Parallel Computation 2

Hartmut Kaiser (hkaiser@cct.lsu.edu)

Cooperative Computing

“Cooperative” computing

- Between Capacity and Capability computing
  - Not a widely used term
  - But an important distinction with respect to these others
- Synonymous with “Coordinated” computing
- Single application
  - Partitioning of data into quasi independent blocks
  - Semi independent processes operate on separate data blocks
  - Limited communication of messages
    - *Coordinate through remote synchronization*
    - *Cooperate through the exchange of some data*
- Scaling
  - Primarily weak scaling
  - Limited strong scaling
- Programming
  - Favors SPMD (Single Program stream Multiple Data stream) style
  - Static scheduling mostly by hand
  - Load balancing by hand
  - Coarse grain
    - *Process*
    - *Data*
    - *Communication*

Data Decomposition

Partitioning the global data into major contiguous blocks

Exploits *spatial locality* that assumes the use of a data element heightens the likelihood of nearby data being used as well (reducing latencies associated with cache misses followed by accesses to main memory)

Exploits *temporal locality* that assumes the use of a data element heightens the likelihood that the same data will be reused again in the near future

Varies in form
- In dimensionality
- Granularity (size)
- Shape of partitions

Static mapping of partitions on to processor nodes

Distributed Concurrent Processes

Each data block can be processed at the same time
- Parallelism is determined by number of processes
- More blocks with smaller partitions permit more processes
- But …

Processes run on separate processors on local data
- Usually one application process per processor
- Usually SPMD i.e., processes are equivalent but separate (same code, different environments)

Execution of inner data elements of the partition block are done independently for each of the processes
- Provides coarse grain parallelism
- Outer loop iterates over successive application steps over the same local data

Data Exchange

In shared memory, no problem, all the data is there

For distributed memory systems, data needs to be exchanged between separate nodes and processes

Ghost cells used to hold local copies of edges of remote partition data at remote processor sites

Communication packets are medium to coarse grain and point to point for most data transfers

- e.g., all edge cells of one data partition may be sent to corresponding ghost cells of the neighboring processor in a single message

Multi-cast or broadcast may be required for some application algorithms and data partitions

- e.g., matrix-vector multiply

Synchronize

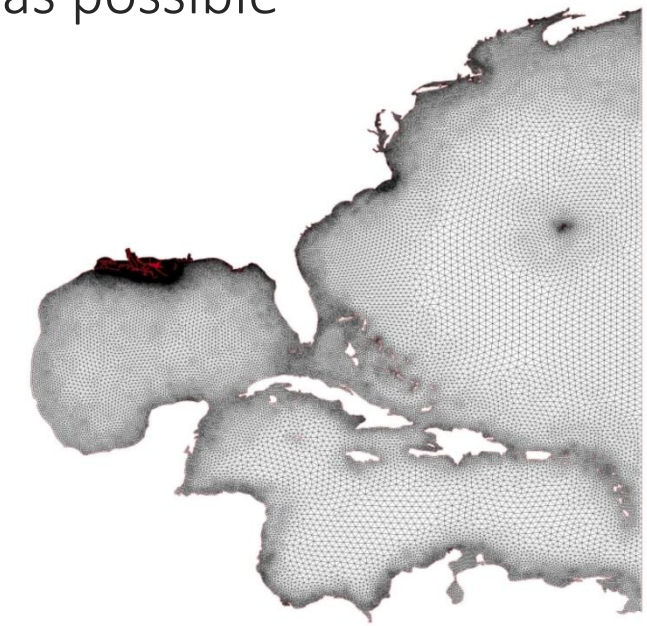
- Global barriers
  - Coarse grain (in time) control of outer-loop steps
  - Usually used to coordinate transition from computation phase to communication phase
- Send/receive
  - Medium grain (in time) control of inner-loop data exchanges
  - Blocks on a send and receive
  - Computation at sender proceeds when data has been received
  - Computation at receiver proceeds when incoming data is available
  - Non-blocking versions of each exist but can lead to race conditions

Making Parallelism Fit

Different kinds of parallelism work best on certain kinds architectures

Need to satisfy two contending requirements:

- Spread work out among as many parallel elements as possible
- Minimize inefficiencies due to:
  - *Starvation*
  - *Overhead*
  - *Latencies*
  - *Waiting for Contention resolution*



Communicating Sequential Processes

A model of parallel computing
- Developed in the 1970s
- Often attributed to Tony Hoare
- Satisfies criteria for cooperative computing
  - *Many would claim it as a means of capability computing*

Process Oriented

Emphasizes data locality

Message passing semantics

Synchronization using barriers among others

Distributed reduction operators added for purposes of optimization

Communicating Sequential Processes

Another form of parallelism

Coarse grained parallelism
- Large pieces of sequential code
- They run at the same time

Good for clusters and distributed memory MPPs

Share data by message passing
- Often referred to as “message-passing model”

Synchronize by “global barriers”

Most widely used method for programming

MPI is dominant API

Supports “SPMD” strategy (Single Program Multiple Data)

CSP Processes

Process is the body of state and work

Process is the module of work distribution

Processes are static
- In space: assigned to a single processor
- In time: exist for the lifetime of the job

All data is either local to the process or acquired through incident messages

Possible to extend process beyond sequential to encompass multiple threaded processes
- Hybrid model integrates the two models together in a clumsy programming methodology

Locality of state

Processes operate on memory within the processor node

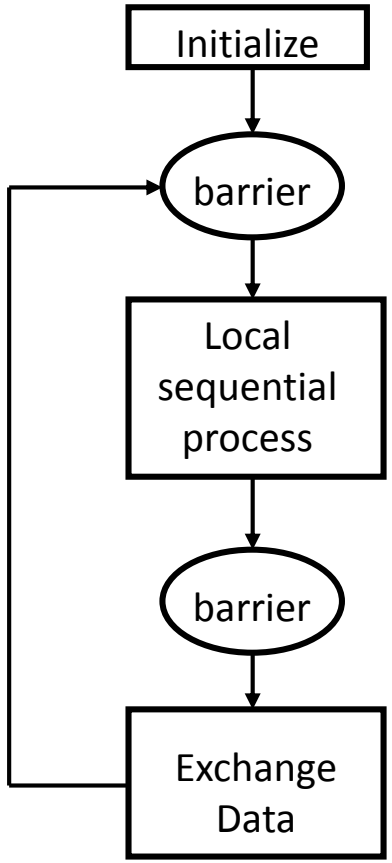
Granularity of process iteration dependent on the amount of process data stored on processor node

New data from beyond local processor node acquired through message passing, primarily by send/receive semantic constructs

Other Key Functionalities

- Synchronization
  - Barriers
  - Messaging
- Reduction
  - Mix of local and global
- Load balancing
  - Static, user defined

Message Passing Model (BSP)



Global Barrier Synchronization

Different nodes finish a piece of work at different times

Cannot exchange data until all work has completed

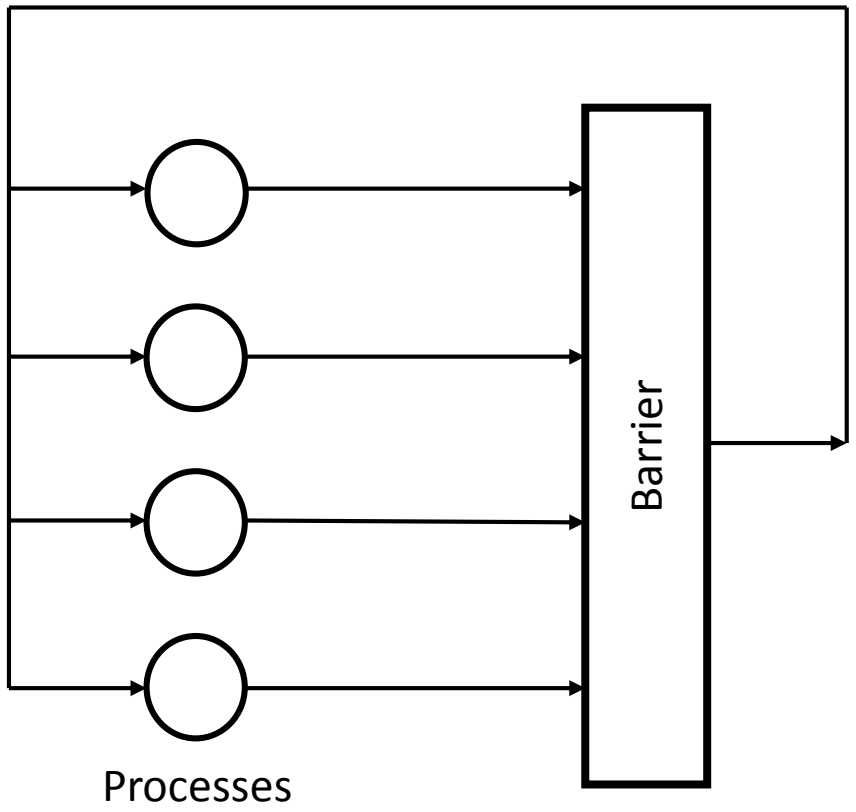
Barriers synchronize all concurrent processes running on separate “nodes”

How it works

- Every process “tells” barrier when it is done
- When all processes are done, barrier “tells” processes that they can continue

“tells” is done by message passing over the network

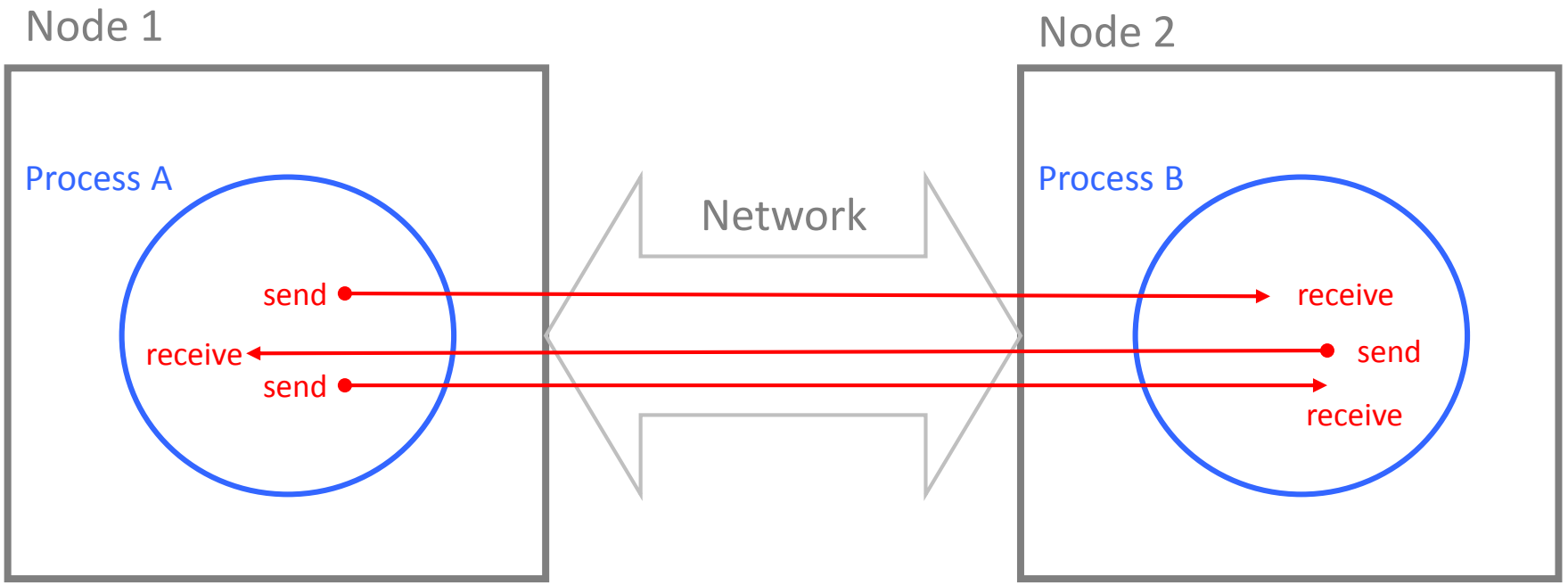
Barrier Synchronization



Message: Send & Receive

- Nodes communicate with each other by packets through the system area network
  - Reminder: network comprises (hardware)
    - *NICs (Network Interface Controller)*
    - *Links (metal wires or fiber optics)*
    - *Switch (N x N)*
  - Operating systems and network drivers (software)
- Processes communicate with each other by application-level messages
  - send
  - receive
- Message content
  - Process port
  - Data

Send & Receive



An Example Problem

Partial Differential Equation (PDE)

- Heat equation
- 2-dimensions discrete point distribution mesh to approximate a unit square
  - *The temperature field is approximated by a finite set of discrete points distributed over the computational domain and the temperature values need to be calculated in this points*
- Static boundary conditions (temperature on the boundaries is predefined function of time)

Stages of Code Development

- Data decomposition
- Concurrent sequential processes
- Coordination through synchronization
- Data exchange

An Example Problem

Heat equation:

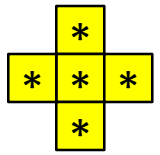
In 2-D:

$$\frac{\partial u}{\partial t} = k\nabla^2 u$$

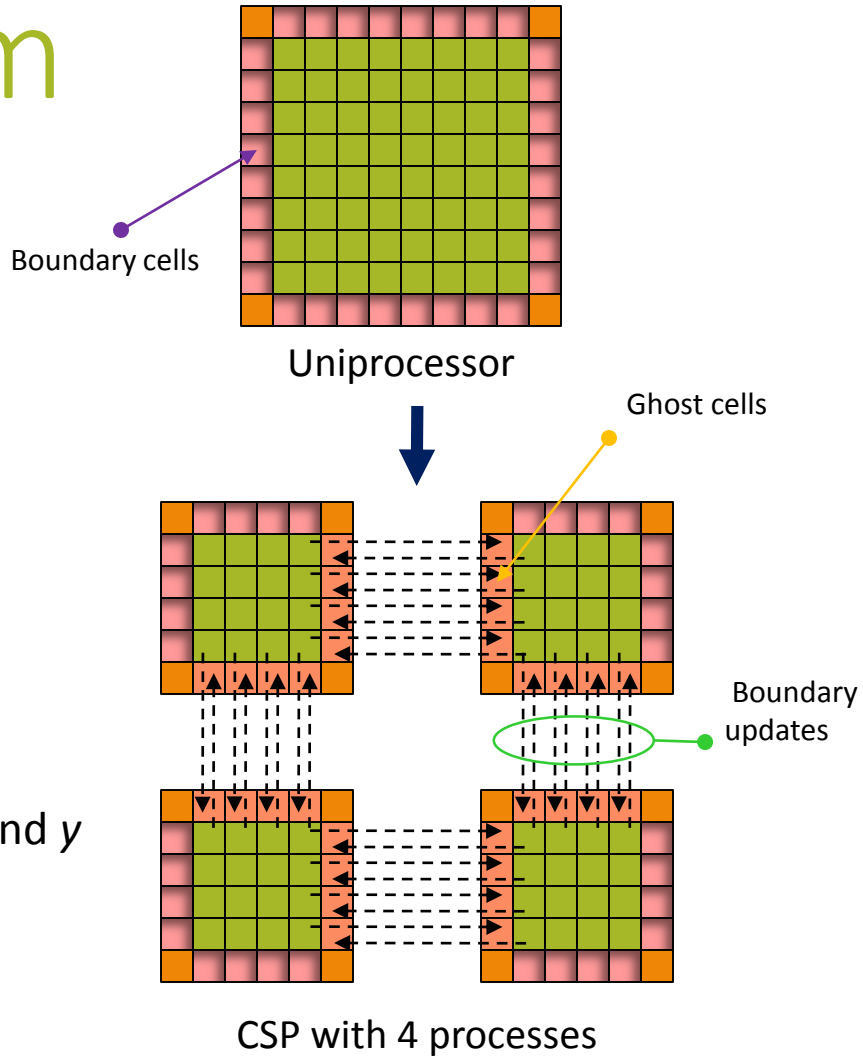
$$u_t = k(u_{xx} + u_{yy})$$

Implementation:

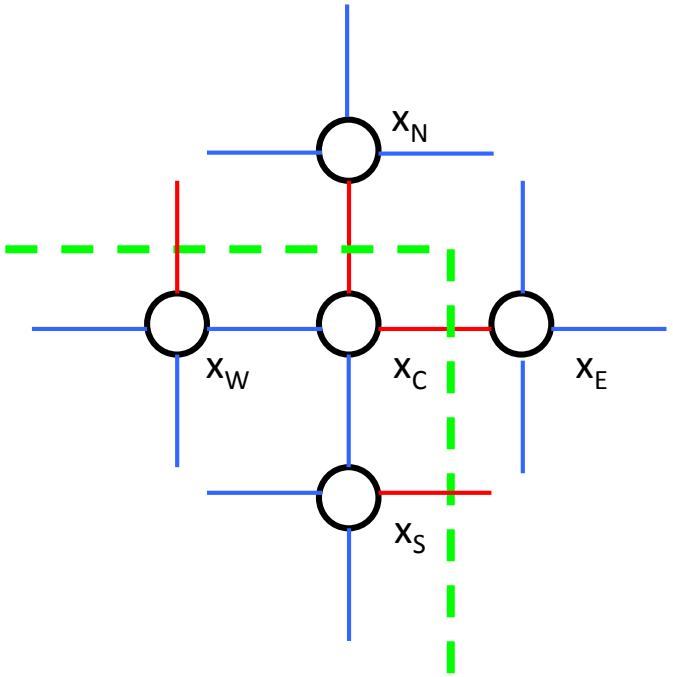
- Jacobi method on a unit square
- Dirichlet boundary condition
- Equal number of intervals along *x* and *y* axis



stencil



Stencil Calculation



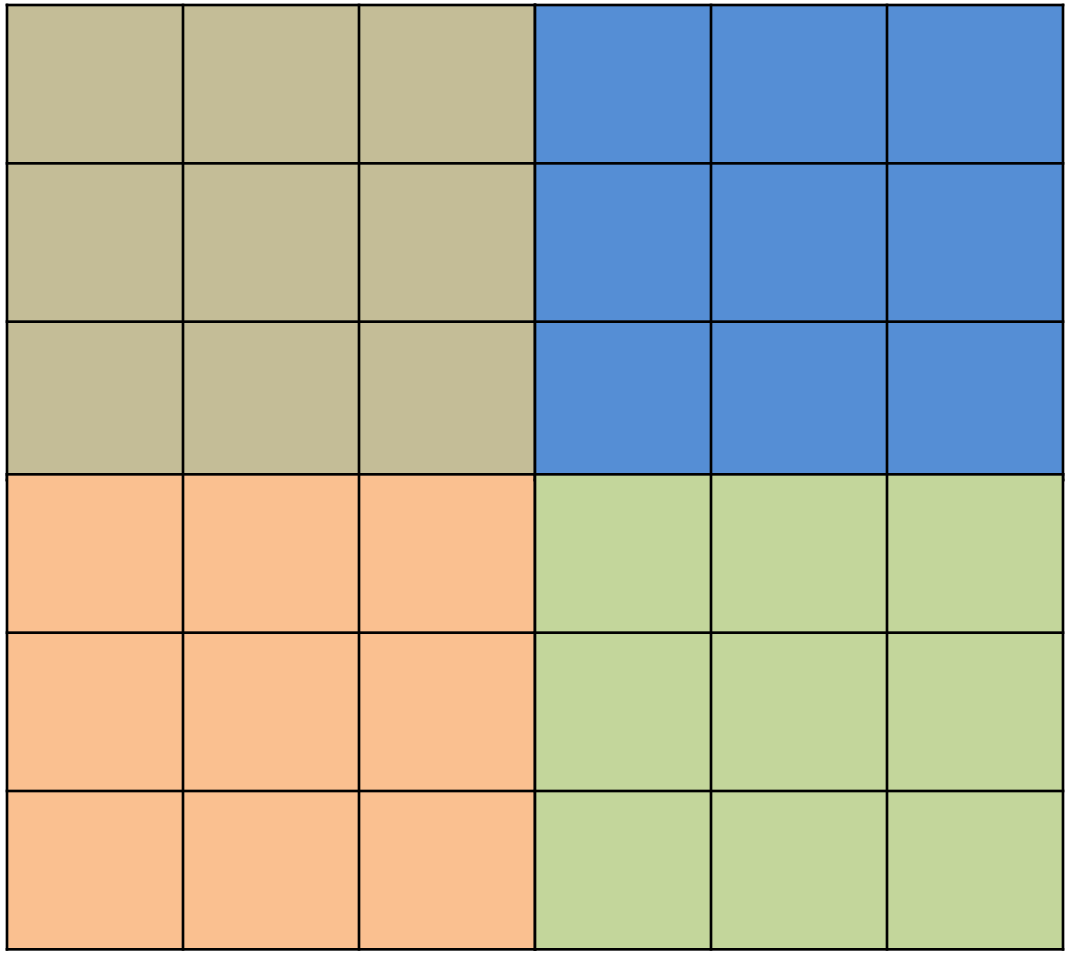
$$x_C(t+1) = \frac{x_N(t) + x_E(t) + x_S(t) + x_W(t)}{4.0}$$

