
HPC Übungen

Release 1.0

Sarah Blume, Sebastian Rieger

01.05.2015

Inhaltsverzeichnis

1 Übung 1	2
1.1 Aufgabe 1	2
1.2 Aufgabe 2	3
1.3 Aufgabe 3	4
2 Übung 2	7
2.1 Aufgabe 1	7
2.2 Aufgabe 2	8
3 Übung 3	17
3.1 Aufgabe 1	17
3.2 Aufgabe 2	18
4 Übung 4	19
4.1 Aufgabe 1	19
4.2 Aufgabe 2	19
a) - c)	19
d)	20
e)	20
4.3 Aufgabe 3	21
a)	21
b)	21
c)	22
4.4 Aufgabe 4	22
a)	22
b)	23
c)	23
5 Übung 5	24
5.1 Aufgabe 1	24
a)	24
b)	24
c)	24
5.2 Aufgabe 2	24
a)	24
b)	25
c)	25
d)	26
e+f)	27
5.3 Aufgabe 3	30

a)	30
b)	31
c)	31
6 Übung 6		31
6.1 a)	31
6.2 b)	31
6.3 c)	32
6.4 d)	33
6.5 e)	33
6.6 f)	34
6.7 g)	34
7 Übung 7		34
7.1 Aufgabe 1	34

1 Übung 1

1.1 Aufgabe 1

1. cd hello-world
make run

2.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

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

    #pragma omp parallel num_threads(4)
    {
        int num_threads = omp_get_num_threads();
        int this_num = omp_get_thread_num();
        printf("Hello World %d von %d\n", this_num, num_threads);
    }
    return 0;
}
```

3. Die Ausgabe ist nicht konstant, weil die Threads bei jeder Ausgabe unterschiedlich schnell sind.

4.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

int main(int argc, char **argv) {
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Hola Mundo from thread %d of %d\n",
                omp_get_thread_num(), omp_get_num_threads());
            printf("Hej varlden from thread %d of %d\n",
                omp_get_thread_num(), omp_get_num_threads());
        }
    }
}
```

```

#pragma omp section
{
    printf("Bonjour tout from thread %d of %d\n",
           omp_get_thread_num(), omp_get_num_threads());
}

#pragma omp section
{
    printf("Hallo Welt from thread %d of %d\n",
           omp_get_thread_num(), omp_get_num_threads());
}

#pragma omp section
{
    printf("Hello World from thread %d of %d\n",
           omp_get_thread_num(), omp_get_num_threads());
}

return 0;
}

```

1.2 Aufgabe 2

- Der Fehler in error1 tritt nur auf, wenn mehr als 2 Threads verwendet werden.

Von 1.36 bis 1.55 wird das Programm in zwei Sections aufgeteilt welche von 2 Threads abgearbeitet werden. Wegen dem “nowait” Befehl laufen alle anderen Threads gegen die Barrier in 1.58 und warten auf die Threads, welche den Sectionblock abarbeiten.

Diese Threads laufen aber gegen die Barrier in 1.86 und warten dort auf die anderen Threads, welche diesen Codeteil nicht ausführen.

Für die Behebung gibt es also 2 Möglichkeiten.

- (a) OMP_NUM_THREADS=2
 - (b) 1.86 löschen
- Der Fehler tritt durch ein Deadlock auf, dies geschieht da die beiden Sections ihre Locks nicht zu Beginn setzen.

In 1.45 locka der erste Thread locka und zeitgleich in 1.59 wird durch den anderen Thread lockb gelocked. Laufen nun beide Threads weiter wartet der erste Thread in 1.48 auf das unlock von lockb und der andere Thread in 1.62 auf das unlock von locka.

Der Fehler kann wieder durch mindesten 3 Arten gelöst werden.

- (a) Der Bereich wird nicht parallelisiert sondern sequentiell hintereinander ausgeführt.
- (b) Nur einen CPU Core verwenden, weil dann die Sections auch sequentiell und nicht parallel ausgeführt werden.
- (c) Jede Section locked zu beginn all ihre locks in der gleichen Reihenfolge.

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[])
{
    int nthreads, tid, i;
    float a[N], b[N];

```

```

omp_lock_t locka, lockb;

/* Initialize the locks */
omp_init_lock(&locka);
omp_init_lock(&lockb);

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
{
    /* Obtain thread number and number of threads */
    tid = omp_get_thread_num();
    #pragma omp master
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);
    #pragma omp barrier

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d initializing a[]\n", tid);
            omp_set_lock(&locka);
            omp_set_lock(&lockb);
            for (i=0; i<N; i++)
                a[i] = i * DELTA;
            printf("Thread %d adding a[] to b[]\n", tid);
            for (i=0; i<N; i++)
                b[i] += a[i];
            omp_unset_lock(&lockb);
            omp_unset_lock(&locka);
        }

        #pragma omp section
        {
            printf("Thread %d initializing b[]\n", tid);
            omp_set_lock(&locka);
            omp_set_lock(&lockb);
            for (i=0; i<N; i++)
                b[i] = i * PI;
            printf("Thread %d adding b[] to a[]\n", tid);
            for (i=0; i<N; i++)
                a[i] += b[i];
            omp_unset_lock(&locka);
            omp_unset_lock(&lockb);
        }
    } /* end of sections */
} /* end of parallel region */
}

```

1.3 Aufgabe 3

1. make run
2. #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <omp.h>

```

#define TRYS 5000000

static int throw() {
    double x, y;
    x = (double)rand() / (double)RAND_MAX;
    y = (double)rand() / (double)RAND_MAX;
    if ((x*x + y*y) <= 1.0) return 1;

    return 0;
}

int main(int argc, char **argv) {
    int globalCount = 0, globalSamples=TRYS, i;

#pragma omp parallel for private(i) shared(globalCount)
for(i = 0; i < globalSamples; ++i) {
    int add = throw();
    if (add != 0){
        #pragma omp atomic
        globalCount += add;
    }
}

double pi = 4.0 * (double)globalCount / (double)(globalSamples);

printf("pi is %.9lf\n", pi);

return 0;
}

3. #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

#define TRYS 5000000

static int throw() {
    double x, y;
    x = (double)rand() / (double)RAND_MAX;
    y = (double)rand() / (double)RAND_MAX;
    if ((x*x + y*y) <= 1.0) return 1;

    return 0;
}

int main(int argc, char **argv) {
    int globalCount = 0, globalSamples=TRYS, i;

#pragma omp parallel for reduction(+:globalCount)
for(i = 0; i < globalSamples; ++i) {
    int add = throw();
    if (add != 0){
        globalCount += add;
    }
}

double pi = 4.0 * (double)globalCount / (double)(globalSamples);

printf("pi is %.9lf\n", pi);

return 0;
}

```

```

}

4. #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

#define TRY 5000000

static int throw() {
    double x, y;
    x = (double)rand() / (double)RAND_MAX;
    y = (double)rand() / (double)RAND_MAX;
    if ((x*x + y*y) <= 1.0) return 1;

    return 0;
}

int main(int argc, char **argv) {
    int globalCount = 0, globalSamples=TRY, i;

#pragma omp parallel reduction(+:globalCount)
{
    #pragma omp for
    for(i = 0; i < globalSamples; ++i) {
        int add = throw();
        if (add != 0){
            globalCount += add;
        }
    }
}

printf("thread %d: i = %d\n", omp_get_thread_num(), globalCount);
}

double pi = 4.0 * (double)globalCount / (double)(globalSamples);

printf("pi is %.9lf\n", pi);

return 0;
}

5. #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

#define TRY 5000000

static int throw() {
    double x, y;
    x = (double)rand() / (double)RAND_MAX;
    y = (double)rand() / (double)RAND_MAX;
    if ((x*x + y*y) <= 1.0) return 1;

    return 0;
}

int main(int argc, char **argv) {
    int globalCount = 0, globalSamples=TRY, i;

