Introduction to Parallel Computing

Dra. Mireya Paredes López

mireya.paredes@udlap.mx

Limits and Costs of Parallel Programming

Amdahl´s Law: Potential program speedup is defined by the fraction of code (P) that can be parallelized.

$$\text{speedup} = \frac{1}{1 - P}$$

- If none of the code can be parallelized, P=0 and the speedup=1 (no speedup).
- If all of the code is parallelized, P=1 and the speedup is infinite (in theory).

Limits and Costs of Parallel Programming

Amdahl´s Law: Potential program speedup is defined by the fraction of code (P) that can be parallelized.

$$\text{speedup} = \frac{1}{1 - P}$$

- If none of the code can be parallelized, P=0 and the speedup=1 (no speedup).
- If all of the code is parallelized, P=1 and the speedup is infinite (in theory).

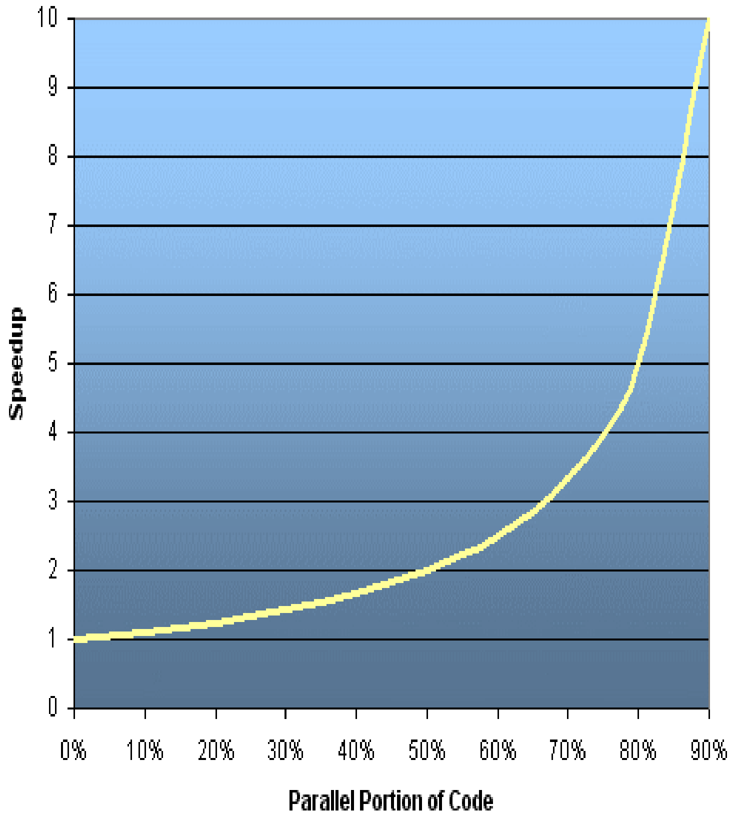
Limits and Costs of Parallel Programming

