Gaussian Elimination Assignment

by Cy Baca

1 How your program stores the data

All of the matrix data used by the program is stored in a global struct variable called `_vectors`, which holds pointers to doubles to:

- `*echelon`, the matrix to perform reduction on
- `*variables`, an array to hold the x values during back substitution
- `*original`, a copy of the values that were generated at the beginning of the program
- `**rows`, an array of pointers to doubles which reprsent the location of the beginning of each row in the echelon matrix.

All of the arrays in `_vectors` are allocated contiguously and aligned to the *cache line size* of the given architecture by making calls to `posix_memalign()` and `sysconf()`, in order to reduce the chance of threads pulling overlapping cache lines.

There is one more global struct variable called `_stats`, but it is only used at the beginning and end of the program to store and print benchmark data. All other variables during runtime are created and used within, or passed to and from, functions.

2 How your program partitions the data and work and exploits parallelism

The program can be broken down into a series of stages.

- *Initialization*
- *Gaussian Elimination*
  - *Matrix Reduction(parallel)*
  - *Back substitution(parallel)*
  - *L2 Norm*
- *Finalization*

Each stage is define by a function, or group of functions, with names that represent its purpose. Functions that are part of a certain stage have the first part of their master function's name with an underscore appended to the front of them so that their purpose is easy to determine just from reading the name. For example, the *Matrix Reduction* stage starts with `echelon_reduce()` and every function that is directly or indirectly used by it are prefixed with "`_echelon_<etc>`".

Initialization

The first stage, *initialization*, begins in `main()` by checking args for **"N"**, and passing the **N** value to `program_init()`. From there, `_vectors` is passed by reference to `vectors_init()`, where a series of functions initializes the pointers with heap memory, and initializes the memory pointed to by `original` with random doubles from -1.0e6 to 1.0e6, and copies that memory over to `echelon`.

Gaussian Elimination

The data is passed to `gaussian_elimination()`, where all of the major matrix math functions are called in a specific order, starting with `echelon_reduce()`

`echelon_reduce()` and each of its helper functions act as a series of nested *for* loops and loop nests, but are broken down into functions. I chose this strategy for two reason:

- To keep the pattern of data mutation concise and manageable.
- To make clear and seperate sections of parallelized code.

Back substitution

When `echelon_reduce()` is finished, it returns to `gaussian_elimination()`, after which `back_substitution()` is called. This section uses a two level nested for loop and employs a slighlty different strategy than the last. This time, the threads all share a given row, and work on it at the same time. When the row is complete, the threads enter a critical section and write to the `variable` array. This way, the chances of reading from a dirty piece of memory while retrieving the next x value is mitigated.

L2 Norm

This section is not parallelized, and it used to validate the approximated x values. If the L2 Norm is not a very low number, there is likey an error in the previous calculations.

Finalization

Finally, the benchmark values are printed to the console, and all heap memory used at runtime is deleted.

Parallel sections:

- in `_echelon_pivot()` for finding the next pivot. Each row gets a chunk of the current diagonal column and searches for the largest value at each index. Afterward, the threads meet at a critical section where they largest index is stored in a shared variable.
- in `_echelon_annihilation()`. Each row gets a private pointer to the beginning of its own row, finds its own multiplier, and reduces that row. This allows threads to simulatneously read data from the pivot row needed by each of them. Meanwhile, each thread is guaranteed to be the only thread with a copy of its write data for all rows with a length greater than the cache line size.
- in `back_substitution()`, where the parallel section starts on the inner loop of a double nested *for* loop. This means, that multiple threads share a row. In order to compensate for possible cache line discrepencies, I used a dynamic schedule with the parameter of the `l1d cache_line_size / sizeof(double)`. This way, threads should have chunk sizes with data that no other thread contains, avoiding false sharing.

3 How the parallel work is synchronized

In `echelon_eliminate()`, in the annihilation loop, the default schedule is used, (meaning it should be static, with chunks the size of the ratio loop length to threads). The only private variable is the pointer pointer used to iterate through the row indexes, and there are seven shared variables.

In the *Partial Pivoting* parallel section, the default schedule is used again for the *for* loop, which divides a column up by chunks of indices. This time, the threads share two variables: one for reducing the largest value in

the column, and one for storing its index. Within the section, each thread finds the highest value within its chunk, and then enters a critical section. Since I couldn't figure out how to make a directive that would synchronize a reduction for a max value **and** store the index, I had to explicitly create a *critical* section where the threads could compare max values, and write if needed.

