Definition:
A statement $S$ depends on a statement $R$ (written $R \rightarrow S$) if there exists an operation $S(x^*_S)$, $R(x^*_R)$, a memory location $m$, and $x^*_S$, $x^*_R$ are respectively two iteration instances such that:

- $S(x^*_S)$ and $R(x^*_R)$ refer to the same memory location $m$, and at least one of them writes to that location.
- $x^*_S$ and $x^*_R$ belongs to the iteration domain of $R$ and $S$.
- In the original sequential order, $R(x^*_R)$ is executed before $S(x^*_S)$. 
Review: Linear Constraints Derived by Dependence Relationship

- $S(\vec{x}_S)$ and $R(\vec{x}_R)$ reference the same memory location:
  
  $$F_s \vec{x}_S + a_S = F_R \vec{x}_R + a_R.$$ 

- $\vec{x}_R$ and $\vec{x}_S$ within loop iteration domains:
  
  $$A_S \vec{x}_S + c_S \geq 0 \text{ and } A_R \vec{x}_R + c_R \geq 0.$$ 

- Precedence order: $R(\vec{x}_R)$ happens before $S(\vec{x}_S)$ for a particular dependence loop level $L$ where the dependence happens,

  $$\begin{align*}
  &\text{for } i < L: \quad x_{R,i} = x_{S,i} \\
  &\text{for } i = L: \\
  &\quad x_{R,L} \leq x_{S,L}
  \end{align*}$$

These linear equalities and inequalities can be rewritten as:

$$P_{l,s} \vec{x}_S - P_{l,R} \vec{x}_R + b \geq 0.$$
The dependence polyhedron for \( R \rightarrow S \) at a given level \( l \) and for a given pair of references \( f_R, f_S \) is described as [Feautrier/Bastoul]:

\[
\mathcal{D}_{R,S,f_R,f_S,l}:
\]

\[
D \left( \begin{array}{c}
\vec{x}_S \\
\vec{x}_R
\end{array} \right) + d = \begin{bmatrix}
F_S & -F_R \\
A_S & 0 \\
0 & A_R \\
P_S & -P_R
\end{bmatrix}
\begin{bmatrix}
\vec{x}_S \\
\vec{x}_R
\end{bmatrix} + \begin{bmatrix}
a_S - a_R \\
c_S \\
c_R \\
b
\end{bmatrix} \geq 0
\]
The example loop
for ( i = 1; i <= 3; ++i ) {
   S1: a[i] = i;
      for ( j = 1; j <= 3; ++j )
         S2: b[j] = (b[j] + a[i])/2;
}

\[ D_{S_2 \delta S_2, <b[i_{S_2'}], b[j_{S_2}]>, 1} : \]
\[ \begin{bmatrix}
0 & 1 & 0 & -1 & 0 \\
1 & 0 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 3 \\
0 & 1 & 0 & 0 & -1 \\
0 & -1 & 0 & 0 & 3 \\
0 & 0 & 1 & 0 & -1 \\
0 & 0 & -1 & 0 & 3 \\
0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & -1 & 3 \\
-1 & 0 & 1 & 0 & -1 \\
\end{bmatrix}
\begin{pmatrix}
i_{S_2'} \\
j_{S_2'} \\
i_{S_2} \\
j_{S_2} \\
1 \\
\end{pmatrix}
\begin{pmatrix}
\leq \vec{0} \\
\geq \vec{0} \\
\end{pmatrix} \]
Step by step:

Initialize DDG graph with one node for every statement
For each pair of statements $R, S$
  For each pair of references $f_R, f_S$
    For each loop level $l$: min_depth to common_depth
      Build dependence polyhedron $D_{R,S,f_R,f_S,l}$
      If it is not empty, then get the dependence type
      Add edge$(R,S,\{l, D_{R,S,f_R,f_S,l,\text{type}}\})$
The example loop

```java
for ( i = 1; i <= 3; ++i ) {
    S1: a[i] = i;
    for ( j = 1; j <= 3; ++j )
        S2: b[j] = (b[j] + a[i])/2;
}
```

