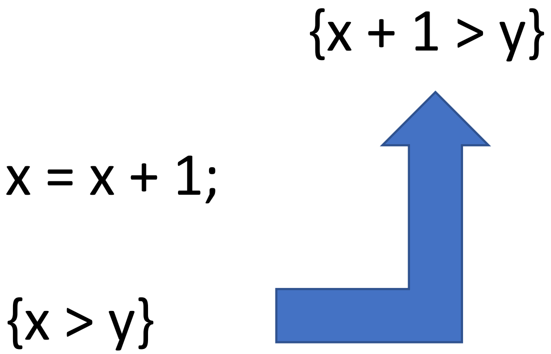
Propositional Logic – Logic of statements

Statements about programs

{P: true before execution}
Program_statement
{Q: true after execution}

Pre conditions and Post conditions

Proofs proceed from Post Conditions to Preconditions



```
{x + 1 > y}
x = x + 1;
{x > y}
```

Back substitution

$$\frac{\{P\}\text{->}\{Q(x)\}}{\{P\text{: } e/x[Q]\}\ x := e\ \{Q\}}$$

{P: (a / 10) > y}

x := a / 10;

{Q: x > y}

Chaining

$$\frac{F0\text{->}F1, F2\text{->}F'2, F3\text{->}F4, \{F1\}S1\{F2\}, \{F'2\}S2\{F3\}}{\{F0\}S1;S2\{F4\}}$$

{F0: **(y + z) > (k – x)** }
a := y + z;
b := k - x;
{F4: a > b}

{F0}
{F1} S1 {F2}
{F’2} S2 {F3}
{F4}

{F0: **(y + z) > (k – x)**}
{F1: **(y + z) > (k – x)}**
a := y + z;
{F2: **a > (k** – **x)**}
{F2’: **a > (k – x)}**
b := k - x;
{F3: **a > b**}
{F4: **a > b**}

Conditional

$$\frac{\{\text{Pre } \textbf{and} \text{ cond}\}\,\text{S1}\,\{\text{Post}\},\ \{\text{Pre } \textbf{and not} \text{ cond}\}\,\text{S2}\,\{\text{Post}\}}{\{\text{Pre}\}\,\textbf{if}\ (\text{cond})\ \text{S1}\ \textbf{else}\ \text{S2}\,\{\text{Post}\}}$$

```
{Pre: (P1 and cond) or (P2 and not cond)}
if (cond)
        {P1} S1 {Post}
Else
        {P2} S2 {Post}
{Post}
```

```
{Pre: (z > 5 and x > 0) or (-z > 5 and x <= 0)}
   if (x > 0)
            {P1: z > 5} y := z {Post: y > 5}
   else
            {P2: -z > 5} y := -z {Post: y > 5}
{Post: y > 5}
```

Loop Invariant

$$\frac{\{I \textbf{ and } \text{cond}\}\,S\,\{I\}}{\{I\}\,\textbf{while}\ (\text{cond})\ \text{do}\ S\,\{I \textbf{ and not } \text{cond}\}}$$

```
{I}
while (cond)
do
     {I and cond}S{I}
end
{I and not cond}
```

```
          {i = 0 and mysum = 0}
           I = { mysum = SUM(k=0..i-1)(a[k]) }
while (i < n) do
           { mysum = SUM(k=0..i-1)(a[k]), i < n } ->
           { mysum + a[i] = SUM(k=0..i)(a[k]), 0 <= i + 1 <= n}
    mysum = mysum + a[i];
           { mysum = SUM(k=0..i)(a[k]), 0 <= i + 1 <= n }
    i = i + 1;
           { mysum = SUM(k=0..i-1)(a[k]), 0 < i <= n }
end
           { mysum = SUM(k=0..i-1)(a[k]),  i = n }
```

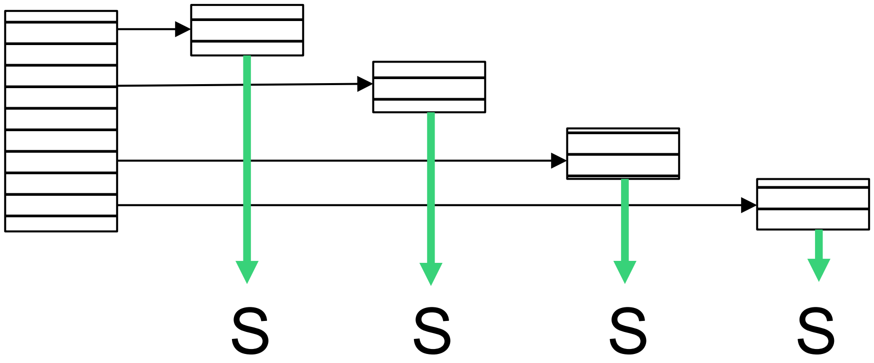
```
// counting loop
        Pre:{0<=n} ->
        {0<=n}
i = 0;
        I:{i<=n}
while (i < n)
begin
        I:{i<=n} and C:{i<n} ->
        {i+1<=n}
     i = i + 1;
        I:{i<=n}
end
        I: {i<=n} and nC:{i>=n} ->
        Post:{i=n}
```

```
// find largest between m and n
        Pre:{m<n} ->
        {A[m] >= A[m:m-1] and m-1<=n}
j = m;
        {A[j] >= A[m:m-1] and m-1<=n}
i = m;
        I:{A[j] >= A[m:i-1] and i-1<=n}
while( i <= n )
begin
           I:{A[j] >= A[m:i-1] and i-1<=n} and C:{i<=n} ->
           {A[i] >= A[m:i] and i<=n and A[i] > A[j]},
           {A[j] >= A[m:i] and i<=n and A[i] <= A[j]}
   if (A[i] > A[j])
      j = i;
           {A[j] >= A[m:i] and i<=n}
   i++;
           I:{A[j] >= A[m:i-1] and i-1<=n}
end
        I:{A[j] >= A[m:i-1] and i-1<=n} and nC:{i>n} ->
        {A[j] >= A[m:i-1] and i=n+1} ->
        Post:{A[j]>=A[m:n]}
```

∀x|m<=x<=i-1 A[j] >= A[x]

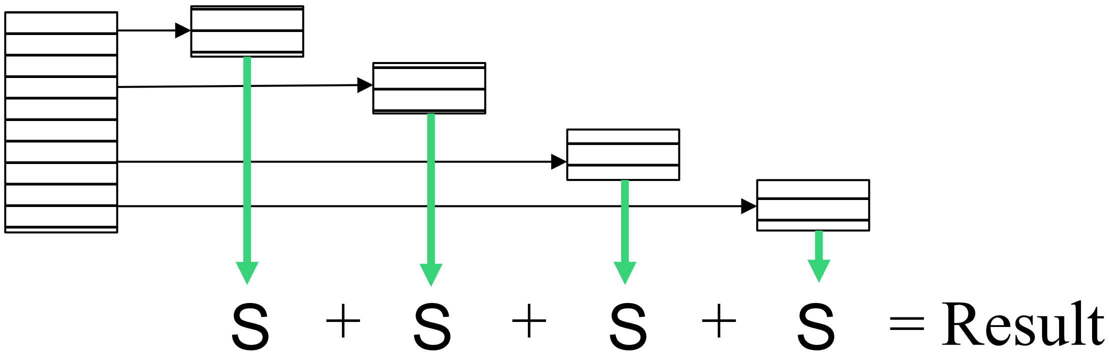
What is Parallel Computing?

- A mechanism for speeding up 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

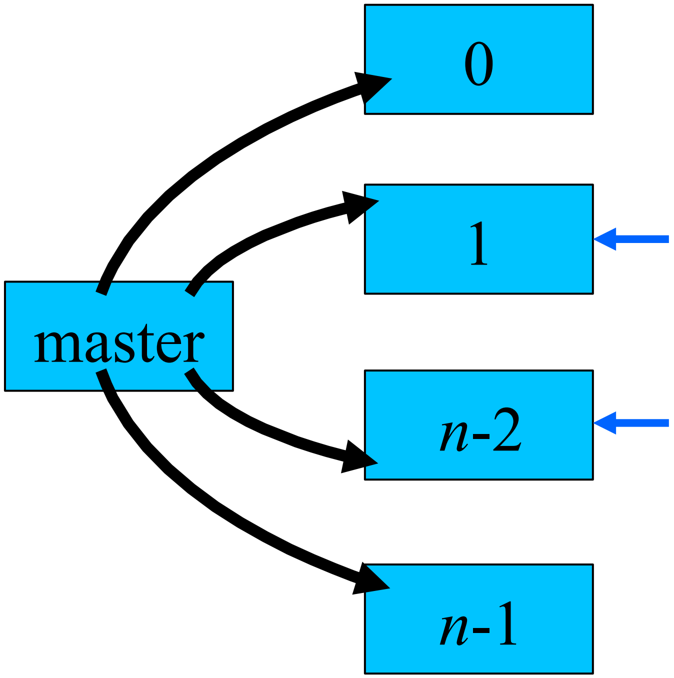
- TCP/IP
- UDP/IP
- GM (Myrinet)
- VIA
- PVM
- MPI
  - MPI 1.1
  - MPI 2.0

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

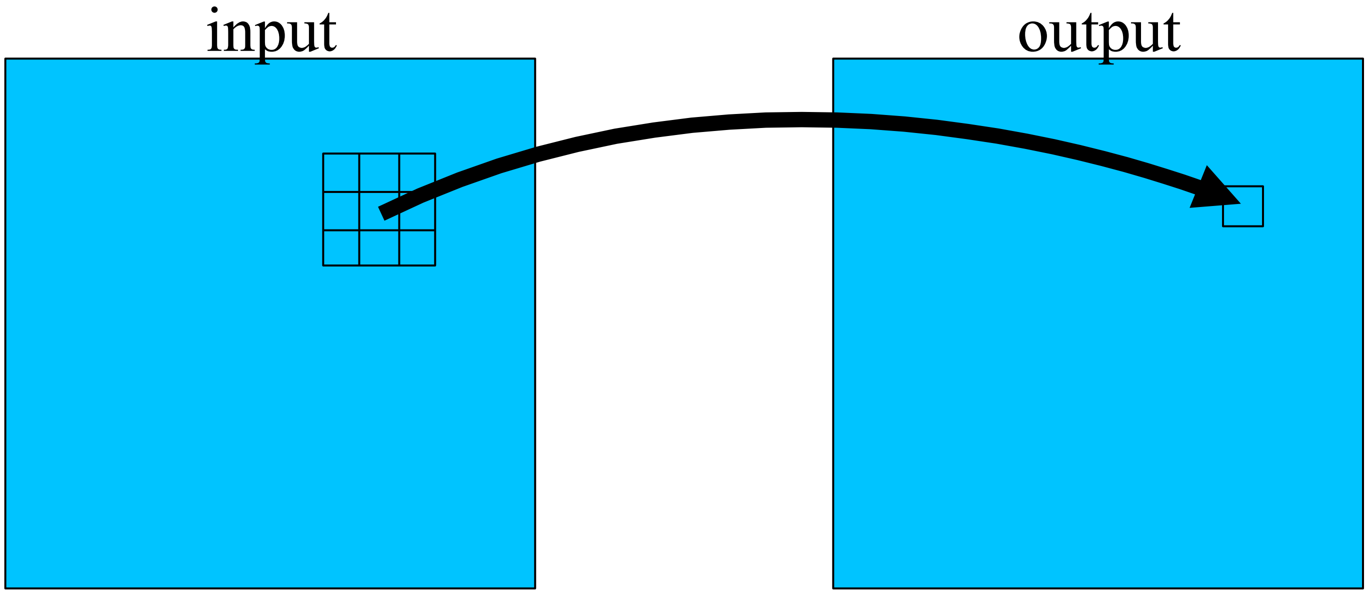
An MPI Job

- 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)



