

Locking using Pthreads, Python, and OpenGL

Nicholas Burlett and Matthew Karr

October 30, 2002

1 Locking and Deadlock Prevention Strategies

A struct *grid_t* holds all the necessary information about the grid of numbers and its associated data. Three sets of locks are allocated inside the *grid_t*. A single mutex for grid locking, an array for row locking, and a two dimensional array for cell locking. All three are included in the *grid_t* to simplify coding at a slight (but relatively insignificant) cost in memory. The threads look at a *sleep_time* variable to determine how long they should sleep for. If the *sleep_time* variable is 0, then they sleep for one second, as in the original version. If sleep time is less than 0, they do not sleep at all. Otherwise they call *usleep()* to sleep for *sleep_time* microseconds. Rather than count only in seconds, the timing algorithm counts milliseconds. Thus, the reported time taken is millisecond accurate.

To avoid deadlock, circular wait is prevented. The method for doing this is simply to require that the element with the lowest index into the mutex array be locked first.

2 Analysis of running time

Rather than graph only two lines per locking method, 3d plots of all 100 points on the 1-10 size and 1-10 thread scales were computed. The method by which they were computed will be discussed in detail in the next section. First, we should look at the general trends of the four types of locking:

- *Cell locking*: It should be apparent that cell locking will provide the best performance for nearly all cases. Figure 1 clearly demonstrates how cell locking remains relatively consistent over most of the graph, only blowing up as the number of threads gets large on small grid sizes. The reasons for this behavior should be obvious: threads rarely need to wait when the number of cells available is similar to the number of threads running, but when the number of cells gets close to the number of threads, blocking occurs much more often.
- *Row locking*: Row locking provides similar performance to cell locking for low thread counts, but grows linearly with increasing threads, as seen in Figure 2. The rate at which it grows

increases with decreasing grid size. Here, how often threads block is not dependent on the square of the size as it is above, but only on the size relative to the number of threads.

- *Grid locking:* Here, the calculation time is dependent only on the number of threads, because only one mutex is being used. How often a thread must wait for the mutex is directly proportional to the number of threads. This is well shown in Figure 3.
- *No locking:* When no locking occurs, each thread runs as quickly as it can (approximately one second per swap for the default timing configuration), and the time taken is dependent only on context switch overhead. In Figure 4, the random variation is quite apparent.

The `sleep(1)` is essentially there to simulate the program actually "doing" something. Without it, the entire program runs nearly instantaneously, minimizing the chances for data corruption, as well as making it impossible to perform useful timing tests. The `sleep(1)` is effectively simulating the program doing computations on the data before writing it back.

3 Programs and Paradigms

For the Windows port of the `gridapp` program, we used the `#define` command liberally with an `#ifdef WIN32`, which can easily be seen near the top of `'gridapp.h'`. We found there is essentially a one-to-one mapping of `pthread` commands to windows threading commands, and we exploited that in horrible ways. C is a beautiful language.

We decided the program would be far more effective and useful if it could actually be observed running. To do so we made a 3-D interface, using OpenGL and windowing using the GLUT library (<http://www.opengl.org/developers/documentation/glut.html>). This version is called `ymgridapp`, and is run with the same command line options. Press 'f' during execution to fullscreen it. The graphics are all symbolic of what the program is doing. The grid is simply the grid, with the green square representing the upper left corner. The Ziggurats are binary representations of the numbers in the grid, with the bottom block being 64, and the higher blocks representing consecutively smaller numbers (32,16,8,4,2,1), with the number in that cell being represented being the sum of the values of each block. The clouds hovering above represent the threads, each thread being a different color. The lightning represents which cells the thread is working on. The lightning surrounding the Ziggurats represents a lock. The glowing ball at the end of the lightning means it is being stopped from getting a lock, and is waiting for a different thread to release. The floating ghost Ziggurats represent the temp value currently in the thread, if there is one.

The swapping can be run concurrently with itself within a single program because nearly all global variables were removed and brought into a single struct. None of these structs are created on the stack, so the function that creates them can exit without corruption if the thread is still running. Two functions were created to aid in the timing of the swapping algorithm. The first `do_timing_test` creates the necessary data on the heap and starts the swapping threads. It returns a pointer to the grid structure that also contains information about the threads. The `timing_done` function can then be called with that pointer to check if all of the threads have completed. This function returns -1 if the threads are still running, and the time it took to complete in milliseconds otherwise. This setup allows another language, in this case Python, to simplify the statistics gathering.