#pragma omp parallel reduction(+:globalCount) num_threads(6)
{

```

```

#pragma omp for
for(i = 0; i < globalSamples; ++i) {
    int add = throw();
    if (add != 0){
        globalCount += add;
    }
}

printf("thread %d: i = %d\n", omp_get_thread_num(), globalCount);
}

double pi = 4.0 * (double)globalCount / (double)(globalSamples);

printf("pi is %.9lf\n", pi);

return 0;
}

```

Durch das `num_threads(6)` wird unterbunden, dass der Benutzer die Threadanzahl verändern kann. Er könnte dies ohne diese Angabe durch setzen von `OMP_NUM_THREADS` tun.

2 Übung 2

2.1 Aufgabe 1

- Bei der Ausführung kann beobachtet werden, dass ein Philosoph immer mehrmals hintereinander denkt und isst.

Nach einer Weile erfolgt ein Wechsel und ein anderer Philosoph isst bzw. denkt. Diese Beobachtung wiederholt sich endlos.

- Unsere Philosophen sind höflich, nachdem sie gegessen haben, denken sie erst einmal wieder eine Weile nach, dies verhindert Deadlocks, da ihre Kolegen, welche essen wollen in der Zwischenzeit sich die Gabel nehmen/locken können.

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

// number of philosophers
#define N 5
// left neighbour
#define LEFT (id)
// right neighbour
#define RIGHT ((id + 1) % num_threads)

#define TRUE 1
#define FALSE 0

// Global variables
int num_threads;
omp_lock_t forks[N];

void think(int philosopher) {
    printf("%d is thinking.\n", philosopher);
}
void eat(int philosopher) {
    printf("%d is eating.\n", philosopher);
}

```

```

void philosopher(int id) {
    while(TRUE) {
        think(id);
        sleep(1);

        omp_set_lock(&forks[LEFT]);
        omp_set_lock(&forks[RIGHT]);
        eat(id);
        omp_unset_lock(&forks[LEFT]);
        omp_unset_lock(&forks[RIGHT]);
    }
}

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

    for (i = 0; i < N; i++){
        omp_init_lock(&forks[i]);
    }

    omp_set_num_threads(N);
#pragma omp parallel private(id) shared(num_threads, forks)
{
    id = omp_get_thread_num();
    num_threads = omp_get_num_threads();

    philosopher(id);
}

    for (i = 0; i < N; i++){
        omp_destroy_lock(&forks[i]);
    }
    return 0;
}

```

2.2 Aufgabe 2

1. Ohne weitere Ändungerung am Programm passiert nichts außer das 2 Threads mit jeweils 100% Auslastung starten.

```

2. #include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>

#define NUMITER 26

#define TRUE 1
#define FALSE 0

typedef struct parallelstack {
    omp_lock_t stacklock; //lock for accessing the stack
    int cancel;           //flag that indicates if threads should stop working
    char *buffer;         //stack elements
    int size;             //size of the stack
    int count;            //current position in the stack
} ParallelStack;

static inline ParallelStack* newParallelStack() {
    return calloc(1, sizeof(ParallelStack));
}

```

```

}

static inline ParallelStack* ParallelStack_init(ParallelStack* pq, int size) {
    omp_init_lock(&pq[0].stacklock);

    return pq;
}

static inline ParallelStack* ParallelStack_deinit(ParallelStack* pq) {

    omp_destroy_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* freeParallelStack(ParallelStack* pq) {
    free(pq);
    return pq;
}

static int ParallelStack_put(ParallelStack* pq, char item) {
    int writtenChars = FALSE; // TRUE if the stack was able to put the data,
    // FALSE if the stack is full, the data will be rejected
    omp_set_lock(&pq[0].stacklock);

    omp_unset_lock(&pq[0].stacklock);
    return writtenChars;
}

int ParallelStack_get(ParallelStack* pq, char *c) {
    int numReadedChars = 0; // TRUE if the stack was able to get the data,
    // FALSE if the stack is empty
    omp_set_lock(&pq[0].stacklock);

    omp_unset_lock(&pq[0].stacklock);
    return numReadedChars;
}

void ParallelStack_setCanceled(ParallelStack* pq) {
    omp_set_lock(&pq[0].stacklock);

    omp_unset_lock(&pq[0].stacklock);
}

int ParallelStack_isCanceled(ParallelStack* pq) {
    int canceled = FALSE;
    omp_set_lock(&pq[0].stacklock);

    omp_unset_lock(&pq[0].stacklock);
    return canceled;
}

///////////////////////////////
// DO NOT EDIT BEYOND THIS LINE !!!! 
///////////////////////////////

void producer(int tid, ParallelStack* pq) {
    int i = 0;
    char item;
    while( i < NUMITER) {
        item = 'A' + (i % 26);

        if ( ParallelStack_put(pq, item) == 1) {

```

```

        i++;
        printf("->Thread %d is Producing %c ... \n", tid, item);
    }
    //sleep(1);
}
ParallelStack_setCanceled(pq);
}

void consumer(int tid, ParallelStack* pq)
{
    char item;
    while( ParallelStack_isCanceled(pq) == FALSE) {

        if (ParallelStack_get(pq, &item) == 1) {
            printf("<-Thread %d is Consuming %c\n", tid, item);
        }
        sleep(2);
    }
}

int main()
{
    int tid;
    ParallelStack* pq = ParallelStack_init(newParallelStack(), 5);

    #pragma omp parallel private(tid) num_threads(4)
    {
        tid=omp_get_thread_num();

        if(tid==1)
        {
            producer(tid, pq);
        } else
        {
            consumer(tid, pq);
        }
    }

    freeParallelStack(ParallelStack_deinit(pq));
}

return 0;
}

3. #include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>

#define NUMITER 26

#define TRUE 1
#define FALSE 0

typedef struct parallelstack {
    omp_lock_t stacklock; //lock for accessing the stack
    int cancel;           //flag that indicates if threads should stop working
    char *buffer;         //stack elements
    int size;             //size of the stack
    int count;            //current position in the stack
} ParallelStack;

static inline ParallelStack* newParallelStack() {

```

```

    return calloc(1, sizeof(ParallelStack));
}

static inline ParallelStack* ParallelStack_init(ParallelStack* pq, int size) {
    omp_init_lock(&pq[0].stacklock);
    char array[size];

    omp_set_lock(&pq[0].stacklock);
    pq[0].buffer = array;
    pq[0].size = size;
    pq[0].count = -1;
    omp_unset_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* ParallelStack_deinit(ParallelStack* pq) {
    omp_destroy_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* freeParallelStack(ParallelStack* pq) {
    free(pq);
    return pq;
}

static int ParallelStack_put(ParallelStack* pq, char item) {
    int writtenChars = FALSE; // TRUE if the stack was able to put the data, FALSE if the stack
    if(pq[0].count < pq[0].size){
        omp_set_lock(&pq[0].stacklock);
        pq[0].count++;
        pq[0].buffer[pq[0].count] = item;
        writtenChars = TRUE;
        omp_unset_lock(&pq[0].stacklock);
    }
    return writtenChars;
}

int ParallelStack_get(ParallelStack* pq, char *c) {
    int numReadedChars = 0; // TRUE if the stack was able to get the data, FALSE if the stack is
    if(pq[0].count > -1){
        omp_set_lock(&pq[0].stacklock);
        *c = pq[0].buffer[pq[0].count];
        pq[0].count--;
        numReadedChars = TRUE;
        omp_unset_lock(&pq[0].stacklock);
    }
    return numReadedChars;
}

void ParallelStack_setCanceled(ParallelStack* pq) {
    omp_set_lock(&pq[0].stacklock);
    pq[0].cancel = TRUE;
    omp_unset_lock(&pq[0].stacklock);
}

int ParallelStack_isCanceled(ParallelStack* pq) {
    int canceled = FALSE;
    omp_set_lock(&pq[0].stacklock);
    canceled = pq[0].cancel;
    omp_unset_lock(&pq[0].stacklock);
    return canceled;
}

```

```

}

///////////////
// DO NOT EDIT BEYOND THIS LINE !!!!
///////////////

void producer(int tid, ParallelStack* pq) {
    int i = 0;
    char item;
    while( i < NUMITER) {
        item = 'A' + (i % 26);

        if ( ParallelStack_put(pq, item) == 1) {
            i++;
            printf("->Thread %d is Producing %c ...\n",tid, item);
        }
        //sleep(1);
    }
    ParallelStack_setCanceled(pq);
}

void consumer(int tid, ParallelStack* pq)
{
    char item;
    while( ParallelStack_isCanceled(pq) == FALSE) {

        if (ParallelStack_get(pq, &item) == 1) {
            printf("<-Thread %d is Consuming %c\n",tid, item);
        }
        sleep(2);
    }
}

int main()
{
    int tid;
    ParallelStack* pq = ParallelStack_init(newParallelStack(), 5);

    #pragma omp parallel private(tid) num_threads(4)
    {
        tid=omp_get_thread_num();

        if(tid==1)
        {
            producer(tid, pq);
        } else
        {
            consumer(tid, pq);
        }
    }

    freeParallelStack(ParallelStack_deinit(pq));
}

return 0;
}

```

```

4. #include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>

#define NUMITER 26

