CS 170 - Lab 1

Laolu Osuntokun & Lin Chai

Coding Tips

- Learn to use git
  - Distributed Version Control System (VCS)
  - Preserves history of all changes
  - You **will** use this later in your career
- Learn to use gdb
  - Print statement debugging won’t cut it
- **Always** zero out all values:
  - `void *memset(void *s, int c, size_t n);`
- **Always** check pointers for NULL before use
- Piecewise decomposition:
  - Fully understand problem domain
  - Decompose into individual sub-steps
- Testing
  - Unit - each sub-step in isolation
  - Integration - all sub-steps composed for final solution

Suggested Plan of Attack

1. Command line argument parsing
   a. Handle all errors, check sanity
2. Matrix data structure + string parsing
   a. Handle all edge cases: comments, invalid dimensions etc
3. Matrix multiplication
   a. Write tests to ensure correctness
4. Single-threaded matrix multiplication
   a. Learn pthread API, passing arguments to threads (work-unit)
5. Multi-threaded matrix multiplication
   a. Properly divide up rows, handling uneven splits

Mastering C

- How to C in 2016
- The Clockwise/Spiral Rule
  - Type declarations can be a bit ambiguous, use this rule for clarity
- String Parsing
  - What if I told you…there are no strings in C
  - `char yo[6] = {'H', 'e', 'l', 'l', 'o', '\0'};`
  - `char yo[]  = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’};`
  - `char *yo = (char *)malloc(6*sizeof(char))`
    - `y[5] = ‘\0’`
- Pointers
  - Dereferencing, arithmetic, malloc, free
- Carefully read appropriate man pages

C String Parsing

- `#include <stdlib.h>`
  - Conversion routines from string -> {int, double, float, unit}
  - `atoi, strtoll, etc...`
- `#include <string.h>`
  - Routines for:
    - Copying
    - Comparison
    - Searching
- Reading a line from a file:
  - `char *fgets(char *s, int size, FILE *stream);`
- Command-line parsing:
  - `#include <unistd.h>`
  - `int getopt(int argc, char * const argv[], const char *optstring);`
- Think “state machine” when parsing

Matrix Multiplication

- Computes: A x B = C
- Matrix A **must** have the same number of <u>columns</u> that B has <u>rows</u>
  - (<u>M</u> x **N**) * (**N** x <u>P</u>) = OK
    - Results in M x P matrix
  - (M x M) * (P x N) = NOPE
  - Solution should handle cases where matrix multiplication is undefined
- Naive algorithm is O(N^3) or O(mpn)
  - ```
    for i in range(num_c_rows):
      for j in range(num_c_cols):
        for k in range(num_cols_a):
          c[i][j] += (a[i][k] * b[k][j])
    ```
- Can be easily parallelized by partitioning range of two outer loops
    ```

Threading

- Thread vs Process
  - In the beginning…there were only processes
  - Processes are "containers", abstraction around a unit of execution + its resourc
    - Virtual address space, code, file descriptors, PID, etc.
  - Threads are "lighter weight" units of execution within a process
    - Threads across a process share resources, added special thread context
    - code, locals, globals
- pthreads
  - ```
    int pthread_create(pthread_t *thread,
                       const pthread_attr_t *attr,
                       void *(*start_routine) (void *),
                       void *arg);
    ```
  - ```
    int pthread_join(pthread_t thread, void **retval);
    ```
- Throw away all notions of implicit "order", need synchronization for explicit "order"

Threaded Multiplication

- Need to define a “unit of work” to be passed as the last argument to `pthread_create`
    - ```
      struct work {
          ....
      }
      ```
- Properly partition work:
    - `rows_per_thread = num_rows_a / num_threads`
    - `left_over = num_rows_a % num_threads`
- Watch out for race conditions due to improper work allocation
    - Threads executing overlapping work-units
- `pthread_join` to wait/collect
      ```

Error Handling

- Matrix parsing
- Command line arguments
- Matrix multiplication correctness

Extra Credit - CPU Cache Optimization

- Memory hierarchy: L1, L2, L3, main memory
  - Main memory is **slow** compared to the CPU caches
- Locality is King
  - Temporal Locality
    - If a piece of memory was just accessed, it’ll likely be accessed in the future
  - Spatial Locality
    - If a piece of memory was just accessed, it’s neighbors will likely also be accessed
  - Both forms of “pre-fetching” essentially
- Loading may cause evictions, which are slow. Goal is to avoid “**cache thrashing**”.

Extra Credit - CPU Cache Optimization

- Caches are divided into “lines” or “blocks”
- Granularity of loads/stores is at the cache line level
- Optimize spatial+temporal locality:
  - ```
    for i in range(num_c_rows):
      for j in range(num_c_cols):
        for k in range(num_cols_a):
          c[i][j] += (a[i][k] * b[k][j])
    ```
- Problems:
  - Each new inner loop iteration may cause **cache misses** on array **b** due to column-wise traversal
  - Dividing C by strips (strips of each thread would be much longer than one cache line) may cause cache misses when storing values into C
    ```

Extra Credit - CPU Cache Optimization

- Solution 1
  - switch to row-wise traversal of **b**
- Solution 2
  - divide C into several tessellating rectangles. No overlap between rectangles.
  - Example:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

so then we rewrite C = A*B as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$