**Summing \( m \) Numbers**

**Example:** summing \( m \) numbers

On a sequential computer we have,

```plaintext
sum = a[0]  
for (i=1;i<m;i++)  
    sum = sum + a[i]  
end for
```

Would expect running time be be roughly proportional to \( m \). We say that the running time is \( \Theta(m) \).

What if we have \( N \) processors, with each calculating the \( m/N \) numbers assigned to it. We must add these partial sums together to get the total sum.

**Summing Using Shared Memory**

The \( m \) numbers, and the global sum, are held in global shared memory

```plaintext
global_sum = 0  
for (each processor)  
    local_sum = 0  
    calculate local sum of \( m/N \) numbers  
    LOCK  
    global_sum = global_sum + local_sum  
    UNLOCK
end for
```

Since \( \text{global}_\text{sum} \) is a shared variable each processor must have mutually exclusive access to it - otherwise the final answer may be incorrect.

The running time (or algorithm time complexity) is

\[
\Theta(m/N + \Theta(N))
\]

where

- \( m/N \) comes from finding the local sums in parallel
- \( N \) comes from adding \( N \) numbers in sequence

**Summing Using Distributed Memory**

Suppose we have a square mesh of processors \( N \) processors

```
P11  P12  P13  
|    |    |    |    |
P21  P22  P23  
|    |    |    |    |
P31  P32  P33  
```

The algorithm proceeds as follows

1. Each processor finds the local sum of its \( m/N \) numbers
2. Each processor passes its local sum to another processor in a coordinated way
3. The global sum is finally in processor P11.

**Distributed Memory Pseudocode**

```
FORALL processors  
find local_sum of \( m/N \) numbers  
for (j=sqrt(N);j>1;j--)  
    IFPROC in column j  
    send local_sum to neighbor in column j-1  
    END IFPROC  
    IFPROC in column j-1  
    receive local_sum from neighbor in column j  
    local_sum = local_sum + new_local_sum  
    END IFPROC  
end for  
IFPROC in first column  
for (i=sqrt(N);i>1;i--)  
    IFPROC in row i  
    send local_sum to neighbor above  
    END IFPROC  
    IFPROC in row i-1  
    receive new_local_sum from neighbor below  
    local_sum = local_sum + new_local_sum  
    END IFPROC  
end for  
IFPROC in first row  
    global_sum = local_sum
END IFPROC
END FORALL
```
**Summing Example**

There are $\sqrt{N} - 1$ additions and $\sqrt{N} - 1$ communications in each direction, so the total time complexity is

$$\Theta(m/N) + \Theta(\sqrt{N}) + C$$

where $C$ is the time spent communicating.

<table>
<thead>
<tr>
<th>Initially</th>
<th>$j = 3$</th>
<th>$j = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 10 10</td>
<td>10 20</td>
<td>30</td>
</tr>
<tr>
<td>10 10 10</td>
<td>10 20</td>
<td>30</td>
</tr>
<tr>
<td>10 10 10</td>
<td>10 20</td>
<td>30</td>
</tr>
</tbody>
</table>

$i = 3$ $i = 2$

<table>
<thead>
<tr>
<th>30</th>
<th>90</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed-Up and Efficiency**

We now define some metrics which measure how effectively an algorithm expedites parallelism.

- **Speed-up** is the ratio of the time taken to run the best sequential algorithm on one processor of the parallel machine divided by the time to run on $N$ processors of the parallel machine.

$$S(N) = \frac{T_{seq}}{T_{par}(N)}$$

- **Efficiency** is the speed-up per processor.

$$\epsilon(N) = \frac{S(N)}{N} = 1 - \frac{T_{seq}}{N T_{par}(N)}$$

- **Overhead** is defined as

$$f(N) = \frac{1}{\epsilon(N)} - 1$$

**Example**: Suppose the best known sequential algorithm takes 8 seconds, and a parallel algorithm takes 2 seconds on 5 processors. Then

- Speed-up $= \frac{8}{2} = 4$
- Efficiency $= \frac{4}{5} = 0.8$
- Overhead $= 1 - 0.8 = 0.25$

- **Self speed-up** is defined using the parallel algorithm running on one processor.
- If the speed-up using $N$ processors is $N$ then the algorithm is said to exhibit linear speed-up.

**Factors that Limit Speed-Up**

1. **Software overhead**

   Even when the sequential and parallel algorithms perform the same computations, software overhead may be present in the parallel algorithm. This includes additional index calculations necessitated by how the data were decomposed and assigned to processors, and other sorts of “bookkeeping” required by the parallel algorithm but not the sequential algorithm.

2. **Load imbalance overhead**

   Each processor should be assigned the same amount of work to do between synchronization points. Otherwise some processors may be idle while waiting for others to catch up. This is known as load imbalance. The speedup is limited by the slowest processor.

3. **Communication overhead**

   Assuming that communication and calculation cannot be overlapped, then any time spent communicating data between processors reduces the speed-up.

The **grain size** is the amount of work done between communication phases of an algorithm. We want the grain size to be large so the relative impact of communication is less.

**Analysis of Summing Example**

Recall the example of summing $m$ numbers on a square mesh of $N$ processors.

$$N \text{ processors}$$

The algorithm proceeds as follows

1. Each processor finds the local sum of its $m/N$ numbers
2. Each processor passes its local sum to another processor in a coordinated way
3. The global sum is finally in processor P11.
Analysis of Summing Example

Time for best sequential algorithm is

\[ T_{\text{seq}} = (m - 1)t_{\text{calc}}. \]

where \( t_{\text{calc}} \) is time to perform one floating-point operation.

Time for each phase of parallel algorithm

1. Form local sums
\[ T_1 = (m/N - 1)t_{\text{calc}} \]

2. Sum along processor rows
\[ T_2 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}}) \]

where \( t_{\text{comm}} \) is time to communicate one floating-point number between neighbouring processors.

3. Sum up first column of processors
\[ T_3 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}}) \]

Total time for parallel algorithm is
\[ T_{\text{par}}(N) = (m/N + 2\sqrt{N} - 3)t_{\text{calc}} + 2(\sqrt{N} - 1)t_{\text{comm}} \]

Analysis of Summing Example

So the speed-up for the summing example is

\[ S(N) = \frac{(m - 1)t_{\text{calc}}}{(m/N + 2\sqrt{N} - 3)t_{\text{calc}} + 2(\sqrt{N} - 1)t_{\text{comm}}} \]

\[ = \frac{1/N}{1 + (N/m)(2\sqrt{N} - 3) + 2(N/m)(\sqrt{N} - 1)/\tau} \]

where \( \tau = t_{\text{comm}}/t_{\text{calc}}. \)

In this algorithm a good measure of the grain size, \( g \), is the number of elements per processor, \( m/N \). We can write \( S \) as

\[ S(g, N) = \frac{(N - 1)/g}{1 + (2\sqrt{N} - 3)/g + 2(\sqrt{N} - 1)/g} \]

\[ = \frac{N}{1 + 2\sqrt{N}(1 + \tau)/g} \]

- As \( g \to \infty \) with \( N \) constant, \( S \to N. \)
- As \( N \to \infty \) with \( g \) constant, \( S \approx gN/(2(1 + \tau)). \)
- As \( N \to \infty \) with \( m \) constant, \( S \to 0. \)

If \( m \gg 1 \) and \( N \gg 1 \),

\[ S = \frac{N}{1 + 2\sqrt{N}(1 + \tau)/g} \]

\[ = \frac{1}{1 + 2\sqrt{N}(1 + \tau)/g} \]

\[ f = 2\sqrt{N}(1 + \tau)/g \]

Scalable Algorithms

- Scalability is a measure of how effectively an algorithm makes use of additional processors.
- An algorithm is said to be scalable if it is possible to keep the efficiency constant by increasing the problem size as the number of processors increases.

