Introduction to High Performance Computing and Parallel Computing

Deepika H.V

System Software Development Group

C-DAC, Bengaluru

deepikahv@cdac.in

Contents

- High Performance Computing
- Applications
- Machine Architectures
  - Parallel Architectures
  - Memory architectures
- Evolution of Supercomputers
  - World Top 5
  - India Top 5
- Parallel Computing
  - Classifications
  - Program Design
  - Parallel Programming Paradigms

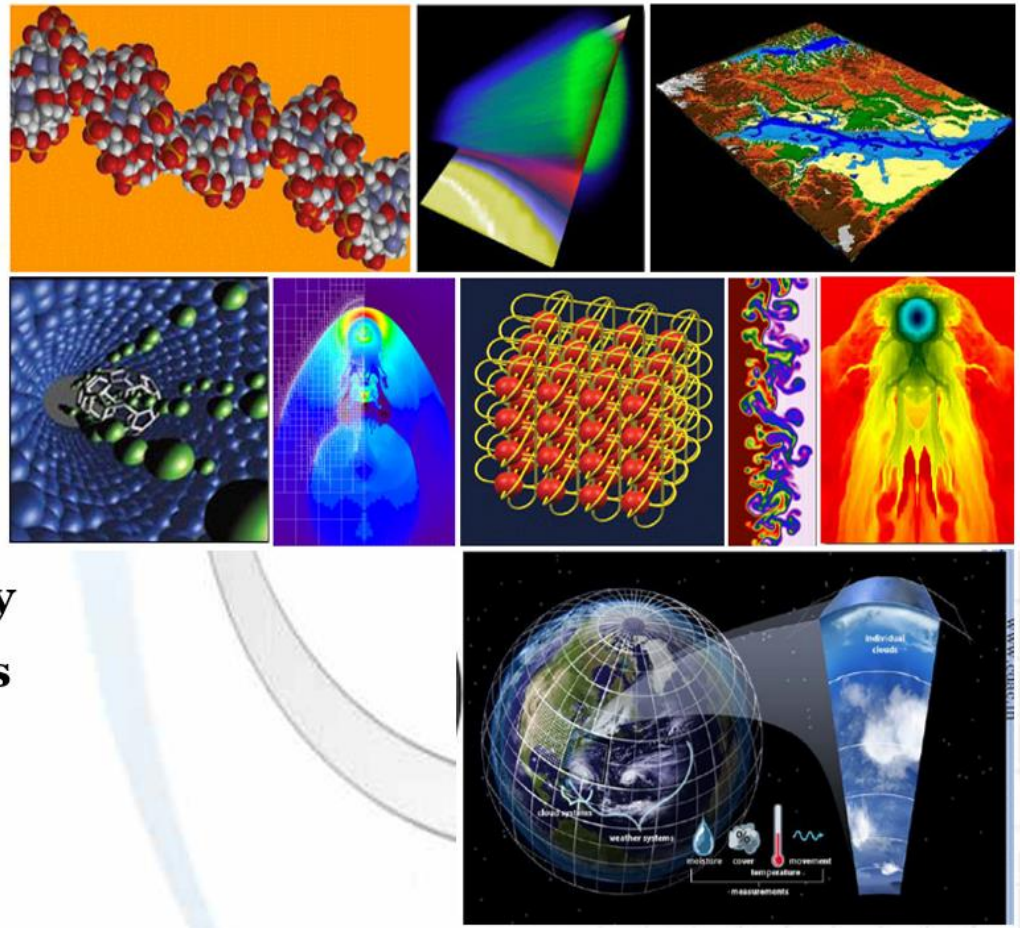
High Performance Computing

What is HPC?

- Use of **supercomputers** and **parallel computing** techniques to solve complex **computational problems**.
- It is the practice of aggregating computing power in a way that delivers **superior performance** as opposed to a typical desktop computer.
- HPC architecture is influenced by the lowest-level technologies and circuit design, and how they can be most effectively employed in supercomputers.

