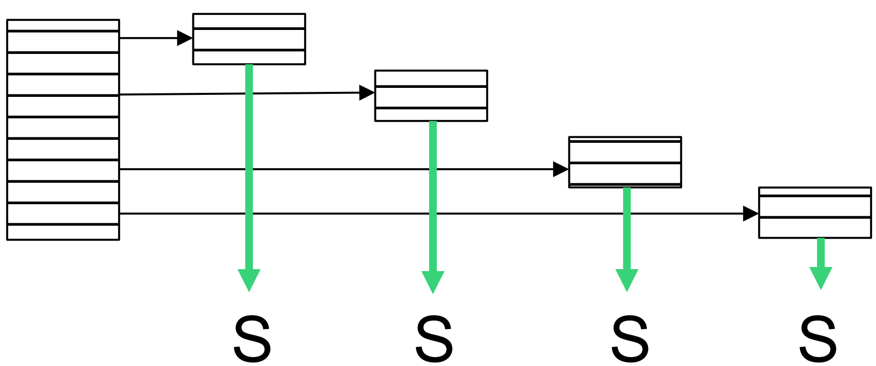
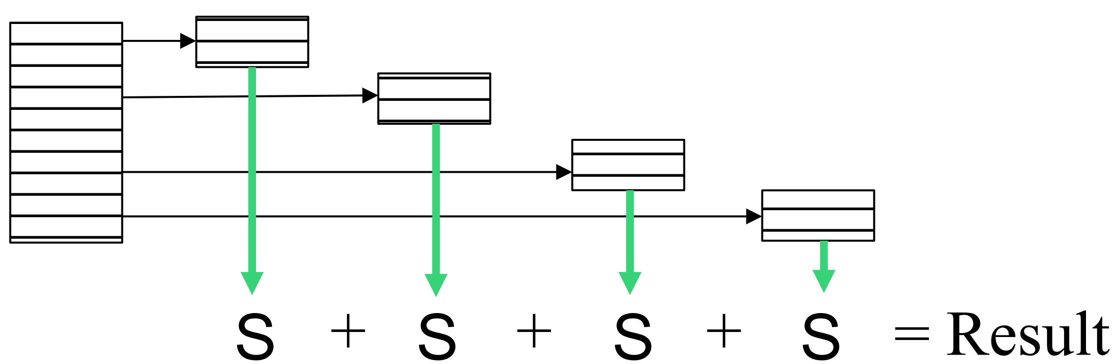
What is Parallel Computing?

- A mechanism for <u>speeding up</u> computation
- Multiple processes work together to solve a problem
- Multiple processors allow multiple processes to run at the same time (in parallel)
- Example: compute sum of 1M numbers
  - with 4 processors - runs 4 times faster!



Interesting Parallel Programs Need Communication

- Summarize or combine results
- Propagate updates across processors
- Distribute work to processors
- Handle boundary conditions



Two Ways to Communicate

- Pass Messages
  - Explicitly send
  - Explicitly receive
- Share Memory
  - Read and write shared variables
    - Data implicitly passed between processes
  - Explicitly control access to shared variables
    - Prevent inconsistent state
    - Prevent race conditions

Message Passing Semantics

- Primary functions:
  - Send Data (what data, where to send it)
  - Receive Data (who to receive from, where to put it)
- Many subtle shades to consider:
  - synchronization
  - buffering
  - naming
  - data size and type

Synchronization

- *blocking* - operation might block until other task makes progress
- *non-blocking* - operation will not block, but might fail if it must otherwise wait
- *asynchronous* - occurs “in the background” concurrently with main thread
- *synchronous* - requires that both sender and receiver read send/receive before either can complete

Buffering

- *No buffering* - requires synchronous comm
- *Infinite buffering* - make non-blocking
- *Partial buffering* - might block, might not
- *Explicit buffering* - user guarantees enough buffer (thus non-blocking)

Naming

- *Direct* - processes named each other directly
- *Indirect* - send and receive via a “mailbox”
- *Symbolic* - processes refer to each other with a symbol or logical number
- *Symmetric* - both processes must name the other
- *Asymmetric* - sender must name destination, receiver receives sender's ID

Data Size and Type

- Fixed message size
- Variable message size
- Infinite data stream
- Bytes only
- Complex types
- Non-contiguous access

Collective Message Passing

- Involves a group of processes
  - manage process naming
  - manage process groups
- Compute while communicating
  - Summarizing
  - Searching
- Re-arrange data
  - Distribute data
  - Gather data
  - Move data around

Message Passing Systems

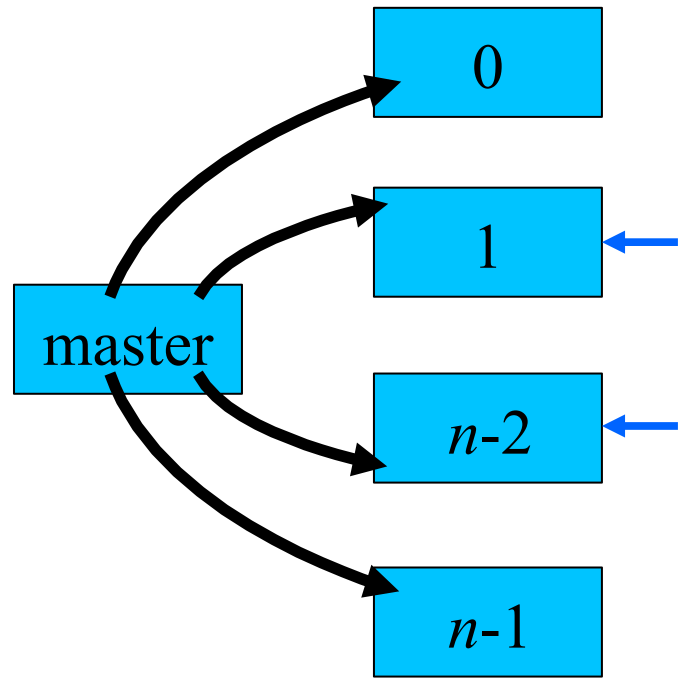
- TCP/IP
- UDP/IP
- GM (Myrinet)
- VIA
- PVM
- MPI
  - MPI 1.1
  - MPI 2.0