The summing algorithm is scalable since we can take \( g \propto \sqrt{N}. \)

- An algorithm is said to be perfectly scalable if the efficiency remains constant when the problem size and the number of processors increase by the same factor.

The summing algorithm is not perfectly scalable.

- An algorithm is said to be highly scalable if the efficiency depends only weakly on the number of processors when the problem size and the number of processors increase by the same factor.

The summing algorithm is highly scalable.

- “Problem size” may be either
  - the work performed, or
  - the size of the data

Amdahl’s Law

Amdahl’s Law states that the maximum speedup of an algorithm is limited by the relative number of operations that must be performed sequentially, i.e., by its serial fraction.

If \( \alpha \) is the serial fraction, \( n \) is the number of operations in the sequential algorithm, and \( N \) the number of processors, then the time for the parallel algorithm is

\[ T_{\text{par}}(N) = (1 - \alpha)n/N + \alpha n + C \]

where \( C \) is the time for overhead due to communication, load balancing, etc. Then the speed-up satisfies

\[ S(N) = \frac{n}{(1 - \alpha)n/N + \alpha n} \approx \frac{1}{(1 - \alpha)/N + \alpha} \]

Note that

\[ S(N) \to \frac{1}{\alpha} \text{ as } N \to \infty \]

so the speed-up is always limited to a maximum of \( 1/\alpha \), no matter how many processors are used.
Example of Amdahl’s Law

Consider the effect of Amdahl’s Law on speed-up as a function of serial fraction, $\alpha$, for $N = 10$ and $N = 1024$ processors.

If 1% of a parallel program involves serial code, the maximum speed-up is 9 on a 10-processor machine, but only 91 on a 1024-processor machine.

Scaled Speed-Up

- *Scaled speed-up* is the ratio between how long the best sequential algorithm would take on a single processor and how long it actually took to run on multiple processors.
- Speed-up and scaled speed-up differ because to measure speed-up the problem must be small enough to fit into the memory of one processor. This limits us to measuring the speed-up of only small problems.
- In finding the scaled speedup we are allowed to estimate the time to run on one processor, so much larger problems can be considered.
- In general overhead costs increase with problem size, but at a slower rate than the grain size. Thus, speed-up is an increasing function of problem size, and so use of scaled speed-up allows us to measure larger speed-ups.

Implications of Amdahl’s Law

- Amdahl’s Law says that the serial fraction puts a severe constraint on the speed-up that can be achieved as the number of processors increases.
- Amdahl’s Law suggests that it is not cost effective to build systems with large numbers of processors because sufficient speed-up will not be achieved.
- It turns out that most important applications that need to be parallelised contain very small serial fractions, so large machines are justified.

Speed-Up and Problem Size

For a given number of processors, speedup increases with problem size, $m$.
Semantics of Message Sends

Suppose one node sends a message to another node

\[
\text{send (data, length, destination)}
\]

There are two possible behaviours

- **Blocking send**

  The send does not return until the data to be sent has “left” the application. This usually means that the message has been copied by the message passing system, or it has been delivered to the destination process. On return from the `send` routine the `data` buffer can be reused without corrupting the message.

- **Non-blocking send**

  Upon return from the `send` routine the `data` buffer is volatile. This means that the data to be sent is not guaranteed to have left the application, and if the `data` buffer is changed the message may be corrupted. The idea here is for the `send` routine to return as quickly as possible so the sending process can get on with other useful work. A subsequent call is used to check for completion of the send.

Semantics of Message Receives

Suppose one node receives a message from another node

\[
\text{receive (data, max_length, source)}
\]

There are two possible behaviours

- **Blocking receive**

  The receive does not return until the data to be sent has “entered” the application. This means that the message has been copied into the `data` buffer and can be used by the application on the receiving process.

- **Non-blocking receive**

  Upon return from the `receive` routine the status of the `data` buffer is undetermined. This means that it is not guaranteed that the message has yet been received into the `data` buffer. We say that a receive has been `posted` for the message. The idea here is for the `receive` routine to return as quickly as possible so the receiving process can get on with other useful work. A subsequent call is used to check for completion of the receive.

MPI Communication Modes

- In MPI the *mode* of a point-to-point communication operation governs when a send operation is initiated, or when it completes.

- **Standard mode:**
  - A send may be initiated even if a matching receive has not been initiated.

- **Ready mode:**
  - A send may be initiated only if a matching receive has been initiated.

- **Synchronous mode:**
  - The same as standard mode, except the send will not complete until message delivery is guaranteed.

- **Buffered mode:**
  - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this.

Message Passing Protocols

Suppose one node sends a message and another receives it.

\[
\text{SOURCE: send (data, length, destination)}
\]

\[
\text{DEST: receive (data, max_length, source)}
\]

Two important message passing protocols are

- **Synchronous send protocol**

  The send and receive routines overlap in time. The send does not return until the receive has started. This is also known as a *rendezvous* protocol.

- **Asynchronous send protocol**

  The send and receive routines do not necessarily overlap in time. The send can return regardless of whether the receive has been initiated.
**MPI Blocking Send**

- Consider the standard blocking send routine in C:
  ```c
  int MPI_Send (  
    void  *start_of_buffer,  
    int   number_of_items,  
    MPI_Datatype datatype,  
    int   destination_rank,  
    int   tag,  
    MPI_Comm communicator)
  ```

**Return Status Objects**

- The return status object may be used after completion of a receive to find the actual length, source, and tag of a message.
- Return status object is MPI-defined type.
- In C `status` is a structure:
  ```c
  status
  
  status.source gives source process
  status.tag gives the message tag
  
  Number of elements in message is given by
  ```
  ```c
  MPI_Get_count (status, datatype, count)
  ```

**MPI Blocking Send**

- `start_of_buffer`: Address of the start of the data to be sent.
- `number_of_items`: Number of items of the specified datatype to be sent.
- `datatype`: The datatype of the items to be sent. User-defined datatypes can be used, but mostly the standard MPI-supplied datatypes are used, such as `MPI_INT` and `MPI_FLOAT`.
- `destination_rank`: The process number of the destination process.
- `tag`: The message tag. This is can be used to distinguish between different types of message.
- `communicator`: The communicator defines which processes may be involved in the communication. In most elementary applications the MPI-supplied communicator `MPI_COMM_WORLD` is used.

**MPI Blocking Receive**

- Consider the standard blocking receive routine in C:
  ```c
  int MPI_Recv (  
    void  *start_of_buffer,  
    int   number_of_items,  
    MPI_Datatype datatype,  
    int   source_rank,  
    int   tag,  
    MPI_Comm communicator,  
    MPI_Status status)
  ```

- In a receive routine, `source_rank` and `tag` can have the values `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.
- The tag and communicator values specified by the receiver must match those of the sender for a communication to take place.
Point-to-point Communication

- MPI provides for point-to-point communication between pairs of processes.
- Message selectivity is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Rank and tag may be wildcarded.
- The components of a communicator may not be wildcarded.

Process Ranks

When an MPI program is started the number of processes, \( n \), is supplied to the program from the invoking environment. The number of processes in use can be determined from within the MPI program with the routine \( \text{MPI Comm size} \).

\[
\text{int MPI Comm size (MPI Comm comm, int *size)}
\]

In most applications, the communicator is \( \text{MPI Comm WORLD} \).

Each of the \( n \) processes is identified by a unique integer in the range 0 to \( n - 1 \). This is called the process rank. A process can determine its rank with the routine \( \text{MPI Comm rank} \).

\[
\text{int MPI Comm rank (MPI Comm comm, int *rank)}
\]

Communication Completion

- A communication operation is **locally complete** on a process if the process has completed its part in the operation.
- A communication operation is **globally complete** if all processes involved have completed their part in the operation.

A communication operation is globally complete if and only if it is locally complete for all processes.