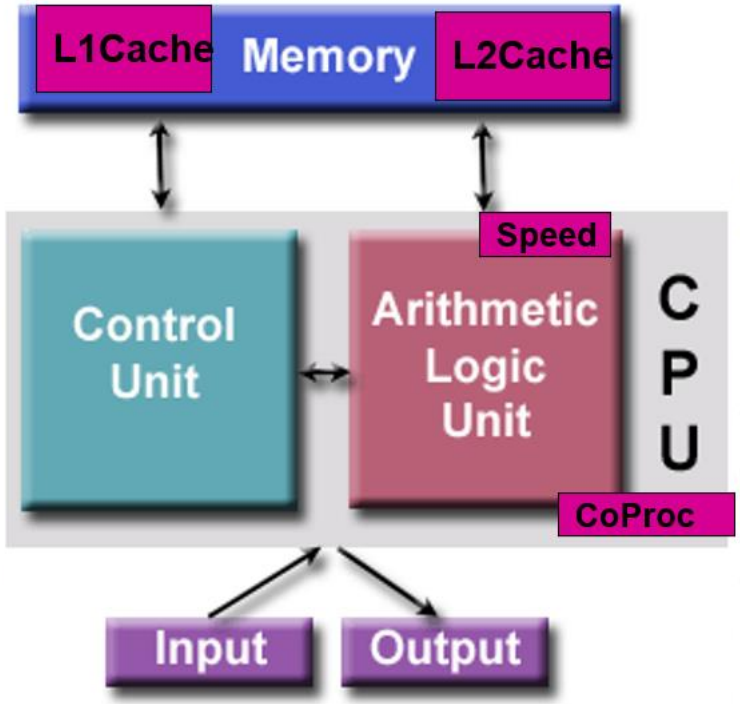
HPC Domains / Applications

- Prediction of weather, climate, global changes
- Challenges in materials sciences
- Semiconductor design
- Structural biology
- Design of drugs
- Human genome
- Astronomy
- Challenges in transportation
- Vehicle dynamics
- Nuclear fusion
- Enhanced oil and gas recovery
- Computational ocean sciences
- Speech
- Vision
- Visualization and graphics



