Parallel & Distibuted Computing: Lecture 5

from Blaise N. Barney, HPC Training Materials, by kind permission of Lawrence Livermore National Laboratory's Computational Training Center

March 20, 2017

Parallel Architectures and Programming Models

Parallel Computer Memory Architectures

Shared Memory

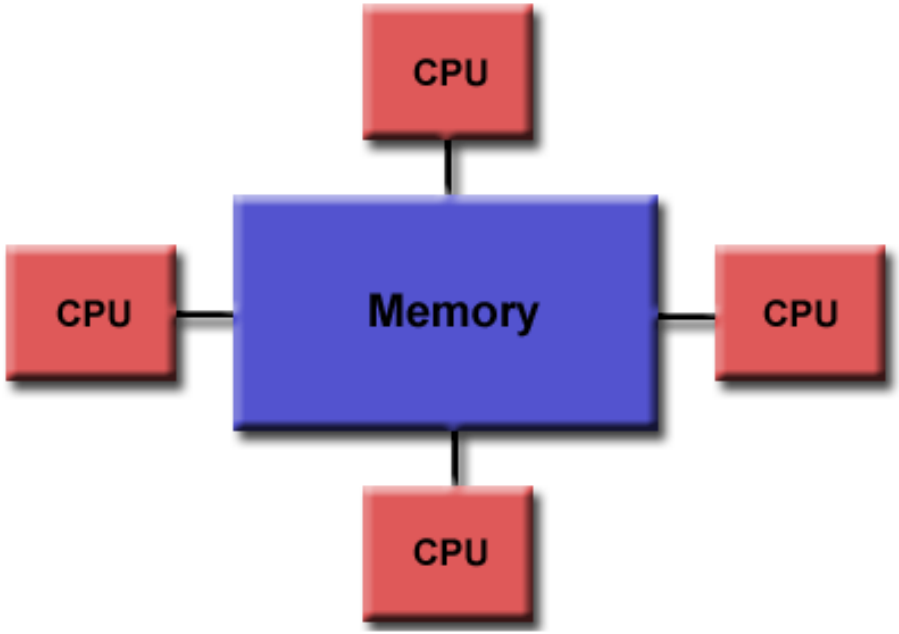
General Characteristics

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Historically, shared memory machines have been classified as UMA and NUMA, based upon memory access times.

Shared Memory

Uniform Memory Access (UMA)

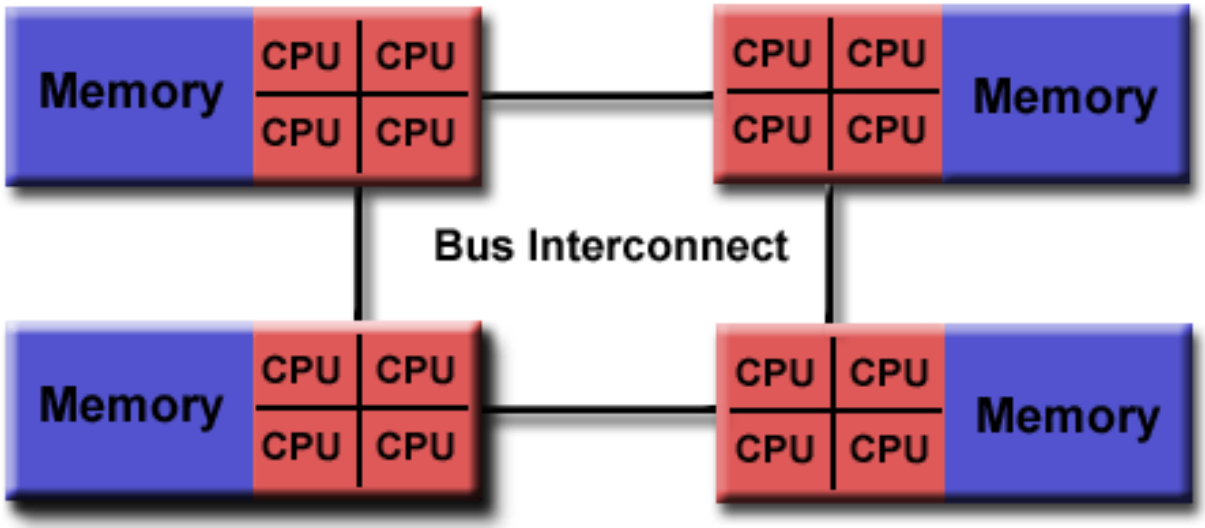
- Most commonly today Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA.
- CC: if one processor updates a location in shared memory, all processors know about the update. CC at hardware level.



