# Lösungen zu Übungsblatt 6 (Parnum)

**Aufgabe 20**

siehe Kopie und Listing

**Aufgabe 21**

Beschreibung des Ablaufs:

1. Verteile Block $A_{ii}$ in Zeile $i$ (Broadcast)
2. Berechne $C_{ij} = A_{ii} \cdot B_{ij}$
3. Iteriere $k = 1, \ldots, m-1$
   (a) Blöcke in den Spalten zyklisch an den nächsten Prozessor verteilen, d.h. Prozessor $t_{i,j}$ schickt seine Matrix $B$ an Prozessor $t_{i-1,j}$
   (b) Verteile Matrix $A_{i,i+k \; mod \, m}$ in Zeile $i$
   (c) Berechne $C_{ij} += A_{i,i+k \; mod \, m} \cdot B_{i+k \; mod \, m}$

nach $m$ Schritten beinhaltet jeder Prozessor den Block $C_{ij}$

C-PVM Programm

```
/* Matrix Multiply */

/* defines and prototypes for the PVM library */
#include <pvm3.h>
#include <stdio.h>

/* Maximum number of children this program will spawn */
#define MAXNTIDS 100
#define MAXROW 10

/* Message tags */
#define ATAG 2
#define BTAG 3
#define DIMTAG 5

/* Initialize the matrix blocks with random numbers */
void InitBlock(float *a, float *b, float *c, int blk, int row, int col)
{
```

```
    int len, ind;
     int i,j;

    srand(pvm_mytid()); /* set seed of srand */
    len = blk*blk; /* number of elements in submatrix */
    for (ind = 0; ind < len; ind++)
    {
        a[ind] = (float)(rand()%1000)/100.0;
        c[ind] = 0.0;
    }
    for (i = 0; i < blk; i++)
    {
        for (j = 0; j < blk; j++)
        {
            if (row == col)
                b[j*blk+i] = (i==j)? 1.0 : 0.0;
            else
                b[j*blk+i] = 0.0;
        }
    }
}

/* Multiplication of two submatrices */
void BlockMult(float* c, float* a, float* b, int blk)
{
    int i,j,k;

    for (i = 0; i < blk; i++)
        for (j = 0; j < blk; j ++)
            for (k = 0; k < blk; k++)
                c[i*blk+j] += (a[i*blk+k] * b[k*blk+j]);
}

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

```
 /* number of tasks to spawn, use 3 as the default */
 int ntask = 2;
 /* return code from pvm calls */
 int info;
 /* my task and group id */
 int mytid, mygid;
 /* children task id array */
 int child[MAXNTIDS-1];
 int i, m, blksize;
/* array of the tids in my row */
int myrow[MAXROW];
/* data structure for matrices */
float *a, *b, *c, *atmp;
int row, col, up, down;

/* find out my task id number */
mytid = pvm_mytid();
/* check for error */
if (mytid < 0)
    /* exit the program */
    return -1;
/* join the mmult group */
mygid = pvm_joingroup("mmult");
if (mygid < 0)
    /* exit the program */
    return -1;

 /* if my group id is 0 then I must spawn the other tasks */
if (mygid == 0)
{
     /* find out how many tasks to spawn */
    if (argc >= 3)
    {
        m = atoi(argv[1]);
        blksize = atoi(argv[2]);
```

```
    }
    else
    {
        fprintf(stderr, "usage: mmult m blk\n");
        pvm_lvgroup("mmult");
        pvm_exit();
        return -1;
    }
    /* make sure ntask is legal */
    ntask = m*m;
    if ((ntask < 1) || (ntask >= MAXNTIDS))
    {
        pvm_lvgroup("mmult");
        pvm_exit();
        return -1;
    }
     /* spawn the child tasks */
    info = pvm_spawn("mmult", (char**)0, PvmTaskDefault, (char*)0, ntask-1, child);
    /* make sure spawn succeeded */
    if (info != ntask-1)
    {
        pvm_lvgroup("mmult");
        pvm_exit();
        return -1;
    }
    /* send the matrix dimension */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&m, 1, 1);
    pvm_pkint(&blksize, 1, 1);
    pvm_mcast(child, ntask-1, DIMTAG);
}
else
{
```