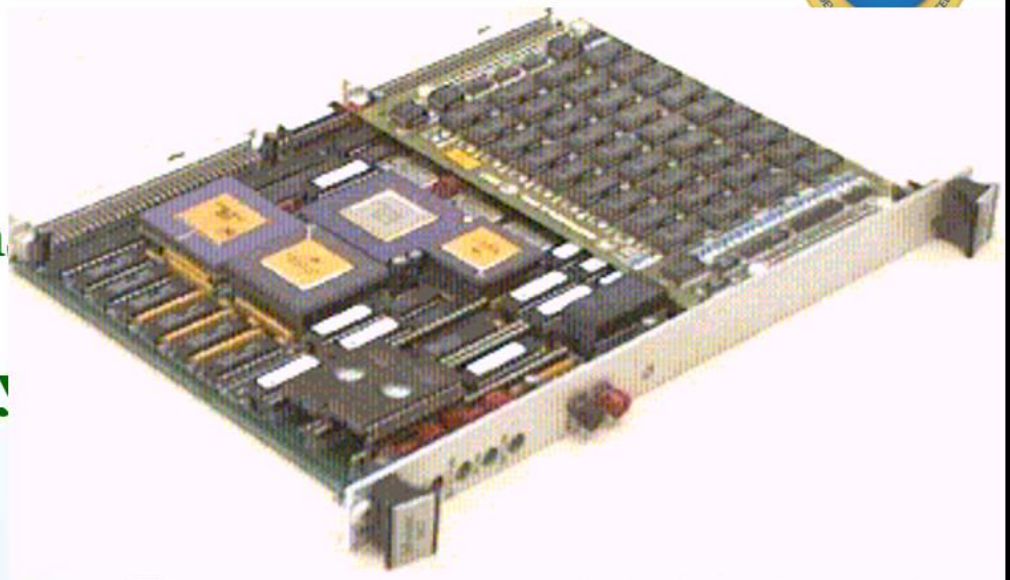
Evolution of Architectures

Von Neumann Machines

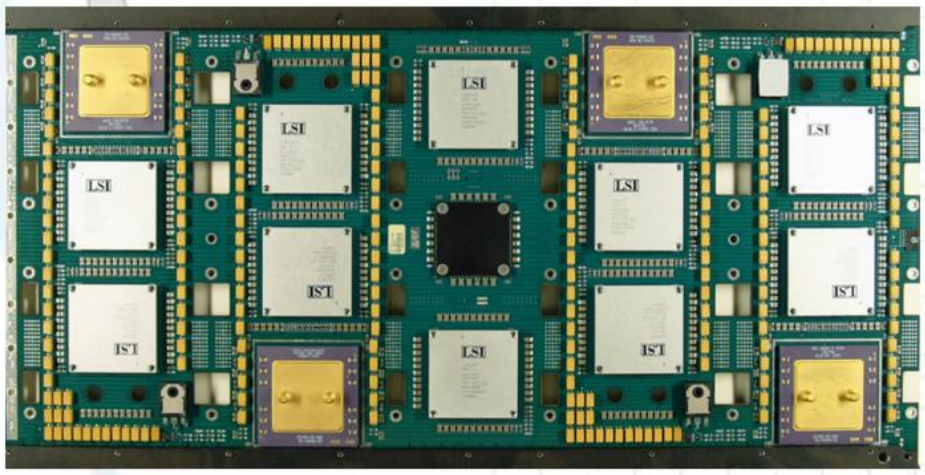


Parallel Architectures

- **Vector processors**
  - able to run mathematical operation on multiple data elements simultaneously



- **Superscalar processors**
  - Instruction level parallelism with a processor
  - Intel 960, SunSparc, MIPS R, and AMD Am29000



