Parallel Sort using Multiple Processes and Threads

Goal

In this assignment, you will implement a parallel version of the Bubble Sort algorithm to sort a bunch of numbers using first multiple processes and then multiple threads. You will learn about working with multiple processes, concurrency, race conditions, and simple inter-process and inter-thread synchronization.

Part A: Sequential Bubble Sort

1. Learn the following concepts
    a. The bubble sort algorithm
    b. fork() system call
    c. pipe() system call
    d. pthread_create() and pthread_join()
    e. shmget(), shmat(), shmdt(), and shmctl() system calls.
    f. gettimeofday() to time the sorting performance.
2. Study, compile, and run this sequential bubble sort program. It takes an integer argument N, generates a set of N random integers, sorts them in ascending order using bubble sort, and prints the sorted list to standard output.

```
$ seq_bubble 4
Generating.
20 10 15 2
Sorted sequence is as follows:
2 10 15 20
```

3. Now study, compile, and run this even-odd pass variant of sequential bubble sort. You will find this version more amenable to a parallel implementation.
4. Test the above programs with hundreds, thousands, or millions of numbers. Adjust MAX_NUM and MAX_COUNT in the code as needed.
5. You can redirect very large output to a file using the > operator to examine later.
6. Use `gettimeofday()` to measure and print the sorting time for different values of N.

Requirements for parts B and C below.

- Use only the C language and glibc, so that you can understand low-level behavior or your code. No other languages, libraries or packages are necessary/allowed.
- Do not use any pre-existing bubble-sort implementations/libraries other than the ones provided in this assignment.
- Implement only a parallel version of bubble sort, NOT any other sorting algorithm, even though there may be other (possibly better) parallel sorting algorithms, such as parallel merge sort.

Part B: Parallel Bubble Sort using Processes

Now, implement a parallel version of the even-odd bubble sort using multiple processes.

- Besides N, the parallel bubble sort should take an additional argument P, representing the number of concurrent worker processes. For example, for N=100 and P=5,

```
$ multi_process_bubble 100 5
```

- The initial (parent) process creates a shared memory segment in which it stores an array of N random integers.
- Next, the parent process forks P worker processes. After fork, each worker process automatically inherits the attached shared memory segment that was created by the parent. Hence there is no need for child processes to call the shmat() system call.
- Each worker process should execute the parallel bubble sort operation on (about equal sized) overlapping segments of the array, as explained in the slides.
- Each worker should use a barrier (busy looping for now) to coordinate with "neighboring" workers at the end of each pass.
- You can also store any additional data in the shared memory that's needed for inter-worker synchronization.
- The parent prints out the fully sorted array once all worker processes have finished.
- Is your parallel version faster or slower than the sequential version? Time it using gettimeofday(). Think about how to make a fair comparison.

Part C: Parallel Bubble Sort using Threads

Now implement the same parallel version of even-odd bubble sort using POSIX threads (instead of multiple processes, as in Part B). The logic is the same, except the following

- Name your program `multi_thread_bubble`
- Create multiple threads using `pthread_create()` instead of multiple processes using `fork()`.
- There's no need to create shared memory, since the global memory is shared by default among all threads of a process.

Grading Guideline

Grades are based on both correct implementation and explanation during demo.

- Part A: 20
- Part B: 80
- Part C: 80
- Error handling: 20
- Total = 200

Academic Honesty Reminder

- Review and follow these CS550 academic honesty policies.

- Review and follow these Watson College Academic Honesty policies that spell out the consequences of academic dishonesty.
- Do not ask/give solutions from/to others, including internet forums.
- Don’t post your code publicly on the internet, including github and other repositories.
- You may get zero if you submit a working code that you can’t explain during the demo.