Task 1: Theory (General) (12 Points)

Explain the following terms. Use **examples** to illustrate your explanations.

1. Race Condition
2. Deadlock

Task 2: Find the Bug (MPI) (15 Points)

The following MPI program is erroneous. Find the five bugs and describe them in short words. Additionally give for each bug a work-around.

Hints:

1. *You can assume all syntactical matters (C-Code, ordering of function parameters) as correct.*
2. *If there is an error that occurs at multiple lines of code, this is **only one** error.*
3. *There can be more than one error per line of code.*

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

int main(int argc, char** argv)
{
    int procRank, procCount, n;

    MPI_Initialize(&argc, &argv);

    procCount = MPI_Comm_size(MPI_WORLD);
    procRank = MPI_Comm_rank(MPI_WORLD);

    if(procRank == 0)
    {
        printf("Please enter an integer number\n");
        scanf("%d", &n);
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_WORLD);
    }
    else
    {
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_WORLD);
        printf("Number %d received\n", n);
    }

    MPI_Bcast(&n, 1, MPI_INT, -1, MPI_WORLD);

    return 0;

    MPI_Finalize();
}
```

Task 3: Sending and Receiving (MPI) (9 Points)

Modify the following program so that it uses non-blocking methods instead of blocking methods.

Hint: The behavior of the program does not need to stay exactly the same.

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

int main(int argc, char *argv[])
{
      int my_rank, size, i;
      MPI_Status status;

      MPI_Init(&argc, &argv);

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

      if (my_rank == 0)
      {
            int* msg = (int*) malloc(size * sizeof(int));

            for(i=1; i<size; i++)
            {
                  msg[i-1] = 4711 + i;
                  MPI_Send(&msg[i-1], 1, MPI_INT, i, 110, MPI_COMM_WORLD);
            }

            free(msg);
      }
      else
      {
            int msg;
            MPI_Recv(&msg, 1, MPI_INT, 0, 110, MPI_COMM_WORLD, &status);
            printf("Process %i out of %i: received msg=%d.\n",
            my_rank, size, msg);
      }

      MPI_Finalize();

      return 0;
}
```

Task 4: Theory (MPI) (12 Points)

a) Explain the difference between asynchronous (buffered) and synchronous communication.

b) Explain the principle of collective communication. Why is this approach useful?

c) Explain the difference between blocking and non-blocking communication.

Task 5: Sum of an Array (OpenMP) (15 Points)

Complete the following method, so that it computes the sum `s` over all elements in the array `a` in parallel.

a) Solve the assignment by using a `reduction` clause.

b) Solve the assignment **without** using a `reduction` clause.

Hints:

- *There are* `num` *elements in the array.*
- *The result should be stored in the variable* `long s`.
- *You possibly don't need to fill out all the blank lines, depending on your solution*

```
void sum_up(long* a, int num) {
     long s = 0;

     _____________________________________________

     _____________________________________________
     {

          _____________________________________________
          for(int i = 0; i < num; i++) {

               _____________________________________________
               {

                    _____________________________________________
               }
          }

          _____________________________________________

          _____________________________________________
     }

     printf("Sum: %ld\n", s );
}
```

```
void sum_up(long* a, int num) {
     long s = 0;

     _____________________________________________

     _____________________________________________
     {

          _____________________________________________
          for(int i = 0; i < num; i++) {

               _____________________________________________
               {

                    _____________________________________________
               }
          }

          _____________________________________________

          _____________________________________________
     }

     printf("Sum: %ld\n", s );
}
```

Task 6: Worksharing (OpenMP) (12 Points)

What possibilities exist in OpenMP to distribute the iterations of a for-loop over the worker threads? Name **three** of them. Give the particular keyword for the `schedule`-clause and explain what this keyword does.

Task 7: Java Streams (16 Points)

a) Write a function

```
public static Stream<String> getLetters()
```

that returns a stream with the letters A-Z, followed by the letters a-z.

b) Write a function

```
public static int[] getPositiveNumbers(int[] arr)
```

that returns an array with all the positive (>0) elements of arr. Write a parallel code and use streams for the parallelization.

c) Goldbach's conjecture is one of the oldest and best-known unsolved problems in number theory and all of mathematics. It states:

Every even integer greater than 2 can be expressed as the sum of two primes.

The following Java function determines, whether a given even integer > 2 can be expressed as the sum of two primes (turn the page). Parallelize the biggest performance hotspot using Java streams.

```
public static int[] checkGoldbachsConjecture(int z) {

	if ( z % 2 != 0 || z < 2 ) {
		throw new RuntimeException(z+ "must be even and >2");
	}

	//get prime numbers 2...z
	ArrayList<Integer> primes = new ArrayList<>();

	pLoop:
	for (int p=2; p<z; p++) {

		//is p prime?
		for (int i = 2; I <= Math.sqrt(p); i++) {
			if (p%i==0) {
				continue pLoop; // p is not prime
						// continue outer loop
			}
		}

		primes.add(p);
	}

	//check sums of two primes
	int left = 0;
	int right = primes.size() - 1;

	while ( left < right ) {

		int sum = primes.get(left) + primes.get(right);

		if (sum == z) {
			//found pair of two primes
			return new int[] {primes.get(left), primes.get(right)};
		} else if (sum < z) {
			left++;
		} else {
			right--;
		}
	}

	return null; //Goldbachs conjecture is disproved
}
```

Task 8: Theory (Java Threads) (9 Points)

a) Why is it better to use Thread Pools and Futures than simple Threads and Thread.join()?

b) Examine the following code and answer these questions:

1. The code is running in two threads, but it is actually not parallelized. Why?
2. Why can there be race conditions anyway?

```
public static void test(){
      int[] x={0};
      Thread n = new Thread(()->inc(x, 1000000));
      n.start();
      inc(x, 1000000);
      System.out.println(x[0]);
}

public static synchronized void inc(int[] x, int max) {
      for (int i=0; i<max; i++) {
            x[0]++;
      }
}
```

c) Give one advantage and one disadvantage of atomic datatypes in comparison to synchronized blocks.