```

```

#define TRUE 1
#define FALSE 0

typedef struct parallelstack {
    omp_lock_t stacklock; //lock for accessing the stack
    int cancel;           //flag that indicates if threads should stop working
    char *buffer;         //stack elements
    int size;             //size of the stack
    int count;            //current position in the stack
} ParallelStack;

static inline ParallelStack* newParallelStack() {
    return calloc(1, sizeof(ParallelStack));
}

static inline ParallelStack* ParallelStack_init(ParallelStack* pq, int size) {
    omp_init_lock(&pq[0].stacklock);
    char array[size];

    omp_set_lock(&pq[0].stacklock);
    pq[0].buffer = array;
    pq[0].size = size;
    pq[0].count = -1;
    omp_unset_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* ParallelStack_deinit(ParallelStack* pq) {
    omp_destroy_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* freeParallelStack(ParallelStack* pq) {
    free(pq);
    return pq;
}

static int ParallelStack_put(ParallelStack* pq, char item) {
    int writtenChars = FALSE; // TRUE if the stack was able to put the data,
                           // FALSE if the stack is full, the data will be rejected
    omp_set_lock(&pq[0].stacklock);
    if(pq[0].count < pq[0].size){
        pq[0].count++;
        pq[0].buffer[pq[0].count] = item;
        writtenChars = TRUE;
    }
    omp_unset_lock(&pq[0].stacklock);
    return writtenChars;
}

int ParallelStack_get(ParallelStack* pq, char *c) {
    int numReadedChars = 0; // TRUE if the stack was able to get the data,
                           // FALSE if the stack is empty
    omp_set_lock(&pq[0].stacklock);
    if(pq[0].count > -1){
        *c = pq[0].buffer[pq[0].count];
        pq[0].count--;
        numReadedChars = TRUE;
    }
    omp_unset_lock(&pq[0].stacklock);
    return numReadedChars;
}

```

```

}

void ParallelStack_setCanceled(ParallelStack* pq) {
    omp_set_lock(&pq[0].stacklock);
    pq[0].cancel = TRUE;
    omp_unset_lock(&pq[0].stacklock);
}

int ParallelStack_isCanceled(ParallelStack* pq) {
    int canceled = FALSE;
    omp_set_lock(&pq[0].stacklock);
    canceled = pq[0].cancel;
    omp_unset_lock(&pq[0].stacklock);
    return canceled;
}

///////////////////////////////
// DO NOT EDIT BEYOND THIS LINE !!!! 
///////////////////////////////

void producer(int tid, ParallelStack* pq) {
    int i = 0;
    char item;
    while( i < NUMITER) {
        item = 'A' + (i % 26);

        if ( ParallelStack_put(pq, item) == 1) {
            i++;
            printf("->Thread %d is Producing %c ...\n",tid, item);
        }
        //sleep(1);
    }
    ParallelStack_setCanceled(pq);
}

void consumer(int tid, ParallelStack* pq)
{
    char item;
    while( ParallelStack_isCanceled(pq) == FALSE) {

        if (ParallelStack_get(pq, &item) == 1) {
            printf("<-Thread %d is Consuming %c\n",tid, item);
        }
        sleep(2);
    }
}

int main()
{
    int tid;
    ParallelStack* pq = ParallelStack_init(newParallelStack(), 5);

    #pragma omp parallel private(tid) num_threads(4)
    {
        tid=omp_get_thread_num();

        if(tid==1)
        {
            producer(tid, pq);
        } else
        {

```

```

        consumer(tid, pq);
    }
}

freeParallelStack(ParallelStack_deinit(pq));

return 0;
}

```

5. Die folgende Implementierung ist nur die schnellste Art aus dem gegebenen Sourcecode eine Queue zu machen.
Es ist nicht unbedingt die performanteste Art und Weise.

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>

#define NUMITER 26

#define TRUE 1
#define FALSE 0

typedef struct parallelstack {
    omp_lock_t stacklock; //lock for accessing the stack
    int cancel;           //flag that indicates if threads should stop working
    char *buffer;         //stack elements
    int size;             //size of the stack
    int count;            //current position in the stack
} ParallelStack;

static inline ParallelStack* newParallelStack() {
    return calloc(1, sizeof(ParallelStack));
}

static inline ParallelStack* ParallelStack_init(ParallelStack* pq, int size) {
    omp_init_lock(&pq[0].stacklock);
    char array[size];

    omp_set_lock(&pq[0].stacklock);
    pq[0].buffer = array;
    pq[0].size = size;
    pq[0].count = -1;
    omp_unset_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* ParallelStack_deinit(ParallelStack* pq) {
    omp_destroy_lock(&pq[0].stacklock);
    return pq;
}

static inline ParallelStack* freeParallelStack(ParallelStack* pq) {
    free(pq);
    return pq;
}

static int ParallelStack_put(ParallelStack* pq, char item) {
    int writtenChars = FALSE; // TRUE if the stack was able to put the data, FALSE if the stack
    omp_set_lock(&pq[0].stacklock);
    if(pq[0].count < pq[0].size){
        pq[0].count++;
        pq[0].buffer[pq[0].count] = item;
    }
}

```

```

        writtenChars = TRUE;
    }
    omp_unset_lock(&pq[0].stacklock);
    return writtenChars;
}

int ParallelStack_get(ParallelStack* pq, char *c) {
    int numReadedChars = 0; // TRUE if the stack was able to get the data, FALSE if the stack is
    omp_set_lock(&pq[0].stacklock);
    if(pq[0].count > -1){
        *c = pq[0].buffer[0];
        for(int i = 0; i < pq[0].count; ++i){
            pq[0].buffer[i] = pq[0].buffer[i+1];
        }
        pq[0].count--;
        numReadedChars = TRUE;
    }
    omp_unset_lock(&pq[0].stacklock);
    return numReadedChars;
}

void ParallelStack_setCanceled(ParallelStack* pq) {
    omp_set_lock(&pq[0].stacklock);
    pq[0].cancel = TRUE;
    omp_unset_lock(&pq[0].stacklock);
}

int ParallelStack_isCanceled(ParallelStack* pq) {
    int canceled = FALSE;
    omp_set_lock(&pq[0].stacklock);
    canceled = pq[0].cancel;
    omp_unset_lock(&pq[0].stacklock);
    return canceled;
}

///////////////////////////////
// DO NOT EDIT BEYOND THIS LINE !!!
///////////////////////////////

void producer(int tid, ParallelStack* pq) {
    int i = 0;
    char item;
    while( i < NUMITER) {
        item = 'A' + (i % 26);

        if ( ParallelStack_put(pq, item) == 1) {
            i++;
            printf("->Thread %d is Producing %c ...\\n",tid, item);
        }
        //sleep(1);
    }
    ParallelStack_setCanceled(pq);
}

void consumer(int tid, ParallelStack* pq)
{
    char item;
    while( ParallelStack_isCanceled(pq) == FALSE) {

        if (ParallelStack_get(pq, &item) == 1) {
            printf("<-Thread %d is Consuming %c\\n",tid, item);
        }
    }
}

```

```

        }
        sleep(2);
    }
}

int main()
{
    int tid;
    ParallelStack* pq = ParallelStack_init(newParallelStack(), 5);

#pragma omp parallel private(tid) num_threads(4)
{
    tid=omp_get_thread_num();

    if(tid==1)
    {
        producer(tid, pq);
    } else
    {
        consumer(tid, pq);
    }
}

freeParallelStack(ParallelStack_deinit(pq));

return 0;
}

```

6. Wenn bei einer Queue nur ein Lock verwendet wird, dann kann ein hinzufügen eines Elements zu der Queue den Zugriff auf das vorderste Element der Queue blockieren. Im Gegensatz zum Stack sind diese beiden Operationen nicht immer von einander abhängig. Sie blockieren sich nur, wenn die Queue leer ist.

3 Übung 3

3.1 Aufgabe 1

1. Der Quellcode wurde um die folgenden Regeln ergänzt.

```

int evolve(unsigned* currentfield, unsigned* newfield, int w, int h) {
int changes = 0;

for (int y = 0; y < h; y++) {
    for (int x = 0; x < w; x++) {
        int sum = 0;
        sum += currentfield[calcIndex(w,h,x+1,y)];
        sum += currentfield[calcIndex(w,h,x+1,y+1)];
        sum += currentfield[calcIndex(w,h,x+1,y-1)];

        sum += currentfield[calcIndex(w,h,x-1,y)];
        sum += currentfield[calcIndex(w,h,x-1,y+1)];
        sum += currentfield[calcIndex(w,h,x-1,y-1)];

        sum += currentfield[calcIndex(w,h,x,y-1)];
        sum += currentfield[calcIndex(w,h,x,y+1)];
        sum += currentfield[calcIndex(w,h,x,y)];

        if(sum == 2 || sum == 3){
            newfield[calcIndex(w,h,x,y)] = 1;
        } else {
            newfield[calcIndex(w,h,x,y)] = 0;
        }
    }
}

```

```

    }

    if(changes==0 && currentfield[calcIndex(w,h,x,y)] != newfield[calcIndex(w,h,x,y)]) {
        changes = 1;
    }
}

return changes;
}

```

Die periodischen Ränder wurden in der Methode currentfield ungesetzt. Hierfür wurde die Parameterliste um die angabe der Höhe des Feldes ergänzt.

