CS 61C Spring 2024 — Semester Review

Discussion 13

1 Coding in C

Suppose we've defined a linked list **struct** as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

1.1 Implement `prepend`, which adds one new `value` to the front of the linked list. Hint: why use `ll_node** lst` instead of `ll_node* lst`?

```
1  void prepend(struct ll_node** lst, int value) {
2      struct ll_node* item = (struct ll_node*) malloc(sizeof(struct ll_node));
3      item->first = value;
4      item->rest = *lst;
5      *lst = item;
6  }
```

1.2 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
1  void free_ll(struct ll_node** lst) {
2      if (*lst) {
3          free_ll(&((*lst)->rest));
4          free(*lst);
5      }
6      *lst = NULL; // Make writes to **lst fail instead of writing to unusable memory.
7  }
```

2 C Generics

2.1 **True or False:** In C, if the variable `ptr` is a generic pointer, then it is still possible to dereference `ptr` when used on the right-hand side of an assignment operator, e.g., `... = *ptr`

False. To dereference a pointer, we must know the number of bytes to access from memory at compile time. Generics employ generic pointers and therefore cannot use the dereference operator!

2.2 Generic functions (i.e., generics) in C use void * pointers to operate on memory without the restriction of types. Such generics pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time. They instead use byte handling functions such as `memcpy` and `memmove`.

Implement `rotate`, which will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_int_array(array, 10);
  rotate(array, array + 5, array + 10);
  print_int_array(array, 10);
  rotate(array, array + 1, array + 10);
  print_int_array(array, 10);
  rotate(array + 4, array + 5, array + 6);
  print_int_array(array, 10);
  return 0;
}
```

Output:

```
$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
```

Your Solution:

```
void rotate(void *front, void *separator, void *end) {




}
```

```
void rotate(        ,           ,          ) {
  size_t width = (char *) end - (char *) front;
  size_t prefix_width = (char *) separator - (char *) front;
  size_t suffix_width = width - prefix_width;
  char temp[prefix_width];
  memcpy(temp, front, prefix_width);
  memmove(front, separator, suffix_width);
  memcpy((char *) end - prefix_width, temp, prefix_width);
}
```

3 Data-Level Parallelism

3.1 SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = __mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
    }
    return result[0] * result[1] * result[2] * result[3];
}
```

4 Amdahl's Law

4.1 Derive Amdahl's Law using the ratio: $\textbf{Speedup} = t_{\text{naive}}/t_{\text{optimized}}$

First, we can split the overall time a program takes into the time it takes for the part of the program that can be optimized and the rest of it. Letting F represent the fraction that can be sped up, we have:

$t_{\text{naive}} = F(t_{\text{naive}}) + (1 - F)t_{\text{naive}}$

Then, we can implement the optimization, known as the speedup factor S into our equation by dividing the optimizable portion to get:

$t_{\text{optimized}} = \frac{F(t_{\text{naive}})}{S} + (1 - F)t_{\text{naive}}$

Solving for the ratio $\textbf{Speedup} = t_{\text{naive}}/t_{\text{optimized}}$ leads to Amdahl's Law.

4.2 Assuming we have infinite threads and resources, what would our overall speedup be for a program with some fraction of our code that can be parallelized F?

With infinite scaling factor S, our total speedup will approach $\frac{1}{1-F}$. However, in reality there would be some non-zero overhead that is required to properly split up work.

4.3 You write code that will search for the phrases "Hello Sean", "Hello Jon", "Hello Dan", "Hello Man", "Bora is the Best!" in text files. With some analysis, you

determine you can speed up 40% of the execution by a factor of 2 when parallelizing your code. What is the true speedup?

Using Amdahl's Law with F=0.4, S=2:

$$\frac{1}{0.6 + \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

4.4 You run a profiling program on a different program to find out what percent of time within the program each function takes. You get the following results:

Function	% Time
`f`	30%
`g`	10%
`h`	60%

(a) Assuming that each of these functions can be parallelized by the same speedup factor, which one, if parallelized, would cause the most speedup for the entire program?

`h`

(b) What speedup would you get if you parallelized just this function with 8 threads? Assume that work is distributed evenly across threads and there is no overhead for parallelization.

$1/(0.4 + 0.6/8) \approx 2.1$

5 Thread-Level Parallelism

5.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the number of threads can be any integer greater than 1. Assume no thread will complete in its entirety before another thread starts executing. Assume `arr` is an `int[]` of length `n`.

(a)
```
// Set element i of arr to i
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

Slower than serial: There is no **`for`** directive, so every thread executes this loop in its entirety. `n` threads running `n` loops at the same time will actually execute in the same time as 1 thread running 1 loop. The values should all be correct at the end of the loop since each thread is writing the same values. Furthermore, the existence of parallel overhead due to the extra number of threads will slow down the execution time.