Example Program 1

In this simple program every process sends its rank to process 0 where it is output.

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

main (argc, argv)

int argc;

char **argv;

{ int rank, n, i, message;
  MPI_Status status;
  MPI_Init (argc, argv);
  MPI_Comm_size (MPI_COMM_WORLD, &n);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if (rank==0) { /* Process 0 will output data */
    printf ("%n Hello from process %d\n", rank);
    for (i=0; i<n; i++) {
      MPI_Send (message, 1, MPI_INT, MPI_COMM_WORLD, 0, MPI_COMM_WORLD);
      printf ("%n Hello from process %d\n", message);
    }
  }
  else
  MPI_Send (rank, 1, MPI_INT, 0, 111, MPI_COMM_WORLD);

  MPI_Finalize ();
}
```
Notes on Program 1

Note the following:

- All MPI calls must come between the calls to `MPI_Init` and `MPI_Finalize`.
- The information from the processes is not necessarily output in ascending order because the program does not specify the order in which process 0 receives messages.
- We could have each process output its rank instead of sending it to process 0.

Compiling and Running

MPI programs may be run on a network of workstations such as those making up the UTK TORC cluster. Just log on to `torc0.cs.utk.edu` and use the following command to compile:

```
mpicc -O -o file file.c -I/usr/local/mpich/include
-L/usr/local/mpich/build/lib/Linux/chp4
-mpich -lm
```

where `file.c` is the C source code, and `file` is the name of the executable. The above should be input all on one line.

To run a program on the network of workstations do the following:

```
mpirun -np n -machinefile machines file
```

where `n` is the number of processes you want to use, `file` is the name of the executable, and `machinefile` is a file you have previously set up containing the names of the machines you want to use, one on each line. For example, to run the program `sum` with 4 processes:

```
mpirun -np 4 -machinefile machines sum
```

Collective Communication

The send and receive style of communication between pairs of processes is known as **point-to-point communication**. This is distinct from **collective communication** in which several processes are involved in a coordinated communication task. Examples include:

- Broadcasting data. One process, known as the **root**, sends the same data to all processes.
- Data reduction. Data from all processes is combined using a **reduction function** to produce a single result. The result may reside on a single process or on all processes.

Broadcast

A common form of broadcast algorithm is based upon a **broadcast tree**. Suppose node 0 is the root of the broadcast. Consider the following tree:

For a hypercube a tree broadcast uses direct links and if hardware can send on all link simultaneously then

- **Step 1.** 0 sends to 1, 2, and 4
- **Step 2.** 1 sends to 3 and 5, 2 sends to 6
- **Step 3.** 3 sends to 7

If hardware can send on only one link at a time

- **Step 1.** 0 sends to 1
- **Step 2.** 0 sends to 2, 1 send to 3
- **Step 3.** 0 sends to 4, 1 sends to 5, 2 sends to 6, 3 sends to 7
Reduction

Reduction can also be represented by a tree algorithm. For example, if we want to sum numbers on all nodes to one node see Fig. 1.

![Figure 1: Reduction to node 0.](image1)

If we want to perform the sum so that all nodes end up with the result see Fig. 2.

![Figure 2: Reduction to all nodes.](image2)

Collective Routines

- Other forms of reduction include finding the maximum or minimum of a set of numbers over all processes.
- These reduction and broadcast algorithms are logarithmic in number of nodes, i.e., time to run is approximately proportional to log n.
- On hypercubes the logarithmic algorithms involve communication only between neighbouring processes.
- Other algorithms may be better for other network topologies.
- MPI provides routines for broadcasting and reduction.

Application Topologies

- In many applications, processes are arranged with a particular topology, e.g., a regular grid.
- MPI supports general application topologies by a graph in which communicating processes are connected by an arc.
- MPI also provides explicit support for Cartesian grid topologies.
  
  MPI_Cart_create (comm old, ndims, *dims, *period, reorder, *comm_cart)

- Periodicity in each grid direction may be specified.
- Inquiry routines transform between rank in group and location in topology
- For Cartesian topologies, row-major ordering is used for processes.

Topological Inquiry Routines

- mpi_topo_test returns the type of topology associated with a communicator.
- Can find number of dimensions in a Cartesian topology:
  
  MPI_Cartdim_get (comm, *ndims)

- More information on a Cartesian topology can be obtained with:
  
  MPI_Cart_get (comm, maxdims, *dims, *periods, *coords)

- Mapping of coordinate position in Cartesian topology to rank:
  
  MPI_Cart_rank (comm, *coords, *rank)

- Mapping of rank to coordinate position:
  
  MPI_Cart_coords (comm, rank, maxdims, *coords)
Uses of Topologies

- Knowledge of application topology can be used to efficiently assign processes to processors.
- Cartesian grids can be divided into hyperplanes by removing specified dimensions.
- MPI provides support for shifting data along a specified dimension of a Cartesian grid.
- MPI provides support for performing collective communication operations along a specified grid direction.

Topologies and Data Shifts

- Consider the following two types of shift for group of size \( N \):
  - Circular shift by \( J \). Data in process \( K \) is sent to process \( (K + J) \mod N \).
  - End-off shift by \( J \). Data in process \( K \) is sent to process \( K + J \) if this is between 0 and \( N - 1 \). Otherwise, no data are sent.
- Topological shifts are performed using `MPI_Sendrecv`.
- `MPI_Cart_shift` returns the ranks of the processes that a process must send to and receive from when performing a shift on a topological group.

Send/Receive Operations

- In many applications, processes send to one process while receiving from another.
- Deadlock may arise if care is not taken.
- MPI provides routines for such send/receive operations.
- For distinct send/receive buffers:
  ```
  MPI_Sendrecv
  ```
- For identical send/receive buffers:
  ```
  MPI_Sendrecv_replace
  ```

Vibrating String Problem

We shall now study the vibration of waves on a string, and design a parallel MPI program to solve the partial differential equation that describes the problem mathematically.

Problem

A string of length \( L \) and fixed at each end is initially given a known displacement. What is the displacement at later times?

- Introduce coordinate \( x \) so that one end of the string is at \( x = 0 \) and the other end is at \( x = L \).
- Denote the displacement of the string at position \( x \) and time \( t \) by \( \psi(x, t) \).
- We want to know \( \psi(x, t) \).
The Wave Equation

Mathematically the vibrating string problem is described by the wave equation. The mathematical description of the problem is to solve

\[ \frac{1}{c^2} \frac{\partial^2 \psi}{\partial t^2} - \frac{\partial^2 \psi}{\partial x^2} = 0 \]

subject to the boundary conditions

\[ \psi(0, t) = \psi(L, t) = 0 \]

and the initial condition

\[ \psi(x, 0) = u(x) \]

where \( u(x) \) is some known function describing the initial displacement of the string.

We shall solve this problem numerically by approximating the solution at a number of equally-spaced values of \( x \).

![Displacement](image)

\[ x = 0 \hspace{2cm} x = L \]

Sequential Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX_POINTS 128

int main()
{
    double psi[MAX_POINTS];
    double new_psi[MAX_POINTS];
    double old_psi[MAX_POINTS];
    double psi[1] = 3.141592654;
    double tau = 0.05;
    double x;
    int i, j;
    int n_points = 128;
    int nsteps = 100;
    for(i=0;i<npoints;i++)
    { x = 2.0*psi[i]+(double)(npoints-1);
        x = sin(x);
        psi[i] = old_psi[i] = x;
    }
    for(i=0;i<nsteps;i++)
    { new_psi[i] = 2.0*psi[i] - old_psi[i] +
        tau*tau*(psi[i+1]-2.0*psi[i]+psi[i-1]);
    }
    for(i=0;i<npoints;i++)
    { old_psi[i] = psi[i];
        psi[i] = new_psi[i];
    }
    }
    for(i=0;i<npoints;i++)
    printf("%d %d%g\n", i,(i+1)%npoints, psi[i]);
    exit(0);
}
```