```

int calcIndex(int width, int height, int x, int y) {
    if(x > width){
        x = 0;
    }else if(x < 0){
        x = width;
    }
    if(y > height){
        y = 0;
    }else if(y < 0){
        y = height;
    }
    return y * width + x;
}

```

Das Rausschreiben der Daten erfolgte über die schon vorhandene Methode.

3.2 Aufgabe 2

```

int evolve(unsigned* currentfield, unsigned* newfield, int w, int h) {
int changes = 1;

#pragma omp parallel
{
    int num_threads = omp_get_num_threads();

    //Teilen in der Hoehe
    int partHeight = h/num_threads;

    //start Index berechnen
    int startIndex = partHeight * omp_get_thread_num();

    //teile in vertikale streifen
    for (int y = startIndex; y < startIndex + partHeight -1; y++) {
        for (int x = 1; x < w-1; x++) {
            int sum = 0;
            sum += currentfield[calcIndex(w,h,x+1,y)];
            sum += currentfield[calcIndex(w,h,x+1,y+1)];
            sum += currentfield[calcIndex(w,h,x+1,y-1)];

            sum += currentfield[calcIndex(w,h,x-1,y)];
            sum += currentfield[calcIndex(w,h,x-1,y+1)];
            sum += currentfield[calcIndex(w,h,x-1,y-1)];

            sum += currentfield[calcIndex(w,h,x,y-1)];
            sum += currentfield[calcIndex(w,h,x,y+1)];

            if(!currentfield[calcIndex(w,h,x,y)] && (2 == sum)) {
                newfield[calcIndex(w,h,x,y)] = 0;
            }
        }
    }
}

```

```

    }
    if(!underpopulated(sum) && !overpopulated(sum)) {
        newfield[calcIndex(w,h,x,y)] = 1;
    }else{
        newfield[calcIndex(w,h,x,y)] = 0;
    }

    if(changes==0 && currentfield[calcIndex(w,h,x,y)] != newfield[calcIndex(w,h,x,y)]) {
        changes = 1;
    }
}
}

