

# Homework 1

MacMillan, Kyle

September 28, 2018

# Contents

Title	
Table of Contents	i
List of Figures	ii
List of Algorithms	ii
1 Problem 1	1
2 Problem 2	1
3 Problem 3	4
3.1 Problem 3a . . . . .	4
3.2 Problem 3b . . . . .	4
3.3 Problem 3c . . . . .	4
3.4 Problem 3d . . . . .	4
3.5 Problem 3e . . . . .	4
3.6 Problem 3f . . . . .	5
3.7 Problem 3g . . . . .	5
3.8 Problem 3h . . . . .	5
4 Problem 4	6
5 Problem 5	7
6 Graduate Assignment	8

## List of Figures

1	Example Debug Output . . . . .	3
2	Timed Reduction Methods . . . . .	3

## List of Algorithms

1	Hypercube Traversal . . . . .	7
---	-------------------------------	---

# 1 Problem 1

What are the identity values for the operators: &&, ||, |, ^?

```
&& : 1
||  : 0
|   : 0
^   : 0
```

# 2 Problem 2

Suppose OpenMP did not have the reduction clause. Show how to implement an efficient parallel reduction by adding a private variable and using the critical pragma.

```
/* File:      problem2.cpp
 * Purpose: Alternates sign of integer added to sum
 *
 *           sum = 0 + 1 + -2 + 3 + -4...
 *
 * Compile: g++ -Wall -fopenmp -o problem2 problem2.cpp -std=c++11
 *          g++ -Wall -fopenmp -o problem2 problem2.cpp -DDEBUG -std=c++11
 * Run:     ./problem2
 *
 * Input:    none
 * Output:   Times for each of the three runs
 *
 * Notes:
 * 1. If ran with the -DDEBUG flag you can see what the sum should
    be based on n
 *
 */

#include <inttypes.h> // Better integer functionality
#include <stdio.h>    // Printing to console
#include <omp.h>       // Multithreading
#include <chrono>      // High precision clock

using namespace std::chrono;

// Global
uint8_t  thrds  = omp_get_num_procs();

int main(int argc, char* argv[]) {
    uint8_t times = 20;
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> no_omp_time = duration_cast<duration<double>>\
        (high_resolution_clock::now() - high_resolution_clock::now());
    duration<double> omp_time = duration_cast<duration<double>>\
        (high_resolution_clock::now() - high_resolution_clock::now());
    duration<double> no_reduc_time = duration_cast<duration<double>>\
        (high_resolution_clock::now() - high_resolution_clock::now());
    for(uint8_t j = 0; j < times; ++j)
```

```

{
    uint64_t    n        = 80000000,
               k        = 0;
    int64_t     sum      = 0;

    // RESET for baseline
    t1 = high_resolution_clock::now();

    for (k = 0; k < n; ++k)
    {
        sum += ((k & 1) == 0 ? 1.0 : -1.0) * k;
    }

    t2 = high_resolution_clock::now();
    no_omp_time += duration_cast<duration<double>>(t2 - t1);
#ifdef DEBUG
    if (j == 0){
        printf("No OMP sum      : %" PRIi64 "\n", sum);
    }
#endif

    // RESET for reduction + omp
    sum = 0;
    t1 = high_resolution_clock::now();

    #pragma omp parallel for num_threads(thrds) reduction(+: sum) private(k)
    for (k = 0; k < n; ++k)
    {
        sum += ((k & 1) == 0 ? 1.0 : -1.0) * k;
    }

    t2 = high_resolution_clock::now();
    omp_time += duration_cast<duration<double>>(t2 - t1);
#ifdef DEBUG
    if (j == 0){
        printf("OMP sum          : %" PRIi64 "\n", sum);
    }
#endif

    // RESET for no reduction
    sum = 0;
    k = 0;
    t1 = high_resolution_clock::now();

    #pragma omp parallel num_threads(thrds)
    {
        int64_t thread_sum = 0;
        #pragma omp for
        for(uint64_t i = k; i < n; ++i){
            // Locally (privately) runs this
            thread_sum += ((i & 1) == 0 ? 1.0 : -1.0) * i;
        }
    }
}