Numerical Approximation

- Let the \( n \) values of \( x \) at which we wish to approximate the solution be denoted by \( x_0, x_1, \ldots, x_{n-1} \), where \( x_i = i\Delta x \) for \( i = 0, 1, \ldots, n-1 \), and \( \Delta x = L/(n-1) \).
- We shall denote the displacement, \( \psi(x, t) \), at position \( x \) and time \( t \) by \( \psi(t) \).
- We shall find the solution at a series of time steps \( t_j = j\Delta t \) for \( j = 0, 1, \ldots, J-1 \).
- Given estimates of the solution at times \( t \) and \( t - \Delta t \) it can be shown that the solution at time \( t + \Delta t \) can be approximated by

\[ \psi(t+\Delta t) = 2\psi(t) - \psi(t-\Delta t) + \Delta t^2 \frac{\partial^2 \psi(t)}{\partial x^2} \]

where \( \Delta t = \Delta x / (\omega c) \).
- Thus, \( \psi \) at a new time step depends on the value at the same position for the previous two time steps, and the values at neighbouring positions for the most recent time step.
- We shall specify the initial displacement to be a sine wave, i.e., \( u(x) = \sin x \).

Data Distribution

- Give each process a block of points on the string.
- Each process should have approximately the same number of points to ensure good load balance.
Communication Requirements

The update formula is
$$\psi_i(t+\Delta t) = 2\psi_i(t) - \psi_i(t-\Delta t) + \tau^2(\psi_{i-1}(t) - 2\psi_i(t) + \psi_{i+1}(t))$$
so to update a point we need to know the value of $\psi$ at the neighbouring points. This entails communication.

0 1 2 3

Each process needs to communicate the $\psi$ values for its first and last points before performing an update.

Array Declarations

Each process needs to store the endpoint values received from the neighbouring processes. These are stored at the 0 and local_npts+1 positions in the psi array.

```
local_npts = 5

0 1 2 3 4 5 6
```

Thus, the psi array needs two “extra” entries, and is declared as
```
double psi[MAX_POINTS+2];
double new_psi[MAX_POINTS+2];
double old_psi[MAX_POINTS+2];
```

Outline of Parallel Code

The main phases of the parallel version of the code are as follows.

- **Initialise data distribution**
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes to left and right

- **Initialise arrays**
  - Determine how many points each process handles and the index of the first point in each.
  - Set the psi and old_psi arrays

- **Perform Update**
  - Communicate end points
  - Do update locally

- **Output results**

Initialising the Data Distribution

```
int period=0, record=0;
int mpierr, mpierr, rank;
MPI_Comm new_comm;

MPI_Comm_size (MPI_COMM_WORLD, &period);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Cart_create (MPI_COMM_WORLD, 1, &period, &period,
                 &rank, &new_comm);
MPI_Cart_coords (new_comm, rank, 1, &mpierr);
MPI_Cart_shift (new_comm, 0, -1, &right, &left);
```

- **MPI_Comm_size** gives us the number of processes in use.
- **MPI_Comm_rank** gives the rank of each process.
- **MPI_Cart_create** sets up a new communicator, new_comm, with a 1D topology associated with it.
- **MPI_Cart_coords** gives the position in the 1D topology of each process.
- **MPI_Cart_shift** gives left and right, the neighbouring processes to the left and right respectively.
Initialisation of Arrays

The arrays are initialised as follows.

\[
\begin{align*}
    \text{local} \_\text{npts} &= n\text{points} / n\text{procs}; \\
    \text{lb} \text{bound} &= \text{my} \text{pos} \* \text{local} \_\text{npts}; \\
    \text{for}(i=0; i<\text{local} \_\text{npts}; i++) &
    \lbrace \\
        x &= 2.0 \pi \text{\*(double)}(\text{lb} \text{bound} + i) \/(\text{double})(n\text{points} - 1); \\
        x &= \sin(x); \\
        \text{ps}i[i+1] &= 0.0 \text{\*ps}i[i] + x;
    \rbrace
\end{align*}
\]

- \text{local} \_\text{npts} is the number of points updated by each process.
- \text{lb} \text{bound} is the index of the first point in each process, i.e., it is the global index corresponding to local index 1.
- We initialise the arrays for indices 1 up to \text{local} \_\text{npts}. Indices 0 and \text{local} \_\text{npts}+1 will be used later to store values received from neighbouring processes.

Communication Code

First all processes send \text{ps}i[1] to the process to the left, and receive data from the process to the right, storing it in \text{ps}i[\text{local} \_\text{npts}+1].

\[
\text{MPI} \_\text{Send} \_\text{recv}(\text{ps}i[1], 1, \text{MPI} \_\text{DOUBLE}, \text{left, 111,}} \\
\text{ps}i[\text{local} \_\text{npts}+1], 1, \text{MPI} \_\text{DOUBLE}, \text{right, 111,}} \\
\text{new} \_\text{comm, &status});
\]

Next all processes send \text{ps}i[\text{local} \_\text{npts}] to the process to the right, and receive data from the process to the left, storing it in \text{ps}i[0].

\[
\text{MPI} \_\text{Send} \_\text{recv}(\text{ps}i[\text{local} \_\text{npts}], 1, \text{MPI} \_\text{DOUBLE}, \text{right, 111,}} \\
\text{ps}i[0], 1, \text{MPI} \_\text{DOUBLE}, \text{left, 111,}} \\
\text{new} \_\text{comm, &status});
\]

Update Phase

The update phase has 3 main parts.

1. Communicate endpoints between neighbours
2. Update points locally
3. Copy arrays ready for next update step

\[
\text{i} \text{start} = (\text{my} \text{pos} = 0) \ ? \ 1: \\
\text{if} \text{end} = (\text{my} \text{pos} = n\text{procs} - 1) \ ? \ \text{local} \_\text{npts} - 1 : \text{local} \_\text{npts};
\text{for}(i=0; i<\text{start}; i++)
\lbrace
    \text{new} \_\text{ps}i[1] &= 2.0 \pi \text{\*(ps}i[1] - \text{old} \_\text{ps}i[1]) + \\
    \text{tan}\text{\*(ps}i[1] - 2.0 \pi \text{\*(ps}i[1]);}
\rbrace
\text{for}(i=\text{end}; i<\text{local} \_\text{npts}; i++)
\lbrace
    \text{old} \_\text{ps}i[1] &= \text{ps}i[1];
    \text{ps}i[1] &= \text{new} \_\text{ps}i[1];
\rbrace
\}
\]

Output

- We would like the format of the output to be independent of the number of processes used, and the same as for the sequential program.
- We can ensure this happens by passing the data to be output to process 0 and outputting it from there.

\[
\text{for}(j=0; j<n\text{procs}; j++)
\lbrace
    \text{if} \text{end} = 0 \lbrace
    \text{MPI} \_\text{Cart} \_\text{rank} \text{\*(new} \_\text{comm, &j, &proc});
    \text{if} \text{end} = 0 \lbrace
    \text{MPI} \_\text{Recv}(\text{ps}i[1], \text{local} \_\text{npts}, \text{MPI} \_\text{DOUBLE}, \text{proc, a} \_\text{tag, new} \_\text{comm, &status});
    \text{for}(i=1; i<\text{local} \_\text{npts}; i++)
    \text{printf}("\%d.6f\n", \text{ps}i[1], (i+1)\%n) \ ? \ \text{\"n' : ' \")};
    \rbrace
    \text{else} \text{\{ \text{MPI} \_\text{Send}(\text{ps}i[1], \text{local} \_\text{npts}, \text{MPI} \_\text{DOUBLE}, 0, \text{a} \_\text{tag, new} \_\text{comm});}
    \rbrace
\rbrace
\]
**Performance Analysis**

To analyse the performance of the parallel wave equation code we just look at the update phase.