Diagram:

```
S1

D_{S1 \delta S2,1, <a[i_{S1}],a[i_{S2}>}

D_{S2 \delta S2,1, <b[j_{S2'}],b[j_{S2}>}
```
Assign a time stamp to every statement instance:

Definition: Affine Schedule
Given a statement $S$, a $p$-dimensional affine schedule $\theta_S$ is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written:

$$\theta_S(\vec{x}_S) = T_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$
Definition: Precedence condition
Given \( \theta_R \) a schedule for the instances of \( R \), \( \theta_S \) a schedule for the instances of \( S \). \( \theta_R \) and \( \theta_S \) are legal schedules if \( \forall (\vec{x}_R, \vec{x}_S) \in \mathcal{D}_{R,S} \)
\[
\theta_R(\vec{x}_R) \prec \theta_S(\vec{x}_S)
\]

Objective function:
Find the best schedule such that it can:
- Minimize latency
- Maximize fine grained parallelism
- Enhance locality
- ...
We start with one dimensional time schedule:

- Find legal affine schedules
- Find best schedule according to the objective function
Review: A naive Approach

- For all pairs of $(\vec{x}_R, \vec{x}_S) \in D_{R,S}$
- Replace the variables $\vec{x}$ with the values of dependence instances above to get a linear system of inequalities
  \[ \theta_R(\vec{x}_R) + 1 \leq \theta_S(\vec{x}_S) \]
- Solve for the affine coefficients of $\theta_R$ and $\theta_S$
  \[ \theta_R(\vec{x}) = \vec{a}_R \vec{x}^l + b_R \]
- Challenge: too many pairs of statement instances in dependence
Theorem
Let $\mathcal{D}$ be a non-empty polyhedron defined by $p$ affine inequalities

$$a_k \bar{x}' + b_k \geq 0, \quad k = 1, p$$

Then an affine form $\phi$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:

$$\phi(x) = \lambda_0 + \sum_k \lambda_k (a_k \bar{x}' + b_k), \quad \lambda_k \geq 0$$

The set of $\lambda_k$ coefficients are called Farkas multipliers.
Review: Application of Farkas Lemma

Two conditions:

- \( \theta(\vec{x}) \) is non-negative
- Precedence condition, for dependence polyhedron:
  \[
  \theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) \geq 0
  \]

Two steps:

- Equate the coefficients of \( x \) variables and \( x \) variables are gone
- Use Fourier-Motzkin elimination to eliminate as many Farkas multipliers as possible so that only affine schedule coefficients are left

Now we only have the affine scheduling coefficients as unknown variables. We add around linear times of \( p \) inequalities, with \( p \) being the dimension of dependence polyhedron compared to \( 2^p \).
Assume iteration polyhedron:

\[ a_{Sk} \begin{pmatrix} x \\ n \end{pmatrix} + b_{Sk} \geq 0, k = 1, m_{S} \]

Dependence polyhedron:

\[ c_{ek} \begin{pmatrix} x \\ y \\ n \end{pmatrix} + d_{ek} \geq 0, k = 1, m_{e} \]
Non-negative schedule constraint:

\[ \theta(S, x) = \mu_{s0} + \sum_{k=1}^{ms} \mu_{sk} (a_{sk} \begin{pmatrix} x \\ n \end{pmatrix} + b_{sk}) \]

Non-negative precedence constraint:

\[ \theta(S, y) - \theta(R, x) - 1 = \lambda_{e0} + \sum_{k=1}^{me} \lambda_{ek} \begin{pmatrix} c_{ek} \begin{pmatrix} x \\ y \\ n \end{pmatrix} + d_{ek} \end{pmatrix} \]
1
for i = 0, n
S1: s(i) = 0
for j = 0, n
S2: s(i) = s(i) + a(i,j)*x(j)
endfor
endfor