```

```

        #pragma omp critical
        sum += thread_sum;
    }

    t2 = high_resolution_clock::now();
    no_reduc_time += duration_cast<duration<double>>(t2 - t1);
#ifdef DEBUG
    if (j == 0){
        printf("No Reduc sum : %" PRIi64 "\n", sum);
    }
#endif
}

printf("Averages over %" PRIu8 " runs:\n", times);
printf("No OMP      : %.14f\n", no_omp_time.count() / times);
printf("OMP        : %.14f\n", omp_time.count() / times);
printf("No Reduc   : %.14f\n", no_reduc_time.count() / times);

return 0;
}

```

```

kyle@HW1$ g++ -Wall -fopenmp -o problem2 problem2.cpp -DDEBUG -std=c++11
kyle@HW1$ ./problem2
No OMP sum      : -40000000
OMP sum         : -40000000
No Reduc sum    : -40000000
Averages over 20 runs:
No OMP      : 0.40371242310000
OMP         : 0.06024930730000
No Reduc    : 0.06116008355000
kyle@HW1$

```

Figure 1: Example debug output.

```

kyle@HW1$ ./problem2
Averages over 20 runs:
No OMP      : 0.40169680075000
OMP         : 0.05187247365000
No Reduc    : 0.05121839645000
kyle@HW1$ ./problem2
Averages over 20 runs:
No OMP      : 0.40076352375000
OMP         : 0.05140341830000
No Reduc    : 0.05126510895000
kyle@HW1$ ./problem2
Averages over 20 runs:
No OMP      : 0.40068608615000
OMP         : 0.05138620015000
No Reduc    : 0.05121338355000

```

Figure 2: Better performance without reduction.

As can be seen in the figures the sums are performing as expected. An interesting, and expected outcome is that in Figure 1 it takes 0.06 seconds to run *OMP* and *No Reduc* but in Figure 2 it takes 0.05 seconds. The *No OMP* takes 0.40 seconds regardless. The reason for this behavior is that *OMP* uses the cores you give it and at the time of recording the first figure the browser was open and running a video. When I recorded the second Figure I had closed my browser to maximize performance for multi-core processing. The *No OMP* section of code was only running on one core, so it did not care that I had a video playing.

### 3 Problem 3

Please see my [repository](#) for the full code breakdown for [this problem](#).

#### 3.1 Problem 3a

```
#pragma omp for
for(i = 0; i < (long) sqrt(x); ++i) {
    a[i] = 2.3 * i;
    if (i < 10)
        b[i] = a[i];
}
```

#### 3.2 Problem 3b

This code section is not suitable for OpenMP because of the `&&` operator in comparison. Per the [OpenMP documentation](#) §2.6, p53:

*test-expr*      One of the following:  
                  *var relational-op b*  
                  *b relational-op var*

*relational-op*   One of the following:  
                  <  
                  <=  
                  >  
                  >=

#### 3.3 Problem 3c

This code can be ran with *OMP* but it is dependent on whether or not *foo()* is threadsafe.

```
#pragma omp for
for(i = 0; i < n; ++i) {
    a[i] = foo(i);
}
```

#### 3.4 Problem 3d

This problem is similar to 3c in that it is dependent on *foo()*. Nothing else is preventing this from being made parallel. Only comparisons and assignments are used. Access of *b[i]* is fine because nothing is writing to it so there is no opportunity for trouble.

```
#pragma omp for
for(i = 0; i < n; ++i) {
    a[i] = foo(i);
    if(a[i] < b[i])
        a[i] = b[i];
}
```

#### 3.5 Problem 3e

Cannot be ran in parallel because the *break*.

### 3.6 Problem 3f

Standard use of *OMP*, though I feel that it didn't actually use *OMP*. I tried to throw a reduction at it but I kept getting an error about *dotp* being private. I threw a private local *sum* into the for loop and added *pragma omp critical* followed by *dotp += sum* but it gave no performance increase.

```
dotp = 0;
#pragma omp for
for(i = 0; i < n; i++){
    dotp += a[i] * b[i];
}
```

### 3.7 Problem 3g

I thought this one was going to have trouble because of the  $i \leftarrow k$  portion but testing showed the values matched for the sequential sanity test and the *OMP* portion. Of note though, *OMP* was significantly slower than sequential until I ramped up  $k$ .

```
#pragma omp for
for(i = k; i < 2 * k; ++i){
    a[i] = a[i] + a[i-k];
}
```