Multicore Machines

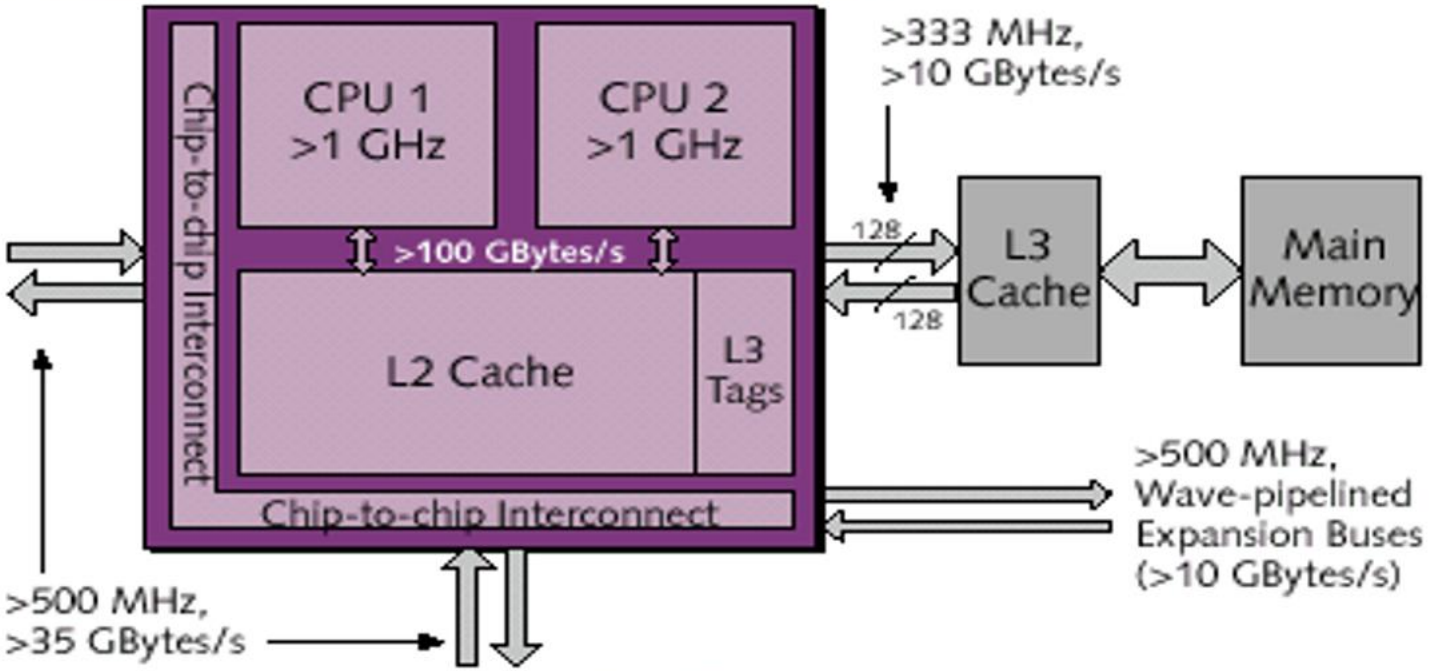
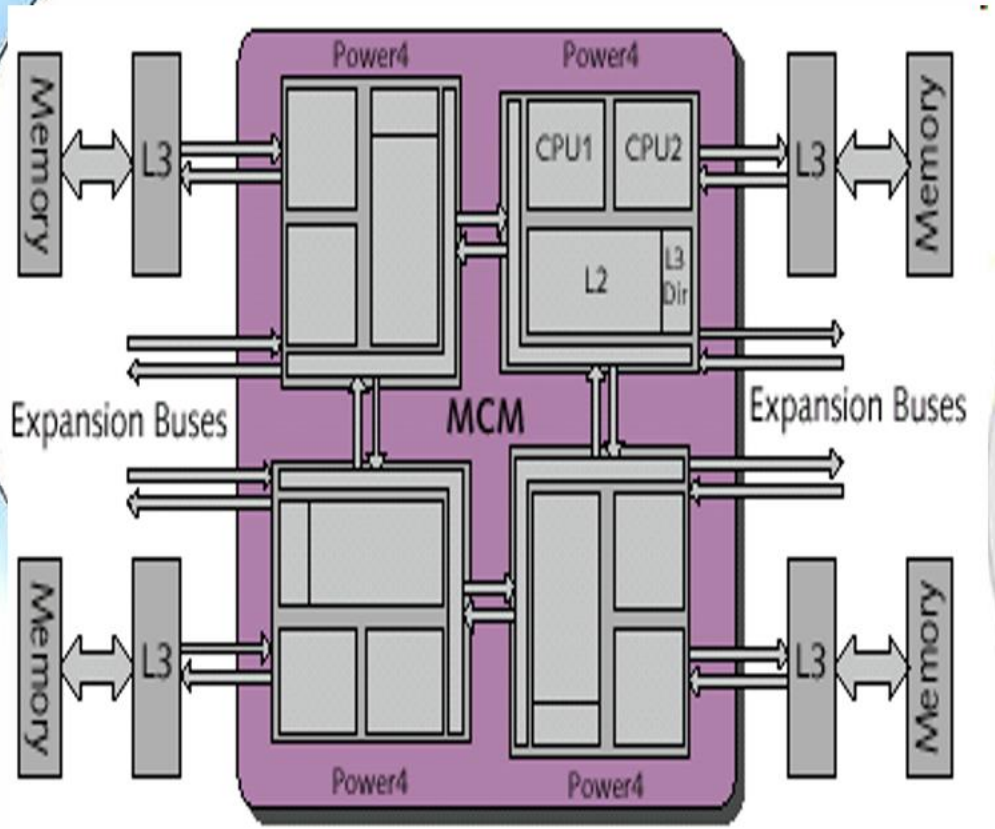
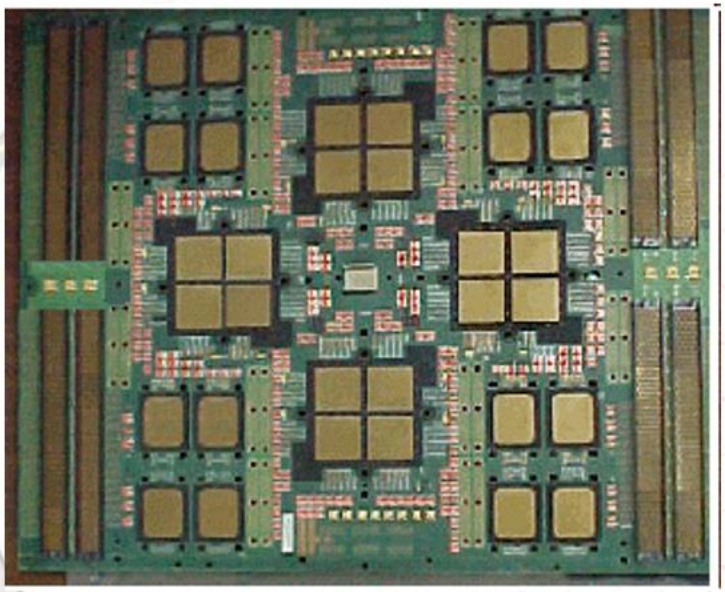


Fig: A 2-Processor Chip

Fig: An Intel Core 2 Duo E6750 dual-core processor

Multi Chip Module (MCM)

