Introduction to Parallel Computing Concepts

ParaPhrase International Summer School in Parallel Patterns

Peter Kilpatrick
Queen’s University Belfast

http://www.cs.qub.ac.uk/~p.kilpatrick/ParaPhrase.pptx
Flynn's Classical Taxonomy

Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of *Instruction* and *Data*. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*.

The matrix below defines the 4 possible classifications according to Flynn:

<table>
<thead>
<tr>
<th>S I S D</th>
<th>S I M D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction, Single Data</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>M I S D</td>
<td>M I M D</td>
</tr>
<tr>
<td>Multiple Instruction, Single Data</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
</tbody>
</table>
Single Instruction, Multiple Data (SIMD)

Single instruction: All processing units execute the same instruction at any given clock cycle

Multiple data: Each processing unit can operate on a different data element

Examples:
- Processor Arrays: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV
- Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs), employ SIMD instructions and execution units.

Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.

Synchronous (lockstep) and deterministic Execution. Two varieties: Processor Arrays and Vector Pipelines
SIMD - branch instruction

Figure 2.4  Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.
Multiple Instruction, Multiple Data (MIMD)

Currently, the most common type of parallel computer. Most modern computers fall into this category.

Multiple Instruction: every processor may be executing a different instruction stream

Multiple Data: every processor may be working with a different data stream

Execution can be synchronous or asynchronous, deterministic or non-deterministic

Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.

Note: many MIMD architectures also include SIMD execution sub-components
Limits and Costs of Parallel Programming

Amdahl's Law states that potential program speedup is defined by the fraction of code \( P \) that can be parallelized:

\[
\text{speedup} = \frac{1}{(P/N) + S}
\]

where \( P \) = parallel fraction, \( N \) = number of processors and \( S \) = serial fraction.

It soon becomes obvious that there are limits to the scalability of parallelism. For example:

<table>
<thead>
<tr>
<th>( N )</th>
<th>( P = .50 )</th>
<th>( P = .90 )</th>
<th>( P = .99 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.82</td>
<td>5.26</td>
<td>9.17</td>
</tr>
<tr>
<td>100</td>
<td>1.98</td>
<td>9.17</td>
<td>50.25</td>
</tr>
<tr>
<td>1000</td>
<td>1.99</td>
<td>9.91</td>
<td>90.99</td>
</tr>
<tr>
<td>10000</td>
<td>1.99</td>
<td>9.91</td>
<td>99.02</td>
</tr>
<tr>
<td>100000</td>
<td>1.99</td>
<td>9.99</td>
<td>99.90</td>
</tr>
</tbody>
</table>

![Graph showing speedup vs. number of processors](Graph.png)
Limits and Costs of Parallel Programming

However, certain problems demonstrate increased performance by increasing the problem size. For example:

<table>
<thead>
<tr>
<th>Description</th>
<th>Time 1</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Grid Calculations</td>
<td>85 seconds</td>
<td>85%</td>
</tr>
<tr>
<td>Serial fraction</td>
<td>15 seconds</td>
<td>15%</td>
</tr>
</tbody>
</table>

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

<table>
<thead>
<tr>
<th>Description</th>
<th>Time 2</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Grid Calculations</td>
<td>680 seconds</td>
<td>97.84%</td>
</tr>
<tr>
<td>Serial fraction</td>
<td>15 seconds</td>
<td>2.16%</td>
</tr>
</tbody>
</table>

Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.
Granularity

Computation / Communication Ratio

• In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

• Periods of computation are typically separated from periods of communication by synchronization events.
Granularity

**Fine-grain Parallelism**
- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

**Coarse-grain Parallelism**
- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Harder to load balance efficiently
Race Conditions

When two or more threads access a shared variable in uncontrolled manner there is the possibility of a race condition.

Various means are used to avoid race conditions:

- critical sections
- locks
- monitors.
Parallel Computer Memory Architectures
Shared Memory

**General Characteristics:**

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

Multiple processors can operate independently but share the same memory resources.

Changes in a memory location effected by one processor are visible to all other processors.

Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.
Shared Memory - UMA

Uniform Memory Access (UMA):

Most commonly represented today by Symmetric Multiprocessor (SMP) machines

Identical processors

Equal access and access times to memory
Shared Memory - NUMA

Non-Uniform Memory Access (NUMA):

Often made by physically linking two or more SMPs

One SMP can directly access memory of another SMP

Not all processors have equal access time to all memories

Memory access across link is slower
Shared Memory

**Advantages:**

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

**Disadvantages:**

- Primary disadvantage is the lack of scalability between memory and CPUs.
  - Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and
  - for cache coherent systems, geometrically increase traffic associated with cache/memory management.

- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.
Distributed Memory

**General Characteristics:**

Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
Distributed Memory

**General Characteristics (continued):**

Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.

When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
Distributed Memory

**Advantages:**
- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

**Disadvantages:**
- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
It is usually easier to parallelize a program on a shared memory system. However, most systems are distributed memory because of the cost advantages.

To gain both advantages people have investigated *virtual shared memory*, or *partitioned global address space (PGAS)*, using software to simulate shared memory access.

Current projects include Unified Parallel C (UPC), Co-Array Fortran and Titanium.

Asynchronous PGAS languages include: Chapel and X10
Limitations of Memory System Performance

Memory system, and not processor speed, is often the bottleneck for many applications.

Memory system performance is largely captured by two parameters, latency and bandwidth.

Latency is the time from the issue of a memory request to the time the data is available at the processor.

Bandwidth is the rate at which data can be pumped to the processor by the memory system.
Bandwidth and Latency

It is very important to understand the difference between latency and bandwidth.

Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.

Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.

If you want immediate response from the hydrant, it is important to reduce latency.

If you want to fight big fires, you want high bandwidth.
Memory Latency: An Example

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches).

Assume that the processor has two multiply-add units and is capable of executing four floating-point instructions in each cycle of 1 ns. The following observations hold:

The peak processor rating is **4 GFLOPS**.

Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.
Memory Latency: An Example

On the above architecture, consider the problem of computing a dot-product of two vectors.

For each pair of words, the dot-product performs one multiply-add i.e. two FPOs.

It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating!
Caches are small and fast memory elements between the processor and DRAM.

This memory acts as a low-latency high-bandwidth storage.

If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.

The fraction of data references satisfied by the cache is called the cache hit ratio of the computation on the system.

Cache hit ratio achieved by a code on a memory system often determines its performance.
### Improving Effective Memory Latency Using Caches

<table>
<thead>
<tr>
<th>Speed</th>
<th>Cache</th>
<th>nanosec</th>
<th>MByte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ram</td>
<td>100 nanosec</td>
<td>GByte</td>
</tr>
<tr>
<td></td>
<td>Disk</td>
<td>10 millisec</td>
<td>100 GByte</td>
</tr>
<tr>
<td></td>
<td>Tape</td>
<td>minute</td>
<td>100 TByte</td>
</tr>
</tbody>
</table>

Data is moved between levels in blocks (cache lines, pages). Effective high-performance computing includes arranging data and program so that entire block is used while resident in the faster memory.
Improving Effective Memory Latency Using Caches

Multi-core or Multi-processor Cache-based Chip

Typical Layout:
- Each processor:
  - L1 caches
  - Registers
  - Functional units
- Each chip (shared)
  - L2 cache
  - L3 cache
  - Path to memory

On a multi-core chip, get more computational power with (often) same bandwidth to memory, so need to be effective with cache reuse

Note: Different access for L3 cache

L1 = level 1
L2 = level 2, etc.
Impact of Caches: Example

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions $32 \times 32$. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C.

The following observations can be made about the problem:

• Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.

• Multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.

• The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., $200 + 16$ µs.

• This corresponds to a peak computation rate of $64K/216$ or 303 MFLOPS.
OpenMP

https://computing.llnl.gov/tutorials/openMP/
Exercise 1: Solution
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```c
#include "omp.h"
void main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d) ", ID);
        printf("world(%d) \n", ID);
    }
}
```

Sample Output:
- hello(1) hello(0) world(1)
- world(0)
- hello(3) hello(2) world(3)
- world(2)

OpenMP include file
Parallel region with default number of threads
End of the Parallel region
Runtime library function to return a thread ID.
C / C++ - General Code Structure

```c
#include <omp.h>
main () {
    int var1, var2, var3;
    Serial code

    Beginning of parallel section. Fork a team of threads.
    Specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        Parallel section executed by all threads