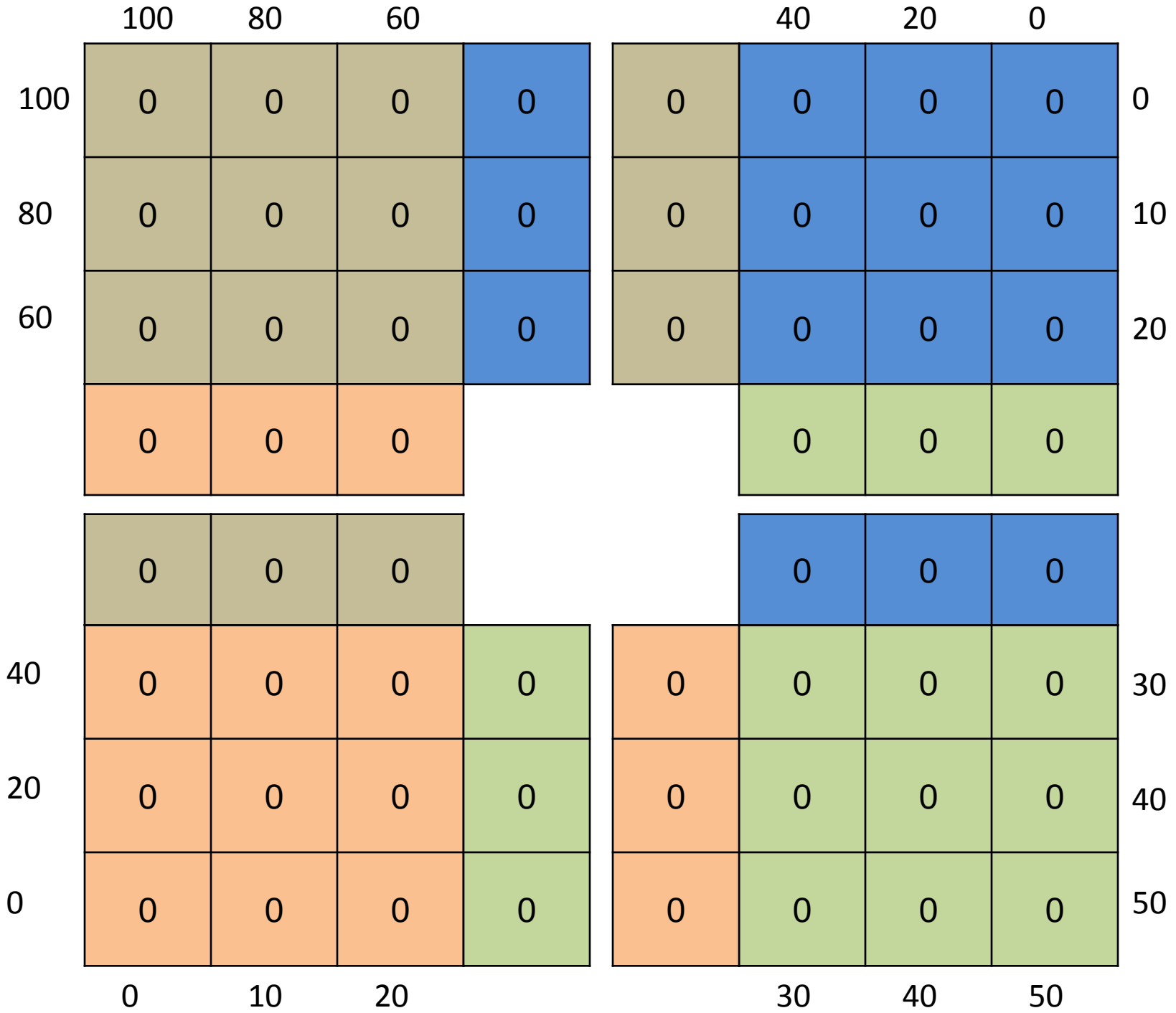
Heat Distribution : Interactive Session

We are going to act out a heat distribution problem.

The take-home message we want to convey is :

- How cooperative computing works
- Notion of mesh/grid partitioning
- Use of messaging
- Explicit synchronization

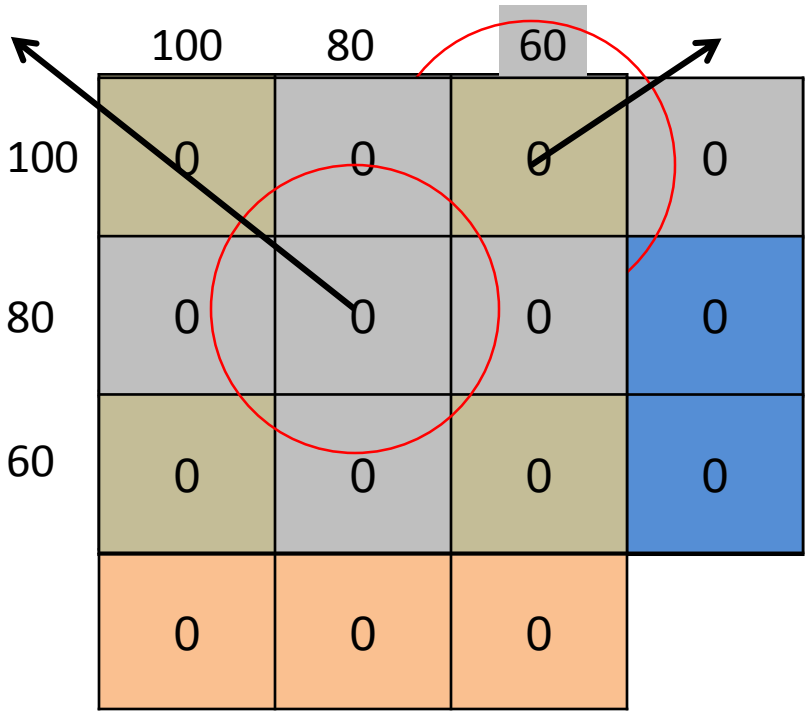



100
80
60
40
20
0
100
80
60
40
20
0
0
10
20
30
40
50

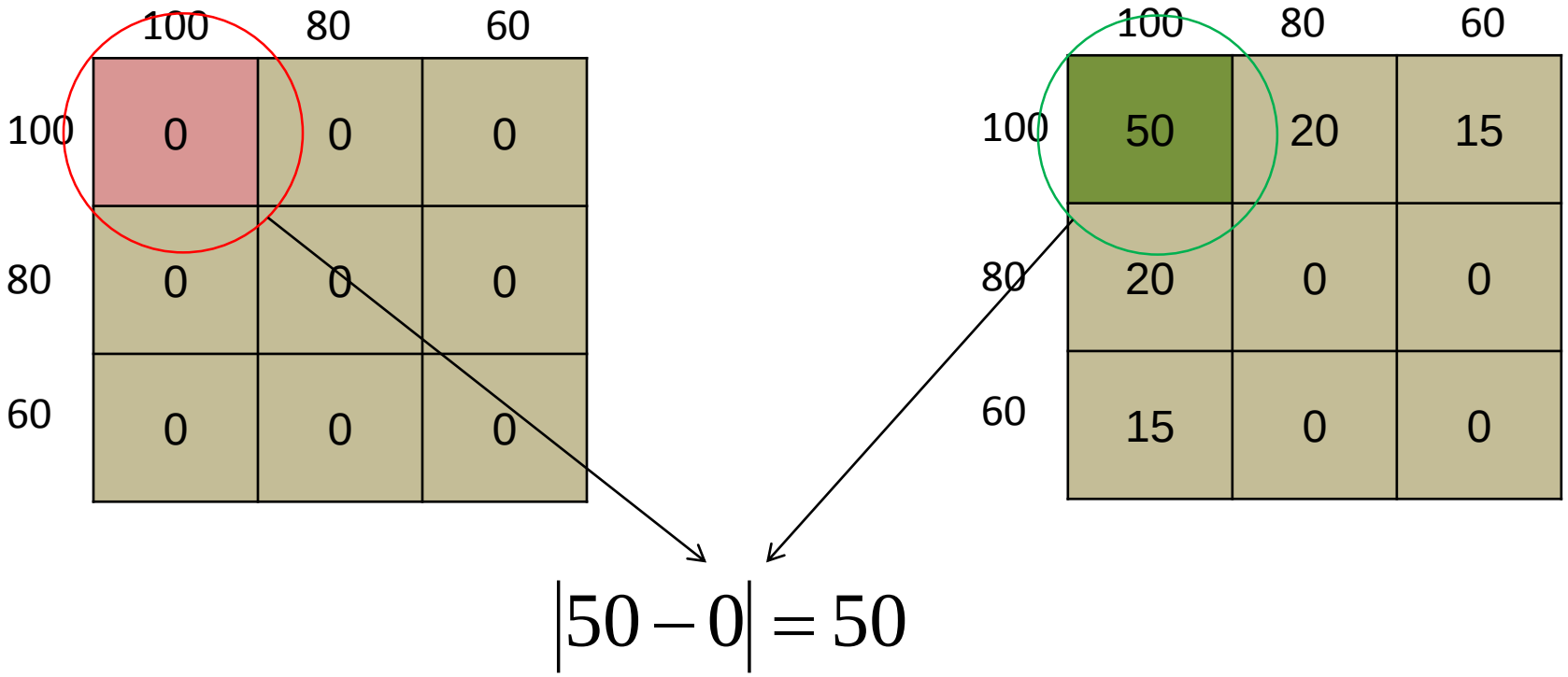
Calculate the value of each cell by averaging its 4 neighboring cells

$$\frac{0+0+0+0}{4}=\frac{0}{4}=0$$

$$\frac{60+0+0+0}{4}=\frac{60}{4}=15$$

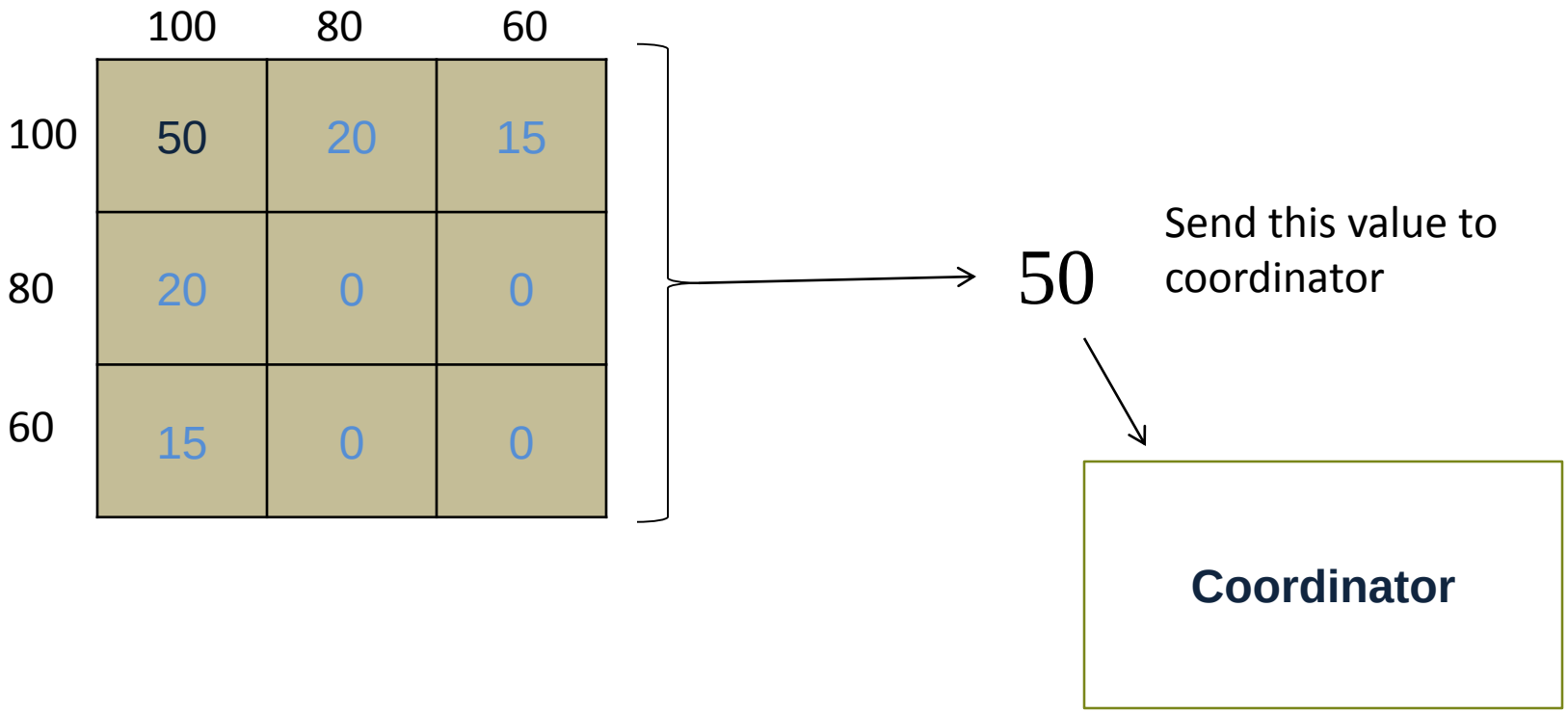


Calculate the difference between the previous cell values and new cell values

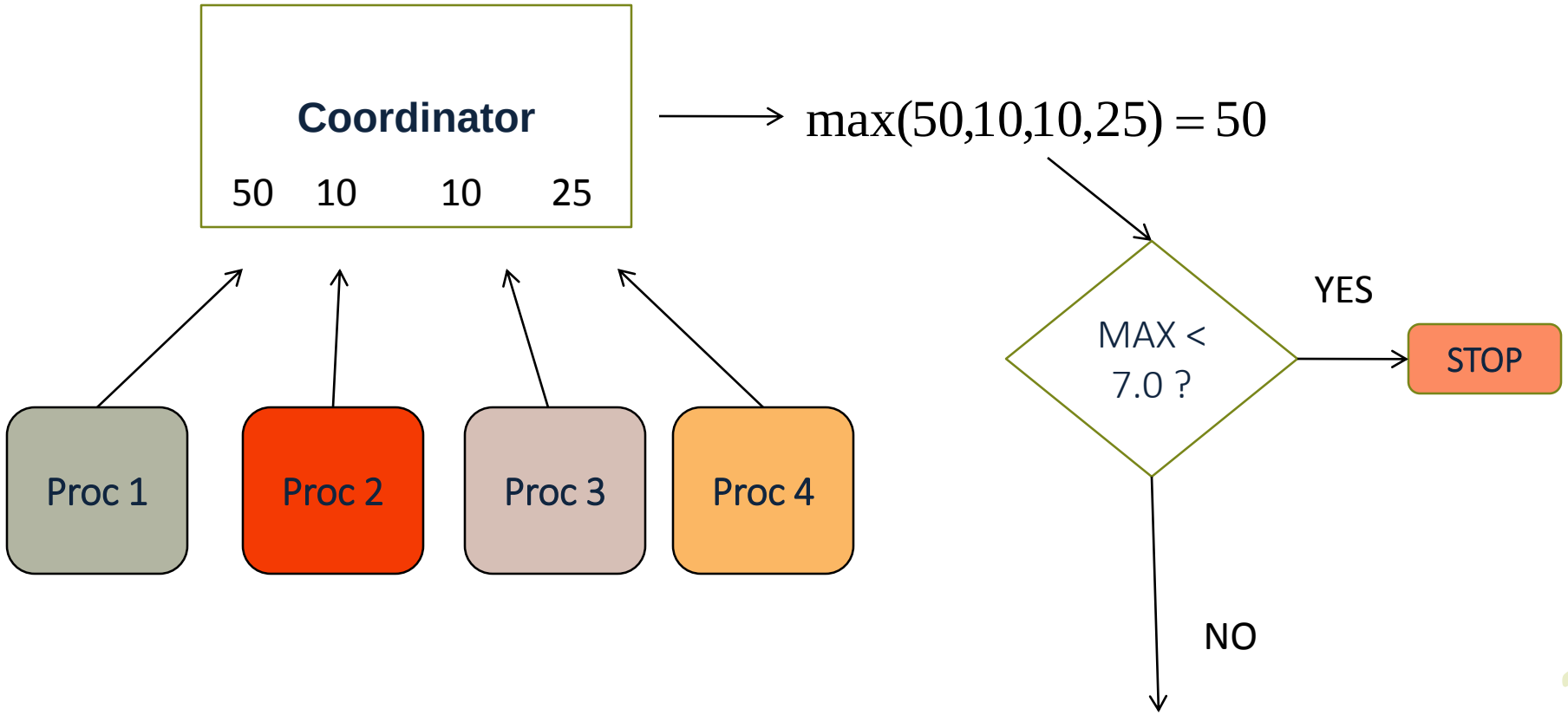


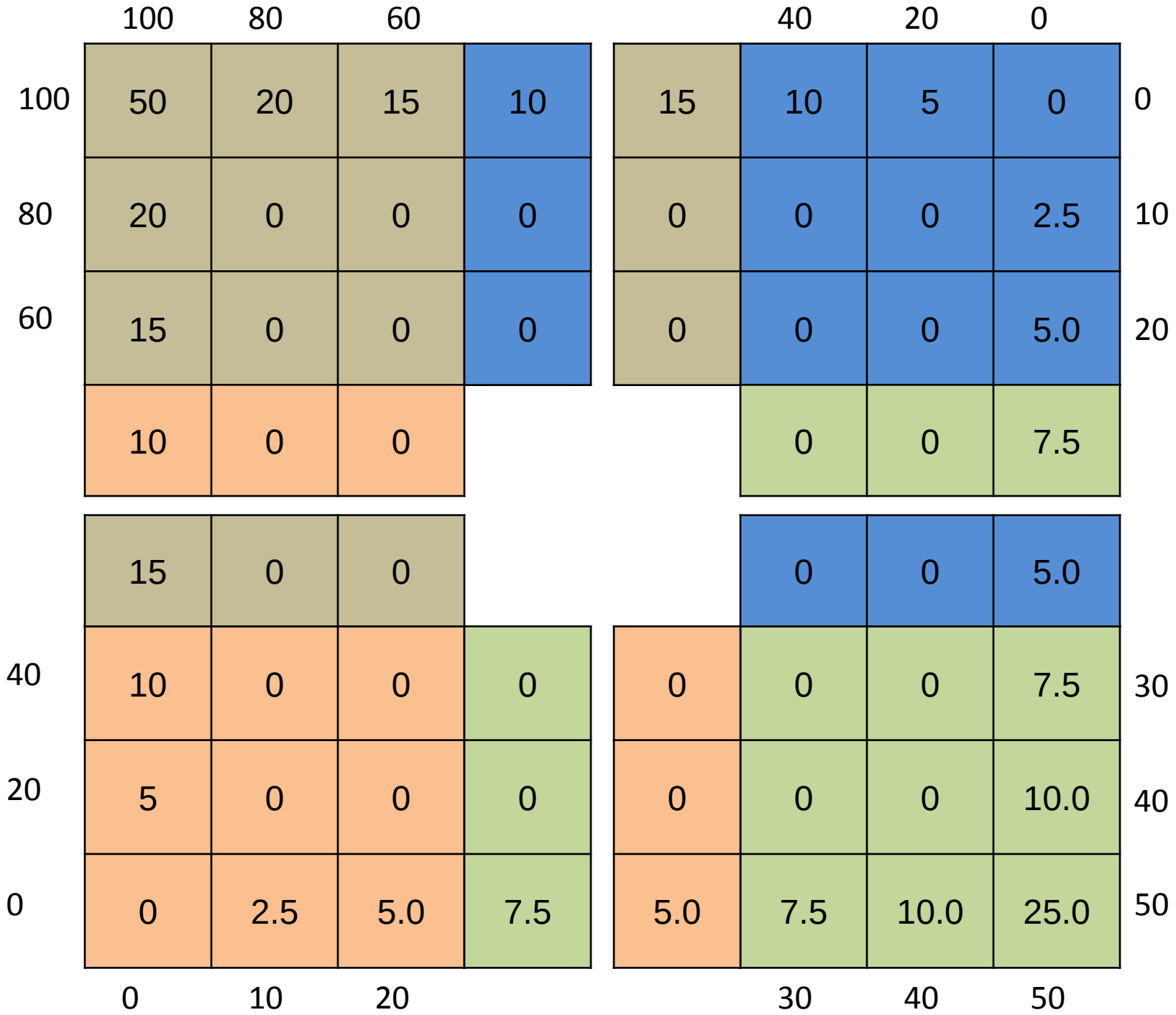
$$|50 - 0| = 50$$

After computing the difference for each cell, Determine the Maximum Temperature ACROSS your problem chunk

