A Crash Course on

Parallel Programming

For Novice, Dummies and Dumbheads

Authored by

Prabha Shankar

Under the supervision of

Mr. Noboby,

After the enlightenment that Ignorance is definitely not blissful.

Abstract

We will get here later. And we sure will!!!!!

Contents

List of Figures

Chapter 1

The Fundamentals of Parallel Programming

1.1 What is Parallel Programming?

In this chapter...

- Introduction to Parallel Programming
- Parallel System Performance metrics
- Methodologies of parallelization
- Classification of Parallel Systems

But you would have guessed it already, right? Anyways.....

According to Wikipedia, ***Parallel Programming*** *is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.*

1.2 Performance metrics

A) Speedup: How fast did the work become when we used p processors instead of 1 processor.

$$\text{Speedup (with p processors)} = \frac{\text{performance(p processors)}}{\text{performance(1 processor)}}$$

From a scientific computing point of view,

$$\text{performance} = \frac{\text{work}}{\text{time}}$$

Thus, the above equation becomes,

$$\text{Speedup (with p processors)} = \frac{\text{time (1 processor)}}{\text{time (p processors)}}$$

B) Speedup based on throughput:

Keywords:

(a) **Latency:** In layman terms, latency is the time taken for a message to traverse a system. A low latency indicates high network efficiency.

(b) **Throughput:** Throughput can be thought of as the amount of data or material that can traverse the system. It can be thought of as the amount that a computer can do in a given period of time. Depends on: the speed of the CPU, the amount of available memory, the performance of the operating system, the kind of transmission media, and so on. This is measured in terms of **transactions per minute** or simply **tpm**.

$$\text{Speedup (with p processors)} = \frac{\text{tpm (p processors)}}{\text{tpm (1 processor)}}$$

C) Parallel Efficiency:

$$\text{Efficiency (with p processors)} = \frac{\text{speedup (p processor)}}{\text{p}}$$

1.3 Amdahl's Law

It is a formula which gives the **maximum theoretical speedup** in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

Assumption:

- a fixed amount of workload when comparing parallel and non-parallel systems
- parallel regions have full speedup

Parameters:

- f = Fraction of the parallelable part of the work
- p = Number of parallel threads/tasks/processes

Considering the above, the execution time of a system that allows parallelization is,

$$T(p) = (1-f) * T + \frac{f * T}{p}$$

and corresponding **maximal speedup**,

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1-f) * T + \frac{f*T}{p}} = \frac{1}{1-f+\frac{f}{p}}$$

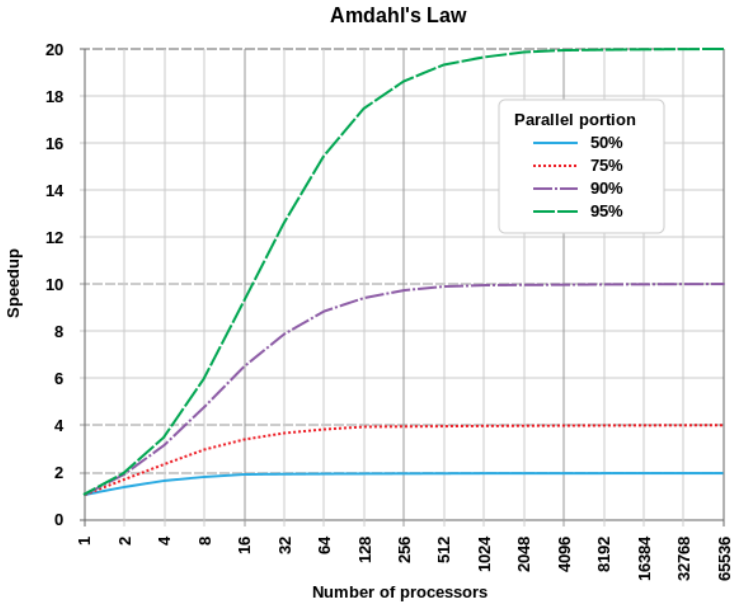

Figure 1.1: *Evolution of speedup with available processors*

Figure(1.1) shows a non-linear scaling of speedup with addition of every new processor and further addition causes no performance improvement.

Consequences of the law:

- small portions of sequential work impacts scalability
- in most practical (non-ideal cases), perfect scaling within a parallel code is hard due to
  - Communication and Synchronization
  - Resource bottlenecks (Memory bandwidth, Hyperthreading, etc.)
- Load imbalance

1.4 Common methodologies for parallelism

A) Multiple Concurrent Execution or SPMD:

- Multiple Concurrent Executions (CEs)
- Distributed data
- Communication between Parallel Executions (PEs)

Advantages: limited orchestration overhead, explicit mapping of problems
Disadvantages: need to explicitly split data, possibility of load imbalance

B) Master/Worker:

Central work queue at the master and workers ask for tasks from the master.

Advantages: simple approach, easy load balancing
Disadvantages: extra orchestration overhead, memory requirements (at masters), more communication, scalability challenges

C) Pipelining:

- Split functionality among PEs
- Pass "task" once functionility is done

Advantages: specialized units of task in parallel execution
Disadvantages: more communication and limited parallelism

Data Parallelism v/s Functional Parallelism

(a) **Data Parallelism (e.g., SPMD):**

- Same operations executed parallely for elements of a large data, say, arrays
- Tasks are the operations on each individual elements or on subsets of the elements
- Length of the task whether same or different depends on the application

(b) **Functional Parallelism (e.g. Pipelining):**

- Entirely different calculations can be performed concurrently on either the same or different data
- Tasks are usually specified via different functions or code regions
- The degree of available functional parallelism is usually modest
- Tasks are of different length in most cases

1.5 Flynn's Taxonomy/Classification

(a) **SISD:** Single Instruction, Single Data (Sequential Processing)

(b) **SIMD:** Single Instruction, Multiple Data (Pipelines, Vectors, GPUs);
Synchronized execution of the same instruction on a set of data;
Vector Programming like CUDA, OpenGL, and so on

(c) **MISD:** Multiple Instruction, Single Data (???/Systolic Arrays).

(d) **MIMD:** Multiple Instruction, Multiple Data (MPP Systems Clusters).
Asynchronous execution of different instructions

SIMD and MISD are the relevant items covering parallel systems and shall be focused on hereon.[††]

Classification of Parallel System can be seen in the figure below:

[††]An important thing to note here is the difference between a core and a processor. **Processor** is the entire chipset including all the cores. **Cores** are like 2 (or more like 4 core, 6 core) parts of the processor that does parallel processing (processing two different data simultaneously in different units) without causing much strain on the processor.

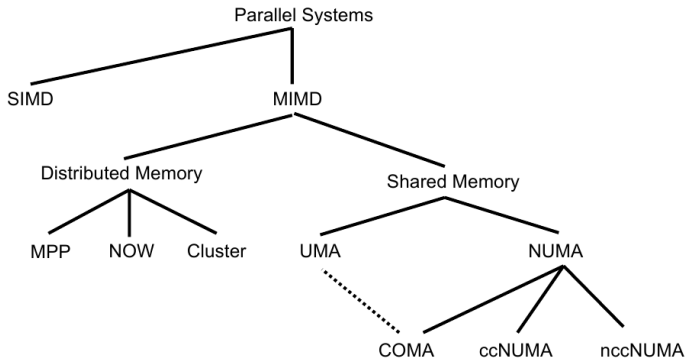

Figure 1.2: *Classification tree for Parallel Systems*

Overview on SIMD and MIMD:

- **SIMD** is based on vector programming in form of pragmas (many vectorizing compilers)
  - one instruction operates on a many data streams
  - GPGPU (General Purpose Computing on Graphics Processing Units) fits this model

whereas

- **MIMD** has two different programming models:
  (a) **Shared Memory (SM) Programming** matching shared memory architecture
    - Multiprocessor System
    - System provides a shared address space
    - Communication is based on read/write operation to global addresses
  (b) **Distributed Memory (DM) Programming** matching distributed memory architecture
    - Multicomputer System
    - Building blocks are nodes with private physical space
    - Communication is based on messages

1.5.1 Shared Memory

The Shared Memory Systems have the following architectures:

(A) **Uniform Memory Access (UMA):** Symmetric Multiprocessors SMP architecture

- Centralized shared memory
- Access to global memory from all processors have "same latency"
- Transition from bus to crossbars
- Memory contention high[§]: as more CPUs are added, competition for access to the bus leads to a decline in performance.

(B) Non-uniform Memory Access (NUMA): Distributed Shared Memory Systems - HW-DSM[*]

- Memory Distributed among the nodes
- Local accesses much faster than the remote ones
- Memory contention low: A processor's own internal computation can be done in its local memory, causing a reduction in memory contention

(C) More exotic:

- **COMA (Cache only):** Cache[†] only memory architecture (COMA) is a computer memory organization for use in multiprocessors in which the local memories (typically DRAM) at each node are used as cache. This is in contrast to using the local memories as actual main memory, as in NUMA organizations.
- **ncc-COMA (Non-cache coherent)**[‡]

[§]**Memory or CPU contention:** It is an event wherein individual CPU components and machines in a virtualized hardware system wait too long for their turn at processing.

[*]**HW-DSM:** Hardware DSM

[†]A **CPU cache** (or simply **cache**) is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations.

[‡]**Cache coherence:** A cache can be used to improve the performance of accessing a given resource. When there are several such caches for the same resource, as shown in the picture, this can lead to problems. **Cache coherence** or **Cache coherency** refers to a number of ways to make sure all the caches of the resource have the same data, and that the data in the caches makes sense (called data integrity). Cache coherence is a special case of memory coherence.

1.5.2 Shared Memory Models match Shared Memory

Mechanism features of shared memory with the details are as follows:

(A) Assumes a global address space with random access

- any read/write can reach any memory cell
- this is true for NUMA as well except that locality becomes tricky
- most models assume cache coherency

(B) Communication through memory access

- load/store operations to arbitrary address
- pass data from PE to the next

(C) Synchronization constructs to coordinate accesses

- need to ensure consistency, i.e., data synchronization
- need to ensure control flow, i.e., control synchronization

Examples: POSIX threads, OpenMP, …

1.6 Distributed Memory

1.6.1 Distributed Memory Models match Distributed Memory

Mechanism features of distributed memory with the details are as follows:

(A) Assumes no global address space

- independent nodes with their own memory connected via network
- no direct visibility of data

(B) All the communications via messages

- Explicit Send/Recv pairs

- Remote memory access via get/put is also an option
- Messages carry data and synchronization

Examples: MPI, PVM

1.7 Hybrid Programming

Most programming structures are hybrid due to their versatility.

1.7.1 Rationale for Hybrid Programming to appear in programming models:

- pure shared memory models do not scale beyond node: memory contention
- pure message passing models create on-node performance issues
  - longer latency than necessary
  - too many message endpoints

1.7.2 Mapping: Critical Step in Hybrid Programming implementation

(a) Which task to execute where?

- What to express in shared memory and what in message passing?
- What to map to threads and processes?
- Where to locate processes and threads (statically or dynamically)?

(b) Some key issues to consider

- Increase data locality
- Minimizing communications
- Resource contention (similar to memory contention)
- Memory usage

Chapter 2

Threading

In this chapter...

- Threading concepts
- Threading APIs
- POSIX Threads

2.1 Threads - Basics

A **Thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

- A thread is a component of the process.
- Multiple thread can exist within a process **executing** concurrently and **sharing** the resources like memory, while different process do not share these resources.

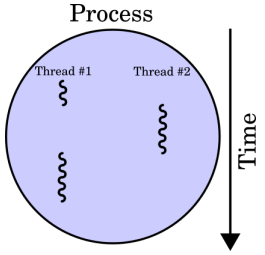

Figure 2.1: *Diagramatic representation of Processes and corresponding threads*

Hardware threads and Software threads

Hardware Threads: It is the execution of the execution stream inside the hardware (similar von Neumann machine in hardware). A Separate Control Unit executing a sequence of instructions.

Software Threads: These are programming abstractions that represent a stream of execution and remain exposed to programmer.