Shared Memory

Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-UMA or CC-NUMA



Shared Memory

Advantages

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

Shared Memory

Advantages

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

Disadvantages

- Primary disadvantage is the lack of scalability between memory and CPUs.
  Adding more CPUs can geometrically increases traffic on the shared memory-CPU path
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Distributed Memory

General Characteristics

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

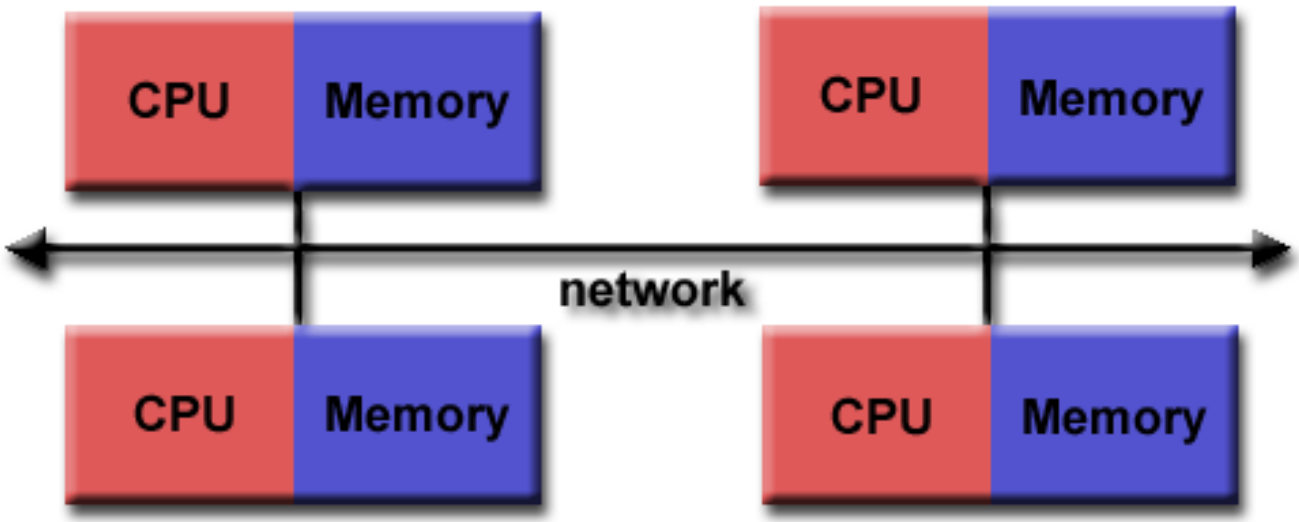


Figure 3: Distributed Memory

Distributed Memory

General Characteristics

- Processors have their own local memory.
  Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently.
  Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.
  Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Distributed Memory

Advantages

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

Distributed Memory

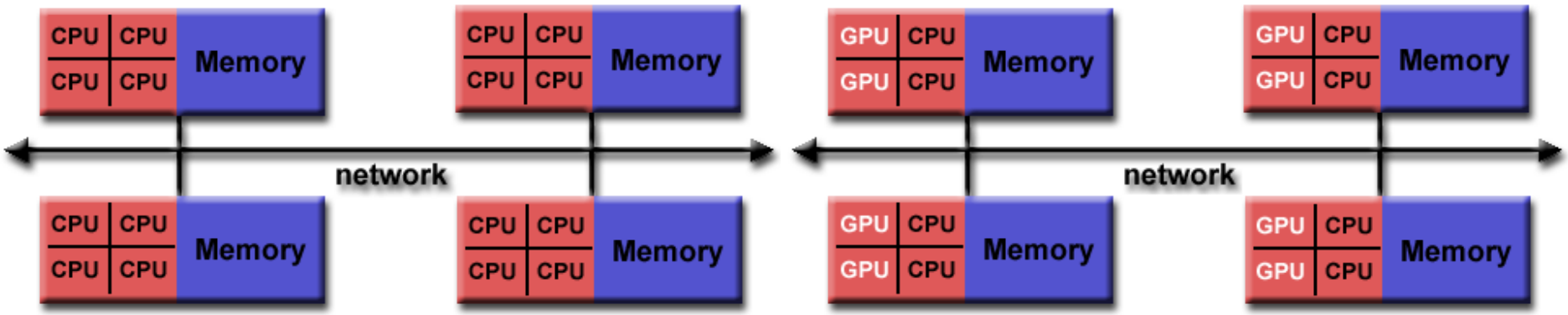
Disadvantages

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data

Hybrid Distributed-Shared Memory

General Characteristics

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



Hybrid Distributed-Shared Memory

General Characteristics

- The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory

  Therefore, network communications are required to move data from one machine to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

Hybrid Distributed-Shared Memory

Advantages and Disadvantages

- Whatever is common to both shared and distributed memory architectures.
- Increased scalability is an important advantage
- Increased programmer complexity is an important disadvantage

Parallel Programming Models

Overview

Parallel programming models exist as an abstraction above hardware and memory architectures.