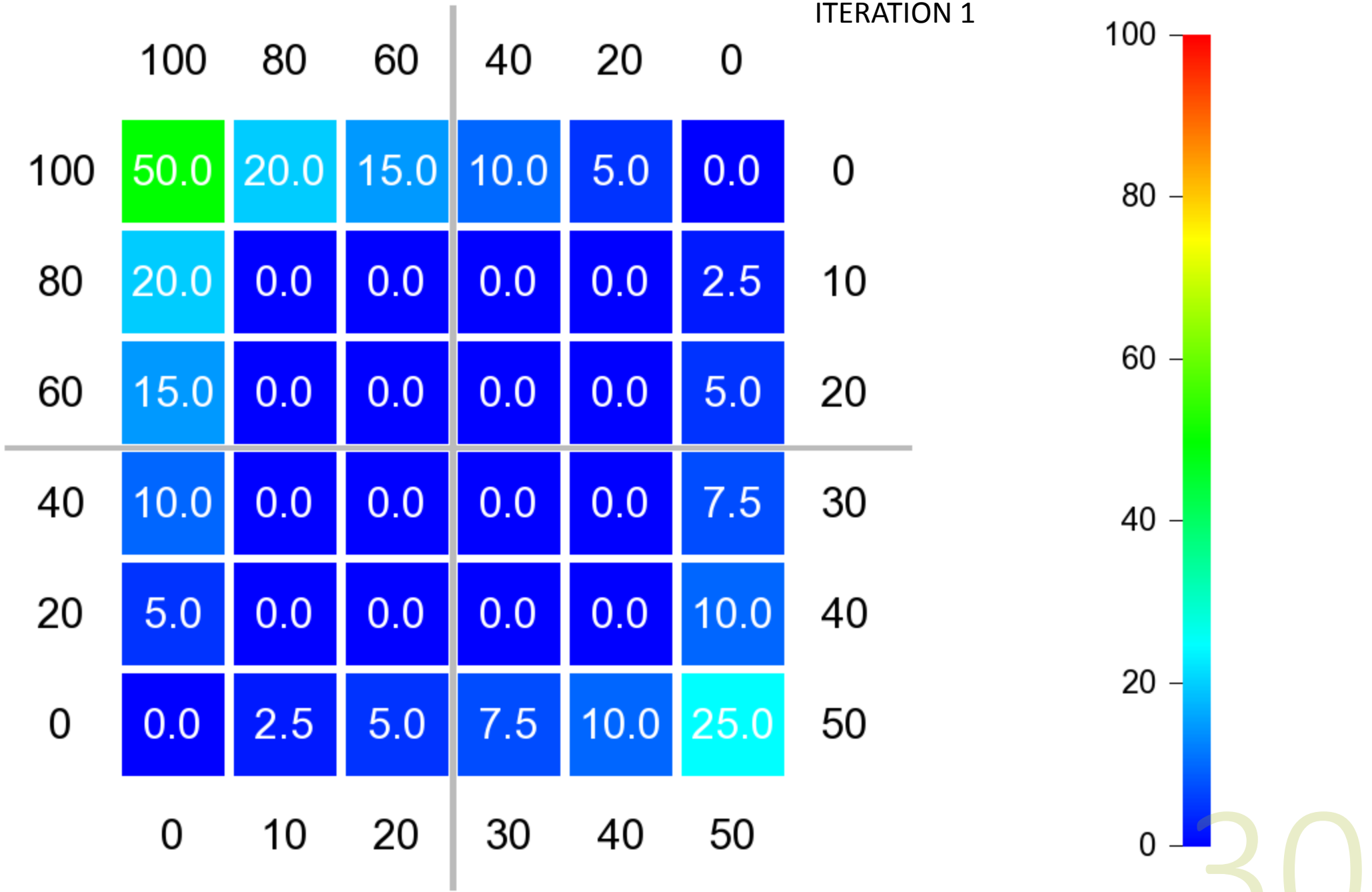


Coordinator Waits for all processing elements to send their values and determines the maximum of all the values it receives

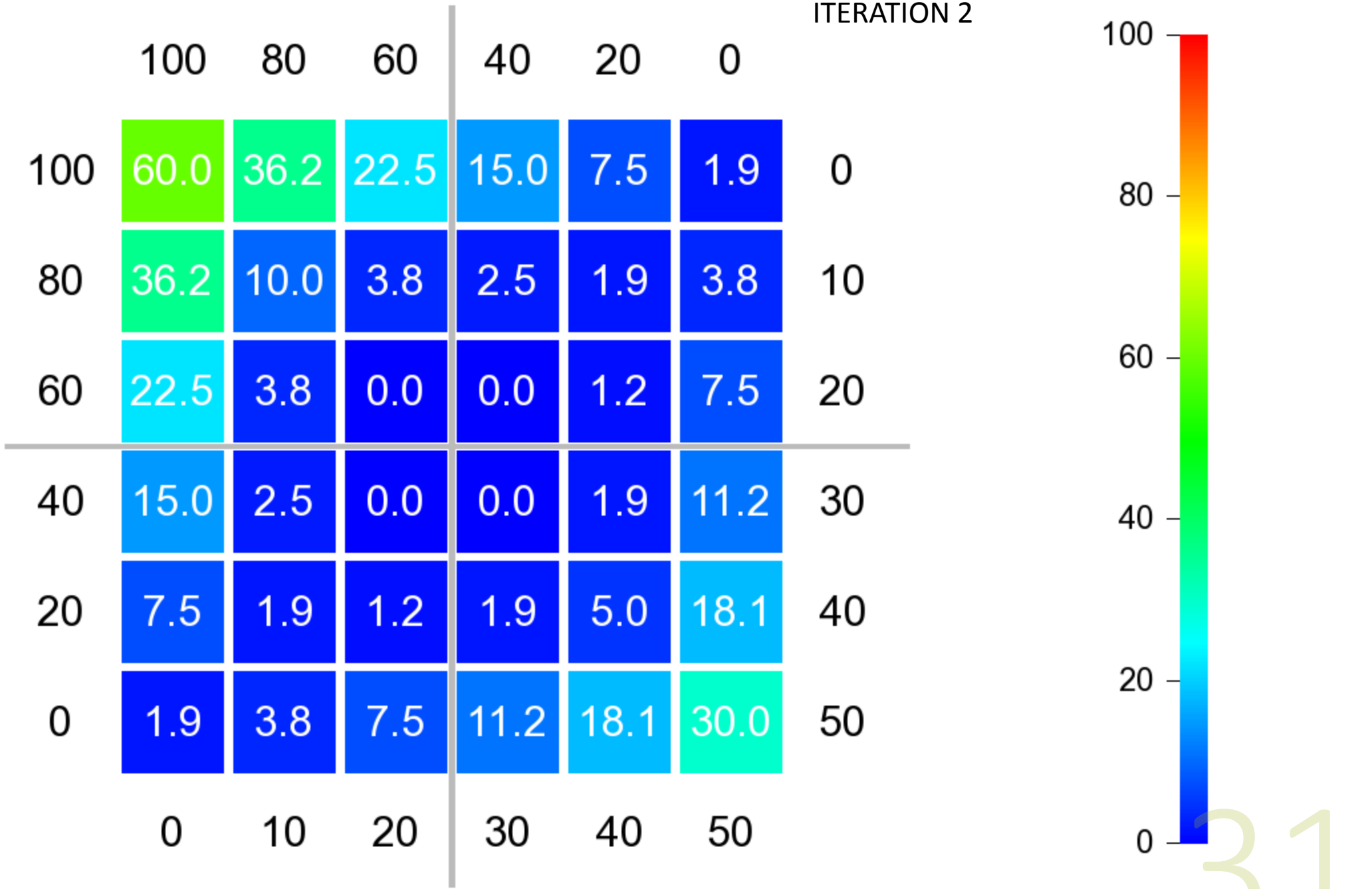



100 80 60 40 20 0
100 80 60 40 20 0
50 20 15 10 15 10 5 0
20 0 0 0 0 0 0 2.5
15 0 0 0 0 0 0 5.0
10 0 0 0 0 7.5
15 0 0 0 0 5.0
10 0 0 0 0 0 0 7.5
5 0 0 0 0 0 0 10.0
0 2.5 5.0 7.5 5.0 7.5 10.0 25.0
0 10 20 30 40 50

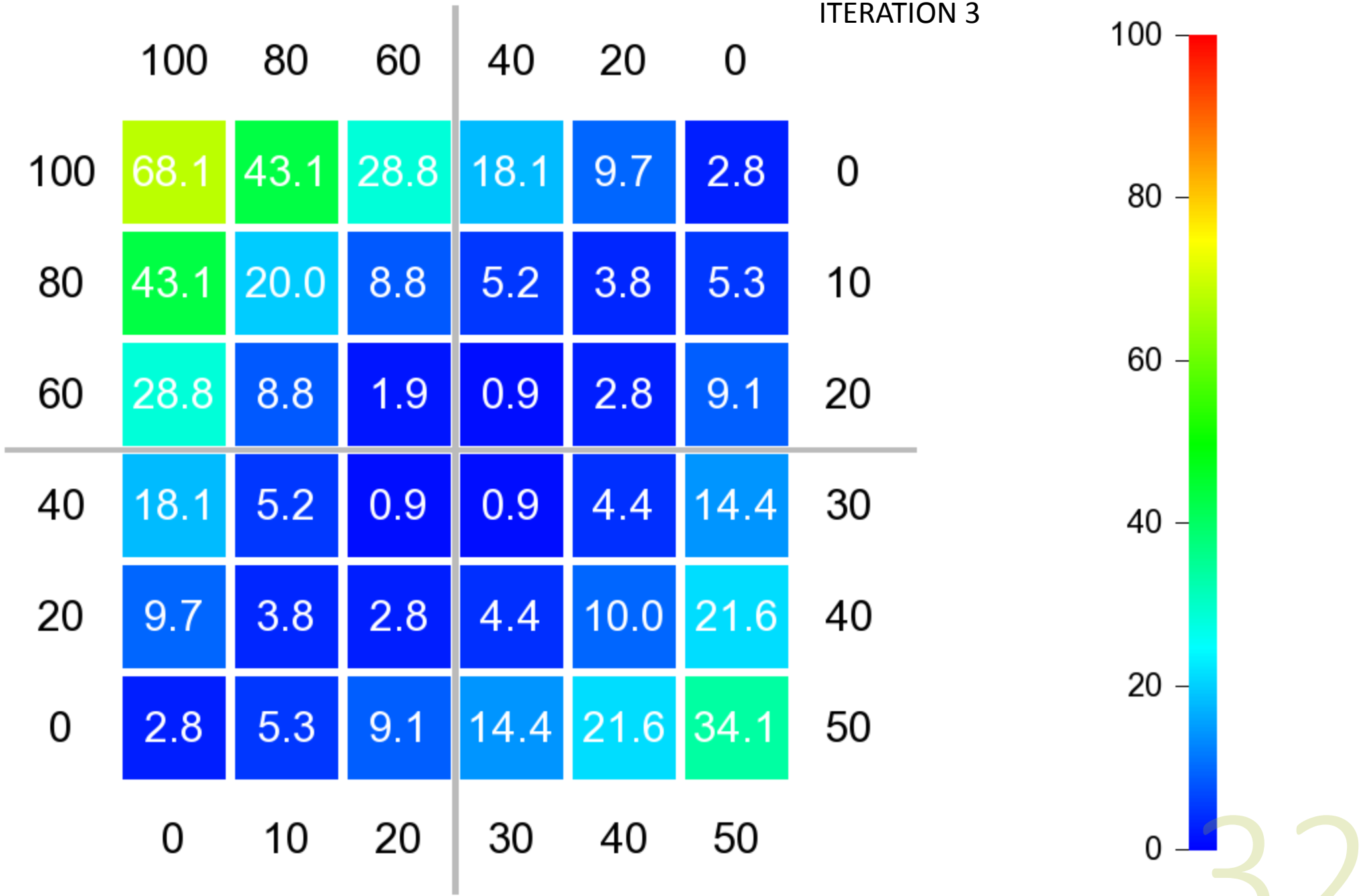
ITERATION 1


100 80 60 40 20 0
100 50.0 20.0 15.0 10.0 5.0 0.0 0
80 20.0 0.0 0.0 0.0 0.0 2.5 10
60 15.0 0.0 0.0 0.0 0.0 5.0 20
40 10.0 0.0 0.0 0.0 0.0 7.5 30
20 5.0 0.0 0.0 0.0 0.0 10.0 40
0 0.0 2.5 5.0 7.5 10.0 25.0 50
0 10 20 30 40 50
100
80
60
40
20
0

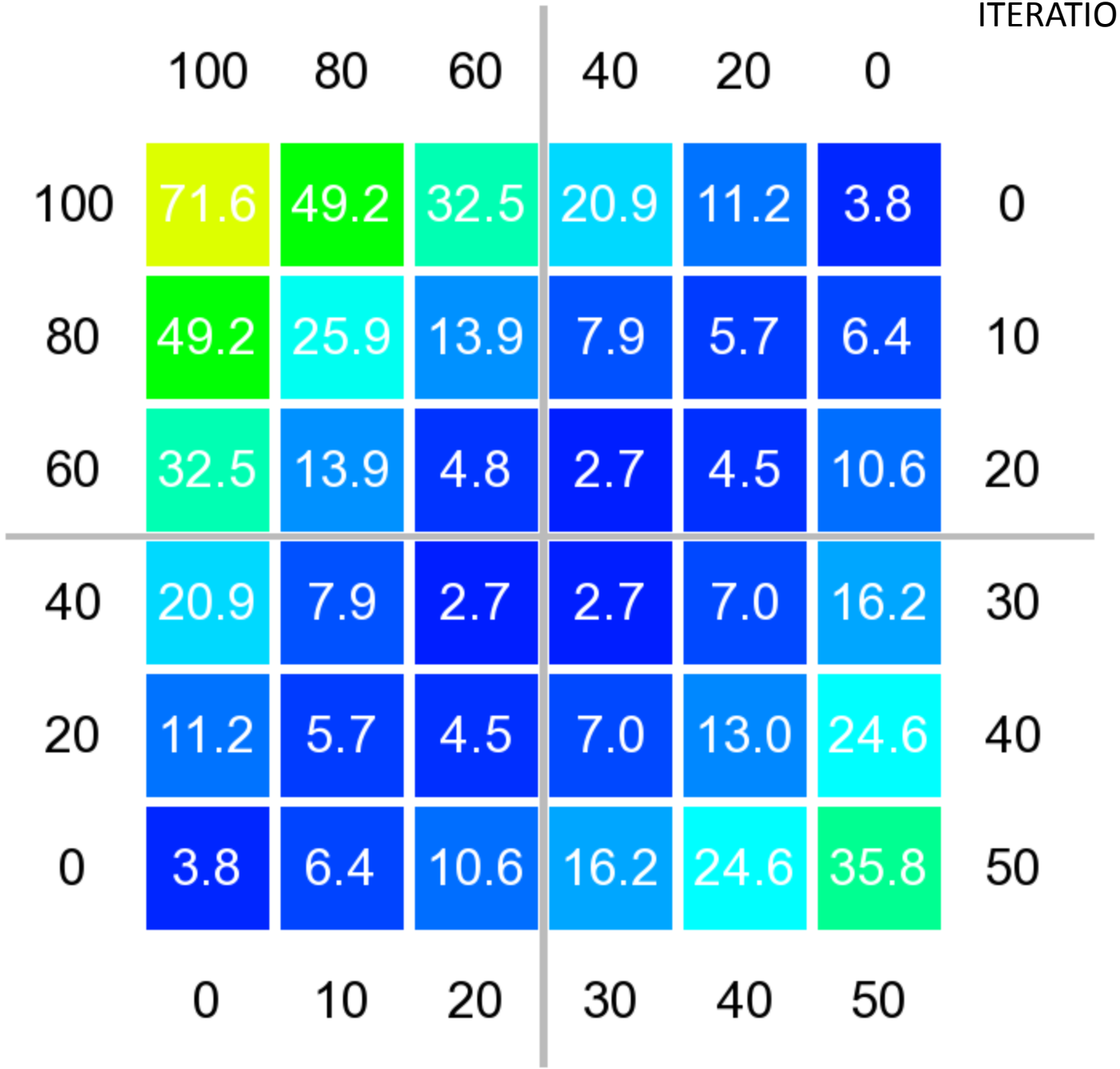
ITERATION 2


100 80 60 40 20 0
100 60.0 36.2 22.5 15.0 7.5 1.9 0
80 36.2 10.0 3.8 2.5 1.9 3.8 10
60 22.5 3.8 0.0 0.0 1.2 7.5 20
40 15.0 2.5 0.0 0.0 1.9 11.2 30
20 7.5 1.9 1.2 1.9 5.0 18.1 40
0 1.9 3.8 7.5 11.2 18.1 30.0 50
0 10 20 30 40 50
100
80
60
40
20
0

ITERATION 3

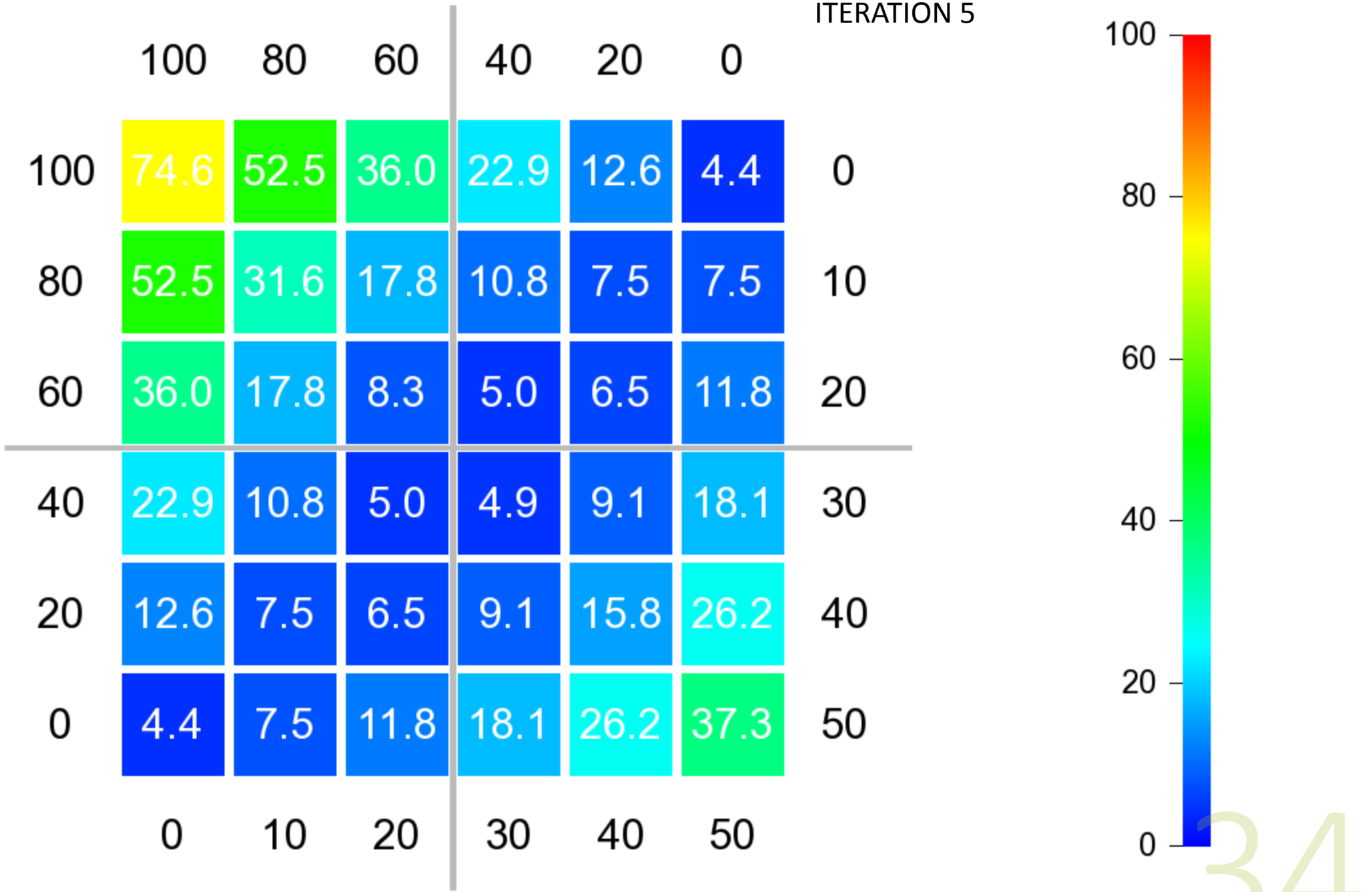

100 80 60 40 20 0
100 68.1 43.1 28.8 18.1 9.7 2.8 0
80 43.1 20.0 8.8 5.2 3.8 5.3 10
60 28.8 8.8 1.9 0.9 2.8 9.1 20
40 18.1 5.2 0.9 0.9 4.4 14.4 30
20 9.7 3.8 2.8 4.4 10.0 21.6 40
0 2.8 5.3 9.1 14.4 21.6 34.1 50
0 10 20 30 40 50
100
80
60
40
20
0

