INTRODUCTION TO HIGH PERFORMANCE COMPUTING PART 2

Based on Introduction to Parallel Computing

Lawrence Livermore National Laboratory

https://computing.llnl.gov/tutorials/parallel_comp/

PARALLEL COMPUTER MEMORY ARCHITECTURES

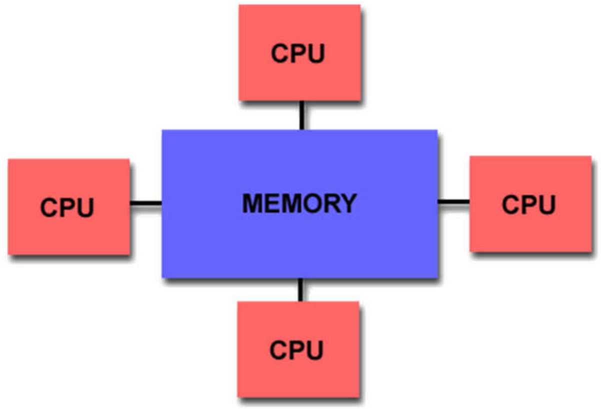
Memory architectures

- There are only three memory architectures
  - Really, there are two – the third is a combination of the first two!

1. Shared Memory
2. Distributed Memory
3. Hybrid Distributed-Shared Memory

Shared Memory

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



- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory : UMA vs. NUMA

- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA.
    - Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update.
    - Cache coherency is accomplished at the hardware level.

Shared Memory : UMA vs. NUMA

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

Shared Memory: Pros and Cons

- 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: Pros and Cons

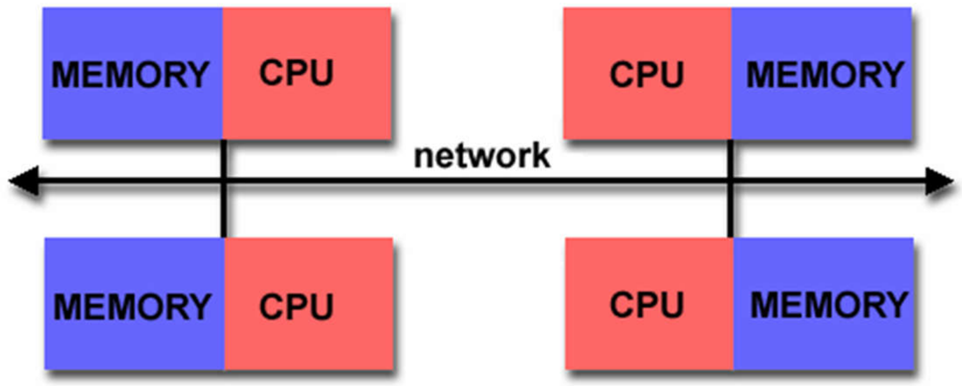
- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense of scalability: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic.
  - Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory.
  - Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- 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.

Distributed Memory

- 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 can be as simple as Ethernet.
  - There are popular alternatives ‘in-between’ these extremes such as Infiniband and Myrinet.



Distributed Memory: Pros and Cons

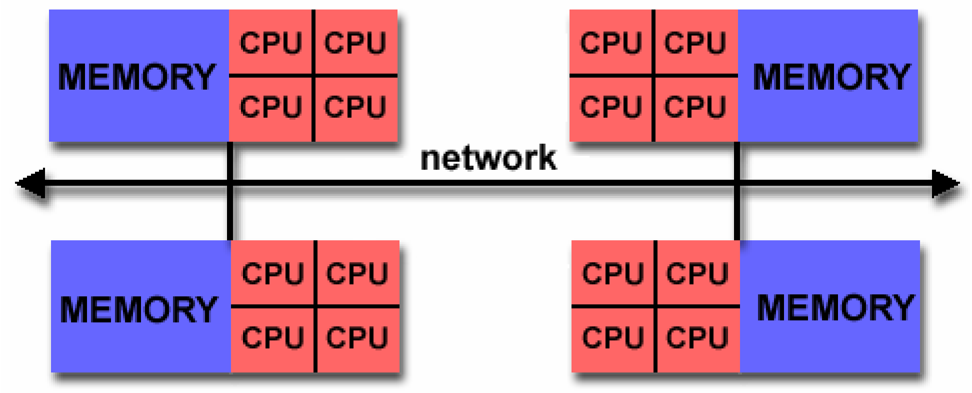
- Advantages
  - Memory is scalable with number of processors.
    - Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.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

Distributed Memory: Pros and Cons

- 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 (NUMA) times

Hybrid Distributed-Shared Memory

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



- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.

