

## Homework #2 Solutions

### Ex 1: The stream computational model

```
#include <stdio.h>
#include <sys/archlib.h>
#include <ilib.h>

#define NUM_CPUS 16
#define PROBLEM_SIZE 2048
#define ARRAYSIZE (PROBLEM_SIZE+2)
#define ITERS 12

#define DATA int

ILIB_RAW_SEND_PORT(send_to_left, 0);
ILIB_RAW_SEND_PORT(send_to_right, 1);

ILIB_RAW_RECEIVE_PORT(recv_from_left, 0);
ILIB_RAW_RECEIVE_PORT(recv_from_right, 1);

#define SEND_LEFT 0
#define SEND_RIGHT 1
#define RECV_LEFT 2
#define RECV_RIGHT 3

void open_channels(int rank, int size) {

    if(rank == 0) {
        ilib_rawchan_connect(ILIB_GROUP_SIBLINGS,
        0,
        SEND_RIGHT,
        1,
        RECV_LEFT);

        ilib_rawchan_open_sender(SEND_RIGHT,
        send_to_right);
        ilib_rawchan_open_receiver(RECV_RIGHT,
        recv_from_right);
    }
}
```

```

    }
    else if (rank == size-1) {
        ilib_rawchan_connect(ILIB_GROUP_SIBLINGS,
rank,
SEND_LEFT,
rank-1,
RECV_RIGHT);

        ilib_rawchan_open_receiver(RECV_LEFT,
            recv_from_left);
        ilib_rawchan_open_sender(SEND_LEFT,
            send_to_left);
    }
    else {
        ilib_rawchan_connect(ILIB_GROUP_SIBLINGS,
rank,
SEND_LEFT,
rank-1,
RECV_RIGHT);
        ilib_rawchan_connect(ILIB_GROUP_SIBLINGS,
rank,
SEND_RIGHT,
rank+1,
RECV_LEFT);

        ilib_rawchan_open_receiver(RECV_LEFT,
            recv_from_left);
        ilib_rawchan_open_sender(SEND_RIGHT,
            send_to_right);

        ilib_rawchan_open_receiver(RECV_RIGHT,
            recv_from_right);
        ilib_rawchan_open_sender(SEND_LEFT,
            send_to_left);
    }
}

```

```

void print_array(DATA *array)
{
    int i;
    for (i = 1; i <= PROBLEM_SIZE; ++i)
        printf("%d ", array[i]);
    printf("\n");
}

```

```

void update_array(DATA *oldarr, DATA *newarr,
    int rank, int group_size, int local_writes)
{
    int from, to, i;
    int last_local_write = local_writes;

    from = 1;
    to = local_writes;

    for (i = from; i <= to; ++i)
        newarr[i] = (oldarr[i-1] + oldarr[i+1])/2;

    if(rank == 0) {
        ilib_rawchan_send(send_to_right, newarr[last_local_write]);
        newarr[last_local_write+1] = ilib_rawchan_receive(recv_from_right);
    }
    else if(rank == group_size -1) {
        ilib_rawchan_send(send_to_left, newarr[1]);
        newarr[0] = ilib_rawchan_receive(recv_from_left);
    }
    else {
        ilib_rawchan_send(send_to_left, newarr[1]);
        ilib_rawchan_send(send_to_right, newarr[last_local_write]);
        newarr[0] = ilib_rawchan_receive(recv_from_left);
        newarr[last_local_write+1] = ilib_rawchan_receive(recv_from_right);
    }
}

#define SWAP(a,b,t) (((t) = (a)), ((a) = (b)), ((b) = (t)))

int main(void)
{
    int i, t;
    DATA *oldarr, *newarr, *tmp;
    long long start, stop;
    int rank, size;
    int local_array_size;
    DATA master_array[ARRAYSIZE];

    ilib_init();

    if (ilib_proc_go_parallel(NUM_CPUS, NULL) != ILIB_SUCCESS)
        ilib_die("Failed to go parallel.");

    rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);
    size = ilib_group_size(ILIB_GROUP_SIBLINGS);

