HW #3

CS410/510: Introduction to Performance Measurement, Modeling and Analysis
Due February 27, 2019

Jason Graalum

February 24, 20190

Part A: System Performance

1. Use the record and report commands to generate and display a profile of your code.
2. Use the lock command to see information about lock behavior for your code.
3. Use the stat command to see performance counter values. There are many options for this command and you do not need to use them all. For example, you might choose –a to monitor all CPUs, as opposed to just one. You can use the perf list command to see all of the possible events to choose from, choosing at most 3. Your stat command follows the syntax: perf stat [¡options¿] [¡command¿]

Part B: Tracing Overhead

1. Add tracing instructions to all `MPI` calls EXCEPT `MPI_init` and `MPI_finalize`.

Tracing instructions are timing calls using `MPI_wtime()` before and after a function call Subtract the second call to `MPI_wtime()` from the first to measure elapsed time.

You can either print to stdout, OR collect calls into a local buffer, then at the end send to rank 0 for writing to a file.

THINK: which MPI rank(s) should you measure?
This implementation of merge sort uses the Broadcast, Scatter, Send, Receive and Reduce MPI functions. Since the Broadcast, Scatter and Reduce are global communicator functions, it makes sense to measure the time for all ranks. Because the Send and Receive functions are one-to-one communications, it only makes sense to measure the time of the specific senders and receivers.

2. Determine the Overhead of the tracing calls on the total execution time of the code.
I took data from 5 trials for each condition - with tracing enabled and with tracing disabled. From these runs, I found an average overhead of 9%.
The *.xlsx file with the raw data and calculated overhead is included in the homework tar file - as is the C-code, makefile and raw output.

Part C: System Performance Part 2: Slash Proc

Part C1 - Prep

Working with Multiple Processes

WRITE a short program in a file named showid.c that takes one command-line argument. The showid program outputs to standard error a single line with its command-line argument, its process ID, and parent process ID. Then showid starts an infinite loop that does nothing. [Hint: supply argc

and argv parameters to main(); use getpid() and getppid().]

EXECUTE the following and observe what you see:
./showid 1 — ./showid 2 — ./showid 3

THINK ABOUT the following questions: Which processes in the pipeline are children of the shell?
Are any grandchildren?

How does this change if the pipeline is started in the background (to observe this you will need to execute the above in the background using &)?

SUBMIT showid.c

Part C2

Part C3

Part D. Please read the following paper: https://dl.acm.org/citation.cfm?id=1349636