HPC Übungen

Release 1.0

Sarah Blume, Sebastian Rieger

12.03.2015

Inhaltsverzeichnis

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

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

   Von l.36 bis l.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 l.58 und warten auf die Threads, welche den Sectionblock abarbeiten.

   Diese Threads laufen aber gegen die Barrier in l.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) l.86 löschen

2. Der Fehler tritt durch ein Deadlock auf, dies geschieht da die beiden Sections ihre Locks nicht zu Beginn setzen.

   In l.45 locked der erste Thread locka und zeitgleich in l. 59 wird durch den anderen Thread lockb gelocked. Laufen nun beide Threads weiter wartet der erste Thread in l.48 auf das unlock von lockb und der andere Thread in l.62 auf das unlock von locka.

   Der Fehler kann wieder durch mindesten 3 Arten gelöst werden.

   (a) Der Bereich wird nicht paraellisiert sondern sequentiell hintereinander ausgeführt.

   (b) Nur einen CPU Core verwenden, weil denn 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 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 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 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 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

1. Bei der Ausführung kann beobachtet werden, das 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.
   ```

2. Unsere Philosophen sind höflich, nachdem sie geggessen haben, denken sie ersteinmal 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 abel 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 abel 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 abel 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 abel to get the data, FALSE if the stack i
    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 abel 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 abel 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 folgene Implemenierung ist nur die schnellste Art aus dem gegegben Sourcecode eine Queue zu machen. Es ist nicht umbedingt die performateste 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 abel 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 abel to get the data, FALSE if the stack i