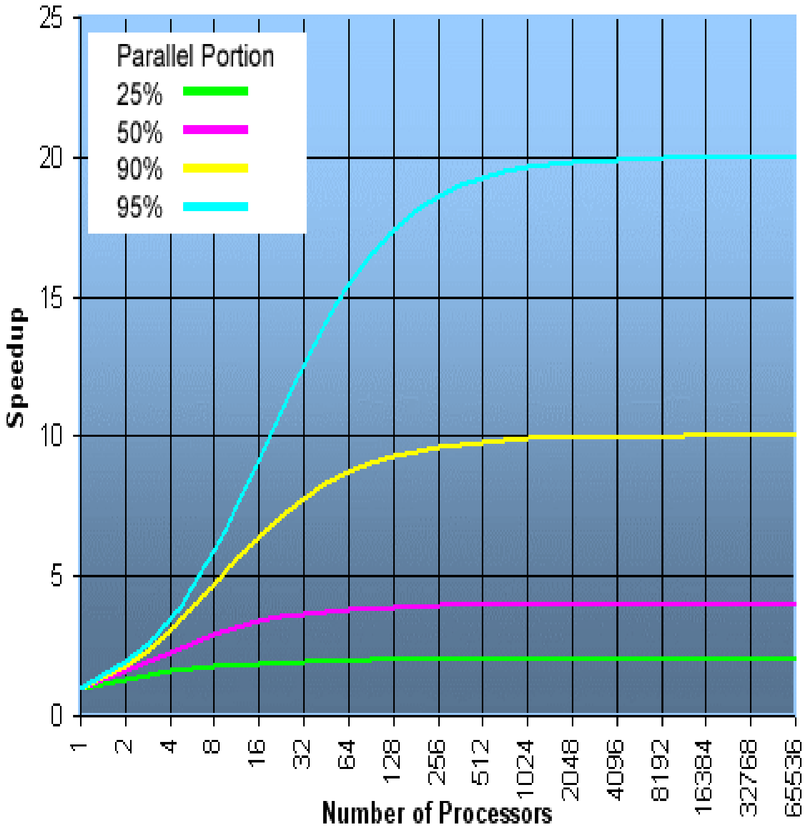
Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

- Where P = parallel fraction, N =number of processors and
- S=serial fraction.

Limits and Costs of Parallel Programming





Limits and Costs of Parallel Programming

It soons becomes obvious that there are limits to the scalability of parallelism.

N	speedup P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

“Famous” quote: You can spend a lifetime getting 95% of your code to be Parallel, and never achieve betten than 20x speedup.

Limits and Costs of Parallel Programming

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

2D Grid Calculations	85 seconds	85%
Serial fraction	15 seconds	15%

Doubling the grid dimensions and halving the time step.

2D Grid Calculations	680 seconds	97.84%
Serial fraction	15 seconds	2.16%

Problems that increase the percentage of parallel time are more ***scalable***.

Complexity

- Parallel applications are much more complex than serial applications.
- The cost or complexity are measure in programmer time.
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance
  - Adhering to “good” software development practices is essential.

Portability

- **Standarization in several APIs**, such as MPI, POSIX threads, and OpenMP → portability issues are not as serious as in the past.
- **Operating Systems** can play a key role in code portability issues.
- **Hardware architectures** are characteristically highly variable and can affect portability.
- All usual **portability issues** associated with serial programs apply to parallel programs.
  - If you use vendor "enhancements" to Fortran, C/C++, portability will be a problem.

Resource Requirements:

- The primary intent of parallel programming is to **decrease execution clock time.** → more CPU time is required.

  Example:

  parallel code running **in 1 hour on 8 processors**

  it **uses 8 hours** of CPU time
- The amount of **memory** required can be greater for parallel codes than serial codes → data replication / overhead costs
- For **short running** parallel programs →decrease in performance
- overhead costs of setting

Scalability

Strong scaling

- The **total problem size** stays **fixed** as more processors are added.
- Goal is to run the same problem size faster.
- Perfect scaling means problem is solved in 1/P time (compared to serial).

Weak scaling

- The problem *size per processor* stays fixed as more processors are added.
- Goal is to run larger problems in the same amount of time.
- Perfect scaling means problem Px runs in same time as single processor run.

Shared memory

General characteristics:

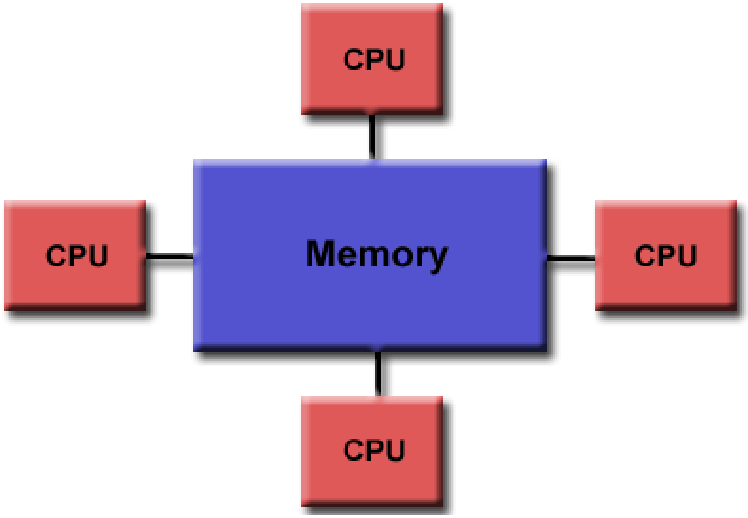
- **Shared memory** → access all memory as global address space.
- **Processors are independent** → share memory resources
- Changes in a memory location → affect all other processors.
- Shared memory is classified into two **UMA** and **NUMA**

Uniform Memory Access(UMA)

Most Symmetric Multiprocessors (SMP) machines

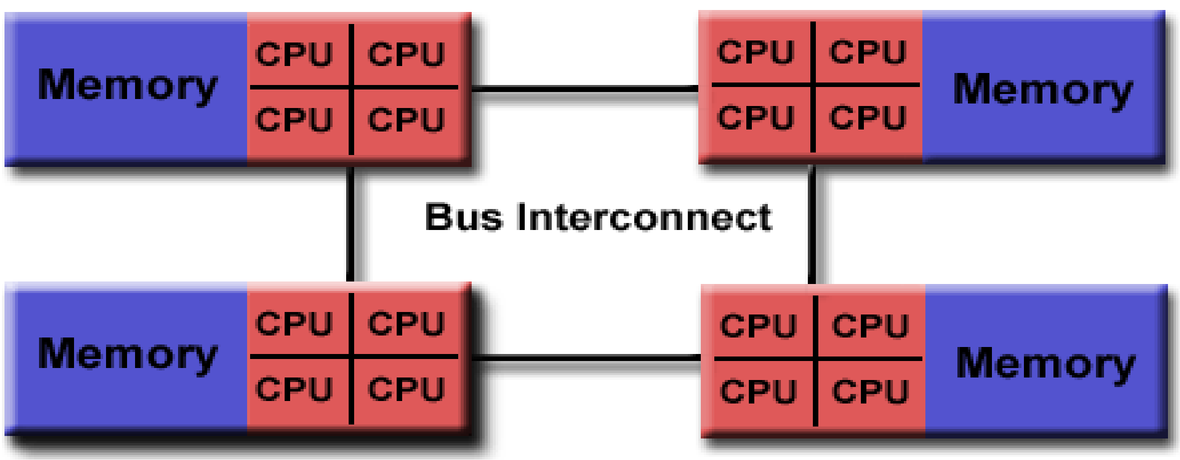
Identical processors

Equal access and access times to memory



Non-Uniform Memory Access(NUMA)

- Often made by physically linking two or more SMPS
- Not all processors have equal access time to all memories.
- Memory access across link is slower.