(b)
```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
```

```
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

Sometimes incorrect: While the loop has dependencies from previous data, in a interweaved scheme where the threads take turns completing each iteration in sequential order (e.g.

```
for (int i = omp_get_thread_num(); i < n; i += omp_get_num_threads())
```

is the work allocation per thread and the order of execution is based on the shared variable `i` from 2 to `n`), each thread will have the correctly updated shared `arr` to compute the next Fibonacci number. Note that this scheme would still be slower than serial due to the amount of overhead required as the threads need to wait for each other's execution to finish as well as deal with coherency issues regarding the shared data.

(c)
```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = 0;
```

Faster than serial: The **for** directive automatically makes loop variables (such as the index) private, so this will work properly. The **for** directive splits up the iterations of the loop to optimize for efficiency, and there will be no data races.

(d)
```
// Set element i of arr to i;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    *arr = i;
    arr++;
```

Sometimes incorrect: Because we are not indexing into the array, there is a data race to increment the array pointer. If multiple threads are executed such that they all execute the first line, *arr = i; before the second line, arr++;, they will clobber each other's outputs by overwriting what the other threads wrote in the same position. However, taking a similar interweaved scheme as in 4.1b, there is an order that will not encounter data races, though it will be slower than serial.

5.2 What potential issue can arise from this code?

```
1 // Decrements element i of arr. n is a multiple of omp_get_num_threads()
2 #pragma omp parallel
3 {
4     int threadCount = omp_get_num_threads();
5     int myThread = omp_get_thread_num();
6     for (int i = 0; i < n; i++) {
7         if (i % threadCount == myThread) arr[i] -= 1;
8     }
9 }
```

False sharing arises because different threads can modify elements located in the same memory block simultaneously. This is a problem because some threads may have incorrect values in their cache block when they modify the value `arr[i]`, invalidating the cache block. This will slow down the execution of this program due to frequent coherency misses.

6 Fall 2021 Final Q9 - Parallelism

6.1 Fred's Factorization Factory has unveiled their latest product: an algorithm that factorizes an array of numbers provided. You want to test their factoring algorithm, so you decide to write the following function:

`int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64_t *c);`

- n: The length of each list of integers. For simplicity, you may assume that n is a multiple of 4.
- a, b, c: Pointers to arrays of 64-bit integers.

`testFactor` returns 1 if, for all `i` from 0 to n-1, `a[i]*b[i] == c[i]`. Otherwise, it returns 0.

You have access to the following SIMD instructions:

- `_mm256 vectorLoad(void* ptr)`: Loads four `uint64_t` from `ptr` into a SIMD vector
- `void vectorStore(void* ptr, _mm256 mm)`: Stores the four `uint64_t` in `mm` at `ptr`
- `_mm256 vectorMul(_mm256 a, _mm256 b)`: Multiplies the values in `a` and `b`, and returns the result
- `_mm256 vectorSet0()`: Returns a vector containing only 0s.
- `_mm256 vectorOr(_mm256 a, _mm256 b)`: Computes the bitwise OR of the two vectors, and returns the result.
- `_mm256 vectorXor(_mm256 a, _mm256 b)`: Computes the bitwise XOR of the two vectors, and returns the result.

```
int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64 *c)
{
```

```
    uint64_t output[4];

    _mm256 total = ____________________________________________;

    for(int i = 0; i < ___________________; i+= ___________________)
    {
        _mm256 adata = vectorLoad(a+i);
        _mm256 bdata = vectorLoad(b+i);
        _mm256 cdata = vectorLoad(c+i);

        _mm256 prod = ____________________________________________;

        _mm256 isequal = ____________________________________________;

        ____________________________________________________________;
    }
    vectorStore(output, total);

    return ____________________________________________ ? 1 : 0;
}
```

Blank 1: `vectorSet0()`

Blank 2: `n`

Blank 3: `4`

Blank 4: `vectorMul(adata, bdata)`

Blank 5: `vectorXor(prod, cdata)`

Blank 6: `total = vectorOr(total, isequal)`

Blank 7:

`(output[0] == 0) && (output[1] == 0) && (output[2] == 0) && (output[3] == 0)`

Other solutions may exist. Note that the solution:

`output[0]+output[1]+output[2]+output[3] == 0` is incorrect, since we could have received outputs that happened to add to 0, even if they aren't all 0. As an example consider the inputs $A = [0, 0, 0, 0]$, $B = [0, 0, 0, 0]$, $C = [1 << 30, 1 << 30, 1 << 30, 1 << 30]$.

The main idea of this accumulator is noting that two numbers are equal if and only if their XOR is exactly 0. By ORing next, we ensure that the total values are nonzero if any instance of isequal ended up returning a nonzero value. Thus, we can just check if the output vector is all zero at the end.

In general, bitwise operations tend to be much faster than branch comparators, so it is generally preferable to use bitwise and simple arithmetic operations when possible.