There are several parallel programming models in common use:

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

Overview

Parallel programming models exist as an abstraction above hardware and memory architectures.

There are several parallel programming models in common use:

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

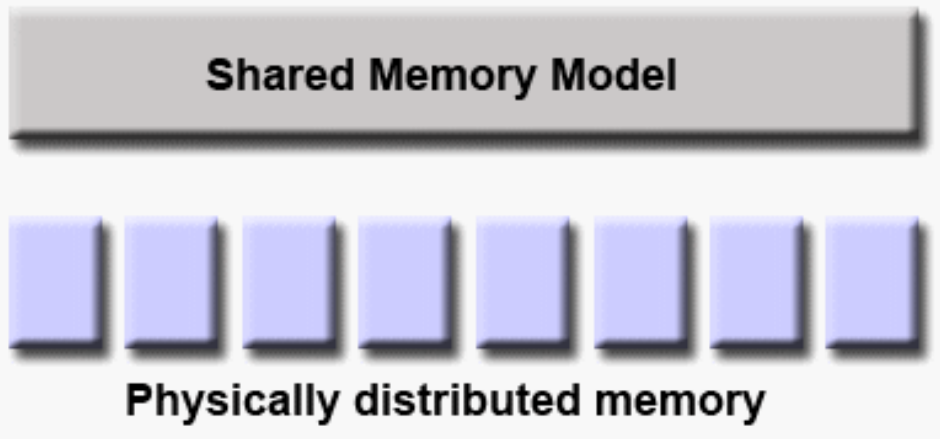
- Although it might not seem apparent, these models are NOT specific to a particular type of machine or memory architecture.

  In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples from the past are discussed below.

SHARED memory model on a DISTRIBUTED memory machine

Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across networked machines, but appeared to the user as a single shared memory global address space.

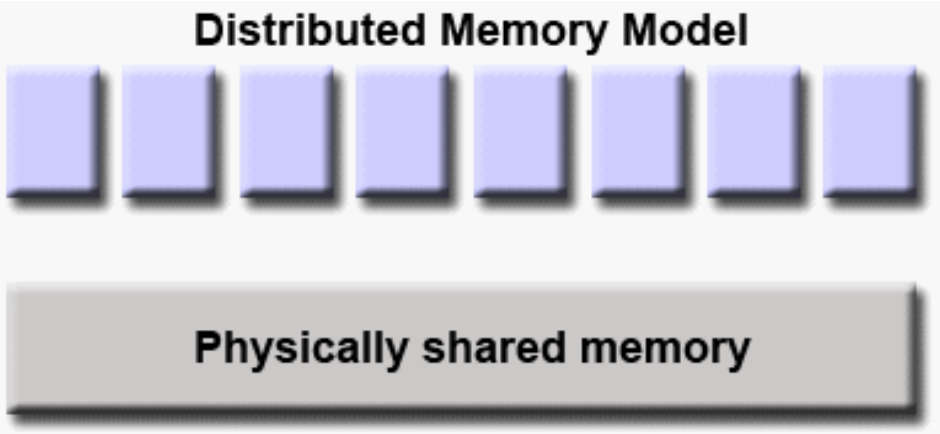
Generically, this approach is referred to as "virtual shared memory"



DISTRIBUTED memory model on a SHARED memory machine

Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of shared memory architecture, where every task has direct access to global address space spread across all machines.

However, the ability to send and receive messages using MPI, as is commonly done over a network of distributed memory machines, was implemented and commonly used.



Shared Memory Model (without threads)

- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory.

  Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality:
  - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

Shared Memory Model (without threads)

Implementations

- On stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming.

  For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc).

- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software.

