

Parallel Architecture **[High Performance (HPC) Architecture]**

Classes of Parallelism and Parallel Architecture

- **Parallelism at multiple level** is the driving force of computer design
 - Energy and cost being the constraints
- **Classes of parallelism in applications:**
 - Data-Level Parallelism (DLP)
 - Task-Level Parallelism (TLP)
- **Classes of architectural parallelism:**
 - Instruction-Level Parallelism (ILP)
 - Vector architectures/Graphic Processor Units (GPUs)
 - Thread-Level Parallelism
 - Request-Level Parallelism

Categories based on Flynn's Taxonomy

- Single instruction stream, single data stream (SISD):
 - A sequential computer which exploits instruction level parallelism - uniprocessors
- Single instruction stream, multiple data streams (SIMD):
 - A single instruction operates on multiple different data streams
- Multiple instruction streams, multiple data streams (MIMD)
 - Multiple autonomous processors simultaneously executing different instructions on different data
- Multiple instruction streams, single data stream (MISD)
 - Multiple instructions operate on one data stream
 - No commercial implementation

3

Single Instruction Stream, Single Data Stream (SISD)

- Uniprocessor
- Programmer thinks of this as sequential computer
- SISD exploits instruction-level parallelism (ILP)
- ILP exploits data level parallelism
 - With compiler help using pipelining
 - Overlapped execution of instructions, One instruction per clock cycle (1 IPC)
 - Using techniques like superscalar and speculative execution
 - Instruction-level parallelism, Multiple instructions per clock cycle (> 1 IPC)
- Reason for slowdown in uniprocessor:
 - Diminishing returns in exploiting ILP
 - Growing concern over power

4

Single Instruction Stream, Multiple Data Stream (SIMD)

- Same instruction is executed by multiple processors using different data streams
- SIMD exploits **data-level parallelism** by applying the same operation to multiple items of data in parallel
- Data parallelism for
 - Matrix-oriented scientific computing
 - Media-oriented image and sound processors
- Each processor has its own data memory, but there is a single instruction memory and control processor
- **Types of SIMD architecture:**
 - Vector (array) architectures
 - Graphics processor units (GPUs)

5

Multiple Instruction Stream, Multiple Data Stream (MIMD)

- **Multiprocessor:**
 - Computers consists of **tightly coupled processors**
 - Coordination and usage are controlled by a single operating system
 - Share memory through a shared address space
- **Each processor fetches its own instructions and operate on its own data**
- MIMD targets **task-level parallelism** as well as **data-level parallelism**
- **Types of MIMD architectures:**
 - **Tightly-coupled MIMD**
 - All the processing elements have shared memory model.
 - **Loosely-coupled MIMD**
 - All the processing elements have separate memory model

6

Tightly Coupled MIMD Architecture

- Exploits thread-level parallelism
- Thread is a light-weight process
- Threads of a process share the same virtual address space
- Multiple cooperating threads operate in parallel
- Parallel processing: Tightly coupled set of threads collaborating on a single task
- This architecture ranges from dual processor to dozens of processors
 - Communicate and coordinate through the sharing memory
- Multiprocessors include in
 - Computers consists of single chip with multiple cores (multicore)
 - Computers consisting of several chips, each may be multicore design

7

Loosely Coupled MIMD Architecture

- Exploits request-level parallelism
 - Exploits parallelism among largely decoupled tasks specified by programmer or the operating system
- Many independent tasks processed in parallel with little need for communication and synchronization
- Examples: Clusters and Warehouse-scale computers (Cloud computers)

8

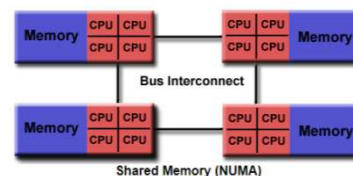
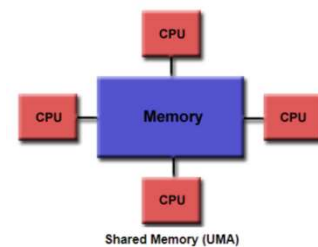
Multiprocessors: Parallel Architecture

