Show Your Work! Point values are in square brackets. There are 30 points possible.

1. In two sentences or less, explain why integrated circuit manufacturers can no longer increase the speed of processors by simply increasing processor clock speed. [2 points]

   Increasing clock speed requires increased power, and most of the additional power is dissipated as heat. But when processors get too hot, they become unreliable.

2. A parallel program has the following run-times when run with problem size \(10^6\).

   \[
   \begin{array}{c|c|c|c|c|c}
   \text{Number of procs} & 1 & 2 & 4 & 8 & 16 \\
   \hline
   \text{Run-time (secs)} & 200 & 100 & 75 & 50 & 40 \\
   \end{array}
   \]

   (a) Find the speedup when the program is run with 16 processes.

   (b) Find the efficiency when the program is run with 8 processes.

   (c) On the basis of this data is the program scalable? Why?

   [3 points]

   (a) Speedup = \(\frac{200}{40} = 5\).

   (b) Efficiency = \(\frac{200}{(50 \times 8)} = 0.5\)

   (c) The program is not scalable. The following table shows the efficiencies for different numbers of processes:

   \[
   \begin{array}{c|c|c|c|c|c}
   \text{Number of procs} & 1 & 2 & 4 & 8 & 16 \\
   \hline
   \text{Efficiency} & 1.0 & 1.0 & 0.67 & 0.5 & 0.31 \\
   \end{array}
   \]

   So we see that as the number of processes is increased the efficiency decreases dramatically. So the program is not scalable.
3. Bob is writing an MPI program that has a bug in it: process 1 sends a message to process 0 using `MPI_Send`, but process 0 never calls `MPI_Recv`. If Bob’s MPI software implements `MPI_Send` using synchronous mode, what would you expect to happen when process 1 calls `MPI_Send`? [2]

Process 1 will “hang.” Since it’s using synchronous mode, it will block until process 0 starts to receive, and since process 0 doesn’t receive, 1 will remain blocked indefinitely.

4. Sally has written a C program containing the following code:

```c
int w, x, y, z;
scanf("%d %d", &x, &y);
printf("x = %d, y = %d\n", x, y);
z = First_function(x, y);
printf("z = %d ", z);
w = Second_function(x, y, z);
printf("w = %d ", w);
```

When she runs the program it crashes with the dread “Segmentation fault:”

```
x = 1, y = 2
Segmentation fault
```

Since the output of the second `printf` statement `printf("z = %d ", z)` never appears on the terminal screen, Sally reasons that the problem must be with `First_function`. Is Sally’s reasoning correct? Explain your answer in two sentences or less. [2]

No. Since the output produced by `printf` is buffered, it’s possible that the second `printf` was executed, but when the program crashed, the contents of the buffer were lost. So the program may have crashed after the second `printf` — e.g., in the call to `Second_function`.

Figure 1: Variables in Problem 5
5. Find the output of the following program. [4]

```c
#include <stdio.h>
#include <stdlib.h>

int star(int x, int* y_p, int* a_p, int** b_pp);

int main(void) {
    int x = 3, y = 5, z;
    int* a_p = &x;
    int* b_p = malloc(sizeof(int));

    *b_p = 7;
    z = star(x, &y, a_p, &b_p);
    printf("z = %d, x = %d, y = %d, *a_p = %d, *b_p = %d\n",
            z, x, y, *a_p, *b_p);

    return 0;
} /* main */

int star(int x, int* y_p, int* a_p, int** b_pp) {
    x++;
    *y_p += 4;
    a_p = malloc(sizeof(int));
    *a_p = 1;
    *b_pp = y_p;
    printf("x = %d, *y_p = %d, *a_p = %d, **b_pp = %d\n",
            x, *y_p, *a_p, **b_pp);

    return (*y_p + *a_p);
} /* star */

Output:

x = 4, *y_p = 9, *a_p = 1, **b_pp = 9
z = 10, x = 3, y = 9, *a_p = 3, *b_p = 9

See Figure 1.
6. Sally’s MPI library has implemented MPI_Send so that it always uses buffered mode. Find the output of the following MPI program on Sally’s system if it’s run with

(a) One process [1]
(b) Four processes [4]

Continue on the following page if you need more space.

```c
#include <stdio.h>
#include <mpi.h>

void Assign_vals(int my_rank, int *d_p, int* s_p, int* v_p) {
    switch (my_rank) {
    case 0:
        *d_p = 2; *s_p = 1; *v_p = 5;
        break;
    case 1:
        *d_p = 0; *s_p = 3; *v_p = 4;
        break;
    case 2:
        *d_p = 3; *s_p = 0; *v_p = 3;
        break;
    case 3:
        *d_p = 1; *s_p = 2; *v_p = 5;
        break;
    }
} /* Assign_vals */