ITERATION 4

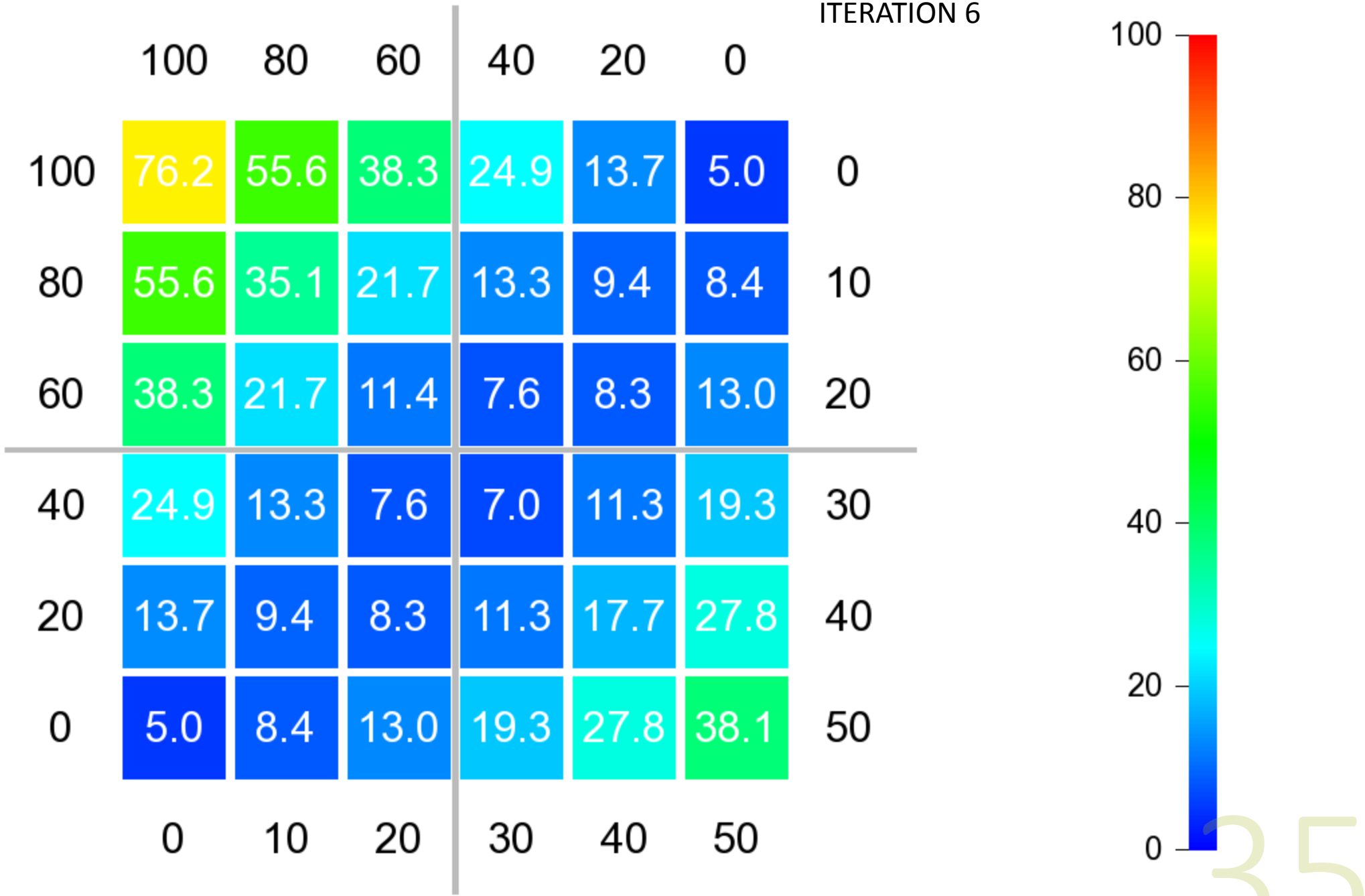

100 80 60 40 20 0
100 71.6 49.2 32.5 20.9 11.2 3.8 0
80 49.2 25.9 13.9 7.9 5.7 6.4 10
60 32.5 13.9 4.8 2.7 4.5 10.6 20
40 20.9 7.9 2.7 2.7 7.0 16.2 30
20 11.2 5.7 4.5 7.0 13.0 24.6 40
0 3.8 6.4 10.6 16.2 24.6 35.8 50
0 10 20 30 40 50


100
80
60
40
20
0

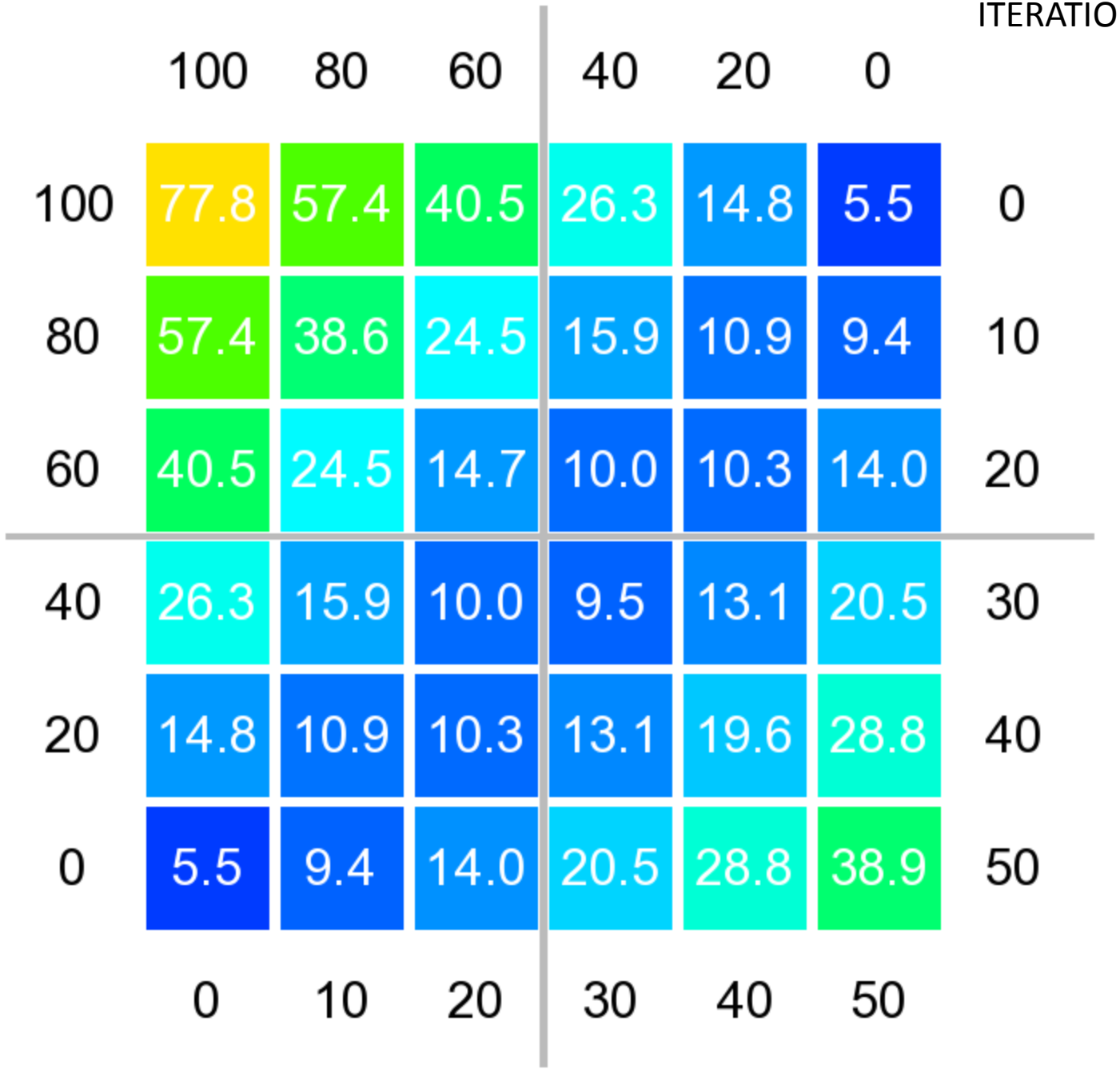
ITERATION 5


100 80 60 40 20 0
100 74.6 52.5 36.0 22.9 12.6 4.4 0
80 52.5 31.6 17.8 10.8 7.5 7.5 10
60 36.0 17.8 8.3 5.0 6.5 11.8 20
40 22.9 10.8 5.0 4.9 9.1 18.1 30
20 12.6 7.5 6.5 9.1 15.8 26.2 40
0 4.4 7.5 11.8 18.1 26.2 37.3 50
0 10 20 30 40 50
100
80
60
40
20
0

ITERATION 6

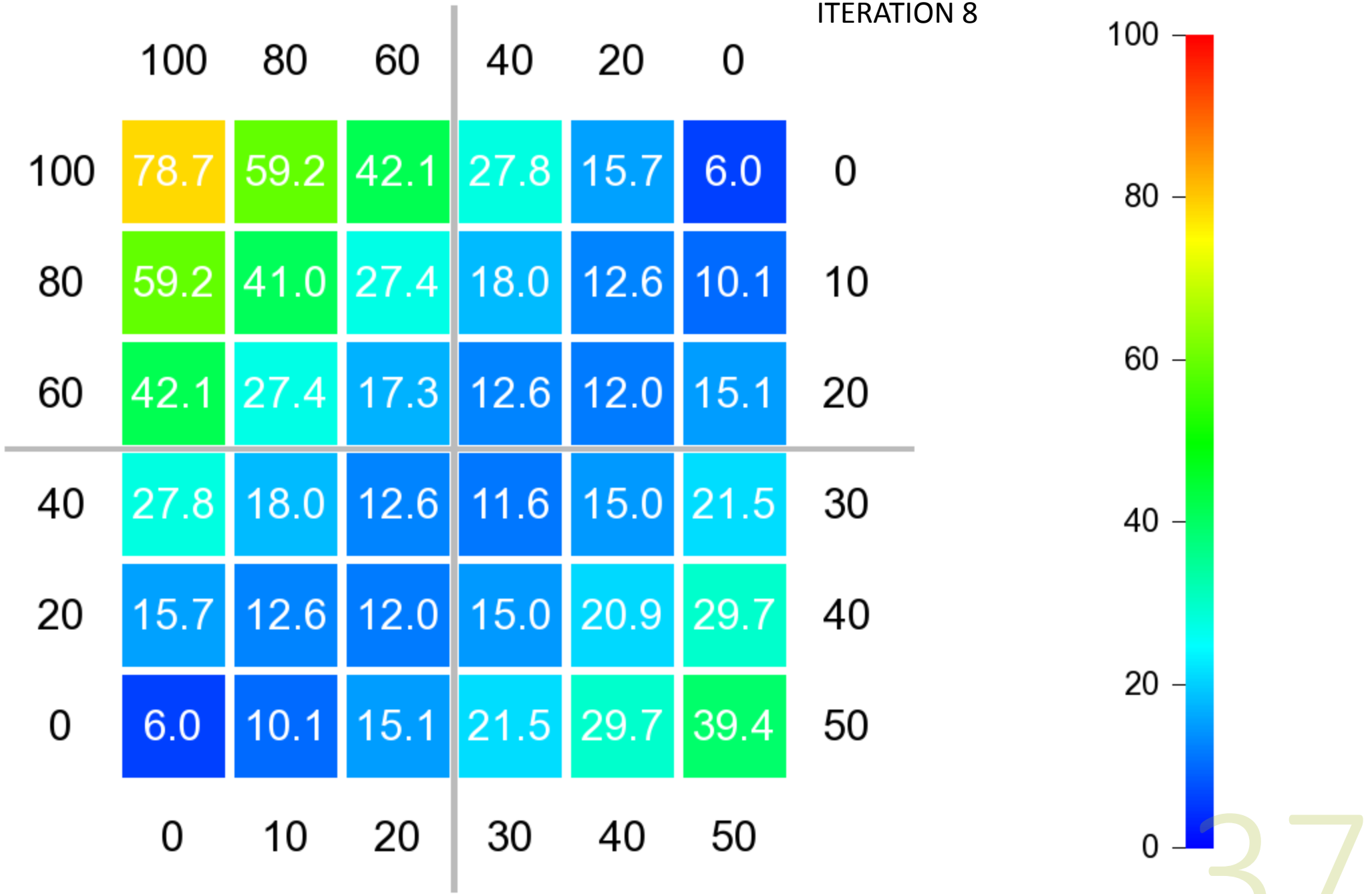

100 80 60 40 20 0
100 76.2 55.6 38.3 24.9 13.7 5.0 0
80 55.6 35.1 21.7 13.3 9.4 8.4 10
60 38.3 21.7 11.4 7.6 8.3 13.0 20
40 24.9 13.3 7.6 7.0 11.3 19.3 30
20 13.7 9.4 8.3 11.3 17.7 27.8 40
0 5.0 8.4 13.0 19.3 27.8 38.1 50
0 10 20 30 40 50
100
80
60
40
20
0

ITERATION 7

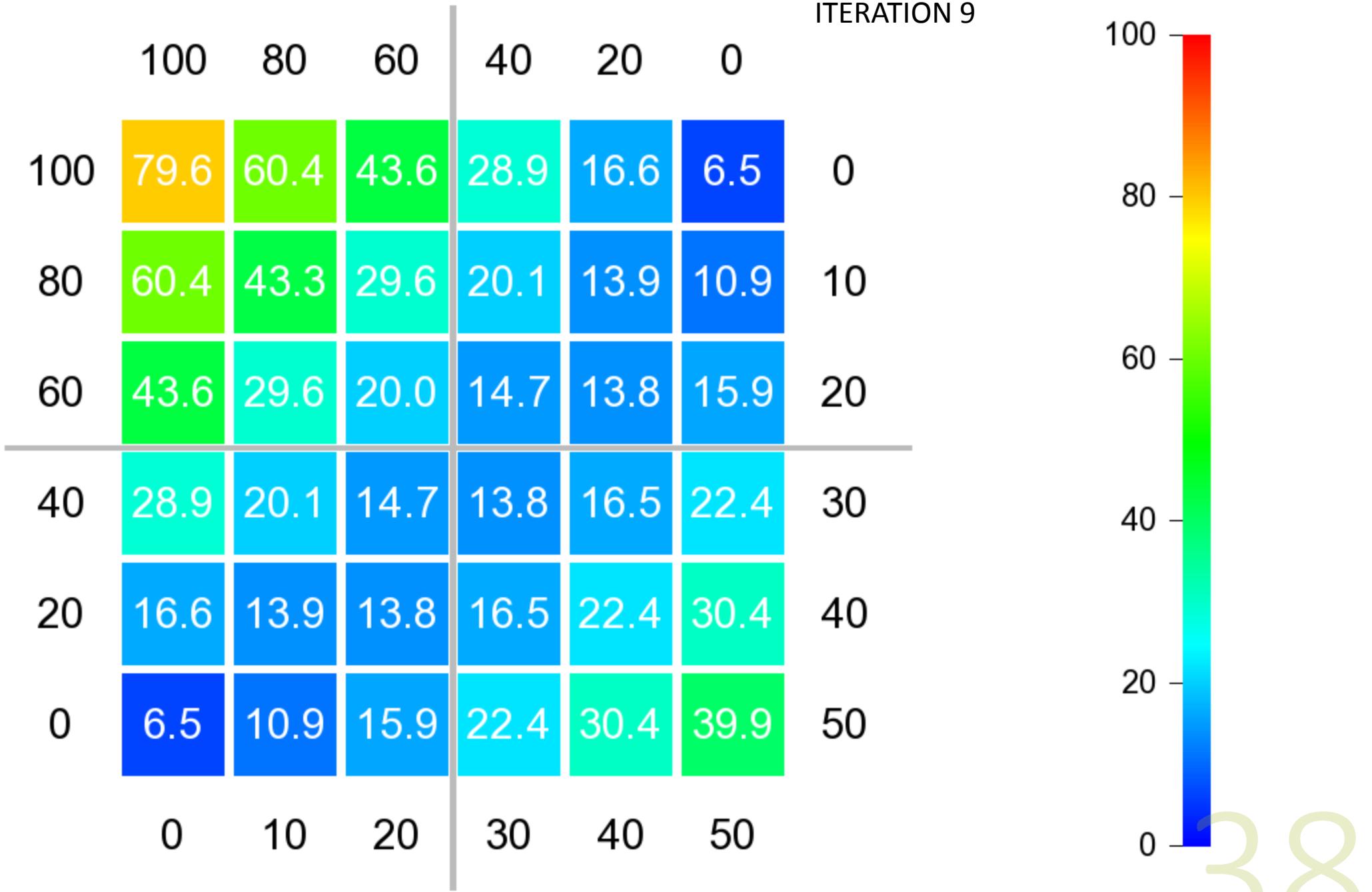

100 80 60 40 20 0
100 77.8 57.4 40.5 26.3 14.8 5.5 0
80 57.4 38.6 24.5 15.9 10.9 9.4 10
60 40.5 24.5 14.7 10.0 10.3 14.0 20
40 26.3 15.9 10.0 9.5 13.1 20.5 30
20 14.8 10.9 10.3 13.1 19.6 28.8 40
0 5.5 9.4 14.0 20.5 28.8 38.9 50
0 10 20 30 40 50


100
80
60
40
20
0

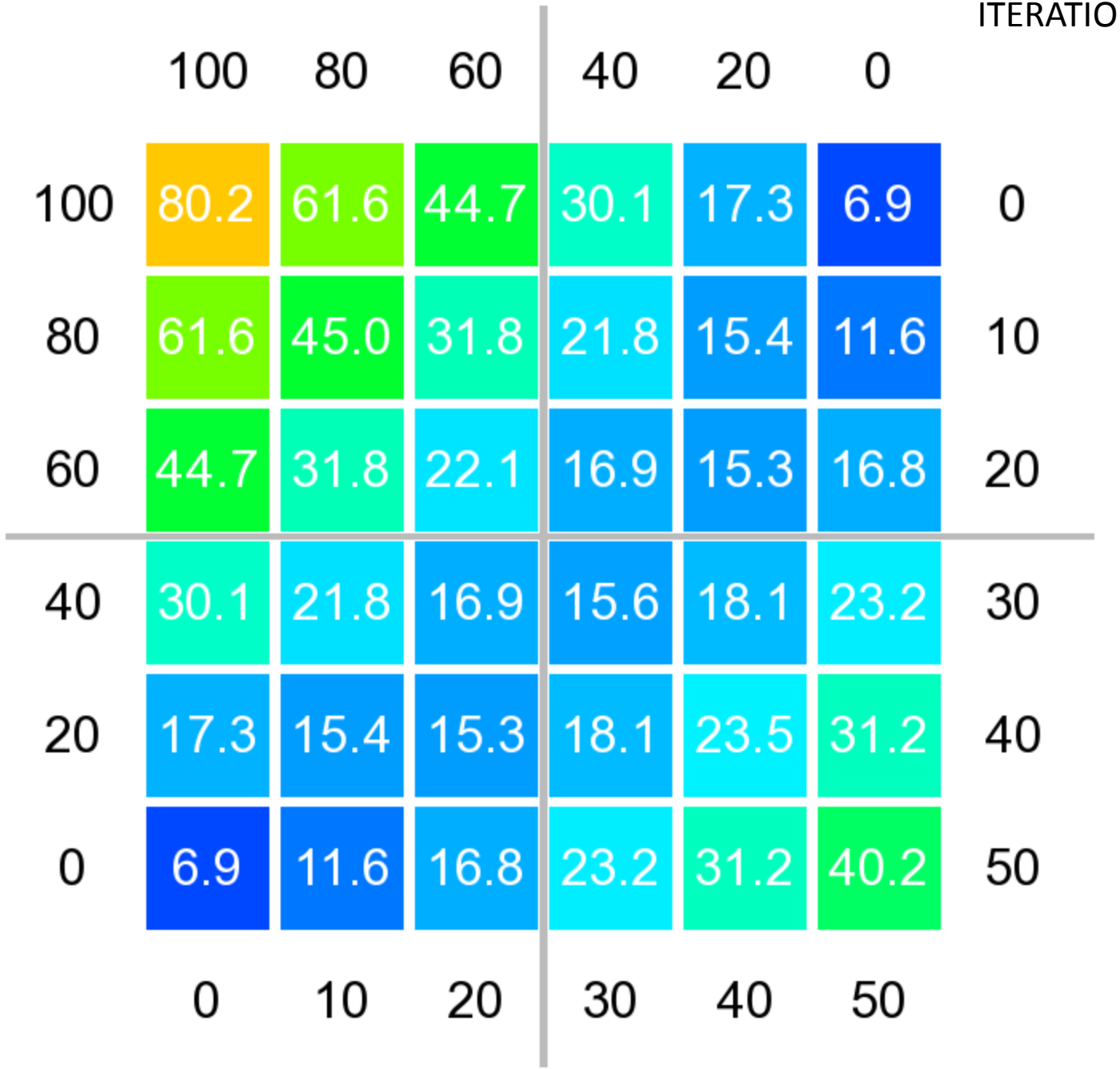
ITERATION 8


100 80 60 40 20 0
100 78.7 59.2 42.1 27.8 15.7 6.0 0
80 59.2 41.0 27.4 18.0 12.6 10.1 10
60 42.1 27.4 17.3 12.6 12.0 15.1 20
40 27.8 18.0 12.6 11.6 15.0 21.5 30
20 15.7 12.6 12.0 15.0 20.9 29.7 40
0 6.0 10.1 15.1 21.5 29.7 39.4 50
0 10 20 30 40 50
100
80
60
40
20
0