### 3.8 Problem 3h

This problem is the same as 3g except we are using what is essentially a shared constant,  $b$  and we are running through the whole list, depending on what  $k$  is. Other than that it behaves the same. Passed sanity check and ran without issue.

```
#pragma omp for
for(i = k; i < n; ++i){
    a[i] = b * a[i-k];
}
```



## 4 Problem 4

*Given a task that can be divided into  $m$  subtasks, each requiring one unit of time, how much time is needed for an  $m$ -stage pipeline to process  $n$  tasks?*

How many of the subtasks are dependent on other subtasks? If all  $m$  subtasks require the previous subtask then  $m * n$  time is needed to process  $n$  tasks. If they can all be ran independently and if we have enough threads we can run it in  $n$  time. It is more likely that we will have some combination of subtasks that require other subtask completion, therefore the only blanket assumption that can be safely made is that it will take  $m * n$  time.

## 5 Problem 5

If the address of the nodes in a hypercube has  $n$  bits. How many nodes can it be at the most and how many edges does each node have? Give an algorithm that routes a message from node  $u$  to node  $v$  in this  $k$ -node hypercube in no more than  $\log(k)$  steps.

Hypercubes are generalized by dimensionality. A hypercube address containing  $n$  bits will exist in  $n$  dimensions ( $d$ ). The maximum number of nodes is determined by  $d$ , and is defined as  $2^d$ . A trait of hypercubes is that each node will have  $d$  edges. The total amount of edges would be defined as  $d * 2^{d-1}$ . The number of nodes you have to travel between  $u$  and  $v$  is equal to the number of differing bits.

For example:

```
000  ⇒  111  : 3 edges
001  ⇒  100  : 2 edges
110  ⇒  011  : 2 edges
```

A traversal algorithm would be along the lines of Algorithm 1

---

**Algorithm 1** Hypercube Traversal

---

- Move from node  $u$  to  $v$

$i \leftarrow 0$

$current\_bit \leftarrow u[i]$

$left\_bit \leftarrow u[i]$

**while**  $i < dimension(hypercube)$  **do**

$current\_bit \leftarrow u[i]$

**if**  $current\_bit = v[i]$  **then**

$i \leftarrow i + 1$

**else**

$route(current\_bit)$

▷ Routes to neighbor node containing the correct bit

**end if**

**end while**

---

If built properly there will only be one node the algorithm can route to during the  $route(current\_bit)$  section. Essentially we look at the left-most bit and if it is the same we move to the next bit, if it is different we must move to a new node. There will only be one node available because if they are on the same dimension they would have the same bit in that slot. For example on a 3-dimensional hypercube  $000 \Rightarrow 100$  looks at the left bit and sees it is on a different dimension. There is only one node connected to 000 that follows the format  $1xx$ , so that is the only place it can go. If we wanted to route from  $000 \Rightarrow 010$  we would identify we are on the correct 3rd dimension, then look at the middle bit. From there we would see a need to move in the second dimension. There is only one connected node that matches in the second dimension, so that is where it would go.

## 6 Graduate Assignment

*Do a search on Shuffle-exchange network topology. Draw the network with 16 processor nodes (carefully numbering each node binary and showing what is a shuffle link, what is an exchange link). If  $k$  is the number of digits in the binary address, how many nodes ( $n$ ) are there? With  $n$  nodes what is the diameter ( $d$ ) of the networks, the bisection width ( $b$ ) and how many edges/node?*

There are a plethora of ways to make a shuffle-exchange network topology, such as:

- Banyan
- Omega
- Indirect binary

In the shuffle-exchange we can expect the following:

- With  $k$  bits in the address we can expect there to be  $2^k$  input nodes  $n$
- We can expect  $n/2 * \log_2(n)$  shuffle-exchange nodes.
- Given the above, we expect a diameter  $d$  to equal  $k$ .
- The bisection width is the number of inputs,  $n$ .
- There are 4 edges per node.

So for our example problem with 16 processor nodes:

- We have  $k = 4$ ,  $n = 16$
- $16/2 * 4$ , or 32 shuffle-exchange nodes
- A diameter  $d = 4$
- Bisection width of 16
- 4 edges per shuffle-exchange node