To execute a program. This happens with the software and harware coming together.
A programmer defines a software thread which then gets mapped to hardware thread for execution.

2.1.1 Hyperthreading/SMT - Simultaneous multithreading

Hyperthreading is intel's proprietary simultaneous multithreading to improve the parallelization of computation. For each physical core on the processor, the Operating system addresses to virtual(logical) cores and shares work load between them when possible. Hyperthreads also show up as "CPUs" and not distingushable by default. BIOS initializes them along with cores and sockets as boot which require a reboot for changes.

Functions and Salient Features:

- The main function of hyper-threading is to increase the number of independent instructions in the pipeline
- It takes advantage of the superscalar architecture, in which multiple instructions work on separate data in parallel
- Concurrent workloads: With HTT, one physical processor core appears as two to the operating system, allowing concurrent scheduling of two processes per core
- Resource Sharing: In addition two two or more processes can use the same resources: if resources of one process are not available, then another process can continue if its resources are available
- Mostly useful for systems daemons and I/O operations

Problems with Hyperthreading use in parallel programming.

- multiple instances of same program with same resource requirements
- the above often causing heavy impacts on speedup
- a careful need to understand what needs to be scheduled on HT/SMT

2.1.2 Software Threading Basics

- Traditional view:
  - Operating systems maintain processes which get scheduled to available hardware threads (aka. processors). Each process has one execution stream.
    (**Stream processing** is about processing continuous streams of data by programs in a workflow.
    Continuous execution is discretized by grouping input stream tuples into batches and using one batch at a time for the execution of programs.)
- Processes maintain isolation for protection by separating address space and files. They are:
  - coupled with user IDs
  - communication only via IPC (Inter-Process Communication)
- Threading was intended to make things easier by:
  - Sharing of data without protection boundaries with cooperative behavior to support asynchronous behaviour
    (**Synchronous**, or **Synchronized** means "connected", or "dependent" in some way. In other words, two synchronous tasks must be aware of one another, and one task must execute in some way that is dependent on the other, such as wait to start until the other task has completed.
    Asynchronous means they are totally independent and neither one must consider the other in any way, either in initiation or in execution.)
  - OS still responsible for rescheduling (preemption and progress)

2.1.3 User-Level Threads

User-Level threads, unlike Hyperthreads that require kernel-level modifications, are managed entirely by the run-time system (user-level library).
Features :

- The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes
- User-level threads are small and fast, each thread is represented by a PC, register, stack, and small thread control block
- Switch between threads is easy and can be done as needed
- Use cases:
  - Light weight, fine-grained task based parallel programming
  - Closely coordinated activities with well-defined switchpoints

Disadvantages. There is no guarantee for scheduling as preemption and progress is very hard to guarantee (if not impossible).

Process and its components:

A **process** is an instance of a program that is being executed by one or many threads. It contains the program code and its activity. Depending on the operating system, a process may have multiple threads of execution that execute instructions concurrently.

Component of a process. A process in general is said to own the following resources:

- an **image** of the executable machine code associated with a program
- **Memory** that include the following:
  - executable code
  - process-specific data (input and output)
  - a **call stack** to keep track of active subroutines and/or other events
  - a **heap** to hold intermediate computation data generated during runtime

- Operating system **descriptors** of resources allocated to the process for the execution of the instruction such as file descriptors (UNIX terminology) or handles (Windows)
- Security attributes, such as process owners and the process' set of permissions (permissible operations)
- **Processor state** such as the content of registers and physical memory addressing

NOTE. Though a process has all the components similar to an OS, it doesn't execute on its own.

2.1.4 Rationale for Threads in Parallel Programming

The **traditional use** of threads is **concurrency**, however, they can as well be used for parallel programming. Threads expose hardware threads to programmer and **acts as a native interface to access parallelism** in the architecture.

Thus, the following motivations to talk about threads in Parallel Programming:

- Motivation 1:
  - exploit hardware threads as directly as possible
  - direct mapping to resources
  - understanding bottlenecks and overheads where they occur
- Motivation 2:
  - runtime systems for parallel programming build on them, example, OpenMP
  - properties of threads and how their usage have a large impact on performance

Programming with Threads. Threads are mostly programmed as packages that are **mapped to native APIs** (like POSIX threads, Win32 threads, Solaris threads, and Java threads) and are often **user-level threads**.

Motivation for custom APIs. Lower overheads, customized for particular tasks, and custom hardware with special properties.

2.2 POSIX Threading

2.2.1 Fork/Join Model

In parallel computing, the **fork-join model** is a way of setting up and executing parallel programs, such that execution "branches-off" (fork) in parallel at designated points in the program, to "merges" (join) at a subsequent point and resume sequential execution.
All the threads operate concurrently and location is transparent at first. At the end of the parallel regions, master waits for other thread to complete and continues the sequential execution.

Syntax:

- **Header:**

```
#include <pthread.h>
```

- **Keyword identifier:**

```
pthread_t
```

- **Creating a thread (Fork):**

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    (void*) *kernel (void*),
                    void* args );
```

- **Termination of parallel threads (Join):**

```
int pthread_join(   pthread_t thread,
                    void* (*retval) );
```

2.3 Synchronization between threads

Unless pure concurrency, i.e., independent tasks and no dependencies on allocation computing resources, the threads need to be synchronized.
Common examples to show the need for synchronization are: enforcing common completion of task, enforce happens before initiation into the parallel regions and guard updates to common control unit assigned to the process.
Two main concepts of Synchronization in POSIX threads.

- Locks/Mutual Exclusion (mutex),
- Condition Variables

Concept A: Mutual Exclusion

Mutual Exclusion, also called **mutex**, is used for problems of **concurrent access to shared resources**.These can be:

- shared variables, memory locations
- access to I/O
- two or more threads concurrently updating a variable leading to inconsistencies

API Syntax:

- **Variable Identifier/keyword:**

  ```
  pthread_mutex_t
  ```

- **Initialization:**

  ```
  PTHREAD_MUTEX_INITIALIZER
  ```

  and written as

  ```
  pthread_mutex_t mutexname = PTHREAD_MUTEX_INITIALIZER;
  ```

  or dynamically as:

```
int pthread_mutex_init( pthread_mutex_t &mutexname,
                        const pthread_mutexattr_t *attr );
int pthread_mutex_destroy( pthread_mutex_t &mutexname );
```

Here, for mutex attributes one can do:

```
pthread_mutexattr_t *attr = PTHREAD_MUTEX_RECURSIVE;
```

- **Lock a mutex:** blocks until mutex is granted

  ```
  int pthread_mutex_lock( &mutexname );
  ```

- **Unlock a mutex:** returns immediately

  ```
  int pthread_mutex_unlock( &mutexname );
  ```

- **Lock a mutex, if available:** returns immediately

  ```
  int pthread_mutex_trylock( &mutexname );
  ```

`mutex` is a standalone variable that is not explicitly associated with the memory it protects. It is thus the programmmer's responsibility for its correct usage.
Critierion for implementations of locks. Correctness: gaurantees mutual exclusion, every process gets the mutex, fairness.

Concept B: `<atomic>` Operations

In general from a C++ standpoint, `<atomic>` types are types that encapsulate a value whose access is gauranteed to not cause data races[†] and can be used to synchronize the memory access to shared resources from different concurrent threads.
Common examples in threading are: test and set; compare and swap

```
volatile int *mutex;
void lock()
{ while(test_and_set(&mutex) == 1); }
```

[†]A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operation must be done in a proper sequence to be done correctly. This leads to an inconsistent output

Concept C: Condition Variables

This is another way of prevention of undesirable behaviour as a result when two or more threads try to access and/or manipulation the input at a given memory location. The key features of condition variables are:

- Waiting is based on the shared variable value and it works in conjugation with a mutex variable to protect that data
- A thread needs to wait till a condition is true and therefore, blocked until this happens
- A different threads keeps checking the conditions and once met signals the other threads about this

API Syntax:

- **Variable Identifier/keyword:**

  ```
  pthread_cond_t
  ```

- **Initialization:**

  ```
  PTHREAD_COND_INITIALIZER
  ```

  and written as

  ```
  pthread_cond_t condition_variable = PTHREAD_COND_INITIALIZER;
  ```

  or dynamically as:

  ```
  int pthread_cond_init(      pthread_cond_t &condition_variable,
                              const pthread_cond_attr *attr );
  int pthread_cond_destroy(   pthread_cond_t &condition_variable );
  ```

- **Wait on a condition to be set in another thread:**
  - unlocks mutex, blocks until conditions is signaled, acquires lock
  - Mutex must be locked before calling

    ```
    int pthread_cond_wait(  &condition_variable,
    ```

```
                                        &mutex );
```

- **Signal a condition to a waiting thread:**
  - send condition to other thread
  - must be followed by an unlock

```
int pthread_cond_signal(    &mutex );
int pthread_cond_broadcast( &mutex );
```

Concepts D: Miscellaneous

Barriers

It is a point in the program where we want all the threads to reach before continuing with further executions. This is a very useful tool for synchronization.

API Syntax:

- **Variable Identifier/keyword:**

```
pthread_barrier_t
```

- **Initialization:** Done dynamically as:

```
int pthread_barrier_init(       pthread_barrier_t *barrier,
                                const pthread_barrierattr_t *attr,
                                unsigned int count )
int pthread_barrier_destroy(    pthread_barrier_t *barrier )
```

- **Wait for the threads to complete their processing:**

```
int pthread_barrier_wait( *barrier )
```

2.3.1 Performance Aspects for Threading

The primary aspects that impact the perfomance of the threads are:

- **Overheads**: thread creation and destruction can be a very expensive process, so ensure that the parallel regions are large
- **Contention:**
  - Locks are low-overhead when few threads try access them
  - These overheads grow with more threads accessing them more often indicating high resource contention
- **Thread pinning**: also called **cache affinity/processor affinity**, enables the binding and unbinding of a process or a thread to a CPU or a range of CPUs, so that the process or thread will execute only on the designated CPU(s) rather than any CPU.
  Advantage of this is that remnants of a process that was run on a processor may remain in that same processor's state after another process was run on it. This improves the performance by reducing the performance reducing performance-degrading events such as **cache misses** [††]
  In the context of the given topic, for system-level threads the OS does scheduling of the SW thread to the HW thread, therefore, deciding the location of a thread's execution that largely impacts the performance. **Pinning or fixing** a SW thread to a hardware thread improves the performance due to the above stated reasons.
  NOTE: Thread pinning becomes extremely complicated on a NUMA architecture. Also, thread pinning does not solve the persistent load-balancing problem.

2.3.2 Lock Granularity

Lock Granularity deals with the cost of implementing locks depending upon the space and time where space refers to the data structure of the DBMS[‡] for each lock and time refers to the handling

[††] A **cache miss** is a state where the requested data for processing of a component of application is not found in the cache memory causing execution delay as the data has to be fetched from other cache levels or main memory.

[‡] **DBMS:** Database Management System

of the lock requests and release. The cost of lock implementation depends on the size of data items. There are two types of lock granularity:

- **Fine Granularity:** It refers to small item sizes
  - one lock for each data element
  - maximizes concurrency (multiple threads can have lock at a time)
  - require multiple locks and many locking calls at the same time making the implementation complicated
- **Coarse Granularity:** It refers to large item sizes.
  - one single lock for all data
  - limits concurrency
  - easy to implement

A too-fine lock granularity will increase the frequency for the lock requests and lock releases which will add additional instructions.
A too-coarse lock granularity increases the idle state of threads as they wait longer for acquiring the locks. Also, upon acquiring the lock, they would take longer to execute the larger data increasing the waiting time for other threads. This limits concurrency.

2.4 Drawbacks of Pthreads

- **Pthreads represent direct abstraction of system-level parallelism**
  - Direct map to underlying hardware threads (in most cases)
  - Even parallel loops are difficult to think about
- **Association of data and threads is up to the programmer**
  - No program construct to do this
  - Data locality has to be done explicitly

- **Need to manage the data visibility manually** such as variables per thread and variables shared across the threads
- **Simple synchronization primitives**; these are not sufficient to do easy parallel coordination

Chapter 3

OpenMP - Open Multi-Processing and Shared Memory Systems

In this chapter…

- OpenMP Introductions
- Contructs of Worksharing within OpenMP
- Compiler Directive and Internal Control Variables (ICVs)
- Explicit Tasking and performance considerations
- Memory models in shared memory systems
- Dependence Analysis and Loop Transformations
- Challenges in Shared Memory Parallelism

3.1 What is OpenMP? and what it is not?

OpenMP is an Application Program Interface (API) which is used to program multi-threaded, shared memory parallelism (plus accelerators) usually of the type UMA and NUMA.
Primary component of OpenMP API. They are:

- **Compiler Directives (for C/C++ and Fortran)**
- **Runtime Library Routines**

- **Environment Variables**

What it is not intended to be?
OpenMP is **NOT** ...

- intended for distributed memory systems
- necessarily implemented identically by all vendors
- guaranteed to automatically make the most efficient use of shared memory
- required to check for data dependencies, race condition, deadlocks, etc.
- designed to handle parallel I/O

API Syntax:

- A simple example demonstrating the header for OpenMP and the use of the comiler directives for creating a parallel region