The MPI Interface

- MPI is an *interface* standard
  - Is **not** a specific implementation
  - Does not specify much about processes
- MPI designed for parallel computing
  - Not very good for general purpose messaging
- Two “levels” of implementation:
  - MPI 1.1: basic level
  - MPI 2.0: more advanced features

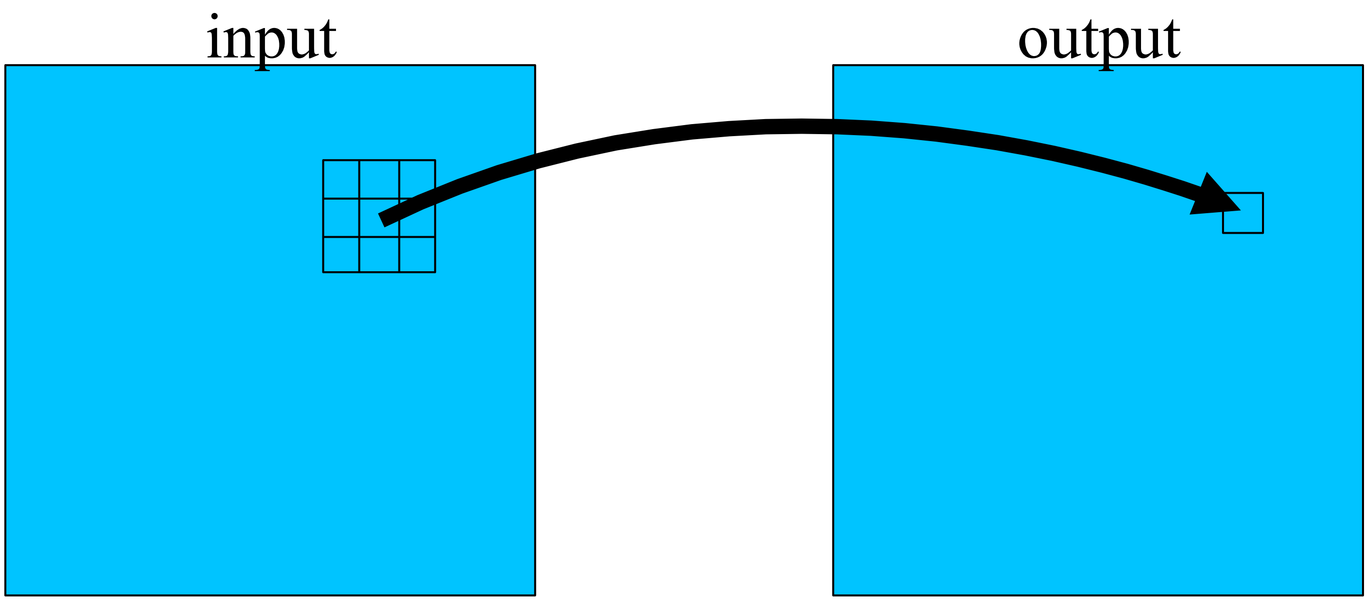
An MPI Job

- A Job creates *n* copies of your program (tasks)
- One process stays on the master node to manage IO
- Tasks can send/receive messages to/from the other tasks
- Each task as a RANK and knows SIZE (n)



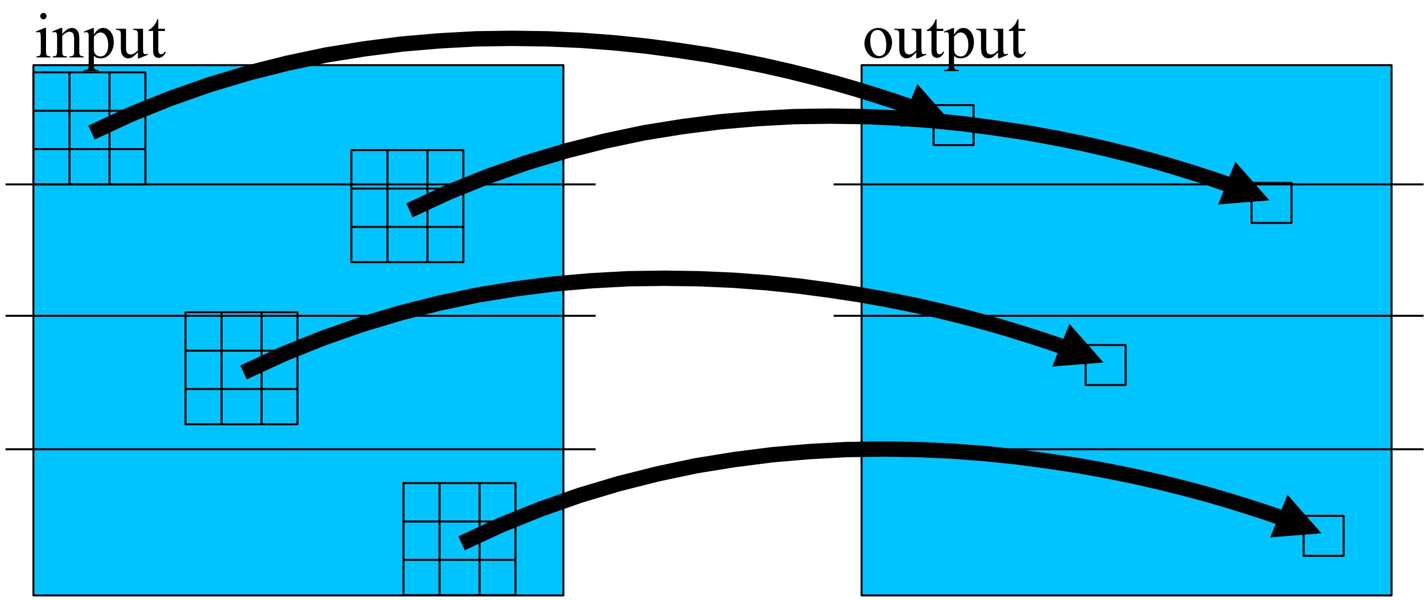
Example - Image Smoothing

- Image data is modified to reduce noise
- Each pixel replaced by the average of the 8 surrounding pixels, and itself