        All threads join master thread and disband
    }
    Resume serial code

}
Specifying Concurrent Tasks in OpenMP

**FOR**K - shares iterations of a loop across the team. Represents a type of "data parallelism".

**SECTIONS** - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".
The sections Directive

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```
The for Directive

```c
#pragma omp parallel . . .

#pragma omp for
for (i = 0; i < . . . ; i++) {
}
```
Assigning Iterations to Threads: Example

/* static scheduling of matrix multiplication loops */

#pragma omp parallel default(none) private(i, j, k) \
    shared (a, b, c, dim) num_threads(4)

#pragma omp for schedule(static)
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        for (k = 0; k < dim; k++) {
            c(i,j) += a(i, k) * b(k, j);
        }
    }
}
OpenMP Programming: Example

```c
/* ****************************************************
An OpenMP version of a threaded program to compute PI.
******************************************************/

#pragma omp parallel default(none) shared(npoints) \
private(i, rand_no_x, rand_no_y) reduction(+: sum) \
num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;

    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y = (double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) + 
              (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
    }
}
Specifying Concurrent Tasks in OpenMP: Example

```c
#pragma omp parallel default(none) \ 
private(i, rand_no_x, rand_no_y) reduction(+: sum) \ 
num_threads(8)
{
  sum = 0;
  #pragma omp for
  for (i = 0; i < npoints; i++) {
    rand_no_x = (double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y = (double)(rand_r(&seed))/(double)((2<<14)-1);
    if ((rand_no_x - 0.5) * (rand_no_x - 0.5) +
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
      sum ++;
  }
}```
Reduction Clause in OpenMP

- The reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.

- The usage of the reduction clause is `reduction (operator: variable list)`.  

- The variables in the list are implicitly specified as being private to threads.

- The operator can be one of `+`, `*`, `−`, `&`, `|`, `^`, `&&`, and `||`.

```c
#pragma omp parallel reduction(+: sum) num_threads(8) {
    /* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```
The critical Directive

```cpp
#pragma omp critical (name) {
}
```
MPI

https://computing.llnl.gov/tutorials/mpi/
MPI: the Message Passing Interface

MPI include file

Declarations, prototypes, etc.

Program Begins

Serial code

Initialize MPI environment

Parallel code begins

Do work and make message passing calls

Terminate MPI Environment

Parallel code ends

Serial code

Program Ends

General MPI Program Structure
A First MPI Program

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

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

- The message-tag can take values ranging from zero up to the MPI defined 
  constant `MPI_TAG_UB`. 
Sending and Receiving Messages

- Examine and run the program `mpi_ping.c`

- Copy `mpi_ping.c` to `mpi_ping_3.c`. Modify `mpi_ping_3.c` so that it has 3 MPI processes. Process 1 and 2 should each send a unique character to process 0: use a delay to ensure that process 2 sends first. Process 0 should receive from process 1 and then process 2. Make sure that your program blocks accordingly.

- Copy `mpi_ping_3.c` to `mpi_ping_any.c`. Modify `mpi_ping_any.c` to use `MPI_ANY_SOURCE` in process 0 and observe the result.
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, &Stat);
...
```

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, &Stat);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD, &Stat);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
             MPI_COMM_WORLD);
}
...
```
Avoiding Deadlocks

Create a deadlock free program `mpi_ring.c` with 4 processes and where each process, \( i \), receives message from process \( i-1 \) (modulo 4).

For example,

- processor 0 received 3 from processor 3
- processor 2 received 1 from processor 1
- processor 1 received 0 from processor 0
- processor 3 received 2 from processor 2
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)
```
Sending and Receiving Messages Simultaneously

Rewrite the program `mpi_ping.c` using

- `MPI_Ssendrecv`
- `MPI_Ssendrecv_replace`

Task 1: Received 1 char(s), x, from task 0 with tag 1
Task 0: Received 1 char(s), y, from task 1 with tag 1
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.

```c
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.
Overlapping Communication with Computation

Function MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished.

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

- MPI_Wait waits for the operation to complete.