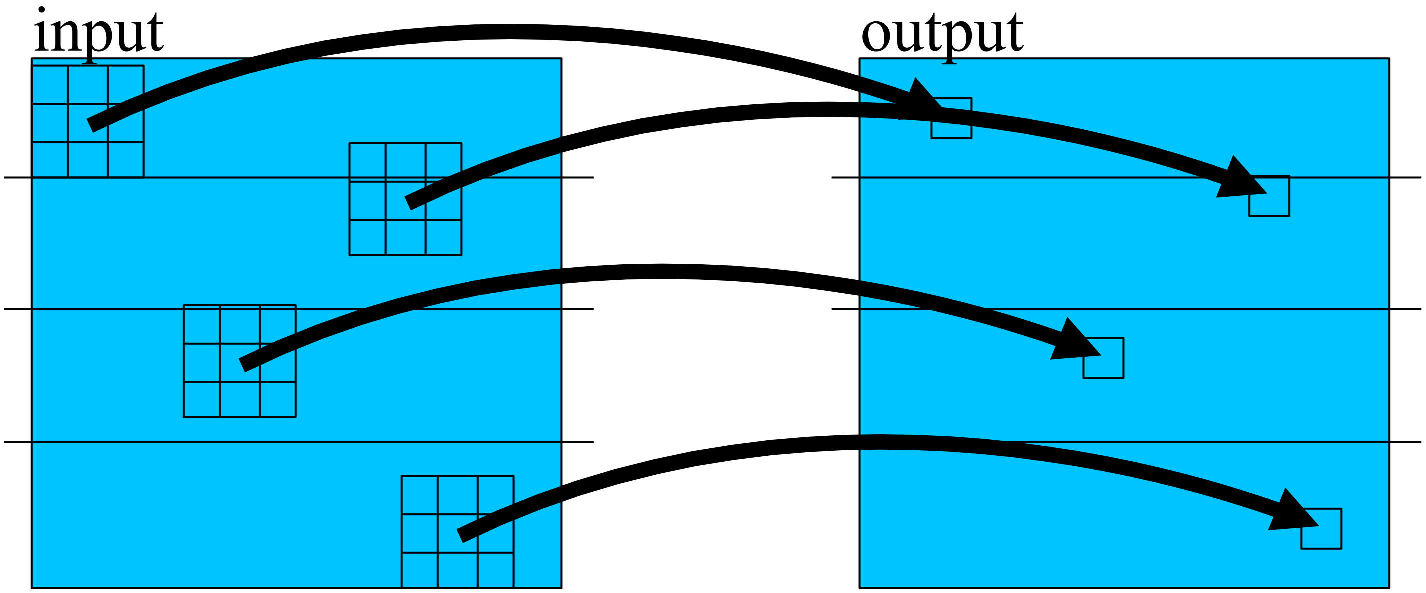
Example - Image Smoothing

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



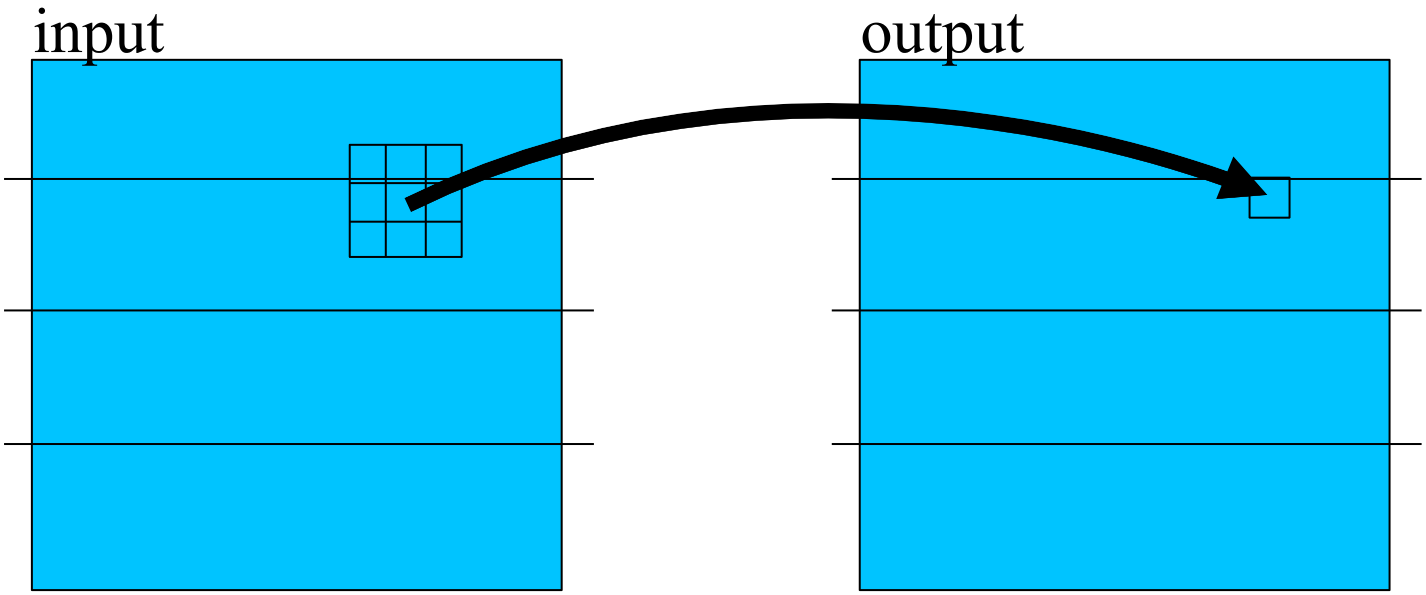
Parallel Smoothing

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



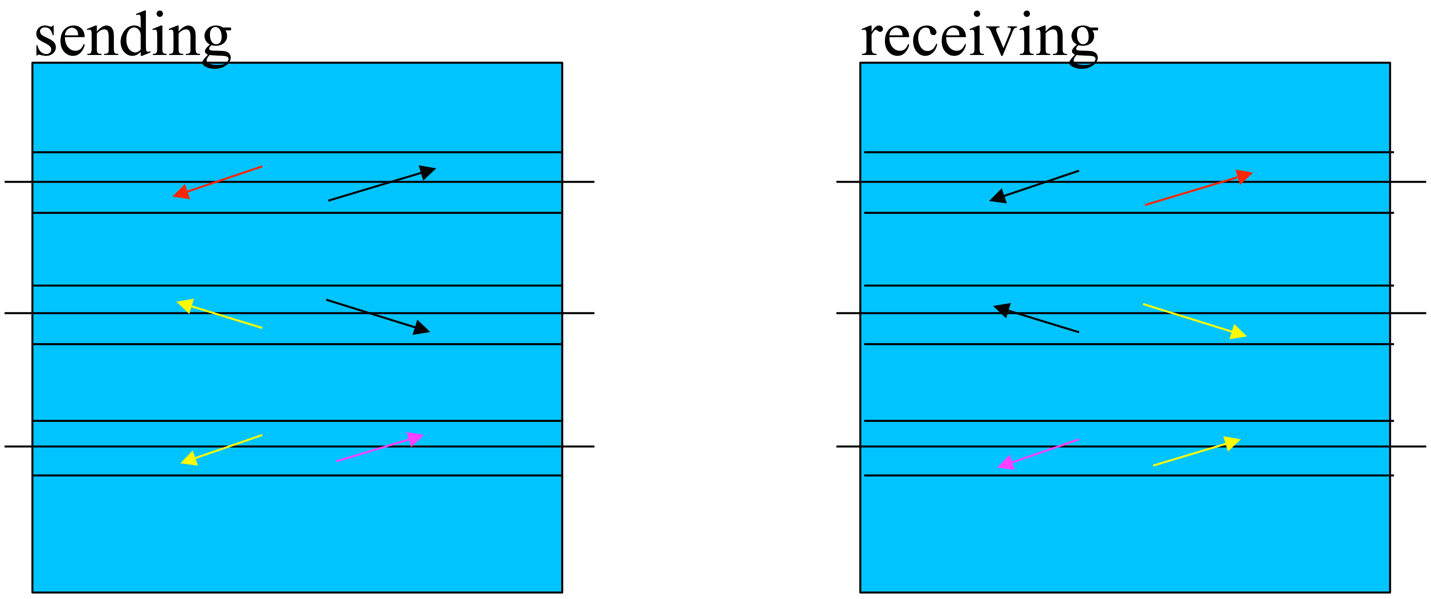
Border pixels need overlapping data

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



Tasks exchange border data

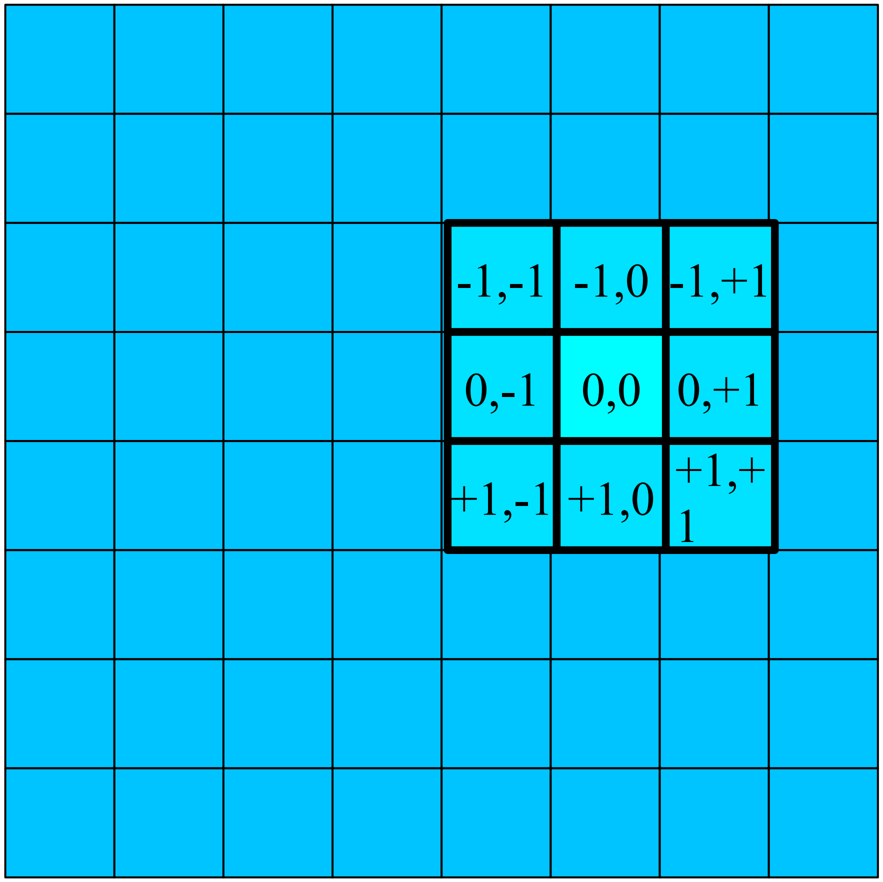
- Each task sends to the other tasks
- Each task receives from the other tasks
- When more tasks, each exchanges with 8 adjacent neighbors



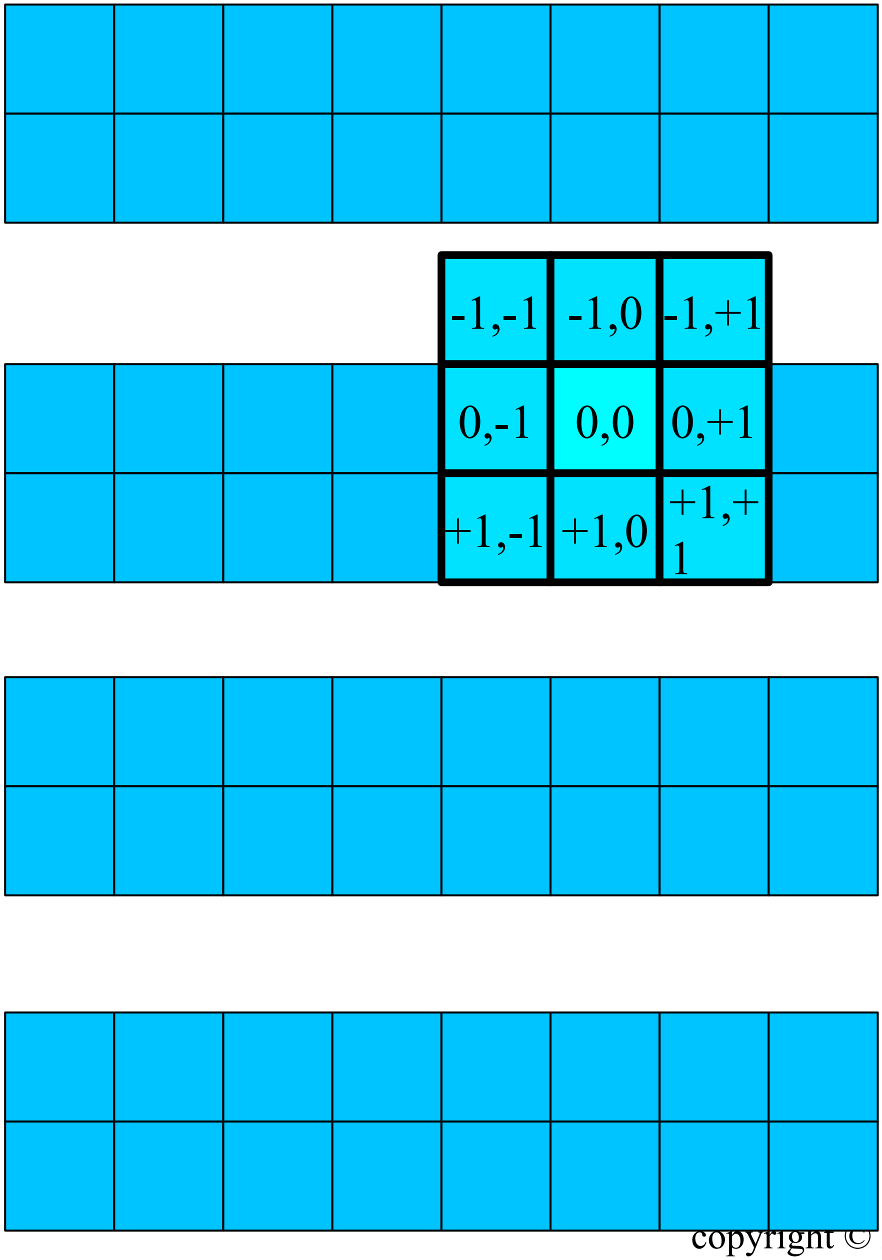
Code for Smoothing Program

```
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;
   }
```