return changes;
}
}

```

Das Feld wird in vertikale Streifen geteilt und dann berechnet. Am Ende werden die Streifen wieder zu einem Feld zusammengesetzt.

4 Übung 4

4.1 Aufgabe 1

Gründe:

- Laufzeitanalyse
- Zusammenhang zwischen paralleler und sequenzieller Ausführung zeigen
- Beschleunigungs Auswertung, bzw. Effizienzbetrachtungen

4.2 Aufgabe 2

a) - c)

Testreihe 1

CPU	1	10	20	30	40	50	60	70	80	90	100
Zeit	1000	105,3	55,6	39,2	31,2	26,7	23,8	21,9	20,8	20,2	20
Id.	1	10	20	30	40	50	60	70	80	90	100
Speedup											
Speedup	1	9,5	18,0	25,5	32,1	37,5	42,0	45,7	48,1	49,5	50
Effizienz	1	0,95	0,9	0,85	0,8	0,75	0,7	0,65	0,6	0,55	0,5

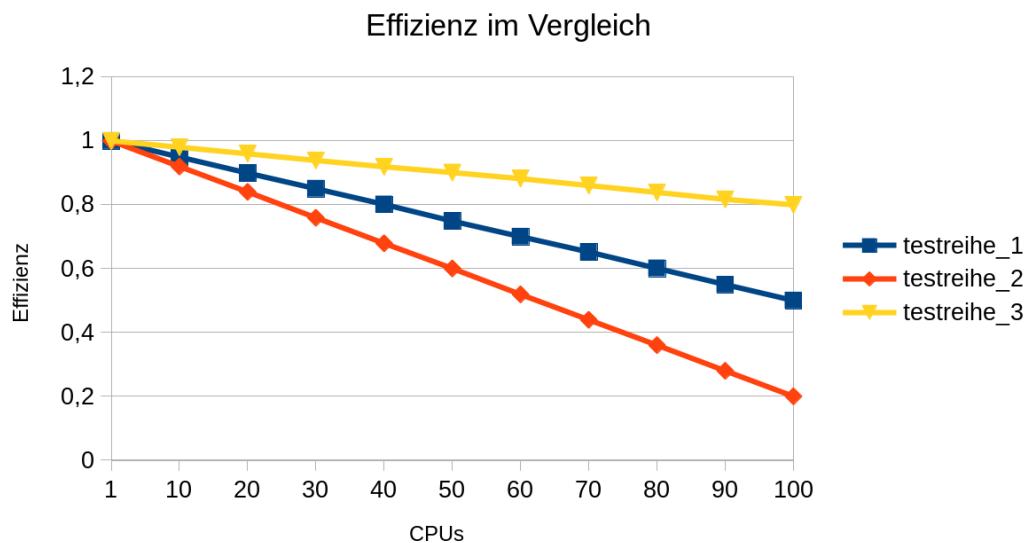
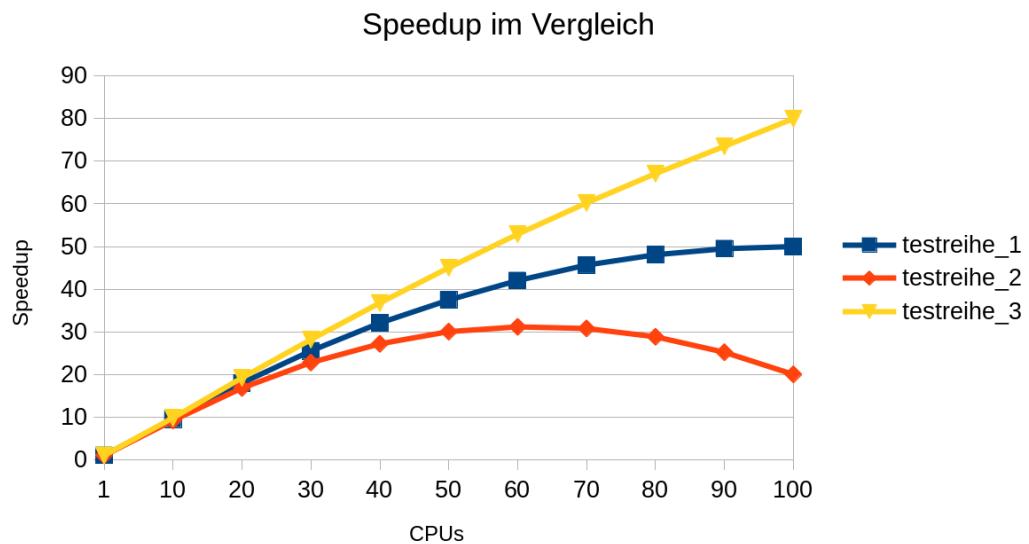
Testreihe 2

CPU	1	10	20	30	40	50	60	70	80	90	100
Zeit	1000	108,7	59,5	43,9	36,8	33,3	32,1	32,5	34,7	39,7	50
Id.	1	10	20	30	40	50	60	70	80	90	100
Speedup											
Speedup	1	9,2	16,8	22,8	27,2	30	31,2	30,8	28,8	25,2	20
Effizienz	1	0,92	0,84	0,76	0,68	0,6	0,52	0,44	0,36	0,28	0,2

Testreihe 3

CPUs	1	10	20	30	40	50	60	70	80	90	100
Zeit	1000	102,0	52,1	35,5	27,2	22,2	18,9	16,6	14,9	13,6	12,5
Id.	1	10	20	30	40	50	60	70	80	90	100
Speedup											
Speedup	1	9,8	19,2	28,2	36,8	45	52,9	60,2	67,1	73,5	80
Effizienz	1	0,98	0,96	0,94	0,92	0,90	0,88	0,86	0,84	0,82	0,80

d)



e)

Über die erste Reihe, kann gesagt werden, dass diese ab ca. 100 CPUs nicht schneller wird. Der Speedup der zweiten Reihe bricht ab etwa 60 CPUs ein, also sie skaliert nicht weiter. Die dritte Reihe skaliert sehr gut und bricht auch bei 100 CPUs nicht ein.

4.3 Aufgabe 3

a)

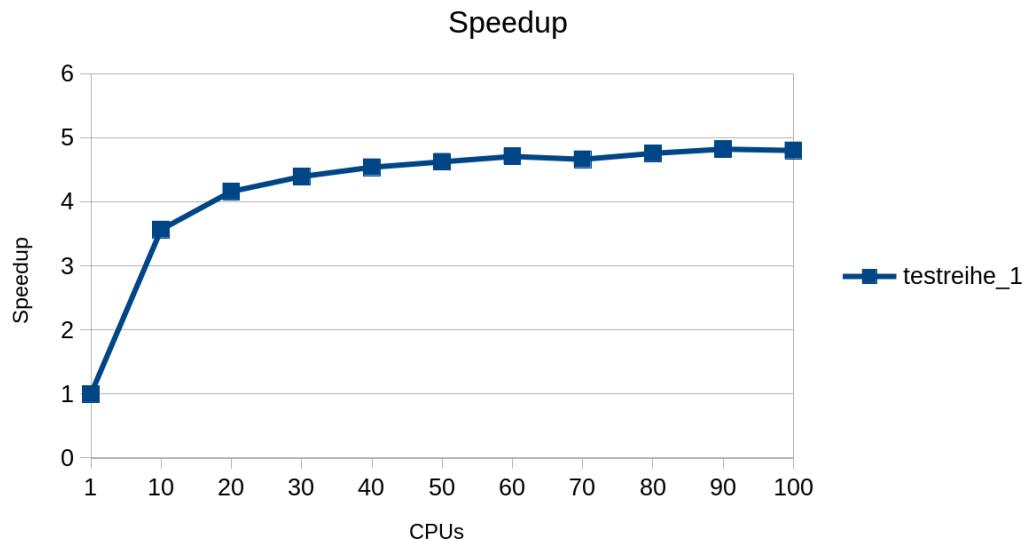
Testreihe 1

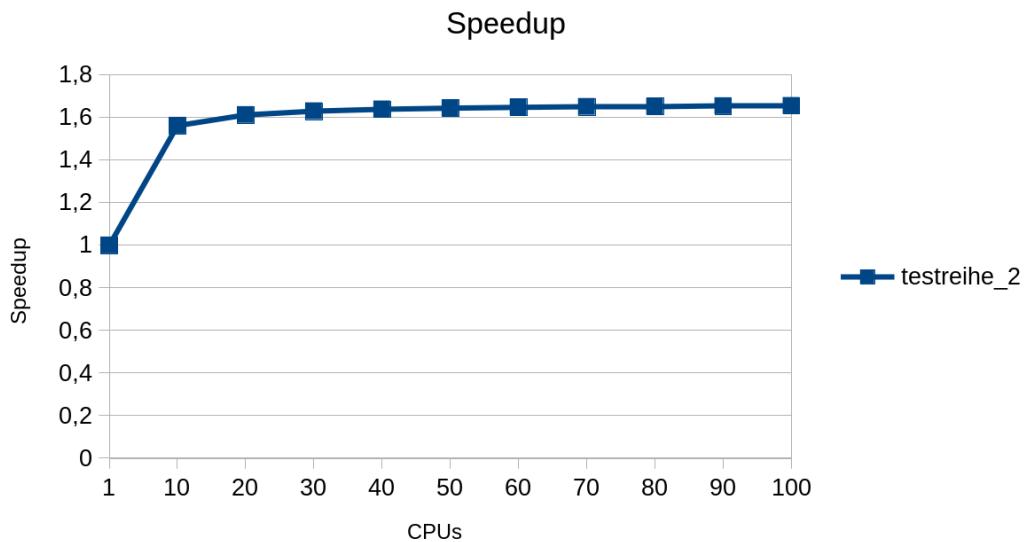
CPUs	1	10	20	30	40	50	60	70	80	90	100
par. Laufzeit	800	80	40	26,7	20	16	13,3	11,4	10	8,9	8
seq. Laufzeit	200	20	10	6,7	5	4	3,3	2,9	2,5	2,2	2
T gesamt	1000	100	50	33,4	25	20	16,6	14,3	12,5	11,1	10
t para	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
t seq	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
Id. Speedup	1	10	20	30	40	50	60	70	80	90	100
Speedup	1	3,57	4,17	4,4	4,55	4,63	4,71	4,67	4,76	4,83	4,81

Testreihe 2

CPUs	1	10	20	30	40	50	60	70	80	90	100
t para	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4
t seq	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6
Id. Speedup	1	10	20	30	40	50	60	70	80	90	100
Speedup	1	1,56	1,61	1,63	1,64	1,64	1,65	1,65	1,65	1,65	1,65

b)





c)

Beide Testreihen weisen nur einen geringen Speedup auf, was darauf hindeutet, dass sich das Problem nicht gut parallelisieren lässt, oder dass ein Fehler in der Implementierung vorliegt.

4.4 Aufgabe 4

a)

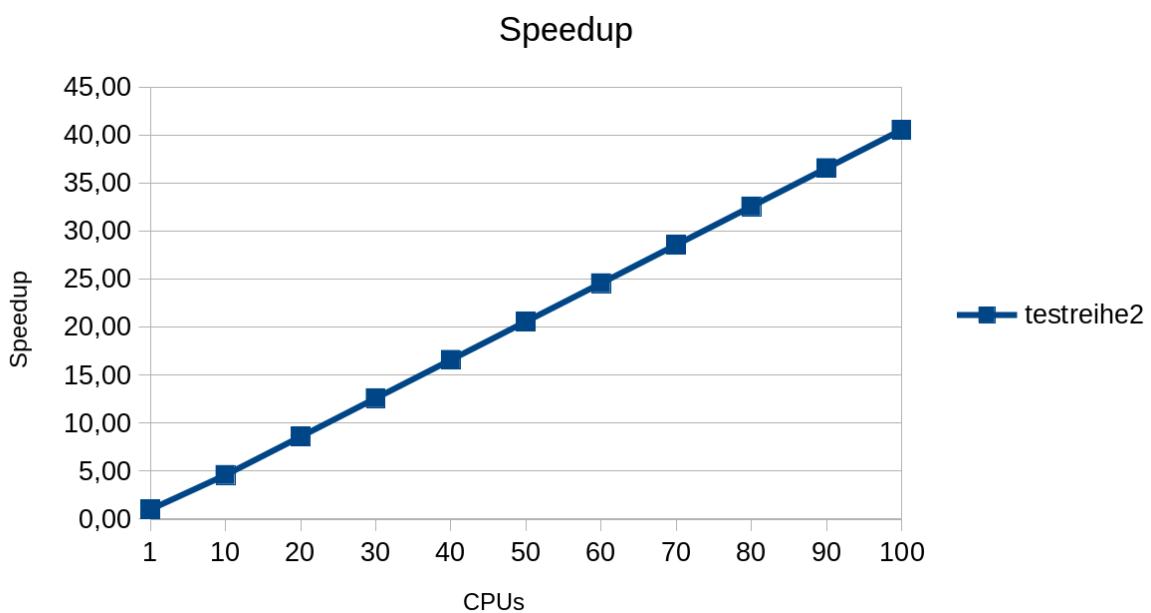
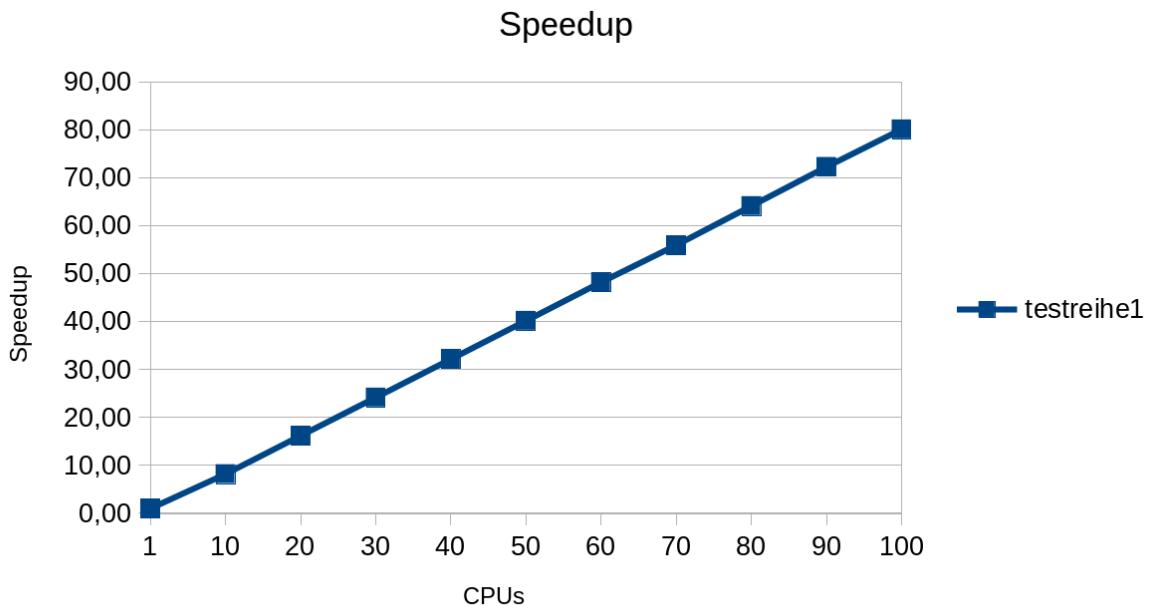
Testreihe 1

CPUs	1	10	20	30	40	50	60	70	80	90	100
par. Laufzeit	800	80	40	26,7	20	16	13,3	11,4	10	8,9	8
seq. Laufzeit	200	20	10	6,7	5	4	3,3	2,9	2,5	2,2	2
T gesamt	1000	100	50	33,4	25	20	16,6	14,3	12,5	11,1	10
t para	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
t seq	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
Id. Speedup	1	10	20	30	40	50	60	70	80	90	100
Speedup	1	8,2	16,2	24,18	32,2	40,2	48,27	56,01	64,2	72,36	80,2

Testreihe 2

CPUs	1	10	20	30	40	50	60	70	80	90	100
t para	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4
t seq	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6
Id. Speedup	1	10	20	30	40	50	60	70	80	90	100
Speedup	1	4,6	8,6	12,6	16,6	20,6	24,6	28,6	32,6	36,6	40,6

b)



c)

Über die beiden Messreihen kann gesagt werden, dass sie einen in ungefähr liniaren Speedup besitzen.

5 Übung 5

5.1 Aufgabe 1

a)