Schedules may be written as:
\[ \theta(1, i) = \mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n-i) \]
\[ \theta(2, i, j) = \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n-i) + \mu_{2,3}j + \mu_{2,4}(n-j) \]

Two dependences in DDG:
S1 < S2 when j = 0,
S2(j') < S2(j) when j' = j - 1

2
For the first dependence edge in DDG, use Farkas Lemma:
\[ \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n-i) + \mu_{2,3}j + \mu_{2,4}(n-j) \]
\[ - (\mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n-i)) - 1 \]
\[ = \lambda_{1,0} + \lambda_{1,1}i + \lambda_{1,2}(n-i) + \lambda_{1,3}j + \lambda_{1,4}(n-j) - \lambda_{1,5}j \]

For the second dependence edge in DDG, it is uniform dependence, which is:
\[ \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n-i) + \mu_{2,3}j + \mu_{2,4}(n-j) \]
\[ - (\mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n-i) + \mu_{2,3}(j-1) + \mu_{2,4}(n-j+1)) - 1 \]

And we have,
\[ \mu_{2,3} - \mu_{2,4} - 1 \geq 0 \]

We do not apply Farkas Lemma since it does not depend on loop iterators and therefore does not depend on any constraints on the loop iteration domain.

3
By a process of identification for loop iterator coefficients, we have:
\[ \mu_{2,0} - \mu_{1,0} - 1 = \lambda_{1,0} \]
\[ \mu_{2,1} - \mu_{2,2} - \mu_{1,1} + \mu_{1,2} = \lambda_{1,1} - \lambda_{1,2} \]
\[ \mu_{2,3} - \mu_{2,4} = \lambda_{1,3} - \lambda_{1,4} - \lambda_{1,5} \]
\[ \mu_{2,2} + \mu_{2,4} - \mu_{1,2} = \lambda_{1,2} + \lambda_{1,4} \]

Eliminate as much as unknown as possible. One possible result:
\[ \lambda_{1,0} = \mu_{2,0} - \mu_{1,0} - 1 \geq 0 \]
\[ \lambda_{1,1} = \mu_{2,1} + \mu_{2,4} - \mu_{1,1} - \lambda_{1,4} \geq 0 \]
\[ \lambda_{1,3} = \mu_{2,3} - \mu_{2,4} + \lambda_{1,4} + \lambda_{1,5} \geq 0 \]
\[ \lambda_{1,2} = \mu_{2,2} + \mu_{2,4} - \mu_{1,2} - \lambda_{1,4} \geq 0 \]
\[ \mu_{2,3} - \mu_{2,4} - 1 \geq 1 \]

Finally we have:
\[ 0 \leq \mu_{1,1} \leq \mu_{2,1} + \mu_{2,4} \]
\[ 0 \leq \mu_{1,2} \leq \mu_{2,2} + \mu_{2,4} \]
\[ \mu_{2,0} \geq 1 + \mu_{1,0} \]
\[ \mu_{2,3} \geq 1 + \mu_{2,4} \]
Constraints on $\mu$ coefficients:

\begin{align*}
0 & \leq \mu_{1,1} \leq \mu_{2,1} + \mu_{2,4} \\
0 & \leq \mu_{1,2} \leq \mu_{2,2} + \mu_{2,4} \\
\mu_{2,0} & \geq 1 + \mu_{1,0} \\
\mu_{2,3} & \geq 1 + \mu_{2,4}
\end{align*}

Schedules that satisfy precedence (causality) condition:

\begin{align*}
\theta(1, i) &= \mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n - i) \\
\theta(2, i, j) &= \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n - i) + \mu_{2,3}j + \mu_{2,4}(n - j)
\end{align*}

Possible schedules: practice - how the $\mu$ coefficients are set?