- To update each point requires 6 floating-point operations in the parallel and sequential codes.
- In the parallel code each process sends and receives two floating-point numbers in each update step.
- Ignoring the time to copy to the arrays old\_psi and \psi, the speed-up is

\[
S(N) = \frac{6n_{\text{tcalc}}}{6n/N + 2f_{\text{comm}}} 
\]

\[
= \frac{1}{1 + \tau / (3g)}
\]

where \( N \) is the number of process, \( n \) is the number of points, \( g = n/N \) is the grain size, and \( \tau = t_{\text{comm}} / t_{\text{tcalc}} \).

- The efficiency is

\[
\epsilon(N) = \frac{1}{1 + \tau / (3g)}
\]

and the overhead is \( f = \tau / (3g) \).

- Since the efficiency depends on \( g \) but not independently on \( N \) the parallel algorithm is perfectly scalable.

---

**Laplace Equation Problem**

- The next problem we shall look at may be used to determine the electric field around a conducting object held at a fixed electrical potential inside a box also at a fixed electrical potential.
- As with the vibrating string problem, this problem can also be expressed mathematically as a partial differential equation known as the Laplace equation.
- We shall design a parallel MPI program to solve the partial differential equation.

**Potential = 1**

**Potential = 0**

---

**Laplace Equation**

Mathematically the problem of determining the electric field in the box is described by the two-dimensional Laplace equation,

\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0
\]

subject to the boundary conditions

\[
\phi(0, y) = \phi(L_1, y) = 0
\]

\[
\phi(x, 0) = \phi(x, L_2) = 0
\]

and \( \phi(x, y) = 1 \) on \( S(x, y) \), where \( S(x, y) \) gives the shape of the object inside the box.

---

**Numerical Approximation**

- We solve the problem numerically by approximating the solution at equally-spaced points on the rectangle.
- For a square problem \( L_1 = L_2 = L \) we find the approximate solution at \( n^2 \) points lying in a regular \( n \times n \) grid. The \( x \) values are denoted by \( x_0, x_1, \ldots, x_{n-1} \), and the \( y \) values by \( y_0, y_1, \ldots, y_{n-1} \).
- The solution is found using an algorithm called \textit{Jacobi iteration}. Starting with an initial approximation we refine the approximation in a series of steps as follows

\[
\phi_{i,j}^k = \frac{1}{4}(\phi_{i-1,j}^{k-1} + \phi_{i+1,j}^{k-1} + \phi_{i,j-1}^{k-1} + \phi_{i,j+1}^{k-1})
\]

where \( \phi(x_i, y_j) \) at iteration step \( k \) is denoted by \( \phi_{i,j}^k \).
- We keep iterating until the solution has converged.
Sequential Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NMAX 32

main()
{
    double phi[NMAX][NMAX];
    double oldphi[NMAX][NMAX];
    int nptxs = 32, nptys = 32;
    int nsteps = 20;

    setup_grid(phi, nptxs, nptys, mask);

    for(k=0; k<nsteps; k++)
    {
        for(j=0; j<nptys; j++)
            for(i=0; i<nptxs; i++)
                oldphi[i][j] = phi[i][j];

        for(i=0; i<nptxs; i++)
            if (mask[i][j])
                phi[i][j] = 0.25*(oldphi[i][j-1] + oldphi[i][j+1] + oldphi[i-1][j] + oldphi[i+1][j]);

        for(j=0; j<nptys; j++)
            for(i=0; i<nptxs; i++)
                printf("%.3f \t", phi[i][j], ((k%2)==0) ? "," : "");

        exit(0);
    }
}
```

Initial Values of Arrays

The routine `setup_grid` initialises the arrays `phi` and `mask` as follows:

(a) \( \phi \)

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) \( \text{mask} \)

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Initialising the Arrays

```c
define phis (phi, nptxs, nptys, mask)

double phi [NMAX][NMAX];

int nptxs, nptys;

mask [NMAX][NMAX];

int i, j;

for(j=0; j<nptys; j++)
    for(i=0; i<nptxs; i++)
    {
        phi[i][j] = 0.0;
        mask[i][j] = 1;
    }

for(i=0; i<nptxs; i++)
    for(j=0; j<nptys; j++)
        mask[i][j] = 0;
```

Data Distribution

- Give each process a 2D block of points.
- Each process should have approximately the same number of points to ensure good load balance.
- Use a MPI's topology routines to map each block of points to a process.
Communication Requirements

The update formula is

$$\phi_{ij}^n = \frac{1}{4} (\phi_{i+1,j}^{n-1} + \phi_{i-1,j}^{n-1} + \phi_{i,j+1}^{n-1} + \phi_{i,j-1}^{n-1})$$

so to update a point we take the average of the 4 neighbouring points from the previous iteration. Points lying along the boundary of a process need data from neighbouring processes.

Each process needs to communicate the points lying along its boundary before performing an update.

Array Declarations

Each process needs to be able to store the boundary values received from its neighbours. These are stored in rows 0 and local_npts+y+1 and in columns 0 and local_npts+x+1 of the phi array.

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Thus, the arrays are declared as

```c
double phi[NMAX+2][NMAX+2];
double oldphi[NMAX+2][NMAX+2];
int mask[NMAX+2][NMAX+2];
```

Outline of Parallel Code

The main phases of the parallel version of the code are as follows.

- **Initialise data distribution**
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes in the left, right, up, and down directions.

- **Initialise arrays**
  - Determine how many points each process handles.
  - Set the phi and mask arrays

- **Perform Update**
  - Copy phi array to oldphi array.
  - Communicate boundary points.
  - Do update locally

- **Output results**

Initialising the Data Distribution

```c
/* Find rank and number of processors */
MPI_Comm_size (MPI_COMM_WORLD, &nproc);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* Work out number of processes in each direction */
dims[0] = dims[1] = 0;
MPI_Dim_get (nproc, 2, &dims);
mp_all = dims[0];
mp_y = size[0];
mp_x = size[1];

/* Set up 2D topology */
periods[0] = periods[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods,
						1, &mpmx, &pmpy);
MPI_Cart_coords (size, nproc, 2, coords);
mp_all = coords[0];
mp_all = coords[1];

/* Determine neighbouring processes for communication */
MPI_Cart_shift (size, 0, -1, &mpnmx, &mpnpx);
MPI_Cart_shift (size, 1, -1, &mpnpx, &mpnpy);
```
Initialising the Data Distribution

- `np` and `np` are the number of processes in the process grid in each direction. We make the grid as square as possible.
- This time we set up a 2D communicator, `new_comm`.
- `MPI_Cart_coords` gives the position in the topology of each process.
- Calls to `MPI_Cart_shift` give the ranks of the neighbouring processes in the four directions.

```
Initialisation of Arrays

local_nptx = nptx/nx;
local_npty = npty/nx;

for(j=0;j<local_nptx;j++)
   for(i=0;i<local_npty;i++)
      phi[i][j] = 0.0;
      mask[i][j] = 1;
   }

if (nptx == 0)
   for(i=0;i<local_nptx;i++)
      mask[i][0] = 0;

if (nptx == nx)
   for(i=0;i<local_nptx;i++)
      mask[local_npty][i] = 0;

if (nptx == 0)
   for(i=0;i<local_nptx;i++)
      mask[0][i] = 0;

if (nptx == nx)
   for(i=0;i<local_nptx;i++)
      mask[local_npty][i] = 0;

for(j=0;j<local_nptx;j++)
   
   if (global_y == npty) global_x = global_x + 1;
   if (global_x == nptx) global_x = 0;
   
   if (global_x == 0 || global_x == nptx - 1 ||
       global_y == 0 || global_y == npty - 1)
      mask[i][j] = 0;
      phi[i][j] = 1.0;
   }

Update Phase

The update phase has three main parts.