- Shared memory architecture
- Distributed memory architecture
- Hybrid architecture

9

Shared Memory Architecture

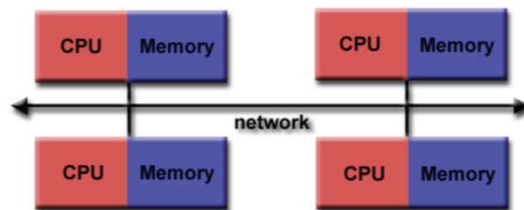
- **Centralised multiprocessors:**
 - Multiple processors attached to a bus
 - all the processors share the same primary memory
 - **Uniform Memory Access (UMA)** or **Symmetric Multiprocessor (SMP)**
- **Distributed multiprocessors:**
 - Processors are distributed and connected by bus interconnect
 - Distributed collection of memories forms one logical address space
 - Same address on different processors refers to the same memory location
 - **Nonuniform Memory Access (NUMA)**

Support of **Cache Coherence**

10

Distributed Memory Architecture

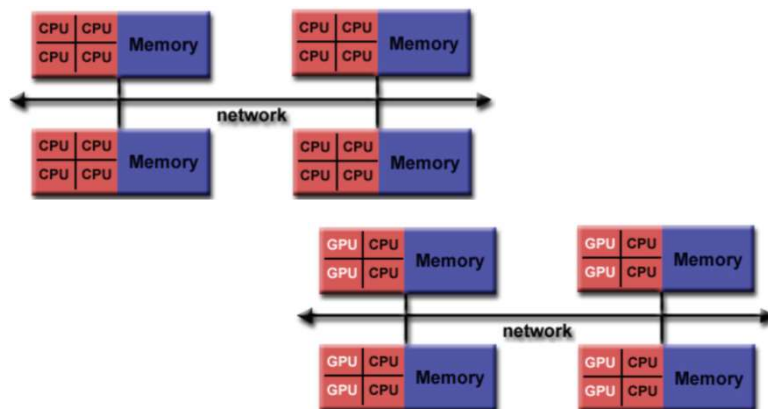
- **Distributed multiprocessors:**
 - Each processor has its local memory with the address space available only for this processor
 - Processors can exchange data through the **interconnection network** by means of communication by **passing messages**
 - Required a communication network to connect inter-processor memory
 - Each processor operated independently
 - No concept of cache coherency



11

Hybrid Memory Architecture

- **Distributed multiprocessors:**
 - Employ both shared and distributed memory architectures
 - Shared memory could be **shared memory machine** or with **GPUs**



12

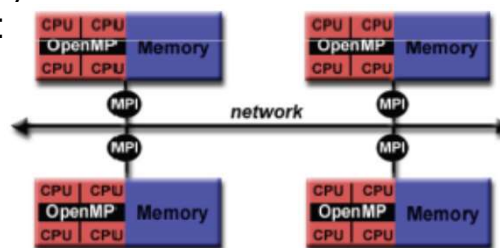
Parallel Programming Models

- **Shared memory:** All tasks have visibility to a global address space
 - **OpenMP**
- **Distributed memory:** All tasks have their own private address space and communicate via messages
 - **Message Passing Interface (MPI)**

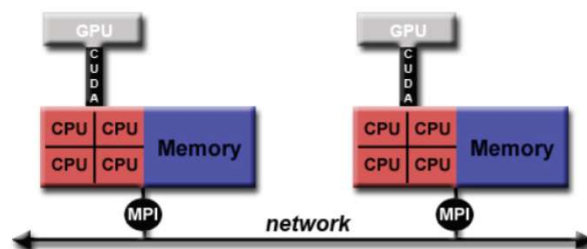
13

Parallel Programming Models

- **Hybrid memory:** Combination of shared and distributed memory
 - **OpenMP + MPI**