int main(int argc, char* argv[]) {
    int my_rank, p, d, s, v, x = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    Assign_vals(my_rank, &d, &s, &v);
    printf("Proc %d > d = %d, s = %d, v = %d, x = %d\n", my_rank, d, s, v, x);
    if (p > 1) {
        MPI_Send(&v, 1, MPI_INT, d, 0, MPI_COMM_WORLD);
        MPI_Recv(&x, 1, MPI_INT, s, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    printf("Proc %d > d = %d, s = %d, v = %d, x = %d\n", my_rank, d, s, v, x);
}
} /* main */
```
Continue with Problem 6

(a) 

<table>
<thead>
<tr>
<th>my_rank</th>
<th>p</th>
<th>d</th>
<th>s</th>
<th>v</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Output:

Proc 0 > d = 2, s = 1, v = 5, x = 0
Proc 0 > d = 2, s = 1, v = 5, x = 0

(b) 

<table>
<thead>
<tr>
<th>my_rank</th>
<th>p</th>
<th>d</th>
<th>s</th>
<th>v</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Proc 0 > d = 2, s = 1, v = 5, x = 0
Proc 1 > d = 0, s = 3, v = 4, x = 0
Proc 2 > d = 3, s = 0, v = 3, x = 0
Proc 3 > d = 1, s = 2, v = 5, x = 0
Proc 0 > d = 2, s = 1, v = 5, x = 4
Proc 1 > d = 0, s = 3, v = 4, x = 5
Proc 2 > d = 3, s = 0, v = 3, x = 5
Proc 3 > d = 1, s = 2, v = 5, x = 3

(The exact order of the output is random.)
7. Bob is using the struct defined on page 9 in his linked list program. The program contains a function `Delete_all` (see below). Its purpose is to delete all occurrences of a value from the list. It proceeds by searching the list for the user-specified value. Each time it finds the user-specified value, it deletes it from the list. If there are no occurrences, it simply traverses the list. The function performs as advertised. However, Prof P has told Bob that the function has a memory leak. Modify the function so that it correctly deletes all occurrences of `value` and correctly frees deleted nodes. [4]

```c
struct list_node_s* Delete_all(struct list_node_s* head_p, int value) {
    struct list_node_s* curr_p = head_p;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p; /* NEW */

    while (curr_p != NULL) {
        if (curr_p->data == value) {
            if (pred_p == NULL)
                head_p = curr_p->next_p;
            else
                pred_p->next_p = curr_p->next_p;
            temp_p = curr_p; /* NEW */
            curr_p = curr_p->next_p;
            free(temp_p); /* NEW */
        } else {
            pred_p = curr_p;
            curr_p = curr_p->next_p;
        }
    } /* while */

    return head_p;
} /* Delete_all */
```
8. A linked list program uses the struct definition on page 9. Write a C function Count that is passed the head_p pointer and two ints, start and value. The function returns the number of times value occurs in the list after the first occurrence of start. For example, if head_p references the list shown on page 9, start = 3, and value = 3, then the function should return 1. Using the same list, if start = 1 and value = 2, then the function should return 2, and if start = 7 and value = 1, the function should return 0. You can assume that start ≠ value.

Here’s a prototype for the function:

    int Count(struct list_node_s* head_p, int start, int value);

[4]

    int Count(struct list_node_s* head_p, int start, int value) {
        int count = 0;
        struct list_node_s* curr_p = head_p;

        while (curr_p != NULL) {
            if (curr_p->data == start) {
                curr_p = curr_p->next_p;
                break;
            } else {
                curr_p = curr_p->next_p;
            }

            while (curr_p != NULL) {
                if (curr_p->data == value) {
                    count++;
                    curr_p = curr_p->next_p;
                }
            }
        }
        return count;
    }
9. Write a function `Delete` that is passed an array of ints `arr`, a value `val`, and a pointer to an int `n_p`. The function should remove all occurrences of `val` from the array. The pointer `n_p` is an input/output argument: when the function is called, it points to the number of elements in the array. When the function returns, it points to the reduced number of elements in the array. For example, if `arr` stores the elements

```
4 3 1 1 2 4 4 2
```

and `val = 4`, then when the function is called `n_p` should point to the value 8. When the function returns, `arr` should store the elements

```
3 1 1 2 2
```

and `n_p` should point to the value 5. Here’s a prototype:

```c
void Delete(int arr[], int val, int* n_p);
```

```c
void Delete(int arr[], int val, int* n_p) {
    int i, j = 0, count = *n_p;
    int temp[*n_p];

    for (i = 0; i < *n_p; i++)
        if (arr[i] != val) {
            temp[j] = arr[i];
            j++;
        } else {
            count--;
        }

    for (i = 0; i < count; i++)
        arr[i] = temp[i];
    *n_p = count;
} /* Delete */
```
Problems 7 and 8 use the following data structure.

```c
struct list_node_s {
    int     data;
    struct list_node_s* next_p;
};
```

The programs that use this data structure store unsorted lists of ints with repeated elements allowed. If the list is empty the `head_p` pointer is `NULL`.

Here’s an example linked list that’s been built with this structure.

```plaintext
+--+ +---+--+ +---+--+ +---+--+ +---+--+ +---+--+ +---+--+
head_p| -+-->| 3 | -+-->| 1 | -+-->| 2 | -+-->| 3 | -+-->| 2 | -+-->| 4 |
| +--+ +---+--+ +---+--+ +---+--+ +---+--+ +---+--+ +---+--+
```