```
#include <omp.h>
int main(){
    #pragma omp parallel        //Compiler Directive
    {
        printf("Hello World!\n");
    }
}
```

- for compiling: `gcc -O3 -openmp program.c`

3.1.1 Fork/Join Execution Model

Similar to the model discussed in the Pthreads sections, the fork/join model goes as:

- The execution of an openmp code begins with a single thread called **master thread**
- Upon encountering the parallel region, the master threads creates additional threads
- After completion of the parallel region, the additional threads return to runtime

- The master thread then continues the execution sequentially
- Additionally, the parallel threads are synchronized at the end of every parallel region via an **(implicit) barrier**

Also, the threads in the parallel region can themselves create additional thread all of which follow the same execution model as above. Therefore, OpenMP is not required to spawn/use additional threads in the parallel region.

3.2 OpenMP Implementations

While OpenMP implementations may differ from system to system, the following features remain inherent to all of them:

- OpenMP is a **language extension** usually over C/Fortran
  - Pragmas to the base language can be ignored during the usage
- Due to the above, it **requires a new compiler** which is built inside the existing one
- Additionally, OpenMP **requires runtime library** which schedule the execution to the hardware threads. These libraries are often built over pthreads
  Standard defines some user functions but not the entire environment. Most common examples are Intel's OpenMP/LLVM, and so on

NOTE: Well defined environment variables are also called **"Internal Control Variables" (ICVs)**. Degree of parallelism is controlled by ICV: `OMP_NUM_THREADS`

3.2.1 OpenMP System Stack

a) OpenMP Application

b) Compiler Directives → OpenMP Runtime APIs → Environment Variables (ICVs)

c) OpenMP Runtime Library

d) Operating System (typically with system level threads, mostly POSIX threads)

e) Memory Architecture (UMA or NUMA)

3.3 More on API Syntaxes and Compiler Directives

3.3.1 Sections in a Parallel Region

This should appear within a parallel section. Each section inside the sections block is executed once by one of the threads. When a thread finishes its section it waits for others threads at the ends of the sections block or region.

API Syntaxes:

- **General Syntax:**

```
#pragma omp sections [parameters]
{
    #pragma omp section
        //...task to be executed in the first section
    #pragma omp section
        //...task to be excecuted in the second section and so on.....
}
```

- **Example:**

```
int main(){
    int a[100], b[100];
    #pragma omp parallel
    {
        #pragma omp sections    //"Sections" block line
        {
            #pragma omp section
            for(int n = 0; n < 100; n++)
                a[n] = 100;
```

```
            #pragma omp section
            for(int n = 0; n < 100; n++)
                b[n] = 200;
        }
    }
}
```

To avoid writing the `"Sections" block line`, we can write the above code as:

```
int main(){
    int a[100], b[100];
    #pragma omp parallel sections
    {
        #pragma omp section
        for(int n = 0; n < 100; n++)
            a[n] = 100;

        #pragma omp section
        for(int n = 0; n < 100; n++)
            b[n] = 200;
    }
}
```

3.3.2 Work Sharing in Parallel Region

The important thing in the parallelism of shared memory, as in this case, is an easy construct for division of data in the parallel region. Most important construct being that of loops escpecially `for` loops.

Parallel Loop

- The iterations of the loop are distributed among the parallel threads that concurrently execute the loop
- There is no synchronization at the beginning of the loop
- However, each thread wait at the end of the loop by an implicit barrier for synchronization of the team of threads unless there is used of *nowait*
- Note: the expressions in the for-statement are very restricted (constrained) in order to apply parallel for without inconsistent output

The constraints on the loop in order to obtain a consistent behavior while using this construct are:

- no data dependencies
- can be implemented in any order
- since there is no programming construct available within OpenMP to do this, it is the responsibility of programmer to ensure this

API Syntax(es)

- **Compiler Directive:** The compiler directive shown below must be followed by a `for` loop:

```
#pragma omp for
for .....
```

- **Examples:**

```
int main(){
    int a[100];
    #pragma omp parallel
    {
        #pragma omp for     //compiler directive line for "for"
        for(int n = 0; n < 100; n++)
            a[n] =  1000;
```

```
    }
}
```

Alternatively, the `compiler directive line for "for"` can be done alternatively written with the compiler directive for `parallel` region as shown below:

```
int main(){
    int a[100];
    #pragma omp parallel for
    {
        for(int n = 0; n < 100; n++)
            a[n] = 1000;
    }
}
```

How do Loops get split up?

Iterations for a loop must be split between threads. Loop scheduling defines chunk size and decides how these are to be distributed amongst the threads (or mapped to the threads). OpenMP provides several options under the construct `schedule`. The compiler directive is:
`#pragma omp for schedule(<option name>, <chunk-size>)`

Available Loop Schedules

- `static`
  - static scheduling with fixed size chunk (decided by the ratio of the iterations and the number of threads) distributed in a round robin fashion
  - default size of `chunk-size` is 1
  - threads are allocated the chunks in a sequential fashion, namely, first allocation to thread 1, then thread 2, and so on ...

  - this is used when all the iterations have the same computational cost

- `dynamic`
  - each threads executes a chunk of the specified size and then requests another chunk until no more chunks are left, hence, dynamic allocation of chunks
  - no particular order of allocation of chunk and it changes each time we execute the loop
  - default size of `chunk-size` is 1
  - `dynamic` scheduling is good when there is different computational cost associated with the iterations which mean that the iterations are poorly balanced between each other
  - due to the dynamic scheduling it has a higher overhead in comparison to `static` scheduling

- `guided`
  - similar to `dynamic` scheduling: each thread executes a chunk of iterations and then requests another chunk
  - the **difference** is that at any instant the **size of chunk is proportional to the number of iterations left divided by the number of threads**
  - `chunk-size` decides the **minimum size of a chunk**. However, the chunk which contains the last iterations may have size smaller than `chunk-size`

- `runtime`
  - the decision for scheduling in this case is deferred until the runtime
  - there are different ways of specifying the scheduling type; one option is through the environment variable `OMP_SCHEDULE` and the other option is with the function `omp_set_schedule`

- `auto`
  - this scheduling delegates the decision of scheduling to the compiler and/or runtime system

Example for Loop Scheduling:

```
#define S 25
int main(int argc, char** argv){
    int a[S],b[S],c[S];
    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (int i=0; i<S;i++)
            a[i] = omp_get_thread_num();
        #pragma omp for schedule(dynamic, 4)
        for (int i=0; i<S;i++)
            b[i] = omp_get_thread_num();
        #pragma omp for schedule(guided)
        for (int i=0; i<S;i++)
            c[i] = omp_get_thread_num();
    }
    for (int i=0; i<S;i++)
        printf("Iter %4d: %4d %4d %4d\n",i,a[i],b[i],c[i]);
}
```

```
Iter    0:  0    3    2
Iter    1:  0    3    2
Iter    2:  0    3    2
Iter    3:  0    3    1
Iter    4:  0    1    1
Iter    5:  0    1    0
Iter    6:  0    1    0
Iter    7:  1    1    1
Iter    8:  1    0    1
Iter    9:  1    0    0
Iter   10:  1    0    0
Iter   11:  1    0    2
Iter   12:  1    1    1
Iter   13:  2    1    0
Iter   14:  2    1    2
Iter   15:  2    1    0
Iter   16:  2    3    2
Iter   17:  2    3    1
Iter   18:  2    3    0
Iter   19:  3    3    2
Iter   20:  3    0    0
Iter   21:  3    0    2
Iter   22:  3    0    1
Iter   23:  3    0    0
Iter   24:  3    1    2
```

Figure 3.1: *Output for the above program*

3.3.3 Data Visibility in Parallel

In this part of chapter, we shall see how certain constructs affect the visibility of data across threads in a parallel region.

Concept I: Shared v/s Private Data

Shared Data:

- the clause `shared(list)` declares all the variables within `list` as shared
- this creates **one copy of the variables per thread**
- **example**: a reference `a[5]` to a shared array accesses
- OpenMP puts **no restriction to prevent data races between shared variables**, thus making it the programmer's responsibility to assure prevention of such an event. This might cause to give an **inconsistent behavior after the parallel region**

Private Data:

- with the clause `private(list)`, OpenMP replicates the variables and assigns its local copy to each thread.
- **Each thread gets its own unique memory where it should store the value of the variable while in the parllel region**. Therefore, the variable copy is accessible only by the thread that owns it. After the parallel region ends, the memory is freed and these variable no longer exist
- private variables sometimes have an **unintuitive behavior in parallel region**.
  Following important remarks: Assume a private variable has value before the parallel region. However, **the value of the variable at the beginning of parallel region is undefined**. Additionally, the value of the variable is the same as the initialized value before the parallel region, in case of initialization. In case uninitialized, the value after the parallel region is inconsistent
- **NOTE:** Iterator variables are **private by default**

Scope of certain variable:

- the default for global variables: **shared**
- variables defined within the "dynamic extent" of the parallel regions: **local**. This also includes routines called from within the parallel regions

Some examples of the Private and Shared data:

Example A:

```
int main(){
    int a[100], t, n;
    #pragma omp parallel
    {
        #pragma omp for private(t)
        for(int i = 1; i<n; i++){
            t    = f(i);
            a[i] = t;
        }
    }
}
```

Global Variable: a, t, n
Local varible to threads: i (by default since it is an **iterator variable**)
Variable `t` by default was shared but is now "privatized"

Example B:

```
int main (){
    int iam, nthreads;
    #pragma omp parallel private(iam,nthreads)
    {
        iam = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        printf(‘‘ThradID %d, out of %d threads\n’’, iam, nthreads);
        if (iam == 0)
```

```
            printf(‘‘Here is the Master Thread.\n’’);
            else
            printf(‘‘Here is another thread.\n’’);
        }
    }
```

REMARKS: A better programming practice is to declare `iam, nthreads` within the parallel region as these are being used privately. Doing that alters the compiler directive to:
`#pragma omp parallel`

Example C: Outputs in private and shared options

- example for `shared` case

```
int i=3;
#pragma omp parallel for
for (int j=0; j<4; j++){
    i=i+1;
    printf("-> i=%d\n", i);
}
printf("Final Value of I=%d\n", i);
```

  - Output for **first run**:

```
-> i=4
-> i=7
-> i=6
-> i=5
Final Value of I=7
```

  - Output for **second run**:

```
-> i=4
-> i=5
-> i=4
-> i=4
Final Value of I=5
```

  - Since OpenMP has no preventive contruct for data race, the value of the shared variable `i` is inconsistent after the parallel loop finsihes

- example for `private` case

```
int i=3;
#pragma omp parallel for private(i)
for (int j=0; j<4; j++){
    i=i+1;
    printf("-> i=%d\n", i);
}
printf("Final Value of I=%d\n", i);
```

- Output for the above:

```
-> i=?
-> i=?
-> i=?
-> i=?
Final Value of I=3
```

- the parallel region shows an undefined value for `i`. However, at the end of the parallel region, the value resumes with the one used for initialization before the parallel region

Concept II: First/Last Private Data

This section is a following the concept of `private` data. Thus, we would consider that as the preceding idea as we move through this section.

Last Private:

- Unlike, the `private` data beahavior, if we want the private data to take forward the last value of the parallel region, we would use `lastprivate`
- **Example A:**

```
int i=3;
#pragma omp parallel for lastprivate(i)
for (int j=0; j<4; j++){
```

```
    i=i+1;
    printf("-> i=%d\n", i);
}
printf("Final Value of I=%d\n", i);
```

- Output for above is:

```
-> i=?
-> i=?
-> i=?
-> i=?
Final Value of I=?
```

- **Remarks:** The output is undefined since the value of `i` is undefined at the beginning of the parallel region. Let's see another example

- **Example B:**

```
int i;
int x;
x=44;
#pragma omp parallel for lastprivate(x)
for(i=0;i<=3;i++){
    x=i;
    printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);
```

- Output for above is:

```
Thread number: 0     x: 0
Thread number: 3     x: 3
Thread number: 2     x: 2
```

```
Thread number: 1     x: 1
x is 3
```

- **Remarks:** The output is `3` and not `1`, i.e., **it is the last iteration and not the last operation** that is kept

First Private:

To understand what happens with the use of the keyword `firstprivate`, we make changes to the examples used for `lastprivate` section.

- `firstprivate` specifies that each thread should have its own instance of a variable (like in case of `private`) and that the variable **should be initialized to the value of the variable before the parallel construct**
- **Example C:**

```
int i=3;
#pragma omp parallel for firstprivate(i)
for (int j=0; j<4; j++){
    i=i+1;
    printf("-> i=%d\n", i);
}
printf("Final Value of I=%d\n", i);
```

  - Output for above is:

```
-> i=4
-> i=4
-> i=4
-> i=4
Final Value of I=3
```

- **Example D:**

```
int i;
```

```
int x;
x=44;
#pragma omp parallel for firstprivate(x)
for(i=0;i<=3;i++){
    x=i;
    printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);
```

- Output for above is:

```
Thread number: 0     x: 0
Thread number: 3     x: 3
Thread number: 2     x: 2
Thread number: 1     x: 1
x is 44
```

- **Remarks on both examples' outputs:** As discussed above the outputs in the both case, i.e., `x=44` and `I=3` are the values of the variable to which it was initialized before the parallel construct

First Private and Last Private:

- Combined behavior of `firstprivate` and `lastprivate` is such that each thread has its own instance of a variable initialized to the value of the variable before the parallel construct and the last value, i.e., the one after the parallel region is the one that is carried foward.
  The example below demonstrate the same:

- **Example E:**

```
int i=3;
#pragma omp parallel for firstprivate(i) lastprivate(i)
for (int j=0; j<4; j++){
    i=i+1;
```

```
        printf("-> i=%d\n", i);
    }
    printf("Final Value of I=%d\n", i);
```

- Output for above is:

```
-> i=4
-> i=4
-> i=4
-> i=4
Final Value of I=4
```

Concept III: Miscellaneous

Default(shared):

- `default(shared)` clause sets the data-sharing attribute to all variables in the construct shared
- the `private` variables needs to be explicity specified
- **Example (a):**

```
int a, b, c, n;
...
#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
    // using a, b, c
    //a, b, c and n are all shared
}
```

- **Example (b):**

```
int a, b, c, n;
...
#pragma omp parallel for default(shared) private(a, b)
for (int i = 0; i < n; i++)
{
    //private variables, here a and b mentioned explicitly mentioned
    //c and n are shared variables
}
```

3.3.4 Reductions

In addition to synchronization, there is often a need of tasks like:

- communication of results of the region
- aggregate the data at master thread
- inform next parallel region

These are often carried out using reductions. Primarily, the function of this construct is to perform operations on the result of all threads or aggregate results and make it available to the master thread.

The key issues in the implementation of `reduction` are:

- potentially costly and time sensitive operations
- hard to implement by oneself
- should be a part of the base language

OpenMP Reductions

- OpenMP offers a special clause for reductions:
  `#pragma omp reduction(<operator>:<list>)`

- This clause performs a reduction on the variables that appear in the `list` with the operator `operator` across the values "at the the thread end/last iteration" of each thread
- variables appearing on the list must be scalar
- operators is one the following: `+,-,*,&, ^, |, &&, ||`
- reduction variable can appear only in the statements with the following forms:
  - x = x operator expr
  - x binop= expr
  - x++, ++x, x–, –x
- user defined reductions are an option in the newer versions of OpenMP
- **Example (a):**

```
#pragma omp parallel for reduction(+ : a)
for(int j=0; j<4; j++){
    a = omp_get_thread_num();
}
printf("Final Value of a=%d\n", a);
```

  - Output:

```
OMP_NUM_THREADS=4
Final Value of a=6 //0+1+2+3
OMP_NUM_THREADS=2
Final Value of a=4 /*   1+3 (in most cases, assuming
                        static loop scheduling) */
```

- **Example (b):**

```
#pragma omp parallel for reduction(+ : a)
for(int j=0; j<4; j++){
    a = j;
```

```
}
printf("Final Value of a=%d\n", a);
```

- Output:

```
OMP_NUM_THREADS=4
Final Value of a=246 //29+49+77+99

Thread 0:    0-24 last value is 24
Thread 1:    25-49 last value is 49
Thread 2:    49-74 last value is 74
Thread 3:    75-99 last value is 99
```

- **Example (c):**

```
#pragma omp parallel for reduction(+ : a)
for(int j=0; j<4; j++){
    a += j;
}
printf("Final Value of a=%d\n", a);
```

- Output:

```
Final value of a=4950 //for any number of threads
```

3.3.5 Data Synchronization in Work Sharing

Similar to Pthreads, often data synchronization is required in OpenMP as well. In OpenMP, the synchronization of data takes place through shared variables but needs to be implemented through a separated construct.
OpenMP offers a number of different options for this, namely:

- Barriers
- Master regions
- Single regions
- Critical sections
- Atomic statements
- Ordered constructs
- Runtime routines for locking

Barrier

`barrier` is the key synchronization construct used to synchronize a team of threads. Each threads pauses the execution upon reaching such a barrier and waits until all the other threads have reached the barrier. Some key features of the construct:

- each parallel region has an implicit barrier at the end to synchronize the parallel threads
- this can be switched off by adding `nowait`
- additional barriers can be added when needed using the following: `#pragma omp barrier`
- **Important caution:** `barrier` construct can cause load imbalancing leading to severe performance issue and therefore should be used only when really needed

Master Region

- **Syntax:**

```
#pragma omp master
    //.....block.....
```

- only the master thread executes this block; other threads skip over
- no synchronization at the beginning of the block

- **Possible use case:** Printing to console/screen, file I/O, and Keyboard entries

Single Region

- **Syntax:**

```
#pragma omp single [parameters]
    //.....block.....
```

- only one arbitrary thread amongst the team of threads execute this block; others skip over
- implicit barrier to synchronize the team of threads (unless `nowait` is specified)
- **Possible cases:** initialization of variables or data structures

Critical Section

- **Syntax:**

```
#pragma omp critical [(Name)]
    //.....block.....
```

- **Similar to "Mutual Exclusion" concept of Pthreads**
  - a critical section is a block of code
  - can be executed by only one thread at a time
- all threads wait at the beginning of the block until its available
- all unnamed critical derivatives map to the same name
- **Important Remarks:**
  - Critical section names are global entities to the program, thus in case of name conflict the bahvior of the program is inconsistent
  - For efficient performance of the code, the critical section should be kept short

Atomic Statements

- **Syntax:**

```
#pragma ATOMIC
    //.....expression-stmt.....
```

- the `ATOMIC` directive ensures that a specific memory location is updated atomically
- `expression-stmt` has to be of the following form:
  - x binop=expr
  - x++ or ++x
  - x– or –x

  where, `x` is an lvalue expression with scalar type and the `expr` doesn't reference the object designated by `x`
- this construct is equivalent to using `critical` to protect updated
- useful for updating the values of shared variables
- since it is implemented by native instructions, it avoids locking

Simple Runtime Locks

In addition to compiler directives based options, i.e., `pragma`, OpenMP also offers runtimes locks similar to the ones seen in pthreads.

- the lock is held by only one thread at a time
- a lock is represented by a lock variable type `omp_lock_t`
- the thread that obtained the lock cannot set it again
- **Syntax:**

```
-   void omp_lock_init(&lockvar);          //initialization of lock
-   void omp_lock_destroy(&lockvar);       //destroy a lock
```

```
-   void omp_set_lock(&lockvar);            //set lock
-   void omp_unset_lock(&lockvar);          //free lock
-   int logical = omp_test_lock(&lockvar);  //returns true or false
```

- **Example:**

```
#include <omp.h>
omp_lock_t myLock;
int main(){
    omp_init_lock(&myLock);
    #pragma omp parallel
    {
        TheCookieJar();
    } // End of parallel region
    omp_destroy_lock(&myLock);
}
void TheCookieJar(){
    omp_set_lock(&myLock); // acquire lock
    grabMyFavouriteCookie(); // get cookie
    omp_unset_lock(&myLock); // release lock
}
```

Nestable Locks

- the concept similar to the simple runtime locks
- nestable locks, however, can be set multiple times by a single thread
- each time locking by the thread which has the lock increments the lock counter value by 1
- similarly, each unlocking decrements the lock counter by 1
- if the lock counter is `0` after an unset operation, lock can then be set by a different thread
- nestable locks have a separate set of routines

- **Syntaxes:**
  - `void omp_init_nest_lock(&lockvar);`
  - `void omp_destroy_nest_lock(&lockvar);`
  - `void omp_set_nest_lock(&lockvar);`
  - `void omp_unset_nest_lock(&lockvar);`
  - `int omp_test_nest_lock(&lockvar);`

Ordered Clause

- **Syntax:**

```
#pragma omp for Ordered
{
    for(.....){
        ...
        #pragma omp for ordered
        { ... }
    }
}
```

- construct must be within the dynamic extent of an `omp for` construct with an ordered clause
- ordered constructs are executed strictly in the order similar to which it would have been executed in a sequential execution of the loop

Important routines and environment variables...

Runtime offers additional routines to control behavior:

- Determine the number of threads for parallel regions:
  `void omp_set_num_threads(int count);`
- Query the maximum number of threads for team creation:
  `int omp_get_max_threads(void);`

Important summary for understanding above: the `omp_get_max_threads()` routine an upper bound on the number of threads that could be used to form a new team if a `parallel` construct without a `num_threads` clause were encountered after execution returns from this routine

- Query maximum number of threads in the current team:

  ```
  int omp_get_num_threads(void);
  ```

- Query own thread number (for any arbitrary threads returns the value of its alloted thread number)

  ```
  int omp_get_thread_num(void);
  ```

- Query the number of processors:

  ```
  int omp_get_num_procs(void);
  ```

Environment Variables (or Internal Control Variables, ICVs) are sets of variable constructs programmed within the OpenMP framework whose values can be determined at runtime to the desired value. Thus, it provides an option to defer the assignment of the required variable until the runtime.

- Number of threads in a parallel region:

  `OMP_NUM_THREADS=4`

  Executed as: `OMP_NUM_THREADS = 4 ./example`

- Selecting a scheduling strategy to be applied at runtime:

  `OMP_SCHEDULE="dynamic"` or `OMP_SCHEDULE="guided,4"`

  Remember that the schedule clause in the code takes precedence

- Allow runtime to determine the number of threads:

  `OMP_DYNAMIC=TRUE`

- Allow nesting of parallel regions if supported by the runtime:

  `OMP_NESTED=TRUE`

3.3.6 Drawbacks of Work Sharing

Main concept discussed during the shared memory parallelism of OpenMP is that of **Work Sharing** which is aimed at distributing the work amongst the team of threads. However, **the require-**

ment of worksharing is different in different cases: static distribution is required in case of NUMA architecture and in other cases equal work distributed.
Here are some drawbacks associated with work sharing:

- **Drawback 1:** possible imbalance caused by workload
  - static distribution of equal chunks can cause load imbalance
  - dynamic distribution can help but issue with NUMA architecture and associated overheads due to fine grained scheduling
- **Drawback 2:** possible imbalance caused by the machine
  - machines are no longer homogenous now with differences between nodes, threads, runs, ...
- **Drawback 3:** limited programming flexibility
  - limited (in OpenMP) to for loops or similar coarse grained sections
  - heirarchical parallelism not easy to follow and optimize

3.4 Explicit Tasking (since OpenMP v3.0)

Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel. These are independent pieces of task executable in any order. Like worksharing constructs, **tasks must also be created inside the parallel region.**
Example snippet:

```
#pragma omp parallel
{
    #pragma omp task
        printf("hello from a random thread\n");
}
```

Features of task construct:

- OpenMP runtime schedules the tasks as needed and **maintains one or more task queues**

- Tasks are distributed to the threads
  - **Task scheduling** can be done by any thread in the team
  - Usually they are dispatched to the threads as they become available
  - once the task associated with a thread finishes, the thread becomes free
  -
- The example above showed a general construct of the `task`. In order to only spawn a task once, the `single` construct is used

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        printf("Hello world!\n");

        #pragma omp task
        printf("Hello again!\n");
    }
}
```

  The above ouputs the two statements on the console. However, the sequence of the output is not guaranteed

- **taskwait:** In order to guarantee sequential execution, we use `taskwait`. This waits until the completion of the immediate child tasks. **Child Tasks** are the tasks generated since the beginning of the current task

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
```

```
            printf("Hello world!\n");

            #pragma omp taskwait    \\ensures the sequence

            #pragma omp task
            printf("Hello again!\n");
        }
    }
```

- **taskyield:** OpenMP provides the `taskyield` constructs for rescheduling of a task such that the current task can be suspended in favor of execution of a different task. **It is important to note that rescheduling is allowed and not required**. This forms a part of the **explicit task scheduling point**. The syntax is given by:

```
#pragma omp taskyield
{ ... }
```

- **Implicit task scheduling point:**
  - task creation
  - end of a task
  - taskwait
  - barrier synchronization
- **Tied tasks:**
  - once the task starts it remains on attached to the the thread that scheduled it
  - this is the default behavior and easy to reason about

  Untied tasks:
  - tasks can be rescheduled to a thread different to the one that scheduled it at the first place

- the execution in this case is usually interrupted or suspended
- usually **advantageous** since this ensures more flexibility and better allocation of resources

3.4.1 Task Dependencies (since OpenMP v4.0)

This construct is used to defined the in/out dependencies between tasks. This construct **impacts the scheduling order of the tasks and thereby the sequence of execution of a program. Features:**

- `in`: the variables consumed by a task
- `out`: the variables produced by a task
- `inout`: the variables both consumed and produced by a task, i.e., modified

This is implemented as a clause for the task construct:

```
#pragma omp task depend(<dependency_type>: <variable_name>)
{ ... }
```

Examples:

```
#pragma omp task shared(x, ...) depend(out: x)  //T1
    preprocess_some_data( ... );
#pragma omp task shared(x, ...) depend(in: x)   //T2
    do_something_with_the_data( ... );
#pragma omp task shared(x, ...) depend(inout: x) //T3
    do_something_arbitrary_with_the_data( ... );
```

Important example on implementation of `task` construct in binary tree

```
struct node {
    struct node *left;
    struct node *right;
};
```

```
void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);
}
#pragma omp parallel
#pragma omp single
traverse(root);
```

3.4.2 Performance Considerations for Tasking

Following are the advantages of the tasking:

- implicit load balancing
- simple programming model
- many complications and bookkeeping is pushed to runtime making it an efficient tool

Tasking, however, requires certain considerations to be kept while its usage.These can heavily impact the performance.

Consideration 1: Task Granularity

- fine grained allows better resource utilization; however, it adds to the overheads significantly
- coarse grained reduces overheads, however, causes schedule fragmantation
- therefore, it is important to ensure an optimum level of granularity

Consideration 2: NUMA optimization

- each task can be run anywhere and the programmers cannot influence this

- modern runtimes provides optimizations like NUMA aware scheduling

3.5 Memory Models

A system is **coherent** if,

- program orders for loads and stores are preserved
- all the stores eventually becomes visible
- all processors see the same order for writes

Two mechanisms to ensure coherence:

- Snoop-based protocols:
  - this tends to be faster if enough bandwidth is available, since all transactions are a request/response seen by all the processors
  - the drawback is that it is not scalable; the reason for this is that every request must be broadcast to all nodes of the system, meaning that as size of the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides must grow
- Directory based protocols:
  - the data being shared is placed in a common directory that maintains the coherence between caches
  - the directory as a filter through which the processor must ask the permission to load an entry from the primary memory to its cache
  - when an update is made the directory either updates or invalidates the data of the other caches with that entry

Sequential Consistency

A multiprocessor is said to be **sequentially consistent** if,

- the result of any execution is same as if the operations of the processors are executed in some sequential order, **and**,
- the operation of each individual processor appears in this sequence in the order specified by the program

Why we need relaxed consistency requirements?

- necessary for well performing systems with increase in the level of concurrency
- some amount of sequential consistency can be achieved via synchronization of contents which we need anyway for the synchronization of contents

Relaxed Consistency Models

- **Processor Consistency:**
  - writes by any thread is seen by all the other threads
  - however, different threads may see a different order
- **Weak Consistency:**
  - Data operations v/s synchronization operations
  - synchronization operations are sequentially consistent
  - when a synchronization operation is issued, the memory pipeline is flushed
- **Release Consistency:**
  - further subdivide synchronization operations into “acquire” and “release”; similar to acquiring a lock and releasing a lock
  - before accessing a variable, all acquire operations have to be complete
  - before completing a release, all read and write operations must be complete
  - all acquire and release have be to be sequentially consistent

3.5.1 The OpenMP Memory Model

OpenMP uses a relaxed consistency **similar to that of weak consistency**, which have the following key features:

- temporarily the view of memory on different threads can be inconsistent
- memory state is made consistent only at particular constructs
- programmer need to know about these constructs and synchronize additionally as needed

Memory Synchronization Points (or Flush Points)

- Entry and exit of parallel regions
- barriers (both implicit and explicit)
- entry and exit of critical regions
- use of OpenMP runtime locks
- every task scheduling points

It is also important to know that **entry and exit of worksharing regions** and **entry and exit to master regions** are not memory synchronization points.

3.5.2 Implementing Manual Synchronization

OpenMP's Flush Directive

```
#pragma omp flush [(list)]
```

- Synchronizes data of the executing thread with the main memory, e.g., copies in register or cache
  - it doesn't update the implicit copies at the other threads
- operates on variables in the given list; if no list is specified, all shared variables accessible in the region
- Semantic:

  - loads/stores executed before the flush have to be finished
  - loads/stores following the flush are not allowed to be executed early

- Examples:

Thread 1

```
a = foo();
flag = 1;
```

Thread 2

```
while(flag);
b = a;
```

is modified into:

Thread 1

```
a = foo();



#pragma omp flush
flag = 1;
```

Thread 2

```
while(flag){
    #pragma omp flush
}
#pragma omp flush
b = a;
```

3.6 Dependence and Dependence Graph

Control Dependence:
S1 is said to be dependent on S2 (written as S1 → S2) iff.

- there is a possible execution path from S1 to S2
- not all path lead from S1 to S2

Data Dependence:
S1 is said to be dependent on S2 (written as S1 → S2) iff.

- there is a feasible path from S1 to S2
- the same data is accessed in S1 and S2 with at least one being write

3.6.1 Type of Data Dependencies

We use the following conventions to understand the data dependencies:

$$I(S) = \text{set of memory locations read (input)}$$
$$O(S) = \text{set of memory locations written to (output)}$$

The types of data dependencies are:

- **True/Flow Dependence:** If the output of S1 overlaps with the input of S2

$$O(S1) \cap I(S2) \neq \varnothing \qquad \Rightarrow \qquad (S1 \delta S2)$$

- **Anti Dependence:** If the input of S1 overlaps with the output of S2

$$I(S1) \cap O(S2) \neq \varnothing \qquad \Rightarrow \qquad (S1 \delta^{-1} S2)$$

- **Output Dependence:** If the output of S1 and S2 are written to same location

$$O(S1) \cap O(S2) \neq \varnothing \qquad \Rightarrow \qquad (S1 \delta^{o} S2)$$

3.6.2 Types of Loop Dependencies

Types of loop dependencies:

- **Loop Independent Dependencies:**
  - All dependencies are within an iteration; none across iterations
  - Example:

```
for( i = 0; i < 4; i++ ){
    S1: b[i] = 8;
    S2: a[i] = b[i] + 10;
}
```

  - Iterations can be parallelized

- **Loop Carried Dependencies:**
  - Dependencies across iterations
  - Computation on iteration depends on the data written in the another

```
for (i = 0; i < 4; i++){
    S1: b[i] = 8;
    S2: a[i] = b[i-1] + 10;
}
```

  - Iterations in this case cannot be parallelized directly (or easily)

3.6.3 Concept of Aliasing

In computing, **aliasing** describes a situation in which a data location in memory can be addressed through symbolic names in the program. Thus,

- modifying the data through one name can implicitly modifies the values associated with all aliased names, which may not be expected by the programmer
- aliasing when done inadvertently creates inconsistent output making it particularly difficult to understand, analyze and optimize programs
- therefore, it forms to be an important idea to be taken care while parallelizing codes

3.6.4 Program Order v/s Dependence

The **sequential order** imposed by the program is too restrictive as it limits the way one can write the program and hides the possibility of parallelism. In order to guarantee correctness of a program, one only needs **to ensure that the dependencies don't change while reordering the code.** Code transformations can be essential in making it parallelizable and improve the overall perform of the code. Here are key definitions for **loop transformations**:

- A reordering transformation **preserves a dependence** if it preserves the relative execution order of the source and sink of that dependence

- Consequence: any reordering transformation that preserves every dependence in a program leads to an **equivalent** computation
- A tranformation is said to be **valid** for a program to which it applies if it preserves all dependencies in the program

3.6.5 Loop Terminology

In a nested loops, which is loops within loops,

- Loop's nesting level = numbe of surrounding loops + 1
- **Iteration number** in a normalized (0 to n-1) loop is equal to the value of the iterator
- **Iteration vector** for one iteration at the innermost loop is the tupel that contains all the iteration numbers. Example: $\mathbb{I} = (3, 2, ..., 7)$
- the set of all possible iteration vectors for a statement is an **iteration space**
- Iteration $\mathbb{I}$ **precedes** iteration $\mathbb{J}$, denoted by $\mathbb{I} < \mathbb{J}$, iff

$$\exists k \qquad \mathbb{I}[r] = \mathbb{J}[r] \qquad \forall r \quad 1 \leq r < k \qquad \text{and} \qquad \mathbb{I}[k] < \mathbb{J}[k]$$

- For each statement S in a loop nest with n loops, exists a specific statement instance for each iteration vector $\mathbb{I}$ which we denote by $S(\mathbb{I})$

3.6.6 Loop Dependencies - Definition

There exists a dependence from $S1(\mathbb{I})$ to $S(\mathbb{J})$ in a common nest of loops, iff.

1. $\mathbb{I} < \mathbb{J}$ or $\mathbb{I} = \mathbb{J}$ and there is a path from $S1$ to $S2$ in the body of the loop
2. Statement $S1$ accesses memory location M on iteration I
3. Statement $S2$ accesses memory location M on iteration J
4. At least one of the two statement is a write

We denote this as $(S1(\mathbb{I}) \delta S2(\mathbb{J}))$. Thus, there exists a loop dependence if these condition are fulfilled.

Few more terminologies:

- **Dependence Distance for** $S1(\mathbb{I})\delta S2(\mathbb{J})$: $\mathbb{J}-\mathbb{I}$
- **Dependence Direction:** tupel with "<", ">", "=" for each loop showing sign of distance

3.7 Loop Transformations

Transformation 1: Loop Interchange

- A loop interchange is safe for a perfect loop nest, if **the direction vectors of all dependencies has only "="**
- A loop interchange is safe for a perfect loop nest, if **the direction vectors of all dependencies do not have a ">" as the leftmost non- "=" direction**

Transformation 2: Loop Fission/Loop Distribution

- It is a **transformation of loop carried dependencies to loop independent dependencies**
- Safety of loop distribution:
  - two sets of statements in a loop nest can be distributed into separate loop nests **if there exists no dependence cycle between these groups**
  - dependency of the outermost loop can be ignored
  - the order of the new loops needs to preserve the dependencies among the statement sets
- It must also be considered, that
  - this transformation generates multiple loops, thus decreasing granularity
  - barrier synchronization between the generated loops might be required
- Loop fission increases the chances of parallelism by separating the parallelizable part from the non-parallelizable part

Transformation 3: Loop Fusion

- Loop fusion transformation combines two loops
- this increases the granularity
- Loop fusion reduces parallelism by creating a sequential loop
- Loop fusion can introduce loop carried dependencies

Transformation 4: Loop Alignment

Loop alignment changes a carried dependence into an independent dependence

- **Key Idea:** Shift of computations into other iterations
- Extra iterations and tests are overheads
- The idea of **loop peeling** is often used to get the desired transformation
- In general, we can eliminate all carried dependencies in a loop if
  - loop contains no recurrence (dependence cycle), and distance of each dependence is a constant independent of the loop index

Thus, we see that:

- transformations that **eliminate carried dependencies** are **loop distribution and loop alignment**
- transformations that **improve efficiency** are **loop fusion and loop interchange**

3.8 Dealing with race conditions for correctness

Race Condition

It is an undesirable situation or event that occurs when two or more threads attempts to perform operations to a given memory location in an arbitrary order leading to an inconsistent/non-deterministic behavior. In general:

- this occurs when two threads access the same shared variable, and
- at least one thread modifies the variable
- the accesses are concurrent, i.e., unsynchronized
- leads to a non-deterministic behavior depending on timing of accesses

Type of Races

- **Read after Write (RAW):**