1. Copy `phi` to `oldphi` array.
2. Communicate boundary data.
3. Update points locally.

```
**Communication**

- Communication takes place by shifting data in each of the four directions (left, right, up, and down).
- Before communicating in the left and right direction, we must explicitly buffer the data to be sent, and unpack it when it is received.

**Performance Analysis**

- The update formula is
  \[ \phi_{i,j} = \frac{1}{4} (\phi_{i+1,j}^{-1} + \phi_{i-1,j}^{-1} + \phi_{i,j+1}^{-1} + \phi_{i,j-1}^{-1}) \]
  which requires 4 floating-point operations per grid point.
- The number of grid points per processor shifted in the left/right direction is \( n/P \), where \( n \times n \) is the size of the grid and \( P \) is the number of processors in the \( y \) direction of the processor mesh.
- The number of grid points per processor shifted in the up/down direction is \( n/Q \), where \( Q \) is the number of processors in the \( x \) direction of the processor mesh.

**Output**

- As with the vibrating string example, we want the output to be formatted the same in the sequential and parallel programs. We do this by having process 0 do all the output.
- We output the \( \phi \) array one row at a time. The outer two for loops below pick out the rows of \( \phi \).

```c
for (n=0; n<ny; n++)
for (j=1; j=local_nptxy; j++)
  for (m=0; m<mx; m++)
    if (mpnyx == n && m == ny)
      if (rank==0) MPI_Send (phi[j][i], local_nptxy, MPI_DOUBLE, 0, 115, MPI_COMM_WORLD);
      else
        for (i=0; i<local_nptxy; i++, k++)
          printf("%9.6f%c", phi[j][i],
        ((k+1)%8==0) ? \"n\" : \"\n\")
      else if (rank==0){
        coords[0] = m;
        coords[1] = n;
        MPI_Cart_rank (new_comm, coords, &source);
        MPI_Rcv (phi[i][j], local_nptxy, MPI_DOUBLE, source, 115, MPI_COMM_WORLD, &status);
        for (i=0; i<local_nptxy; i++, k++)
          printf("%9.6f%c", phi[i][j],
        ((k+1)%8==0) ? \"n\" : \"\n\")
        }
      }

```

**Performance Analysis**

- Speed-up is
  \[ S(N) = \frac{4n^2l_{\text{sub}}}{(4n^2/N)l_{\text{sub}} + (2n/Q)l_{\text{sub}} + (2n/P)l_{\text{sub}}} \]
  \[ = \frac{1 + (P+Q)/2}{1 + (Q/N)(1+\alpha)(\tau/\sqrt{g})/2} \]
  where \( M = n \times n \) is the size of the grid, \( P \times Q \) is the processor mesh, \( P = \alpha Q \), and \( \tau = t_{\text{sub}}/t_{\text{sub}} \).
- Since \( N = PQ = \alpha Q^2 \) and \( M = n^2 \) the efficiency is
  \[ e(N) = \frac{1}{1 + (1+\alpha)/(2\sqrt{g})/\sqrt{N/M}} \]
  \[ = \frac{1}{1 + (1+\alpha)/(2\sqrt{g})/\sqrt{\tau/\sqrt{g}}} \]
  where \( g = M/N \) is the grain size, and the overhead is \( f = (1+\alpha)/(2\sqrt{g})(\tau/\sqrt{g}) \).
- Since the efficiency depends only on \( g \), and not independently on \( n \) and \( N \), the parallel algorithm is perfectly scalable.
Irregular Communication

- In the wave equation and Laplace equation problems the communication is very regular. Once we set the number of processes and the size of the problem the communication requirements of the algorithm are fully determined.

- We shall now consider a parallel molecular dynamics simulation. In this simulation we know that data may need to be communicated between processes at a particular point in the program, but we do not know which data it will be. In this example the communication is slightly irregular.

Cutoff Distance

- We could find the force on particle $i$ by summing over all the other particles,

$$ F_i = \sum_{j \neq i}^{n} f_{ij} $$

where $f_{ij}$ is the force exerted by particle $j$ on particle $i$. This results in an $O(n^2)$ algorithm.

- We can improve the running time if we make use of the fact that the force $f_{ij}$ is zero for particles more than distance $r_0$ apart.

Linked Lists

- The linked list for a particular cell tells us what particles are in that cell.

- Each cell has its own list. The starts of the lists are stored in a 2D array.

- A particle can find out which cell it is in from its position.

Molecular Dynamics Simulations

- We have $n$ particles in a periodic square domain.

- The particles interact in a known pairwise way. Each particle exerts a force on the other particles so that
  - if the particles are close enough they repel each other
  - if the particles are far enough apart they are attracted to each other
  - if the particles are more than some distance, $r_0$, apart they do not influence each other.

- This sort of interaction is typical of many molecules.

- Given initial positions and velocities for the particles, we follow the movement of the particles at a series of discrete time steps.
Data Structures

- Each particle has a position and velocity, so the particle data can be stored in an array of structures.

```c
struct particle {
    double position[2];
    double velocity[2];
} Particles[MAX_PARTICLES];
```

- We need to be able to access the particles in a particular cell. We could do this by associating a vector with each cell to store the indices of the particles in that cell. But different cells contain different numbers of particles so memory may be wasted.

- Instead we store pointers to the particles in a particular cell in a linked list.

```c
struct list {
    struct particle *part;
    struct list *list;
} *Lists[MAX CELLY][MAX CELLX];
```

- This is called **geometric hashing**.

Outline of Sequential Code

1. Initialise arrays.
   (a) initialise particle positions and velocities
   (b) initialise the 2D array of cell lists by inserting particles into the linked lists

2. Update loop.
   For each time step
   (a) find force on each particle
   (b) move particle under influence of the force exerted on it and update particle positions and velocities

3. Output
   Perform whatever output is required.

The Update Loop

At each time step we do two main things

1. Find the force on each particle
2. Move each particle according to the force on it

The pseudocode for the update loop is as follows.

```c
for (each time step)
    for (each particle, p)
        find out the location (i,j) of cell p is in
        set force[p] = 0
        for (cell (i,j) and the 8 neighbouring cells)
            for (each particle q in cell)
                add force of q on p to force[p]
        end for
    end for
    for (each particle, p)
        update velocity and position of p using force[p]
    end for
end for
```

Data Distribution

- The particles are distributed to processes by assigning a rectangular block of cells to each process.

- We find out the node number, location in the process mesh, and the node numbers of the neighbouring processes as in the Laplace equation problem.
Data Dependencies

Particles in cells along the boundary of a process need
to know about particles in other processes in order
to evaluate the force on them.

Each process needs particle data from 8 neighbouring
processes.

Communication Requirements

- Each particle needs information about the particles
  in the neighbouring cells in order to determine the
  force on it. So we need to communicate particles
  lying in cells along the boundary of each process.
- When particles move they may travel from the set of
cells owned by one process to those of another pro-
cess. This is called particle migration and requires
communication.

Array Declarations

- We maintain a 2D array of linked lists—one for each
cell in a process.
- When we receive particle information from cells lying
  along the boundary of adjacent processes we store the
data at the end of the particle array.
- Pointers to the particles received are placed into cell
  lists.

```c
struct list {
    struct particle *part;
    struct list *list;
} *Lists[MAX_CELLY+2][MAX_CELLX+2];
```

Parallel Update Loop

In the parallel algorithm there are two communication
phases—one for boundary cell data and one for particle
migration.