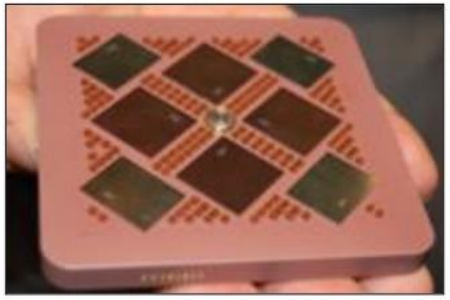


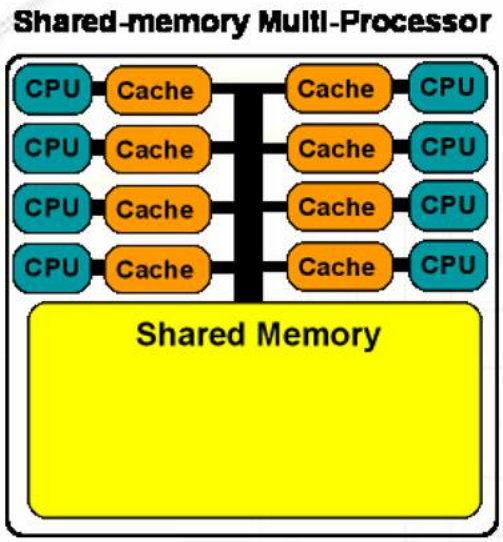


32-way System with 4 MCM and L3 cache

Symmetric Multiprocessor (SMP)

- two or more identical processors are connected to a single shared main memory





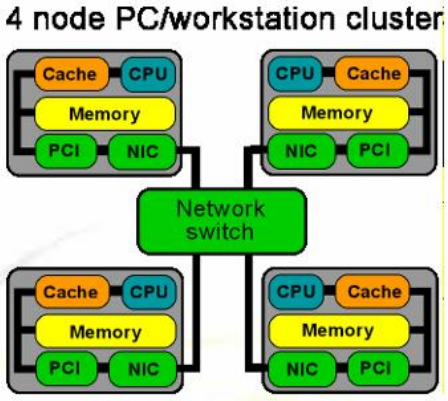
Supercomputers

- tightly coupled computers that work together closely as though they are a single computer

Parallel Architectures

Distributed Computing

- parts of a program are run at the same time, on separate computers communicating over a network

