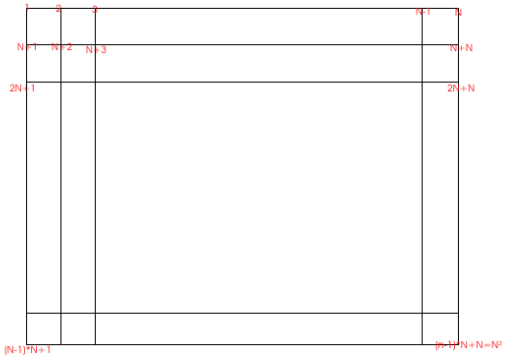
Verena Horak
1st Homework
HPSC 2004

Exercise 2: Write a parallel algorithm for a two-dimensional finite difference problem in which the value of each point in a two-dimensional grid of size $N \times N$is updated as the average of the current point value and of the four nearest neighbouring point values. Study the performance of the resulting program as a function of problem size and processor count, assuming on task per processor.

Extend the algorithm and program developed to incorporate a simple convergence test: terminate execution when the difference between values computed at successive steps is less than a specified threshold for all grid points. Study the impact of communication.

Solution: We will number the nodes in the $N \times N$ grid with linear numbers, starting in the left upper corner and proceeding row by row, numbering each row always from left to right.



The values for the nodes at the grid's margin are given by boundary conditions and will therefore remain static (unchanged) during the computation.
We will determine the number of neighbouring nodes for each grid node by integer arithmetic (referring to the numbering scheme explained above). Calculations only have to be done for interior grid points; each of these nodes has four neighbours, so we don't need to handle any special cases here.

The algorithm may look like this:

```
int n = ...; //number of rows/columns of NxN-grid
int[n] boundcond = ...; //array of boundary values or starting
                          values for all nodes
int rank = get_processor_id(); //number of current process
                                 (which is equal to the number
```

```
                                    of the grid point for which this
                                    processor is supposed to compute
                                    values)
**boolean endop = false;
**int thresh; //specified threshold (termination criterium)
int newval, **oldval;
int n1val, n2val, n3val, n4val; //values of neighbouring grid
                                     points
newval = boundcont[rank]; //initialize newval with starting
                              value
oldval = newval;
**while(no message has arrived){
**//message at this point means operation has to be finished
    if (rank>=n && rank<=(n-1)*n && (rank %n)!=0 && (rank %n)!=1){
                //if rank is number of interior grid point
        SENDRECV(newval, n1val, rank-1);
        //send and receive value from left neighbour
        SENDRECV(newval, n2val, rank+1);
        SENDRECV(newval, n3val, rank-n);
        SENDRECV(newval, n4val, rank+n);
        **oldval = newval;
        newval = (newval + n1val + n2val + n3val + n4val)/5;
    }
    elsif (rank>1 && rank<n)
        {SENDRECV(newval, n4val, rank+n);}
    elsif (rank>1 && rank<(n-1)*n+1 && (rank %n)==1)
        {SENDRECV(newval, n2val, rank+1);}
    elsif (rank>n && rank<n*n && (rank %n)==0)
        {SENDRECV(newval, n1val, rank-1);}
    elsif (rank>(n-1)*n && rank<n*n)
        {SENDRECV(newval, n3val, rank-n);}
    **REDUCE(abs(newval-oldval)<thresh,endop,&&,1);
    **//somthing sent from process number 1. here is signal to
        terminate all operations
    **if (rank==1 && endop)
        **{BCAST(any_data);}
}
```

This algorithm is - evidently - written in pseudo-code, but it may be implemented using C and MPI with only minor syntactical adaptations.

Lines starting with two stars (**) are extensions implementing a simple convergence test as specified in the problem description.

Performance of resulting programm:
With every loop, there has to be done one floating point operation vor every inner grid point, yielding $n^2 - 2n - 2(n-1)$ operations. One task per processor

means we need to have n^2 processors for n^2 grid points.

Impact of communication:
With every loop, there have to be handled

$$\underbrace{4 \cdot (n-2)^2}_{interior\ points} + \underbrace{1 \cdot 4 \cdot (n-2)}_{margin\ points}$$

SENDRECV-operations.

Conclusion:
Performance as well as operations needed for communication are quadratic dependent on the problem size.
If we can assume one task per processor we might suppose that execution time is nearly constant for arbitrary problem size n as each processor always carries a constant load of work and communication is only needed with at most 4 neighbouring processes.