```
cd hello-world  
make run
```

b)

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <mpi.h>  
  
int main(int argc, char **argv) {  
    int rank = 0;  
    int size = 0;  
  
    MPI_Init (&argc, &argv);      /* starts MPI */  
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);          /* get current process id */  
    MPI_Comm_size (MPI_COMM_WORLD, &size);          /* get number of processes */  
    printf( "Hello world from process %d of %d\n", rank, size );  
    MPI_Finalize();  
    return 0;  
}
```

c)

Die Ausgabe ist nicht konstant, weil die Threads bei jeder Ausgabe unterschiedlich schnell sind.

5.2 Aufgabe 2

a)

```
static void a(double pi) {  
    double receivedPi;  
    int length = 1;  
    MPI_Status status;  
    //Ring  
    if(rank == 0){  
        MPI_Send(&pi, length, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD);  
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, size - 1, 99, MPI_COMM_WORLD, &status);  
        pi = (pi + receivedPi)/2;  
        printf("pi from Ring is %.9lf\n", pi);  
    }else if(rank < size - 1){  
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD, &status);  
        pi = (pi + receivedPi)/2;  
        MPI_Send(&pi, length, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD);  
    }else{  
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD, &status);  
        pi = (pi + receivedPi)/2;  
        MPI_Send(&pi, length, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);  
    }  
}
```

Jeder Prozess, sendet und empfängt eine Nachricht. Somit werden n Sende- und Empfangsschritte benötigt, wobei n die Anzahl der Prozesse darstellt.

b)

```
static void b(double pi) {
    //Nachbar austausch
    double receivedPi;
    int length = 1;
    MPI_Status status;
    if(rank == 0){
        MPI_Send(&pi, length, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD);
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, size - 1, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        MPI_Send(&pi, length, MPI_DOUBLE, size - 1, 99, MPI_COMM_WORLD);
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        printf("pi from Austausch is %.9lf\n", pi);
    }else if(rank < size - 1){
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        MPI_Send(&pi, length, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD);
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank + 1, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        MPI_Send(&pi, length, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD);
    }else{
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        MPI_Send(&pi, length, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
        MPI_Recv(&receivedPi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &status);
        pi = (pi + receivedPi)/2;
        MPI_Send(&pi, length, MPI_DOUBLE, rank - 1, 99, MPI_COMM_WORLD);
    }
}
```

Jeder Prozess, sendet und empfängt zwei Nachrichten. Somit werden n^2 Sende- und Empfangsschritte benötigt, wobei n die Anzahl der Prozesse darstellt.

c)

```
double sumPi;
double avageragePi;
int length = 1;
int root = 0;

MPI_Reduce(&pi, &sumPi, length, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD );

if (rank == root){
    avageragePi = sumPi/size;
    printf("pi from Reduce&BroadCast is %.9lf\n", avageragePi);

    MPI_Bcast(&avageragePi, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
} else {
    MPI_Bcast(&avageragePi, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
}
```

Aufgabe gelöst über MPI Reduce.

d)