```

```

open_channels(rank,size);

local_array_size = (PROBLEM_SIZE/size)+2;

oldarr = (DATA *) malloc(local_array_size * sizeof(DATA));
newarr = (DATA *) malloc(local_array_size * sizeof(DATA));

if( rank == 0)
    oldarr[0] = newarr[0] = 32768;
else
    oldarr[0] = newarr[0] = 0;

for (i = 1; i < local_array_size; ++i)
    oldarr[i] = newarr[i] = 0;

ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

start = get_cycle_count();
for (t = 0; t < ITERS; ++t) {
    update_array(oldarr, newarr, rank, size, (PROBLEM_SIZE/size));
    SWAP(oldarr, newarr, tmp);
}
ilib_msg_barrier(ILIB_GROUP_SIBLINGS);
stop = get_cycle_count();

if(rank == 0) {
    ilibStatus status;
    memcpy(master_array+1, oldarr+1, (PROBLEM_SIZE/size)*sizeof(DATA));

    for(i = 1; i < size; i++) {
        ilib_msg_receive(ILIB_GROUP_SIBLINGS,
            i, 0,
            &master_array[(PROBLEM_SIZE/size)*i+1],
            (PROBLEM_SIZE/size)*sizeof(DATA), &status);
    }
}
else {
    ilib_msg_send(ILIB_GROUP_SIBLINGS,
0, 0, &oldarr[1], (PROBLEM_SIZE/size)*sizeof(DATA));
}

if(rank == 0) {

```

```

    print_array(master_array);
    printf("Program took %lld cycles\n", stop-start);
}

return 0;
}

```

## Ex 2: The shared-memory work-queue computational model

(1) The best tour (cost 14 is): 0 1 4 3 2.

Here is one possible implementation:

```

#include <stdio.h>
#include <ilib.h>

#define NUM_CITIES      5
#define START_CITY     0          /* city to start trip from */
#define NUM_CPUS 4

/* Infinity = 1000 since 1000 is more than a trip could possibly be. */
#define INF            1000

/* it's private and distributed at boot time with the code. */
int cost[NUM_CITIES][NUM_CITIES] = {
    { INF, 2,  3,  1,  INF },
    { INF, INF, 5,  7,  4  },
    { 3,  1,  INF, 6,  1  },
    { 8,  2,  4,  INF, 3  },
    { 7,  INF, 6,  1,  INF }
};

typedef struct {
    int path[NUM_CITIES];
    int unvisited[NUM_CITIES];
    int cost;
    int length;
} partial_tour;

struct queue_entry {
    partial_tour tour;
    struct queue_entry* next;
};

typedef struct queue_entry queue_entry;

```

```

typedef struct {
    ilibMutex mutex;
    partial_tour tour;
} locked_tour;

typedef struct {
    ilibMutex mutex;
    int value;
} locked_int;

typedef struct {
    ilibMutex mutex;
    queue_entry* next;
    int length;
} work_queue;

void locked_int_increment (locked_int* i) {
    ilib_mutex_lock(&i->mutex);
    i->value++;
    ilib_mutex_unlock(&i->mutex);
}

void locked_int_decrement (locked_int* i) {
    ilib_mutex_lock(&i->mutex);
    i->value--;
    ilib_mutex_unlock(&i->mutex);
}

void queue_init(work_queue* q) {
    q->next = NULL;
    q->length = 0;
}

void queue_add_work(work_queue* q, queue_entry* work) {
    ilib_mutex_lock(&q->mutex);
    work->next = q->next;
    q->next = work;
    q->length++;
    ilib_mutex_unlock(&q->mutex);
}

queue_entry* queue_get_work(work_queue* q) {
    queue_entry* result = NULL;
    ilib_mutex_lock(&q->mutex);

```

```

    if(q->length > 0) {
        result = q->next;
        q->next = result->next;
        q->length--;
    }
    ilib_mutex_unlock(&q->mutex);
    return result;
}

void print_tour(locked_tour* best) {
    ilib_mutex_lock(&best->mutex);
    printf("The best tour (cost %d):", best->tour.cost);

    for(int i = 0; i < NUM_CITIES; i++) {
        printf(" %d", best->tour.path[i]);
    }
    printf("\n");
    ilib_mutex_unlock(&best->mutex);
}

void do_work(queue_entry* work, work_queue* q, locked_tour* best) {
    int current_length = work->tour.length;
    int current_best = best->tour.cost;
    int current_cost = work->tour.cost;
    int current_city = work->tour.path[current_length-1];
    int current_unvisited = NUM_CITIES - current_length;

    if(current_length == NUM_CITIES) {

        int new_cost = (current_cost + cost[current_city][0]);

        if( new_cost < current_best) {
            ilib_mutex_lock(&best->mutex);

            if(new_cost < best->tour.cost) { // value may have changed since locked
                best->tour.cost = new_cost;
                memcpy(best->tour.path,
                    work->tour.path,
                    current_length*sizeof(int));
            }
            ilib_mutex_unlock(&best->mutex);
        }
    }
    else {
        for(int i = 0; i < current_unvisited; i++) {
            int new_cost = (current_cost +

