5.8 refers to 5.5 and 5.7. We have seen solutions to these problems in class.

5.5, 5.6 and 5.7 all refer to the problem of adding n numbers.

In serial case, \( T_1 = t_c \cdot n \)

In parallel case
\[
T_p = (t_c \cdot n/p) + (t_c + t_{comm}) \cdot \log(p)
\]
\[
= (t_c \cdot n + p \cdot \log(p) \cdot (t_c + t_{comm}))/p = (W + T_0)/p
\]
\[
T_s = p \cdot \log(p) \cdot (t_c + t_{comm})
\]

We are also told that \( t_c = 1 \) and \( t_{comm} = 10 \), so \( T_s = (n/p) + 11 \cdot \log(p) \)

Speed up = \( T_1/T_s \rightarrow \) this is normal speedup

Scaled speedup (5.6)
\[
T_{p \text{-scaled}} = p \cdot (t_c \cdot n/p) + (t_c + t_{comm}) \cdot \log(p) \rightarrow \text{communication cost does not change}
\]
\[
T_{p \text{-scaled}} = p \cdot n/p + 11 \cdot \log(p)
\]
Figure 5.2 Efficiency corresponding to the speedups in Figure 5.1.

As we discussed, scaled efficiency is less than 1

5.14 refers to concurrency vs iso-efficiency. We discussed these terms in class

The maximum parallelism in an algorithm \( \Rightarrow \) degree of concurrency
\[ = \text{the maximum number of processors we can use} \]

Iso-efficiency is related what is the number of processors that can be used effectively
Note that to keep the same efficiency we need to increase work
But if the increased work is sublinear (grows slower than \( p \))
then at some point, processors will be idle

You are asked to find \( T_{\text{min}} \) \( \Rightarrow \) the minimum time that can be achieved with as many processors as you can use

In this problems, we are told that two different algorithms have the same Iso-efficiency
\[ \Rightarrow \text{same } T_p = (W+T_0)/p \]

But they have different degrees of concurrency \( \Rightarrow \) different maximum number of processors
So, compute the \( T_{\text{min}} \) using different maximum number of processors
CSCE 5160 Parallel Processing

Review
Scalability Metrics: Iso-Efficiency
Scaled Speedup
Liner and Memory scaling
Time scaling ➔ Gustavson’s law

Key MPI Functions ➔ Capter 6 contains more details

```
int MPI_Init (int * argc, char ** argv[]);
int MPI_Finalize ();
int MPI_Comm_Size (MPI_Comm comm, int * number_of_processes);
int MPI_Comm_rank( MPI_Comm comm, int * my_process_id);
```

```
int MPI_Send (void * message_data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator);
int MPI_Recv (void * message_data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status * status);
int MPI_Bcast (void * message, int count, MPI_Datatype data_type, int root, MPI_Comm communicator, /*sending process */ MPI_Comm_Comm /*receivers*/);
```
CSCE 5160 Parallel Processing

```c
int MPI_Scatter (void *send_data, /* data source */
    int send_count, /* count of items to be sent */
    MPI_Datatype send_type, /* data type sent */
    void* recv_data, /* storage for received data */
    int recv_count, /* count of items to be received */
    MPI_Datatype recv_type, /* data type received */
    int sender, /* sender or root */
    MPI_Comm comm /* communication group */);

int MPI_Gather (void *send_data, /* data source */
    int send_count, /* count of items to be sent */
    MPI_Datatype send_type, /* data type sent */
    void* recv_data, /* storage for received data */
    int recv_count, /* count of items to be received */
    MPI_Datatype recv_type, /* data type received */
    int receiver, /* receiver or root */
    MPI_Comm comm /* communication group */);
```

Note: Bcast, Scatter and Gather functions are Executed by the senders and receivers.

YOU CAN ONLY SCATTER THE SAME NUMBER OF ITEMS TO EACH PROCESS
YOU CAN ONLY RECEIVE THE SAME NUMBER OF ITEMS FROM EACH PROCESS

---

CSCE 5160 Parallel Processing

```c
int MPI_Reduce (void *recv_data, /* data source */
    void* accumulated_data, /* result */
    int count, /* per process count */
    MPI_Datatype send_type, /* data type sent */
    MPI_Op operation, /* operation like, sum, min */
    int receiver, /* receiver or root */
    MPI_Comm comm /* communication group */);
```

All processes execute the same command
The result (or accumulated data) is only at the root
Count refers to amount sent by each node
Result stays with only root