Two Python classes provide access to the grid timing information. The first encapsulates a single instance of the timing run, and extends the Python thread class such that it itself can be run concurrently with other timing runs. The second extends the first, and encapsulates multiple instances of the first, and then averages the results together. Both can generate graphs of the data by calling gnuplot through the gnuplot.py interface.

4 Building and Running

- *Linux*: `untar`, enter directory, type `'make'`, to build the OpenGL version you need to have GLUT installed. To run, type `'gridapp [args]'` for the command line application, or type `'ymgridapp [args]'` to get the OpenGL version.
- *Windows*: `untar`, enter directory, open `'gridapp.dsw'`, build `gridapp` or `ymgridapp` (You need to have GLUT installed and set up properly to build `ymgridapp`), `'gridapp.exe'` and `'ymgridapp.exe'` should be built in the `'Win32_Build'` directory. Alternatively, they should already be there anyway, along with `'glut32.dll'`. Just open up a command line, `cd` to the directory, and run either `'gridapp'` or `'ymgridapp'` with appropriate options.
- *Scripting*: requires Python 2.2, Numeric Python, Gnuplot.py, and `LATEX`Running `gen_report.sh` on a Un*x system will generate this document. We don't recommend you try to run this without a deep understanding of Python, Gnuplot, and `LATEX` as the shell script is not designed to be particularly robust, and because the script takes upwards of four hours to complete.

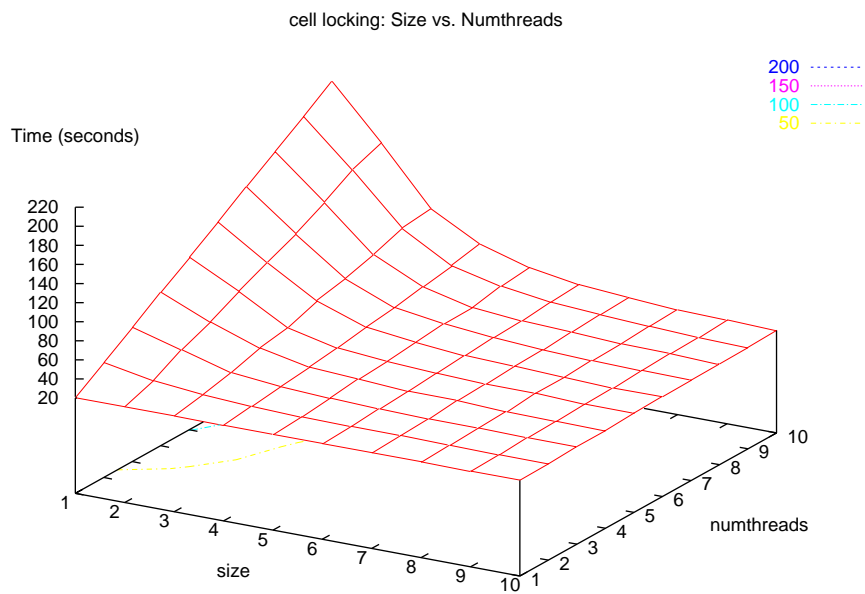


Figure 1: Cell locking: Note the increasingly poor performance as the number of threads increases and the size decreases.