A variety of SHMEM implementations are available

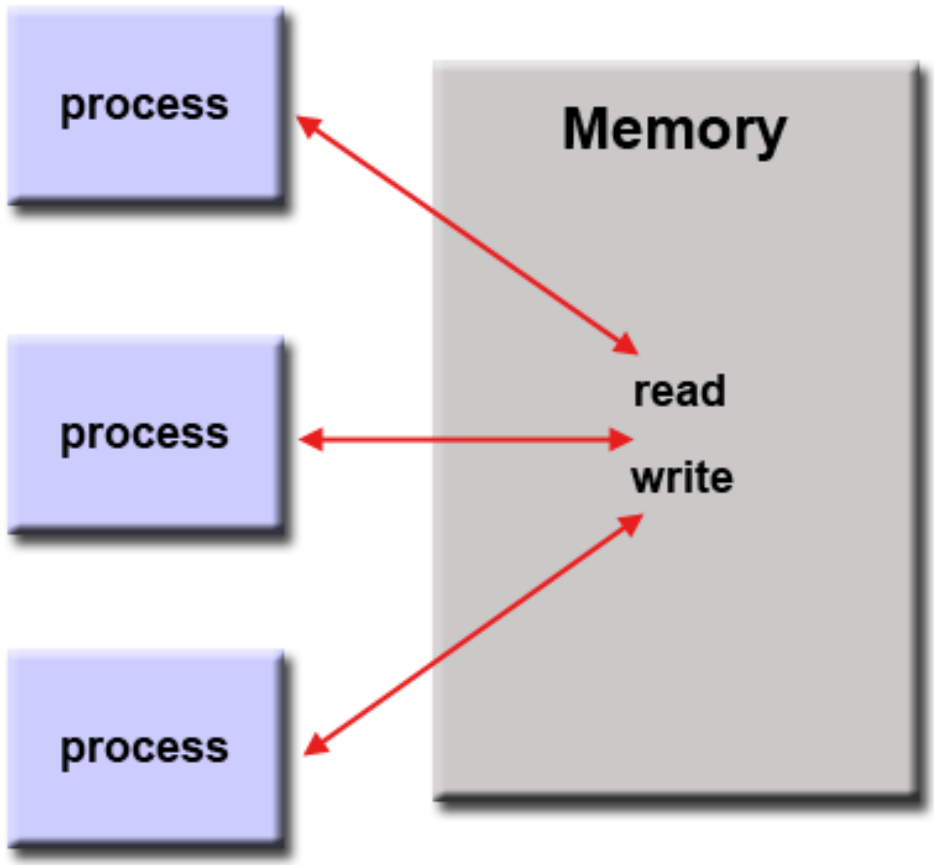
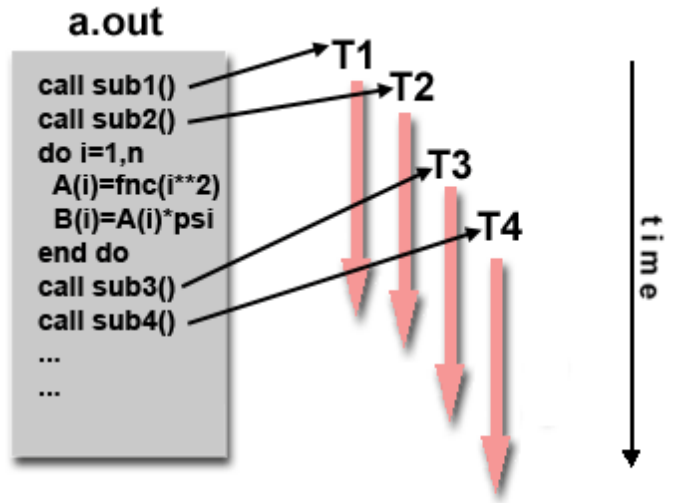


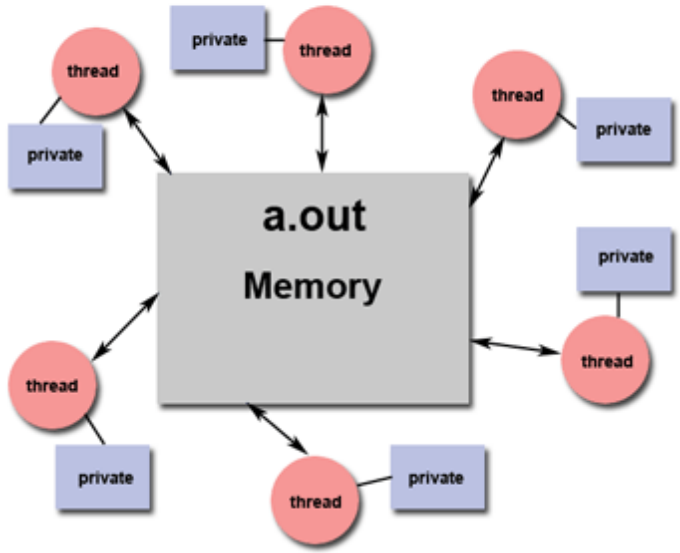
Figure 4: Shared Memory Model (without threads)

Threads Model

This programming model is a type of shared memory programming.

- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
- 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. This is the "heavy weight" process.
- 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.
- 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 ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of a.out.
- 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). 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.





Threads Model (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

  In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).
- Threaded implementations are not new in computing.
  Historically, hardware vendors have implemented their own proprietary versions of threads.
  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.

Threads Model (Implementations)

POSIX Threads

- Specified by the IEEE POSIX 1003.1c standard (1995). C Language only.
- Part ofUnix/Linux operating systems
- Library based
- Commonly referred to as Pthreads.
- Very explicit parallelism; requires significant programmer attention to detail.

POSIX Threads tutorial computing.llnl.gov/tutorials/pthreads

Threads Model (Implementations)

OpenMP

- Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals.
- Compiler directive-based
- Portable / multi-platform, including Unix and Windows platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.

Other threaded implementations exist, such as Microsoft's

OpenMP tutorial computing.llnl.gov/tutorials/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.
- 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.

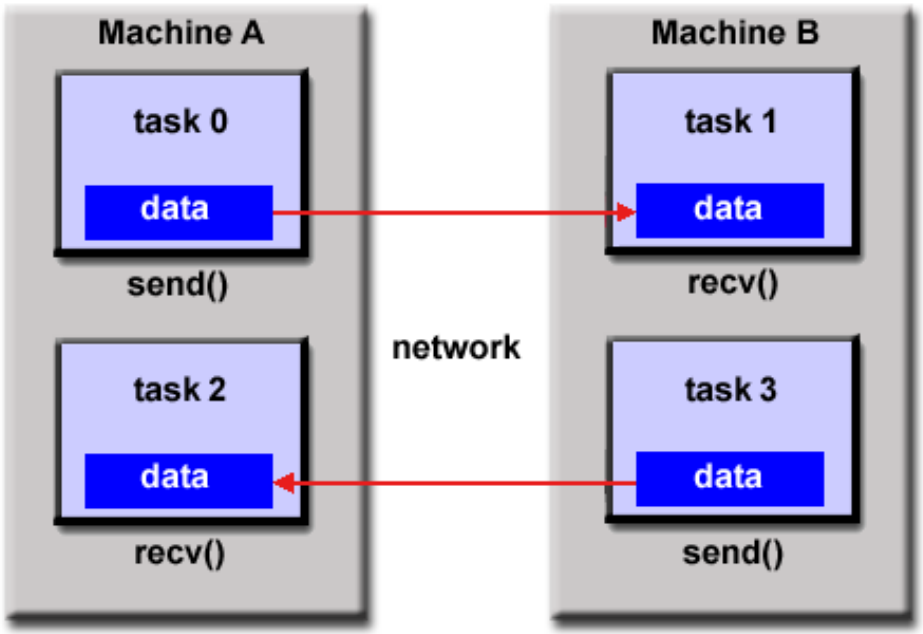


Figure 6: Message Passing Model

MPI tutorial computing.llnl.gov/tutorials/mpi

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.
- 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.
- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012. All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.
- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2 or MPI-3.

Data Parallel Model

- May also be referred to as the Partitioned Global Address Space (PGAS) model.
- The data parallel model demonstrates the following characteristics:

- Address space is treated globally
- 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.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".