  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 Elemet 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 4

3.1 Aufgabe 1

Gründe:

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

3.2 Aufgabe 2

a) - c)

Testreihe 1

CPUs	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. Speedup	1	10	20	30	40	50	60	70	80	90	100
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

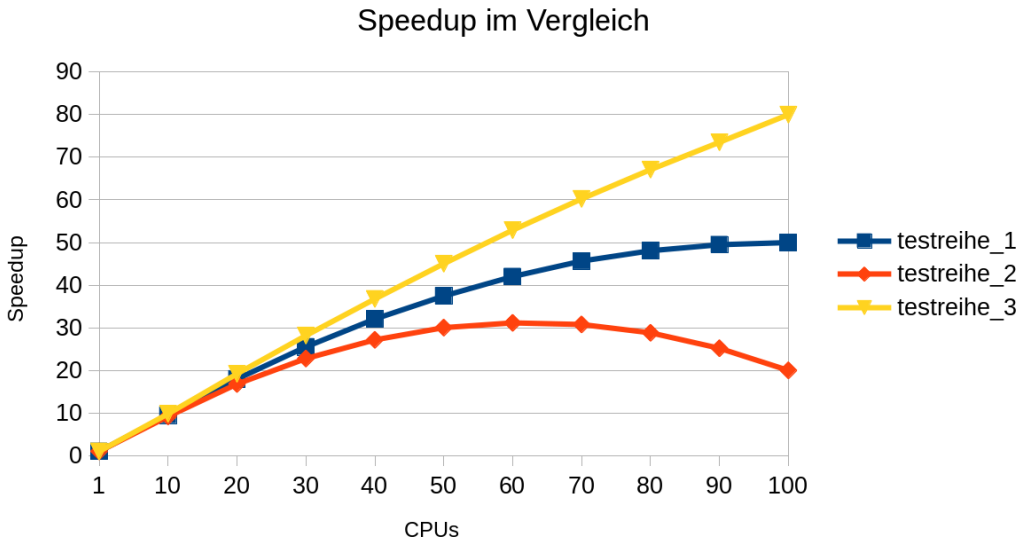
Testreihe 2

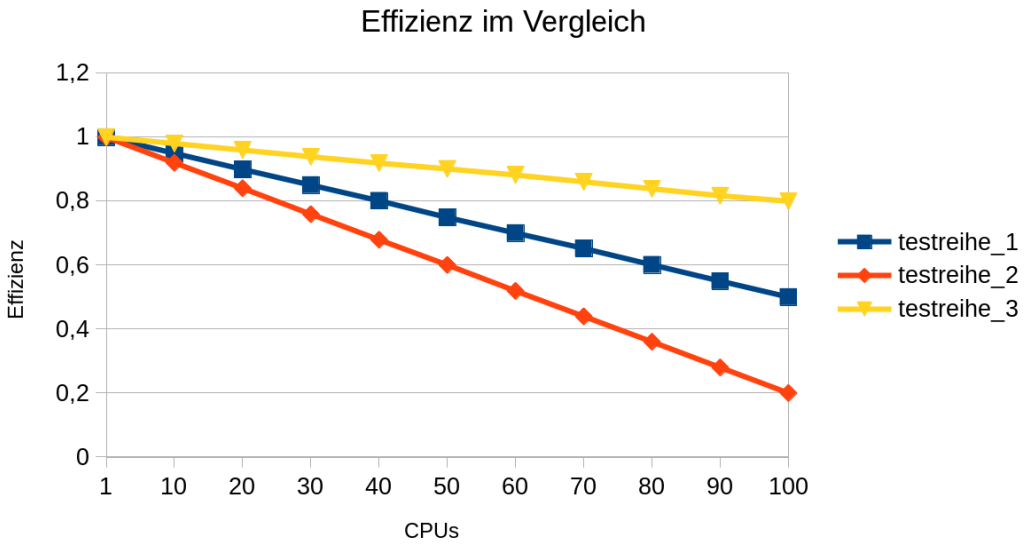
CPUs	1	10	20	30	40	50	60	70	80	90	100
Zeit	1000	108,7	59,5	43,9	36,8	33,3	32,1	32,5	34,7	39,7	50
Id. Speedup	1	10	20	30	40	50	60	70	80	90	100
Speedup	1	9,2	16,8	22,8	27,2	30	31,2	30,8	28,8	25,2	20
Effizienz	1	0,92	0,84	0,76	0,68	0,6	0,52	0,44	0,36	0,28	0,2

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. Speedup	1	10	20	30	40	50	60	70	80	90	100
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 zweite Reihe bricht ab etwa 60 CPUs ein, also sie skalliert nicht weiter. Die dritte Reihe skalliert sehr gut und bricht auch bei 100 CPUs nicht ein.