```
T1: X = .....
T2: ..... = X
```

- **Write after Read (WAR):**

```
T1: ..... = X
T2: X = .....
```

- **Write after Write (WAW):**

```
T1: X = .....
T2: X = .....
```

In most cases either synchronization and/or privatization of variables can be used to tackle with this issue.

Race conditions are hard to find with traditional approaches as they are **non-deterministic** and since debugging remains as a method to solve issues but **debugging in parallel case can change timings**. Therefore, dedicated race detection tools are used to detect the following:

- Static and/or dynamic instrumentation of all memory accesses
- tracking synchronization
- detection of uncynschronized access of the same memory

Some examples of the diagnostic tool could be **Helgrind, Intel inspector** and so on .....

3.9 Key Challenges in Shared Memory/OpenMP Parallelism

- How to get a parallel program?
  ⇒ **removing dependencies through code transformations**
- How to get a correct program?
  ⇒ **Avoiding race conditions**
- How to get a fast program?
  ⇒ **balancing lock granularity and hence, improving lock contention**

3.9.1 Key Performance Aspects

The key performance issues that improve the behavior of a parallel program are (however, one must consider the optimization of sequential section first):

- **Issue 1: Synchronization Overheads**
  - **Synchronizations** add to performance overheads as nothing useful is being done at those points in the program
  - **Good locking behavior**
    - To use lock only when necessary
    - effective implementation of lock in order to reduce the time spent in locks
    - reordering of locks in order to avoid deadlock cases
    - if critical section has to be substantial, then overlapping helps
  - Alternative: **Lock-free data structures**
    - avoid use of mutex/critical sections
    - carefully manipulate memory to avoid bad "immediate" states

- **Issue 2: Cache Behavior and Locality**

  Improving cache performance is critical for sequential codes. In parallel codes, this situation is worse because:

  - More threds requires more bandwidth
  - More pressure on caches
  - Contention of main memory
  - Additonal problem: **False Sharing**

    This problem occurs when two or more cores hold a copy of the same memory cache line. If one core writes, the cache line holding the memory line is invalidated on other cores. Even though another core may not be using that data (reading or writing), it may be using another element of data on the same cache line. The second core will need to reload the line before it can access its own data again.

    The cache hardware ensures data coherency, but at a potentially high performance cost if false sharing is frequent. **A good technique to identify false sharing is to catch unexpected sharp increase in the next-level cache misses using hardware counters or other performance tools.**

    ***False sharing**, therefore, is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line*

- **Issue 3: Threads and Data Locality, i.e., Mapping**

  Numa systems are dominating with the following features:

  - Multiple sockets per node/system
  - Multiple cores per sockets
  - Each socket has its own memory
  - All memory globally addressable

  Different data access latencies:

  - Local access to memory located on the same socket as the thread: **Short Access Latency**

- Remote access to memory located on a different socket as the thread: **Long Access Latency**

Location of data and threads require some coordination of threads and memory and at least logical association.

Placing Memory: Where does memory get physically located?

⇒ The answer to this question is the **typical policy of "first touch"**

- allocation only creates virtual space
- first access allocates physical memory
- memory is allocated close to the accessing thread
- no later migration

The consequences of the above are:

- initialization is important, as it determines locality; use parallel loops to initialize memory
- should prevent threads from later migration; typical for HPC and OpenMP but not common across all thread programming
- memory placement granularity is pages; important if using large pages

Thread Locations in OpenMP

⇒ These are controlled by ICV: `OMP_PROC_BIND`

- **true**: threads are bound to core/hw-threads
- **false**: threads are not bound and can be migrated
- **master**: new threads are always co-located with master
- **close**: new threads are "close" to their master threads
- **spread**: new threads are spread out as much as possible

NOTE: If not specified or a different value, behavior is implementation defined. It can also

have a list of values to define each nesting level

Description of available locations, also known as places: `OMP_PLACES` ICV. The details are as follows:

- describes ordered list and hierarchy of all available hardware threads
- examples: $\{0:4\}, \{4:4\}, \{8:4\}, \{12:4\}$
- master thread executes on first place

3.9.2 NUMA Management via `libnuma`

These are selection of routines to enable NUMA management for Linux. The primary routines include:

- routines to influence thread location
- routines to influence data location

The header `#include <numa.h>`. Here are some examples:

- Memory allocation:

```
- void* numa_alloc_onnode(size_t size, int node);
- void* numa_alloc_local(size_t size);
```

- Thread Binding:

```
- void numa_bind(struct bitmask *nodemask);
```

- Query routines:

```
- int numa_num_configured_nodes();
```

3.9.3 Scaling Cache Coherent Shared Memory Machines

The shared memory model in hardware and software is straightfoward, then what keeps us from only using it?
The following keys features on UMA and NUMA architecture answer these questions:

- **UMA Architectures:**
  - Central memory system becomes the main bottlenecks
  - Memory consistency rules are necessary, but expensive
  - must sustain the bandwidth requirement for all CPUs

  UMA systems are easy to use with predictable performance. It has a standard thread model and easy to scale codes. **This approach is limited in the sense that the system cannot be scaled**, for example: not many system beyond 8 CPUs with the largest being 36 CPUs, the SGI's Challenge series
- **NUMA Architechtures:**
  - Memory system becomes distributed providing more bandwidth and reduce bottlenecks
  - CC protocol becomes a major overhead; each memory access potentially invalidates data in any cache and hence cannot scale

Chapter 4

Distributed Memory Systems and MPI –Message Passing Interface

In this chapter…

- MPI
- Distributed Memory

4.1 Distributed Memory Machines

Distributed Memory Machines are independent computers with separate compute nodes (that contain separate processors, separate memories and so on) connected through an explicit network. Full system lies on each node.
Of all the most basic is **Compute clusters** that started as a "cheap parallel processing" with several PCs/NOW (Network of Workstations) with simple HW/SW communications. **Beowulf Systems**, an example to these computing systems, is a computer cluster of normally identical commodity-graded computers networked into a small local area network with libraries and programs installed which allows processing to be shared among them.

4.1.1 Networks (or Computer Networks)

A Networks is:

- a digital telecommunications networks which allows nodes to share resources
- computing devices in a network exchange data with each other using connections (data links) between nodes
- these data links are established over cable media such as wires or optic cables, or wireless media such as Wi-Fi
- the **standard communication is through Ethernet** that is cheap and ubiquitous and the standard protocol is TCP/IP
  (**TCP/IP**, also called **Transmission Control Protocol** and **Internet Protocol** suite, provides an end-to-end communication specifying how data should be packetized, addressed, transmitted, routed, and received)
- Example for Computer Networks: **Network of Workstations (NOW)**
- **Alternative Communication: User-Level Communication** which only setup via OS kernel are protected routines to establish security
  - direct communication from user-level removes OS overheads
  - enables high bandwidth/low latencies
- Today: standard for cluster communication
  - hidden from the user
  - VIA Architecture, example: Infiband

Virtual Interface Architecture (VIA)

It is an abstract model of user-level zero-copy network and was sought to standardize the interface for high-performance network technologies known as System Area Networks(SANs).

Traditional Approach:
Network is a shared resource. With traditional network APIs, the kernel is involved in every

network communication. This is a tremendous performance bottleneck when latency is an issue.

Current day developments:
One of the main advancements in computing systems is **virtual memory**, a combination of hardware and software that creates the illusion of private memory for each process. In the same school of thought,

- a **virtual network interface** protected across process boundaries could be accessed at the user level
- with this technology, the **"consumer" manages its own buffers and communication schedules** and the **"provider" handles the protection**
- the **Network Interface Card (NIC)** provides a private network to each of the process, and a process is usually allowed to have multiple such networks
- the **Virtual Interface (VI)** of VIA refers to this network and is merely the destination of the user's communication requests. Communication takes place over a pair of VIs, one on each of the processing nodes involved in the transmission

Idea of Direct Memory Access - DMA and Remote DMA - RDMA

Traditional View:
In traditional networks, the arriving data is first placed in a pre-allocated buffer and then copied to the user-defined final destination. Therefore, copying large messages can take a very long time, and so eliminating this step is beneficial.

Current Day Developments:
Another classic development in computing systems is the idea of **Direct Memory Access**. In this:

- a device can directly access the main memory while the CPU is free to perform other tasks
- In a network with RDMA, the sending NIC uses DMA to read data in the user-specified buffer and transmit it as a self-contained message across the network. The receiving NIC

then uses DMA to place the data into the user-specified buffer. There is no intermediary copying and all of these actions occur without involvement of the CPUs, which has an added benefit of lower CPU utilization

4.1.2 Consequences for Progammability in Distributed Memory Systems

- Problem has to be manually partitioned and distributed: data has to be managed separately in different memories
- Communication must be programmed explicitly: use of detailed communication libraries
- Fault tolerance, fault handling and debugging is more complicated: a need to reason about distributed state
- Large scale MPP systems have slightly different OS environment: reduced services to minimize noise
- Login/Compile/Compute nodes may be different: requires cross compilation

4.2 MPI - Message Passing Interface: Introduction

MPI is a communication protocol for programming parallel computers. It is based on a distributed memory model which allows both point-to-point and collective communication.

Remarks on Point-to-point and Collective Communication:

- **Point-to-point Communication:** It is is used to describe a communication between one source with one destination

whereas

- **Collective Communication**

4.2.1 MPI: Library, functionalities and corresponding API Syntaxes

- **Header**: `#include <mpi.h>`

- **MPI_Init and MPI_Finalize**:
  - `int MPI_Init(int *argc, char ***argv); //argc/argv passed from main`
    This is the first call within MPI which initializes the MPI processes and is to be called by every process
  - `int MPI_Finalize(void)`
    This call is collective and to be done by the MPI processes to finishes the use of MPI and releases all the resources; all communications to be finished before calling this function
- **Communicators and MPI_COMM_WORLD**
  - Communicators are group of MPI processes and carries the communication context
  - Each process has a rank and can go from `0` to (N-1)
  - Communication across communicators are not possible
  - Default communicators:
    `MPI_COMM_WORLD` for all initial MPI processes;
    `MPI_COMM_SELF` for only the own MPI process
  - Other communicators can be derived
- **MPI Process v/s (OS) Process v/s Rank**
  MPI Process are basic units of concurrency in MPI. These are:
  - communication endpoints that can send and receive messages
  - members of MPI Groups and MPI Communicators

  MPI Processes are not the same as OS Processes as the latter is the abstraction/isolation concept in the OS and has no connection to the MPI library. However, there might be similarities in the sense of implementations as they are commonly familiar abstraction and enables isolation of global data
  Rank is an identifier of an MPI process relative to MPI communicator
- **Assigning values to size and rank**
  To be able to define the size of the MPI processes, each MPI process must understand its

role/location in the MPI code. The following commands helps execute this job:

- `int MPI_Comm_size( MPI_Comm comm, int* size)`
  returns the number of process in the process group of the given communicator.
  In the above `comm` is the communicator (by default, it is `MPI_COMM_WORLD`) which the input argument and `size` is the cardinality of the process group for `comm`
  Example: `MPI_Comm_size(MPI_COMM_WORLD, &size);`
- `int MPI_Comm_rank( MPI_Comm comm, int* rank)`
  returns the rank of the executing process of the group relative to the given communicator indicated by `rank`.
  Example: `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

- **Sending and receiving messages**

`MPI_Send`:

- send one message to another MPI process on a given communicator
- This initiates message send and blocks buffer until send buffer can be reused