```c
for (each time step)
    communicate particle data for boundary cells
    for (each particle, p)
        find out the location (i,j) of cell p is in
        set force[p] = 0
        for (cell (i,j) and the neighbouring cells)
            for (each particle q in cell)
                add force of q on p to force[p]
        end for
    end for
    for (each particle, p)
        update velocity and position of p using force[p]
    end for
    migrate particles
end for
```
Communication of Boundary Cell Data

The communication is similar to that for the Laplace equation problem except

1. We have to look in the cell lists to see which particles have to be sent and pack this information into a send buffer.

2. The receiving process does not know beforehand how many particles it is going to receive.

3. We have to communicate between diagonally adjacent processes.

Left and Right Shifts

We need to communicate particles in corner cells to diagonally adjacent processes. This can be done in 4 shift operations.

| B A a a A B | A b b b B A |
| b a a b b a | a b b b a |
| b a a b b a | a b b b a |
| B A a a A B | A b b b B A |

Shift left, then shift right

Up and Down Shifts

To ensure the corner cell data are communicated correctly we shift complete rows of cells indexed from 0 to local_ncellsx.

| D c c c D | C d d d D C |
| B A a a A B | A b b b B A |
| b a a b b a | a b b b a |
| b a a b b a | a b b b a |
| B A a a A B | A b b b B A |
| D c c c D | C d d d D C |

| B A a a A B | A b b b B A |
| D c c c D | C d d d D C |
| d c c d c | d c d d c |
| d c c d c | d c d d c |
| D c c c D | C d d d D C |
| B A a a A B | A b b b B A |

Shift up, then down

Pseudocode for Left Shift

nsend = 0
for (i=1 to local_ncellsy)
    for (each particle, p, in cell (i,1))
        pack position of p into sbuf
        nsend = nsend + 2
    end for
end for
MPI_Sendrecv (sbuf, nsend, MPI_DOUBLE, left, a_tag, rbuf, max_size, MPI_DOUBLE, right, a_tag, new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &nrecv);
for (i=1 to nrecv)
    take next 2 numbers from rbuf, store in x and y
    set position of particle npart+i-1 to (x,y)
    find out which cell (x,y) is in
    add particle npart+i-1 to list for that cell
end for
Pseudocode for Up Shift

```plaintext
nsend = 0
for (i=0 to local_ncellsx+1)
    for (each particle, p, in cell (1,i))
        pack position of p into sbuf
        nsend = nsend + 2
    end for
end for
MPI_Sendrecv (sbuf, nsend, MPI_DOUBLE, up, a_tag,
            rbuf, max_size, MPI_DOUBLE, down, a_tag,
            new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &recv);
for (i=1 to recv)
    take next 2 numbers from rbuf, store in x and y
    set position of particle npart+i-1 to (x,y)
    find out which cell (x,y) is in
    add particle npart+i-1 to list for that cell
end for
```

Particle Migration

We have to be able to handle the case where particles move to diagonally adjacent processes. The pseudocode to update particle positions and velocities and to migrate them if needed to new processes is as follows.

```plaintext
for (each particle, p)
    update position and velocity
    determine which cell p is in
    if (p has moved to new cell)
        delete p from list of old cell
        if (p has moved to different process)
            put p into appropriate communication buffer
            remove p from particle array
        else
            add p to list of new cell
        end if
    end if
end for
shift left
shift right
shift up
shift down
```

Shifting in Particle Migration

After receiving particle data from another process, a process must determine if the particle belongs to it, or if it has to be passed on to another process.

```plaintext
MPI_Sendrecv (leftbuf, nsend_left, MPI_DOUBLE, left, a_tag,
              rbuf, max_size, MPI_DOUBLE, right, a_tag,
              new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &recv);
for (i=1 to recv)
    get next 4 numbers from rbuf, store in x, y, vx, vy
    if (particle belongs in this process)
        add particle to end of particle array
        find out what cell particle is in
        add particle to list for that cell
    else
        put particle in appropriate communication buffer
    end if
end for
```

Notes on Parallel Molecular Dynamics Simulations

- As in previous examples we need to exchange "boundary" data between processes.
- Need to communicate with diagonally adjacent processes. This can be done with four shift operations.
- We do not know beforehand how many particles will be communicated between processes in the boundary data exchange or migration steps. The receiving process determines this from the number of bytes received.
- In general each process holds a different number of particles and this changes over time. However, we don't expect load imbalance to be too bad because particles tend to be evenly distributed in space.
**WaTor — A Dynamical System**

- We shall now look at a very dynamic simulation called WaTor. WaTor is a periodic 2D dimensional ocean (hence, *Watery Tomus*) in which sharks and fish compete and survive.
- The parallel implementation of WaTor has a number of interesting features.
  1. A very inhomogeneous and dynamic load distribution
  2. The need for irregular communication
  3. The possibility of conflicts between updates performed by different processors.
- Other advanced parallel applications share some of these features.

**The Rules of WaTor**

- WaTor takes place on a periodic grid. Each grid cell either contains a fish, a shark, or is empty.
- The grid is initially populated by a specified number of fish and sharks. The populations then evolve in a series of discrete time steps according to certain rules.

**Fish Rules**

- **Moving**
  In each time step, each fish notes which of the 4 neighbouring sites are empty. One of these empty sites is chosen at random and the fish moves there, eating the fish. If there are no empty sites containing fish the shark notes which of the 4 neighbouring sites are empty and moves to one of these sites at random. If all 4 neighbouring sites are already occupied by sharks, the shark stays where it is.

- **Breeding**
  If a fish is past the fish breeding age, then when it moves it breeds, leaving a fish of age zero at its previous location. A fish cannot breed if it doesn’t move.

- **Eating**
  Fish eat plankton available throughout the ocean. Fish never starve.

**Rules for Sharks**

- **Moving**
  In each time step, each shark notes which of the 4 neighbouring sites are occupied by fish. One of these sites is chosen at random and the shark moves there, eating the fish. If there are no neighbouring sites containing fish the shark notes which of the 4 neighbouring sites are empty and moves to one of these sites at random. If all 4 neighbouring sites are already occupied by sharks, the shark stays where it is.

- **Breeding**
  If a shark is past the shark breeding age, then when it moves it breeds, leaving a shark of age zero at its previous location. A shark cannot breed if it doesn’t move.

- **Eating**
  Sharks eat only fish. If a shark does not eat for more than a certain number of time steps (known as the shark starvation age) then it dies.

**Input for WaTor**

The input to the simulation is

1. The size of the grid
2. The initial number of sharks and fish (these are placed at random)
3. The shark and fish breeding ages
4. The shark starvation age
Data Structures

The two fundamental data structures are the 2D grid and a list of sharks and fish.

The ocean data structure is:

```c
struct ocean {
    int   occupier_type;    // Fish, shark, or empty
    struct swimmer *occupier;  // pointer to shark or fish
} Ocean[MAXY][MAXX];
```

Fish and sharks use the same data structure

```c
struct swimmer {
    int    type;            // Fish or shark
    int    x, y;            // Position in ocean
    int    age;
    int    last_age;
    int    iteration;       // Time step number
    swimmer *prev;          // Pointer to previous
    swimmer *next;          // Pointer to next
} *list;
```

Outline of Sequential Code

- **Initialise**
  Initialize `ocean` array by placing fish and sharks at random grid locations.
  Initialise fish/shark list

- **Update**
  In each time step, we process the fish and sharks in the order in which they appear in the list.
  We may also update the display, or perform some other output, in each time step.

- **Finalise**
  Output final state, statistics, etc.

Data Distribution

- Initially we shall use a 2D block data distribution, just the same as in the Laplace equation problem and the molecular dynamics simulation.
- Each processor looks after a block of the ocean and all the fish and sharks in it.
- We need to know the node number, location in the processor mesh, and the node numbers of the neighbouring processors.

Data Dependencies

Fish and sharks lying along the boundary of a processor need to know about the grid cells lying along the boundary of other processors in order to follow the WaTor rules.

Each processor needs data from 8 neighbouring processors.
**Array Declarations**

- When we receive fish and shark data from cells lying along the boundary of adjacent processors we insert the data into the linked list and point to it from the ocean array.