$\theta(1, i) = 0, \theta(2, i, j) = j + 1$

Or:

$\theta(1, i) = i, \theta(2, i, j) = i + j + 1$
Lemma
If all domains are bounded, and if there exists at least one affine schedule \( \theta \), then there exists at least one affine form in the structure parameters:

\[ L = h.n + k \]

such that:

\[ \forall S, x \in D_S : L - \theta(S, x) \geq 0 \]

By Farkas Lemma, we have:

\[ L - \theta(S, x) = v_{S0} + \sum_k v_{Sk} \left( a_{sk} \left( \begin{array}{c} x \\ n \end{array} \right) + b_{sk} \right) \]
Constraints on $\mu$ coefficients:

\[
0 \leq \mu_{1,1} \leq \mu_{2,1} + \mu_{2,4} \\
0 \leq \mu_{1,2} \leq \mu_{2,2} + \mu_{2,4} \\
\mu_{2,0} \geq 1 + \mu_{1,0} \\
\mu_{2,3} \geq 1 + \mu_{2,4}
\]

Schedules that satisfy precedence (causality) condition:

\[
\theta(1, i) = \mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n - i) \\
\theta(2, i, j) = \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n - i) + \mu_{2,3}j + \mu_{2,4}(n - j)
\]

Possible schedules: practice - how the $\mu$ coefficients are set?

\[
\theta(1, i) = 0, \quad \theta(2, i, j) = j + 1
\]

Or:

\[
\theta(1, i) = i, \quad \theta(2, i, j) = i + j + 1
\]
Multi-dimensional Affine Schedule

Recall definition:
Given a statement $S$, a $p$-dimensional affine schedule $\theta_S$ is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written:

$$\theta_S(\vec{x}_S) = T_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$

For precedence rule, if one dimensional, we can simply do:

$$\theta_S(\vec{x}_S) \geq \theta_R(\vec{x}_R) + 1 \text{ if } \theta_R(\vec{x}_R) < \theta_S(\vec{x}_S)$$

Does this inequality apply for multi-dimensional affine schedule at every component of the vector?
A Naive Solution

- Starting from the first component in vector $\theta_S$ and $\theta_R$
- Solve for $\theta_S(i) \geq \theta_R(i)$ as if it is one dimensional schedule
- If there exists solution $\theta_S(i) = \theta_R(i)$, then go to the next component by incrementing $i$, go back to last step
- If there does not exist solution $\theta_S(i) = \theta_R(i)$, terminate here

This approach is complete, however the search space is explosive.
An Improved Solution

- Step 1: Solve $\theta_S(i) > \theta_R(i)$ for some of the dependence edges in DDG
- Step 2: Then solve for the rest dependence edges in DDG by refining solutions from the first step
- Step 3: Repeat Step 1 and Step 2 until all dependence edges are satisfied

There is a termination condition proof from "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time", [Feautrier IJPP’92]

Other possible greedy solutions for different objectives? Open problem ...
References

- "Code Generation in the Polyhedral Model Is Easier Than You Think", PACT’04, Cédric Bastoul
- "Parametrized Polyhedra and Enumerating their Vertices", Widle et al.
- "Parametric integer programming", P. Feautrier
- "Counting Integer Points in Parametric Polytopes using Barvinoks Rational Functions", Verdoolaege et al.
- "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One dimensional Time", Feautrier et al.
- "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time", Feautrier et al.
- "Code Generation in the Polyhedral Model", Griebl et al., PACT’98
Message Passing Interface (MPI)
Review: Parallel Programming Models

- **Shared Memory Programming**  ➤ Pthreads & OpenMP
  - Start a single process and fork threads
  - Threads carry out work
  - Threads communicate through shared memory
  - Threads coordinate through synchronization

- **Distributed Memory Programming** ➤ Message Passing Interface
  - Start multiple processes on multiple systems
  - Processes carry out work
  - Processes communicate through message-passing
  - Processes coordinate either through message-passing or synchronization (generates messages)
• Slides from the book: *Introduction to Parallel Computing* (2nd edition)

Authors: Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar
Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of $p$ processes, each with its own exclusive address space.

- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.

- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.

- These two constraints, while onerous, make underlying costs very explicit to the programmer.
Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.