In the *Back Substitution* section, I tried opened a parallel section using the reduction(-) directive for each X value. I first used the *guided* schedule to try to accomodate for the growing size of the rows, but didn't see improvement. I ended up using *dynamic* with size 8 (cache line size / sizeof(double) which shaved off a few seconds, although the default *static* schedule seemed to work equally as well.

4 Pseudocode that sketches the key elements of your parallelization strategy

If all of the main matrix maths functions where concatenated into one function, it would likely look something like the following:

```
/** start **/
number_of_pivots = N - 1

double **rows =  { pointer to beginning of each row }

foreach cur_pivot_index in rows, cur_row < number_of_pivots

    next_pivot_index = 0;

    /* partial pivot *********************************/

    #parallel section

        start = cur_row[0]

        for index in column_under_start

            reduce(max(column_under_start[index]))

            #critical
```

```
                next_pivot_index = index

            swap(rows[cur_pivot_index], rows[next_pivot_index])

        /* annihilation ********************************/
        pivot = &rows[cur_row][next_pivot_index]

        #parallel for
        for cur_row in rows + 1

            cur_multiplier = cur_row / pivot[0]

            for i in row

                cur_row[i] -= multiplier * pivot[i]

    /* back substitution ********************************/
    foreach last_index in rows, --last_index

        next_variable

        cur_row = &(*rows)[last_index]

        #parallel for private(next_variable) reduction(-)
        foreach upper_triangle_index in cur_row

            next_variable -= cur_row[upper_triangle_index] *
            variables[upper_trianlge_index]

        variables[last_index] = next_variable /
        cur_row[upper_triangle_index];
```

5 Your justification for your implementation choices

I originally wrote the entire program without the partial pivoting section, because I had overlooked that part of the assignment(oops). This influenced the outcome of my program design decisions, for better or for worse, in how I ended up allocating memory, loop structure, and mechanisms for iterating over the data.

Before implementing partial pivoting:

I wanted to use contiguous memory so that I could

- keep it aligned to a cache block
- maintain tight spacial locality

After realizing I needed to implement partial pivoting I needed to figure out a way to iterate over the contiguous memory in a specific order without incurring too much overhead.

So, I ended using a pointer to an array of pointers.

I malloc'd a pointer to an array of double pointers, and pointer each one at indexes corresponding to the beginning of the rows in my original contiguous `echelon` array. Even though I eneded up with a 2D array, I used it as a reference point for other pointers to iterate over rows in *for* loops; then, when partial pivoting was around the corner(ha ha, no pun intended), I could just swap the pointers in the pointer array, and by annihilation time I had a way to iterate in the correct order without any extra fuss.

I don't know how much, if any, extra overhead was caused by the extra pointer variables I needed to add for loop iterations, in contrast to just instatiating my data in 2D array from the beginning, and I didn't have a chance to try this implementation, so it remains a mystery.

I do know that I drastically overestimated the overhead cost of implementing the partial pivoting section of the code, because I didn't notice much(a second or two if anything) change in timing from before I implented it, so that part was a success.

6 How successful your approach was

My setup was very marginally successful. For two two threads, I got a fat speedup increase from 0.0 to 1.8, with an efficiecy of 9.1. After that, the efficiency was neraly halved to 0.56, and this pattern continued all the way up to 40 cores, where I ended up with less tham 0.08 efficiency.

I imagine the lack of efficiency must be at least partially due to the scalability of the problem itself. Many of the operations that need to take place for Gaussian Elimination rely on the completion of previous operations on the data before they can carry on. This means that there is limited space for threads to work concurrently.

Considering that the parallelization of the code required only *six* lines of code, I really impressed with the amount of speedup I was able to squeeze out of it. I don't want to imagine the nightmare I'd have to endure trying to complete the same work using posix threads or the cold hard futex library.

All in all, I guess my approach was at least marginally successful.