- **CUDA + MPI**

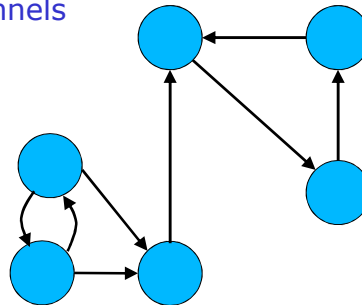


14

Parallel Algorithm Design

Task/Channel Model

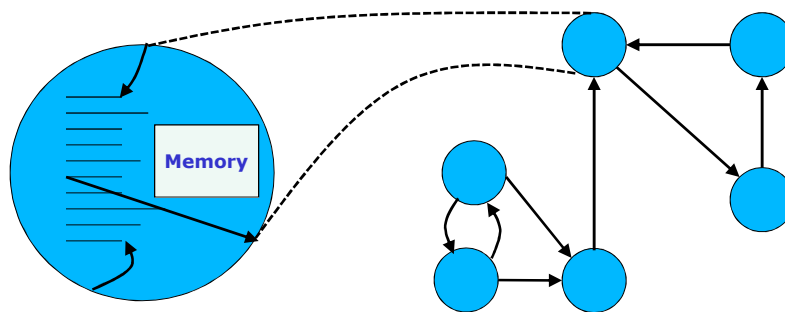
- Described by Ian Foster [1]
- Facilitates the development of **efficient parallel algorithms**, particularly distributed memory architecture
- It represents a parallel computation as a **set of tasks** that may interact with each other by **sending messages through channels**



[1] Ian Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

Task/Channel Model

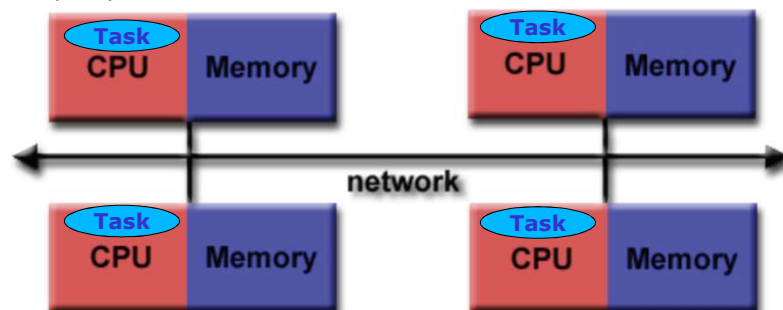
- **Task:** A task is a program, its local memory and a collection of I/O ports
 - The local memory contains the program's instruction and its private data
 - A task can send local data values to other tasks via output ports
 - A task can receive local data values from other tasks via input ports



17

Task/Channel Model

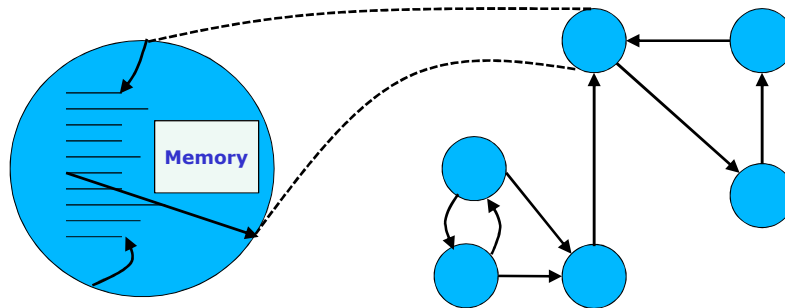
- **Task:** A task is a program, its local memory and a collection of I/O ports
 - The local memory contains the program's instruction and its private data
 - A task can send local data values to other tasks via output ports
 - A task can receive local data values from other tasks via input ports



18

Task/Channel Model

- **Task:** A task is a program, its local memory and a collection of I/O ports



- **Channel:** Message queue that connects one task's output port with another task's input port