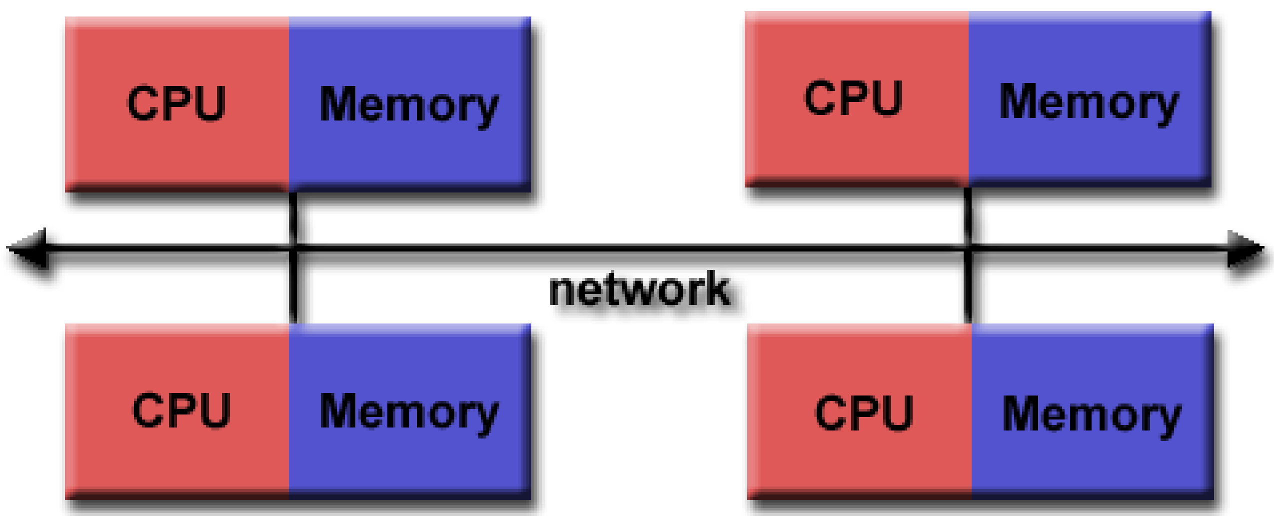


Distributed Memory

Distributed memory systems require a communication network to connect inter-processor memory.

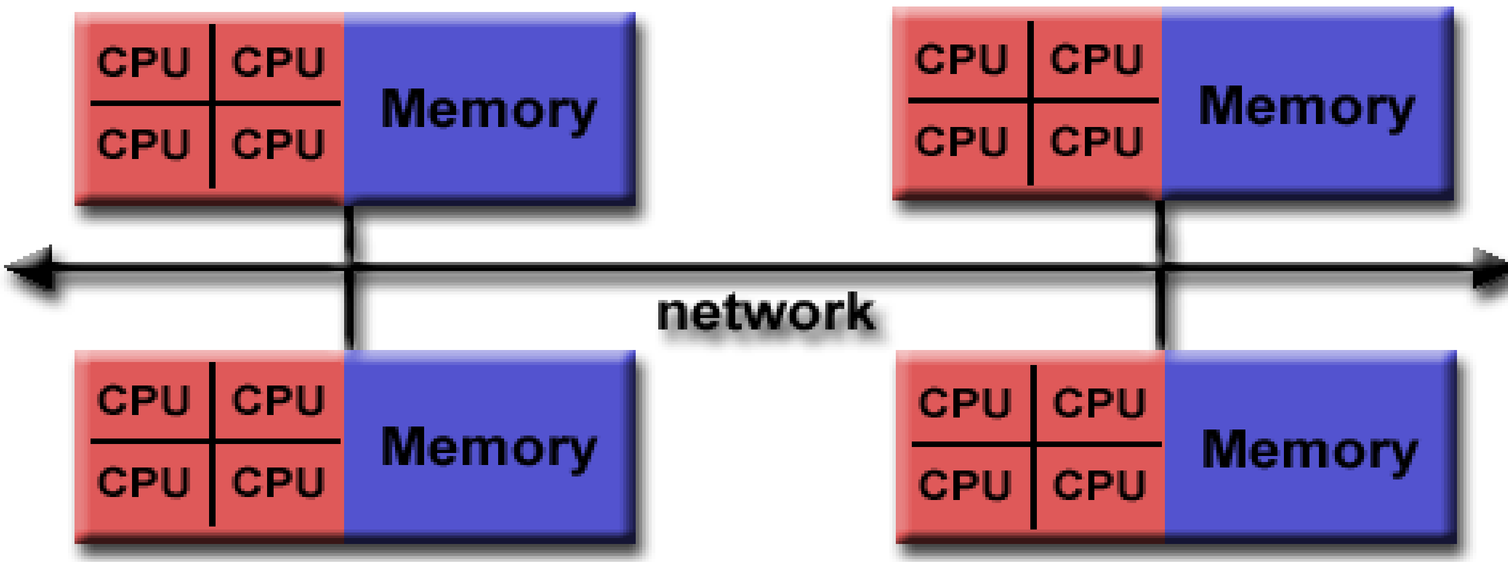
Each processor has its own local memory → it operates independently

