Chair of Computer Architecture and Parallel Systems
Department of Informatics
Technical University of Munich

Note:
- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

Parallel Programming

Exam:	IN2147 / Endterm	**Date:**	Tuesday 24th July, 2018
Examiner:	Prof. Dr. Martin Schulz	**Time:**	16:00 – 17:30

	P 1	P 2	P 3	P 4	P 5	P 6	P 7	P 8	P 9	P 10	P 11
I											

Working instructions

- This exam consists of
  - **16 pages** with a total of **11 problems**.

  Please make sure now that you received a complete copy of the exam.
- The exam duration is exactly 90 minutes.
- The total amount of achievable credits in this exam is 90 credits.
- Detaching pages from the exam is prohibited.
- You are not allowed to bring your own scratch papers. There are extra pages available as scratch in the exam booklet. These pages are only graded if they are clearly referenced by corresponding (sub)problems.
- Neither write with red or green colors nor use pencils. Also avoid white out and ink eraser.
- Clearly cross out answers which you do not want to be graded.
- Subproblems marked by * can be solved without results of previous subproblems.
- This is a closed book exam. No additional resources or devices (calculators of any sort) are allowed.
- Physically turn off all electronic devices (cell phones, smartwatches, etc.) and leave them switched off at all times, put them in your bags, and close the bags. Setting them to silent / airplane mode is not sufficient.

Left room from _____ to _____ / Early submission at _____

Problem 1 **Amdahl's Law (5 credits)**

0 1

a)* What can be computed by applying Amdahl's Law?

0 1 2 3 4

b)* Assume you have a program, which is split into regions that either cannot be parallelized at all or that can be parallelized perfectly. Assuming the regions that cannot be parallelized consume 1% of execution time during a sequential run, what is the maximal speedup that can be achieved?

Problem 2 **Lock Granularity (4 credits)**

Lock granularity can have a significant impact on performance of shared memory codes.

0 1 2

a)* Name one possible reason for a negative performance impact in case lock granularity is chosen too fine:

0 1 2

b)* Name one possible reason for a negative performance impact in case lock granularity is chosen too coarse:

Problem 3 Shared memory parallelization using OpenMP (6 credits)

a)* Parallelize for loop in the following function using OpenMP by replacing **<OPENMP HERE>** with the appropriate OpenMP directive(s) and clause(s). Write your solution in the box below.

0 1 2

```
void function(float *a, float *b, int n)
{
    int i;

    <OPENMP HERE>
    for (i = 0; i < n - 1; i++)
        a[i] = (b[i] + b[i + 1]) / 2;

    a[i] = b[i] / 2;
}
```

b)* Parallelize for loop in the following function using OpenMP by replacing **<OPENMP HERE>** with the appropriate OpenMP directive(s) and clause(s). Write your solution in the box below.

0 1 2 3 4

```
double function(float *x, int *y, int n)
{
    int i, b = y[0];
    float a = 1.0;

    <OPENMP HERE>
    for (i=0; i < n; i++)
    {
        a += x[i];
        if (b < y[i])
            b = y[i];
    }

    return a * b;
}
```

Problem 4 OpenMP Variable Privatization (14 credits)

Consider the following program being executed with OMP_NUM_THREADS set to 16.

```
int value[64], k, t;

for (int i=0; i<64; i++)
        value[i]=63-i;
t=omp_get_thread_num();
k=42;

#pragma omp parallel for schedule(static, 2) CLAUSE
  for (int i=0; i<64; i++)
    { k=value[i]*2+t; }

printf("Final value of k=%i\n", k);
```

0 1 2

a)* Which iterations are executed by thread 13 in the parallel for loop?

0 1 2 3 4 5 6 7 8 9 10 11 12

b)* Fill in the following table. If multiple values are possible, state MULTIPLE.

CLAUSE set to:	Printed value of k
NONE	
private(k,t)	
shared(k,t)	
firstprivate(k,t)	
lastprivate(k,t)	
firstprivate(k,t) lastprivate(k,t)	

Problem 5 OpenMP Tasking (3 credits)

a)* List the OpenMP pragma used to specify a task:

0
1

b)* What is the difference between a `tied` and an `untied` task?

0
1
2

Problem 6 Performance issues of a two-socket node (6 credits)

a)* Name three potential performance drawbacks in the following code snippet if it is executed on a two-socket node.