Start operations

static <T> Stream<T>	concat(Stream<T> a, Stream<T> b)	Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
static <T> Stream<T>	generate(Supplier<T> s)	Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
static <T> Stream<T>	iterate(T seed, (T->T) f)	Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
static <T> Stream<T>	of(T... values)	Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T>	of(T t)	Returns a sequential Stream containing a single element.
static IntStream	range(int startInclusive, int endExclusive)	Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.
static IntStream	rangeClosed(int startInclusive, int endInclusive)	Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

Intermediate operations (Mappings)

DoubleStream LongStream	asDoubleStream() asLongStream()	Returns a Double/LongStream consisting of the elements of this stream, converted to double/long.
Stream<Integer>	boxed()	Returns a Stream consisting of the elements of this stream, each boxed to an Integer.
Stream<R>	map((T->R) mapper)	Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream IntStream LongStream	mapToDouble((T->double) mapper) mapToInt((T->int) mapper) mapToLong((T->long) mapper)	Returns a Double/Int/LongStream consisting of the results of applying the given function to the elements of this stream.

Intermediate Operations (Transformations)

Stream<T>	distinct()	Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
Stream<T>	filter((T->boolean) predicate)	Returns a stream consisting of the elements of this stream that match the given predicate.
Stream<T>	limit(long maxSize)	Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
S	parallel()	Returns an equivalent stream that is parallel
Stream<T>	peek((T->void) action)	Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
S	sequential()	Returns an equivalent stream that is sequential

Stream<T>	skip(long n)	Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted(((T,T)->int) comparator)	Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Stream<T>	sorted()	Returns a stream consisting of the elements of this stream, sorted according to natural order.

Terminal operations

boolean	allMatch((T->boolean) predicate)	Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch((T->boolean) predicate)	Returns whether any elements of this stream match the provided predicate.
OptionalDouble	average()	Returns an OptionalDouble describing the arithmetic mean of elements of this stream, or an empty optional if this stream is empty.
long	count()	Returns the count of elements in this stream.
Optional<T>	findFirst()	Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
void	forEach((T->void) action)	Performs an action for each element of this stream.
Optional<T>	max(((T,T)->int) comparator)	Returns the maximum element of this stream according to the provided Comparator.
OptionalInt OptionalDouble OptionalLong	max()	Returns an OptionalInt describing the maximum element of this stream, or an empty optional if this stream is empty.
Optional<T>	min((T,T)->int) comparator)	Returns the minimum element of this stream according to the provided Comparator.
OptionalInt OptionalDouble OptionalLong	min()	Returns an OptionalInt describing the minimum element of this stream, or an empty optional if this stream is empty.
boolean	noneMatch((T->boolean) predicate)	Returns whether no elements of this stream match the provided predicate.
Optional<T>	reduce(((T,T)->T) accumulator)	Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, ((T,T)->T) accumulator)	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
int / double / long	sum()	Returns the sum of elements in this stream.
Object[]	toArray()	Returns an array containing the elements of this stream.

MPI Functions

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm)
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm)
int MPI_Bsend(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm)


int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Probe(int source, int tag, MPI_Comm comm,
     MPI_Status *status)
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
     int *count)

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
     int root, MPI_Comm comm)
int MPI_Scatter(void *sendbuf, int sendcount,
     MPI_Datatype sendtype,void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
     MPI_Datatype *newtype)
int MPI_Type_vector(int count, int blocklength, int stride,
     MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
int MPI_Type_free(MPI_Ddatatype *datatype)

int MPI_Group_incl(MPI_Group group, int n, int *ranks,
     MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
```

```
     MPI_Group *newgroup)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
     MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color, int key,
     MPI_Comm *newcomm)
int MPI_Comm_free(MPI_Comm *comm)
```

OpenMP Directives

The **parallel** construct forms a team of threads and starts parallel execution.

#pragma omp parallel *[clause[[**,**]clause] ...] new-line*
{
//structured-block
}

clauses:
if(*scalar-expression***)**
num_threads(*integer-expression***)**
default(shared | **none)**
private(*list***)**
firstprivate(*list***)**
shared(*list***)**
copyin(*list***)**
reduction(*operator***:** *list***)**

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads. The most common form of the for loop is shown below:

```
for(var = lb; var relational-op b; var += incr)
{
        //do something
}
```

#pragma omp for *[clause[[**,**] clause] ...] new-line*
for-loops

clauses:
private(*list***)**
firstprivate(*list***)**
lastprivate(*list***)**
reduction(*operator***:** *list***)**
schedule(*kind[, chunk_size]***)**
collapse(*n***)**
ordered
nowait