The programmer needs to eplicitly define communication/synchronization



Hybrid Distributed-Shared Memory

- The fastest computer employ both shared and distributed memory architectures.
- Shared memory component → memory or graphics processing units (GPU)
- Move data from machine to machine



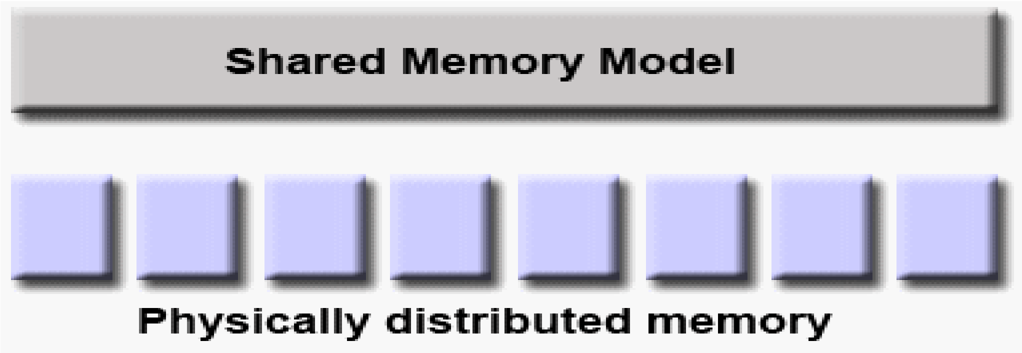
Parallel Programming Models

- Shared memory
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

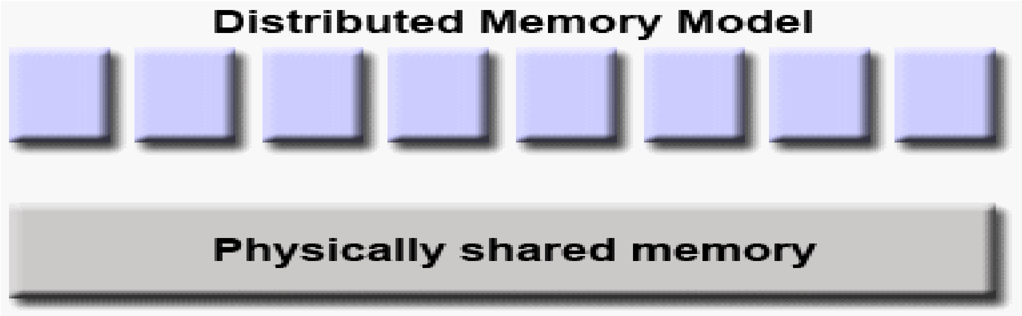
Parallel programming models exist as an abstraction above **hardware** and **memory** architectures.

Parallel Programming Models

- Shared memory model on a Distributed memory machine



- DISTRIBUTED memory model on a SHARED memory machine:



Shared Memory Model(without threads)

- Processes/tasks share a common address space.
- **Locks/semaphores** are used to **control access** to the shared memory.
- The simplest parallel programming model
- Data "ownership" is lacking → all processes have equal access to shared memory.
- Difficult to understand **data locality** → memory accesses, cache refreshes, bus traffic.

Threads Models

- It is a type of shared memory programming.
- A single "heavy weight" process can have multiple "light threads", concurrent execution paths.

For example:

1.- The main program **a.out** → main process

2.- **a.out** performs a serial work → creates threads

3.- Each thread has local data, but also shares resources with **a.out**.

4.- Threads can be seen as a subroutine within the main program.

5.- Threads communicate each other through **global memory**.

Implementations

- Programmers are responsible for determining the parallelism.
- Standards → POSIX Threads and OpenMP.

Posix Threads

- Specified by the IEEE POSIX standard (1995). C language only.
- Part of unix/linux operating systems.
- Library based.
- Communly referred to as Pthreads.
- Very explicit parallelism → requires significant programmer attention to detail.