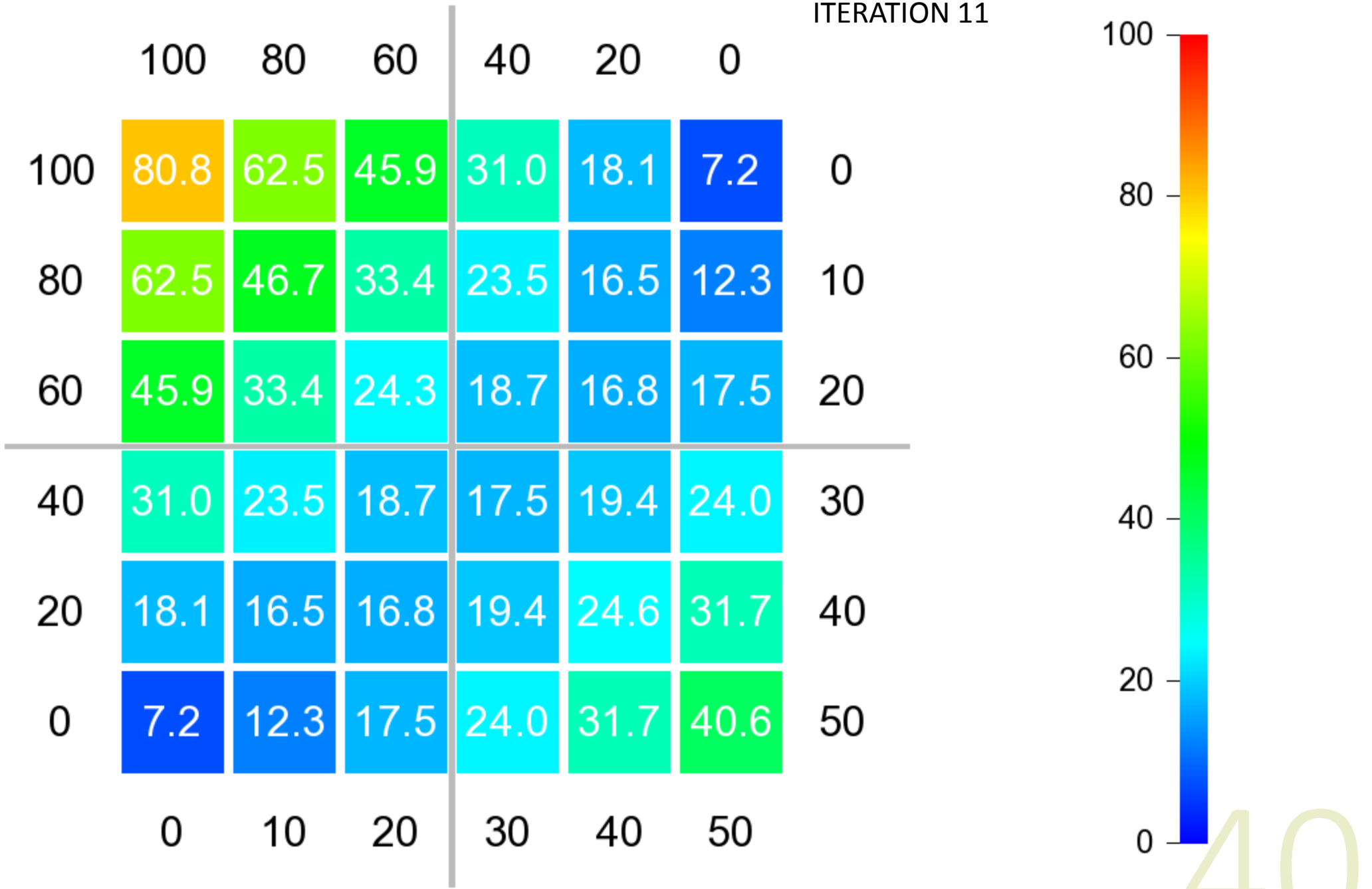
ITERATION 9


100 80 60 40 20 0
100 79.6 60.4 43.6 28.9 16.6 6.5 0
80 60.4 43.3 29.6 20.1 13.9 10.9 10
60 43.6 29.6 20.0 14.7 13.8 15.9 20
40 28.9 20.1 14.7 13.8 16.5 22.4 30
20 16.6 13.9 13.8 16.5 22.4 30.4 40
0 6.5 10.9 15.9 22.4 30.4 39.9 50
0 10 20 30 40 50
100
80
60
40
20
0