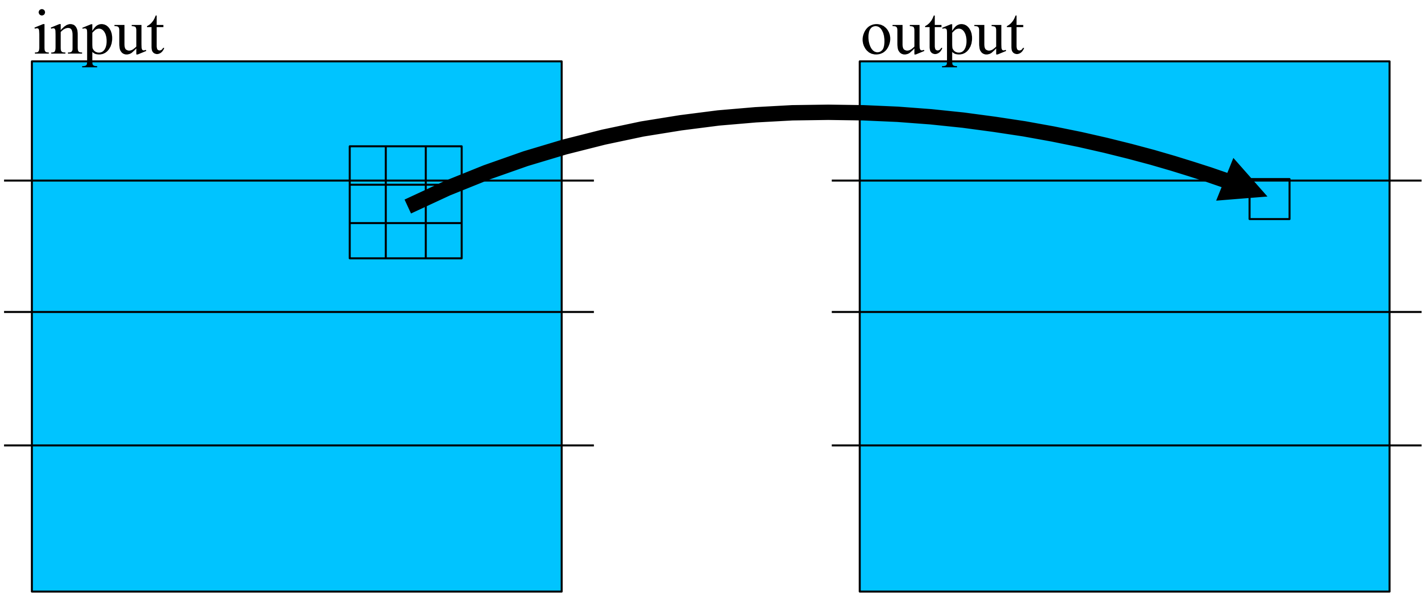
Parallel Smoothing

- Image data divided among tasks
- Each task smooths its portion



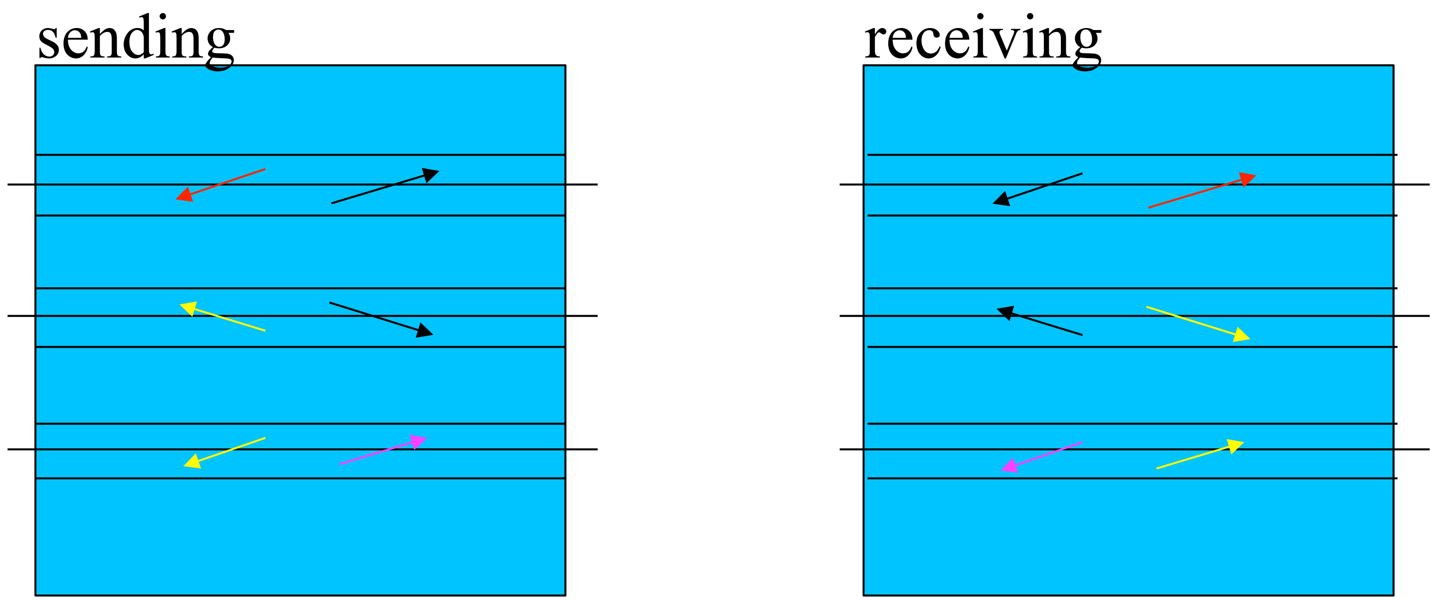
Border pixels need overlapping data

- Data is required from the other tasks
- Other tasks require data as well



Tasks exchange border data

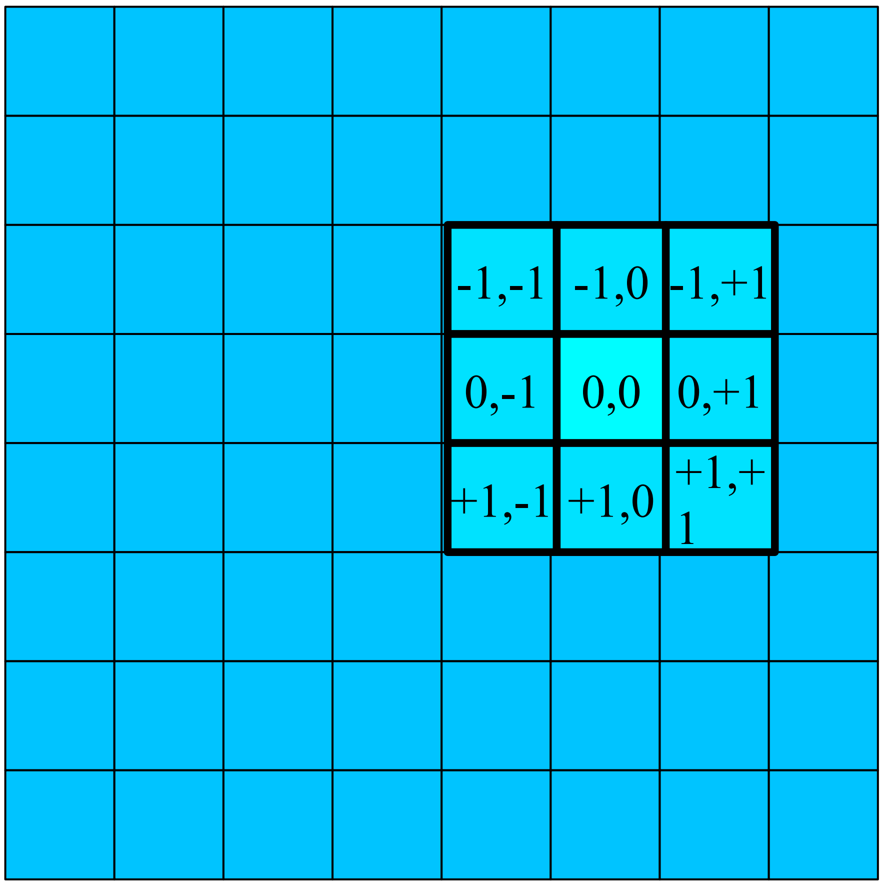
- Each task sends to the other tasks
- Each task receives from the other tasks
- When more tasks, each exchanges with 8 adjacent neighbors