```
    /* recv the matrix dimension */
    pvm_recv(pvm_gettid("mmult", 0), DIMTAG);
    pvm_upkint(&m, 1, 1);
    pvm_upkint(&blksize, 1, 1);
    ntask = m*m;
}
/* make sure all tasks have joined the group */
info = pvm_barrier("mmult",ntask);
if (info < 0)
    pvm_perror(argv[0]);
/* find the tids in my row */
for (i = 0; i < m; i++)
    myrow[i] = pvm_gettid("mmult", (mygid/m)*m + i);
/* allocate the memory for the local blocks */
a = (float*)malloc(sizeof(float)*blksize*blksize);
b = (float*)malloc(sizeof(float)*blksize*blksize);
c = (float*)malloc(sizeof(float)*blksize*blksize);
atmp = (float*)malloc(sizeof(float)*blksize*blksize);
/* check for valid pointers */
if (!(a && b && c && atmp))
{
    fprintf(stderr, "%s: out of memory!\n", argv[0]);
    free(a); free(b); free(c); free(atmp);
    pvm_lvgroup("mmult");
    pvm_exit();
    return -1;
}
/* find my block's row and column */
row = mygid/m;
col = mygid % m;
/* calculate the neighbor's above and below */
up = pvm_gettid("mmult", ((row)?(row-1):(m-1))*m+col);
```

```
down = pvm_gettid("mmult", ((row == (m-1))?col:(row+1)*m+col));

/* initialize the blocks */
InitBlock(a, b, c, blksize, row, col);

/* do the matrix multiply */
for (i = 0; i < m; i++)
{
    /* mcast the block of matrix A */
    if (col == (row + i)%m)
    {
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(a, blksize*blksize, 1);
        pvm_mcast(myrow, m, (i+1)*ATAG);
        BlockMult(c,a,b,blksize);
    }
    else
    {
        pvm_recv(pvm_gettid("mmult", row*m + (row +i)%m), (i+1)*ATAG);         pvm_upkfloat(atmp,
blksize*blksize, 1);
        BlockMult(c,atmp,b,blksize);
    }
    /* rotate the columns of B */
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(b, blksize*blksize, 1);
    pvm_send(up, (i+1)*BTAG);
    pvm_recv(down, (i+1)*BTAG);
    pvm_upkfloat(b, blksize*blksize, 1);
}
printf("Done.\n");
free(a);
free(b);
free(c);
free(atmp);
pvm_lvgroup("mmult");
```

```
    pvm_exit();
    return 0;
}
```

**Aufgabe 22**

Beschreibung des Ablaufs in beiden Varianten siehe Vorlesung.

C-MPI Programm für Variante 1:

```
/* Matrix-Vektor-Multiplikation - Matrix zeilenblockweise verteilen -
==> Vektor broadcasten - lokale Teilergebnisse berechnen -
==> Teilergebnisse aufsammeln: MPI_Gatherv */
#include<stdio.h>
#include<mpi.h>

main(int argc, char *argv[])
{
    int I,P;
    float *A,*x,*b; /* nxn-Matrix A, n-Vektor x, n-Vektor b */
    float *A0,*b0; /* vollstaendige ",-," fuer Proz. 0 */
    int n; /* Seitengroesse von A */
    int i,j; /* Zeilen-/Spaltenindizes */
    int uzeile, ozeile; /* Arbeitsgrenzen fuer Proz. I */
    int index; /* Arrayindex */
    int *sendcountsA, *displsA; /* fuer MPI_Scatterv ... */
    int *recvcountsb, *displsb; /* fuer MPI_Gatherv ... */
    int this, next; /* Grenzen der Proz. durchwandern */
    float sum; /* Hilfsvar. */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &I);
    MPI_Comm_size(MPI_COMM_WORLD, &P);