Operations with MPI_Reduce

- MPI_MAX, MPI_MIN
- MPI_MAXLOC
- MPI_MINLOC
- MPI_SUM, MPI_PROD
- MPI_BAND (Bitwise AND)
- MPI_BAND (Logical AND)
- MPI_BAND (Bitwise OR)
- MPI_BAND (Logical OR)
- MPI_BAND (Bitwise XOR)
- MPI_BAND (Logical XOR)

We can view MPI_Reduce as
for (j=1; j<n; j++) {
    MPI_Recv();
    perform_accumulation;
}
```

```
### CSCE 5160 Parallel Processing

```c
int MPI_Allreduce (void * send_data, /* data source
                   void * accumulated_data, /* result
                   int count, /* per process count
                   MPI_Datatype send_type, /* data type sent
                   MPI_Op operation, /* operation like, sum, min
                   MPI_Comm comm /* communication group
                   );
```

Let us now think of doing the same using OpenMP

- Chapter 7 discusses Pthread and OpenP

Remember here we use shared memory
- That is X and Y vectors are shared
- No need to send out n/p elements of X and Y to each processor

What about result which is the sum of all partial results?

We need to use “locks” so that one processor at a time adds its value to current result

---

### CSCE 5160 Parallel Processing

Each thread executes the following code

- If we assume x and y are in shared memory
- Each thread needs to now which x and y elements it must access
- Each thread accesses n/p elements

```c
float dot_product (float x[], float y[], int n)
{
    int my_start, my_end, my_ID, num_threads;
    float my_sum = 0.0;
    my_ID = omp_get_thread_num(); /* get my thread id
    num_threads = omp_get_num_threads(); /* total # threads

    my_start = my_ID*(n/num_threads);
    my_end = my_start + (n/num_threads);
    for (i=my_start; i< my_end; i++)
        my_sum = my_sum + x[i]*y[i];
    return my_sum;
}
```
Now let us see how to create threads and “reduce individual sum values”

```c
#pragma omp parallel for
for (j= 0; j<num_threads; j++)
{
    dot_product (&x[], &y[], n);

#pragma omp critical
    {sum = sum+my_sum;}
}
```

Critical → critical section
Runtime system for OpenMP creates necessary locks and allows only one thread at a time to execute the statements associated with the critical section.

If there are multiple critical section in your code, you need to tag or name each.

If there is only statement inside the critical section we can use

```c
#pragma omp atomic
    sum = sum+my_sum;
```

Fork and Join model. Note OpenMP forks thread when it sees parallel sections

```c
#pragma omp parallel
```

Most frequently, we define loop level parallelism

```c
#pragma omp parallel for
for (…….)
{…….}
```

Here we are asking for the creation of parallel threads for the loop iterations.
If there are fewer threads than iterations, can control how threads are created

np/iterations per thread
cyclical allocation
more advanced methods
We can also describe parallel tasks using sections

```c
#pragma omp parallel {
    #pragma omp sections {
        #pragma omp section {
        ...
        #pragma omp section {
        ...
    }
```

Remember OpenMP uses Fork and Join.
- Fork at the beginning of `#pragma omp parallel`
- At the end of the parallel construct, all created threads join
- Join is sometimes called Barrier Synchronization

Consider an application like Jacobi linear equation solver

A set of n equations containing n unknown variables:

\[
\begin{align*}
\begin{bmatrix}
    a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \\
    a_{n,0} & a_{n,1} & a_{n,2} & \cdots & a_{n,n-1} \\
    & \ddots & \ddots & \ddots & \ddots \\
    & & \ddots & \ddots & \ddots \\
    & & & \ddots & \ddots \\
\end{bmatrix}
\begin{bmatrix}
    x_0 \\
    x_1 \\
    \vdots \\
    x_{n-1} \\
\end{bmatrix}
&=
\begin{bmatrix}
b_{n-1} \\
b_{n} \\
\vdots \\
b_{n-1} \\
b_{0} \\
\end{bmatrix}
\end{align*}
\]

This can also be written as \( A^*X = B \)
Where A is a nxn matrix, X and B are 1xn vectors

Let us consider the ith equation:

\[ a_{i,0} x_0 + a_{i,1} x_1 + a_{i,2} x_2 + \cdots + a_{i,i} x_i + \cdots + a_{i,n-1} x_{n-1} = b_i \]

This can be written as

\[ x_i = b_i - \frac{1}{a_{i,i}} \sum_{j \neq i} a_{i,j} x_j \]