Implementations

OpenMP

- Industry standard, jointly defined 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 -

Others

- Microsoft threads / Java, Python threads / CUDA threads for GPU.

Distributed Memory / Message Passing Model

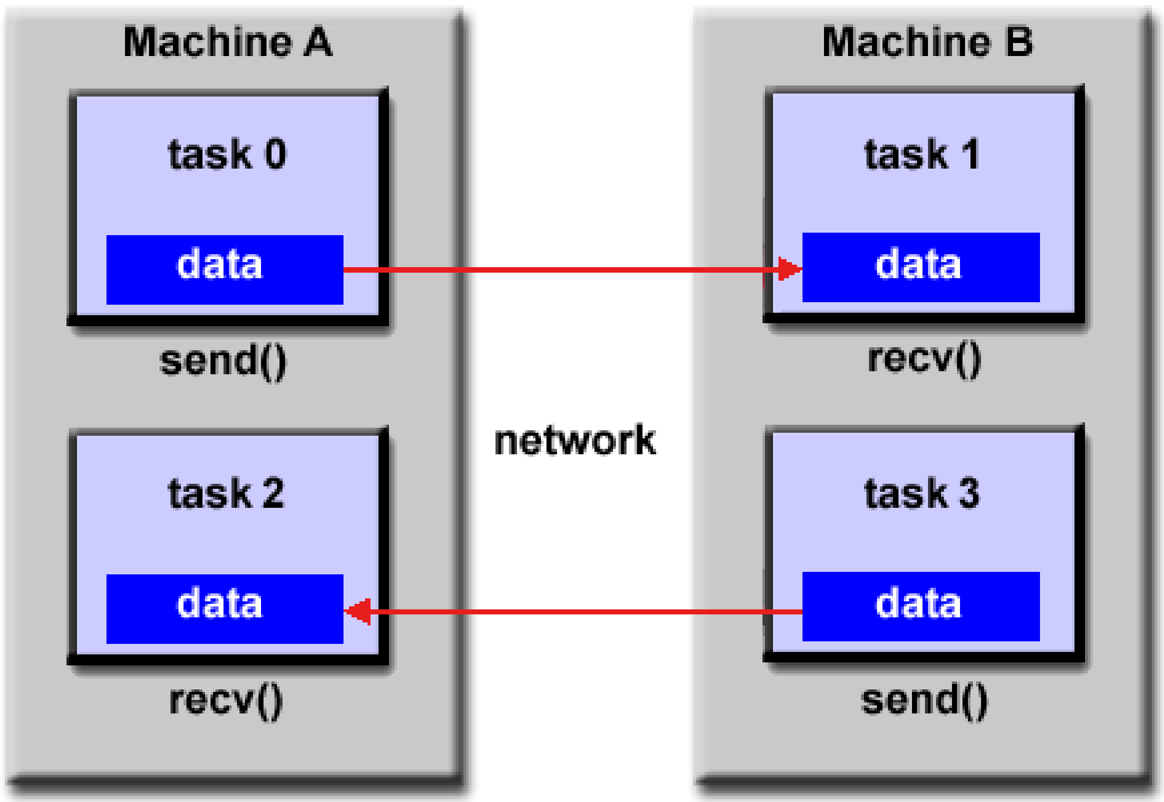
This model demonstrates the following characteristics:

- A set of tasks that use their own local memory during computation.

- Tasks exchange data through communications by sending/receiving messages.

- Data transfer usually requires cooperative operations to be performed by each process.

Distributed Memory / Message Passing Model



Distributed Memory Implementations

- Message passing implementations usually comprise

a library of subroutines.

- A variety of message passing libraries have been available since the 1980s.
- In 1992, The MPI Forum to standarize MP implementations.
- In 1994, part 1 of Message Passing Interface (MPI) was released.
- In 2012, MPI-3 was released.
- MPI implementations exist for all popular parallel computing platforms.