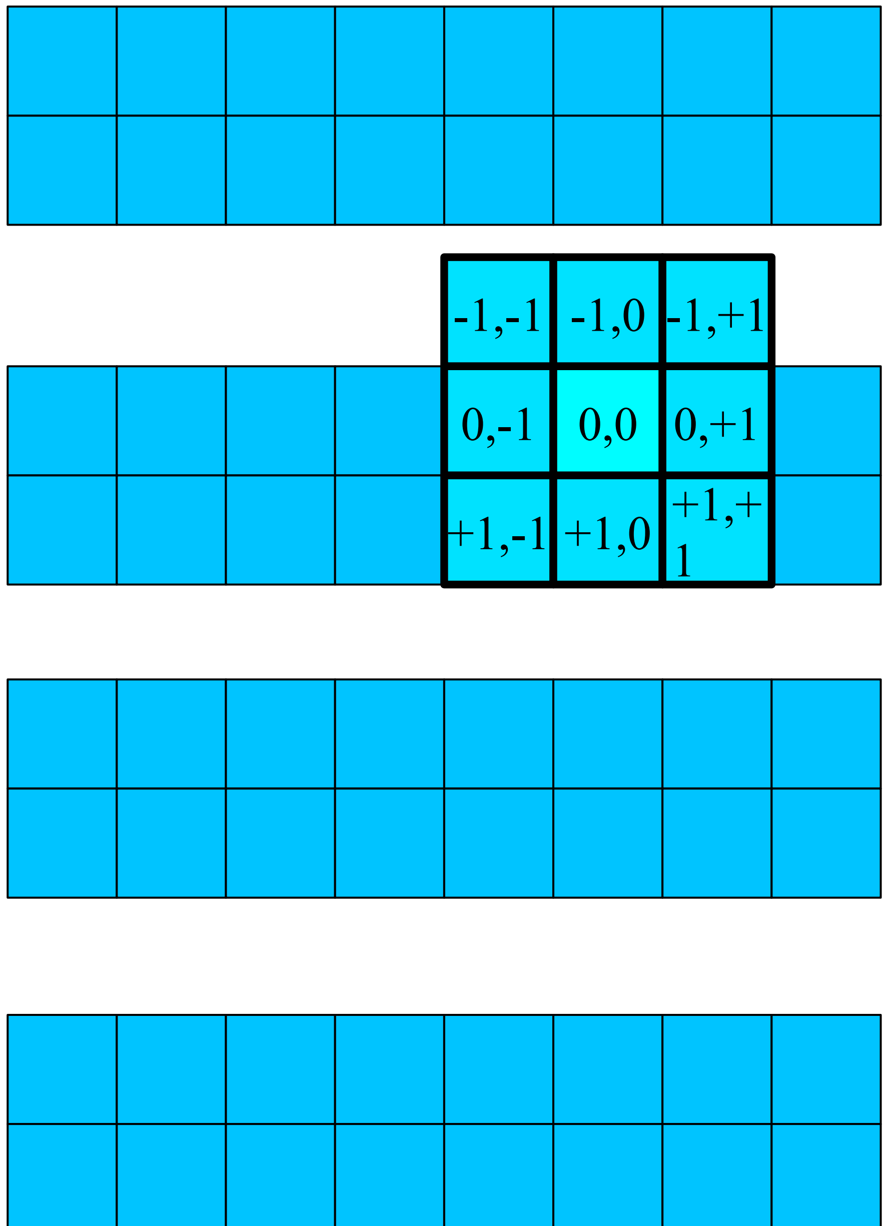
Code for Smoothing Program

```
for (r = 0; r < n; r++)
   for (c = 0; c < n; c++)
   {
      cnt = 0;
      sum = 0;
      for (rm = -1; rm < 2; rm++)
      {
         if (r+rm < 0 || r+rm >= n)
            continue;
         for (cm = -1; cm < 2; cm++)
         {
            if (c+cm < 0 || c+cm >= n)
                  continue;
            sum += input[r+rm][c+cm];
            cnt++;
         }
      }
      output[r][c] = sum / cnt;
   }
```

