Ex 3.1 pg. 140

If strlen(greeting) is used instead of strlen(greeting) + 1, then the output of the program becomes:

```
Greetings from process 0 of 16!
Greetings from process 1 of 16!kup/fib_l
Greetings from process 2 of 16!kup/fib_l
Greetings from process 3 of 16!kup/fib_l
Greetings from process 4 of 16!kup/fib_l
Greetings from process 5 of 16!kup/fib_l
Greetings from process 6 of 16!kup/fib_l
Greetings from process 7 of 16!kup/fib_l
Greetings from process 8 of 16!kup/fib_l
Greetings from process 9 of 16!kup/fib_l
Greetings from process 10 of 16!up/fib_l
Greetings from process 11 of 16!up/fib_l
Greetings from process 12 of 16!up/fib_l
Greetings from process 13 of 16!up/fib_l
Greetings from process 14 of 16!up/fib_l
Greetings from process 15 of 16!up/fib_l
```

Because strlen(greeting) does not include the null character "\0" and therefore this is not sent. If MAX_STRING is used instead of strlen(greeting) then the output becomes:

```
Greetings from process 0 of 16!
Greetings from process 1 of 16!
Greetings from process 2 of 16!
Greetings from process 3 of 16!
Greetings from process 4 of 16!
Greetings from process 5 of 16!
Greetings from process 6 of 16!
Greetings from process 7 of 16!
Greetings from process 8 of 16!
Greetings from process 9 of 16!
Greetings from process 10 of 16!
Greetings from process 11 of 16!
Greetings from process 12 of 16!
Greetings from process 13 of 16!
Greetings from process 14 of 16!
Greetings from process 15 of 16!
```

This is because now, the entire size of the greeting message can be sent.

Ex 3.2 pg. 140

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

double Trap(double left_endpt, double right_endpt, int trap_count, double base_len);

int main(void) {
        int my_rank, comm_sz, n = 1024, local_n;
        double a = 0.0, b = 3.0,h, local_a, local_b;
        double local_int, total_int;
        int source;
        MPI_Init(NULL, NULL);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        h = (b-a)/n; /* h is the same for all processes */
        local_n = n/comm_sz; /* So is the number of trapezoids */
        if (my_rank < (n%comm_sz)) {
                local_n++;
                local_a = a + my_rank*local_n*h;
                local_b = local_a + local_n*h;
        } else {
                local_a = a + my_rank*local_n*h + (n%comm_sz)*h;
                local_b = local_a + local_n*h;
        }
        local_int = Trap(local_a, local_b, local_n, h);
        if (my_rank != 0) {
                MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        } else {
                total_int = local_int;
                for (source = 1; source < comm_sz; source++) {
                        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        total_int += local_int;
                }
        }
        if (my_rank == 0) {
                printf("With n = %d trapezoids, our estimate\n", n);
                printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
        }
        MPI_Finalize();
        return 0;
} /* main */

double Trap(
        double left_endpt /* in */,
        double right_endpt /* in */,
        int trap_count /* in */,
        double base_len /* in */) {
  double estimate, x;
  int i;
  estimate = (left_endpt*left_endpt + right_endpt*right_endpt)/2.0;
  for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i*base_len;
        estimate += estimate*estimate;
  }
  estimate = estimate*base_len;
  return estimate;
} /* Trap */
```

Ex 3.3 pg. 140

In the trapezoidal rule program, the variables my_rank, local_n, local_a, local_b, local_int, total_int are local variables since their values are can be different for each process. On the other hand, the variables comm_sz, n, a, b and h are all global variables as they have constant values across all processes.

Ex 3.4 pg. 140

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

int main(void) {
        int my_rank, comm_sz, data, i;
        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        if (my_rank != 0) {
                data = my_rank;
                MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        } else {
                printf("Proc %d of %d > Does anyone have a toothpick?\n", my_rank, comm_sz);
                for (i = 1; i < comm_sz; i++) {
                        MPI_Recv(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        printf("Proc %d of %d > Does anyone have a toothpick?\n", data, comm_sz);
                }

        }

        MPI_Finalize();
        return 0;
} /* main */
```

Programming Exercise 3.4, p. 148

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

int main(int argc, char *argv[]) {
        MPI_Request     sendreq;
        MPI_Request     recvreq;
        MPI_Status      status;
        double          start, end;
        int             numprocs;
        int             my_rank;
        int             i;
        int             proc_diff;

        MPI_Init(&argc, &argv);
        while (!(numprocs && (!(numprocs&(numprocs-1))))) {
                numprocs++;
        }
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        start = MPI_Wtime();
        i = 2;
        int sum = rand();
        int dst;
        int temp;
        while(i <= numprocs) {
                if((my_rank%i) < i/2){
                        dst = my_rank + i/2;
                }
                else {
                        dst = my_rank - i/2;
                }
                temp = sum;
                MPI_Send(&temp, 1, MPI_INT, dst, 0, MPI_COMM_WORLD);
                if ((my_rank%i) < i/2) {
                        MPI_Recv(&temp, 1, MPI_INT, (my_rank + i/2), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        sum += temp;
                }
                else {
                        MPI_Recv(&temp, 1, MPI_INT, (my_rank - i/2), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        sum += temp;
                }
                i *= 2;
        }
        MPI_Barrier(MPI_COMM_WORLD);
        printf("Global sum from process : %d is %d\n", my_rank, sum);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
}
```

Output for test cases:

```
Global sum from process : 8 is -1196140944
Global sum from process : 12 is -1196140944
Global sum from process : 4 is -1196140944
Global sum from process : 7 is -1196140944
Global sum from process : 10 is -1196140944
Global sum from process : 14 is -1196140944
Global sum from process : 15 is -1196140944
Global sum from process : 0 is -1196140944
Global sum from process : 6 is -1196140944
Global sum from process : 5 is -1196140944
Global sum from process : 1 is -1196140944
Global sum from process : 2 is -1196140944
Global sum from process : 3 is -1196140944
Global sum from process : 9 is -1196140944
Global sum from process : 11 is -1196140944
Global sum from process : 13 is -1196140944
```

```
Global sum from process : 0 is -1372777060
Global sum from process : 1 is -1372777060
Global sum from process : 2 is -1372777060
Global sum from process : 3 is -1372777060
```

Programming Exercise 3.7, p. 148

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <time.h>

int main(int argc, char *argv[]) {
        clock_t start, end;
        double start1, end1;
        int numprocs = 2;
        int my_rank;
        int n, i, temp;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        n = atol(argv[1]);
        start = clock();
        start1 = MPI_Wtime();
        for (i = 0; i < n; i++) {
                if (my_rank == 0) {
                        MPI_Send(&temp, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
                }
                else {
                        MPI_Recv(&temp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
                if (my_rank != 0){
                        MPI_Send(&temp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                }
                else {
                        MPI_Recv(&temp, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
        }
        end = clock();
        end1 = MPI_Wtime();
        if (my_rank == 0) {
                printf("Time using clock = %f seconds\n", ((double) (end - start))/CLOCKS_PER_SEC);
                printf("Time using MP_Wtime() = %f seconds\n", end1 - start1);
        }

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
}
```

Output for various test runs:

Block size	Clock time (s)	MPI_Wtime (s)
1000	0.000000	0.002141
10000	0.010000	0.015582
100000	0.150000	0.149632
1000000	1.100000	1.102033

Times with clock time seem to be slightly smaller than or approximately equal to times with MPI_Wtime().