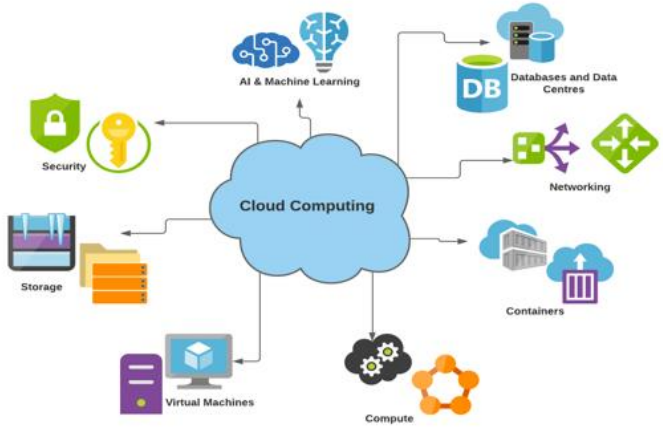


Grid Computing

- Aggregation, sharing of distributed heterogeneous systems

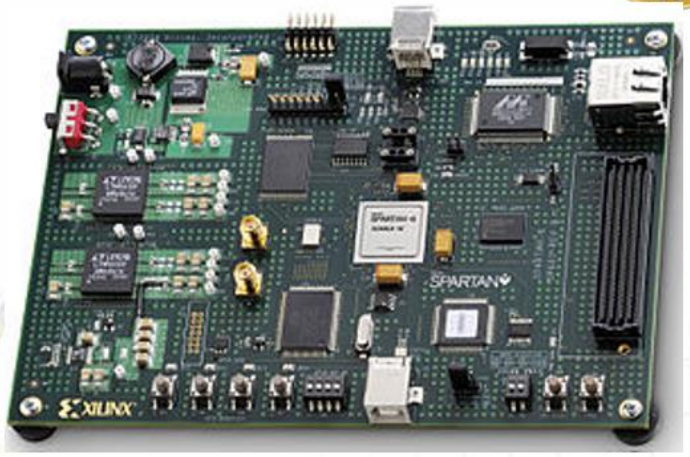
Cloud Computing

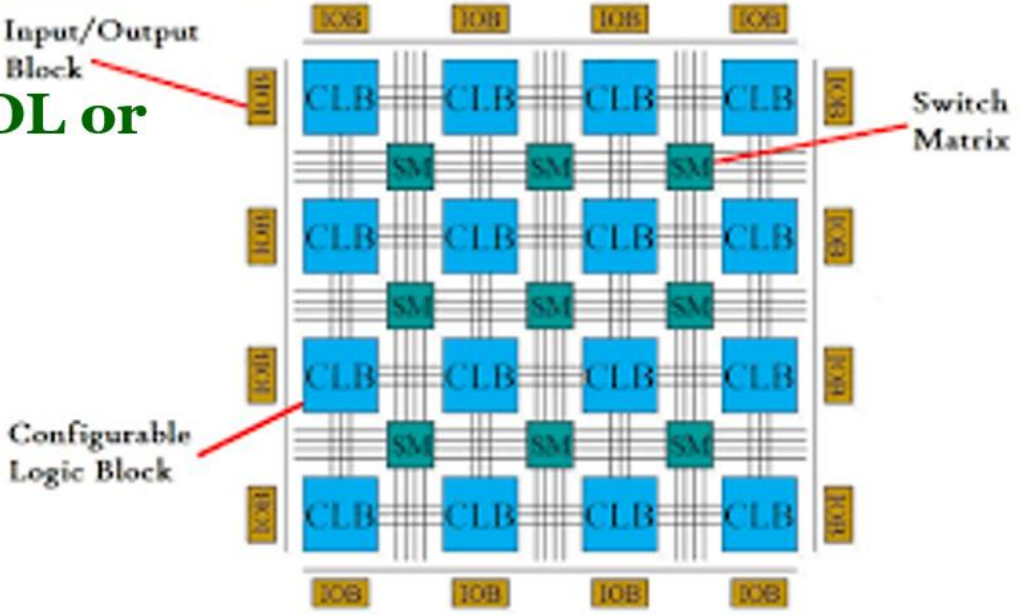
- on-demand available service – IaaS PaaS, SaaS
- pay-as-you-use over the Interne

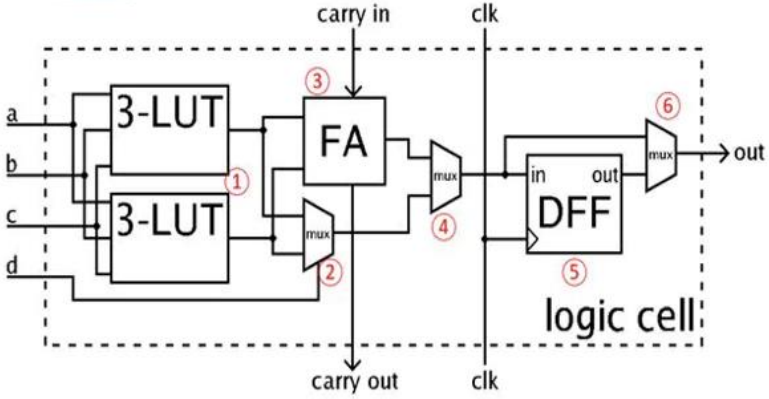


Accelerators

- **Field-Programmable Gate Arrays**
  - computer chip that can rewire itself for given task
  - Intel (Altera) and AMD (Xilinx)
  - can be programmed with hardware description languages such as VHDL or Verilog