Note that this expression shows each unknown as a function of other unknowns.
So, in order to solve this numerically, we will devise an iterative solution.
That is we will start with some assumed values for each \( x_i \), and then update it using corrections.
For example, we can initially set \( x_i = b_i \). Then we will update each \( x_i \) using the above equation.
Consider a simple program to do this:

```c
for (i = 0; i < n; i++)
    x[i] = b[i];

for (repeat = 0; repeat < limit; repeat++) {
    /* how many times to repeat
    for (i = 0; i < n; i++) {
        sum = 0;
        for (j = 0; j < n; j++)
            if (i != j) sum = sum + a[i][j]*x[j];
        new_x[i] = (b[i] - sum)/a[i][i];
    }
    for (i = 0; i < n; i++)
        x[i] = new_x[i];
    }
```

How do we code this in a shared memory system (say using OMP)?

We will assume that we create a separate thread to compute each x[i].
What we have to do is coordinate the shared data in array x?

Two steps that we must be careful about:

- The new value of x[i] **cannot be updated** until all threads have used old values
- New values of x[i] **cannot be read** by threads until all x[i]'s are updated

So,

- After each thread computes its new_x[i], it must wait for all threads to complete computing their new_x's
- Then each thread must update the x[i] using the new_x[i] and must wait until all threads completed the updates before reading x[i] values

How do we code this in a message passing (say using MPI)?

Again, let us assume that each x[i] is computed by a different process.

After each thread computes a new x[i] value, it must broadcast its data to all other processes or threads.

We can use all to all broadcast.
CSCE 5160 Parallel Processing

When dealing with shared memory
We have to wait for all processes to complete computing new values

```c
#pragma omp parallel for
for (i = 0; i < n; i++) { x[i] = b[i];}
```

for (repeat = 0; repeat < limit; repeat++)
```
#pragma omp parallel for
for (i = 0; i < n; i++) {
    sum = 0;
    for (j = 0; j < n; j++)
        if (i != j) sum = sum + a[i][j]*x[j];
    new_x[i] = (b[i] - sum) / a[i][i];
}
```

```
#pragma omp parallel for
for (i = 0; i < n; i++)
    x[i] = new_x[i];
```

OpenMP automatically inserts “barriers” at the end of each parallel portion

```c
#pragma omp parallel for
for (i = 0; i < n; i++)
    x[i] = b[i];
```

for (repeat = 0; repeat < limit; repeat++)
```
#pragma omp parallel for
for (i = 0; i < n; i++) {
    sum = -a[i][i]*x[i]; /* eliminate test
    for (j = 0; j < n; j++)
        sum = sum + a[i][j]*x[j];
    new_x[i] = (b[i] - sum) / a[i][i];
}
```

```
#pragma omp parallel for
for (i = 0; i < n; i++)
    x[i] = new_x[i]; /* if we move this into previous loop, we need a barrier */
```
Of course we can use this type of parallelism only if the computations are independent results of one iteration are not used by another (loop carried dependency) shared variables are handled using mutual exclusion

Private and shared variables

- Private variables = a separate memory location for each thread
- Shared = same address for all threads

OpenMP default is shared
To create private use `private(<variable-list>)`

```c
#pragma omp parallel for private (local_min)
for (i=0; i< BLOCK_SIZE(id,p,n); i++)
  for(j=0; j<n; j++)
    local_min = MIN(a[i][j], a[i][k]+temp[j]);
  a[i][j] = MIN(a[i][j], a[i][k]+local_min);
```

A few more constructs

`firstprivate` clause
All private variables “inherit” an initial value

Consider the following loop
```c
x[0] = some_complex_function();  /*initialization*/
for (i=0; i<n; i++)
  {
    for (j=1; j<4; j++)
      x[j] = g(i, x[j-1]);
    answer[i] = x[i]- x[3];
  }
```

If we want to parallelize the OUTER `i` loop,
We want `x[j]` to be private to each iteration/thread,
but we do not want to initialize `x[0]` in each thread
x[0] = some_complex_function();
#pragma omp parallel for private (j) firstprivate (x)
for (i=0; i<n; i++)
{
    for (j=1; j<4; j++)
        x[j] = g(i, x[j-1]);
    answer[i] = x[i] - x[3];
}

Likewise, if we want the final value(s) of all parallel iterations to be shared, we can use lastprivate clause

For example consider
for (i=0; i<n; i++)
{
    x[0] = 1.0;
    for (j=1; j<4; j++)
        x[j] = x[j-1]*(i+1);
}

n_cubed = x[3];

Now the final x is not private but shared value