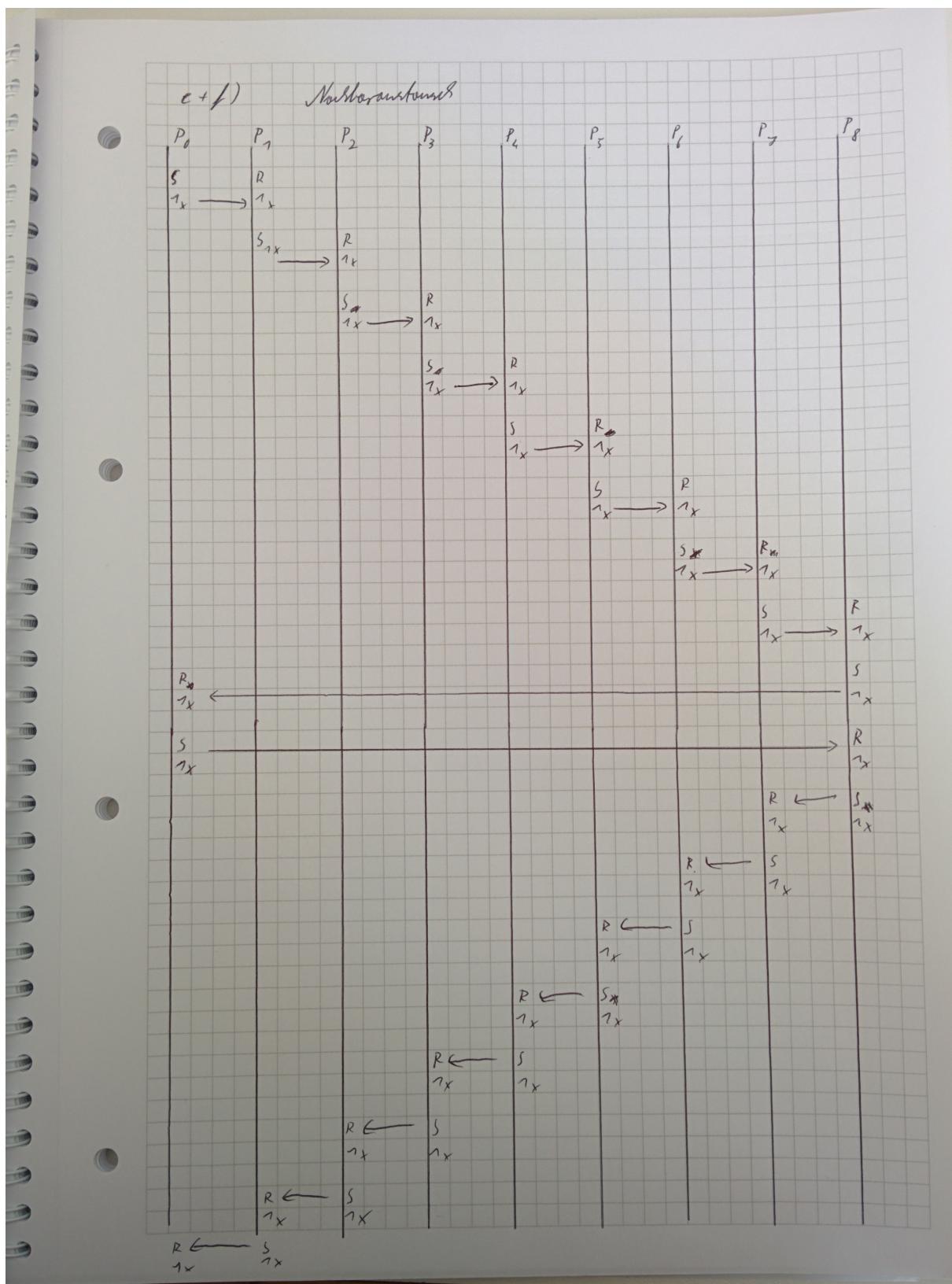
```
double sumPi;
double avageragePi;
int length = 1;
int root = 0;

MPI_Allreduce(&pi, &sumPi, length, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );

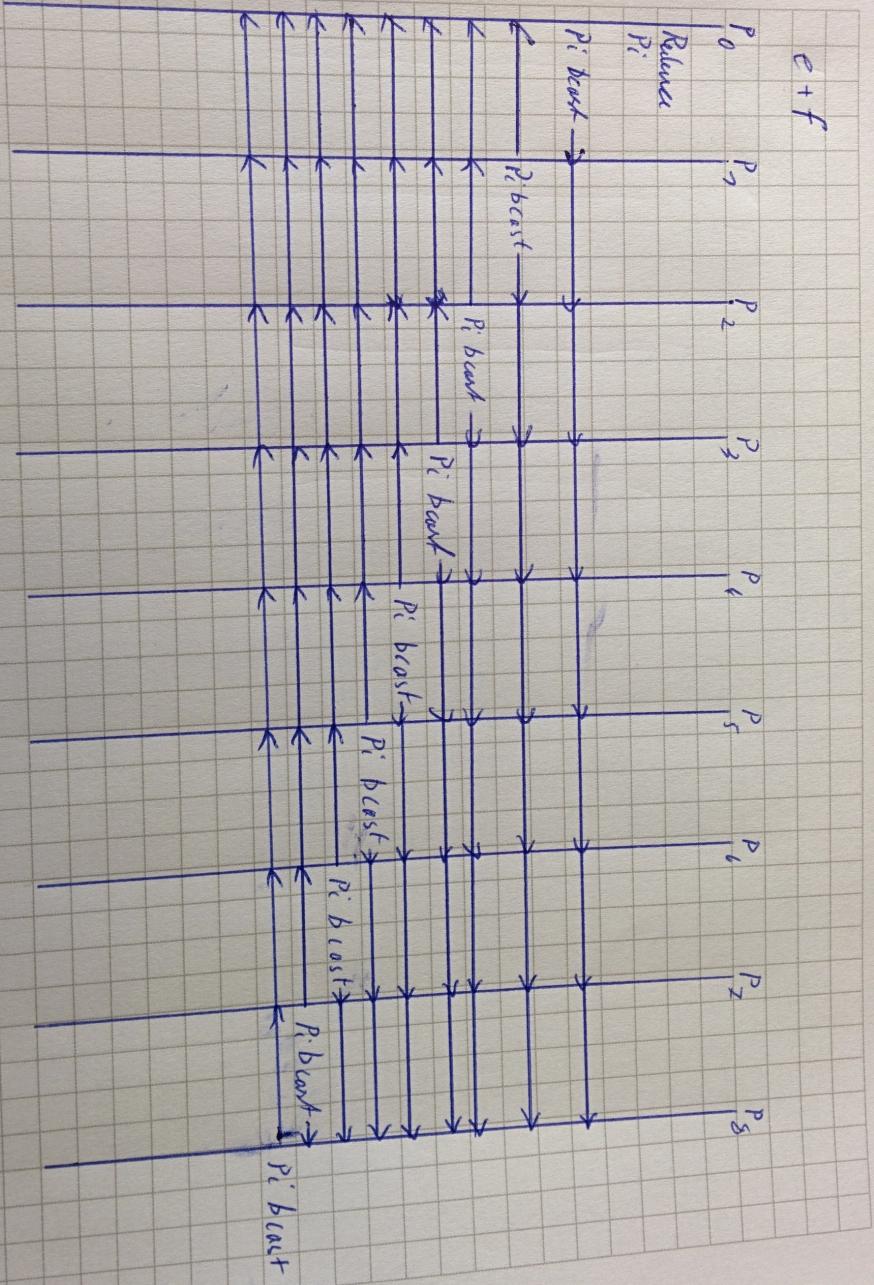
avageragePi = sumPi/size;
if (rank == root){
    printf("pi from Allreduce is %.9lf\n", avageragePi);
}
```

Aufgabe gelöst über MPI ReduceAll.

e+f)

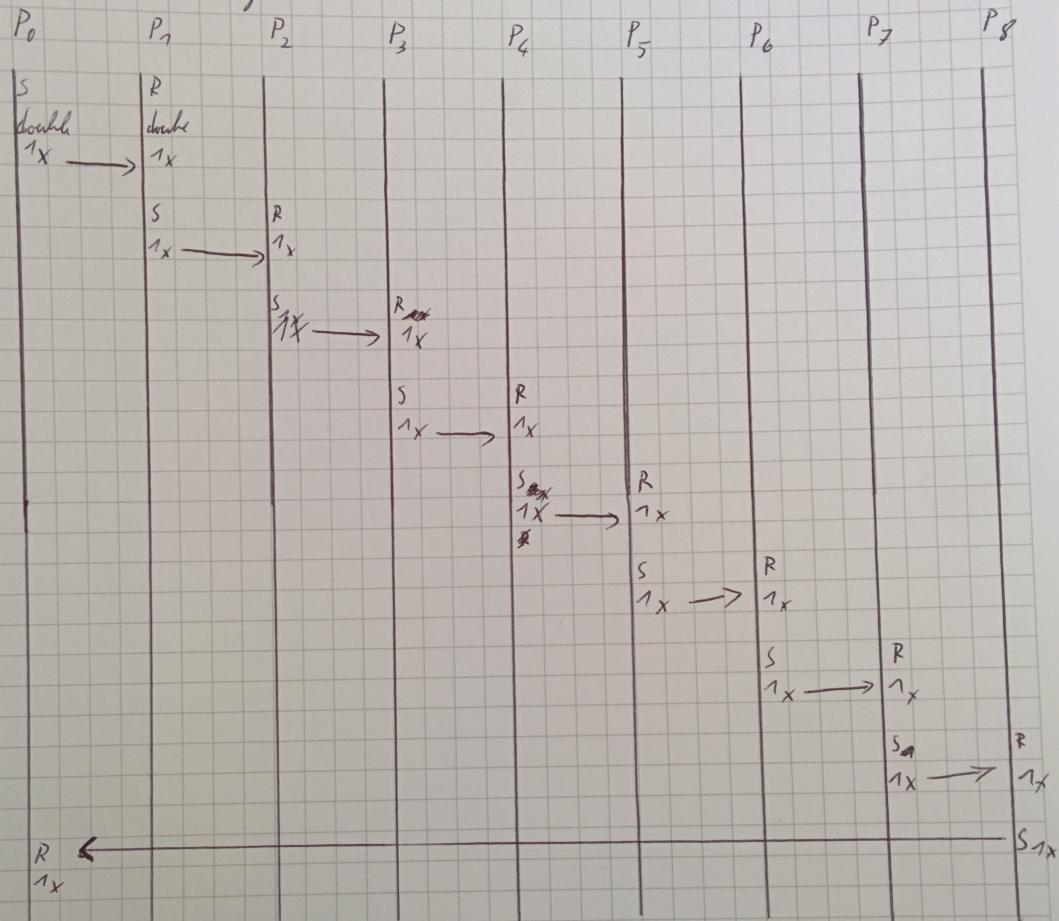


$e + f$



c) f)

Ring



Gesendet wird immer $1x$ double-Wert an den nächst größeren Prozess

Der größte Prozess sendet seine Nachricht an P_0 .

Der Übertragungslauf kann werden. Datentyp, Größe, Tag nicht im Diagramm angeführt. Sender ist mit S und Receiver mit R gekennzeichnet.

$c + r$

P_0	$P_{i,0}$	$P_{i,\text{sum}}$
P_1	$P_{i,1}$	$P_{i,\text{sum}}$
P_2	$P_{i,2}$	$P_{i,\text{sum}}$
P_3	$P_{i,3}$	$P_{i,\text{sum}}$
P_4	$P_{i,4}$	$P_{i,\text{sum}}$
P_5	$P_{i,5}$	$P_{i,\text{sum}}$
P_6	$P_{i,6}$	$P_{i,\text{sum}}$
P_7	$P_{i,7}$	$P_{i,\text{sum}}$
P_8	$P_{i,8}$	$P_{i,\text{sum}}$

AllReduce
↔

5.3 Aufgabe 3

a)

```

while(1) {
    //printf("in while");
    if(tokenA == 0) {
        tokenA = 1;
        writeData();
        tokenA = 0;
        break;
    }
}

```

In einer Schleife wird gewartet, bis das Token frei ist. Ist dies der Fall, lockt der Thread das Token und schreibt. Nach dem schreiben gibt er das Token wieder frei. In der Zeit, wo das Token blockiert ist, kann kein anderer Thread schreiben.

b)

```

while(1) {
    //printf("in while");
    int i;
    int wasWritten = 0;
    for(i = 0; i < N; i++) {
        if(tokenB[i] == 0) {
            tokenB[i] = 1;
            writeData();
            wasWritten = 1;
            tokenB[i] = 0;
            break;
        }
    }
    if(wasWritten)
        break;
}

```

Wie Aufgabe a, so verhält sich auch Aufgabe b, nur das hier drei Token zur Verfügung stehen, welche gleichzeitig allokiert werden können.

c)

Da die maximalen Datenraten von Festplatten und Netzwerktechniken begrenzt sind, muss der I/O gebremst werden. Würden extrem viele Prozesse auf einmal auf ein Laufwerke zugreifen, wäre eine Überlastung vorprogrammiert!