Hybrid Distributed-Shared Memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	**MPI** Threads OpenMP shmem	**MPI** Threads OpenMP shmem	**MPI**
Scalability	to 10s of processors	to 100s of processors	to 1000, 10,000s, 100,000s of processors and more!
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain, plus hardware challenges
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Hybrid Distributed-Shared Memory

- MPI (Message Passing Interface) is the de-facto standard for programming distributed (and hybrid) architectures. We will see a lot more on MPI soon!
- Note that MPI works on both shared and distributed memory architectures.
  - This does not mean that using MPI in a shared memory architecture means that you are passing messages over a network
  - It just ‘looks’ that way to the programmer.
  - Basically, if MPI can communicate via memory, it does. If it can’t, it uses the network (if there is a network)

Hybrid Distributed-Shared Memory

- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to be popular and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.
- Example: Fionn (ICHEC, Ireland): https://www.ichec.ie/about/infrastructure/fionn

PARALLEL PROGRAMMING MODELS

Parallel Programming Models

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other Models

Overview

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- **Parallel programming models exist as an abstraction above hardware and memory architectures.**

Overview

- 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.
- For instance, the shared memory model on a distributed memory machine: Kendall Square Research (KSR) ALLCACHE approach.
  - Machine memory was physically distributed, but appeared to the user as a single shared memory (global address space). Generically, this approach is referred to as "virtual shared memory".
  - Note: although KSR is no longer in business, there is no reason to suggest that a similar implementation will not be made available by another vendor in the future.
  - Message passing model on a shared memory machine: MPI on SGI Origin.

Overview

- Which model to use is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space (**not necessarily a physical one**), which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- 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. 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. This is potentially a problem for “Big Data”* problems.
    - * In HPC, almost everything is “Big Data”. Here, we really mean *Really Really Big Data.*

Shared Memory Model: Implementations

- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No common distributed memory platform implementations currently exist. However, as mentioned previously in the Overview section, the KSR ALLCACHE approach provided a shared memory view of data even though the physical memory of the machine was distributed.

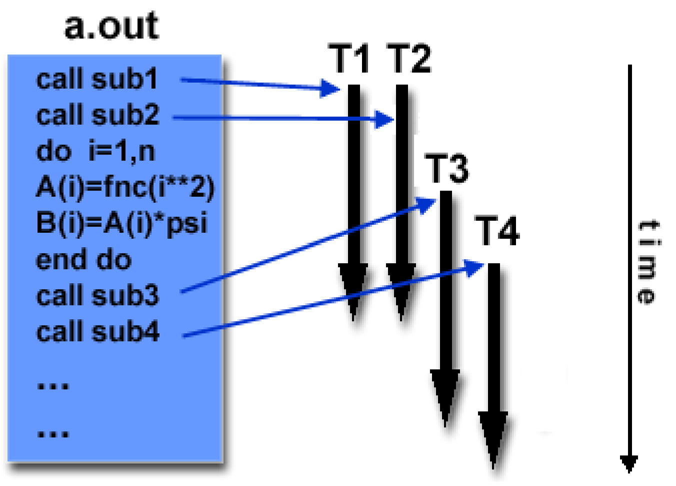
Threads Model

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines as described on the next slide.

Threads Model

- 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.
- **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.
- 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 insure 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



```
call sub1
call sub2
do  i=1,n
A(i)=fnc(i**2)
B(i)=A(i)*psi
end do
call sub3
call sub4
...
...
```

- Threads are commonly associated with shared memory architectures and operating systems.

Threads Model Implementations

- From a programming perspective, threads model implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives embedded in either serial or parallel source code (e.g. OpenMP)
- 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.
  - These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

Threads Model Implementations

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

Threads Model: OpenMP

- **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.
  - Portable / multi-platform, including Unix and other 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.