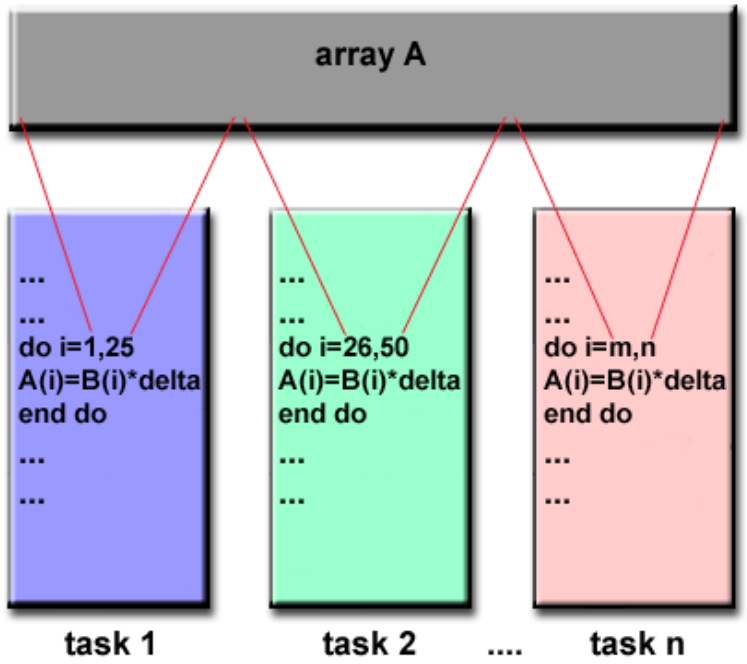


Figure 7: Data Parallel Model

- 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)

Currently, there are several relatively popular, and sometimes developmental, parallel programming implementations based on the Data Parallel / PGAS model.

- Coarray Fortran: a small set of extensions to Fortran 95 for SPMD parallel programming. Compiler dependent. More information: https://en.wikipedia.org/wiki/Coarray_Fortran
- Unified Parallel C (UPC): an extension to the C programming language for SPMD parallel programming. Compiler dependent. More information: http://upc.lbl.gov/
- Global Arrays: provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings. More information: https://en.wikipedia.org/wiki/Global_Arrays
- X10: a PGAS based parallel programming language being developed by IBM at the Thomas J. Watson Research Center. More information: http://x10-lang.org/
- Chapel: an open source parallel programming language project being led by Cray. More information: http://chapel.cray.com/

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
- This hybrid model lends itself well to the most popular (currently) hardware environment of clustered multi/many-core machines.

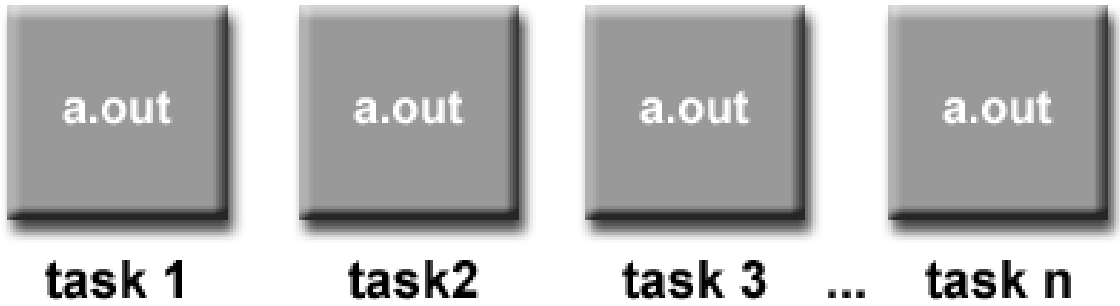
Hybrid Model

- Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU (Graphics Processing Unit) programming.
  - MPI tasks run on CPUs using local memory and communicating with each other over a network.
  - Computationally intensive kernels are off-loaded to GPUs on-node.
  - Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).
- Other hybrid models are common:
  - MPI with Pthreads
  - MPI with non-GPU accelerators
  - ...

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.
- 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. 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.



MPMP

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.
- 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 (discussed later under Partioning).

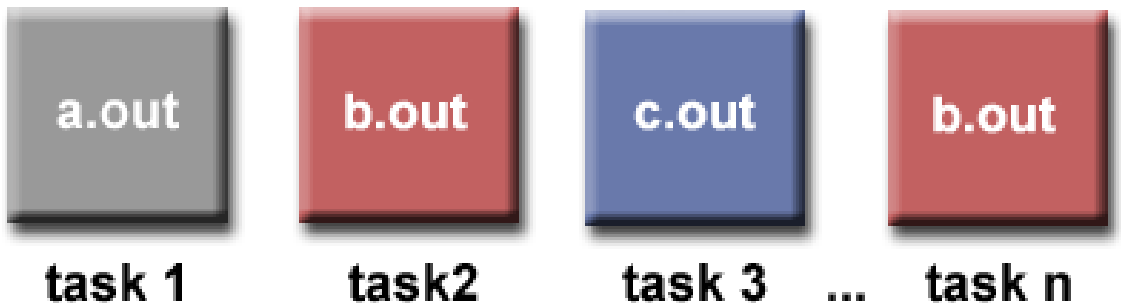


Figure 9:

References