6 Übung 6

6.1 a)

make run

6.2 b)

```

int rank, size, coord[1];
MPI_Comm card_comm;

...

```

```

//init MPI
int dim[1], periodic [1];
periodic[0] = 1;
int reorder = 1;

MPI_Init(&c, &v);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

dim[0] = size;

//decompose Domain and processes with 1D domain decompositon
MPI_Cart_create(MPI_COMM_WORLD, 1, dim, periodic, &card_comm);

//get reordered cart_rank and neighbours ranks
MPI_Cart_coords(card_comm, rank, 1, coord);
int cart_rank;
MPI_Cart_rank(card_comm, coord, &cart_rank);

int left_neighbour_rank, right_neighbour_rank, rank_source;
MPI_Cart_shift(card_comm, 1, -1, &rank_source, &left_neighbour_rank);
MPI_Cart_shift(card_comm, 1, 1, &rank_source, &right_neighbour_rank);

printf("Rank ID: %d, Left Neighbour rank: %d , Right Neighbour rank: %d \n", cart_rank,
      left_neighbour_rank, right_neighbour_rank);

```

6.3 c)

```

void writeVTK(unsigned* currentfield, int w, int h, int t, char* prefix) {
    char name[1024] = "\0";
    sprintf(name, "%s_%d_%d.vtk", prefix, coord[0], t);
    FILE* outfile = fopen(name, "w");

    /*Write vtk header */
    fprintf(outfile, "# vtk DataFile Version 3.0\n");
    fprintf(outfile, "frame %d\n", t);
    fprintf(outfile, "BINARY\n");
    fprintf(outfile, "DATASET STRUCTURED_POINTS\n");
    fprintf(outfile, "DIMENSIONS %d %d %d \n", w, h/size, 1);
    fprintf(outfile, "SPACING 1.0 1.0 1.0\n"); //or ASPECT_RATIO
    fprintf(outfile, "ORIGIN 0 %d 0 \n", coord[0]*(h/size));
    fprintf(outfile, "POINT_DATA %d\n", h/size);
    fprintf(outfile, "SCALARS data float 1\n");
    fprintf(outfile, "LOOKUP_TABLE default\n");

    for (int y = 1; y <= h/size; y++) {
        for (int x = 0; x < w; x++) {

            float value = currentfield[calcIndex(w, x, y)]; // != 0.0 ? 1.0:0.0;
            value = convert2BigEndian(value);
            fwrite(&value, 1, sizeof(float), outfile);
        }
    }
    fclose(outfile);
}

```

Die Ghostboundarys dürfen nicht mit in die Datei geschrieben werden, da diese ja von dem jeweiligen Prozess nicht berechnet wurden, sondern nur von seinen Nachbarprozessen.

6.4 d)

```
int fieldsize = w*((h/size)+2);
unsigned *currentfield = calloc(fieldsize, sizeof(unsigned));
unsigned *newfield      = calloc(fieldsize, sizeof(unsigned));

printf("Rank ID: %d, Coord: %d , Fieldsize: %d \n", cart_rank, coord[0], fieldsize);

filling(currentfield, w, h/size);
```

Das Feld wird vertikal durch die Anzahl der Prozesse geteilt. Bei 5 Prozessen und 30 Felder berechnet also jeder Prozess 6 Reihen. Zusätzlich bekommt jedes Feld noch 2 Reihen für den Ghostboundary.

6.5 e)

```
//exchange boundary
int data_to_left[w], data_to_right[w], data_from_left[w], data_from_right[w];

for(int i = 0; i < w; ++i){
    data_to_left[i] = currentfield[calcIndex(w, i, 1)];
    data_to_right[i] = currentfield[calcIndex(w, i, gridh-1)];
}

//send to left neighbour
MPI_Request to_left;
MPI_Isend(&data_to_left, w, MPI_INT, left_neighbour_rank, 0, card_comm, &to_left);

//send to right neighbour
MPI_Request to_right;
MPI_Isend(&data_to_right, w, MPI_INT, right_neighbour_rank, 0, card_comm, &to_right);

//receive from left neighbour
MPI_Request from_left;
MPI_Irecv(&data_from_left, w, MPI_INT, left_neighbour_rank, 0, card_comm, &from_left);

//receive from right neighbour
MPI_Request from_right;
MPI_Irecv(&data_from_right, w, MPI_INT, right_neighbour_rank, 0, card_comm, &from_right);

//wait
MPI_Status status_left;
MPI_Status status_right;
MPI_Wait(&from_left, &status_left);
MPI_Wait(&from_right, &status_right);

//update boundaries
for(int i = 0; i < w; ++i){
    currentfield[calcIndex(w, i, 0)] = data_from_left[i];
    currentfield[calcIndex(w, i, gridh)] = data_from_right[i];
}
```

Bei einer 1D-Gebietszerlegung wird nur in eine Richtung kommuniziert, da sich die andere Richtung komplett im eigenen Gebiet befindet und deshalb nicht, ausgetauscht werden muss. In diesem Fall wurden die Reihen augetauscht, weil dies Speichertechnisch sinnvoller ist.

Bei der Buffervariante ist die Übertragung simpler, weil kein Datentype definiert werden muss. Dafür kann man sich bei der Datentype-Variante auf den Datentype verlassen.

6.6 f)

```
//exit loop if all processes report no more changes
int allchanges;
MPI_Allreduce(&changes, &allchanges, size, MPI_INT, MPI_SUM, card_comm);
if (allchanges == 0) break;
```

6.7 g)

Bei der 2D Zerlegung muss der Randaustausch in beide Richtungen durchgeführt werden. Was ihn komplizierter macht.

Außerdem muss die Anzahl der Prozesse sich auf das Gebiet aufteilen lassen. Also $w \% y = 0$ und $h \% x = 0$ sowie $y * w = n$, wo bei x =Prozesse in Dimension1, y =Prozesse in Dimension 2 und n =Anzahl der Prozesse.

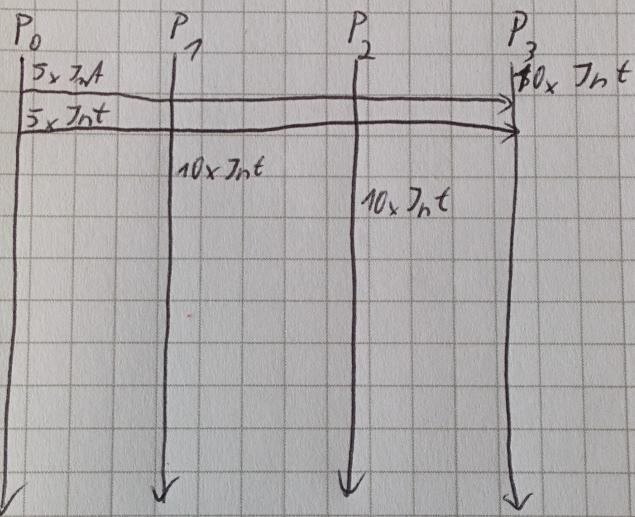
7 Übung 7

7.1 Aufgabe 1

1.

7.1

Für ~~die~~ ^{ein} Prozesse



2.

Um die Deadlocks zu entfernen, müssen die richtigen Absender gewählt werden (Zeile 30, 31).

3.

Die Messagetags bei Send und Receive stimmen nicht überein, hier muss nachgebessert werden.

Die Sendegröße stimmt nicht überein. Es werden 10 empfangen, aber zwei mal 5 gesendet.

4.

Das Programm terminiert, wenn alle erforderlichen Änderungen von McKay vorgenommen wurden.

Sollte er nicht schnell genug sein, legen sie ihn einfach eine Zitrone hin oder bedrohen in! ;)