19

Foster's Design Methodology

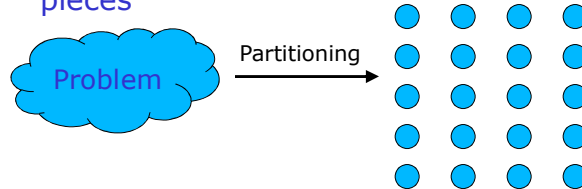
- Four-step process for designing parallel algorithm [1]
 - Partitioning
 - Communication
 - Agglomeration
 - Mapping

[1] Ian Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

20

Partitioning

- Process of dividing the computation and data into pieces

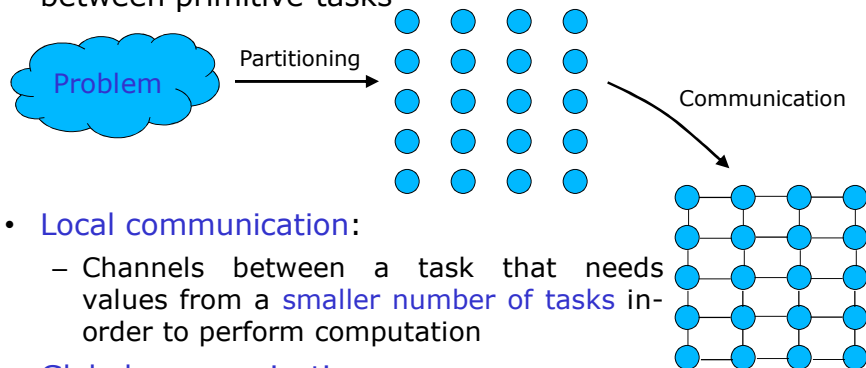


- **Domain decomposition (data-centric):**
 - First divide the data into pieces and then determine how to associate computations with the data
 - Focus on **largest or frequently accessed data structure**
- **Functional decomposition (computation-centric):**
 - First divide the computation into pieces and then determine how to associate data item into individual computations
 - Collection of tasks and running concurrently
- **Primitive task**

21

Communication

- Process of **determining the communication pattern** between primitive tasks

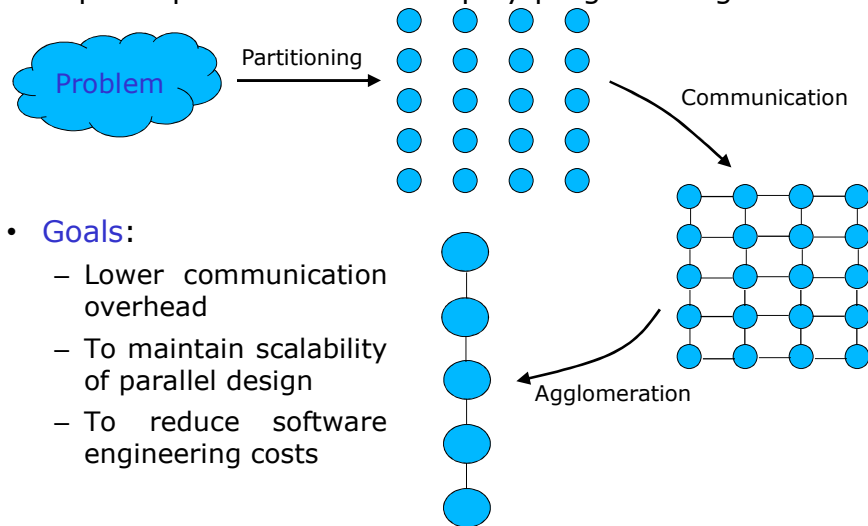


- **Local communication:**
 - Channels between a task that needs values from a **smaller number of tasks** in-order to perform computation
- **Global communication:**
 - Channels between a task and a **significant number of primitive tasks** in-order to perform computation

22

Agglomeration

- Process of **grouping tasks into larger tasks** in-order to improve performance or simplify programming

