## Parallel Programming Models

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

## Parallel Programming Models

- Parallel programming models exist as an abstraction above hardware and memory architectures.
- These models are **NOT** specific to a particular type of machine or memory architecture.
- Any of these models can be implemented on any underlying hardware.

## Shared Memory Model (without threads)

- In this programming model, tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks may be used to control access to the shared memory.
- An advantage is that the notion of data "ownership" is lacking,
- so there is no need to specify explicitly the communication of data between tasks.

## Shared Memory Model (without threads)

- An important disadvantage is that it becomes more difficult to understand and manage data locality.
  - Keeping data local to the processor that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.
  - Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

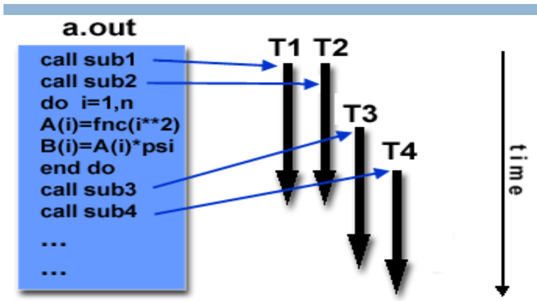
## Implementations

- Native compilers and/or hardware translate user program variables into actual memory addresses, which are global.
- On stand-alone machines, this is straightforward.
- On distributed shared memory machines, such as the SGI Origin, memory is physically distributed across a network of machines,
- but made global through specialized hardware and software.

## Threads Model

- This is a type of shared memory programming.
- In this model, a single process can have multiple, concurrent execution paths.
- An analogy that can be used to describe threads is
- the concept of a single program that includes a number of subroutines:

## Threads Model Example




## Threads Model Example

- The main program a.out is scheduled to run by the native operating system.
- a.out loads and acquires all of the necessary system and user resources to run.
- a.out performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.

## Threads Model Example

- Each thread has local data, but also, shares the entire resources of a.out.
- This saves the overhead associated with replicating a program's resources for each thread.
- Each thread also benefits from a global memory view because it shares the memory space of a.out.

## Threads Model Example

- A thread's work may best be described as a subroutine within the main program.
- Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations).

## Threads Model Example

- This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
- Threads can come and go,
- but a.out remains present to provide the necessary shared resources until the application has completed.

## Implementations:

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code

## Implementations:

- In both cases, the programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing.
- Historically, hardware vendors have implemented their own proprietary versions of threads.

## Implementations:

- These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

## POSIX Threads

- Library based; requires parallel coding
- Specified by the IEEE POSIX 1003.1c standard (1995).
- C Language only
- Commonly referred to as Pthreads.

## POSIX Threads

- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
- Very explicit parallelism; requires significant programmer attention to detail.

## OpenMP

- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.

## OpenMP

- Portable / multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

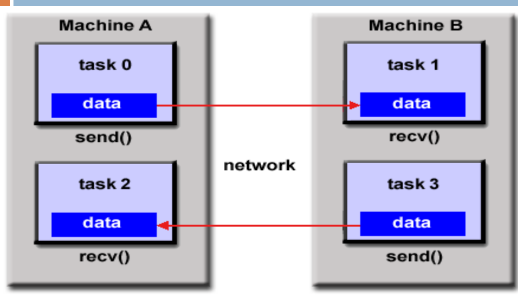
## Distributed Memory / Message Passing Model

- This model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation.
  - Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.

## Distributed Memory / Message Passing Model

- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process.
- For example, a send operation must have a matching receive operation.

## Distributed Memory / Message Passing Model




## Implementations

- From a programming perspective, message passing implementations usually comprise a library of subroutines.
- Calls to these subroutines are imbedded in source code.
- The programmer is responsible for determining all parallelism.

## Implementations

- Historically, a variety of message passing libraries have been available since the 1980s.
- These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

## Implementations

- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996.
- MPI is now the "de facto" industry standard for message passing,
- replacing virtually all other message passing implementations used for production work.

## Implementations

- MPI implementations exist for virtually all popular parallel computing platforms.
- Not all implementations include everything in both MPI1 and MPI2.

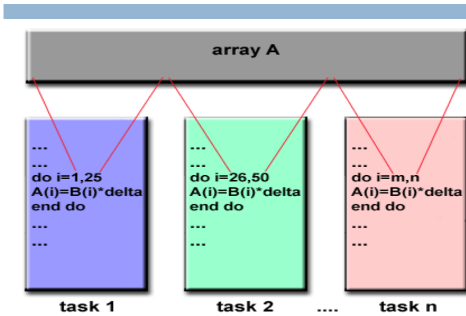
## Data Parallel Model

- The data parallel model demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set.
  - The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.

## Data Parallel Model

- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

## Data Parallel Model




## Implementations

- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs.
- The constructs can be calls to a data parallel subroutine library or,
- compiler directives recognized by a data parallel compiler.

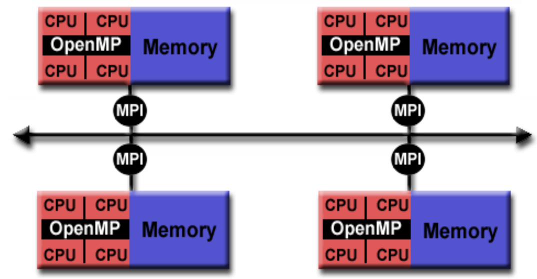
## Hybrid Model

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

## Hybrid Model

- This model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming.
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

## Hybrid Model




## Single Program Multiple Data (SPMD):

- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- SINGLE PROGRAM:
- All tasks execute their copy of the same program simultaneously.
- This program can be threads, message passing, data parallel or hybrid.

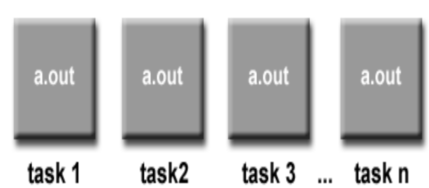
## SPMD

- MULTIPLE DATA:
- All tasks may use different data
- SPMD programs usually have the necessary logic programmed into them
- to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute.

## SPMD

- That is, tasks do not necessarily have to execute the entire program –
  - perhaps only a portion of it.
- The SPMD model, using message passing or hybrid programming,
- is probably the most commonly used parallel programming model for multi-node clusters

## SPMD



## Multiple Program Multiple Data (MPMD):

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MULTIPLE PROGRAM:
- Tasks may execute different programs simultaneously.
- The programs can be threads, message passing, data parallel or hybrid.

## MPMD

- MULTIPLE DATA:
- All tasks may use different data
- MPMD applications are not as common as SPMD applications,
- but may be better suited for certain types of problems,
- particularly those that lend themselves better to functional decomposition than domain decomposition.

## MPMD