```
int MPI_Send(   void* buf,          //Address to the send buffer
                int count,          //Number of dtype data element
                MPI_Datatype dtype, //Data type
                int dest,           //Rank of the receiving process
                int tag,            //Message tag
                MPI_Comm comm       //Communicator
             );
```

`MPI_Recv`:

- receives a message from another MPI process on a given communicator
- Blocking receive of a message: blocks execution until a matching receive message has been received (**possibilities of deadlock!**)

```
int MPI_Send(   void* buf,          //Address to the send buffer
                int count,          //Number of dtype data element
                MPI_Datatype dtype, //Data type
```

```
                    int source,         //rank of sending process
                    int tag,            //Message tag
                    MPI_Comm comm,      //Communicator
                    MPI_Status *status  //status information
    );
```

  - sender and tag can be specified as wild cards: `MPI_ANY_SOURCE` and `MPI_ANY_TAG`

- `MPI_Status`
  - Structure containing information on incoming message
  - it can include the following fields: `MPI_ERROR`, `MPI_SOURCE`, and `MPI_TAG`
- `MPI_Wtime` and `MPI_Wtick`
  - `double MPI_Wtime(void);` returns wall clock time since a reference time stamp in the past (unit: seconds)
  - `double MPI_Wtick(void);` returns the resolution of `MPI_Wtime`, i.e., number of second between successive clock ticks

4.3 Communications within MPI framework

4.3.1 Blocking type send variants

- `MPI_Send`
  - standad send with no extra synchronization
  - sender side handling implementation defined
- `MPI_Bsend`
  - Buffered Send: forces the use of send buffer
  - returns immediately, however, costs resources
- `MPI_Ssend`

  - Synchronous send: only returns once receiving has started
  - adds extra synchronization, but can be costly

- `MPI_Rsend`
  - Ready send: user must ensure that receive has been posted
  - enables faster communication, but need implicit synchronization

Issue with blocked communications is that **there is an extra overhead for copying the messages**.

4.3.2 Eager Protocol v/s Rendezvous Protocol

In Eager Protocol, the message is sent assuming destination can store the message being communicated. Thus, there may not be any existing receive (buffer) at the destination. This reduces the synchronization delays minimizing the latency. Example: MPI_Send

whereas

In Rendezvous Protocol, message is not sent until the destination has prepared the received buffer. Thus, increases the latency. Example: MPI_Isend

4.3.3 Bidirectional Send/Recv

- `MPI_Sendrecv`: Combined blocking send and receive

```
int MPI_Sendrecv(   void *sendbuf, int sendcount, MPI_Datatype sendtype,
                    int dest, int sendtag,

                    void *revcbuf, int recvcount, MPI_Datatype recvtype,
                    int source, int recvtag,

                    MPI_Comm comm, MPI_Status *status
                );
```

- `MPI_Sendrecv_replace`: same principle as before expect that the send and receive buffers are Combined

```
int MPI_Sendrecv_replace(   void* buffer, int count, MPI_Datatype datatype,
                            int dest, int sendtag,
                            int source, int recvtag,
                            MPI_Comm comm, MPI_Status *status
                        );
```

4.3.4 Non-blocking operations

These operations are split into start and completion such that the start finishes immediately and thus, can complete other works in between. The completion can either be **tested for** or **waited for**.
Non-blocking operations are useful for long operations where it is known that either processing or transfer of data (due to a humongous size) would take a lot of time which would be wasted if processes wait idly. The features of non-blocking operations are:

Initiation routines are counterparts to blocking P2P operations within similar arguments and same semantics

- It return with request object to track progress
- can be matched with blocking operations

Initial call only starts operations. Completions are indicated lated. Send and receive buffers are off limits before completion.

`MPI_Request`: It is a request object which is user-allocated but its state is maintained by MPI. It is variable to ensure that the initiated operations are completed.

Non-blocking P2P operations in MPI: API Syntax

- `MPI_Isend`

```
int MPI_Isend(  void* buf, int count, MPI_Datatype dtype,
                int dest, int tag,
                MPI_Comm comm, MPI_Request *request
              );
```

- `MPI_Irecv`

```
int MPI_Irecv(  void* buf, int count, MPI_Datatype dtype,
                int source, int tag,
                MPI_Comm comm, MPI_Request *request
              );
```

4.3.5 Non-Blocking Send Variants

These are similar to the blocking send variants, namely, `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, and `MPI_Irsend`.

4.3.6 Completion Operations

- **Option 1: Blocking Completion**

```
int MPI_Wait(   MPI_Request *request, //INOUT: request
                MPI_Status *status    /*OUT:   status of
                                        completion operation */
            );
```

This operation returns if request is complete or is `MPI_REQUEST_NULL`. If complete, sets request to `MPI_REQUEST_NULL` and frees all the resources. Example:

```
{
    MPI_Request req;
    MPI_Status status;
```

```
        int msg[10];
        ...
        MPI_Irecv(  msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
                    MPI_COMM_WORLD, &req);
        ...
        <do work>
        ...
        MPI_Wait(&req, &status);
        printf(''Processing message from %i\n'', status.MPI_SOURCE);
        ...
    }
```

- **Option 2: Non-Blocking/Polling completion**

```
int MPI_Test(   MPI_Request *request, //INOUT: request
                int* flag,            //OUT: flag, whether complete or not
                MPI_Status *status    //OUT: status of completed operation
            );
```

This operation returns immediately. If the `flag` is set to true, then the request is complete or `MPI_REQUEST_NULL`. If completed, sets the request to `MPI_REQUEST_NULL` and frees up all the resources. Example:

```
    {
        MPI_Request req;
        MPI_Status status;
        int msg[10], flag;
        ...
        MPI_Irecv(  msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
                    MPI_COMM_WORLD, &req);
        do
        {
            ...
```

```
            <do work>
            ...
            MPI_Test(&req, &flag, &status);
        } while (flag==0);
        printf(''Processing message from %i\n'', status.MPI_SOURCE);
    }
```

4.3.7 Extended Wait and Test Operations

Here are some extension to the previously discussed MPI functions, namely, `MPI_Test` and `MPI_Wait`, which are useful in different scenarios.

Wait Operations

- `MPI_Wait`
  This blocks request for one request and outputs the status of the object
- `MPI_Waitall`
  This blocks request for several requests. The input for this is an array of requests (of which some can be `MPI_REQUEST_NULL`). It returns as an array of status objects once all the requests in the input array is complete
- `MPI_Waitany (MPI_Waitsome)`
  It is blocking wait for several requests with the input same as in the case of `MPI_Waitall` (array of requests) and the same output type (an array of status objects). However, it returns once at least one of the requests is complete

Test Operations

- `MPI_Test`
  It checks the completion of one request with two outputs one being the value (in `flag`) set to true if complete and the other being, status object

- `MPI_Testall`

  This checks the completion of several requests that are input in form of an array (of which some could be `MPI_REQUEST_NULL`). It return with the `flag` value true if all the requests are complete. Additionally, it also has array of status objects as the output

- `MPI_Testany (MPI_Testsome)`

  Similar to the `MPI_Testall` in terms of input and output. However, this returns immediately with `flag` value true if at least one of the requests is complete

4.4 Communicators and MPI_COMM_WORLD

Communicators are group of MPI processes that provide communication context for the group. Any communication operation is done in the context of a communicator in which processes have a rank from `0` to N-1 (where N becomes the number of processes associated with the communicator context). Communication across communicators are not possible as context differs. **It must be noted that a communicator can be in multiple communicators**.
Default communicator:

- `MPI_COMM_WORLD`: all initial MPI processes
- `MPI_COMM_SELF`: contains only the own MPI process

Other communicators can be derived from these.

4.4.1 Creating New Communicators

Creating of subgroups communicator is to isolate communications such that only certain small groups of processes are able to communicate between each other. This avoids cross-talks and reduce its complications. Such a feature is very important for libraries where different libraries should use different communicators.

Typical strategy for libraries

- libraries get a communicator passed at initialization, often called `MPI_COMM_WORLD`

- the library then clones the communicator and stores the handle for further operations

```
int MPI_Comm_dup(   MPI_Comm comm,      //IN: communicator(handle)
                    MPI_Comm *newcomm   //OUT: copy of comm(handle)
                );
```

Creating subcommunicators

```
int MPI_Comm_split( MPI_Comm comm,      //IN: Parent communicator
                    int color,          //IN: Subset color
                    int key,            //IN: rank order in newcomm
                    MPI_Comm *newcomm   //OUT: new communicator
                  );
```

Synopsis:

- `color`:
  an integer that permits to decide in which of the sub-communicators the current process will fall. All the processes of `comm` for the same numerical value of `color` will be a part of the new sub-communicator `newcomm`. For example: if we define `color = rank%2` (`rank` being the rank of process in `comm`), then one would create (globally) two new communicators: one with processes with odd ranks and other with processes with even ranks. However, each processes will only be seeing one of these new communicators they are part of
- `key`:
  It just permits to optionally decide how the processes will be ranked into the new communicators they are part of. For example, if you set `key = rank` then the order of ranking(not the ranking itself) in each new communicators `newcomm` will follow the order of ranking in original communicator `comm`. It we set this value to `0`, then the ranking in each new communicator is whatever the library decides
- Communicators should be freed when no longer in use:
  `int MPI_Comm_free(MPI_Comm *newcomm)`
- **REMARKS:**

- `MPI_Comm_split(comm, 0, rank, &newcomm)` will duplicate `comm` into `newcomm` (just as `MPI_Comm_dup()`)
- `MPI_Comm_split(comm, rank, rank, &newcomm)` will just return an equivalent of `MPI_COMM_SELF` for each of the processes
- MPI processes can opt out. For example, if we pass `MPI_UNDEFINED` as `color` then it returns `MPI_COMM_NULL` as the new communicator

- **Example: Row and Column Communicators**