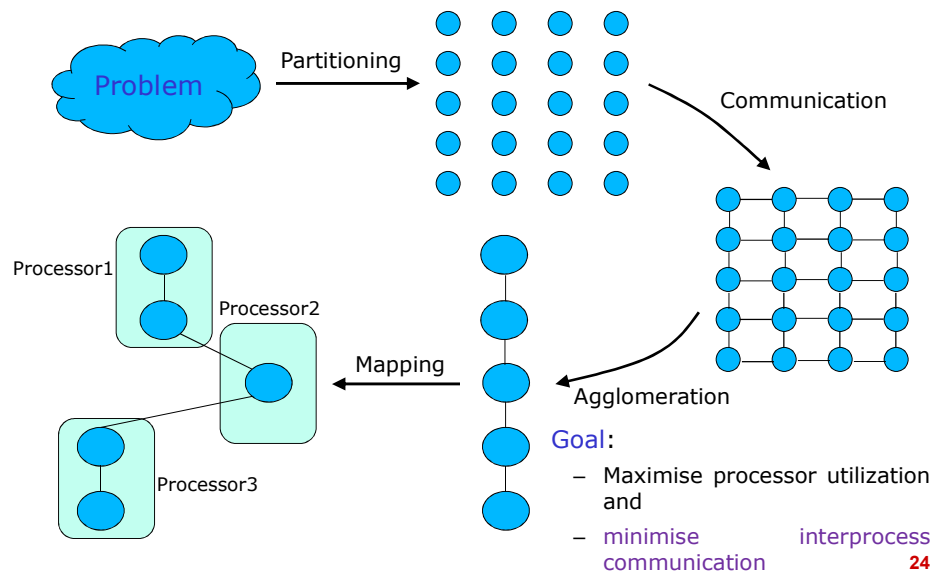


- Goals:**
 - Lower communication overhead
 - To maintain scalability of parallel design
 - To reduce software engineering costs

23

Mapping

- Process of **assigning tasks to processors**



Goal:

- Maximise processor utilization and
- minimise interprocess communication

24

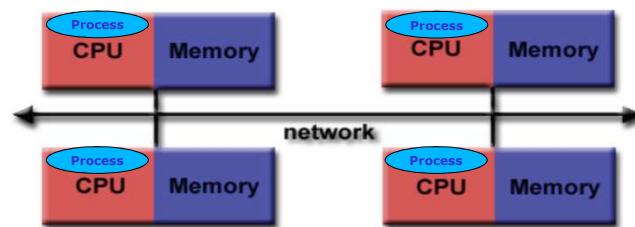
Message Passing Interface (MPI)

Parallel Programming Approaches

- [Approach1](#): Parallelising compilers and high-level parallel programming languages
- [Approach2](#): Augmenting an existing sequential language with low level constructs expressed by functional calls or compiler directives
 - Parallel programming in C, C++ and Fortran with MPI and OpenMP

Message-Passing Model

- Collection of processors, each with its own memory
 - Processor has direct access only to the instruction and data stored in local memory
- Interconnection network supports message passing between the processors
- User specifies the number of concurrent processes when the program begins
- Every process executes the same program
- Each process has unique ID number



