Kenneth Willeford
CSCI 176
HW2

1. Tserial = n, Tparallel = n/p + log2(p)
   If we increase p by factor of k
   Tserial = n, Tparallel = n/(p*k) + log2(p*k)
   If we are looking to maintain constant efficiency then Tserial = Tparallel or
   n/(p*k) + log2(p*k) = n
   n - n/(p*k) = log2(p*k)
   n*(p*k) - n = log2(p*k)*(p*k)
   It is difficult to isolate n although wolfram finds a solution
   wolframalpha.com/input/?i=solve+for+n+in+n+-+n%2F(p*k)+%3D+log2(p*k)
   For my remaining purposes I will use it in this form.
   "How much should we increase n by if we double the number of processes from 8 to 16?"
   nIncrease = nAfter - nBefore
   nAfter: 16n - n = 16*log2(16) ; 15n = 16*4 ; n = 64/15 = 4.27
   nBefore: 8n - n = 8*log2(8) ; 7n = 8*3 ;n = 24/7 = 3.43
   nIncrease = 4.27 - 3.43 = 0.84
   The system requires an increase of input size in order for performance to not change. So it is scalable.
2. "Is a program that obtains linear speedup strongly scalable? Explain your answer."
   I would say "yes." A linear speedup would imply an execution time of n/p which would mean that there are no diminishing returns on further parallelization. If this isn't strongly scalable I don't know what is.
3. Textbook 4.8
   A. This is the classic deadlock example. Deadlock occurs.
   B. With Busy-Waiting the issue would still occur. (Both are waiting on eachother to finish up.)
   C. With a Semaphore the issue can be prevented if the semaphore allows up to two accesses to its synchronization variable. However it would be unsafe if the deadlock example scaled up.

4. Textbook 4.16

Left Operand: 8000x8000 Matrix
Right Operand:Xx8000 Matrix where X is some arbitrary integer
Output Matrix: Xx8000
4 threads: thread0 is on p0, thread2 is on p1
Cache Line = 64bytes (8 doubles)
Is it possible for false sharing to occur along vector y? Why?

Yes, and it depends on the width of the output Matrix which is tied to X.
In order for there to be no false sharing between thread0 and thread2 there must be no way for their cache lines to overlap. There is 1 row separating the two. So we must find an overflow condition. If X*2*8bytes < 64bytes then false sharing will occur.
We can see this at the boundary case
[][][][] Thread0
[][][][] Thread1
[][][][] Thread2
If 64bytes < 64bytes, the condition is false (and we can see that Thread0 just barely keeps out of thread 2.)
[][][]Thread0
[][][]Thread1
[][][]Thread2
If 48bytes < 64 bytes: The condition holds true, and sure enough if we check by hand Thread0's cache line is within Thread2's contents.

What if thread 0 and thread 3 are assigned to different processors, Is False Sharing Possible?

Yes, the condition is just adjusted slightly.

If X*3*8bytes < 64bytes then false sharing will occur.

Let's confirm with our boundary cases.

[][] Thread0

[][] Thread1

[][] Thread2

[][] Thread3

2*3*8bytes < 64bytes ; 48bytes < 64 bytes, false sharing will occur. Checking by hand confirms.

[][][] Thread0

[][][] Thread1

[][][] Thread2

[][][] Thread3

3*3*8bytes < 64 bytes ; 72 bytes < 64 bytes, false sharing will not occur. Checking by hand confirms.

5. Submitted Alongside

```cpp
//////////////////
//// Purpose: Estimate definite integral (or area under curve) using trapezoidal rule.
//// - this version uses void type thread function, which uses global_result as ref para.
////
//// Input:   a, b, n //assume n is evenly divisible by num_threads
//// Output:  estimate of integral from a to b of f(x) using n trapezoids.
////
//// Compile/run: $> g++ -fopenmp -o xxx hw2prog1.cpp
////              $> xxx 4 //any number of threads
////
//// Test Input         0 100000000 1000000
//// Expected Output    3.33333e+023
//////////////////

#include <cstdlib>
#include <iostream>
#include <omp.h>
using namespace std;

double f(double); //f(x)=x^2
double Trap(double, double, int);

int main(int argc, char* argv[])
{
   double  global_result = 0.0;
   double  a, b; //left and right end points
   int     n;    //total number of trapezoids

   int thread_count = atoi(argv[1]); //command line arg
   cout<<"Enter a, b, and n -- each separated by a space:\n";
   cin>>a>>b>>n;
   if(n % thread_count != 0)
   { cerr<<"n should be evenly divisible by "<<thread_count<<endl;
     exit(0);
   }
   #pragma omp parallel num_threads(thread_count)
   global_result += Trap(a, b, n); //global_result is ref para

   cout<<"With n="<<n<<" trapezoids, our estimate of integral from "
       <<a<<" to "<<b<<" is "<<global_result<<endl;

  return 0;
}  /* main */

////////////////////// function f(x) = x * x ;
double f(double x)
{
   return x*x;
}//f

////////////////////// thread function for Trap
//// Purpose: Use trapezoidal rule to estimate definite integral
//// Input args:
////    a, b : left, right endpoints
////    b: right endpoints
////    n: number of trapezoids
//// Output arg (ref para):
////    globla_result: estimate of integral from a to b of f(x)
//////////////////
double Trap(double a, double b, int n)
{
   double  h, x, my_result;
   double  local_a, local_b;
```

```
64      int i, local_n;
65      int my_rank = omp_get_thread_num();
66      int thread_count = omp_get_num_threads(); //returns the count passed from pragma
        parallel
67
68      h = (b-a)/n;
69      local_n = n/thread_count;
70      local_a = a + my_rank*local_n*h;
71      local_b = local_a + local_n*h;
72      my_result = (f(local_a) + f(local_b))/2.0;
73      for (i = 1; i <= local_n-1; i++)
74      { x = local_a + i*h;
75        my_result += f(x);
76      }
77      my_result = my_result*h;
78      #pragma omp critical
79       cout<<"Thread_"<<my_rank<<", local_result = "<<my_result<<endl;
80
81      return my_result;
82    }//
83
```

```
Enter a, b, and n -- each separated by a space:
Thread_0, local_result = 3.33333e+023
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```