```c
struct ocean {
    int occupier_type;
    struct swimmer *occupier;
} Ocean[MAXX+2][MAXX+2];
```

<table>
<thead>
<tr>
<th>local_nptsx = 5,</th>
<th>local_nptsy = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 0.1 0.2 0.3 0.4 0.5 0.6</td>
<td></td>
</tr>
<tr>
<td>1.0 1.1 1.2 1.3 1.4 1.5 1.6</td>
<td></td>
</tr>
<tr>
<td>2.0 2.1 2.2 2.3 2.4 2.5 2.6</td>
<td></td>
</tr>
<tr>
<td>3.0 3.1 3.2 3.3 3.4 3.5 3.6</td>
<td></td>
</tr>
<tr>
<td>4.0 4.1 4.2 4.3 4.4 4.5 4.6</td>
<td></td>
</tr>
<tr>
<td>5.0 5.1 5.2 5.3 5.4 5.5 5.6</td>
<td></td>
</tr>
<tr>
<td>6.0 6.1 6.2 6.3 6.4 6.5 6.6</td>
<td></td>
</tr>
</tbody>
</table>

2D ocean grid array

---

**Potential for Conflict**

- In the course of an update a fish or shark may move into a “border” grid cell. Thus, unlike in the Laplace and molecular dynamics problems, the border cells in a processor may change in an update.
- These changes must be communicated back to the processor that originally send that border strip, to be re-integrated into its data structures.
- However, the processor owning those grid cells may also have updated them, and these updates may be in conflict.

---

**Rollback**

One way to resolve conflicts is called **rollback**, and works as follows.

1. Return the fish or shark that crossed the processor boundary back to its original processor, and place it back in its original position.
2. If that position has been occupied by another fish or shark, then that fish or shark must be rolled back.
3. This rollback process continues until until every fish and shark has a place to go.

---

**Sub-Partitioning**

An alternative solution to rollback in resolving conflicts is to avoid them in the first place. This is can be done as follows,

1. Divide each processor’s grid into 4 smaller sub-grids, labelled 1, 2, 3, and 4.
2. Exchange the parts of sub-grids 2 and 4 that have data along their boundaries needed to update sub-grid 1 in adjacent processors.
3. Update sub-grid 1 in each processor.
4. Return boundary information to original owner and update data structures.
5. Repeat steps 2, 3, and 4 for each of the other sub-grids in turn.
Outline of Parallel Code

The update loop of the parallel version of WaTor using sub-grids is as follows.

for (each time step)
    for (each sub-grid, GI, I=1,2,3,4)
        exchange boundary data along 2 edges of GI
        store data received in border of ocean and in fish/shark list
        update fish and sharks in GI
        exchange boundary data, overwriting original data with updated data
    end for
end for

- Sub-grid 1 communicates with the left and up processors
- Sub-grid 2 communicates with the right and up processors
- Sub-grid 3 communicates with the left and down processors
- Sub-grid 4 communicates with the right and down processors

Load Imbalance in WaTor

- Load balance is an important consideration in WaTor and many other applications.
- In WaTor the workload is generally not evenly distributed over the ocean, so a block data distribution means that some processors have less work than others at certain times.
- Load balance in WaTor changes with time as the fish and sharks move.

Dynamic Load Imbalance

In dealing with dynamic load imbalance the following two approaches are important:

1. Use of a dynamic load balancer so that the distribution of the ocean among the processors changes as the fish and shark system evolves. When dealing with grids some form of recursive bisection is often used.

2. Use of a cyclic, or scattered, data distribution. The parts of the grid assigned to one processor do not form a contiguous block but are scattered in a regular way over the whole domain. The aim in this case is to achieve statistical load balance.

Orthogonal Recursive Bisection

- Orthogonal Recursive Bisection (ORB) first divides the domain orthogonal to the x-direction so there are equal numbers of items in each of the two subdomains.
- Then each of these 2 subdomains is independently divided orthogonal to the y-direction, to give 4 subdomains each with approximately the same number of items in each.
- This process of bisection continues, alternating between the x and y directions, until there is one subdomain for each processor.

ORB is not used when the items are distributed uniformly over the domain – in this case the subdomains would come out about the same size and shape.
Orthogonal Recursive Bisection

- If the items are distributed unevenly over the domain, ORB can give rise to a variety of different shaped processor subdomains.
- Using a dynamic load balance scheme such as ORB adds to the complexity of the software, particularly in deciding which boundary data must be communicated with which processors.

![Orthogonal Recursive Bisection Diagram]

Hierarchical Recursive Bisection

- HRB is a variation of ORB in which we first make all the cuts in one direction, and then all the cuts in the second direction, rather than alternating directions.
- HRB allows the data distribution to be adjusted over just one direction, rather than both.
- ORB and HRB can easily be extended to 3 or more dimensions.

![Hierarchical Recursive Bisection Diagram]

Cyclic Data Distributions

- In a cyclic data distribution the data assigned to each processor is scattered in a regular way over the domain of the problem.
- The figure below shows how a grid might be cyclicly distributed over a $4 \times 4$ mesh of processors.
- The cyclic distribution is a simple way to improve load balance but can result in more communication as it increases the amount of boundary data in a processor.

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.0</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.0</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
<td>2.3</td>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
<td>2.3</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
<td>3.3</td>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
<td>3.3</td>
<td>3.0</td>
<td>3.1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
<td>2.3</td>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
<td>2.3</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
<td>3.3</td>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
<td>3.3</td>
<td>3.0</td>
<td>3.1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>1.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Cyclic Data Distributions

Consider a one-dimensional cyclic data distribution of an array, such as,

$\text{array} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$

- This is known as a cyclic[1] data distribution, and can be regarded as mapping a global index, $m$, to a processor location, $p$, and a local index, $i$.

$$m \mapsto (p, i)$$

where $p$ and $i$ are given by

$$p = m \mod N$$

$$i = \lfloor m/N \rfloor$$

and $N$ is the number of processors.

- The inverse mapping is

$$m = i \times N + p$$

- If we arrange array entries in groups of size $k$ and cyclicly distribute these we get a cyclic[k] data distribution. For example, the following shows a cyclic[2] data distribution.

![Cyclic Data Distribution Diagram]
Block Data Distributions

- For a one-dimensional block data distribution the mapping of global index, \( m \), to a processor location, \( p \), and a local index, \( i \), is

\[
p = \lfloor m/T \rfloor \\
i = m \mod T
\]

where \( T = \lfloor M/N \rfloor \), \( M \) is the number of items in the array, and \( N \) is the number of processors.

- The inverse mapping is

\[
m = p \times T + i
\]

Block Data Distributions

- Multi-dimensional arrays are distributed by applying the desired data distribution separately to each array index. Thus, for a two-dimensional data distribution the global index \( (m, n) \) is mapped so that \( m \mapsto (p, i) \) and \( n \mapsto (q, j) \), where \( (p, q) \) is location on a \( P \times Q \) processor mesh and \( (i, j) \) is the index into the local 2D array.

- Different data distributions can be applied over each array dimension.

- For a 2D \( \text{cyclic}[\mathbf{1}], \text{cyclic}[\mathbf{1}] \) data distribution of a \( M \times N \) array over a \( P \times Q \) processor mesh we would have

\[
m \mapsto (p, i) = (m \mod P, \lfloor m/P \rfloor) \\
n \mapsto (q, j) = (n \mod Q, \lfloor n/Q \rfloor)
\]

<table>
<thead>
<tr>
<th>0,0</th>
<th>1,0</th>
<th>2,0</th>
<th>3,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
<td>0,0</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
</tr>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
</tr>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
</tbody>
</table>