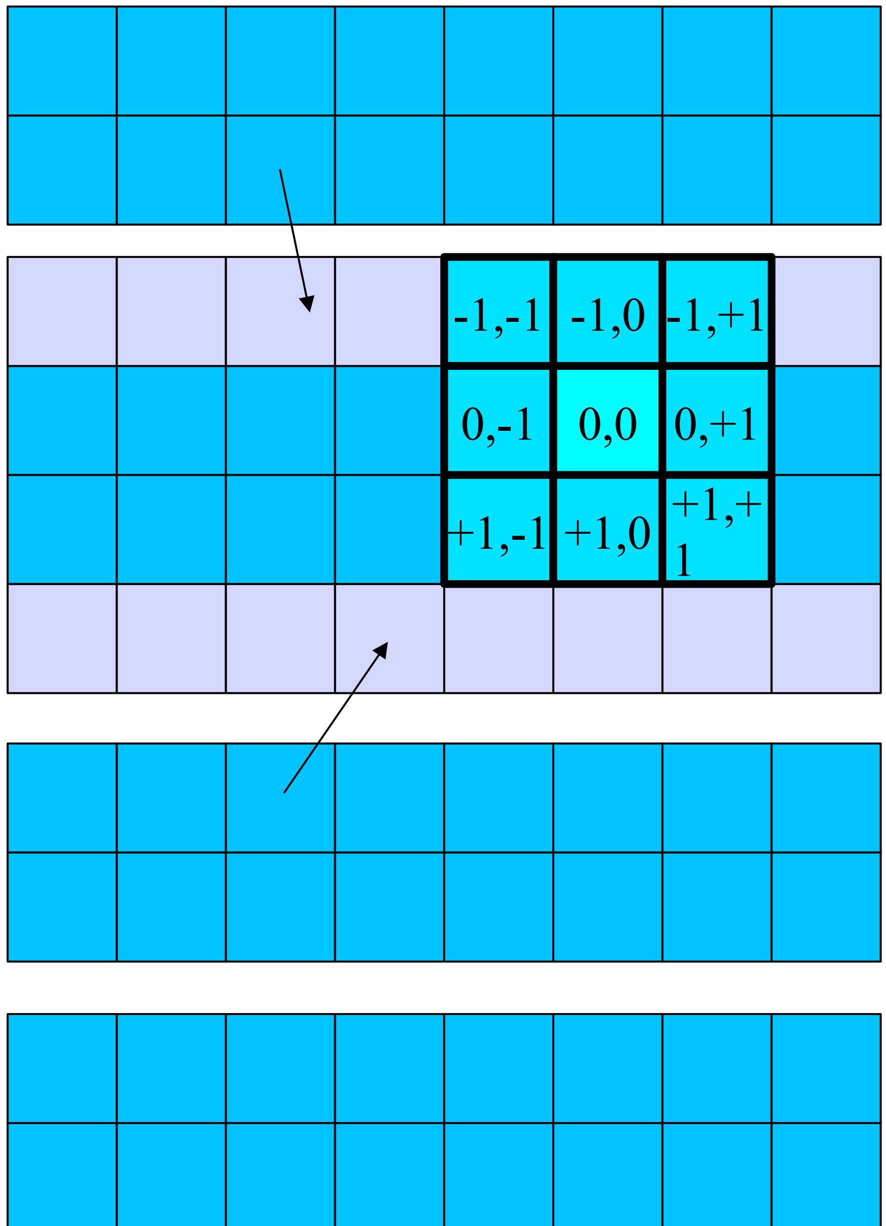
Smoothing Program



Dividing the Data

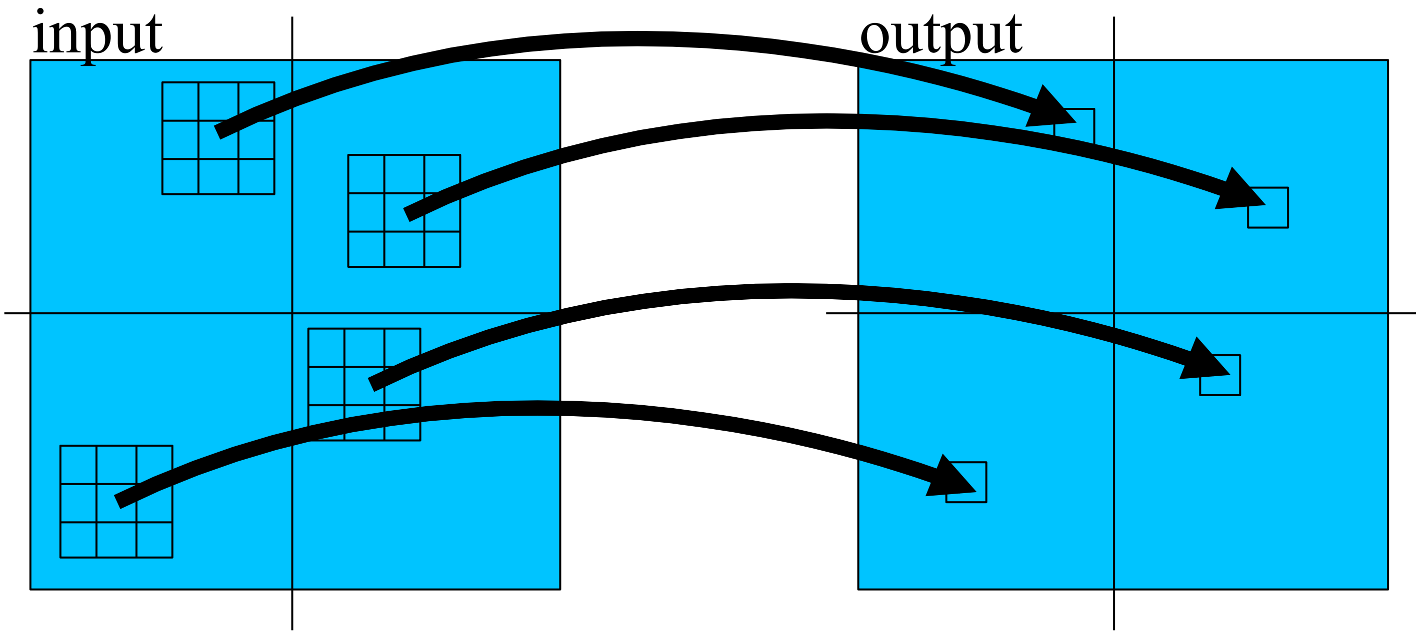


Border Cells



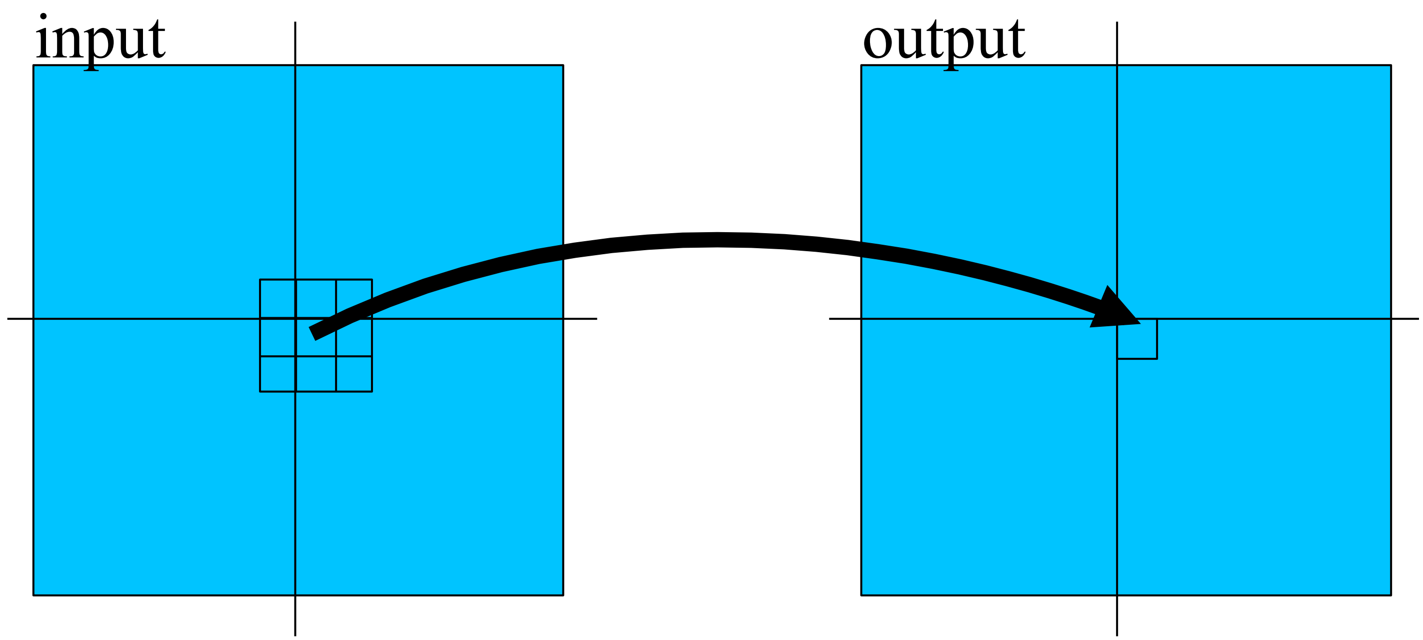
Parallel Smoothing

Image data divided among tasks
Each task smooths its portion



Border pixels need overlapping data