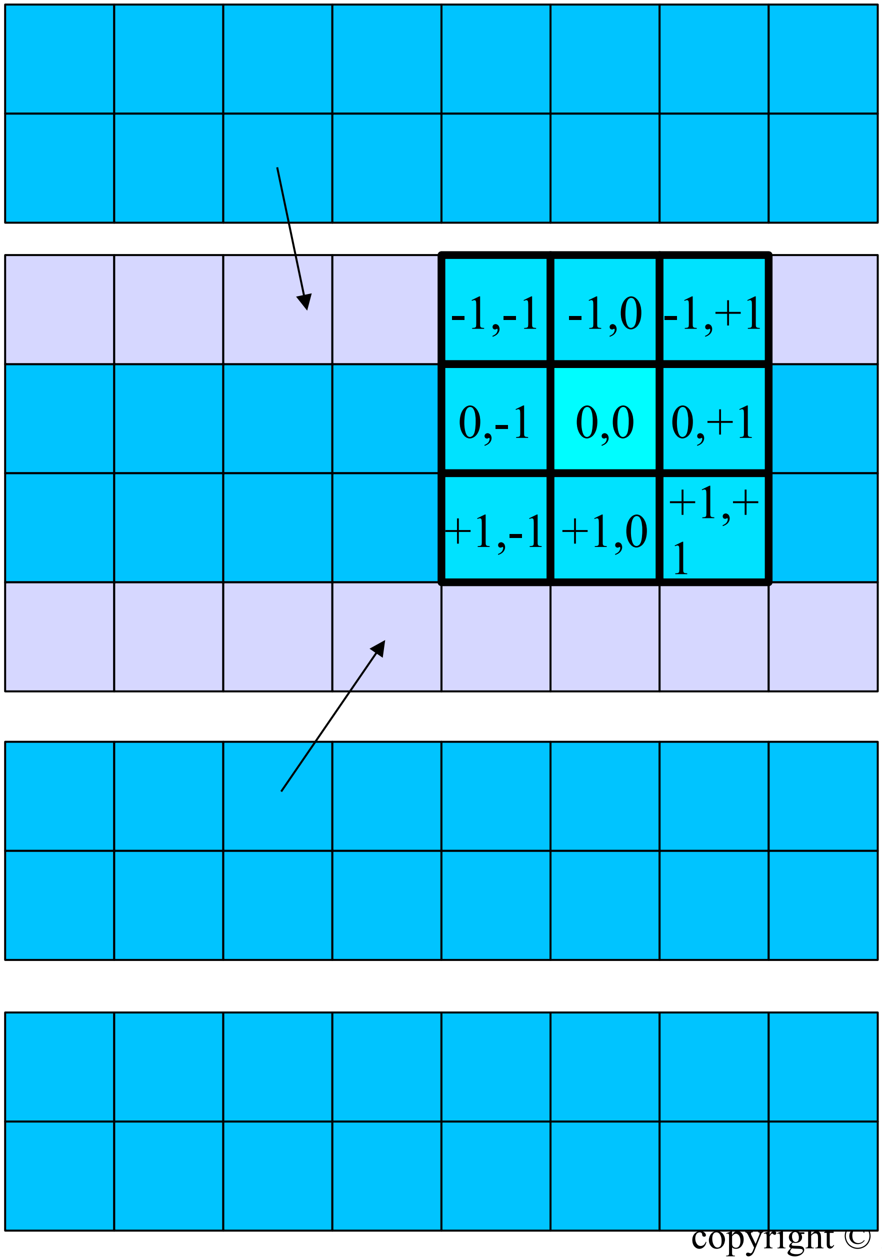
Smoothing Program



Dividing the Data



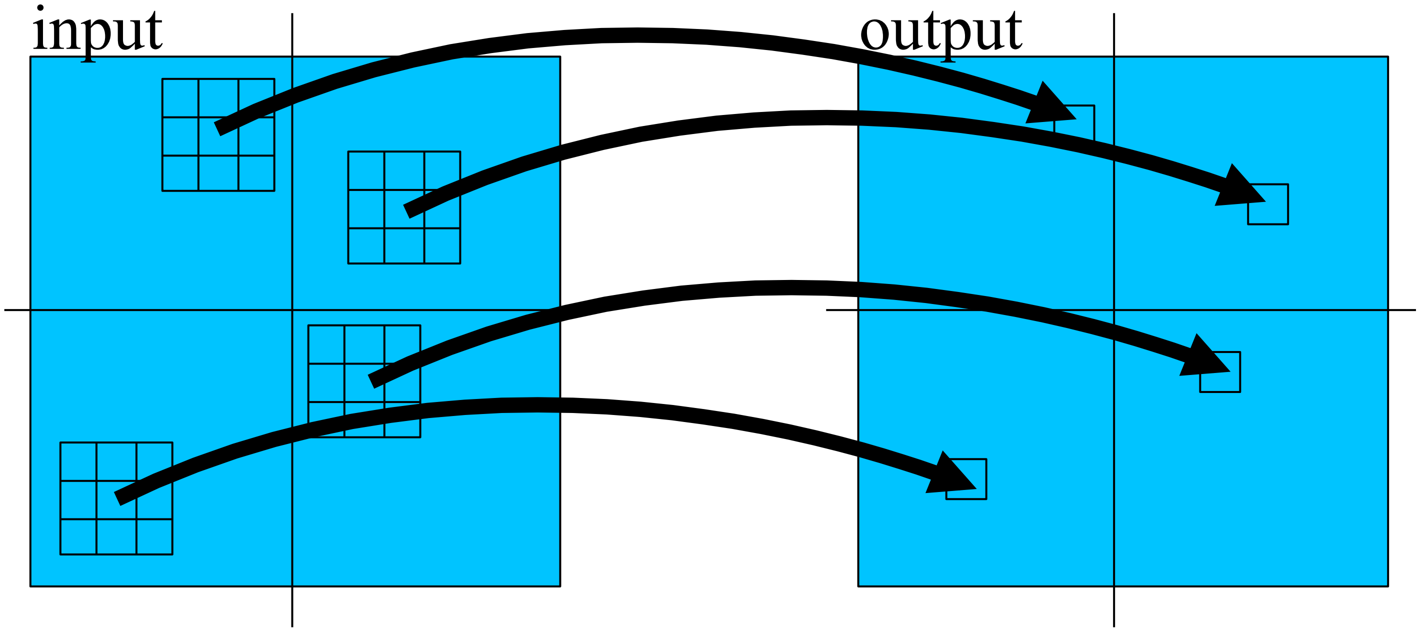
Border Cells



Parallel Smoothing

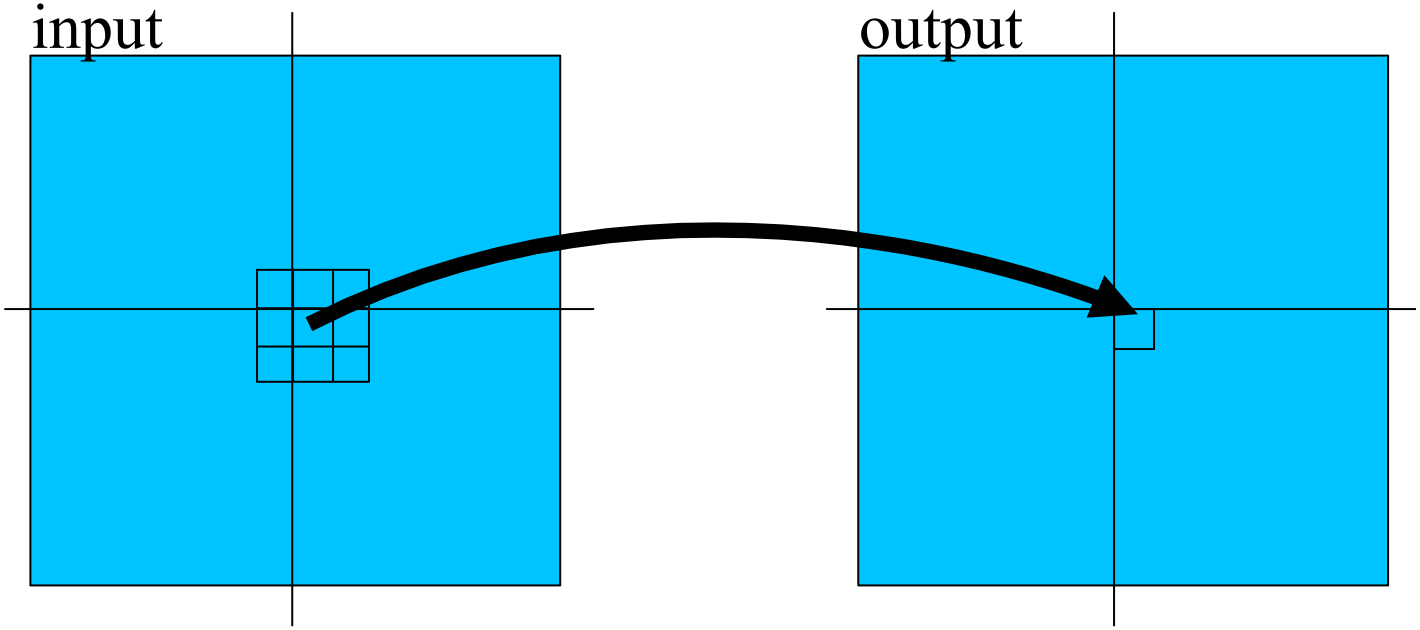
Image data divided among tasks

Each task smooths its portion



Border pixels need overlapping data

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



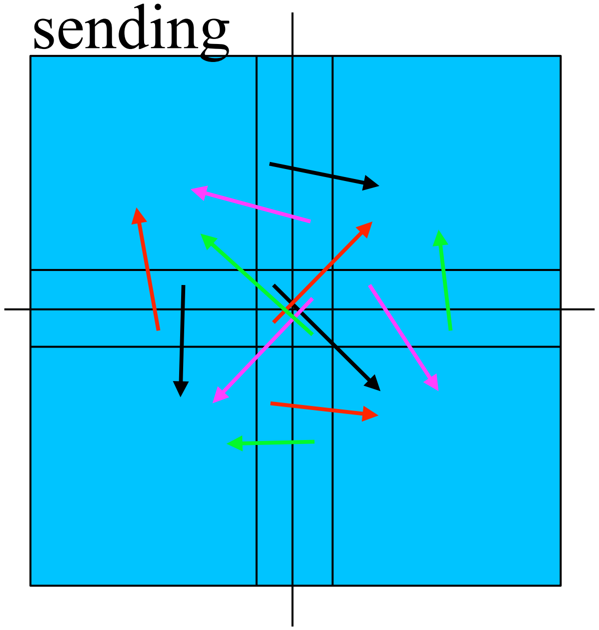
Tasks exchange border data

Each task sends to the other tasks

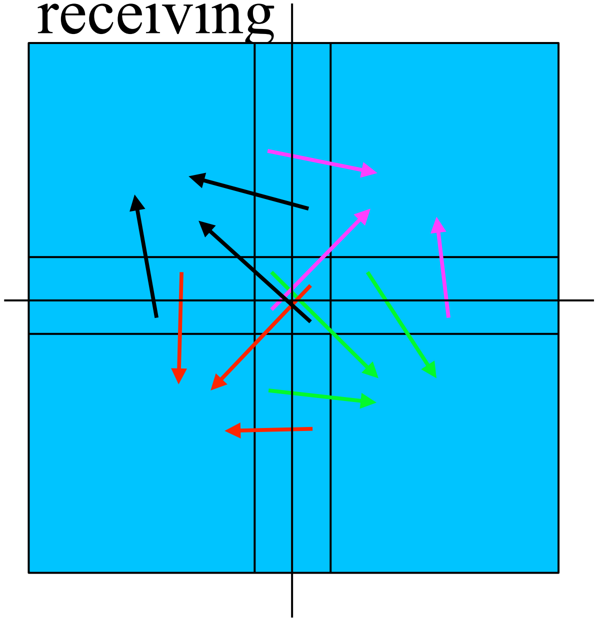
Each task receives from the other tasks

When more tasks, each exchanges with 8 adjacent neighbors

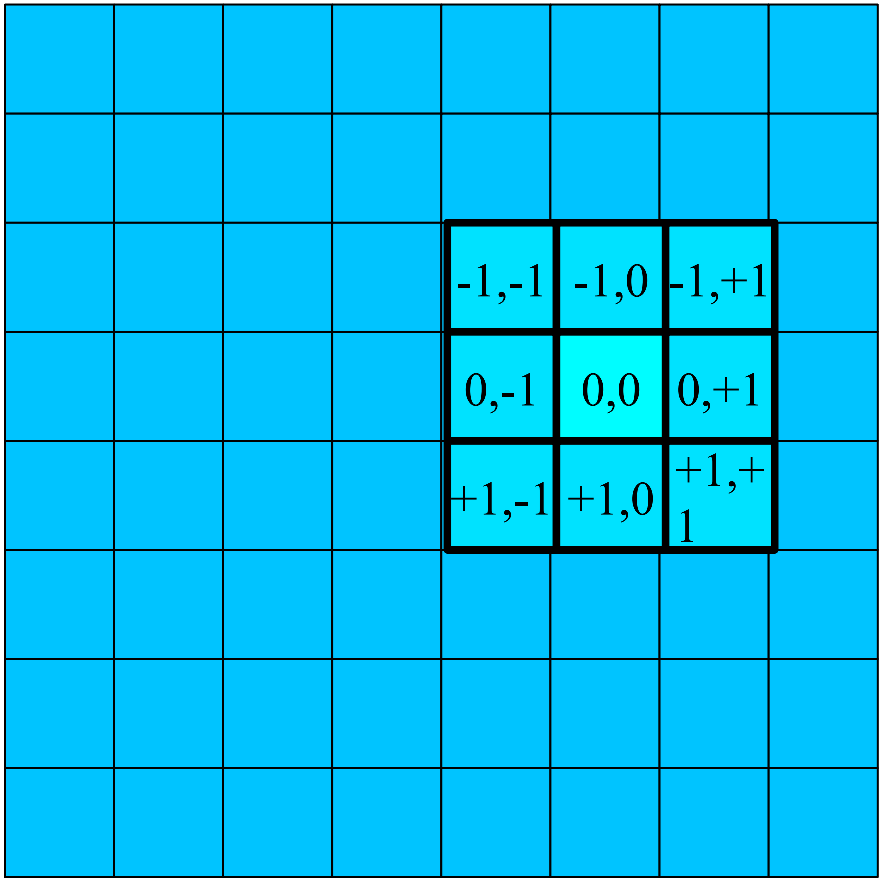
sending



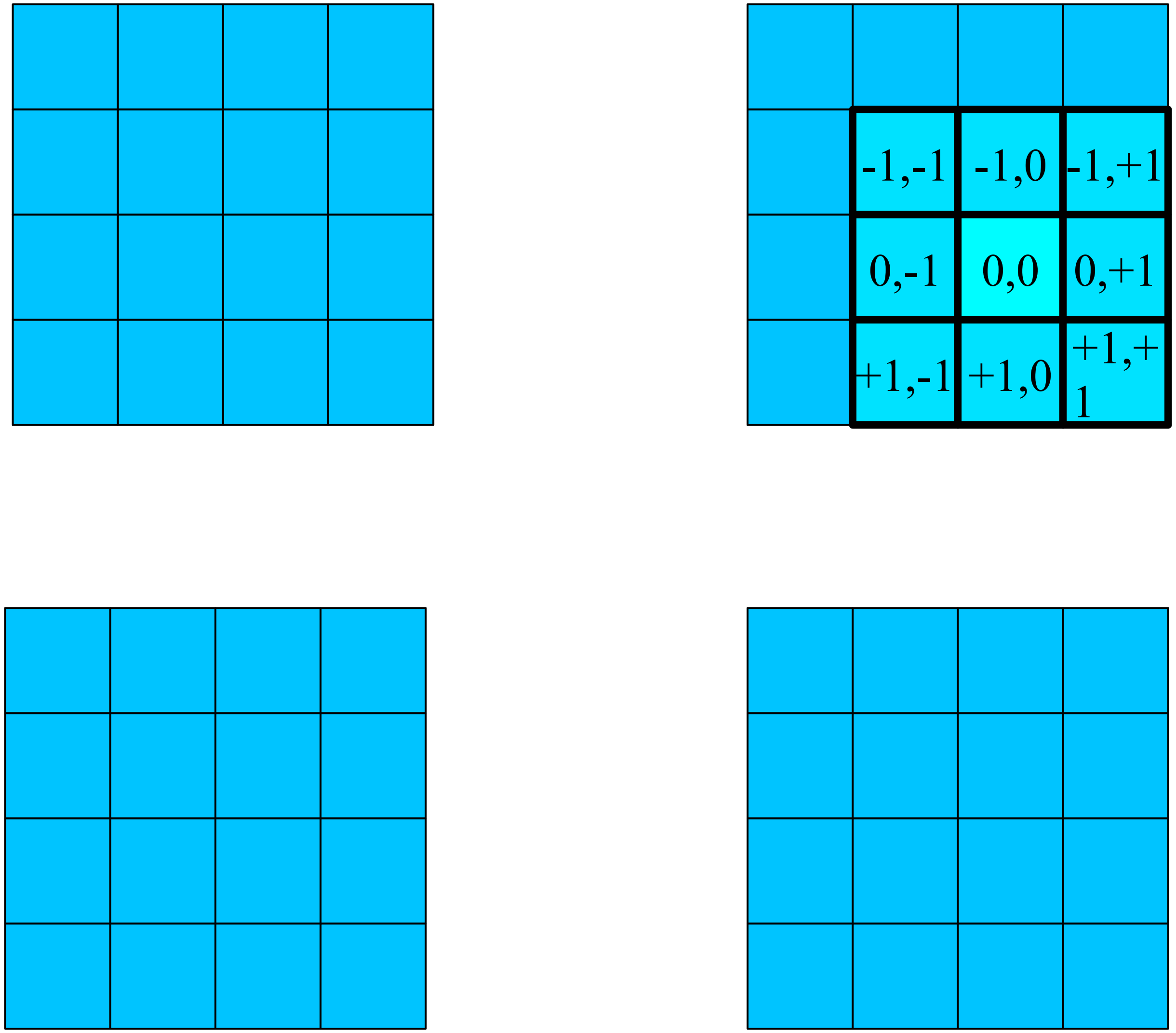
receiving



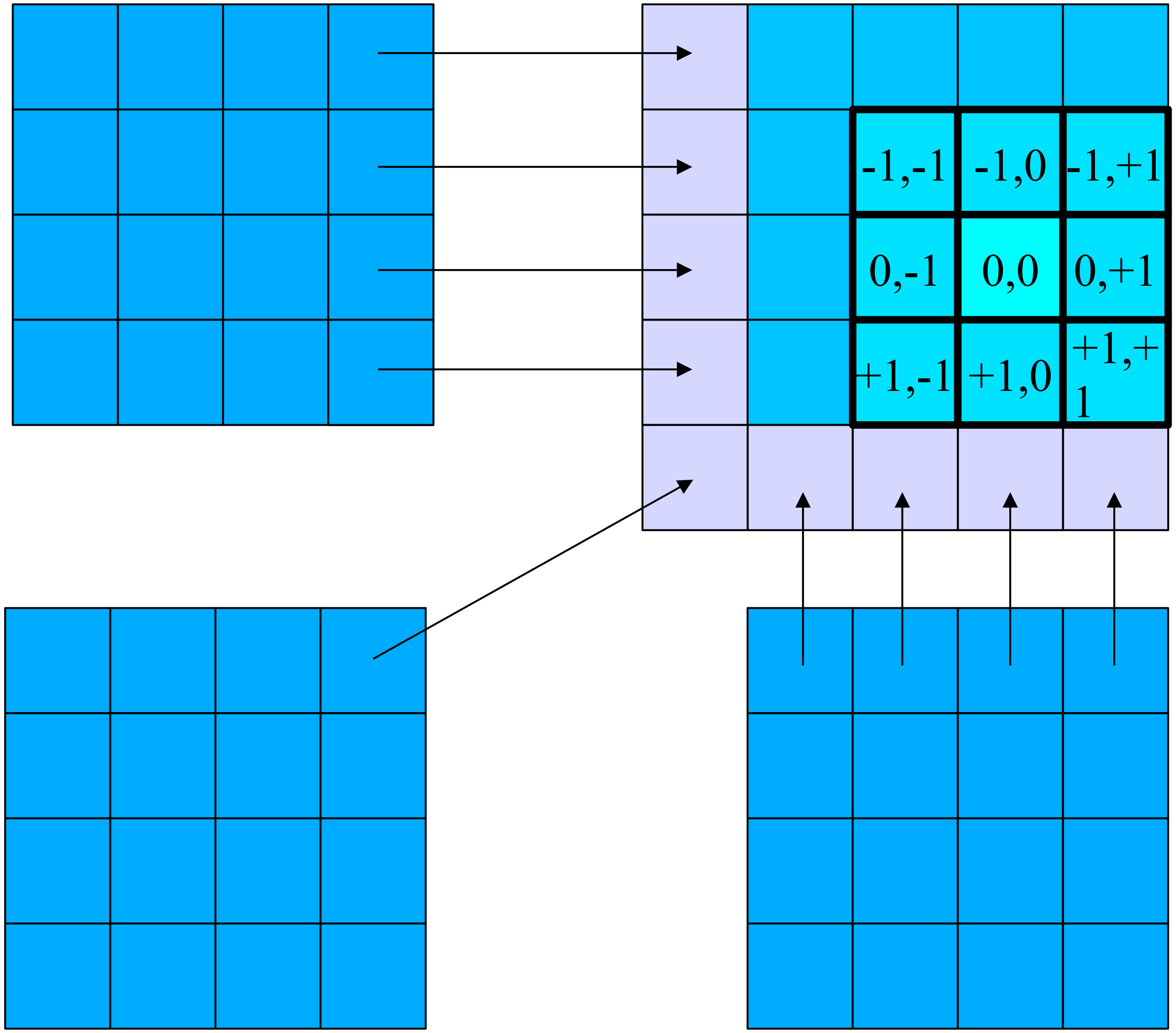
Smoothing Program



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));
   }
}
```

MPI - Part 2

- Basic message passing
- Semantics
- Simple IO

Introduction to MPI

- MPI is a standard for message passing interfaces
- MPI-1 covers point-to-point and collective communication
- MPI-2 covers connection based communication and I/O
- Typical implementations include MPICH (Used by Scyld), and LAMMPI

MPI Basics

Most used MPI commands

```
MPI_Init - start using MPI
MPI_Comm_size - get the number of tasks
MPI_Comm_rank - the unique index of this task
MPI_Send - send a message
MPI_Recv - receive a message
MPI_Finalize - stop using MPI
```

Initialize and Finalize

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

- Must be called before any other MPI calls

```
int MPI_Finalize();
```

- No MPI calls may be made after this
- Program MUST call this (or it won't terminate)

Initialize and Finalize

First MPI call must be to `MPI_Init`.
Last MPI call must be to `MPI_Finalize`.

```
#include <mpi.h>
main(int argc, int **argv)
{
     MPI_Init(&argc, &argv );
     // put program here
     MPI_Finalize();
}
```

Size and Rank

MPI_Comm_size returns the number of tasks in the job

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI_Comm_rank returns the number of the current task (0 .. size-1)

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

A Simple Example

```
#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 process %d of %d
\n",
        rank,size);
     MPI_Finalize();
     return 0;
}
```

Send and Recv

MPI_Send to send a message

```
char sbuf[COUNT];
MPI_Send(sbuf, COUNT, MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
```

MPI_Recv to receive a message

```
char rbuf[COUNT];
MPI_Status status;
MPI_Recv(rbuf, COUNT, MPI_CHAR, 1, 99,
MPI_COMM_WORLD, &status);
```

Anatomy of MPI_Send

```
MPI_Send(sbuf, COUNT, MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
```

sbuf : pointer to send buffer

COUNT : items in send buffer

MPI_CHAR : MPI datatype

1 : destination task number (rank)

99 : message tag

MPI_COMM_WORLD : communicator

Anatomy of MPI_Recv

```
MPI_Recv(rbuf, COUNT, MPI_CHAR, 1, 99,
MPI_COMM_WORLD, &status);
```

```
rbuf : pointer to receive buffer
COUNT : items in receive buffer
MPI_CHAR : MPI datatype
1 : source task number (rank)
99 : message tag
MPI_COMM_WORLD : communicator
Status : pointer to status struct
```

MPI Datatypes

Encodes type of data sent and received

Built-in types

`MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG`
`MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE`
`MPI_BYTE, MPI_PACKED`

User defined types (covered later)

`MPI_Type_contiguous, MPI_Type_vector,`
`MPI_Type_indexed, MPI_Type_struct MPI_Pack,`
`MPI_Unpack`