```

```

        cost[current_city][work->tour.unvisited[i]];
        if( new_cost < current_best) {
queue_entry* new_work =
    (queue_entry*) malloc_shared(sizeof(queue_entry));

new_work->next = NULL;
new_work->tour.length = current_length + 1;
memcpy(new_work->tour.path,
work->tour.path,
current_length*sizeof(int));
new_work->tour.path[current_length] = work->tour.unvisited[i];
new_work->tour.cost = new_cost;
for(int j = 0; j < i; j++) {
    new_work->tour.unvisited[j] = work->tour.unvisited[j];
}
for(int j = i+1; j < current_unvisited; j++) {
    new_work->tour.unvisited[j-1] = work->tour.unvisited[j];
}

queue_add_work(q, new_work);
    }
}

}
}

int main(void) {
    work_queue* queue;
    locked_int* num_workers;
    locked_tour* best_tour;
    queue_entry* start;
    ilibStatus status;

    int iters = 0;

    ilib_init();

    if (ilib_proc_go_parallel(NUM_CPUS, NULL) != ILIB_SUCCESS)
        ilib_die("Failed to go parallel.");

    int rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);
    int size = ilib_group_size(ILIB_GROUP_SIBLINGS);

    if(rank == 0) {
        queue      = (work_queue*) malloc_shared(sizeof(work_queue));
        num_workers = (locked_int*) malloc_shared(sizeof(locked_int));
    }
}

```

```

best_tour    = (locked_tour*) malloc_shared(sizeof(locked_tour));
start       = (queue_entry*) malloc_shared(sizeof(queue_entry));

queue_init(queue);
num_workers->value = 0;
best_tour->tour.cost = INF;
best_tour->tour.path[0] = -1;
best_tour->tour.path[1] = -1;
best_tour->tour.path[2] = -1;
best_tour->tour.path[3] = -1;
best_tour->tour.path[4] = -1;

start->next = NULL;
start->tour.length = 1;
start->tour.cost = 0;
start->tour.path[0] = 0;
for(int i = 1; i < NUM_CITIES; i++) {
    start->tour.path[i] = -1;
    start->tour.unvisited[i-1] = i;
}

queue_add_work(queue, start);
}
ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, 0, &queue,
    sizeof(queue), &status);
ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, 0, &num_workers,
    sizeof(num_workers), &status);
ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, 0, &best_tour,
    sizeof(best_tour), &status);
ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, 0, &start,
    sizeof(start), &status);

while(queue->length != 0 || num_workers->value != 0) {

    queue_entry* work = NULL;

    work = queue_get_work(queue);

    if(work != NULL) {
        locked_int_increment(num_workers);

        do_work(work, queue, best_tour);
        locked_int_decrement(num_workers);
    }

    iters++;
}

```

```
}

ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

if(rank == 0)
    print_tour(best_tour);

ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

ilib_finish();
return 0;
}
```

(2) In order to alleviate bottlenecks in the code, one can do a few things.

One option (implemented here) would be to only lock the bound variable only when a write is to be performed. This procedure involves doing a shared read to decide whether to perform a write. If a write is desired, then lock the variable. Of course, a check needs to be implemented to make sure no lock occurred between the shared read and the locking read. Another solution would be to expand each node more than just one level before actually comparing the current cost of the node to the bound. This would reduce the total number of bound variable accesses, but would also increase the amount of wasteful searching. The optimal number of levels to expand a node would also need to be determined.

A similar bottleneck is caused by the lock on the work queue. On performance enhancement would be to create an entire list of new work entries for the queue as we expand a node, and only lock the queue once when we add the entire list of new work to be done. An additional optimization might include removing multiple items of work from the queue each time we lock it.