```
Enter a, b, and n -- each separated by a space:
Thread_1, local_result = 2.91667e+023
Thread_0, local_result = 4.16667e+022
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```

```
Enter a, b, and n -- each separated by a space:
Thread_1, local_result = 3.64583e+022
Thread_2, local_result = 9.89583e+022
Thread_3, local_result = 1.92708e+023
Thread_0, local_result = 5.20833e+021
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```

```cpp
//////////////////
//// Purpose: Estimate definite integral (or area under curve) using trapezoidal rule.
//// - this version uses void type thread function, which uses global_result as ref para.
////
//// Input:   a, b, n //assume n is evenly divisible by num_threads
//// Output:  estimate of integral from a to b of f(x) using n trapezoids.
////
//// Compile/run: $> g++ -fopenmp -o xxx hw2prog1.cpp
////              $> xxx 4 //any number of threads
////
//// Test Input         0 100000000 1000000
//// Expected Output    3.33333e+023
//////////////////

#include <cstdlib>
#include <iostream>
#include <omp.h>
using namespace std;

double f(double); //f(x)=x^2
double Trap(double, double, int);

int main(int argc, char* argv[])
{
   double  global_result = 0.0;
   double  a, b; //left and right end points
   int     n;    //total number of trapezoids

   int thread_count = atoi(argv[1]); //command line arg
   cout<<"Enter a, b, and n -- each separated by a space:\n";
   cin>>a>>b>>n;
   if(n % thread_count != 0)
   { cerr<<"n should be evenly divisible by "<<thread_count<<endl;
     exit(0);
   }
   #pragma omp parallel num_threads(thread_count) \
     reduction(+: global_result)
       global_result += Trap(a, b, n); //global_result is ref para
   cout<<"With n="<<n<<" trapezoids, our estimate of integral from "
       <<a<<" to "<<b<<" is "<<global_result<<endl;

  return 0;
}  /* main */

////////////////// function f(x) = x * x ;
double f(double x)
{
   return x*x;
}//f

////////////////// thread function for Trap
//// Purpose: Use trapezoidal rule to estimate definite integral
//// Input args:
////    a, b : left, right endpoints
////    b: right endpoints
////    n: number of trapezoids
//// Output arg (ref para):
////    globla_result: estimate of integral from a to b of f(x)
//////////////////
double Trap(double a, double b, int n)
{
   double  h, x, my_result;
   double  local_a, local_b;
```

```
64      int i, local_n;
65      int my_rank = omp_get_thread_num();
66      int thread_count = omp_get_num_threads(); //returns the count passed from pragma
        parallel
67
68      h = (b-a)/n;
69      local_n = n/thread_count;
70      local_a = a + my_rank*local_n*h;
71      local_b = local_a + local_n*h;
72      my_result = (f(local_a) + f(local_b))/2.0;
73      for (i = 1; i <= local_n-1; i++)
74      { x = local_a + i*h;
75        my_result += f(x);
76      }
77      my_result = my_result*h;
78      #pragma omp critical
79       cout<<"Thread_"<<my_rank<<", local_result = "<<my_result<<endl;
80
81      return my_result;
82    }//
83
```

```
Enter a, b, and n -- each separated by a space:
Thread_0, local_result = 3.33333e+023
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```

```
Enter a, b, and n -- each separated by a space:
Thread_0, local_result = 4.16667e+022
Thread_1, local_result = 2.91667e+023
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```

```
Enter a, b, and n -- each separated by a space:
Thread_0, local_result = 5.20833e+021
Thread_1, local_result = 3.64583e+022
Thread_2, local_result = 9.89583e+022
Thread_3, local_result = 1.92708e+023
With n=1000000 trapezoids, our estimate of integral from 0 to 1e+008 is 3.33333e+023

```