MPI Communicators

Abstract structure represents a group of MPI tasks that can communicate

`MPI_COMM_WORLD` represents all of the tasks in a given job

Programmer can create new communicators to subset `MPI_COMM_WORLD`

`RANK` or task number is relative to a given communicator

Messages from different communicators do not interfere

MPI Task Numbers

Each task in a job has a unique `rank` or task number

Numbers run from `0` to `size-1`, where `size` is the number of tasks

```
MPI_Comm_size(MPI_COMM_WORLD, &size)
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
```

`Send` and `Recv` specify destination or source task by rank

`Recv` can specify source of `MPI_ANY_SOURCE` to receive from any task

Message Tags

All messages are sent with an integer message tag

`MPI_Recv` will only receive a message with the tag specified

`MPI_ANY_TAG` can be used to receive messages with any tag

MPI Status Struct

Allows user to query the return status of MPI call

```
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

Allows user to query number of items received

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count)
```

Send and Receive Example

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

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

    int numprocs, myrank, namelen, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    char greeting[MPI_MAX_PROCESSOR_NAME + 80];
    MPI_Status status;

    MPI_Init( &argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
    MPI_Get_Processor_name( processor_name, &namelen);

    sprintf(greeting,"Hello world from process %d of %d on %s \n",
        myrank,numprocs, processor_name);
```

Send and Receive Example

```
  if (myrank == 0) {
          printf("%s\n", greeting);
          for(i=1;i<numprocs;i++) {
               MPI_Recv(greeting,sizeof(greeting), MPI_CHAR,
                    i, 1, MPI_COMM_WORLD);
               printf("%s\n", greeting);
          }
  }
  else {
          MPI_Send(greeting, strlen(greeting) +1, MPI_CHAR,
               0,1,MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return( 0);
}
```

Receive Buffer Size

- Receive buffer must be big enough for the message being received
- If message is smaller, only part of buffer filled in
- If message is too big, overflow error
- MPI_Probe allows programmer to check the next message before receiving it

```
// src, tag, comm, stat
MPI_Probe(1, 99, MPI_COMM_WORLD,
&status);
```

Matching Send and Recv

- When `MPI_Send` called, send is "posted"
- When `MPI_Recv` called, receive is "posted"
- Posted `MPI_Recv` matches posted `MPI_Send` if

  -Destination of `MPI_Send` matches receiving task

  -Source of `MPI_Recv` matches sending task or source is `MPI_SOURCE_ANY`

  -Tag of `MPI_Send` matches tag of `MPI_Recv` or tag of `MPI_Recv` is `MPI_TAG_ANY`

  -Communicator or `MPI_Send` matches communicator of `MPI_Recv`

Semantics

Covered on next pages ...

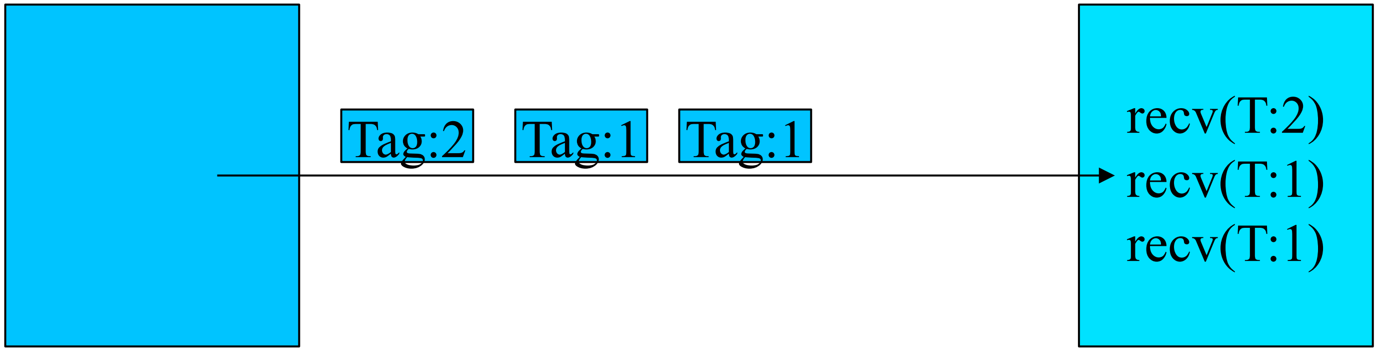
- -Messages are non-overtaking
- -Progress is guaranteed
- -Fairness is not guaranteed
- -System resources may be limited

Non-Overtaking Messages

- If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.
- If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

Quoted from: MPI: A Message-Passing Interface Standard

Non-overtaking Messages

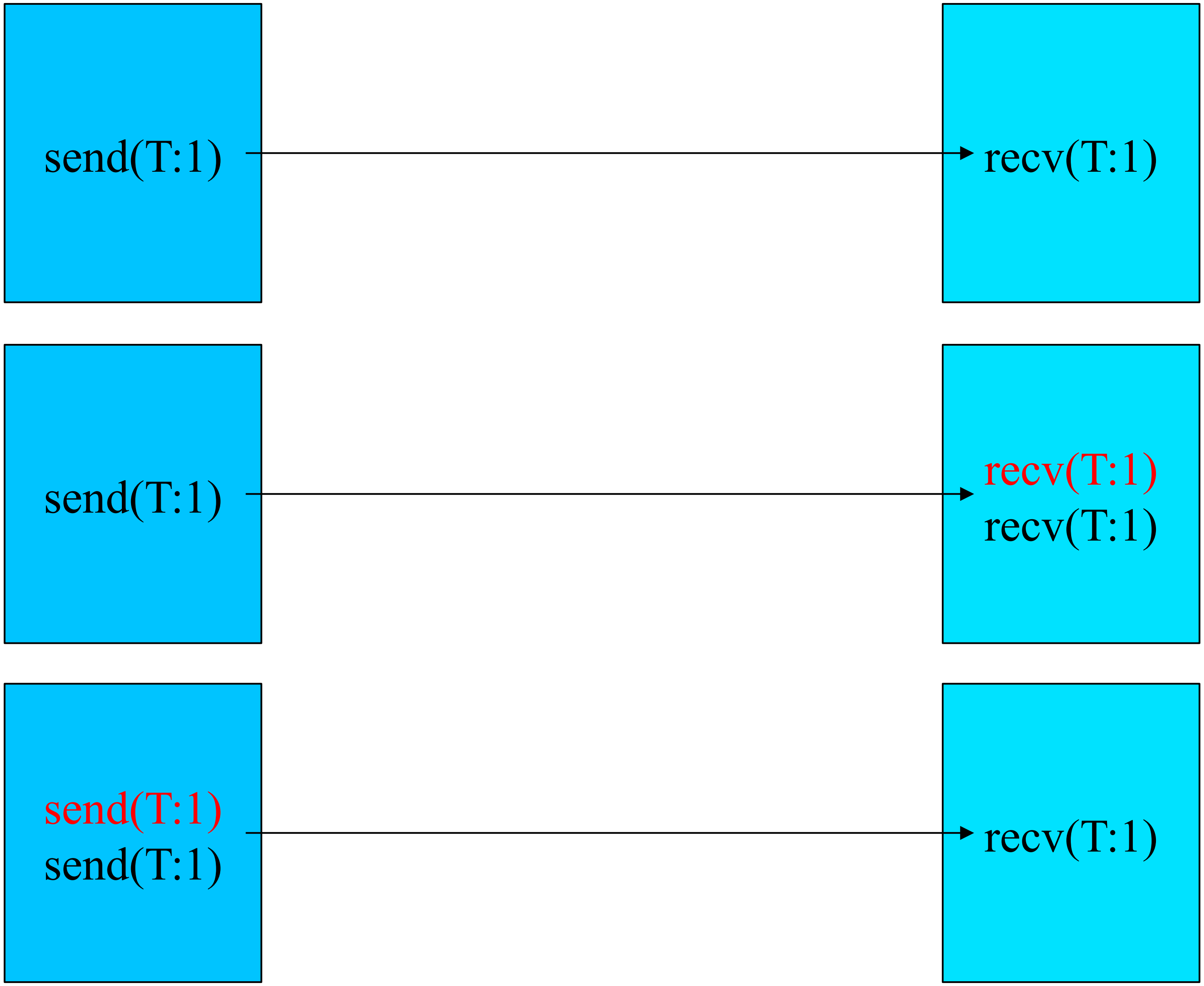


Progress

- If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Quoted from: MPI: A Message-Passing Interface Standard