- In the asynchronous paradigm, all concurrent tasks execute asynchronously.

- In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.

- Most message-passing programs are written using the *single program multiple data* (SPMD) model.
The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

  send(void *sendbuf, int nelems, int dest)
  receive(void *recvbuf, int nelems, int source)

- Consider the following code segments:

  P0
  a = 100;
  send(&a, 1, 1);
  a = 0;

  P1
  receive(&a, 1, 0)
  printf("%d\n", a);

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.

- This motivates the design of the send and receive protocols.
Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.

- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.

- Idling and deadlocks are major issues with non-buffered blocking sends.

- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.

- Buffering alleviates idling at the expense of copying overheads.
Non-Buffered Blocking Message Passing Operations

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.
Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.

- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.

- The data must be buffered at the receiving end as well.

- Buffering trades off idling overhead for buffer copying overhead.
Buffered Blocking Message Passing Operations

Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.
Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

P0

for (i = 0; i < 1000; i++) {
    produce_data(&a);
    send(&a, 1, 1);
}

P1

for (i = 0; i < 1000; i++) {
    receive(&a, 1, 0);
    consume_data(&a);
}

What if consumer was much slower than producer?
Buffered Blocking Message Passing Operations

Deadlocks are still possible with buffering since receive operations block.

P0
receive(&a, 1, 1);
send(&b, 1, 1);

P1
receive(&a, 1, 0);
send(&b, 1, 0);
Non-Blocking Message Passing Operations

- The programmer must ensure semantics of the send and receive.

- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.

- Non-blocking operations are generally accompanied by a check-status operation.

- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.

- Message passing libraries typically provide both blocking and non-blocking primitives.
Non-Blocking Message Passing Operations

(a) Without hardware support

- send
- request to send
- okay to send
- data
- receive
- Unsafe to update data being sent

(b) With hardware support

- send
- request to send
- okay to send
- data
- receive
- Unsafe to update data being sent
- Unsafe to read data being received

Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.
Send and Receive Protocols

<table>
<thead>
<tr>
<th>Buffered</th>
<th>Non-Buffered</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking Operations</strong></td>
<td><strong>Non-Blocking Operations</strong></td>
</tr>
<tr>
<td>Sending process returns after data has been copied into communication buffer</td>
<td>Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return</td>
</tr>
<tr>
<td>Send and Receive semantics assured by corresponding operation</td>
<td>Programmer must explicitly ensure semantics by polling to verify completion</td>
</tr>
</tbody>
</table>

Space of possible protocols for send and receive operations.
MPI in more details
MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.

- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.

- Vendor implementations of MPI are available on almost all commercial parallel computers.

- It is possible to write fully-functional message-passing programs by using only the six routines.
MPI: the Message Passing Interface

The minimal set of MPI routines.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI.</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI.</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes.</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of the calling process.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message.</td>
</tr>
</tbody>
</table>
Starting and Terminating the MPI Library

- **MPI_Init** is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.

- **MPI_Finalize** is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.

- The prototypes of these two functions are:

  ```c
  int MPI_Init(int *argc, char ***argv)
  int MPI_Finalize()
  ```

- **MPI_Init** also strips off any MPI related command-line arguments.

- All MPI routines, data-types, and constants are prefixed by “MPI_”. The return code for successful completion is MPI_SUCCESS.
Communicators

- A communicator defines a *communication domain* – a set of processes that are allowed to communicate with each other.

- Information about communication domains is stored in variables of type `MPI_Comm`.

- Communicators are used as arguments to all message transfer MPI routines.

- A process can belong to many different (possibly overlapping) communication domains.

- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.
Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

  ```c
  int MPI_Comm_size(MPI_Comm comm, int *size)
  int MPI_Comm_rank(MPI_Comm comm, int *rank)
  ```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.
Our First MPI Program

#include <mpi.h>

main(int argc, char *argv[])
{
    int n pes, myrank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n", myrank, npes);
    MPI_Finalize();
}
Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.