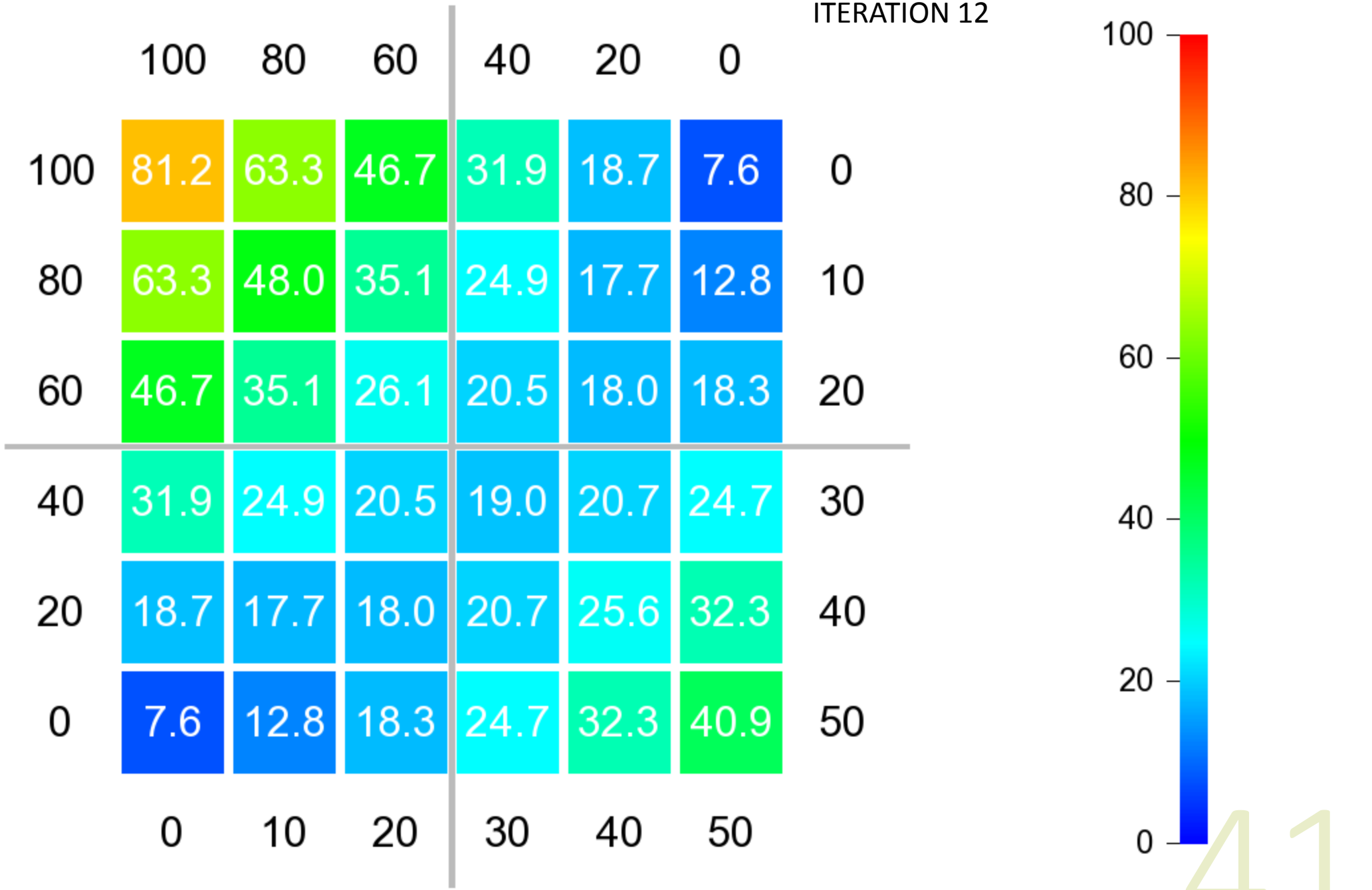
ITERATION 10



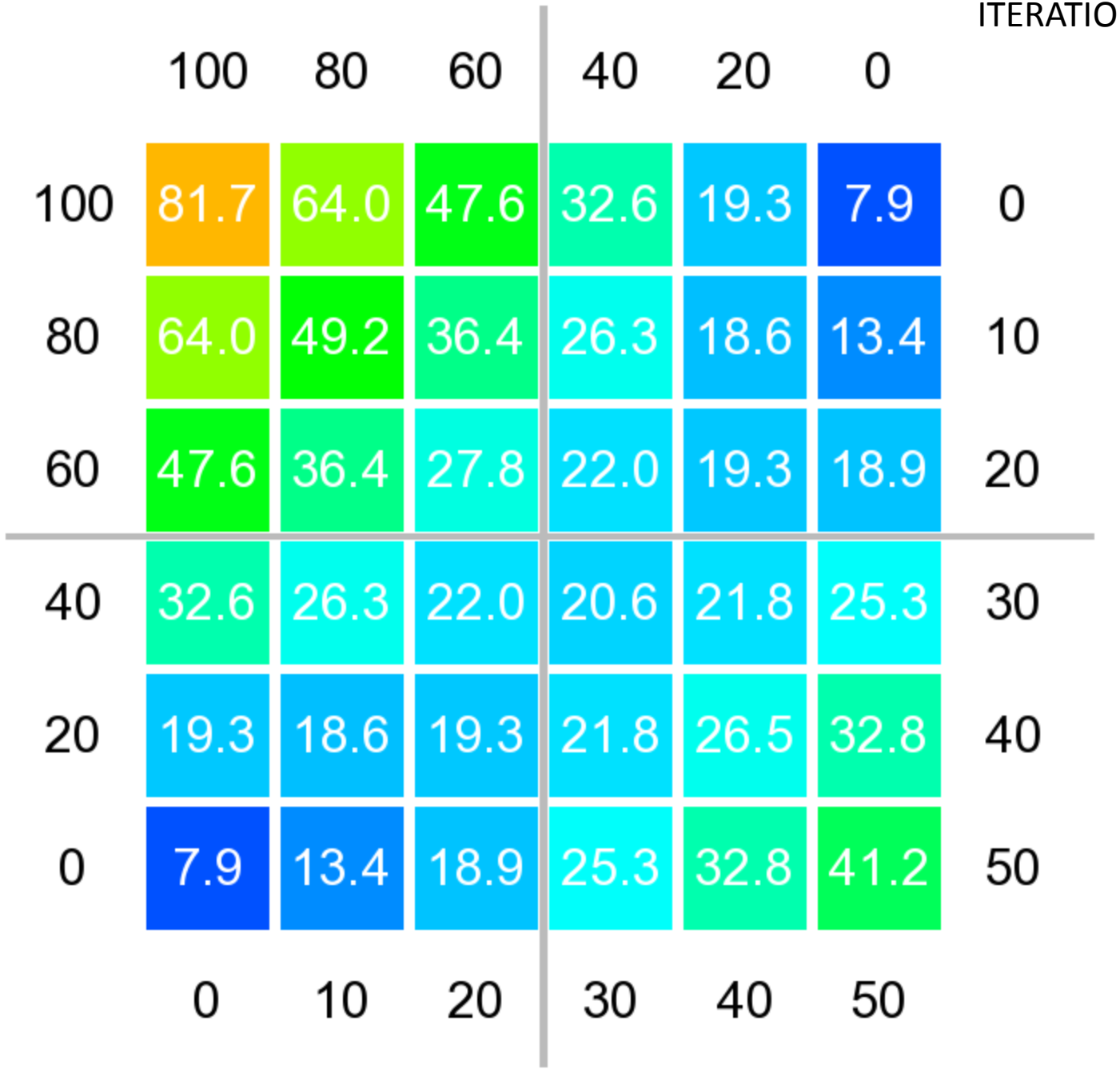
ITERATION 11



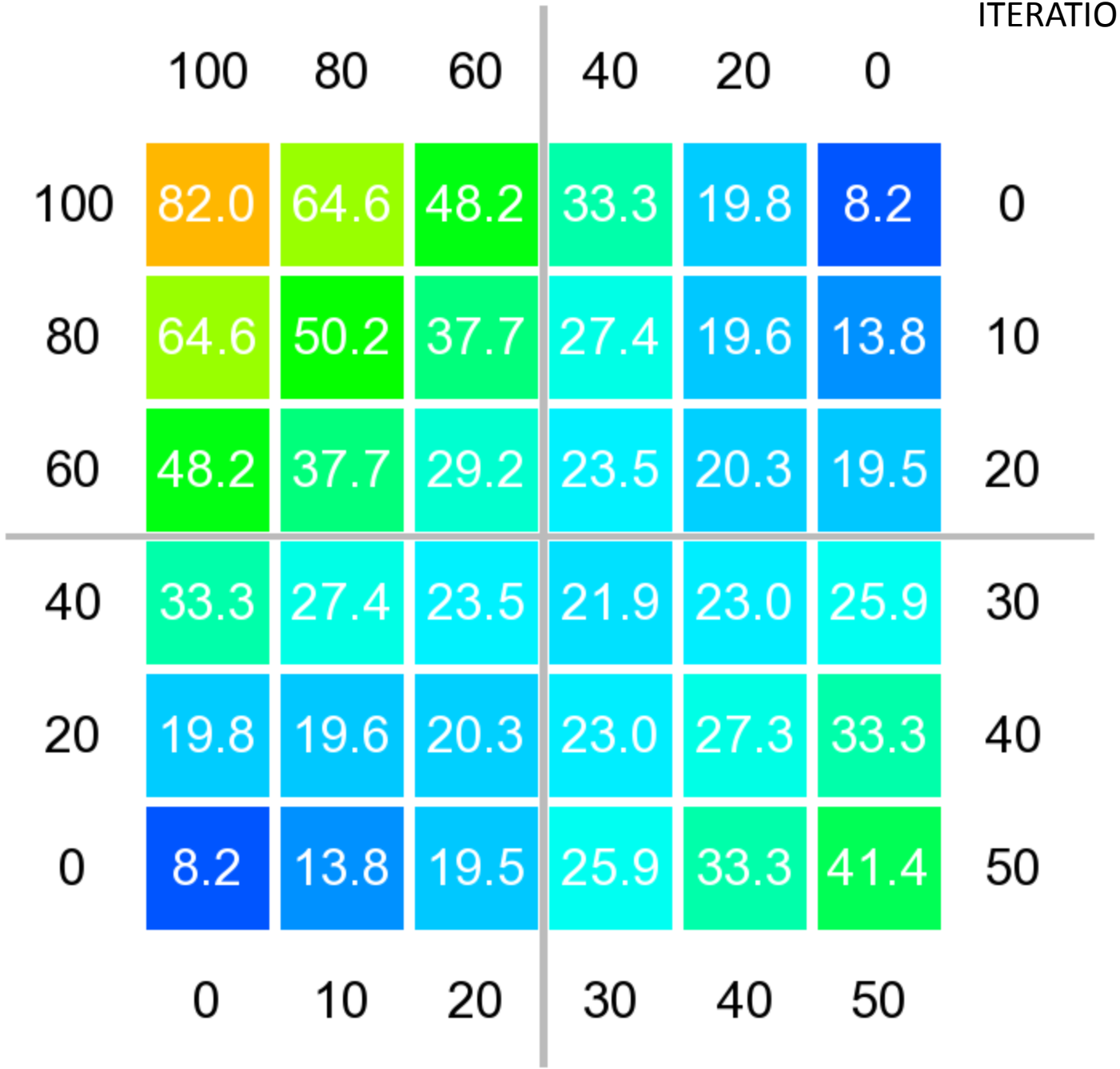
ITERATION 12



ITERATION 13

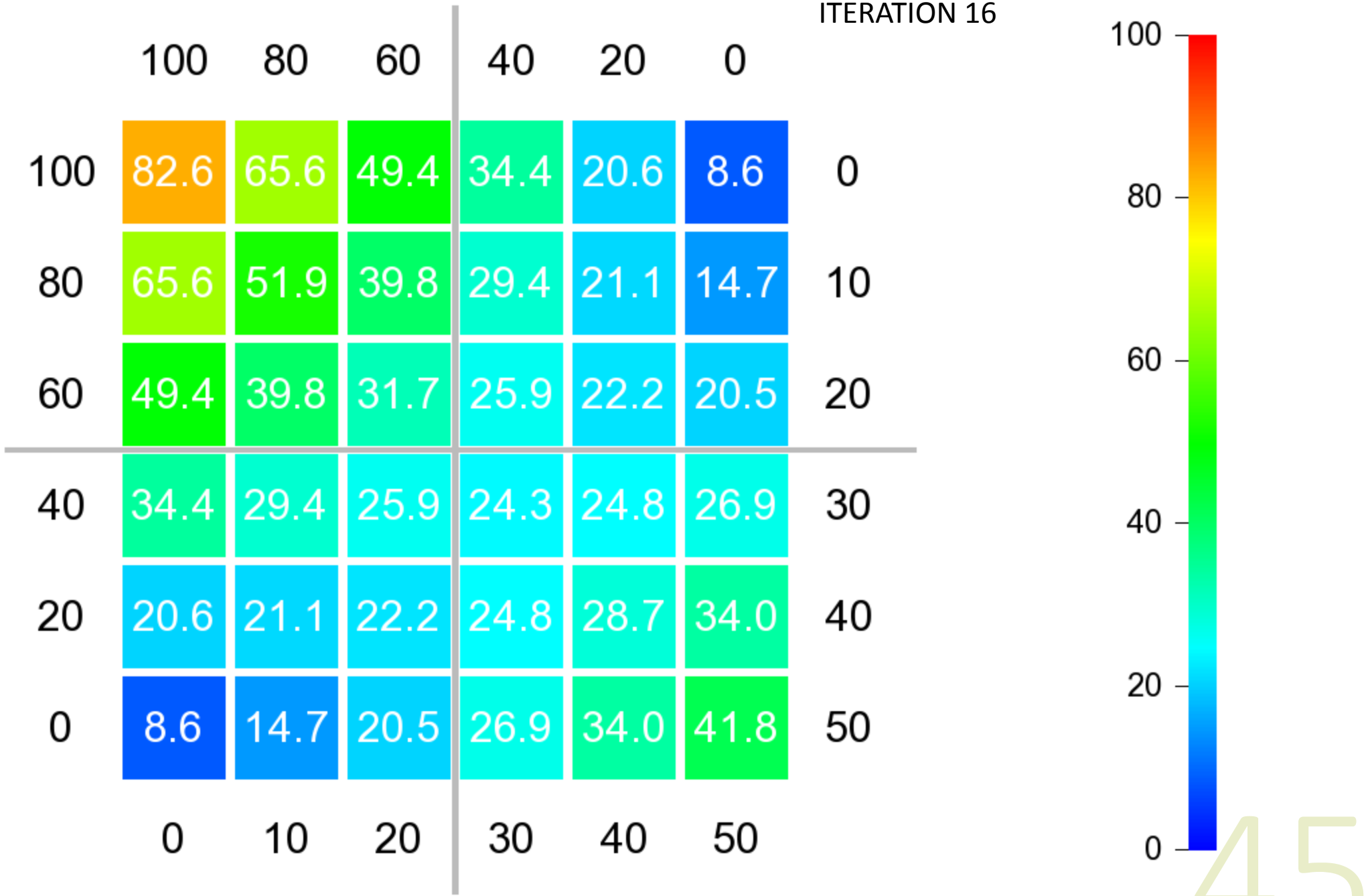


ITERATION 14

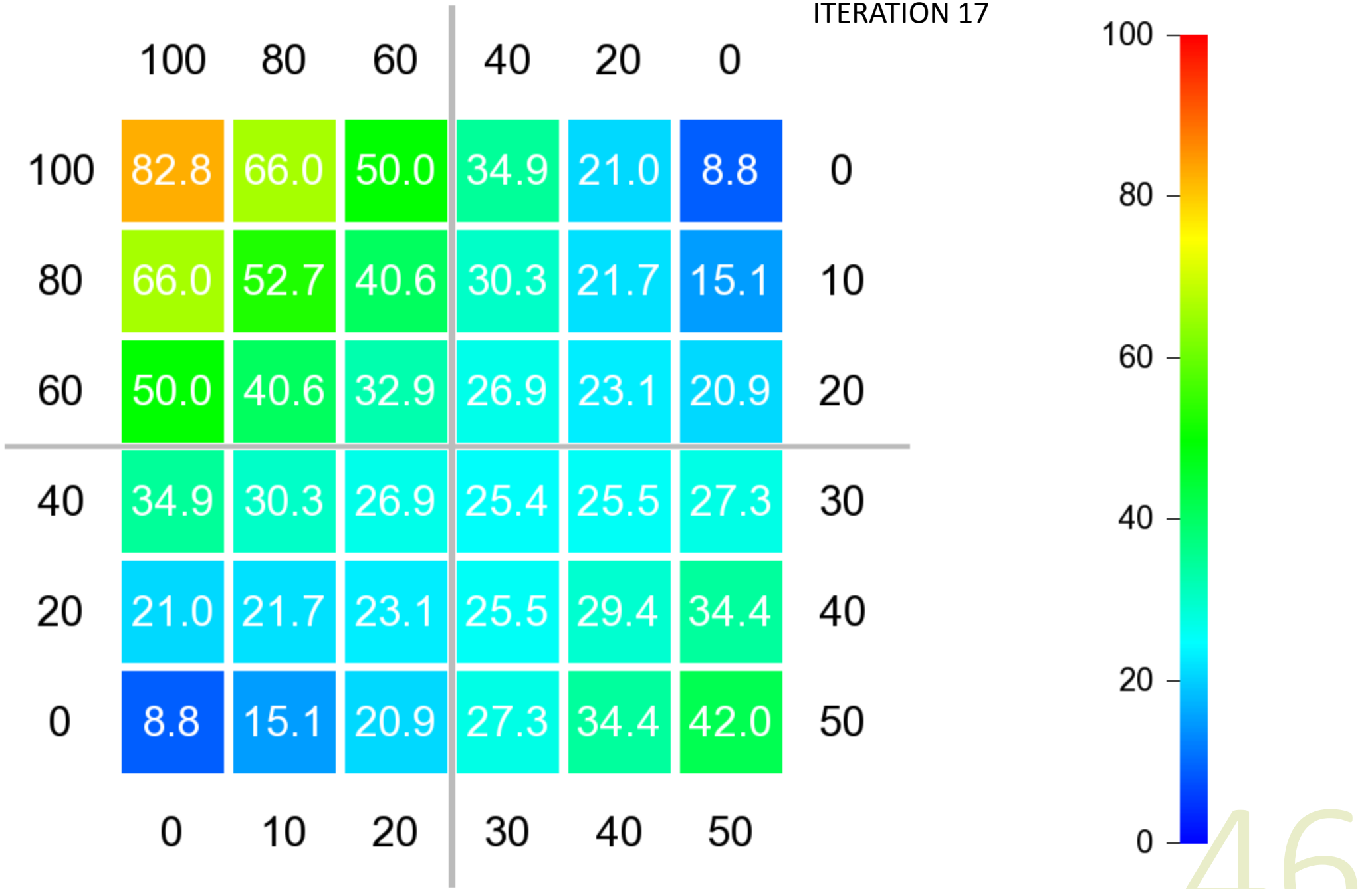


ITERATION 15

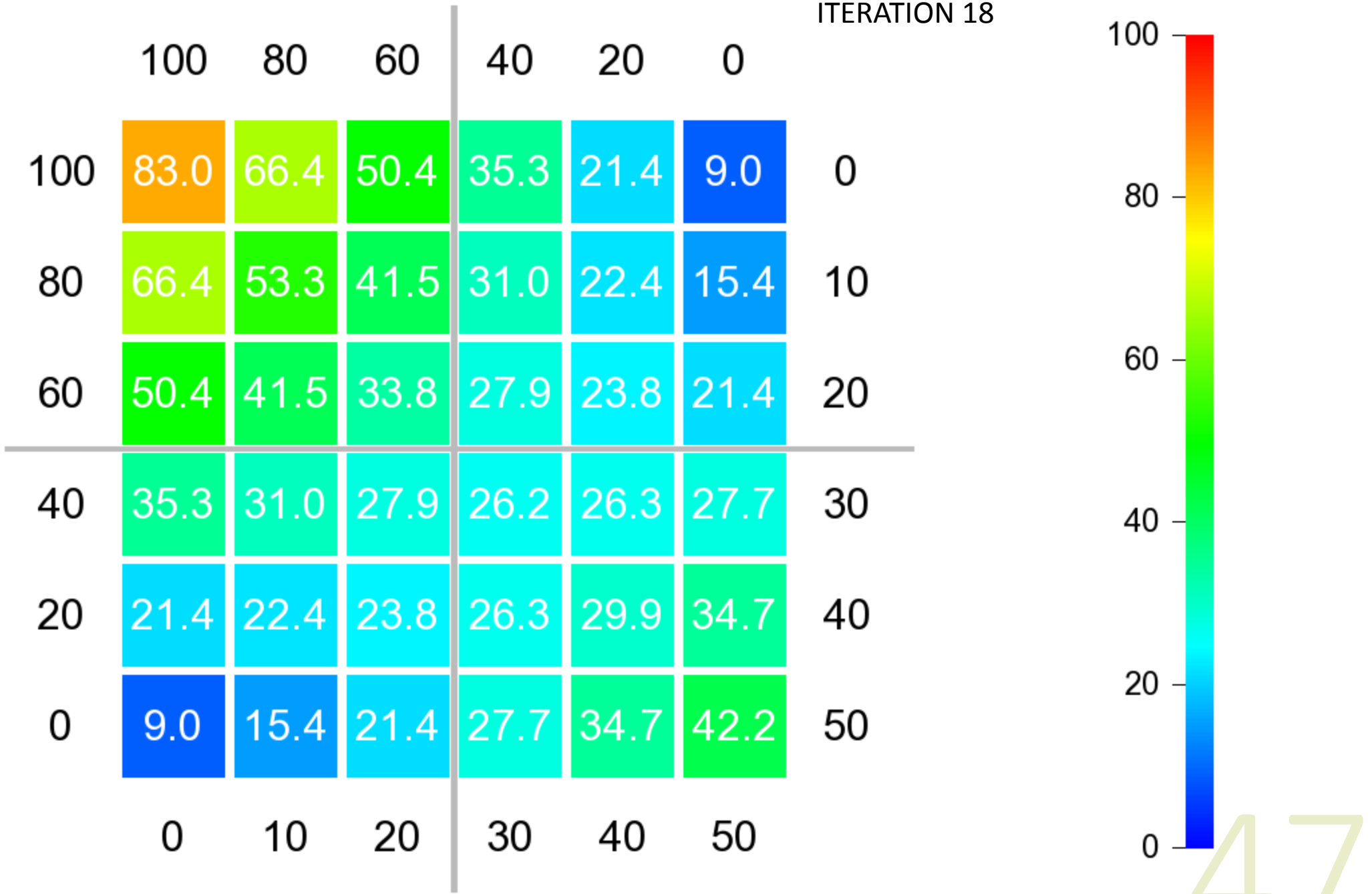
ITERATION 16



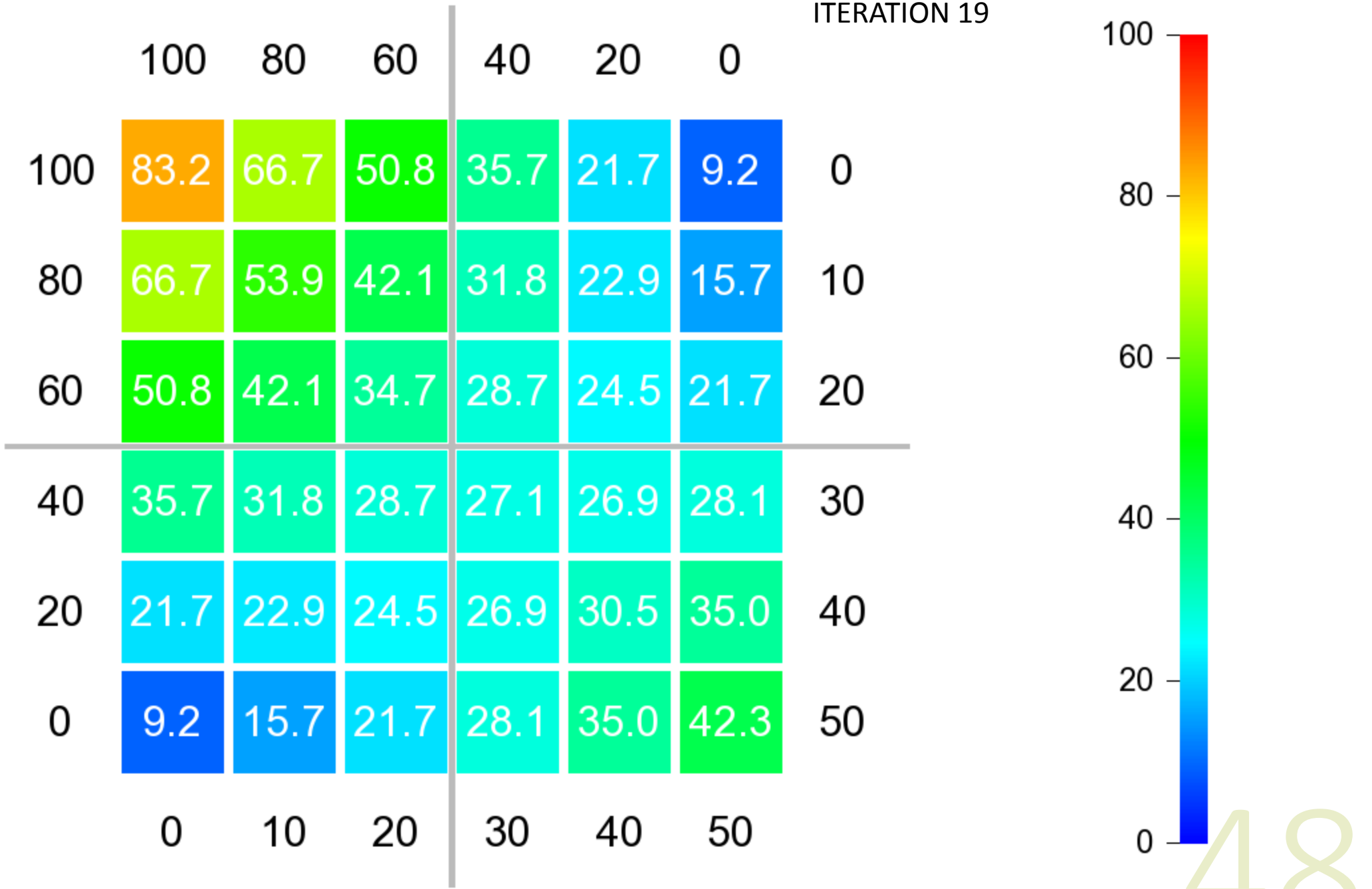
ITERATION 17



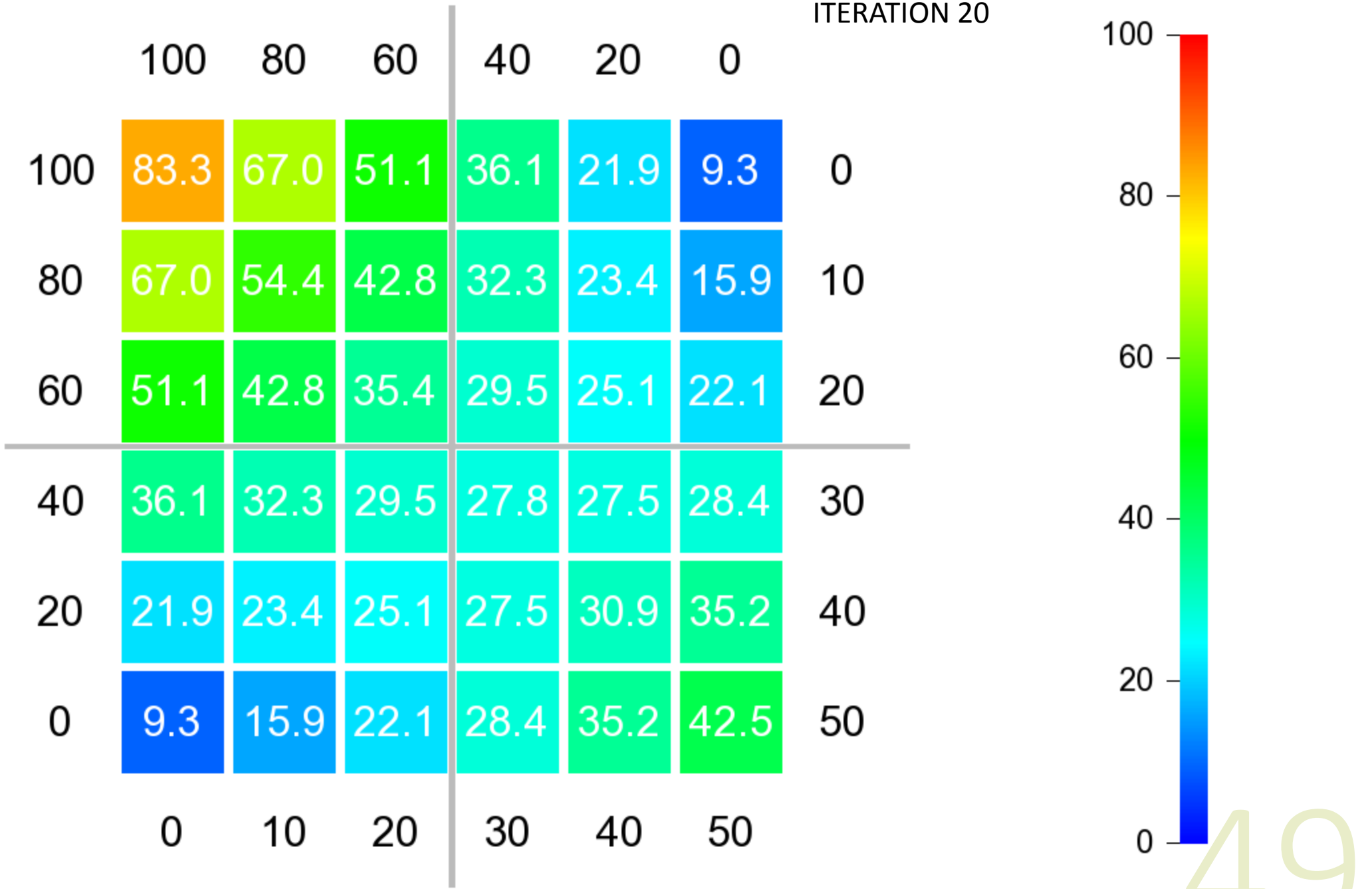
ITERATION 18



ITERATION 19



ITERATION 20



ITERATION 50

ITERATION 100

Performance Issues for CSP

Parallelism speeds things up

Data exchange slows things down

Finer grain partitioning
- provides more parallelism
  - *Can use more processors*
- requires more fine grain messages
  - *Overhead becomes more significant per datum*
- fewer operations per message
  - *Overhead of communication becomes more significant per operation*

Synchronization is another source of overhead

Computation and communication not overlapped

Performance Issues for CSP

- Communication (Gather / Scatter, Data exchange)
  - Latency
    - *Network Distance, Message size*
  - Contention
    - *Network Bandwidth*
  - Overhead
    - *Network interfaces & protocols used*
- Synchronization (Blocking read – writes, barriers)
  - Overhead
- Load Balancing
  - Non-uniform work tasks
  - Starvation
  - Overhead

MPI

Message Passing Interface

Opening Remarks

Context: distributed memory parallel computers

We have communicating sequential processes, each with their own memory, and no access to another process's memory

- A fairly common scenario from the mid 1980s (Intel Hypercube) to today
- Processes interact (exchange data, synchronize) through message passing
- Initially, each computer vendor had its own library and calls
- First standardization was PVM
  - *Started in 1989, first public release in 1991*
  - *Worked well on distributed machines*
  - *Next was MPI*

The MPI Standard

MPI Standard

From 1992-1994, a community representing both vendors and users decided to create a standard interface to message passing calls in the context of distributed memory parallel computers (MPPs, there weren't really clusters yet)

MPI-1 was the result

- "Just" an API
- FORTRAN77 and C bindings
- Reference implementation (mpich) also developed
- Vendors also kept their own internals (behind the API)

MPI Standard

Since then
- MPI-1.1
  - *Fixed bugs, clarified issues*
- MPI-2
  - *Included MPI-1.2*
    - Fixed more bugs, clarified more issues
  - *Extended MPI*
    - New datatype constructors, language interoperability
  - *New functionality*
    - One-sided communication
    - MPI I/O
    - Dynamic processes
  - *FORTRAN90 and C++ bindings*

Best MPI reference
- MPI Standard - on-line at: http://www.mpi-forum.org/

MPI Model and Basic Calls

MPI: Basics

Every MPI program must contain the preprocessor directive **`#include <mpi.h>`**

The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.

mpi.h is usually found in the “include” directory of most MPI installations:

```
...
#include "mpi.h"
...
MPI_Init(&Argc,&Argv);
...
...
MPI_Finalize();
...
```

MPI: Initializing MPI Environment

Function: MPI_init()

int MPI_Init(int *argc, char ***argv)

Description:

Initializes the MPI execution environment. MPI_init() must be called before any other MPI functions can be called and it should be called only once. It allows systems to do any special setup so that MPI Library can be used. ***argc*** is a pointer to the number of arguments and ***argv*** is a pointer to the argument vector. On exit from this routine, all processes will have a copy of the argument list.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Init.html

MPI: Terminating MPI Environment

Function:	MPI_Finalize()

int MPI_Finalize()

Description:

Terminates MPI execution environment. All MPI processes must call this routine before exiting. MPI_Finalize() need not be the last executable statement or even in main; it must be called at somepoint following the last call to any other MPI function.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Finalize.html

MPI Hello World

C source file for a simple MPI Hello World

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

int main( int argc, char *argv[])
{
    MPI_Init( &argc, &argv);
    printf("Hello, World!\n");
    MPI_Finalize();
    return 0;
}
```

Include header files

Initialize MPI Context

Finalize MPI Context

Building an MPI Executable

Library version

User knows where header file and library are, and tells compiler

gcc -Iheaderdir -Llibdir mpicode.c –lmpich

Wrapper version

Does the same thing, but hides the details from the user

mpicc -o executable mpicode.c

You can do either one, but don't try to do both!

use "sh -x mpicc -o executable mpicode.c" *to figure out the gcc line*

Running an MPI Executable

Some number of processes are started somewhere

- Again, standard doesn't talk about this
- Implementation and interface varies
- Usually, some sort of mpiexec command starts some number of copies of an executable according to a mapping
- Example:
  - *'mpirun -n 2 ./a.out' command runs two copies of ./a.out*
- Most production supercomputing resources wrap the mpirun command with higher level scripts that interact with scheduling systems such as PBS, SLURM, or LoadLeveler for efficient resource management and multi-user support

Look online for batch system specific job submission scripts

MPI Communicators

MPI Communicators

Communicator is an internal object

MPI Programs are made up of communicating processes

Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)

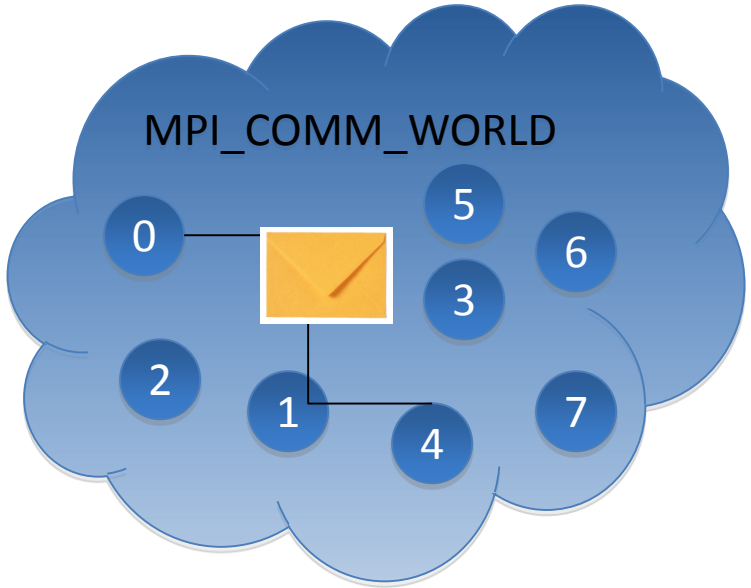
MPI provides functions to interact with it

Default communicator is MPI_COMM_WORLD

All processes are its members
- It has a size (the number of processes)
- Each process has a rank within it
- One can think of it as an ordered list of processes
- Additional communicator(s) can co-exist

A process can belong to more than one communicator

Within a communicator, each process has a unique rank



MPI: Size of Communicator

Function: MPI_Comm_size()

int MPI_Comm_size (MPI_Comm comm, int *size)

Description:

Determines the size of the group associated with a communicator (comm). Returns an integer number of processes in the group underlying *comm* executing the program. If *comm* is an inter-communicator (i.e. an object that has processes of two inter-communicating groups) , return the size of the local group (a size of a group where request is initiated from). The *comm* in the argument list refers to the communicator-group to be queried, the result of the query (size of the *comm* group) is stored in the variable size.