Progress Guaranteed



send(T:1)

recv(T:1)

send(T:1)

recv(T:1)

recv(T:1)

send(T:1)

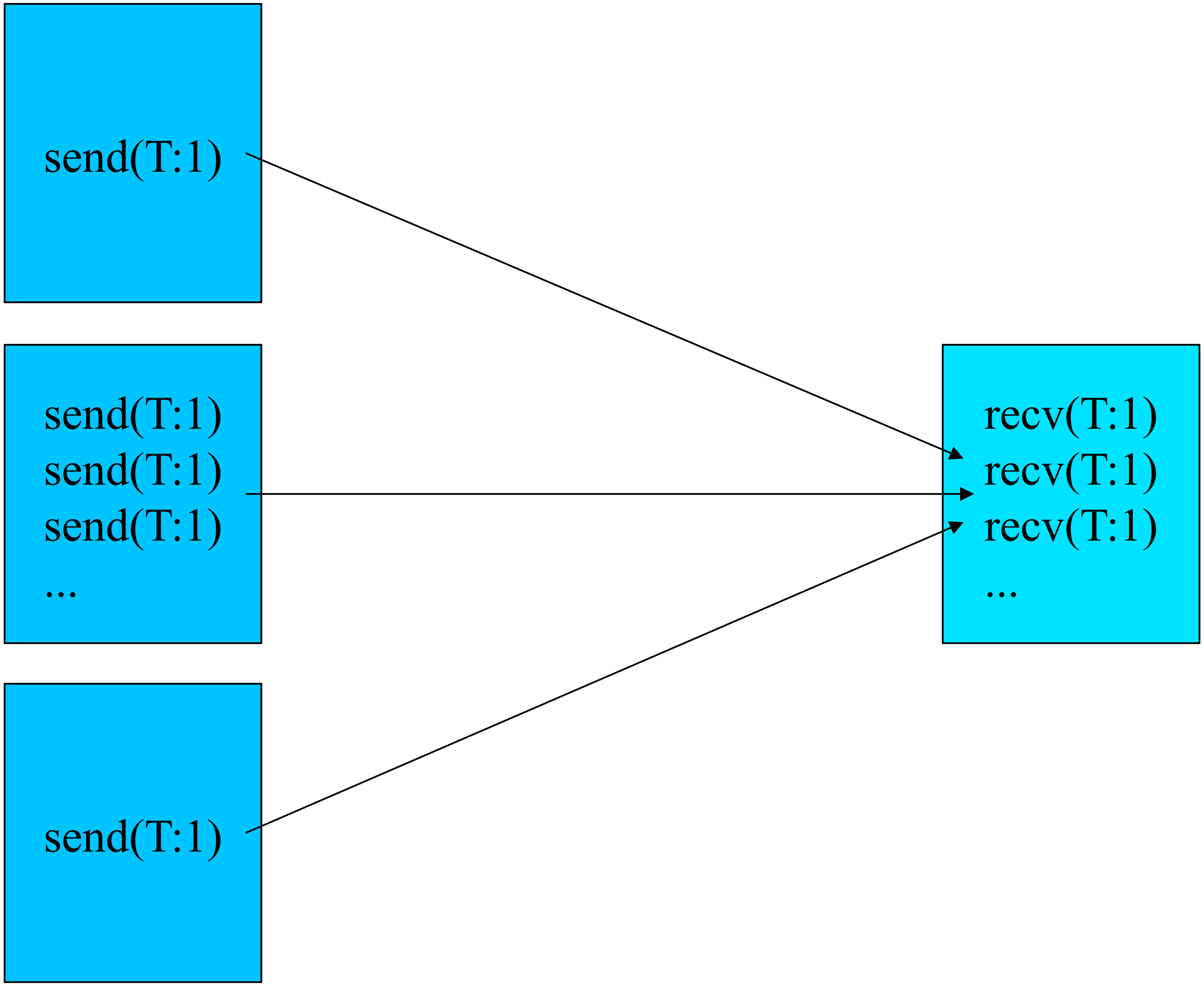
send(T:1)

recv(T:1)

Fairness

- MPI does not guarantee fairness in matching messages
- Messages from different sources may overtake one another
- Programmer's responsibility to prevent starvation

Fairness NOT Guaranteed



send(T:1)

send(T:1)
send(T:1)
send(T:1)
...

send(T:1)

recv(T:1)
recv(T:1)
recv(T:1)
...

Limited System Resources

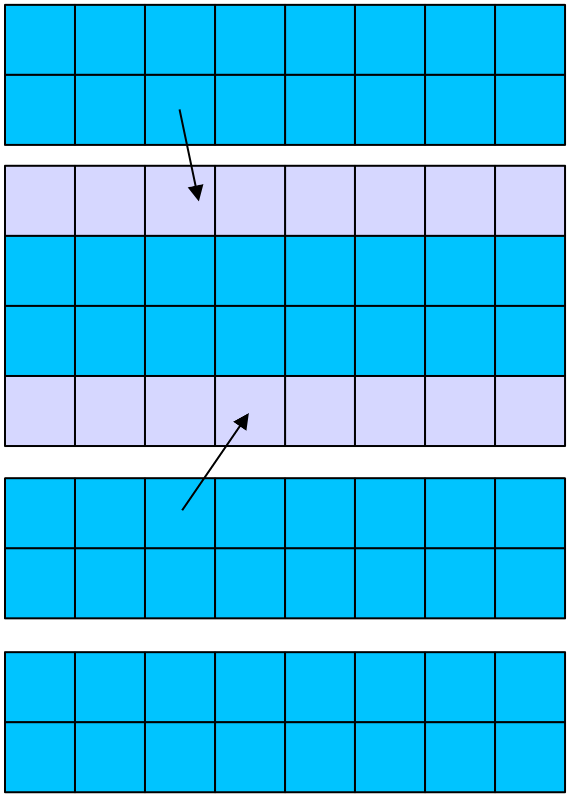
- MPI does not guarantee system resources exist for buffering messages
- Programs that assume system resources available can deadlock if resources become busy
- Properly coded programs usually exist that will complete regardless of available buffer space
- Buffered mode allows programmer to provide adequate buffer space

Combined Send/Recv

- Single call both sends and receives a message
- Can send and receive to same task, or different tasks
- Guarantees that buffering and blocking semantics will not result in deadlock

```
MPI_Sendrecv(sbuf, scount, stype, dest,
stag, rbuf, rcount, rtype, source, rtag,
comm, &status);
MPI_Sendrecv_replace(buf, count, type,
dest, stag, source, rtag, comm,
&status);
```

Smoothing Example



Back to our example

```
#include <mpi.h>
#define n 1000;
main(int argc, char **argv)
{
   int n, SIZE, RANK;
   int *input, *output;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &SIZE);
   MPI_Comm_rank(MPI_COMM_WORLD, &RANK);

   input = (int *)malloc((n/SIZE+2) * n * sizeof(int));
   output = (int *)malloc(n/SIZE * n * sizeof(int));

   read_image(n, SIZE, RANK, input);
   parallel_smooth(n, SIZE, RANK, input, output);
   write_image(n, SIZE, RANK, output);

   MPI_Finalize();
}
```

Smoothing Function

```
parallel_smooth(int n, int SIZE, int RANK,
        int input[][n], int output[][n])
{
    int r, c, rm, cm, sum, cnt;
    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 Function

```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
   MPI_Status status;
   if (RANK < SIZE-1)
   {
      MPI_Send(&input[1][0], n, MPI_INT, RANK+1, 1,
            MPI_COMM_WORLD);
      MPI_Recv(&input[0][0], n, MPI_INT, RANK+1, 1,
            MPI_COMM_WORLD, &status);
   }
   if (RANK > 0)
   {
      MPI_Send(&input[n/SIZE][0], n, MPI_INT, RANK-1, 1,
            MPI_COMM_WORLD);
      MPI_Recv(&input[n/SIZE+1][0], n, MPI_INT, RANK-1, 1,
            MPI_COMM_WORLD, &status);
   }
}
/* THIS COULD DEADLOCK */
```

Exchange Using Sendrecv

```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
   MPI_Status status;
   if (RANK < SIZE-1)
   {
      MPI_Sendrecv(&input[1][0], n, MPI_INT, RANK+1, 1,
            &input[0][0], n, MPI_INT, RANK+1, 1,
            MPI_COMM_WORLD, &status);
   }
   if (RANK > 0)
   {
      MPI_Sendrecv(&input[n/SIZE][0], n, MPI_INT, RANK-1, 1,
            &input[n/SIZE+1][0], n, MPI_INT, RANK-1, 1,
            MPI_COMM_WORLD, &status);
   }
}
/* THIS WILL DEADLOCK */
```

Working Exchange

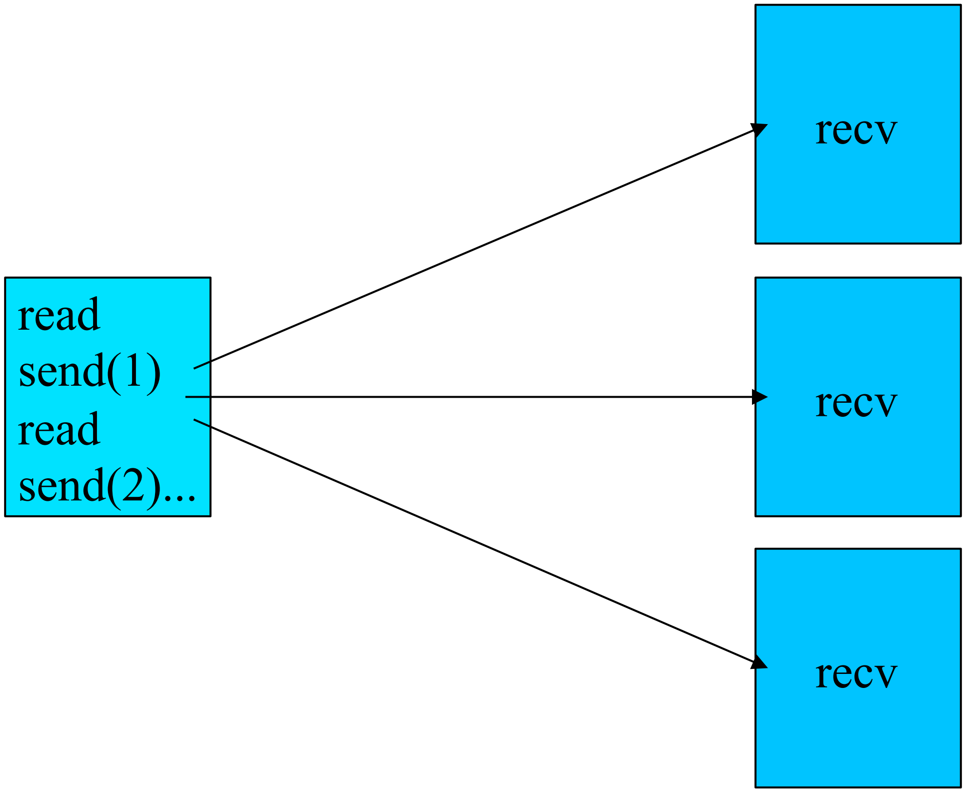
```
exchange_borders(int n; int SIZE, int RANK, int input[][n])
{
   MPI_Status status;
   if (RANK == 0)
      MPI_Send(&input[n/SIZE][0], n, MPI_INT, RANK+1, 1,
             MPI_COMM_WORLD);
   else if (RANK < SIZE-1)
      MPI_Sendrecv(&input[n/SIZE][0], n, MPI_INT, RANK+1, 1,
             &input[0][0], n, MPI_INT, RANK-1, 1,
             MPI_COMM_WORLD, &status);
   else
      MPI_Recv(&input[0][0], n, MPI_INT, RANK-1, 1,
             MPI_COMM_WORLD, &status);
   if (RANK == 0)
      MPI_Recv(&input[n/SIZE+1][0], n, MPI_INT, 1, 1,
             MPI_COMM_WORLD);
   else if (RANK < SIZE-1)
      MPI_Sendrecv(&input[1][0], n, MPI_INT, RANK-1, 1,
             &input[n/SIZE+1][0], n, MPI_INT, RANK+1, 1,
             MPI_COMM_WORLD, &status);
   else
      MPI_Send(&input[1][0], n, MPI_INT, RANK-1, 1,
             MPI_COMM_WORLD, &status);
} /* THIS WORKS */
```

Program IO

- Master task IO model
  - -One task (task 0?) does all IO
  - -Sends/recvs from other tasks
  - -A must if data not available on all nodes
- Independent IO model
  - -Each task does its own IO
  - -Each node must have data available
- Hybrid models
  - -Subset of nodes have access to data
- Parallel IO model
  - -System software supports IO by parallel tasks

Master Task IO

- One task reads data, then sends to other tasks
- Other tasks receive data from IO task

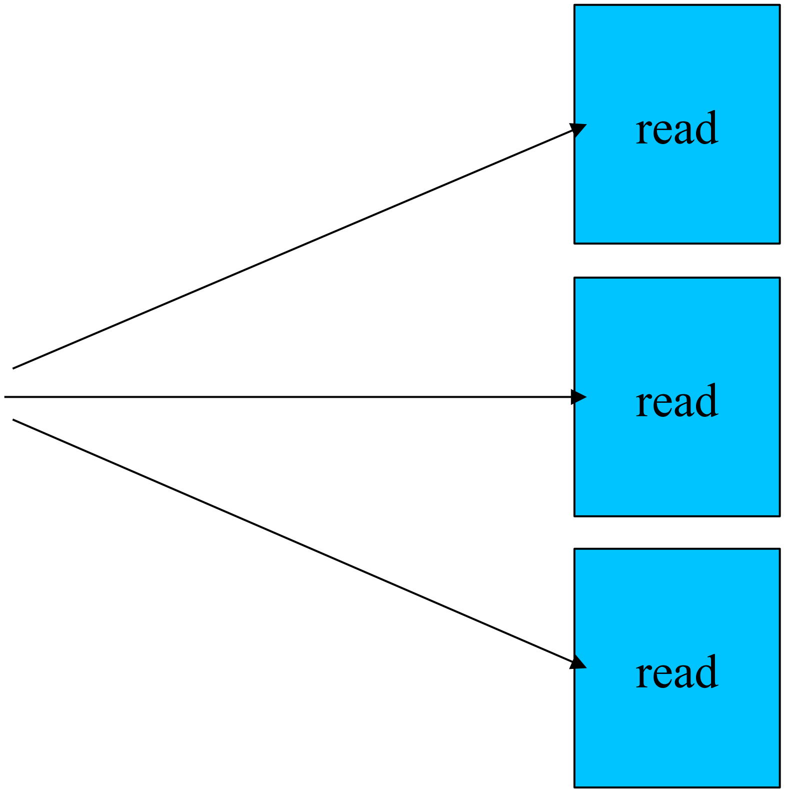


Master task IO