3.3 Aufgabe 3

a)

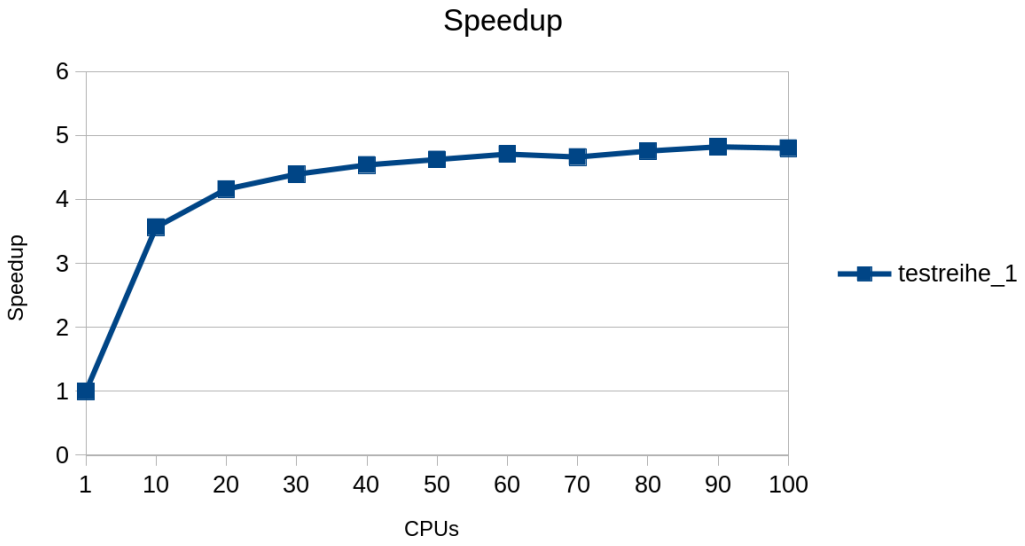
Testreihe 1

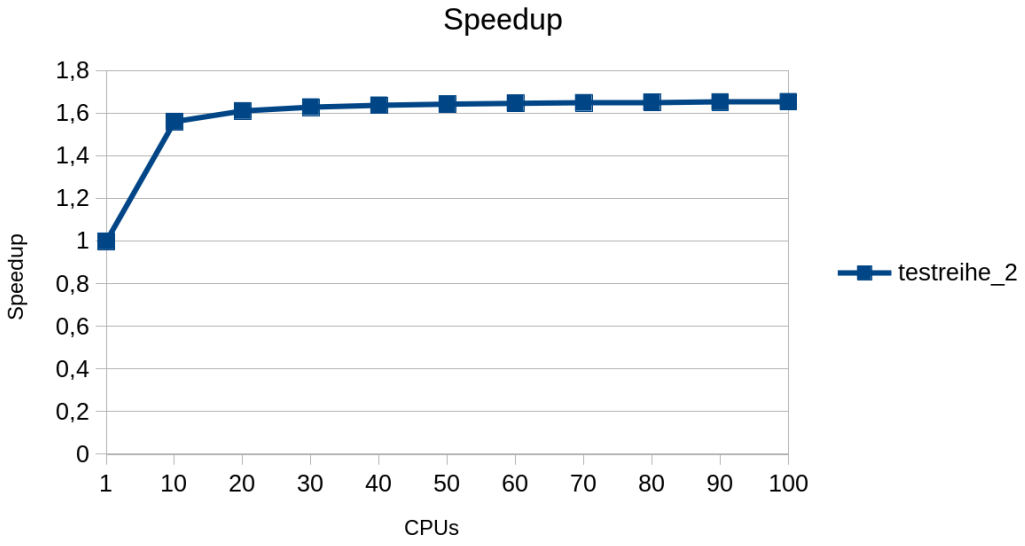
CPUs	1	10	20	30	40	50	60	70	80	90	100
par. Laufzeit	800	80	40	26,7	20	16	13,3	11,4	10	8,9	8
seq. Laufzeit	200	20	10	6,7	5	4	3,3	2,9	2,5	2,2	2
T gesamt	1000	100	50	33,4	25	20	16,6	14,3	12,5	11,1	10
t para	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
t seq	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
Id. 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 hindeuted, dass sich das Probelm nicht gut parallelisieren lässt, oder dass ein Fehler in der Implementierung vorliegt.

3.4 Aufgabe 4

a)

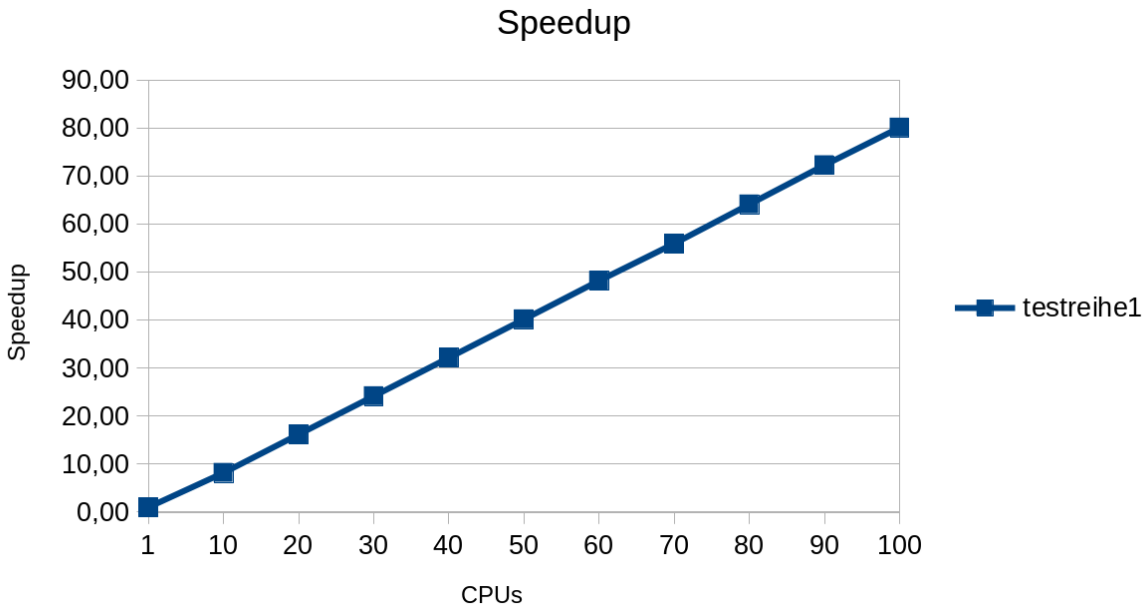
Testreihe 1

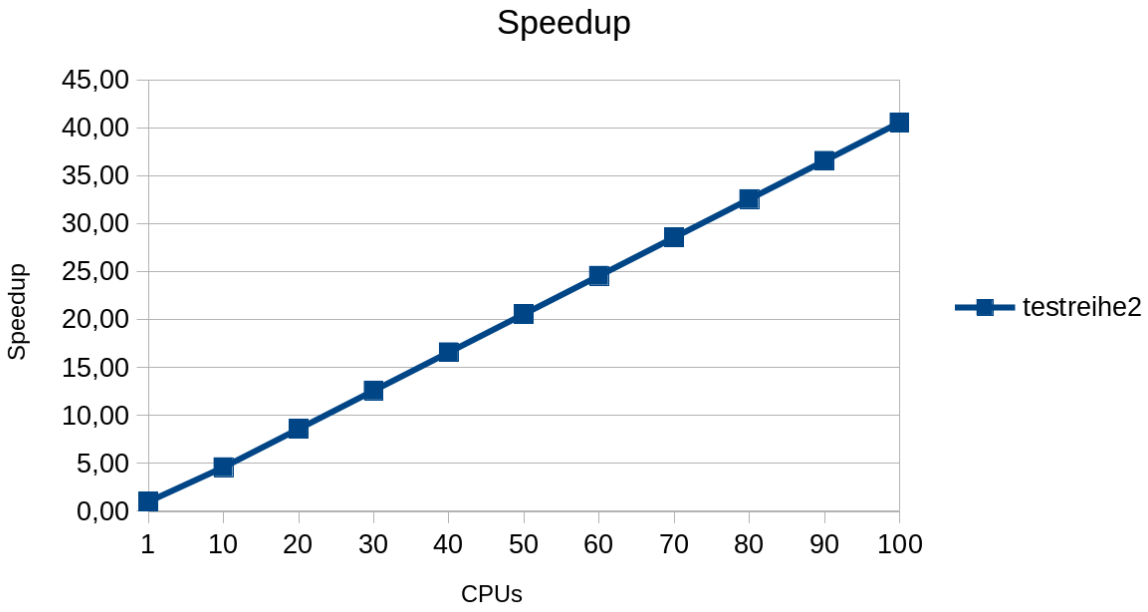
CPUs	1	10	20	30	40	50	60	70	80	90	100
par. Laufzeit	800	80	40	26,7	20	16	13,3	11,4	10	8,9	8
seq. Laufzeit	200	20	10	6,7	5	4	3,3	2,9	2,5	2,2	2
T gesamt	1000	100	50	33,4	25	20	16,6	14,3	12,5	11,1	10
t para	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
t seq	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
Id.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