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?

```
void prepend(struct ll_node** lst, int value)
```

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

```
void free_ll(struct ll_node** lst)
```

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

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.

```c
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:

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








}
```

3 Data-Level Parallelism

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

```c
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 = ________________________________________;
    for (int i = 0; i < _____; i += ___) { // Vectorized loop
        prod_v = _________________________________________________________________________;
    }
    __mm_storeu_si128(__________________________, __________________________);
    for (int i = ______________; i < ____________; i++) { // Handle tail case
        result[0] *= ________________________;
    }
    return _____________________________________________________;
}
```

4 Amdahl's Law

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

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?
rin

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?

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?

(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.

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;
}
```

(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];
```

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

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

5.2 What potential issue can arise from this code?

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

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;
}
```