```
parallel_read_buffer (int fd, char *buffer, int count)
{
    if (RANK == 0)
    {
        int t;
        lseek (fd, count, SEEK_SET); /* skip task 0's data */
        for (t = 1; t < SIZE; t++)
        {
            read(fd, count, buffer); /* assume task data is sequential */
            MPI_Send(buffer, count, MPI_BYTE, t, 1, MPI_COMM_WORLD);
        }
        lseek (fd, 0, SEEK_SET); /* now read task 0's data */
        read(fd, count, buffer);
    }
    else
    {
        MPI_Status status;
        MPI_Recv(buffer, count, MPI_BYTE, 0, 1,
                MPI_COMM_WORLD, &status);
    }
}
```

Independent IO

- Each task reads it own data
- Each task must determine which data to read



Independent IO

```
parallel_read_buffer (int fd, char *buffer, int count)
{
    int start;
    start = RANK * count;
    lseek (fd, start, SEEK_SET);
    read(fd, count, buffer);
}
```

Block Decomposition Macros

```
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))

#define BLOCK_HIGH(id,p,n) \
        (BLOCK_LOW((id)+1,p,n)-1)

#define BLOCK_SIZE(id,p,n) \
        (BLOCK_LOW((id)+1)-BLOCK_LOW(id))

#define BLOCK_OWNER(index,p,n) \
        (((p)*((index)+1)-1)/(n))
```

Independent IO - Redux

```
parallel_read_from (int fd, int offset, char *buffer,
       int total_count, int element_size)
{
    int start;
    start = BLOCK_LOW(RANK, SIZE, total_count) * element_size
        + offset;
    lseek (fd, start, SEEK_SET);
    read(fd, BLOCK_SIZE(RANK, SIZE, total_count) * element_size,
        buffer);
}
```

Independent Writes

- Work just like reads except ...
  - Local cache may cause unexpected behavior
  - File creation can be tricky
  - Extending a file can cause data loss
- When using NFS, usually best to ...
  - Have one task create file
  - Have one task pre-allocate file to final length
- Use of MPI-IO (covered later)
  - Should fix task creation/extending problems
  - May fix cache related issues
  - Depends on file system used for implementation

Parallel Open

```
int parallel_open (char *fname, int flags,
        int mode, int size)
{
    int fd;
    char data = 0;
    if (RANK == 0 && (flags&O_WRONLY || flags&O_RDWR))
    {
        fd = open(fname, flags, mode);
        lseek(fd, size-1, SEEK_SET);
        write(fd, &data, 1);
        lseek (fd, 0, SEEK_SET);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    else
    {
        MPI_Barrier(MPI_COMM_WORLD);
        fd = open (fname, flags&(^(O_CREAT|O_TRUNC)), mode);
    }
    return fd;
}
```

Barrier

Barrier synchronizes all tasks of a communicator

```
MPI_Barrier(MPI_COMM_WORLD);
```

Each task calling `MPI_Barrier` will stop until all tasks in the communicator have called `MPI_Barrier`

Running MPI programs with LAM

- LAMMPI already installed on ullab machines
- Compile your programs with `mpicc`
  - -Automatically handles includes and libraries
  - -Otherwise just like `cc` or `gcc`
- Run `lamboot` to start local area machine
  - -All tasks run on local machine
  - -`bhost` file specifies additional machines
  - -Shut down machine with `wipe`
- Run programs with `mpiexec` or `mpirun`

Mpiexec

Runs one or more tasts on nodes:

```
mpiexec [options] prog-name arg1 arg2 ...
-np <int>     run <int> tasks
-localonly    run all tasks on master node
-host <hname> run task on a specific host
-hosts <int> <hname1> <hname2> …
              Run tasks on given hosts
```

Alternative (older) form:

```
mpirun [options] prog-name arg1 arg2 ...
```

Using PBS or Torque

- Batch scheduler on many clusters
  - Launch job by submitting a script
  - Variables control the job

```
#!/bin/bash

### Set the job name
#PBS -N hello
### Time out in 5 minutes
#PBS -l walltime 00:05:00
### Set the number of nodes that will be used.
#PBS -l select=1:ncpus=4:mpiprocs=4:mem=1gb:interconnect=1g
### Tell PBS to keep both stdout and stderr
#PBS -k oe

mpiexec -np 4 hello
```

PBS Commands

- Submit a job

```
qsub <PBS script>
```

- See status of the queue

```
qstat
```

- Kill a job in the queue

```
qdel <job number>
```

PBS assumptions

- PBS runs your script …
  - In your home directory
  - With your standard path
  - You can cd in your script or change env vars
- PBS drops your stdout and stderr
  - You can keep it with a directive
  - Ends up in your home dir with the job number as a tag
- PBS can change working dir for you …
  - But it doesn't always work as expected

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 12, Part 1: Overview of Semantics

Programming Language Semantics

- A programming language is characterized by syntax and semantics.
- The syntax of the language indicates the exact form and structure of elements that may be produced or recognized (parsed),
- The formal semantics of a programming language relate the language syntax to meaning, which is in many cases a quantitative description of the underlying computation. There are a number of different approaches to formally quantifying semantics.

Combining syntax and semantics leads to the oft-cited result:

Programming Language = Syntax + Semantics

Objectives

- Formal approaches to semantics have the goal of **achieving a precise, mathematical characterization of the behavior of an assumed syntactically correct program in a specific language**.
- A standardized and generally-accepted mathematical formalism for this purpose has been an elusive (and sometimes controversial) goal.

Given the following syntactically correct `minic` (or `c`) program:

```
int main(void)
{
int t1;
t1=10;
return(t1);
}
```

The basic question is:

> What is the **meaning** of this program?

To answer this, we must formally and quantitatively define *meaning*.

The Road Map, Updated

concept	process	resulting mapping
lexical analysis	scanning	input file → Tokens
syntactic analysis	parsing	Tokens → abstract parse tree
semantic analysis	parse interpretation	abstract parse tree → semantic objects

Table 1: Enhanced Interpretation Processes for a Programming Language

Semantics, Semantics, Semantics

Our overall motivation is to:

- Describe what a program does or produces (usually by description of *components* of the program)
- Describe what happens during the execution of a program (or program component)
- Describe *meaning* of statements in a programming language
- Describe what a syntactic fragment should produce via machine translation.

Typically, any approach to studying semantics has the following characteristics:

- Usually a 'divide and conquer', or **compositional** approach, is taken. We decompose a program into syntactic/semantic fragments and consider these individually. Therefore, we usually consider the tightly coupled syntax → semantics descriptions of these fragments.

- Also, typically an 'abstract' syntax is employed for simplicity and clarity.

The utility of any formal approach to semantics varies with particular approach and overall objective(s) of the semantic formalization. While the approaches we present are 'formal' and usually quantitative, in some sense, they are also:

- (Usually) in a form which allows manipulation,
- Not one approach, but family of strategies, and
- Not fully accepted, i.e., possibly controversial.

World's Simplest Example

Consider the following semantic representation example:

> *Describe the semantics of a syntactically correct arithmetic statement program fragment as the value of the statement.*

Consider the statement fragment[a]:

$$1101$$

[a] Which might be part of a more complex (and complete) assignment statement such as

$$E = V1 + 1101;$$

This fragment could be

1. Interpreted syntactically as a string of digits (assuming it is syntactically correct); and

2. Interpreted or describe semantically as a *value*, i.e., the semantics or meaning could be 1101_{10} or 1101_2.

- This semantic interpretation (evaluation) still requires we understand the positional notation used in the syntactic specification.

- To formalize the semantic specification (in this case determining value), we could develop a function `evaluate`, i.e.,

  $\texttt{evaluate}[1101] =$

  $\texttt{evaluate}[1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0] =$

  1101

- Unfortunately, the simplicity of this 'value' example obscures some difficult questions in formalizing semantics:
  - What about the semantics of control or print statements?
  - Whis is the role of the machine environment?

Informal Operational Semantics

- An approach based upon specifying the operation of code fragments on an assumed machine.
- Typically used in ’language references’ , ’how to program in xxx’ and ’introduction to xxx’ type programming language references and documentation.
- The way most programming is (initially) taught/learned.

An example in `c` or `c++`.

In this form, the semantic description is often preceded by a quasi-formal syntactic description or prototype such as:

```
if (<condition>) <statement>
if (<condition>) <statement> else <statement>
```

This is then followed with an (informal) semantic description, for example:

> *In an* `if` *statement, the first (or only) statement is executed if the expression is nonzero and the second statement (if it is specified) is executed otherwise. This implies...*

The Plan for the Study of Semantics

In what follows, we first give an overview of a number of differnt approaches. Then, several approaches are considered in detail.

Semantics via ~~Self-definition~~

An operational self-definition of the language is given by defining the interpretation of the language *in the language itself*, e.g., by showing an implementation of:

- A Lisp interpreter written in Lisp,
- A Prolog interpreter in written in Prolog, or
- A c interpreter written in c.

To some, this appears to be circular reasoning.

Translational Semantics Overview

- Semantics are conveyed by showing the translation of program fragments (syntactically correct statements) into another (e.g., assembly language) language called the target language.
- This requires an actual or hypothetical machine (model), i.e., the target machine.
- Another viewpoint of translational semantics is that the semantics of the language are conveyed by defining an interpreter or compiler for the language.
- Translational semantics is a significant approach for compiler writers; but is of limited general utility in learning or expressing semantics.

A `minic` Example of Translational Semantics. To illustrate translational semantics, we return to the simple `minic` program:

```
int main(void)
{
int t1;
t1=10;
return(t1);
}
```

Using `gcc`, the result (semantics) is the assembly code shown on the next slide.

```
     .file     "minicex1.c"
     .version     "01.01"
gcc2_compiled.:
.text
     .align 4
.globl main
     .type      main,@function
main:
     pushl     %ebp
     movl     %esp, %ebp
     subl     $4, %esp
     movl     $10, -4(%ebp)
     movl     -4(%ebp), %eax
     movl     %eax, %eax
     leave
     ret
.Lfe1:
     .size      main,.Lfe1-main
     .ident     "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-98)"
```

Operational Semantics Overview

- Operational semantics convey language semantics by defining *how a computation is performed*, i.e., by describing the **actions** of the program fragment in terms of an actual or hypothetical machine.
- Operational semantics is also referred to as *intensional semantics*, because the sequence of internal computation steps (the intension) is most important.
- *informal* operational semantics is the typical means of describing programming languages.

It is noteworthy that in operational semantics:

- An actual or hypothetical machine (model), i.e., the target machine is required.
- A **precise** description of the machine is required.
- The machines used tend to be simple.

Operational Semantics Example

Suppose the state of the machine consists of the values of all registers, memory locations and condition codes and status registers.

We show the semantics with the following steps:

1. Record the machine state prior to execution/evaluation of the code fragment.
2. Execute/evaluate the code fragment (assume termination).
3. Examine the new machine state.

Thus, the semantics of the code fragment are conveyed by the change in the machine state.

Denotational Semantics Overview

- *The semantics of an assumed syntactically correct program are represented by a mathematical mapping.*
- The steps taken to calculate the output are unimportant; it is the functional relationship of input to output that matters.
- Denotational semantics is also called *extensional semantics*, because the 'extension' or relation between input and output is the focus.
- Specifically, denotational semantics is used to map syntactic objects into domains of mathematical objects, i.e., an overall function is postulated with:

$$\texttt{meaning} : \texttt{Syntax} \rightarrow \texttt{Semantics}$$

- The divide-and-conquer approach is important in denotational semantics, in the sense that the overall functional mapping is subdivided into a composition of functions and each of the composite functions is related to a program fragment.

Simple Denotational Semantics Example

Recall our previous simplistic example, i.e., the semantic interpretation of the string 1101, i.e., the value 1101. Denotational semantics approaches this by defining a semantic function, D, with the corresponding mapping as follows:

$$D : digit|digit - string \rightarrow integer\ value$$

Notice the domain of this function is a syntactic construct and the range is a semantic concept.

Axiomatic Semantics Overview

- Axiomatic semantics employ logic to convey meaning.
- The meaning of a syntactically correct program is formalized as a logical proposition or assertion (sometimes used as a program specification) that constrains the mapping between program input and output.
- The program semantics are based on assertions about logical relationships that remain the same each time the program executes.
- The basic representational strategy is:

$$\{logical\ preconditions\} \cap \{syntactic\ fragment\} \rightarrow \{logical\ postconditions\}$$

- Axiomatic semantics are more abstract than denotational semantics, and involve assertions and logical formulas from the predicate logic.
- Axiomatic semantics are **machine independent**.

Algebraic Semantics Overview

In algebraic semantics, an algebraic specification of data and language constructs is developed. This approach is noteworthy, since it leads to the notion of abstract data types, which in turn could be thought of as implying OO-programming.

Semantic Equivalence

Two different program fragments may have exactly the same meaning or semantics.

For example, consider the two `minic` fragments:

```
t1 = 5 + 4;
```

and

```
t1= 27 / 3;
```

Notice the similarity between constructs such as

`for(;;)`

and

`while(true)`

This raises the often important issue of *semantic equivalence*.

The exact definition of semantic equivalence depends upon the specific semantic formalism used.

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 12, Part 4: Axiomatic Semantics

Axiomatic Semantics

- Axiomatic semantics provides a machine-independent formalism for quantifying semantics.
- The development of axiomatic semantics is more abstract than denotational semantics, and involves assertions and logical formulas from predicate logic.

Axiomatic Semantics Development

- Assume an imperative language.
- In axiomatic semantics, the program semantics reduces to the semantics of a statement (derived from `<command>`) which is itself usually a sequence of commands derived from `<command-seq>`.
- The statement semantics are described by the addition of assertions that are always TRUE when control of the program reaches the assertions.
- In this sense, *program meaning* = *program correctness* with respect to a specification are given by pre and post condition assertions.
- As with denotational semantics, the divide-and-conquer approach is fundamental.

Axiomatic semantics is commonly used for three primary objectives:

1. To prove programs 'correct';
2. To provide formal specifications from which programs may be derived; and
3. (Our interest:) to develop a formal and quantitative description of the semantics of syntactically correct program fragments.

The relation between an initial logical assertion and a final logical assertion following a code fragment captures the essence of the code fragment semantics.

A Sample Assertion

A typical assertion (where $\cap$ indicates conjunction or AND) might look like:

$$\{m < 5 \ \cap \ k = n^2\}$$

where m, n and k are *variables in predicate calculus which are related to program (fragment) variables.*

The truth value of the sample assertion depends upon the values of m, n and k.

We are concerned with two classes of assertions:

1. Assertions about entities that are TRUE just before execution of the code fragment. These are represented using a set of preconditions, $\{P\}$.

2. Assertions about entities that are TRUE just after the execution of the code fragment. These are represented using a set of postconditions, $\{Q\}$.

Using these yields an axiomatics semantic description of a code fragment of the form:

$$\{P\}\ \texttt{<code fragment>}\ \{Q\}$$

The program semantics are based on assertions about logical relationships that remain the same each time the program executes.

The basic representational strategy[a] is:

$$\{logical\ preconditions\} \cap \{syntactic\ fragment\} \rightarrow \{logical\ postconditions\}$$

[a] Note this involves implication.

With respect to axiomatic semantics:

- Variables in the predicate calculus (logical description) correspond to program (fragment) variables. Note that variables in the logical description get values from the corresponding program variables.
- Function symbols (connectives) in the logical description include all the operations available in the programming language. This includes $<$, $>$, $*$ and so forth.
- The truth value of a statement in predicate logic is dependent upon the values of component variables.
- *Note a significant distinction is made between* $=$ *in logic* (equality test) *and the use of* `=` (assignment) *in the programming language.*

Implication Review

Implication is another important logical connective and plays a role in the rules which accompany axiomatic semantics.

Implication is not the same as equality. The truth table for implication is shown below:

p	q	$p \rightarrow q$	$\neg p \cup q$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

A very important equivalence is:

$$\{(p \rightarrow q) = (\neg p \cup q)\} = T$$

Ordered-Pair and Specification

The axiomatic semantics of `C` are given by the ordered pair

$$(\{P\}, \{Q\})$$

Alternately, $(\{P\}, \{Q\})$ provides a **specification** for code (fragment) `C`.

`minic` Example

A simple `minic` code fragment is shown below. Note the semicolon (;) is only shown to illustrate the termination of the statement. It is **not** part of the string derived from `<command>`.

```
x = 1 / y;
```

One axiomatic representation for this code fragment in the form

$\{P\}$ `<code fragment>` $\{Q\}$ is

$$\{y \neq 0\} \quad \texttt{x = 1 / y} \ \{x = \frac{1}{y}\}$$

It is critical to note in the above example that `x` and `y` are **program variables** whereas x and y are variables in the predicate calculus.

Partial and Total Correctness

Given:

$$\{P\}\ \texttt{C}\ \{Q\}$$

Partial Correctness: A program (fragment) is partially correct if the following compound statement in predicate logic is true:

$$\{P\}\ \cap\ \texttt{C}\text{ executes }\ \rightarrow \{Q\}$$

Total Correctness: Total correctness extends partial correctness with the additional requirement that the program (fragment) is guaranteed to terminate. In other words,

Partial Correctness + Termination = Total Correctness

Lack of Uniqueness of the Axiomatic Specification

Axiomatic representations are not unique, and this is perhaps the Achilles heel of the technique.

Axiomatic Descriptions Using `minic`

Consider the `minic` program fragment involving assignment and an expression:

```
x = y + 1;
```

Suppose further we are given an axiomatic specification corresponding to this fragment as

$$\{y = -3\}\ \texttt{x = y + 1}\ \{x < 0\} \tag{1}$$

Correctness of the Sample Specification

Given the specification $\{y = -3\}$ `x = y + 1` $\{x < 0\}$, note y and x are logical variables and `y` and `x` are program variables. We proceed in the following manner:

1. Assume $\{y = -3\} =$ TRUE. This requires that y (the logical variable) has the value -3. Note this is not assignment, but rather $=$ is used as a logical connective, in this case the test for equality. This also constrains the corresponding program variable, `y`.

2. Assume `x = y + 1` executes.

3. Since `y` is constrained by the value of y, use the substitution indicated in the program fragment to get the value of `x`, and consequently x. Note we are not done.

4. $\{y = -3\}$ and the substitution $\rightarrow$ $\{x = -2\}$ (is TRUE).

5. Observe $\{x = -2\} \to \{x < 0\}$. Now we are done, except for the following important observation.

In the previous example, be careful to note

$$\{x = -2\} \to \{x < 0\}$$

This is implication, not equality, and the converse

$$\{x < 0\} \to \{x = -2\}$$

is not true.

Alternative Specifications for the Same Fragment

An alternate is

$$\{y = 3\}\ \texttt{x = y + 1}\ \{x > 0\} \tag{2}$$

Thus (so far), the axiomatic specification of program fragment `x = y + 1` is either the pair

$$(\{y = -3\}, \{x < 0\})$$

or

$$(\{y = 3\}, \{x > 0\})$$

Furthermore, *another* possible precondition *for the same code fragment and* $\{Q\}$ in (2) is

$$\{P\} = \{y \geq 0\} \tag{3}$$

Standardization: The Weakest Precondition

Formal Definition. Given `C` and $\{Q\}$, the weakest precondition, denoted $\{W\}$, is defined as follows. Suppose $\{P\}$ is any precondition for which

$$\{P\}\ \texttt{C}\ \{Q\}$$

holds. If

$$\{P\} \rightarrow \{W\}$$

(note this is logical implication), then $\{W\}$ is the weakest precondition.

Axiom for Command Sequencing

This applies to a minic command sequence of the form `C1;C2`, where `C1` and `C2` are commands.

$$\frac{\{P\}\ \mathrm{C1}\ \{Q1\} \cap \{Q1\}\ \mathrm{C2}\ \{Q2\}}{\{P\}\ \mathrm{C1; C2}\{Q2\}} \tag{4}$$

For clarity or to avoid possible ambiguity we write this as:

$$\frac{[\{P\}\ \mathrm{C1}\ \{Q1\}] \cap [\{Q1\}\ \mathrm{C2}\ \{Q2\}]}{\{P\}\ \mathrm{C1; C2}\{Q2\}} \tag{5}$$

Axiom for Assignment

The semantics of syntactic constructs using `=` (assignment) must be carefully considered in order to avoid developing nonsensical axiomatic semantics descriptions.

Formulation #1.

$$\{true\}\ \texttt{x = x + 1}\ \{x = x + 1\} \tag{6}$$

Notice the direct mapping of program variables and connectives to logical variables and connectives in (6), most importantly the (erroneous) mapping of $=$. Note the postcondition *must be false*, i.e., there is no value of x such that $x = x + 1$ is TRUE in logic.

The attempt to capture the axiomatic semantics of assignment via logical variable x and the $=$ connective in logic fails.

Formulation #2. We try to improve upon the previous axiomatic semantics description with the following specification:

$$\{x = A\}\ \texttt{x = x + 1}\ \{x = A + 1\}$$

Whereas this captures the intuitive notion of assignment in a logically correct manner, it is not very general.

The Axiom for Assignment

`v = e` is a common command in imperative languages.

To develop the pre/post-condition pairs, the following is useful:

$$\{[e/v]Q\}\ \texttt{v = e}\ \{Q\} \qquad (7)$$

where `v` is derived from `<identifier>` and `e` is derived from `<expr>`. Furthermore, the notation $[e/v]$ means the substitution of e for all free occurrences of v in postcondition Q.

For example, consider the `minic` fragment `y = x+y`. An axiomatic semantics description might be:

$$\{x = 1, x + y > 2\}\ \texttt{y = x + y}\ \{x = 1, y > 2\}$$

where we have used $[x + y/y]\{x = 1, y > 2\}$ to produce $\{x = 1, x + y > 2\}$.

Using The Assignment Axiom

Consider a previous example:

$$\{x = A\}\ \texttt{x = x + 1}\ \{x = A + 1\}$$

Given {Q} = {x=A+1} we can derive {P} via:

$$[\frac{x+1}{x}]\ \{x = A + 1\} = \{x + 1 = A + 1\} = \{x = A\}$$

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 13, Part 1: Event-driven Programming (with GUI)

Event Driven Programming

Examples:

- Xwindows
- Windows (API)
- Windows (MFC)
- PalmOS
- OpenGL
- wxWidgets

A Long Time Ago...

A long time ago, in a galaxy far, far away...

When a program executed, the program code controlled what happened through the sequence of program statements..... The program might go so far as to occasionally prompt the user for input and print output, but the user had no or little control over the order in which events took place while the program was running.....

The (CLI) Typical Sequence of Events

In the 'old days' of command-line interface-based programs, the program followed a deterministic sequence of events:

`input (read)` → `compute` → `output (print)` → `halt`

- This paradigm is 'programmer driven control flow', since the *programmer* decides which part of the program will be run when and when.

- Input is solicited from the user under program control.

Event-Driven Programming- The Paradigm Shift

- *Instead of prompting for user input, it is the job of the program to listen for and respond to user actions (events) which may occur.*
- Typical events include mouse movements, engaging push buttons and sliders, menu selections, and data entry in prespecified fields on the GUI.

This basic programming change required in event-driven program leads to a number of conceptual changes on the part of the programmer. Specifically:

- Event-driven programs have an event loop that waits for and handles events.
- Program control is 'given up' to this loop. When run under a window(ing) system, event-driven systems rely on window(ing) software libraries and utilities.

Creating Event-Driven (and GUI) Software

1. We must determine what the graphical or visual user interface looks like, i.e., we must provide code which specifies the GUI on the chosen hardware display device.

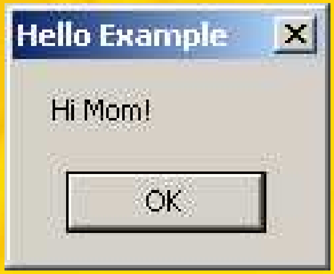
2. We must anticipate all possible events. Otherwise, at best, the user may create inputs the program cannot ’see’.

3. We must associate each possible event with some code which reacts to or ’handles’ the event. This mapping of events to handling code is often referred to as ’registering the handlers for events’.

4. We usually interact with the OS or libraries. In fact, the OS may serve as an intermediary between the event and our code handler.

5. Typically, we program to an API, which in most cases is a library (e.g., Xlib, Motif or OpenGL) or an OS (e.g., PalmOS, MS Windows).

Example: Microsoft Windows (API)



Hello Example

Hi Mom!

```
#include <windows.h>

   int WINAPI WinMain (HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       PSTR szCmdLine,
                       int iCmdShow)
   {
      MessageBox (NULL, "Hi Mom!", "Hello Example", MB_OK);
      return (0);
   }
```

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 13, Part 2: wxWidgets

Cross and Multi-Platform Software Development

Cross-platform development of software involves developing software on one type of machine to run on another type of machine.

1. It is both necessary and common.
2. Often it is necessary to develop applications which run on multiple platforms.
3. It is somewhat inefficient to develop separate applications for each platform.
4. Numerous tool exist to facilitate multi-platform and cross-platform software development.

Cross-Platform Development: wxWidgets

1. Like the MFC, a tool based upon a class library for developing GUI-oriented C++ programs on a variety of different platforms.

2. Defines a common API across platforms, but uses the native graphical user interface (GUI) on each platform.

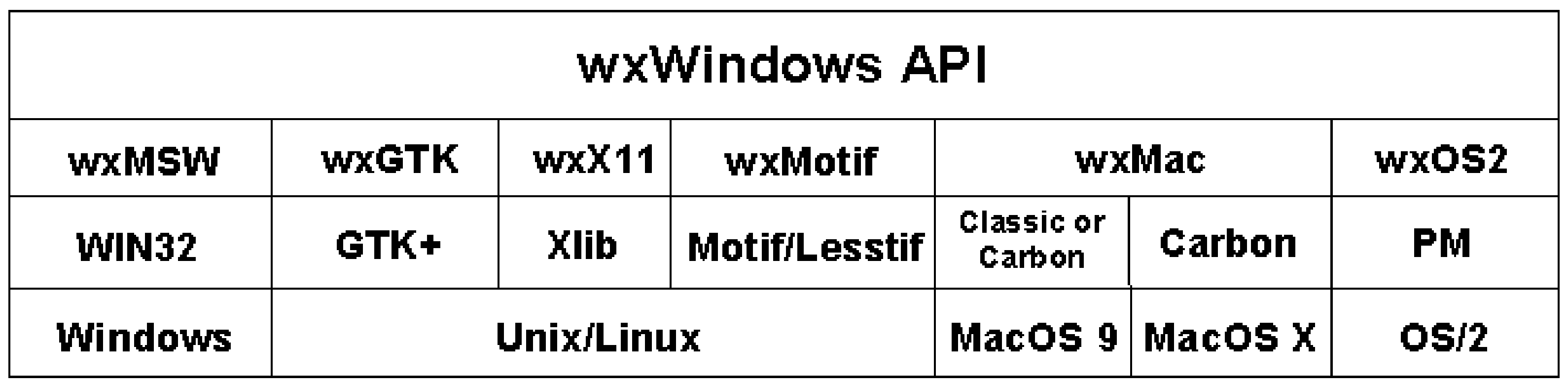
3. Programs developed with wxWidgets exhibit the native 'look and feel' for each platform, yet have common functionality.

4. Distributions of wxWidgets and documentation are available at:

   `http://wxidgets.org/`

Platform-Specific Libraries

Once a program is written to the wxWidgets API, the platform-specific libraries are invoked by the wxWidgets API. This is shown in Figure 1.



<table>
<tr><td colspan="7">wxWindows API</td></tr>
<tr><td>wxMSW</td><td>wxGTK</td><td>wxX11</td><td>wxMotif</td><td colspan="2">wxMac</td><td>wxOS2</td></tr>
<tr><td>WIN32</td><td>GTK+</td><td>Xlib</td><td>Motif/Lesstif</td><td>Classic or Carbon</td><td>Carbon</td><td>PM</td></tr>
<tr><td>Windows</td><td colspan="3">Unix/Linux</td><td>MacOS 9</td><td>MacOS X</td><td>OS/2</td></tr>
</table>

Figure 1: Relationship of the wxWidgets API to Platform-Specific Graphics Libraries

`wxWidgets` Fundamentals: General Notes

1. All wxWidgets applications need need a derived wxApp class and to override the constructor `wxApp::OnInit`.
2. Every application must have a top-level wxFrame or wxDialog window, derived from the respective class.
   - Each frame may contain one or more instances of classes such as wxPanel, wxSplitterWindow or other windows and controls.
   - A frame can have a wxMenuBar, a wxToolBar, a status line, and a wxIcon (used when the frame is 'iconized'). wxPanel is used to place controls (classes derived from wxControl which are used for user interaction.
   - Examples of controls are wxButton, wxCheckBox, wxChoice , wxListBox and wxSlider.

3. An instance of wxDialog can be used for implementing controls. This strategy has the advantage of not requiring a separate frame. This is shown in the 'button' example.
4. For dialogs, the use of wxBoxSizer, for simple windows, may produce acceptable layouts with a minimum of design effort and coding.
5. Argument default values may be omitted from a function call. In addition, size and position arguments may usually be given a value of -1 (the default), in which case wxWidgets will choose a suitable value.
6. Like the MFC, windows (frames and dialogs) and controls in `wxWidgets` programs are referenced by pointers to objects and thus created using `new`[a].

[a] Note `delete` is not used to free these resources because `wxWidgets` takes care of this.

7. An event table is used to map events to (handler) functions. Registering events with user-written functions is achieved using one or more

   BEGIN_EVENT_TABLE ... END_EVENT_TABLE macros. An example is:

   ```
   BEGIN_EVENT_TABLE(ButtonDlg, wxDialog)
     EVT_BUTTON(BUTTON_SELECT, ButtonDlg::ButtonSelect)
     EVT_CLOSE(ButtonDlg::OnCloseWindow)
   END_EVENT_TABLE()
   ```

   Note that the event table is *specific to a frame.* In the above example, the BEGIN_EVENT_TABLE macro indicates that events from the ButtonDlg frame (derived from the wxDialog class) are intercepted and handled. Also note that a DECLARE_EVENT_TABLE macro must be included in the corresponding class definition.

Where's `main`?

Like the MFC approach for MS Windows, wxWidgets `c++` source files for applications *do not explicitly contain the* `main` *function.* In wxWidgets, the `IMPLEMENT_APP` macro creates an application instance and starts the wxWidgets program. The prototype is:

```
IMPLEMENT_APP(theApp)
```

where class `theApp` is derived from class `wxApp`. Recall `wxApp::OnInit()` is called upon class instance creation (startup) and is used to start the wxWidgets program.

wxWidgets: Events and Handling

1. The mapping of events to functions is achieved using a `BEGIN_EVENT_TABLE ... END_EVENT_TABLE` macro block.
2. Between these macros, specific event macros corresponding to specific resources are defined. These map the event (e.g., a button press or a mouse click) to an event handling function.
3. For example, consider the use of a button. The `EVT_BUTTON` macro may be used. The basic prototype is:

   ```
   EVT_BUTTON(id, func)
   ```

   which indicates `func` is invoked when a user clicks the button with identifier `id`.

A Dialog and Button-based wxWidgets Example

```
#include "wx/wx.h"

class ButtonApp : public wxApp
{
public:
// Initialize the application
        virtual bool OnInit();
};

//main window is from wxDialog
class ButtonDlg : public wxDialog
{
public:
// constructor
  ButtonDlg();

  void ButtonSelect(wxCommandEvent &event);
  void OnCloseWindow(wxCloseEvent &event);
```

```
private:
  DECLARE_EVENT_TABLE()
  // ID of only event
  enum
  {
    BUTTON_SELECT = 1000
  };
};

ButtonDlg::ButtonDlg() : wxDialog((wxDialog *)NULL, -1, "ButtonDialog",
                                  wxDefaultPosition, wxSize(150, 150))
{
  wxButton *button = new wxButton(this, BUTTON_SELECT, "Select", wxDefaultPosition
  // Setting the button in the middle of the dialog.
  wxBoxSizer *dlgSizer = new wxBoxSizer(wxHORIZONTAL);
  wxBoxSizer *buttonSizer = new wxBoxSizer(wxVERTICAL);
  buttonSizer->Add(button, 0, wxALIGN_CENTER);
  dlgSizer->Add(buttonSizer, 1, wxALIGN_CENTER);
  SetSizer(dlgSizer);
  dlgSizer->SetSizeHints(this);
}
```

```
void ButtonDlg::ButtonSelect(wxCommandEvent &command)
{
  wxMessageBox("The button was pressed!");
}

// here’s the event table

BEGIN_EVENT_TABLE(ButtonDlg, wxDialog)
  EVT_BUTTON(BUTTON_SELECT, ButtonDlg::ButtonSelect)
// terminate the dialog when the ’x’ window button is clicked
  EVT_CLOSE(ButtonDlg::OnCloseWindow)
END_EVENT_TABLE()

bool ButtonApp::OnInit()
{
// create instance of the dialog
  ButtonDlg *button = new ButtonDlg();
// Show it
  button->Show(TRUE);
// Tell the application that it’s our main window
  SetTopWindow(button);
  return true;
}

void ButtonDlg::OnCloseWindow(wxCloseEvent &event)
{
// close using Destroy rather than Close.
this->Destroy();
}

IMPLEMENT_APP(ButtonApp)
```

Noteworthy aspects of the previous example include:

1. The use of ’sizers’, which relieves the programmer of much of the detailed manual layout decisions (see the `wxWidgets` manual);

2. The creation of a button resource via:

   ```
   new wxButton(this, BUTTON_SELECT, "Select", wxDefaultPosition);
   ```

3. The association of the event resulting from clicking on this button and a user-defined method:

   ```
     EVT_BUTTON(BUTTON_SELECT, ButtonDlg::ButtonSelect)
   ```

4. An illustration of the proper way to terminate an application if the main window is closed.

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 15, Part 1: Parallel Programming

Parallel Programming: Algorithms and Implementation

- Key issues of parallel programming:
  - parallel algorithm decomposition
  - parallel hardware development (or availability)
- A very real and significant problem is *how to (perhaps automatically) decompose a given processing task into one which may be executed in parallel segments.*

Questions

1. Where is the parallelism in the desired computation?
2. How do I articulate/exploit it in my programming language?

The answers usually require some consideration of the underlying computational architecture.

A Simple minic Example

Consider the simple minic code fragment:

```
a=10;
b=sqrt(a);
c=a+b;
d=6;
e=sqrt(d);
f=d+e;
g=f+c;
```

Figure 1: Simple minic Code Fragment for Potential Parallelism Analysis

In this example, it is reasonably easy to (informally) identify parts of this code which are suitable for parallel execution. For example, two blocks:

```
a=10;                               d=6;
b=sqrt(a);                          e=sqrt(d);
c=a+b;                              f=d+e;
```

could be computed concurrently. However, the computation required by

```
g=f+c;
```

must necessarily follow these computations.

Example: Control Constructs and Parallelism

Consider the code fragment:

```
b=sqrt(a);
if (b<4) then
    if (a==1) b=4;
c=a+b;
d=6;
if (c>a) then d=1;
e=sqrt(d);
if (e>a) then e=1;
f=d+e;
g=f+c;
```

Figure 2: `minic` Code Fragment With Conditional Constructs

Clearly identifying parallelism in this case is more challenging.

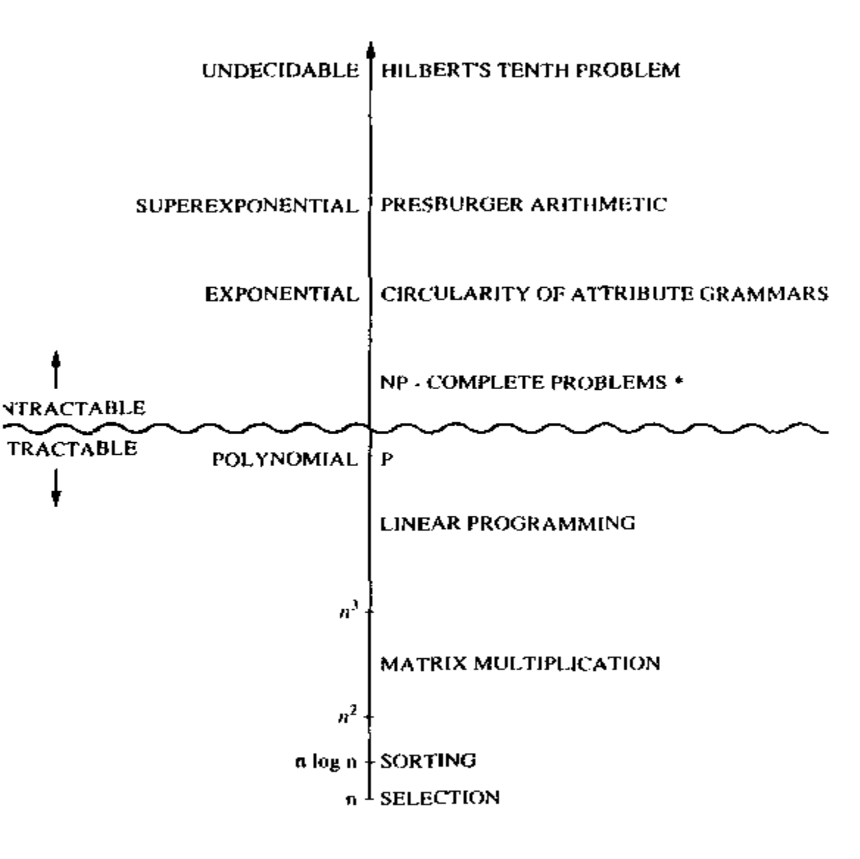


Figure 3: The Spectrum of Computational Complexity

Parallel Performance Limitations

- Intuitively, one might hope that an n processor implementation of an algorithm or parallelized program will achieve the result in $1/n$ (or less) of the time required by a single processor working on the same problem.
- Unfortunately, this is at best an upper bound on the achievable performance.

Metrics for Speedup

For a parallel implementation, *speedup* is the ratio:

$$speedup = \frac{\text{processing time with single processor}}{\text{processing time with } n \text{ processors}} = \frac{t_s}{t_p} \qquad (1)$$

On this basis, the theoretically achievable maximum speedup for an n-processor implementation of an algorithm is n.

Amdahl's Law

Amdahl's law explicitly predicts speedup as a function of the fraction of required serial computations in a given algorithm.

Define the following quantities:

s: the fraction of computations which are necessarily serial;

p : the fraction of computations which may be done in parallel (note $s + p = 1$); and

n : number of processors over which p is distributed.

Amdahl's law predicts speedup as:

$$speedup = \frac{(s+p)}{(s+p/n)} = \frac{1}{(s+p/n)} \tag{2}$$

In general, Amdahl's law is a pessimistic bound on achievable speedup, even for small values of s ($< 5\%$).

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 15, Part 2: MPI

Parallel (Imperative) Programming Language Extensions

Fundamental to any consideration of parallel (declarative) programming languages are three entities:

1. A means to distinguish or delineate a process or code segment and denote the process as PARALLEL, SERIAL or some combination;
2. A means to indicate a block which should be processed in parallel; and
3. A means to allow synchronization or communication between processes.

A Process

For our purposes, a process is an instance of a program running in a computer, and synonymous with the term *task*.

- A process has an associated set of data used to keep track of the process.
- A process can initiate a subprocess, termed a *child* process. The initiating process is referred to as the *parent* process.
- Processes can exchange information or synchronize their operation through several methods of interprocess communication (IPC).
- To allow programming with multiple (concurrent) processes, a technique is needed to spawn multiple parallel processes.

MPI and Beowulf Clusters

- A Beowulf computing cluster is a low-cost, scalable supercomputer using common, off the shelf (COTS) components.
- Most Beowulf PCs currently use a UNIX-like (e.g., linux) OS and thus leverage existing low-cost computing techniques and tools (e.g., linux, GNU tools, MPI, etc.).
- A Beowulf programmer may implement 'conceptual architectures' such as SI(SP)MD, MIMD and tree structured architectures via control of interprocess communications.

MPI

MPI stands for the Message Passing Interface.

- From a programming viewpoint, MPI is a common API for parallel programming.
- There are a number of freely available MPI implementations for heterogeneous networks of workstations and symmetric multiprocessors, based upon both Unix (linux) and Windows NT.
- MPI facilitates processes communication and synchronization using a library of functions.
- MPI allows the overlap of interprocess communication and computation.
- The basic communication primitive in MPI is the transmittal

of data between a pair of processes, i.e. ’point to point communication’.

- The typical application developed using MPI embodies an extension of the SIMD type, denoted the Single Program, Multiple Data (SPMD) computational model. The program is distributed (using `mpirun`) to each the *np* processes, and is executed on each.

Process Rank: 'Who Am I'?

- Each process is identified by a process rank.
- Process ranks are integers and are returned by a call to a communicator using `MPI_Comm_rank( )`.
- *Knowledge of process rank for each process provides a way for the behavior of the distributed program to be tailored to each process.* In addition, interprocess communication is facilitated.

MPI ’Hello World’ Example

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

main(int argc, char** argv) {
        int p; /* Rank of process */
        int n; /* Number of processes */
        int source; /* Rank of sender */
        int dest; /* Rank of receiver */
        int tag = 50; /* Tag for messages */
  char message[100]; // storage for message
        MPI_Status status; /* Return status for receive */


        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &p);
        MPI_Comm_size(MPI_COMM_WORLD, &n);

        if (p != 0) {
                sprintf(message, "Hello World from process %d",
                                p);
                dest = 0;
                MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
                                tag, MPI_COMM_WORLD);
}
        else   {  // p == 0
                for (source = 1; source < n; source++) {
                MPI_Recv(message, 100, MPI_CHAR, source, tag,
                                MPI_COMM_WORLD, &status);
                printf("%s\n", message);
}
}
        MPI_Finalize();
}       /* main */
```

Sample Results: A 16-process version of the program was run:

```
$gcc hello-mpi.c -lmpi -o hello-mpi
$mpirun -np16  hello-mpi
```

with results as shown below:

```
Hello World from process 1
Hello World from process 2
Hello World from process 3
Hello World from process 4
Hello World from process 5
Hello World from process 6
Hello World from process 7
Hello World from process 8
Hello World from process 9
Hello World from process 10
Hello World from process 11
Hello World from process 12
Hello World from process 13
Hello World from process 14
Hello World from process 15
```