# Homework 1

MacMillan, Kyle

September 28, 2018

# Contents

## List of Figures

## List of Algorithms

## 1 Problem 1

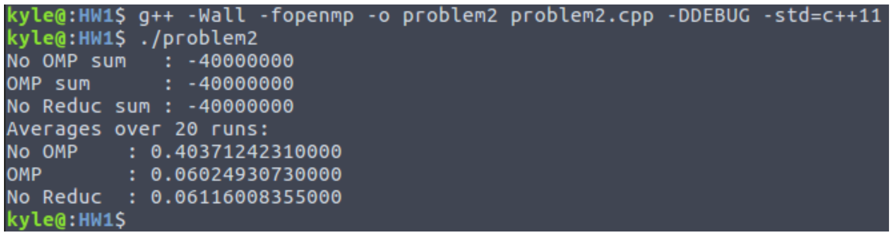
*What are the identity values for the operators: &&, ||, |, ^?*

| | |
|---|---|
| && | : 1 |
| \|\| | : 0 |
| \| | : 0 |
| ^ | : 0 |

## 2 Problem 2

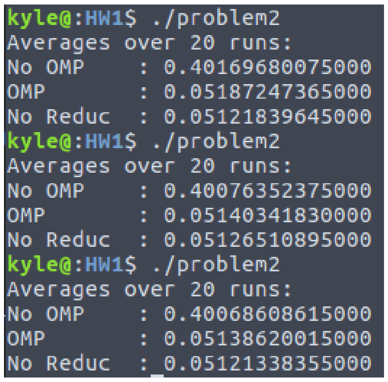
*Suppose OpenMP did not have the reduction clause. Show how to implement an efficient parallel reduction by adding a private variable and using the critical pragma.*

```
/* 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:<br>*var relational-op b*<br>*b relational-op var* |
| *relational-op* | One of the following:<br>&lt;<br>&lt;=<br>&gt;<br>&gt;= |

### 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

asdf

### 3.5 Problem 3e

asdf

### 3.6 Problem 3f

asdf

### 3.7 Problem 3g

asdf

### 3.8 Problem 3h

asdf

## 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
001   =>   100   : 2
110   =>   011   : 2
```

Therefore a traversal algorithm would be:

**Algorithm 1** Move from node $u$ to $v$

```
• Initialize Variables
$i \leftarrow 0$
$current\_bit \leftarrow u[i]$
$left\_bit \leftarrow u[i]$
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 to 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 to 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?*

When you first drew a shuffle-exchange network, also sometimes called a butterfly network, I immediately thought that looked like a sorter I came up with in 300. I later learned that sort wasn't something I had uniquely developed and was in fact a bitonic sort.

If there are $k$ digits in the address then there will be a maximum of $(k + 1) * 2^k$ nodes. With $n$ nodes the diameter will be $2 * k$. If the network is setup to wrap around then the diameter could be $k$. The bisection is $k + 1$. Most nodes would contain $4 edges$, where two come from "below" and two come from "above". The boundary nodes would contain $2 edges$, ethier connecting "up" or "down".