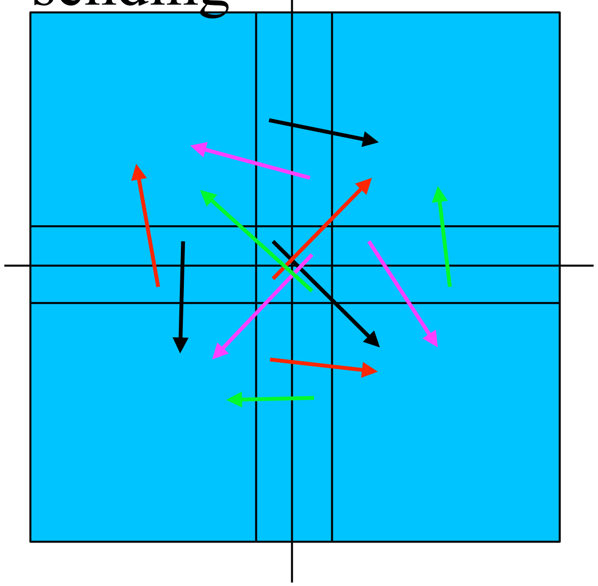
Data is required from the other tasks
Other tasks require data as well



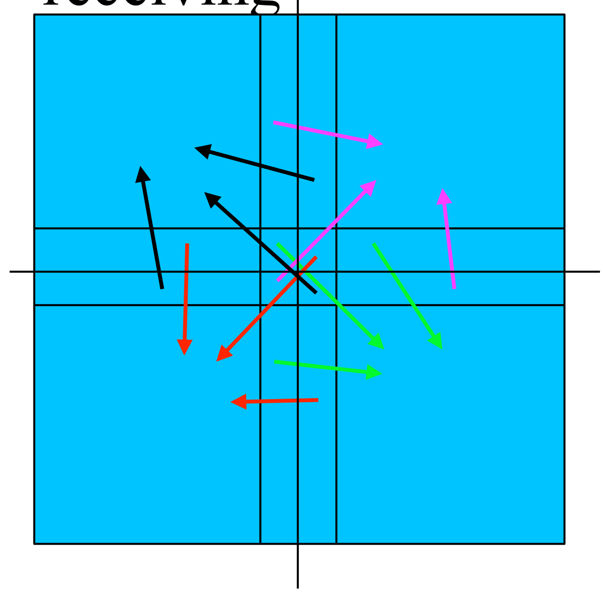
Tasks exchange border data

Each task sends to the other tasks
Each task receives from the other tasks
When more tasks, each exchanges with 8 adjacent neighbors