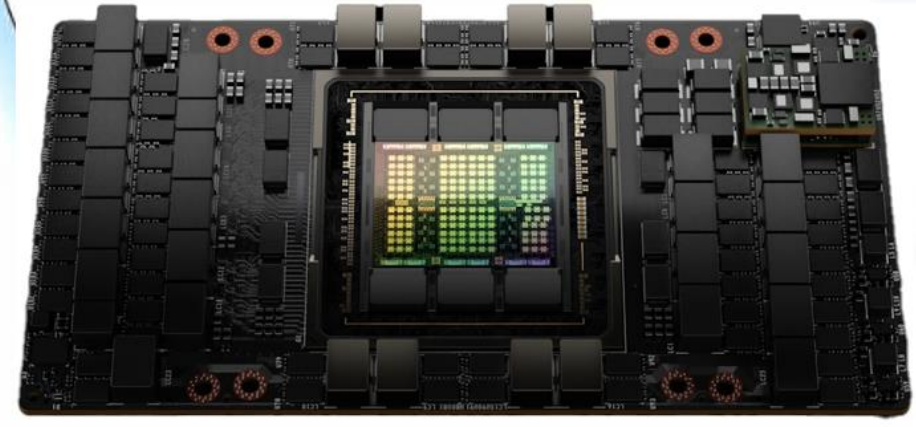


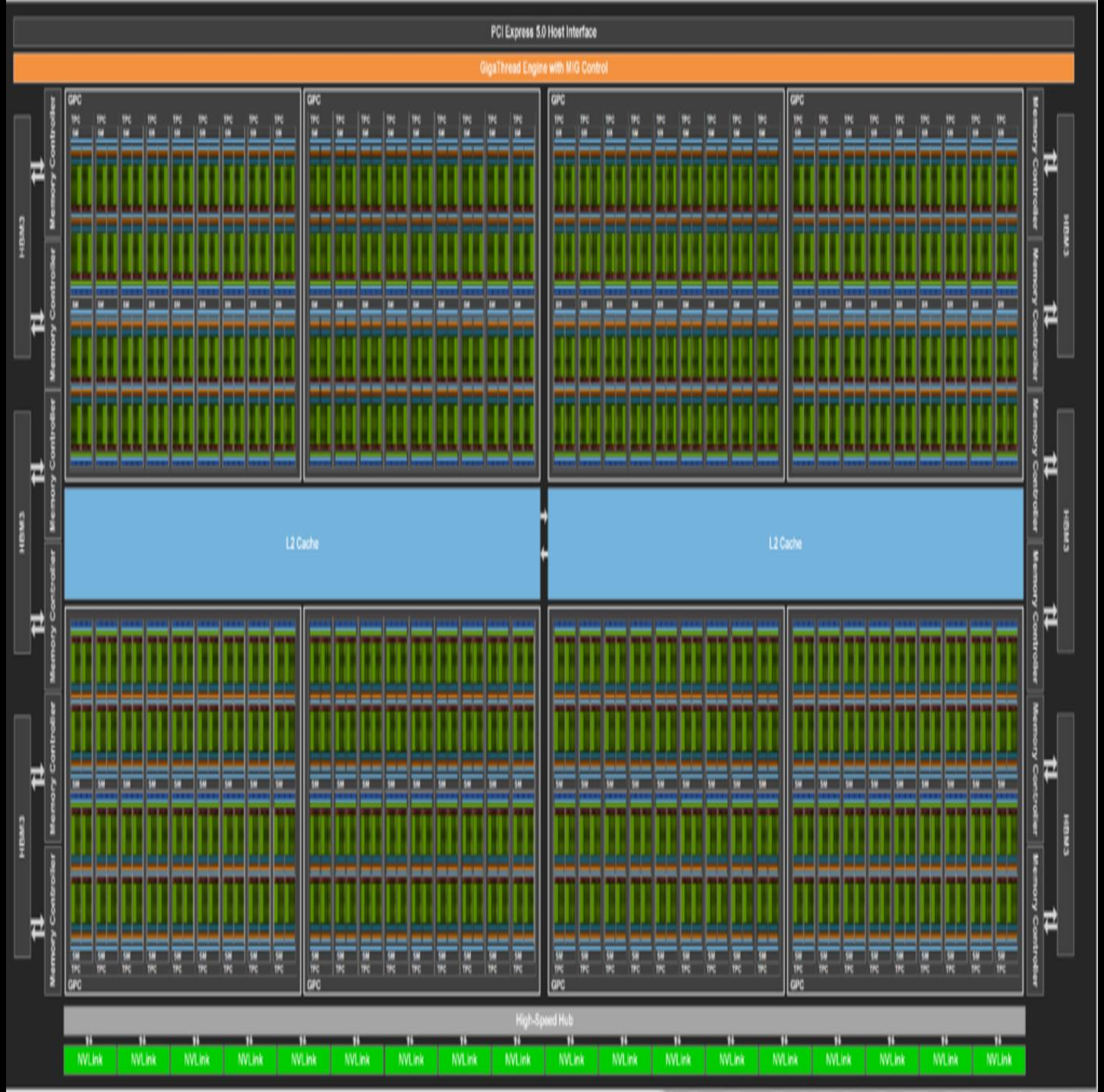




General Purpose GPU

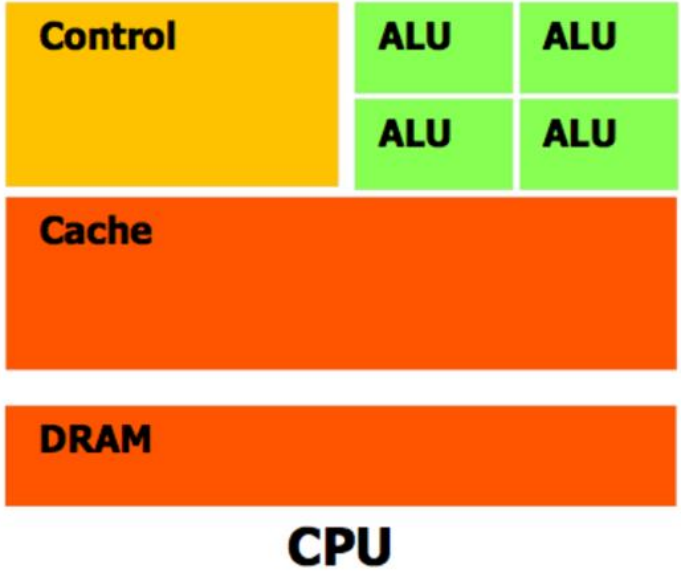
- General-purpose computing on graphics processing units (GPGPU)
- NVIDIA, Intel and AMD
- CUDA/OpenCL programming environment