    /* Matrizengroesse: */
     n=7;

    /* Speicher reservieren: */
    uzeile=((int)((float)(I*n)/(float)P+0.5));
```

```
ozeile=((int)((float)((I+1)*n)/(float)P+0.5));
if(I==0)
{
    /* komplett fuer 0 */
    if((b0=(float *) malloc((n+n*n)*sizeof(float)))==NULL)
    {
        puts("0: malloc b0 failed\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    A0=b0+n;
    /* sendcounts und displs initialisieren: */
    if((sendcountsA=(int *)malloc(4*P*sizeof(int)))==NULL)
    {
        puts("0: malloc sendcountsA failed\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    displsA = sendcountsA + P;
    recvcountsb = displsA + P;
    displsb = recvcountsb + P;
    this=0;
    for(index=0;index<P;index++)
    {
        displsA[index] = n * (displsb[index]=this);
        next=((int)((((float)((index+1)*n))/((float)P))+0.5));
        sendcountsA[index] = n * (recvcountsb[index]=next-this);
        this=next;
    }
}
if((x=(float *)malloc((n+(n+1)*(ozeile-uzeile))*sizeof(float)) )==NULL)
{
    printf("%d: malloc x failed\n",I);
    MPI_Abort(MPI_COMM_WORLD,1);
}
```

```
b=x+n; A=b+ozeile-uzeile;

/* Testinitialisierung: */
if(I==0)
{
   index=0;
   for(i=0;i<n;i++)
   {
      for(j=0;j<n;j++)
      {
         A0[index]=(float)(i*n+j+1);
         index++;
      }
      x[i]=i+1;
   }
}

/* A verteilen: */
MPI_Scatterv(A0, sendcountsA, displsA, MPI_FLOAT, A , (ozeile-uzeile)*n ,
                                   MPI_FLOAT, 0 , MPI_COMM_WORLD);
/* x broadcasten: */ MPI_Bcast(x,n,MPI_FLOAT,0,MPI_COMM_WORLD);

/* Teilergebnisse berechnen: */
index=0;
for(i=uzeile; i<ozeile; i++)
{
   sum=0.0;
   for(j=0; j<n; j++)
   {
      sum+=A[index]*x[j];
      index++;
   }
   b[i-uzeile]=sum;
}

/* Ergebnisse zusammensetzen: */
```

```
    MPI_Gatherv(b , ozeile-uzeile , MPI_FLOAT, b0, recvcountsb, displsb, MPI_FLOAT, 0,
MPI_COMM_WORLD);

    /* Ausgabe bei Proz. 0: */
    if(I==0)
    {
        for(i=0; i<n; i++)
        {
            printf(" %4.0f",b0[i]);
        }
        putchar('\n');
    }
    free(x);
    if(I==0)
    {
        free(sendcountsA);
        free(b0);
    }
    MPI_Finalize();
}
```

C-MPI Programm für Variante 2:

```
/* Matrix-Vektor-Multiplikation - Matrix spaltenblockweise verteilen -
==> Vektor auch scattern - lokale Teilergebnisse berechnen -
==> Teilergebnisse bei Proc. 0 aufsummieren: MPI_Reduce */

#include<stdio.h>
#include<mpi.h>
#define FIRST(i,p,n) ((int)((((float)((i)*(n)))/((float)(p)))+0.5))

main(int argc, char *argv[])
{
    int I,P;
    int n; /* Seitengroesse von A */
    int i,j; /* Zeilen-/Spaltenindizes */
    int uzeile, ozeile; /* Arbeitsgrenzen fuer Proz. I */
```