```
...
#include "mpi.h"
...
int size;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

MPI: Rank of a Process in Communicator

Function: MPI_Comm_rank()

int MPI_Comm_rank (MPI_Comm comm, int *rank)

Description:

Returns the rank of the calling process in the group underlying the comm. If the *comm* is an inter-communicator, the call MPI_Comm_rank returns the rank of the process in the local group. The first parameter *comm* in the argument list is the communicator to be queried, and the second parameter *rank* is the integer number rank of the process in the group of *comm*.

```
...
#include "mpi.h"
...
int rank;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

Example: communicators

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

int main( int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello, World! from %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Determines the rank of the current process in the communicator-group MPI_COMM_WORLD

Determines the size of the communicator-group MPI_COMM_WORLD

```
…
Hello, World! from 1 of 8
Hello, World! from 0 of 8
Hello, World! from 5 of 8
…
```

Point to Point Communication

MPI: Point to Point Communication

A basic communication mechanism of MPI between a pair of processes in which one process is sending data and the other process receiving the data, is called “point to point communication”

Message passing in MPI program is carried out by 2 main MPI functions

- MPI_Send – sends message to a designated process
- MPI_Recv – receives a message from a process

Each of the send and recv calls is appended with additional information along with the data that needs to be exchanged between application programs

The message envelope consists of the following information

- The rank of the receiver
- The rank of the sender
- A tag
- A communicator

The source argument is used to distinguish messages received from different processes

Tag is user-specified int that can be used to distinguish messages from a single process

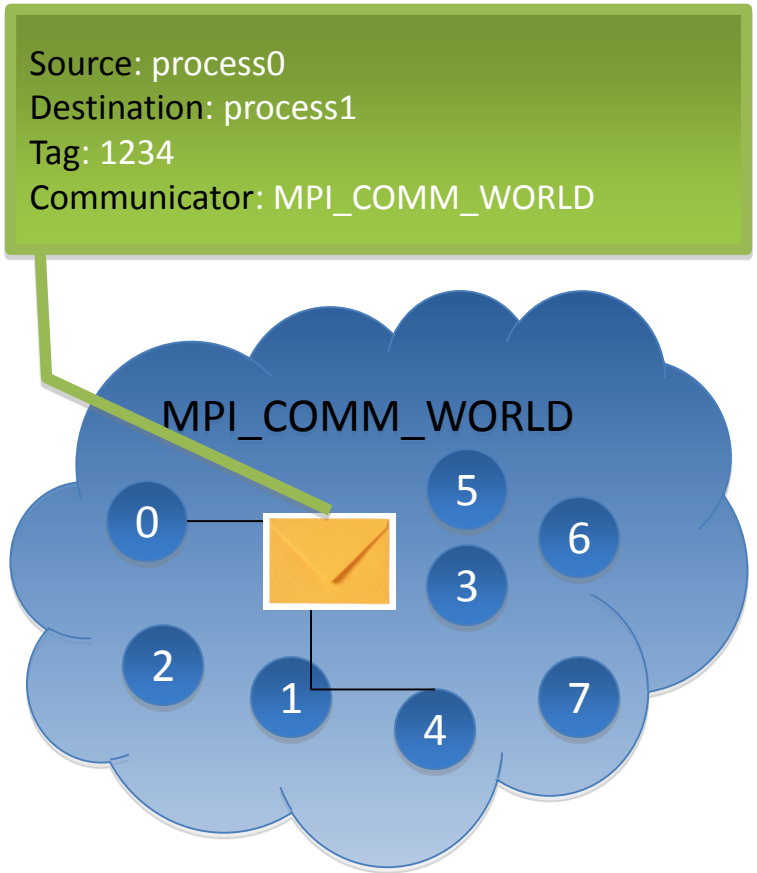
Message Envelope

Communication across processes is performed using messages.

Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :

- Envelope comprises source, destination, tag, communicator
- Message = Envelope + Data

Communicator refers to the namespace associated with the group of related processes



MPI: (blocking) Send message

Function: MPI_Send()

```
int MPI_Send(
        void            *message,
        int             count,
        MPI_Datatype    datatype,
        int             dest,
        int             tag,
        MPI_Comm        comm )
```

Description:

The contents of message are stored in a block of memory referenced by the first parameter *message*. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype.* MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage an overflow error occurs. The *dest* parameter corresponds to the rank of the process to which message has to be sent.

MPI : Data Types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs

MPI: (blocking) Receive message

Function: MPI_Recv()

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

Description:

The contents of message are stored in a block of memory referenced by the first parameter *message*. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype.* MPI allows a message to be received as long as there is sufficient storage allocated. If there isnt sufficient storage an overflow error occurs. The *source* parameter corresponds to the rank of the process from which the message has been received. The MPI_Status parameter in the MPI_Recv() call returns information on the data that was actually received It references a record with 2 fields – one for the source and one for the tag

MPI_Status object

Object: MPI_Status

Example usage :

```
MPI_Status    status;
```

Description:

The MPI_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI_SOURCE), the message tag (MPI_TAG), and error status (MPI_ERROR) .

```
#include "mpi.h"
…
MPI_Status status; /* return status for   */
…
MPI_Init(&argc, &argv);
…
if (my_rank != 0) {
…
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  else { /* my rank == 0 */
    for (source = 1; source < p; source++ ) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
…
  MPI_Finalize();
…
```

MPI_Status object

Object: MPI_Status

Example usage :

```
MPI_Status    status;
```

Description:

The MPI_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI_SOURCE), the message tag (MPI_TAG), and error status (MPI_ERROR) .

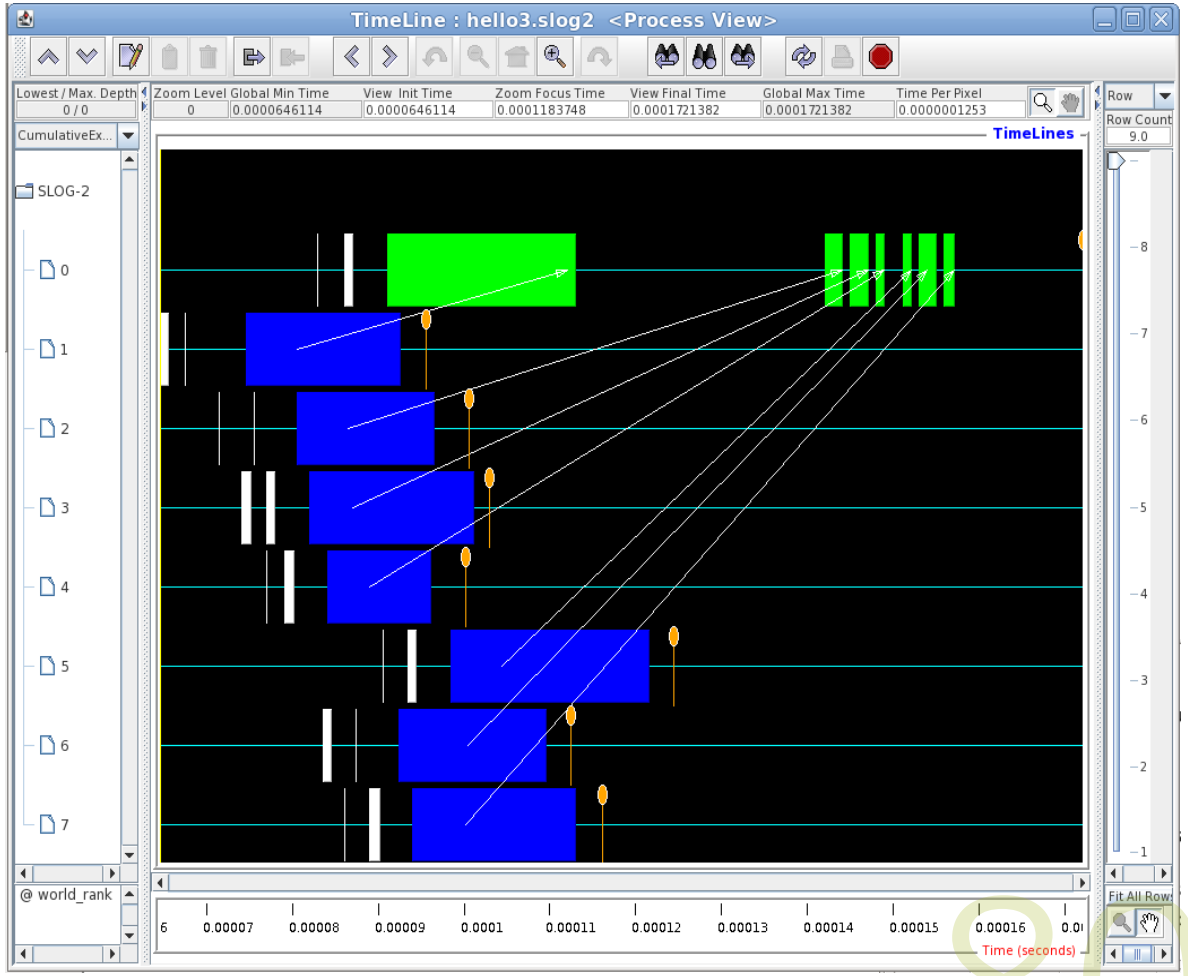
```
#include "mpi.h"

MPI_Status status; /* return status for   */

MPI_Init(&argc, &argv);

if (my_rank != 0) {
   MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
 }
 else { /* my rank == 0 */
   for (source = 1; source < p; source++ )
      MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
 MPI_Finalize();
```

MPI : Example send/recv

```
/* hello world, MPI style */

#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
  int my_rank;       /* rank of process     */
  int p;             /* number of processes */
  int source;        /* rank of sender      */
  int dest;          /* rank of receiver    */

  int tag=0;         /* tag for messages    */
  char message[100]; /* storage for message */
  MPI_Status status; /* return status for   */
                     /* receive             */

  /* Start up MPI */
  MPI_Init(&argc, &argv);

  /* Find out process rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  /* Find out number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (my_rank != 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    /* Use strlen+1 so that \0 gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
      MPI_COMM_WORLD);
  }
  else { /* my rank == 0 */
    for (source = 1; source < p; source++ ) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &status);
      printf("%s\n", message);
    }
    printf("Greetings from process %d!\n", my_rank);
  }

  /* Shut down MPI */
  MPI_Finalize();

} /* end main */
```

Communication Map for the Example.

```
mpirun -n 8 ./mpi_send_recv
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
Greetings from process 0!
```



Deadlocks

Deadlock

Something to avoid

A situation where the dependencies between processors are cyclic

- One processor is waiting for a message from another processor, but that processor is waiting for a message from the first, so nothing happens
  - *Until your time in the queue runs out and your job is killed*
  - *MPI does not have timeouts*

Deadlock Example

```
if (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}
else {
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
}
```

If the message sizes are small enough, this should work because of systems buffers

If the messages are too large, or system buffering is not used, this will hang

Deadlock Example Solutions

```
if (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}
else {
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
}
```

Trapezoidal Rule

An Example

Numerical Integration Using Trapezoidal Rule: A Case Study

In review, the 6 main MPI calls:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv

Using these 6 MPI function calls we can begin to construct several kinds of parallel applications

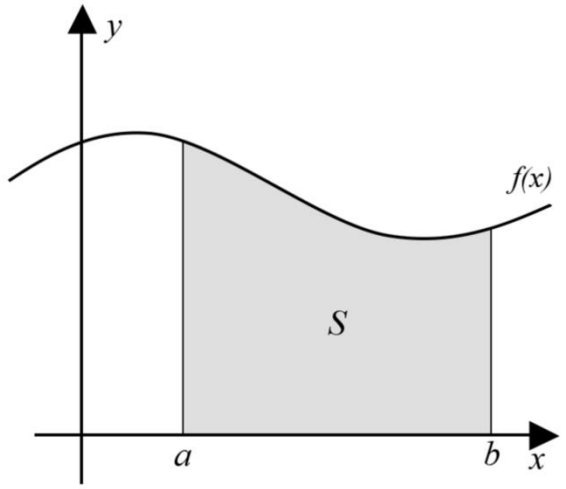
In the following section we discuss how to use these 6 calls to parallelize Trapezoidal Rule

Approximating Integrals: Definite Integral

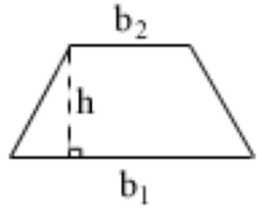
Problem: to find an approximate value to a definite integral

$$\int_a^b f(x)\,dx.$$

A definite integral from a to b of a non negative function f(x) can be thought of as the area bound by the X-axis, the vertical lines x=a and x=b, and the graph of f(x)



Approximating Integrals : Trapezoidal Rule



$$\frac{h(b_1+b_2)}{2}$$

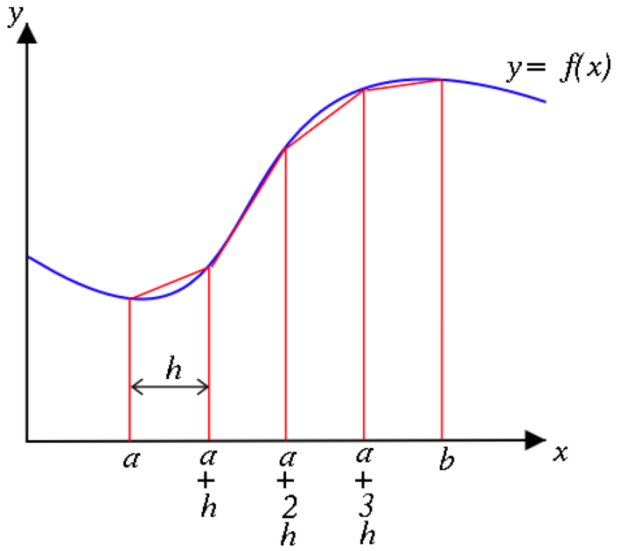
Approximating area under the curve can be done by dividing the region under the curve into regular geometric shapes and then adding the areas of the shapes.

In Trapezoidal Rule, the region between a and b can be divided into n trapezoids of base: h = (b-a)/n

The area of a trapezoid can be calculated as $\frac{h(b_1+b_2)}{2}$

In the case of our function the area for the first block can be represented as

$$\frac{h(f(a)+f(a+h))}{2}$$



The area under the curve bounded by a & b can be approximated as :

$$\left[\frac{h(f(a)+f(a+h))}{2}\right]+\left[\frac{h(f(a+h)+f(a+2h))}{2}\right]+\left[\frac{h(f(a+2h)+f(a+3h))}{2}\right]+\left[\frac{h(f(a+3h)+f(b))}{2}\right]$$

Approximating Integrals: Trapezoid Rule

We can further generalize this concept of approximation of integrals as a summation of trapezoidal areas

$$\frac{1}{2}h[f(x_0)+f(x_1)]+\frac{1}{2}h[f(x_1)+f(x_2)]+\ldots+\frac{1}{2}h[f(x_{n-1})+f(x_n)]$$
$$=\frac{h}{2}[f(x_0)+f(x_1)+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+f(x_n)]$$
$$=\frac{h}{2}[f(x_0)+2f(x_1)+2f(x_2)+\ldots+2f(x_{n-1})+f(x_n)]$$
$$=h\left[\frac{f(x_0)}{2}+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+\frac{f(x_n)}{2}\right]$$

Trapezoidal Rule – Sequential COde

```
/* serial.c -- serial trapezoidal rule
 *
 * Calculate definite integral using trapezoidal rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *    using n trapezoids.
 */

#include <stdio.h>

main() {
    float  integral;   /* Store result in integral   */
    float  a, b;       /* Left and right endpoints   */
    int    n;          /* Number of trapezoids       */
    float  h;          /* Trapezoid base width       */
    float  x;
    int    i;
    float f(float x);  /* Function we're integrating */

    printf("Enter a, b, and n\n");
    scanf("%f %f %d", &a, &b, &n);
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i <= n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n",
        a, b, integral);
} /* main */


float f(float x)
{
    return x * x;
} /* f */
```

Results for the Serial Trapezoidal Rule

a	b	n	f(x) single precision	f(x) double precision
2	25	1	7233.500000	7233.500000
2	25	2	5712.625000	5712.625000
2	25	10	5225.945312	5225.945000
2	25	30	5207.916992	5207.919815
2	25	40	5206.934082	5206.934062
2	25	50	5206.475098	5206.477800
2	25	1000	5205.664551	5205.668694

```
Enter a, b, and n
2 25 5000
With n = 5000 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.550293
```

acheco Ch 4

Parallelizing Trapezoidal Rule

One way of parallelizing Trapezoidal rule :

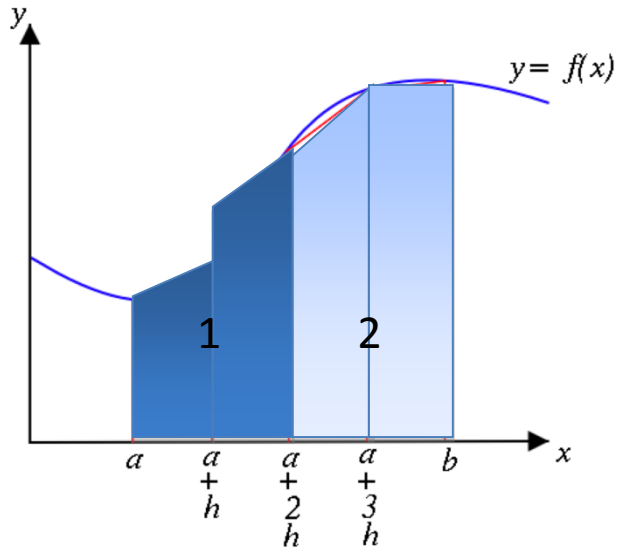
- Distribute chunks of workload (each workload characterized by its own subinterval of [a,b] to each process)
- Calculate f for each subinterval
- Finally add the f calculated for all the sub intervals to produce result for the complete problem [A,B]

Issues to consider

- Number of trapezoids (n) are equally divisible across (p) processes (load balancing).
- First process calculates the area for the first n/p trapezoids, second process calculates the area for the next n/p trapezoids and so on

Key information related to the problem that each process needs is the

- Rank of the process
- Ability to derive the workload per processor as a function of rank
- Assumption: Process 0 does the summation



Parallelizing Trapezoidal Rule

Assumption: Number of trapezoids n is evenly divisible across p processors

Calculate: $$h = \frac{(b-a)}{n}$$

Each process calculates its own workload (interval to integrate)

local number of trapezoids (local_n) = n/p

*local starting point (local_a) = a+(process_rank *local_n* h)*

*local ending point (local_b) = (local_a + local_n * h)*

Each process calculates its own integral for the local intervals

For each of the local_n trapezoids calculate area

Aggregate area for local_n trapezoids

If PROCESS_RANK == 0

Receive messages (containing sub-interval area aggregates) from all processors

Aggregate (ADD) all sub-interval areas

If PROCESS_RANK > 0

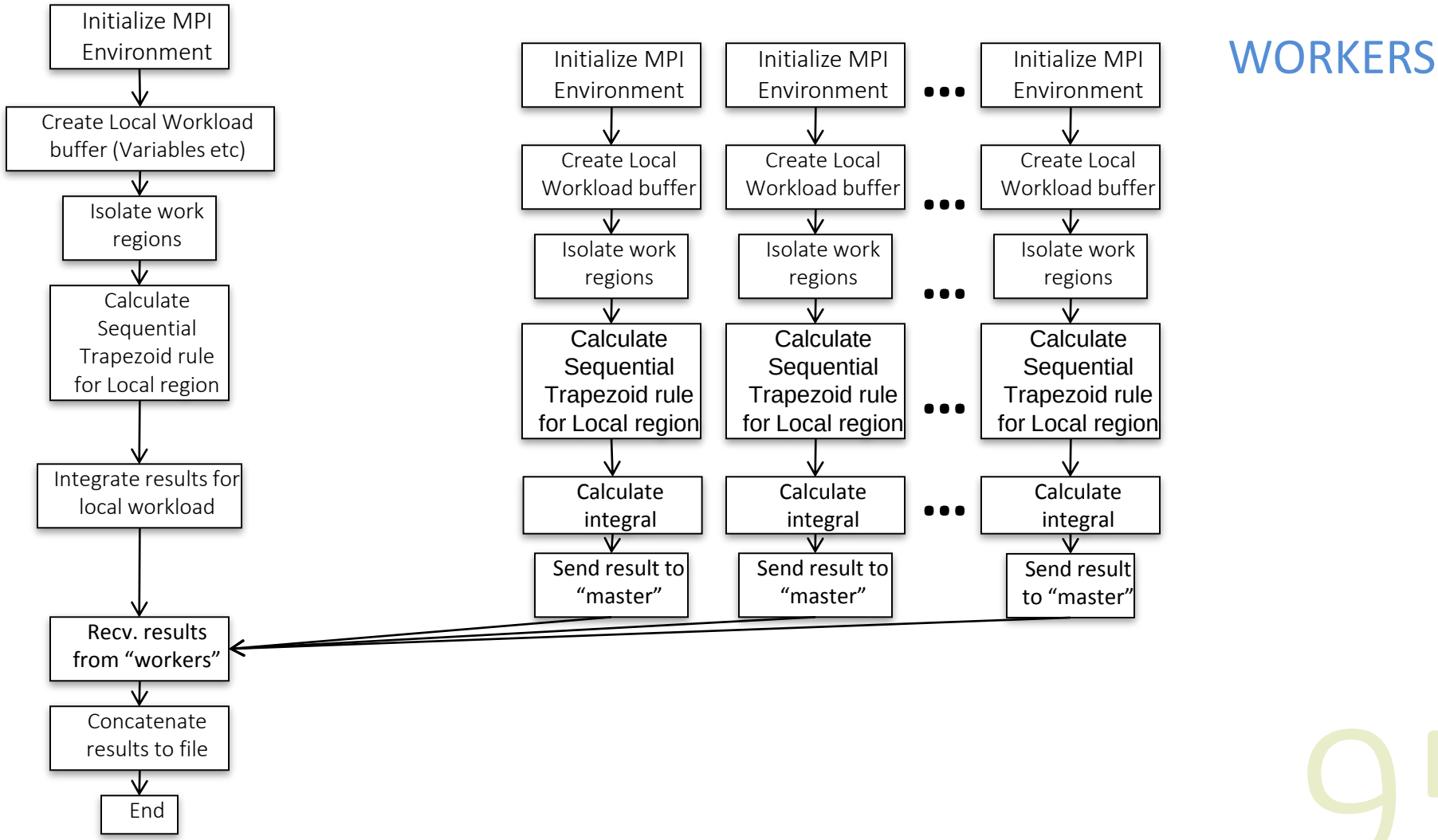
Send sub-interval area to PROCESS_RANK(0)

Classic SPMD: all processes run the same program on different datasets.

Parallel Trapezoidal Rule

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

float Trap(float local_a, float local_b, int local_n,
              float h);    /* Calculate local integral  */

main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for my calculation           */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest = 0;  /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;
```