- The calling sequences of these routines are as follows:

  ```c
  int MPI_Send(void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm)
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm, MPI_Status *status)
  ```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.

- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.

- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`. 
# MPI Datatypes

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Sendand Receiving Messages

- MPI allows specification of wildcard arguments for both source and tag.

- If source is set to MPI_ANY_SOURCE, then any process of the communication domain can be the source of the message.

- If tag is set to MPI_ANY_TAG, then messages with any tag are accepted.

- On the receive side, the message must be of length equal to or less than the length field specified.
Sending and Receiving Messages

- On the receiving end, the `status` variable can be used to get information about the `MPI_Recv` operation.

- The corresponding data structure contains:

  ```c
  typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
  };
  ```

- The `MPI_Get_count` function returns the precise count of data items received.

  ```c
  int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
  ```
Avoiding Deadlocks

Consider:

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
} else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI_Send is blocking, there is a deadlock.
Avoiding Deadlocks

Consider the following piece of code, in which process \( i \) sends a message to process \( i + 1 \) (modulo the number of processes) and receives a message from process \( i - 1 \) (modulo the number of processes).

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if `MPI_Send` is blocking.
Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```
Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function:

```c
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                 int source, int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```c
int MPI_Sendrecv_replace(void *buf, int count,
                         MPI_Datatype datatype, int dest, int sendtag,
                         int source, int recvtag, MPI_Comm comm,
                         MPI_Status *status)
```
Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

  int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm, MPI_Request *request)
  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
                 int source, int tag, MPI_Comm comm, MPI_Request *request)

- These operations return before the operations have been completed. Function MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished.

  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- MPI_Wait waits for the operation to complete.

  int MPI_Wait(MPI_Request *request, MPI_Status *status)
Avoiding Deadlocks

Using non-blocking operations remove most deadlocks. Consider:

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.
Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.

- Each of these operations is defined over a group corresponding to the communicator.

- All processors in a communicator must call these operations.
Collective Communication Operations

- The barrier synchronization operation is performed in MPI using:

  ```c
  int MPI_BARRIER(MPI_Comm comm)
  ```

- The one-to-all broadcast operation is:

  ```c
  int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
                 int source, MPI_Comm comm)
  ```

- The all-to-one reduction operation is:

  ```c
  int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, int target,
                 MPI_Comm comm)
  ```
Predefined Reduction Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise AND</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise OR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bit-wise XOR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max-min value-location</td>
<td>Data-pairs</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min-min value-location</td>
<td>Data-pairs</td>
</tr>
</tbody>
</table>
Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:

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

- To compute prefix-sums, MPI provides:

  ```c
  int MPI_Scan(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  ```
Collective Communication Operations

- The gather operation is performed in MPI using:

```c
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatypen, void *recvbuf, int recvcount,
               MPI_Datatype recvdatatypen, int target, MPI_Comm comm)
```

- MPI also provides the `MPI_Allgather` function in which the data are gathered at all the processes.

```c
int MPI_Allgather(void *sendbuf, int sendcount,
                   MPI_Datatype senddatatypen, void *recvbuf, int recvcount,
                   MPI_Datatype recvdatatypen, MPI_Comm comm)
```

- The corresponding scatter operation is:

```c
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype senddatatypen, void *recvbuf, int recvcount,
                MPI_Datatype recvdatatypen, int source, MPI_Comm comm)
```
Collective Communication Operations

- The all-to-all personalized communication operation is performed by:

  ```c
  int MPI_Alltoall(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf, int recvcount,
                  MPI_Datatype recvdatatype, MPI_Comm comm)
  ```

- Using this core set of collective operations, a number of programs can be greatly simplified.
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.

- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.

- The simplest such mechanism is:

  ```c
  int MPI_Comm_split(MPI_Comm comm, int color, int key,
                     MPI_Comm *newcomm)
  ```

  This operation groups processors by color and sorts resulting groups on the key.
Groups and Communicators

Using MPI_Comm_split to split a group of processes in a communicator into subgroups.