0
1
2
3
4
5
6

```
void main()
{
    const int N = 1000000;
    int a[N];
    int b[N];

    for (int i = 0; i < N; i++)
    {
        a[i] = i;
    }

    #pragma omp parallel for schedule(static, 1)
    for (int i = 0; i < N; i++)
    {
        a[i] = a[i] + 17;
        b[i] = a[i] % 23;
    }
}
```

Problem 7 **Memory Consistency (4 credits)**

0 1 2 3 4

a)* What conditions must hold for a program to be sequentially consistent?

Problem 8 **Message Passing basics (5 credits)**

0 1 2 3

a)* What is the difference between an eager and a rendevouz protocol?

0 1 2

b)* Describe the functionality of `MPI_Waitany`.

This page is intentionally left blank. You can also use it as additional space for solutions. Clearly mark the (sub)problem your answers are related to and strike out invalid solutions.

Problem 9 MPI communicators and collective operations (20 credits)

Consider the following MPI program:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  int main(int argc, char **argv)
6  {
7      int value, temp;
8      int size, rank, row, col;
9      MPI_Comm c1, c2, c3;
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15     col = rank % 4;
16     row = rank / 4;
17
18     MPI_Comm_split(MPI_COMM_WORLD, row, col, &c1);
19     MPI_Comm_split(MPI_COMM_WORLD, col, row, &c2);
20     MPI_Comm_split(MPI_COMM_WORLD, rank, col, &c3);
21
22     MPI_Comm_rank(c1, &value);
23
24     MPI_Comm_rank(c2, &value);
25
26     MPI_Allreduce(&rank, &value, 1, MPI_INT, MPI_MAX, c3);
27     temp = value;
28
29     MPI_Allreduce(&temp, &value, 1, MPI_INT, MPI_SUM, c2);
30     temp = value;
31
32     MPI_Allreduce(&temp, &value, 1, MPI_INT, MPI_MIN, c1);
33
34     MPI_Finalize();
35 }
```

0 1 2

a)* Describe the functionality of `MPI_Comm_split()`.

b)* Assuming the program is executed with 12 MPI processes, sketch the communicators `MPI_COMM_WORLD`, `c1`, `c2` and `c3` by enclosing circles around the MPI processes (represented by the boxes with their respective ranks in `MPI_COMM_WORLD`) that are in the same communicator.

0 1 2 3 4 5 6 7 8

`MPI_COMM_WORLD`

00	01	02	03
04	05	06	07
08	09	10	11

`c1`

00	01	02	03
04	05	06	07
08	09	10	11

`c2`

00	01	02	03
04	05	06	07
08	09	10	11

`c3`

00	01	02	03
04	05	06	07
08	09	10	11

extra

00	01	02	03
04	05	06	07
08	09	10	11

extra

00	01	02	03
04	05	06	07
08	09	10	11

c) Assuming the program is executed with 12 MPI processes, complete the table below according to the content of variable `value` at specific points in program in each rank:

0 1 2 3 4 5 6 7 8 9 10

Line number	rank	value
23	6	
25	7	
27	8	
30	9	
33	10	

Problem 10 **Blocking and non-bocking collective MPI (7 credits)**

Consider the following MPI program:

```
1  // function to initialize two  of size n
2  void init(float*x, float* y, int n){...}
3
4  // very expensive function which only works on elements of input array x
5  void costly_calculation(float* x, int n){...}
6
7  int main(int argc, char *argv[])
8  {
9      int rank, size;
10     MPI_Init(&argc, &argv);
11     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14     float *a = (float *)malloc(size * sizeof(float));
15     float *b = (float *)malloc(size * sizeof(float));
16     init(a, b, size);
17     float data = rank;
18
19     // point to point communication
20     if (rank == 0)
21         for (int r = 1; r < size; r++)
22         {
23             a[0] = data;
24             MPI_Recv(a + r, 1, MPI_FLOAT, r, 0, MPI_COMM_WORLD,
25                      MPI_STATUS_IGNORE);
26         }
27     else
28         MPI_Send(&data, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
29
30     costly_calculation(b, size);
31
32     costly_calculation(a, size);
33
34     free(a); free(b);
35     MPI_Finalize();
36     return 0;
37 }
```

0 1 2 3

a)* List the blocking collective operation with its parameters that can be used to replace the point to point communication between lines 20 to 28.

b)* List the non-blocking collective operation with its parameters that can be used to replace the blocking communication from previous subproblem (a).

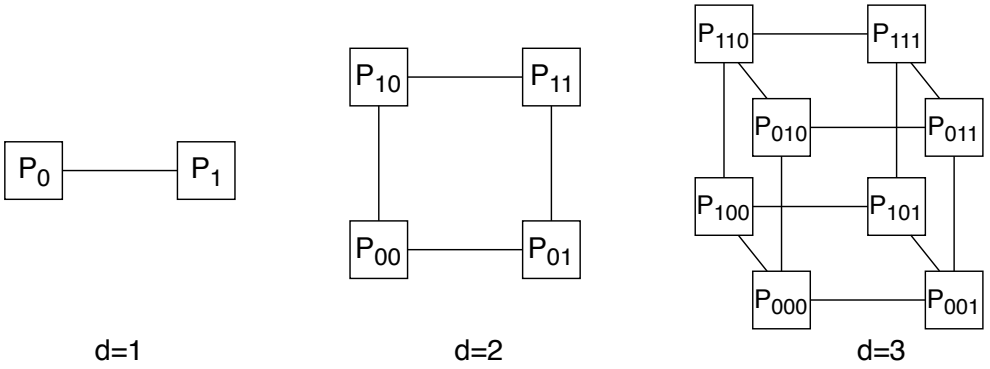
0
1
2

c)* Name the appropriate operation when using a non-blocking collective to insure its completion and specify the proper line number to put this operation for getting the best performance.

0
1
2

Problem 11 Efficient MPI reduction with virtual hypercube topology (16 credits)

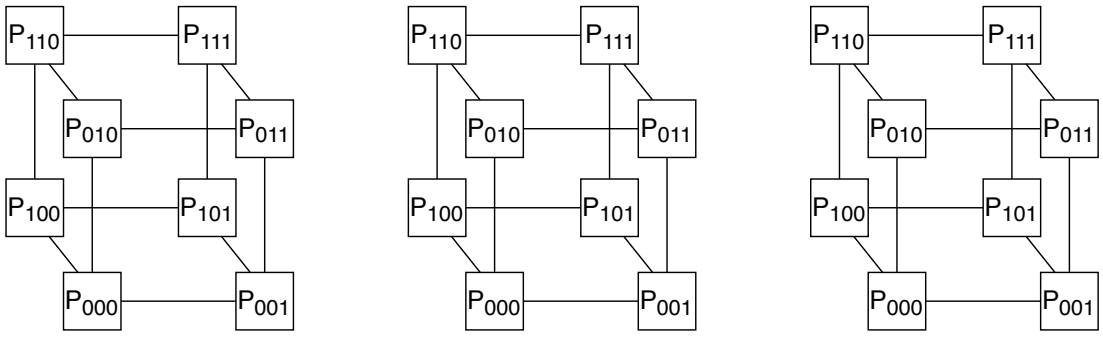
A hypercube of dimension **d** consists of $\mathbf{p} = \mathbf{2^d}$ nodes; there exists a direct connection between two nodes $\mathbf{n_1}$ and $\mathbf{n_2}$ if (and only if) the binary representation of $\mathbf{n_1}$ and $\mathbf{n_2}$ differs by exactly one bit.



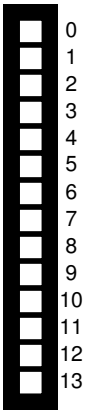
In this task we want to implement an efficient summation reduction across all MPI processes in a communicator optimized for the hypercube topology using point-to-point MPI operations, where MPI process with rank **n** is mapped to node **n** in the hypercube. All ranks are with respect to the communicator **c**. For simplification, we restrict ourselves to the case where rank 0 is the root of the reduction and the size of **c** is a power of **2** (i.e., the size is $\mathbf{2^d}$). Under the assumption that each point-to-point operation takes one timestep, the overall reduction across all MPI processes in **c** should be performed in at most **d** timesteps and communication is only allowed with direct neighbors in the virtual hypercube topology.

0 1 2 3

a)* Sketch the three communication timesteps of the reduction using hypercube topology for $\mathbf{p} = \mathbf{8}$ processes by adding arrows to the edges according to the communication direction.



b) Implement the efficient summation reduction of a **float** buffer according to your communication schema developed in subproblem (a) in MPI process with **rank** = **0** using a virtual hypercube topology of dimension **d** and blocking point-to-point MPI operations.




```
/**
 * @brief efficient summation reduction in rank zero using
 *        virtual hypercube topology
 *
 * @param d dimension of hypercube
 * @param rank rank of MPI process within communicator c
 * @param p number of MPI processes in communicator c
 * @param buffer communication buffer
 * @param c communicator
 * @return float reduced value
 */

float hc_zero_reduce(int d, int rank, int p, float buffer, MPI_Comm c)
{
    if ( p != 1 << d ) {
        MPI_Abort(c, -1);
    }









}
```

Additional space for solutions–clearly mark the (sub)problem your answers are related to and strike out invalid solutions.