Flowchart for Parallel Trapezoidal Rule



Parallel Trapezoidal Rule

```
/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = (b-a)/n;    /* h is the same for all processes */
local_n = n/p;  /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

Parallel Trapezoidal Rule

```
 /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
    }
    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }
    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

Parallel Trapezoidal Rule

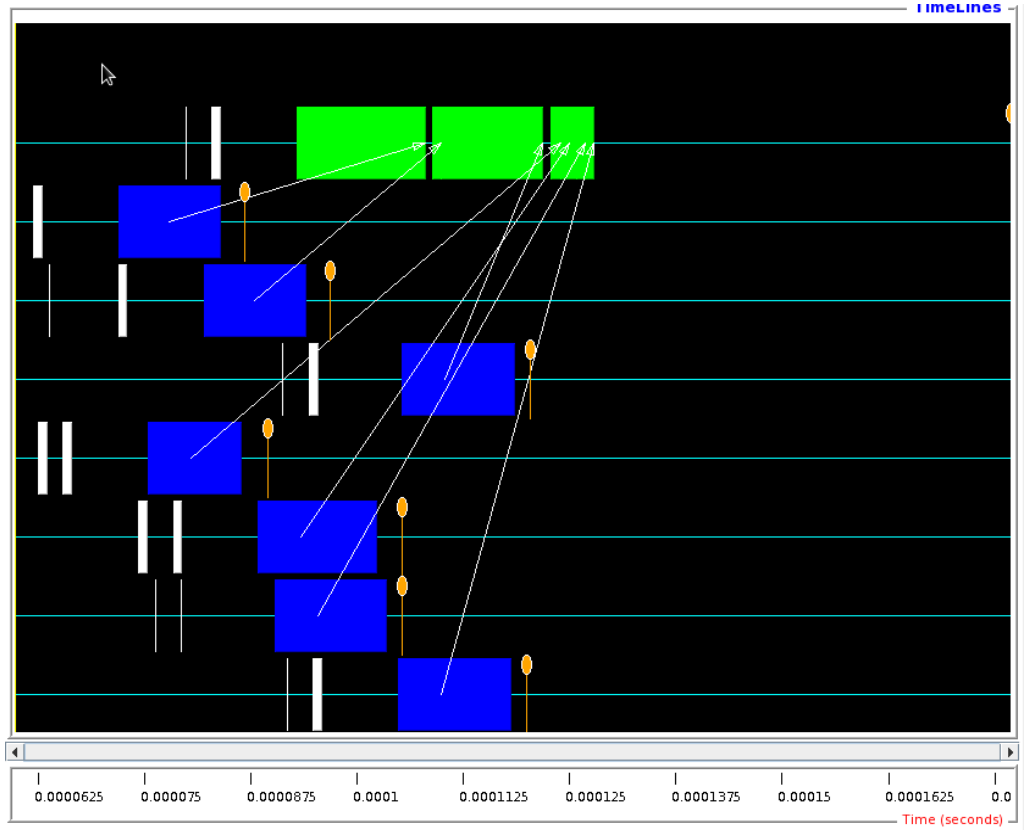
```
float Trap(
          float  local_a   /* in */,
          float  local_b   /* in */,
          int    local_n   /* in */,
          float  h         /* in */) {

    float integral;   /* Store result in integral  */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */
```

```
float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

Parallel Trapezoidal Rule

mpirun -n 8 … ./trap
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000
= 5205.667969

./serial
Enter a, b, and n
2 25 1024
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000
= 5205.666016



MPI Collective Calls: Synchronization Primitives

Collective Calls

A communication pattern that encompasses all processes within a communicator is known as

collective communication

MPI has several collective communication calls, the most frequently used are:

- Synchronization
  - *Barrier*
- Communication
  - *Broadcast*
  - *Gather & Scatter*
  - *All Gather*
- Reduction
  - *Reduce*
  - *AllReduce*

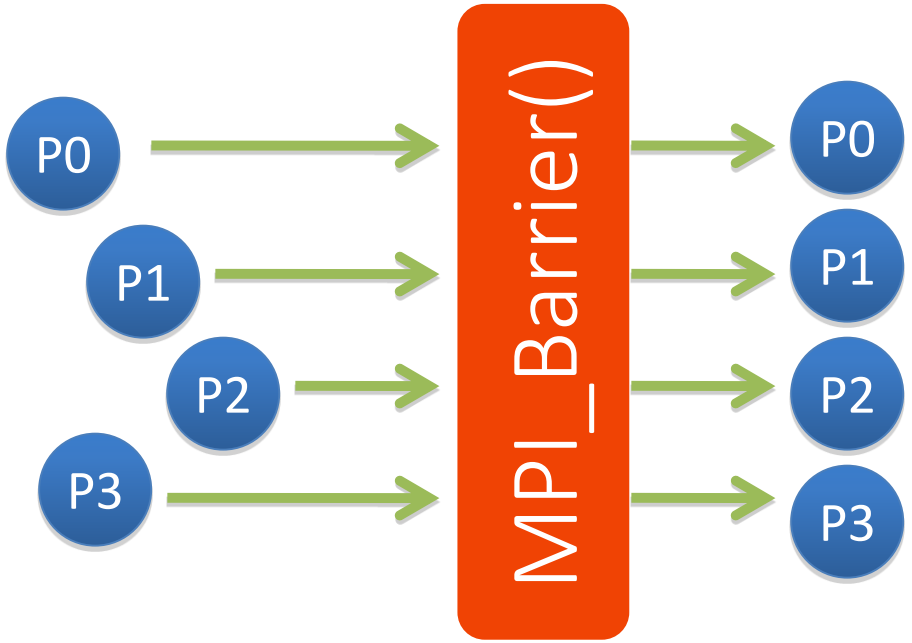
MPI Collective Calls : Barrier

Function: MPI_Barrier()

```
int MPI_Barrier (
        MPI_Comm comm )
```

Description:

Creates barrier synchronization in a communicator group *comm*. Each process, when reaching the MPI_Barrier call, blocks until all the processes in the group reach the same MPI_Barrier call.



Example: MPI_Barrier()

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

int main (int argc, char *argv[]) {
      int           rank, size, len;
      char          name[MPI_MAX_PROCESSOR_NAME];
      MPI_Init(&argc, &argv);
      MPI_Barrier(MPI_COMM_WORLD);

      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Get_processor_name(name, &len);
      MPI_Barrier(MPI_COMM_WORLD);

      printf ("Hello world! Process %d of %d on %s\n", rank,
        size, name);
      MPI_Finalize();
      return 0;
}
```

```
mpirun -np 8 barrier
Hello world! Process 0 of 8 on celeritas.cct.lsu.edu
Hello world! Process 4 of 8 on compute-0-3.local
Hello world! Process 1 of 8 on compute-0-0.local
Hello world! Process 3 of 8 on compute-0-2.local
Hello world! Process 6 of 8 on compute-0-5.local
Hello world! Process 7 of 8 on compute-0-6.local
Hello world! Process 5 of 8 on compute-0-4.local
Hello world! Process 2 of 8 on compute-0-1.local
```

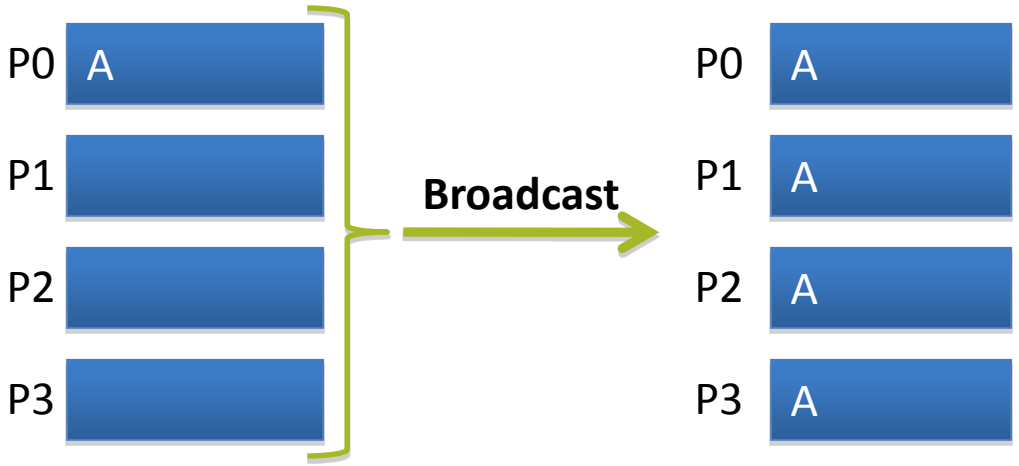
MPI Collective Calls: Communication Primitives

MPI Collective Calls: Broadcast

Function:	MPI_Bcast()

```
int MPI_Bcast (
      void          *message,
      int           count,
      MPI_Datatype  datatype,
      int           root,
      MPI_Comm      comm )
```

Description: A collective communication call where a single process sends the same data contained in the *message* to every process in the communicator. By default a tree like algorithm is used to broadcast the message to a block of processors, a linear algorithm is then used to broadcast the message from the first process in a block to all other processes. All the processes invoke the *MPI_Bcast* call with the same arguments for root and *comm*,



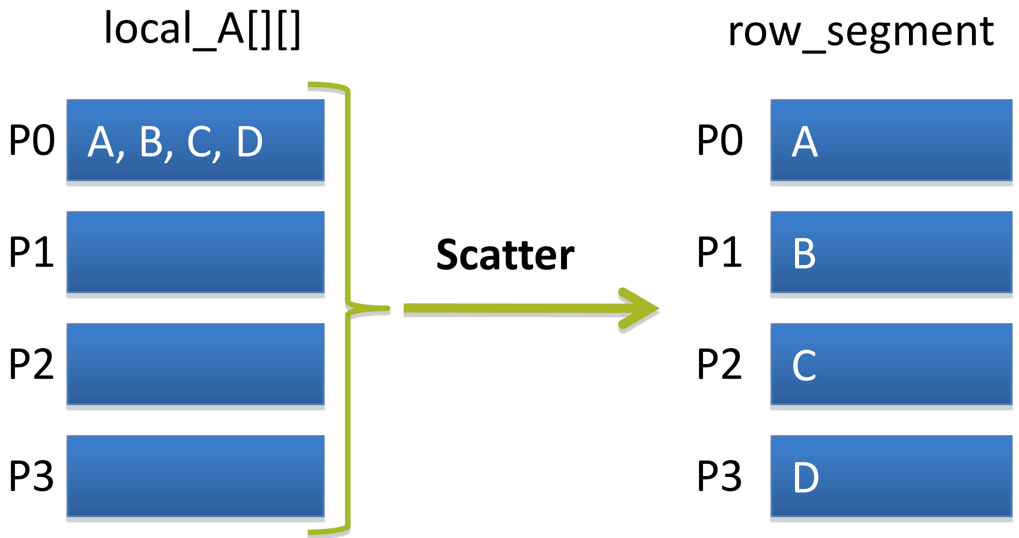

```
 float endpoint[2] = { 1.f, 2.f };
...
 MPI_Bcast(endpoint, 2, MPI_FLOAT, 0,
    MPI_COMM_WORLD);
...
```

MPI Collective Calls: Scatter

Function:	MPI_Scatter()

```
int MPI_Scatter (
        void            *sendbuf,
        int             send_count,
        MPI_Datatype    send_type,
        void            *recvbuf,
        int             recv_count,
        MPI_Datatype    recv_type,
        int             root,
        MPI_Comm        comm)
```

Description: MPI_Scatter splits the data referenced by the *sendbuf* on the process with rank root into *p* segments each of which consists of *send_count* elements of type *send_type*. The first segment is sent to process0 and the second segment to process1.The send arguments are significant on the process with rank *root*.



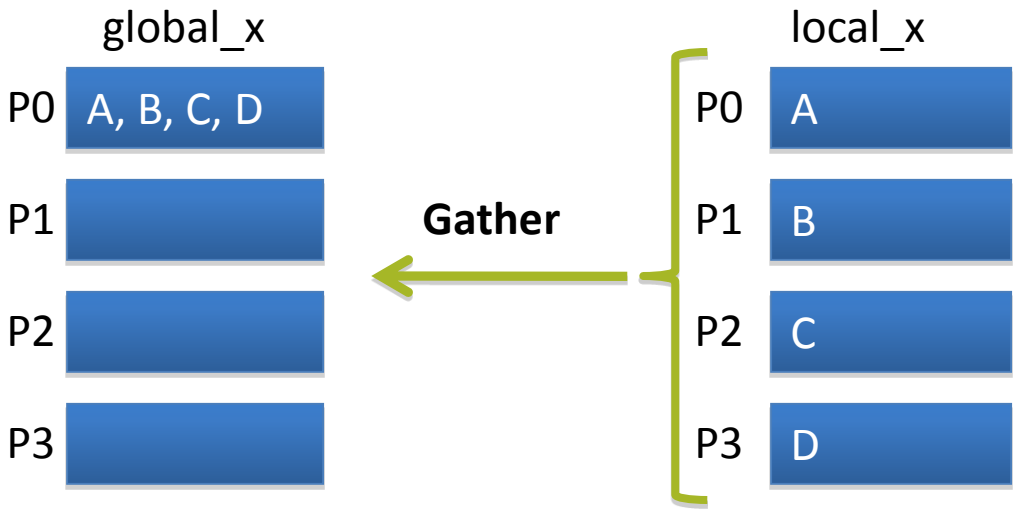

```
...
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT,
    row_segment, n/p, MPI_FLOAT, 0,MPI_COMM_WORLD);
...
```

MPI Collective Calls: Gather

Function:	MPI_Gather()

```
int MPI_Gather (
    void          *sendbuf,
    int           send_count,
    MPI_Datatype  sendtype,
    void          *recvbuf,
    int           recvcount,
    MPI_Datatype  recvtype,
    int           root,
    MPI_Comm      comm )
```

Description: MPI_Gather collects the data referenced by *sendbuf* from each process in the communicator *comm,* and stores the data in process rank order on the process with rank *root* in the location referenced by *recvbuf*. The *recv* parameters are only significant on global_x.




```
...
  MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p,
    MPI_FLOAT, 0, MPI_COMM_WORLD);
...
```

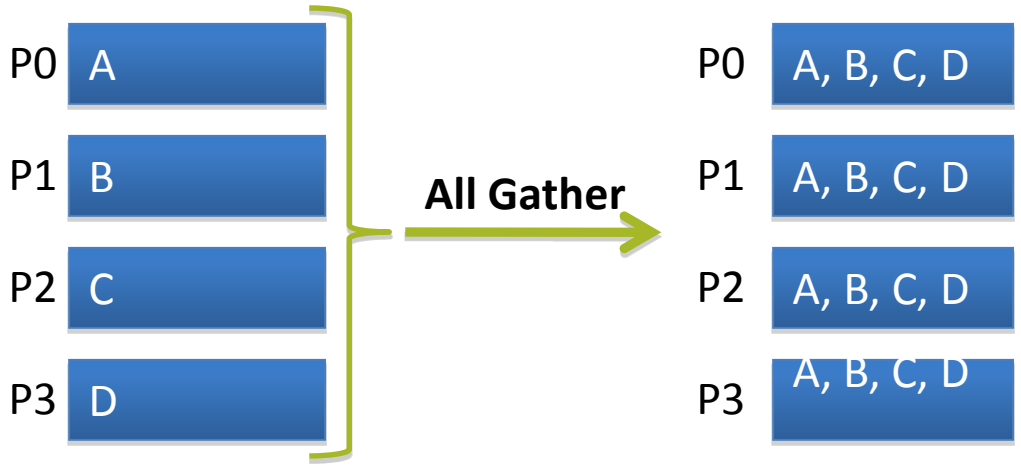
MPI Collective Calls: All Gather

Function: MPI_Allgather()

```
int MPI_Allgather (
    void          *sendbuf,
    int           send_count,
    MPI_Datatype  sendtype,
    void          *recvbuf,
    int           recvcount,
    MPI_Datatype  recvtype,
    MPI_Comm      comm )
```

Description:

MPI_Allgather gathers the content from the send buffer (*sendbuf)* on each process. The effect of this call is similar to executing MPI_Gather() p times with a different process acting as the root.




```
 for (root=0; root<p; root++)
    MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p,
       MPI_FLOAT, root, MPI_COMM_WORLD);
...
```

CAN BE REPLACED WITH :

```
MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x,
   local_n, MPI_FLOAT, MPI_COMM_WORLD);
```

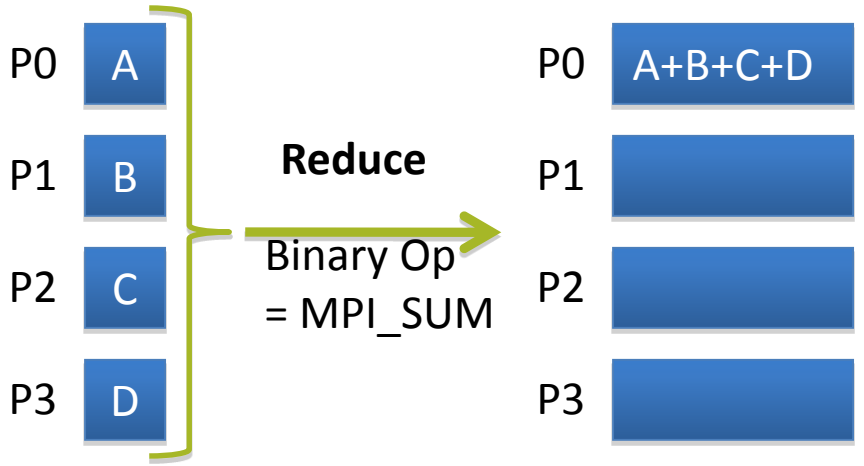
MPI Collective Calls: Reduction Primitives

MPI Collective Calls: Reduce

Function: MPI_Reduce()

```
int MPI_Reduce (
      void              *operand,
      void              *result,
      int               count,
      MPI_Datatype      datatype,
      MPI_Op            operator,
      int               root,
      MPI_Comm          comm)
```

Description: A collective communication call where all the processes in a communicator contribute data that is combined using binary operations (MPI_Op) such as addition, max, min, logical, and, etc. MPI_Reduce combines the operands stored in the memory referenced by *operand* using the operation *operator* and stores the result in **result*. MPI_Reduce is called by all the processes in the communicator *comm* and for each of the processes *count*, *datatype operator* and *root* remain the same.




```
...
MPI_Reduce(&local_integral, &integral, 1,
MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
...
```

MPI Binary Operations

MPI binary operators are used in the MPI_Reduce function call as one of the parameters. MPI_Reduce performs a global reduction operation (dictated by the MPI binary operator parameter) on the supplied operands.

Some of the common MPI Binary Operators used are:

```
MPI_Reduce(&local_integral,
  &integral, 1, MPI_FLOAT,
   MPI_SUM, 0, MPI_COMM_WORLD);
```

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwise Or
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR
MPI_MAXLOC	Maximum and location of max.
MPI_MINLOC	Maximum and location of min.

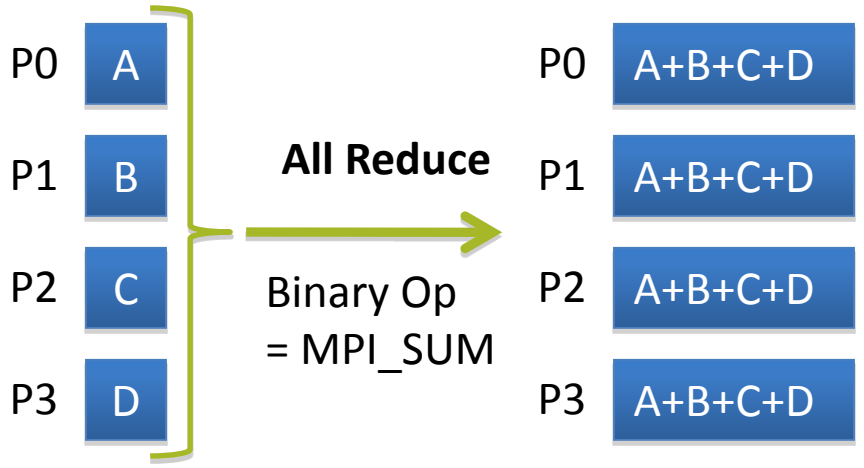
MPI Collective Calls: All Reduce

Function: MPI_Allreduce()

```
int MPI_Allreduce (
        void            *sendbuf,
        void            *recvbuf,
        int             count,
        MPI_Datatype    datatype,
        MPI_Op          op,
        MPI_Comm        comm )
```

Description:

MPI_Allreduce is used exactly like MPI_Reduce, except that the result of the reduction is returned on all processes, as a result there is no *root* parameter.




```
...
MPI_Allreduce(&integral, &integral, 1,
   MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Allreduce.html