```
int my_size; /* ozeile - uzeile */
int index; /* Arrayindex */
float *A; /* n*my_size Matrix A */
float *x,*b; /* my_size-Vektor x, n-Vektor b */
float *A0, *x0, *b0; /* n*n A0, n x0, n b0 fuer Proz. 0 */
float *Ap, *xp, *sp; /* Hilfspointer */
int *sendcounts, *displs; /* fuer MPI_Scatterv ... */
float *scatter_buffer; int this, next; /* Grenzen der Proz. durchwandern */
float sum; /* Hilfsvar. */

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &I);
MPI_Comm_size(MPI_COMM_WORLD, &P);

n=7; /* Problemgroesse */
uzeile=FIRST(I,P,n);
ozeile=FIRST(I+1,P,n);
my_size=ozeile-uzeile;
if(I==0)
{
    /* Matrix A0 und Vektor x0 allokieren: */
    if((x0=(float *)malloc((n+1)*n*sizeof(float)))==NULL)
    {
        puts("0: malloc x0 failed\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    A0=x0+n;
    /* Matrix A0 und Vektor x0 initialisieren: */
    xp=x0;
    Ap=A0;
    index=1;
    for(i=0;i<n;i++)
    {
        *xp=i+1;
        xp++;
```

```
            for(j=0;j<n;j++)
            {
                *Ap=index;
                Ap++;
                index++;
            }
        }
/* Scatter von A0 und x0 vorbereiten: */
        if((scatter_buffer=(float *)malloc((n+1)*n*sizeof(float)))==NULL)
        {
            puts("0: malloc scatter failed\n");
            MPI_Abort(MPI_COMM_WORLD,1);
        }
        /* sendcounts, displs und scatter_buffer: */
        if((sendcounts=(int *)malloc(2*P*sizeof(int)))==NULL)
            {
                puts("0: malloc sendcounts failed\n");
                MPI_Abort(MPI_COMM_WORLD,1);
            }
            displs = sendcounts + P;
            sp=scatter_buffer;
            xp=x0;
            this=FIRST(0,P,n);
            for(index=0;index<P;index++)
            {
                displs[index] = this*(n+1);
                next=FIRST(index+1,P,n);
                sendcounts[index] = (next-this)*(n+1);
                for(i=this;i<next;i++)
                {
                    *sp=*xp; sp++; xp++;
                }
                Ap=A0+this;
```

```
            for(i=0; i<n; i++)
            {
                for(j=this; j<next; j++)
                {
                    *sp=*Ap; sp++; Ap++;
                }
                Ap=Ap-next+this+n;
            }
            this=next;
        }
    }
    /* verteilten Speicher reservieren fuer b, b0, A(I) und x(I): */
    if((b0=(float *)malloc((2*n+(n+1)*my_size)*sizeof(float)))==NULL)
    {
        puts("0: malloc b0 failed\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    b=b0+n; x=b+n;
    A=x+my_size;
    MPI_Scatterv(scatter_buffer, sendcounts,displs, MPI_FLOAT, x , my_size*(n+1) , MPI_FLOAT,
0, MPI_COMM_WORLD);
    /* lokale Berechnungen: */
    Ap=A;
    sp=b;
    for(i=0;i<n;i++)
    {
        sum=0.0;
        xp=x;
        for(j=uzeile; j<ozeile; j++)
        {
            sum+=*Ap * *xp;
            Ap++; xp++;
        }
```

```
            *sp=sum; sp++;
        }
        /* Teilergebnisse akkumulieren: */
        MPI_Reduce(b,b0,n,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
        /* Ausgabe bei Proz. 0: */
        if(I==0) { for(i=0; i<n; i++)
        {
            printf(" %4.0f",b0[i]);
        }
        putchar('\n');
    }
    free(b0);
    if(I==0)
    {
        free(sendcounts);
        free(scatter_buffer);
        free(x0);
    }
    MPI_Finalize();
}
```