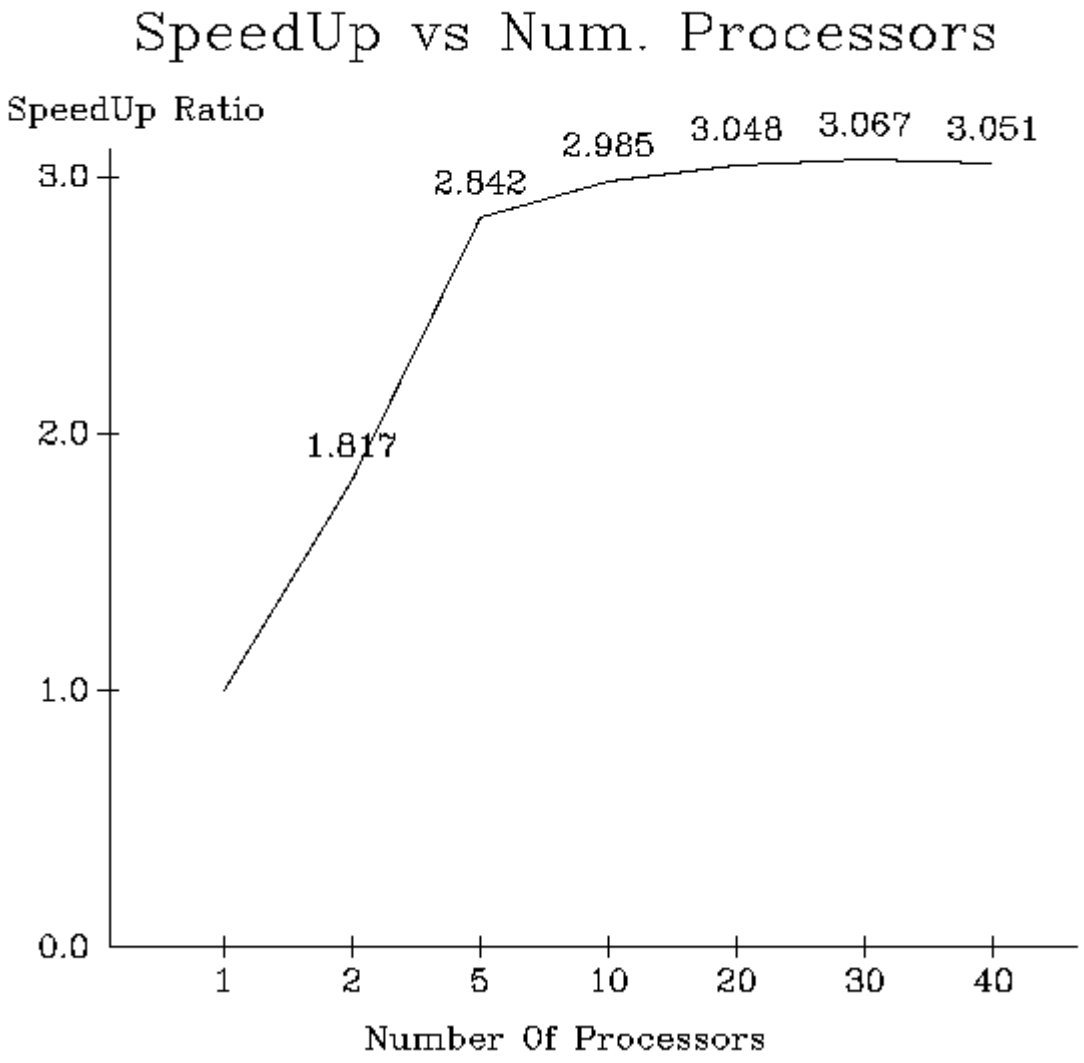
7 Table of the time to execute and the l2-norm, for every run, with the minimum time on p cores highlighted