Message Passing Model

- The message passing 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 as well 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.**

Message Passing Model

- MPI (Message Passing Interface) is the de-facto library for distributed memory parallel programming.
- MPI is designed for C, C++, and FORTRAN
- However, there are bindings for:
  - Java
  - MATLAB
  - OCaml
  - Python
  - R
  - Probably more…

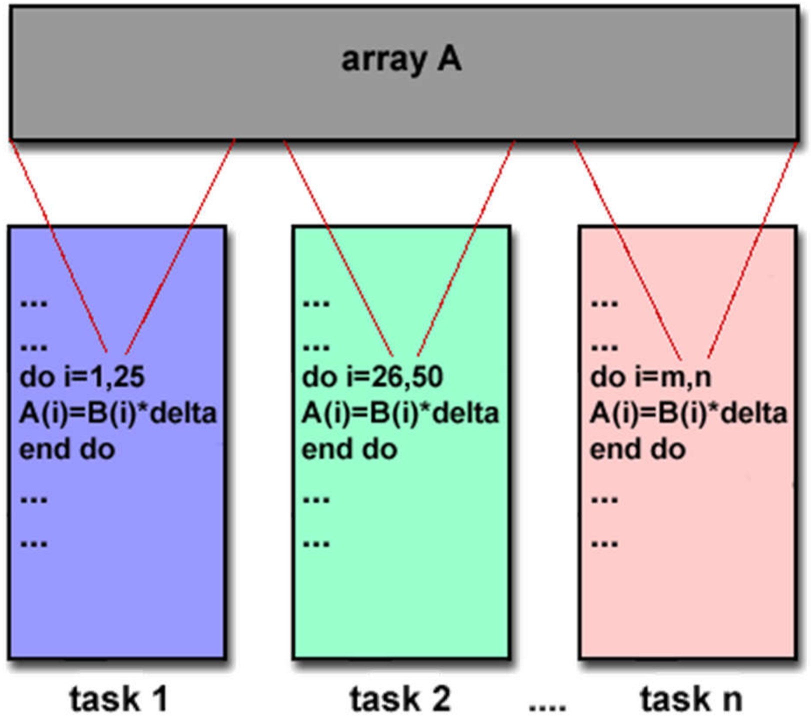
Message Passing Model Implementations: MPI

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that 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. MPI-3 was released in 2012, and 3.1 in 2015. Developments for MPI-4 are well underway.
  - http://mpi-forum.org/

Message Passing Model Implementations: MPI

- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.
  - However no programmer effort is required – MPI just ‘works’ on shared memory architectures
  - It looks like MPI is using the network to pass messages, but if it can, it uses memory. If not, it uses the network.

Message Passing Model Implementations: MPI



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.
  - 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

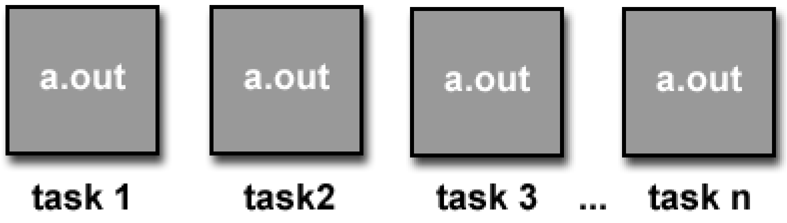
- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler.
- **Fortran 90 and 95 (F90, F95):** ISO/ANSI standard extensions to Fortran 77.
  - Contains everything that is in Fortran 77
  - New source code format; additions to character set
  - Additions to program structure and commands
  - Variable additions - methods and arguments
  - Pointers and dynamic memory allocation added
  - Array processing (arrays treated as objects) added
  - Recursive and new intrinsic functions added
  - Many other new features
- Implementations are available for most common parallel platforms.

Data Parallel Model Implementations

- **High Performance Fortran (HPF):** Extensions to Fortran 90 to support data parallel programming.
  - Contains everything in Fortran 90
  - Directives to tell compiler how to distribute data added
  - Assertions that can improve optimization of generated code added
  - Data parallel constructs added (now part of Fortran 95)
  - Implementations are available for most common parallel platforms
- **Compiler Directives:** Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms.

Data Parallel Model Implementations

- Distributed memory implementations of this model usually have the compiler convert the program into standard code with calls to a message passing library (MPI usually) to distribute the data to all the processes. **All message passing is done invisibly to the programmer**. This is a defining characteristic of the data parallel model.



Other Models

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

Hybrid

- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP).
  - This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing.
  - As mentioned in the data parallel model section previously, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, **transparently** to the programmer.

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.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- **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.**
- All tasks may use different data

Multiple Program Multiple Data (MPMD)

- 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.
- MPMD applications typically have multiple executable object files (programs).
  - While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data