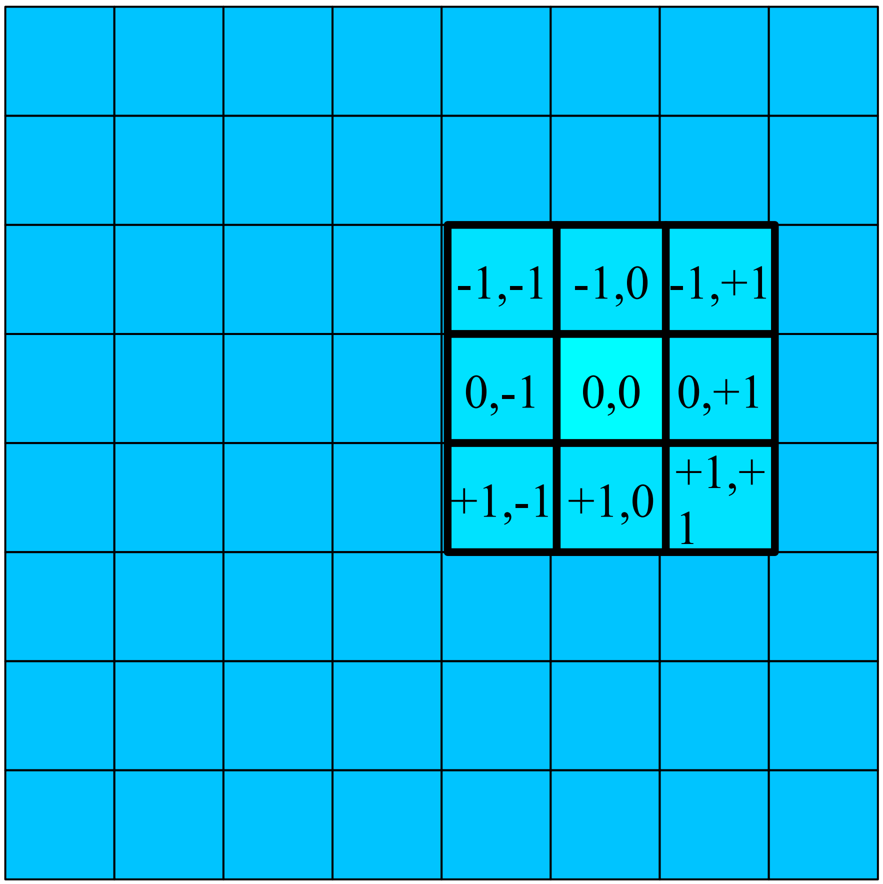
sending



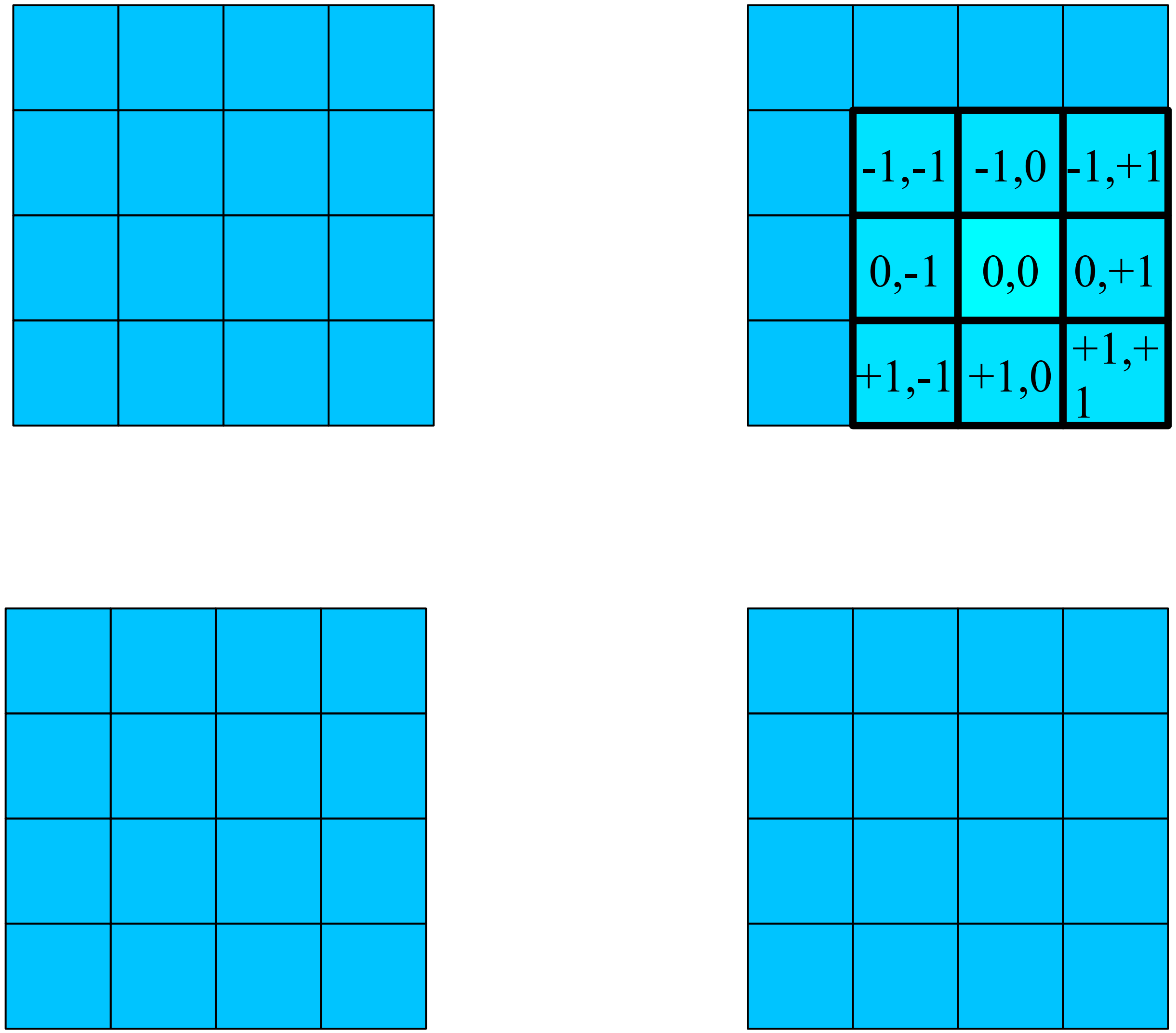
receiving



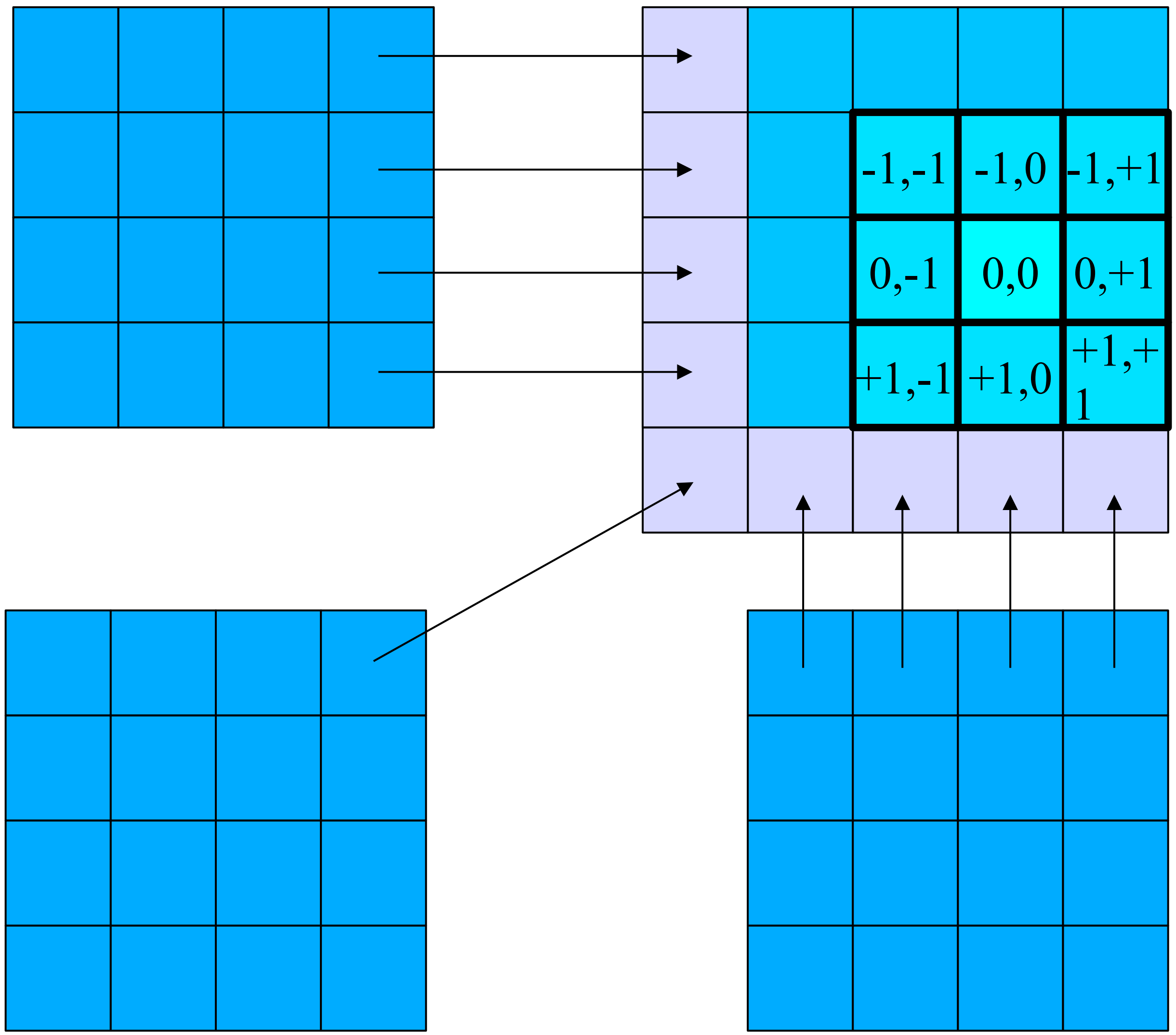
Smoothing Program



Dividing the Data



Border Cells



Code for Smoothing Program

```
exchange_borders(n, SIZE, RANK, input);
for (r = 1; r < (n/SIZE)+1; r++)
    for (c = 0; c < n; c++)
    {
        cnt = 0;
        sum = 0;
        for (rm = -1; rm < 2; rm++)
        {
            if (RANK == 0 && r+rm < 1 ||
                    RANK == SIZE-1 && r+rm > n/SIZE)
                continue;
            for (cm = -1; cm < 2; cm++)
            {
                if (c+cm < 0 || c+cm >= n)
                        continue;
                sum += input[r+rm][c+cm];
                cnt++;
            }
        }
        output[r][c] = sum / cnt;
    }
```

Exchange Code

```
exchange_borders(int n, int SIZE, int RANK, int input[][n])
{
   if (RANK < SIZE-1)
   {
      send(RANK+1, &input[n/size][0], n*sizeof(data));
      recv(RANK+1, &input[n/size+1][0], n*sizeof(data));
   }
   if (RANK > 0)
   {
      send(RANK-1, &input[1][0], n*sizeof(data));
      recv(RANK-1, &input[0][0], n*sizeof(data));
   }
}
```