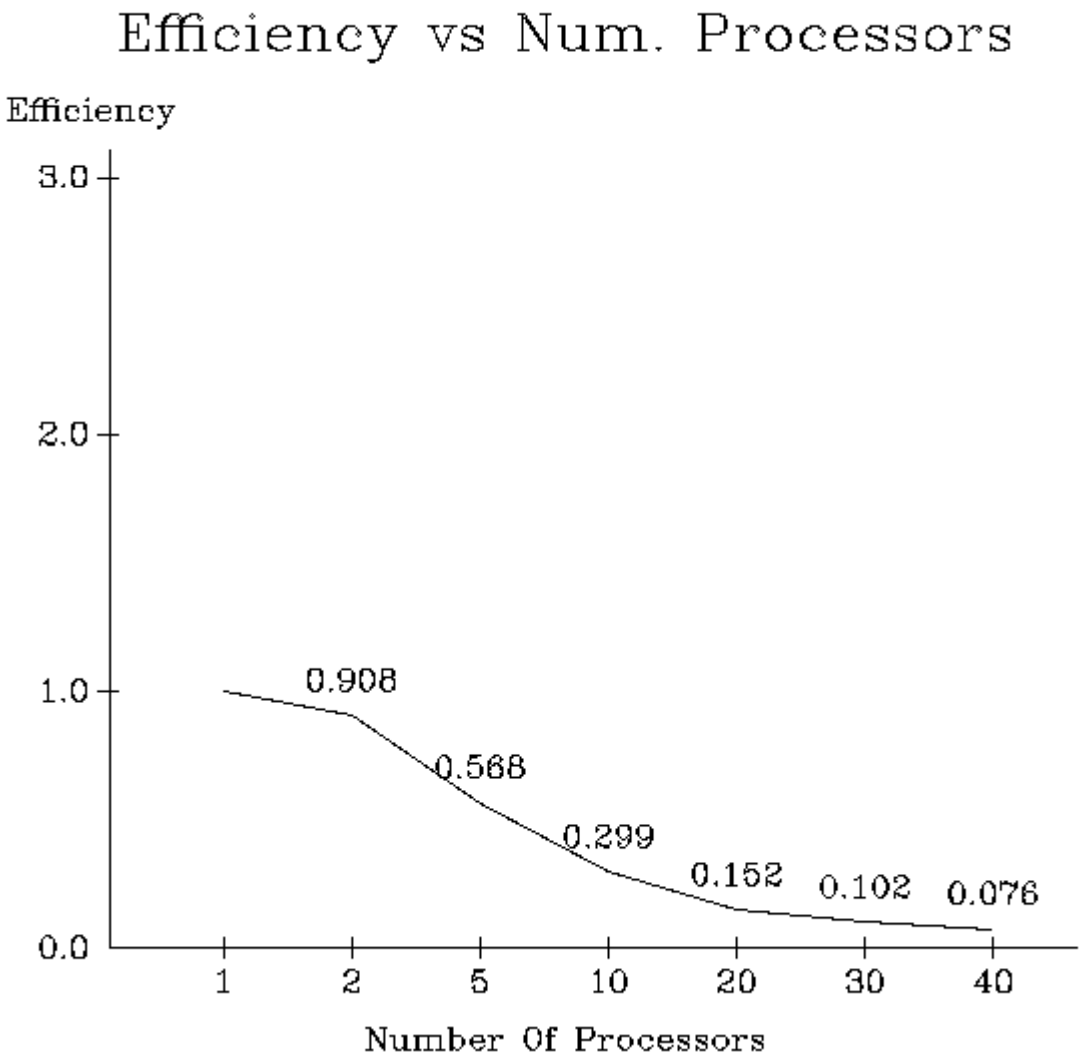
threads:	times:	l2norms:
1	664.261893	2.5800157282e-03
	724.946826	9.4443812796e-04
	654.247888	9.0724052356e-03
2	365.474405	4.5947765776e-04
	360.074551	3.0701745230e-04
	501.253396	6.6478362796e-04
5	230.183935	2.1035538762e-04
	232.148950	1.1235548047e-03
	411.946127	1.5833014563e-03
10	219.263489	2.1327824368e-04
	219.164048	1.1445531847e-03
	219.367498	1.2088836573e-03
20	214.652390	1.3031318008e-03
	404.893739	3.0483806989e-04
	214.777489	9.3423978678e-04
30	213.484887	4.7350450144e-04
	213.462830	3.1031698941e-03
	213.336266	4.1833982675e-04
40	413.856779	3.1035303350e-04
	215.757580	5.5170123099e-04
	214.470610	1.8542336495e-03

8 Table of the minimum times, calculated speedup, efficiency, l2-norm

threads:	minimum time:	speedup:	efficiency
1	654.247888	0.0	1.0
2	360.074551	1.816979	0.908489
5	230.183935	2.842283	0.568457
10	219.164048	2.985197	0.298520
20	214.652390	3.047941	0.152397
30	213.336266	3.066745	0.102225
40	214.470610	3.050525	0.076263

9 Separate graphs of speedup and efficiency




Efficiency vs Num. Processors
Efficiency
3.0
2.0
1.0
0.0
0.908
0.568
0.299
0.152
0.102
0.076
1
2
5
10
20
30
40
Number Of Processors