Parallel Processing Lab 1: Pthreads

Gaussian Elimination

Sarah Peachey & Nathan Schomer

February 21, 2018

Abstract: pthreads is used to parallelize an executable function. When the pthread is created it is given a pointer to a function and a structure containing all the information and needed to run that function. The data structure than needs to be dereferenced, and the code can then run over a section of the data.

1 Design

The design, as seen in the pseudo-code on the following page, is simply to spawn N threads with the use of a for loop. The loop points the threads to a executable/function in which the gaussian reduction is performed. To the perform the reduction data and information need to passed in as well, but the built-in function that creates pthreads only excepts one input. So if you need to pass in multiple pieces of data, a data structure must declared and defined with all the input information. Then in the function that the thread has to dereference the pointer to the input structure and all attributes. So the function I made was called *parallel_gold* and it accepted a TwoMat data structure called $Matrices\_ptr$. Originally the plan was to used a flip flop architecture so two matrices were going to be passed it, so that's what I named it. As the code developed a different design was used so the name seems inaccurate. But the TwoMat data structure has a U, a, b, *num_threads*, and tid properties. U is a pointer to the elements in the input matrix U, a is the start of the section that the thread will operate on, b is the end of the section that the thread will operate on, *num_threads* is the number of threads (N), and tid is the thread ID $[0, 1, ...N - 1]$. The pthreads version is then designed very similar to the serial design, except that the division is parallelized by each thread getting a chuck on the row $[a, b]$ to operate on. Then a barrier is implemented with a mutex lock. The elimination is parallelized by each thread getting a chunk of rows $[a, b]$ to operate on. Then another barrier using mutex lock, to make sure everything is synchronized before going to the next k. One aspect of the parallel design that need to be cared for was when k incremented, for some threads k would be greater then

the lower bound (a) of it's respective chunk, so if $(k+1) > a$ then $a++$.

2 Results

As seen in the table on the next page, the best performance was seen with 8 threads. At 16 threads the overhead for thread creation and synchronization slowed down the program. The speed up time was calculated by (1). It is also interesting to note that for the same thread count the speedup increased as the matrix got larger. This is most likely do to the chunk size utilization. Since a updates for some threads as k increments, that implies that the first thread is going to have a smaller chunk to execute over time, and once k is larger then the upper bound (b) for that thread, it will no longer have anything to execute. So the larger the chunks are, the more data, the longer each thread will continue to evaluate it's respective chunk. A better design could have been to recalculate a and b as k gets larger, so that each thread always has a chunk of data to evaluate.

$$s = \frac{t_{serial}}{t_{parallel}} \tag{1}$$

Thread Count	1024x1024	2048x2048	4096x4096
4	2.08	2.39	2.47
8	2.70	4.03	4.76
16	2.16	3.53	4.62

Result: Function that Performs Gaussian Reduction

initialization;

Dereferance the struct; **for** *each row in temp_array* **do**

    for *each element in section of row* **do**
        Perform Division Step
    end
    Barrier using a mutex Set principle diagonal to 1 **for** *each row in section* **do**
        for *each item in row* **do**
            Perform Elimination Step
        end
    end
    Barrier using a mutex Make lower triangle 0 Barrier using mutex

end