```
int rank, size, col, row;
MPI_Comm row_comm, col_comm;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
Row = rank % N;
Col = rank / N;
MPI_Comm_split( MPI_COMM_WORLD,
                row, col, &row_comm);
MPI_Comm_split( MPI_COMM_WORLD,
                col, row, &col_comm);
...
MPI_Bcast(buf, 1, MPI_INT, 0, col_comm);
```

4.5 MPI Derived Datatypes

Besides the basic datatypes, that are used to specify the data layouts, MPI also provides the functionality to construct complex datatypes/datastructures. The main purposes of doing so is to:

- representation of complex datatypes
- enable MPI implementations for code optimizations
- implicit conversion of data between sender and receiver during communication

- implicit gather/scatter like functionality for data transfer

The use case of the derived datatypes are in the following use cases:

- messages with multiple datatypes
- repeated access to spread out data structures
- simplification of code

However, the usage of **derived datatypes has certain overheads** which arise due to **creation and destruction of datatypes** and also from the fact that **not many implementations are optimized for datatypes**.

Some basic functions/APIs for Derived Datatypes

- **Datatype handle**: `MPI_Datatype`
- **Datatype commit**: `int MPI_Type_commit(MPI_Datatype *datatype)`
  It is necessary for a derived datatype to be committed before using. However, uncommitted datatypes can be used for creating new datatypes
- **Freeing derived datatypes**: `int MPI_Type_free(MPI_Datatype *datatype)`

4.6 Communication Modes

So far all the communications that have been discussed is "two-sided", i.e., the sender and the receiver model for a point to point communication. The advantages of this mode of communication are:

- simple concept
- clear understanding of the communication locations
- implicit synchronization(s)

However, there are also **disadvantages** in form of overheads that can cause performance isses. They are **overheads due to implicit synchronizations, requires active involvement on both**

sides, and **receiver can delay the sender** and in certain cases vice-versa.

4.6.1 One-sided Communication

At this point we introduce an alternative to "two-sided" communication, i.e., **One-sided communication**:

- direct memory access to other memories
- receiver doesn't need to be involved
- requires extra synchronization

Key ideas revolving around "one-sided communication" are:

- to decouple data movement with process synchronization, i.e., movement of data without requiring the remote process to synchronize
- each process exposes a part of its memory to other processes to which the other processes can directly read or write

This is the concept of **RMA, also called "Remote Memory Access"** in MPI. The idea central to this concept is that of borrowing shared memory parallelism within the MPI framework. However, as against shared memory parallelism, it is limited by exposing only parts of the memory and a missing functionality of automatic consistency. Thus as against OpenMP and Pthreads, MPI RMA:

- **shares/exposes on a part of the remote memory**
- **apart from this sharing, runs in distinct processes which assumes "shared nothing"**
- no transparent access to memory (explicit get and put calls; still thinks of the data as message)
- **still a library implementation with the need for hardware support**

4.6.2 RMA Synchronization Models

MPI RMA model allows data to be accessed only within an **"epoch"**. Three types of epochs possible:

- Fence (active target)
- Post-start-complete-wait (active target)
- Lock/Unlock (passive target)

Fence Synchronization

```
MPI_Win_fence(assert, win)
```

- collective over all processes wrt. "win"
- start epoch with fence → do exchange → end epoch with fence

Active Synchronization

- Target: Exposure epoch
  - Opened by `MPI_Win_post`
  - Closed with `MPI_Win_wait`
- Origin: Access epoch
  - Opened by `MPI_Win_start`
  - Closed with `MPI_Win_complete`
- all may block: to enforce P-S/C-W ordering; processes can be both origins and targets
- a) The synchronization between `post` and `start` ensures that the put call of the origin process does not start until the target process exposes the window (with the `post` call); the target process will expose the window only after preceding local accesses to the window have completed (**it is to be noted here that** `post` **call takes a smaller group of processes, to which it exposes the window, as argument**)

b) The synchronization between `complete` and `wait` ensures that the `put` call of the origin process completes before the window is unexposed (with the `wait` call) (**it is to be noted here that the origin can indicate a smaller group of process to retrieve data from;** `start` **takes these processes as an argument**)

c) The target process will execute following local accesses to the target window only after the `wait` returned

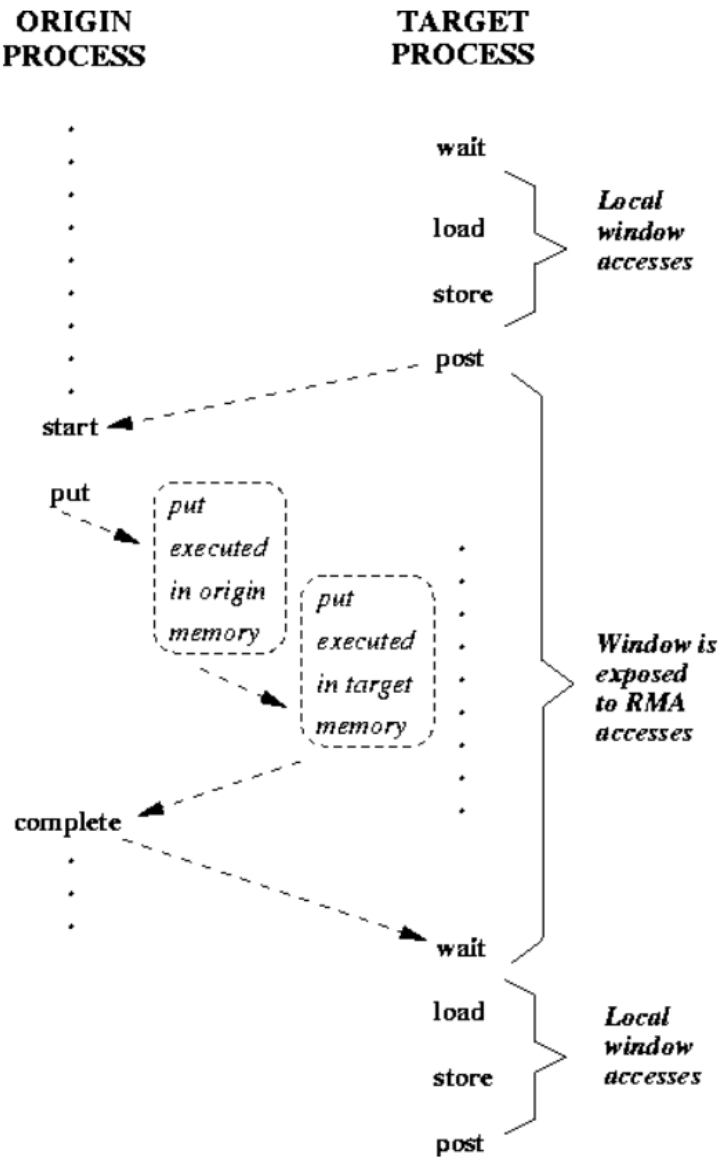

Figure 4.1: *Diagramatic representation of Active Synchronization*

Passive Synchronization

- Epochs for this synchronization are lock and unlock
- communication operations within epoch are all nonblocking
- the first origin process communicates the data to the second origin process, through the memory of the target process
- the target process is not explicity involved in the communication

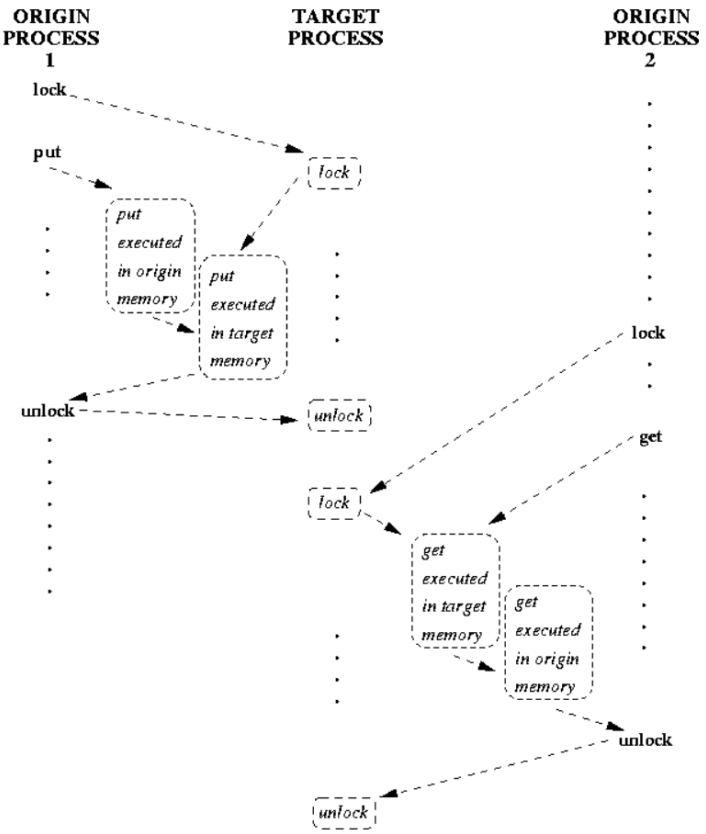

Figure 4.2: *Diagramatic representation of Passive Synchronization*

Difference between Active and Passive Synchronization

The main difference between the Active and Passive Sychronization in MPI RMA/One-sided Communication is that **former the target is actively involed in the communication** by issuing commands for synchronization that are of blocking nature. However, in the latter, there is no issue of any such commands and **the target is not explicity involved in the communications**.

4.7 Hybrid Programming: Combining Shared Memory and Distributed Memory Programming

Nodes in distributed memory systems are shared memory nodes. The need for messaging library like MPI is only for cross-node communication. So, can performance be improved on the node?

Option 1: MPI

- advantageous since it uses one single model, portability
- disadvantages: no full use of sharing capabilities
- thus, there might be performance related issues

Option 2: explores the idea of **Hybrid Programming** which uses shared memory model within an MPI process. MPI + OpenMP is a typical option.

4.7.1 Threading and MPI

Base for any MPI program is a set of sequential programs typically one per base. Each calls `MPI_Init`.

What happens if one uses a threaded program as base?

- MPI 1.x was written without threads in mind
- It is for this reason that `MPI_Init` assumes single thread and MPI does not need to be thread safe
- Access from several threads can cause problems during execution
  - shared data structures will require synchronization and need for some consistency model
  - coordinate access to NIC
  - call back coordination

4.7.2 MPI's Four Levels of Thread Safety

New MPI Initialization routine:

```
int MPI_Init_thread(    int *argc, char ***argv,
                        int required,
                        int* provided
                    );
```

Application states needed level of thread support. MPI returns the one its supports.

- `MPI_THREAD_SINGLE`: only one thread exists in the application
- `MPI_THREAD_FUNNELED`: multithreaded, but only the main thread makes the MPI calls
- `MPI_THREAD_SERIALIZED`: multithreaded, but only one thread makes MPI calls at a time
- `MPI_THREAD_MULTIPLE`: multithreaded and any thread can make API calls at any time

Consequences of using `MPI_THREAD_MULTIPLE`

- Each call to MPI completes on its own and the effect is as if they would have been called sequentially in some order
  - BLocking thread will only block the calling thread
- User is responsible for making sure racing calls are avoided, for example: access to an already freed object
  - Collective operations must be correctly ordered among the threads

4.7.3 Threads-Levels when using OpenMP

Similar to thread + MPI case, we use `MPI_Init_thread` to MPI with the right thread level.

- `MPI_THREAD_SINGLE`: there is no OpenMP multithreading in the program
- `MPI_THREAD_FUNNELED`
  Must ensure that all the MPI calls are made by the master thread

  - outside the parallel region
  - during the master region

  This happens to be the most common scenario

- `MPI_THREAD_SERIALIZED`
  Must ensure that only one thread calls MPI at one time
  - Critical or single regions
  - through added synchronizations

- `MPI_THREAD_MULTIPLE`
  - any thread can make an MPI call at any time
  - also compatible with OpenMP tasking

4.7.4 Hybrid Programming: MPI + OpenMP

OpenMP regions between MPI communications

- Easy to understand
- Fits many programs
- Matches SPMD mode
- Required thread funneled only

The major disadvantage is **Amdahl's law in the non-OpenMP regions**, i.e., complete parallelization of the program is not possible; at some points communication and synchronizations impacts the scalability of the parallelization.

Best practice is follow Hierarchical Data Decomposition:
Need to split data structures(for MPI), then share data points(for OpenMP threads)

Conclusion

And how it ends we shall decide later.

References

[1] Hesthaven J S. *Numerical Methods for Conservation Laws: From Analysis to Algorithm*. Society for Industrial and Applied Mathematics (SIAM) 2018.

[2] Bressan A. *Hyperbolic Conservation Laws: An illustrated Tutorial*. Department of Mathematics, Penn State University (USA) 2009.

[3] Levandosky J. *Lecture Notes MA220A: Partial Differential Equations of Applied Mathematics*. Department of Mathematics, Standford University (USA) 2002.
weblink: https://web.stanford.edu/class/math220a/