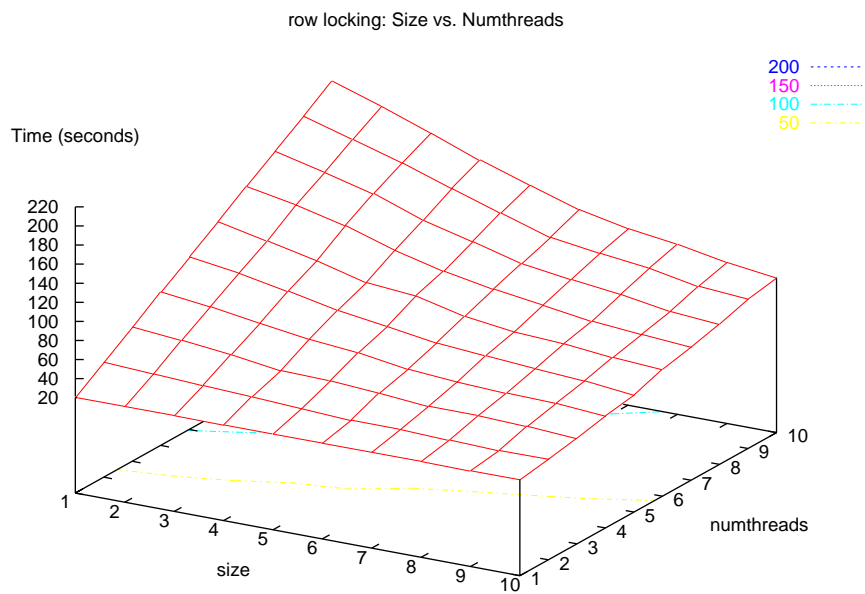


Figure 2: Row locking: Note that performance changes more linearly than cell locking, but increases more quickly where cell locking continues to be fast.

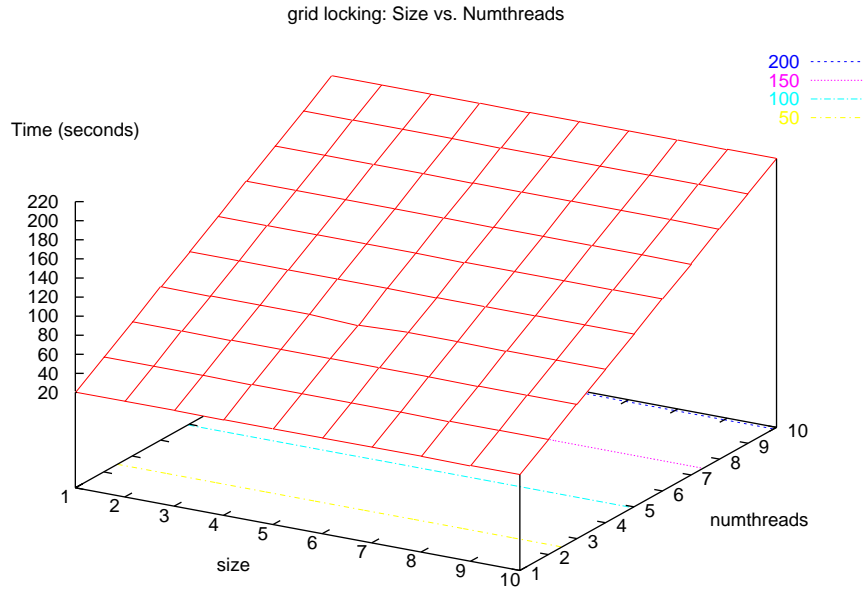


Figure 3: Grid locking: Note that the amount of time depends only on the number of threads.

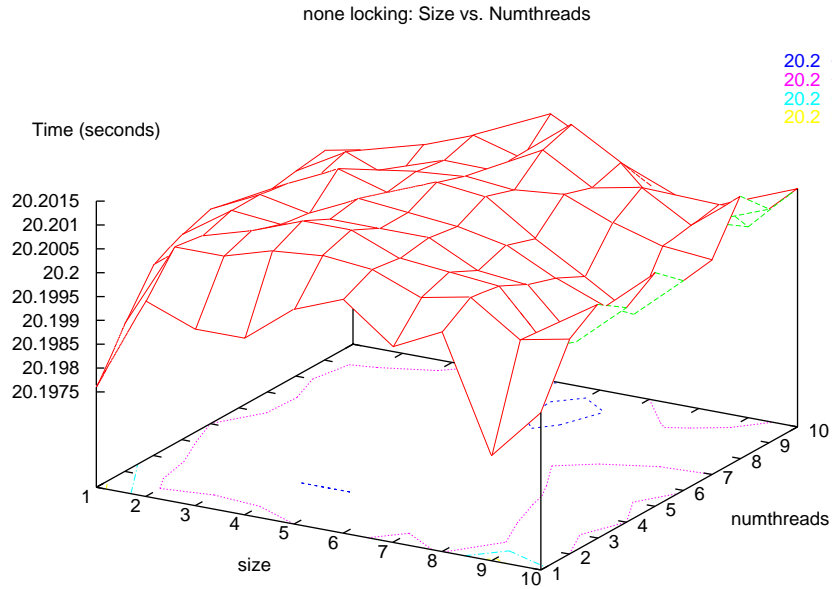


Figure 4: No ("none") locking: Note that the amount of time is very randomly distributed, but centers very closely around 20.19 seconds. This is due to each thread blocking for twenty total seconds, plus context switching overhead. Be aware that the scaling on the Time dimension is considerably different than on the other three graphs.