user time, what speedup did each of your three multithreaded runs obtain compared to the non-threaded mergesort?
4. Explain the relationship between an increase in the number of threads working on a mergesort and the time (both wall clock and user time) required to perform the sort. In other words, how much speedup is gained by doubling the number of threads? Does this speedup increase at a constant rate as the number of threads become larger? Why or why not?

3 Important Points and Hints

1. For full credit, your mergesorts should work for any number of elements; however, you can still receive credit on some tests if your mergesorts work for specific numbers of elements. All implementations should work for arrays of size 2^n, where $n \in [3, 22]$; better implementations might work for arrays of size $4n$, where n is any positive integer. Other things that might cause your implementations to be less-than-perfect are a failure to handle duplicate items correctly, or an inability to handle items that are not integers.

2. Recall that a `char` is defined as always using one byte of memory. Because a `void *` cannot be dereferenced without casting to another type, the use of `char *` can be helpful in performing pointer arithmetic for this project, as pointers to `char` can be used to access arbitrary byte addresses.

4 Grading Criteria

Your project grade will be determined by the following:

Results of public tests	15%
Results of secret tests	50%
Code style grading	10%
Answers in `REPORT`	25%
Extra credit	5%

Instructions on how to run the public tests will be provided along with the tests, in a file named `README` (included in the tarfile). The public tests will be made available shortly after the project is released. You can find out your results on the tests by checking the submit server a few minutes after submitting your project, or by following the instructions in the `README` file (which we strongly prefer). Secret tests, and their results, will not be released until after the project's late deadline has passed.

The style grading will take place after the late deadline, and will follow the same guidelines as used in Project #3.

4.1 Extra credit

Once again, we offer extra credit to students who make an early submission that passes all public tests. You will receive 5 points of extra credit if you make a submission by 6PM on Monday, December 6, that passes all public tests for the project.

5 Submission

5.1 Deliverables

The only files we will grade are (a) your `Makefile`; (b) `mergesort.c`; (c) `mergetest.c`; and (d) `REPORT`. We will use our versions of all other files to build our tests, so do not make any changes to the other files.

5.2 Procedure

As for previous projects, executing "`submit`" in the project directory will submit your project. Again, we **strongly** encourage you to run public tests on Grace rather than submitting to the submit server and waiting for your submission to be evaluated; it is much faster for you to see your results if you run the tests yourself, and the submit server works much more quickly if the class makes fewer submissions.

6 Other Notes

6.1 Academic Integrity

As mentioned in the syllabus, any evidence of cheating will be referred to the Student Honor Council and may result in a grade of XF in this course. Submissions will be checked with an automated source code comparison tool to look for evidence of cheating.

6.2 Deadlines

Submission deadlines are strictly enforced by the submit server, and we will not extend them for things such as network outages or gremlin attacks. Extensions may be given on a case-by-case basis, but will likely only be granted in emergency cases. Therefore, you should start work on this project soon, as last-minute technical problems are not an excuse for missing either the on-time or late deadline.