High Performance Computing: Sheet 3

Nils Döring Michael Mardaus Julian F. Rost
Sebastian Müller

20. November 2013

Question 1

a)

Assign the next $ceil(n/comm\_sz)$ vector parts to each of the first n-1 cpus and the rest will be assigned to cpu n.

b)

For cpu i, assign it all vector parts numbered x, so that $x\%comm\_sz = i$.

c)

For cpu i, assign it all vector parts numbered x, so that $floor((x/b)\%comm\_sz) = i$.

Question 2

Yes it does. In fact, this code will lock every single time. Deadlocks are inevitable. Send will only terminate once the communication partner recieves, and since both try to send before recieving, they will not terminate.

Solutions: Reverse the order of MPI_Recv and MPI_Send commands in exactly one of the communication partners.

Alternatively, use a different function to communicate, as outlined below: Use MPI_Sendrecv instead, which can recieve and send simultaneously. Use MPI_Irecv and MPI_Isend, which will recieve or send asynchronously, respectively, and therefore terminate the subroutine instantly. Use MPI_Bcast to flood the network. Yay.

Question 3

The code snippet are stored in Julian Rosts Sauce-Account.

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

```
int main(int argc, char** argv) {

    MPI_Init(&argc, &argv);

    int size, rank, i, dest, source, sum, temp_val;
    MPI_Status* status;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    dest = (rank + 1) % size;
    source = (rank - 1) % size;
    sum = temp_val = rank;
    for(i = 1; i < size; i++){
        MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest, 0, source, 0, MPI_COMM_WORLD, status);
        sum += temp_val ;
    }

    printf("Process %d: %d\n", rank, sum);

    MPI_Finalize();

    return EXIT_SUCCESS;

}
```

AllReduce is much more performant; its runtime is logarithmic, while the ring-pass structure's is linear.

Question 4

The code snippet are stored in Julian Rosts Sauce-Account.

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

#define SIZE 24

int main(int argc, char* argv[]){

    MPI_Init(&argc, &argv);
    int myid, numprocs;

    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    if(numprocs > SIZE){ MPI_Finalize(); return 0; }
    /* init matrix A and vector x */
    int A[SIZE][SIZE];
    int x[SIZE];
    int i,j;
    if(myid == 0){
        for(i=0;i<SIZE;++i){
            for(j=0;j<SIZE;++j){
```

```
                A[i][j] = (i*SIZE+j) % 3;
            }
            x[i] = (SIZE-i) % 3;
        }
    }

    /************************/
    /* YOUR TASK STARTS HERE */
    /************************/
        int result[SIZE];
    //distribute data
        if (myid == 0) {
            MPI_Bcast(&A, SIZE * SIZE, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(&x, SIZE, MPI_INT, 0, MPI_COMM_WORLD);
        } else {
            MPI_Status* status = 0;
            MPI_Recv(&A, SIZE * SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, status);
            MPI_Recv(&x, SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, status);
        }

    //calculate local values
        int breadth = SIZE / numprocs;
        for (i = 0; i < SIZE; i++) {
            result[i] = 0;
            for (j = myid * breadth; j < (myid + 1) * breadth; j++) {
                result[i] += A[i][j] * x[j];
            }
        }

    //collect data
        int new_result[SIZE];
        if (myid == 0) {
            MPI_Reduce(&result[i], &new_result[i], 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        }


    /**********************/
    /* YOUR TASK ENDS HERE */
    /**********************/

    /* print result vector */
    if(myid == 0){
        for(i=0;i<SIZE;++i){
            printf("%d ", new_result[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```