```c
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

Examine and run the illustrative program `mpi_ringtop.c`
Avoiding Deadlocks

Using non-blocking operations removes 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, MPI_COMM_WORLD, &status);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

Rewrite the program `mpi_ping.c` with non-blocking send and receive 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)
  ```
Collective Communication Operations

**MPI_Bcast**

Broadcasts a message to all other processes of that group

```c
count = 1;
source = 1;  // broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msg (before)

|       | 7     | 7     | 7     |

msg (after)
Collective Communication Operations

Write a simple broadcast program with the following output.

```plaintext
before broadcast buffer in processor 0 is 10
before broadcast buffer in processor 1 is 0
before broadcast buffer in processor 7 is 0
before broadcast buffer in processor 2 is 0
before broadcast buffer in processor 3 is 0
before broadcast buffer in processor 6 is 0
before broadcast buffer in processor 4 is 0
before broadcast buffer in processor 5 is 0
after broadcast buffer in processor 2 is 10
after broadcast buffer in processor 7 is 10
after broadcast buffer in processor 4 is 10
after broadcast buffer in processor 1 is 10
after broadcast buffer in processor 3 is 10
after broadcast buffer in processor 6 is 10
after broadcast buffer in processor 5 is 10
after broadcast buffer in processor 0 is 10
```
# 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

**MPI_Reduce**

Perform and associate reduction operation across all tasks in the group and place the result in one task

```c
count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
dest, MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

→ `sendbuf (before)`

| 10     | 10     | 10     | 10     |

→ `recvbuf (after)`
Collective Communication Operations

Write a simple reduce program with the following output.

before reduce, in processor 0, send_buffer is 1, receive_buffer is 0
before reduce, in processor 3, send_buffer is 4, receive_buffer is 0
before reduce, in processor 1, send_buffer is 2, receive_buffer is 0
before reduce, in processor 2, send_buffer is 3, receive_buffer is 0
after reduce, in processor 2, send_buffer is 3, receive_buffer is 0
after reduce, in processor 1, send_buffer is 2, receive_buffer is 0
after reduce, in processor 3, send_buffer is 4, receive_buffer is 0
after reduce, in processor 0, send_buffer is 1, receive_buffer is 10
Collective Communication Operations

**MPI_Scatter**

Sends data from one task to all other tasks in a group

```c
sendcnt = 1;
recvcnt = 1;
src = 1; task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             src, MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sendbuf (before)

| 1 | 2 | 3 | 4 |

recvbuf (after)
Collective Communication Operations

Write a simple scatter program with the following output.

before scatter receive buffer in processor 0 is 0
before scatter receive buffer in processor 1 is 0
before scatter receive buffer in processor 2 is 0
after scatter receive buffer in processor 0 is 1
after scatter receive buffer in processor 1 is 2
after scatter receive buffer in processor 2 is 3
before scatter receive buffer in processor 3 is 0
after scatter receive buffer in processor 3 is 4
Collective Communication Operations

**MPI_Gather**

Gathers together values from a group of processes

```c
sendcnt = 1;
recvcnt = 1;
src = 1;

messages will be gathered in task 1

MPI_Gather(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

`sendbuf (before)`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`recvbuf (after)`
Collective Communication Operations

Write a simple gather program with the following output.

before gather receive buffer in processor 0 is 0, 0, 0, 0
before gather receive buffer in processor 2 is 0, 0, 0, 0
before gather receive buffer in processor 1 is 0, 0, 0, 0
after gather receive buffer in processor 2 is 0, 0, 0, 0
after gather receive buffer in processor 1 is 0, 0, 0, 0
before gather receive buffer in processor 3 is 0, 0, 0, 0
after gather receive buffer in processor 0 is 1, 2, 3, 4
after gather receive buffer in processor 3 is 0, 0, 0, 0
Collective Communication Operations

```c
int MPI_Scatterv ( void *sendbuf,
    int *sendcounts,
    int *displs,
    MPI_Datatype senddatatype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvdatatype,
    int source,
    MPI_Comm comm )
```

The source process sends sendcounts[i] elements to process i.

The array displs specifies where in sendbuf the elements are sent from.

If sendbuf is of the same type as senddatatype, the data sent to process i start at location displs[i] of sendbuf.

Thus, overlapping regions of sendbuf can be sent.
int hostMatch()
{
    int i, j, k, lastI;
    i=0; j=0; k=0;
    lastI = textLength-patternLength;

    while (i<=lastI && j<patternLength)
    {
        if (textData[k] == patternData[j]) {
            k++; j++;
        }
        else {
            i++; k=i; j=0;
        }
    }
    if (j == patternLength)
        return i;
    else
        return -1;
}