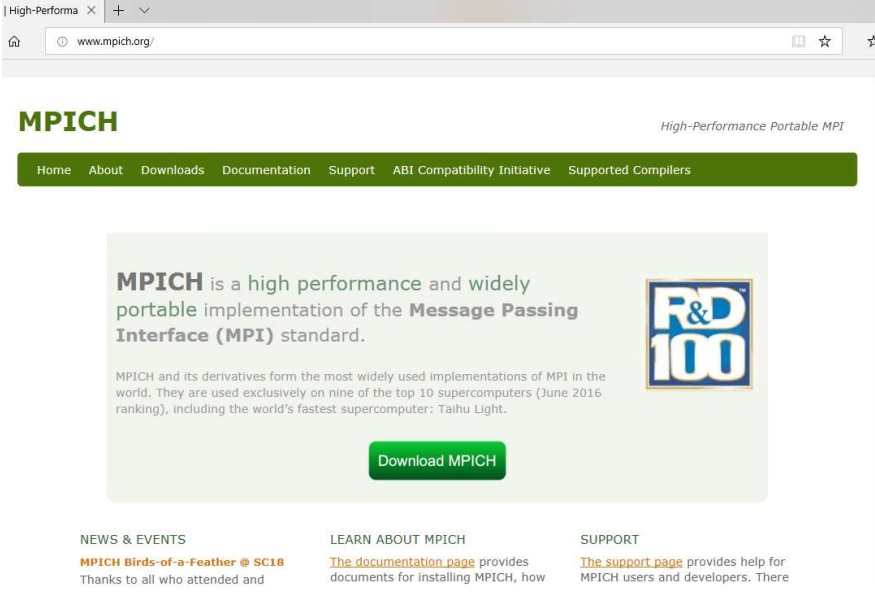
27

MPI Standards and Goals

- MPI-1 standard defined in 1994
- Current standard is MPI-3
- Standard specifies the **names**, **calling sequence** and **subroutines and functions** called from C, C++ and Fortran
- **Goals:**
 - To Provide source code portability
 - To allow efficient implementations across a range of architectures
 - To support heterogeneous parallel architecture

28

Installing MPICH3 in a Single Machine



The screenshot shows the MPICH website with the title "Installing MPICH3 in a Single Machine". The website header includes the MPICH logo and the tagline "High-Performance Portable MPI". A navigation bar contains links: Home, About, Downloads, Documentation, Support, ABI Compatibility Initiative, and Supported Compilers. The main content area features a large green box with the text: "MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard." Below this, it states: "MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (June 2016 ranking), including the world's fastest supercomputer: Taihu Light." To the right of this text is an "F&D 100" logo. A green button labeled "Download MPICH" is positioned below the text. At the bottom of the page, there are three sections: "NEWS & EVENTS" with a link to "MPICH Birds-of-a-Feather @ SC18", "LEARN ABOUT MPICH" with a link to "The documentation page", and "SUPPORT" with a link to "The support page".

29

Installing MPICH3 in a Single Machine

```
>> tar -xzf mpich-3.3.tar.gz

>> cd mpich-3.3

>> ./configure --disable-fortran

>> sudo make install

>> mpiexec --version
```

<https://www.mpi-forum.org/docs/>

30

MPI Programming

- The program begins with pre-processor directive to include header file for MPI

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]){

}
```

- `argc` and `argv` are needed to pass to the function that initialises MPI

- Some of the **MPI function calls**:

- `MPI_Init`: To initialize MPI
- `MPI_Comm_rank`: To determine process's ID number
- `MPI_Comm_size`: To find the number of processes
- `MPI_Finalize`: To shut down the MPI

31

Function `MPI_Init`

- The first MPI function call made by every MPI process
- It allows the system to do any startup needed to handle further calls to the MPI library

```
MPI_Init (&argc, &argv);
```

Function `MPI_Finalize`

- The last MPI function call made by every MPI process i.e. called after a process has completed all its MPI library calls
- It allows the system to free up resources such as memory that has been allocated to MPI

```
MPI_Finalize ();
```

32

MPI Program Compile and Execution

- Compiling:
 - `mpicc -o example.out example.c`
- Execution:
 - `mpiexec -n <num of processes> ./example.out`

33

MPI Overview

- MPI initialization and finalization
- MPI communication, size and rank
- MPI datatypes
- Point-to-point communication
 - MPI send and receive
 - Blocking and deadlock
- Collective communication
 - Barrier
 - Broadcast
 - Reduction
 - Gather
 - Scatter
- Time functions

34

Text Books

- <https://www.mpi-forum.org/docs/>
- M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

35