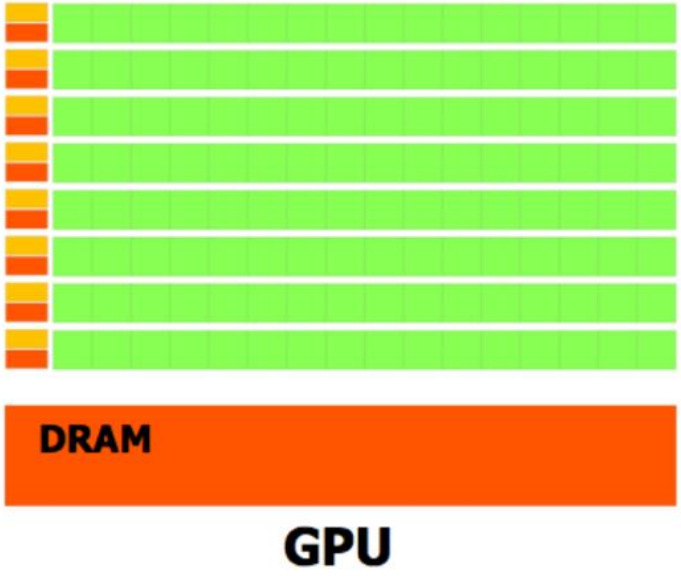



PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM3
L2 Cache
High-Speed Hub
NVLink


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

CPU vs GPU

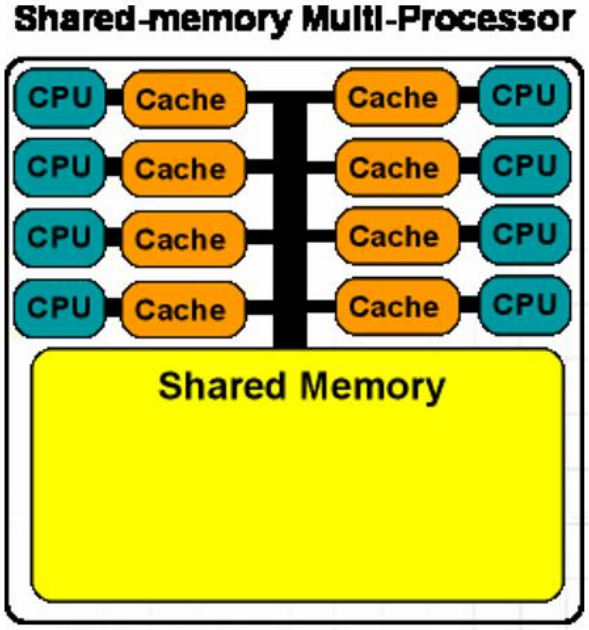




Memory Architecture

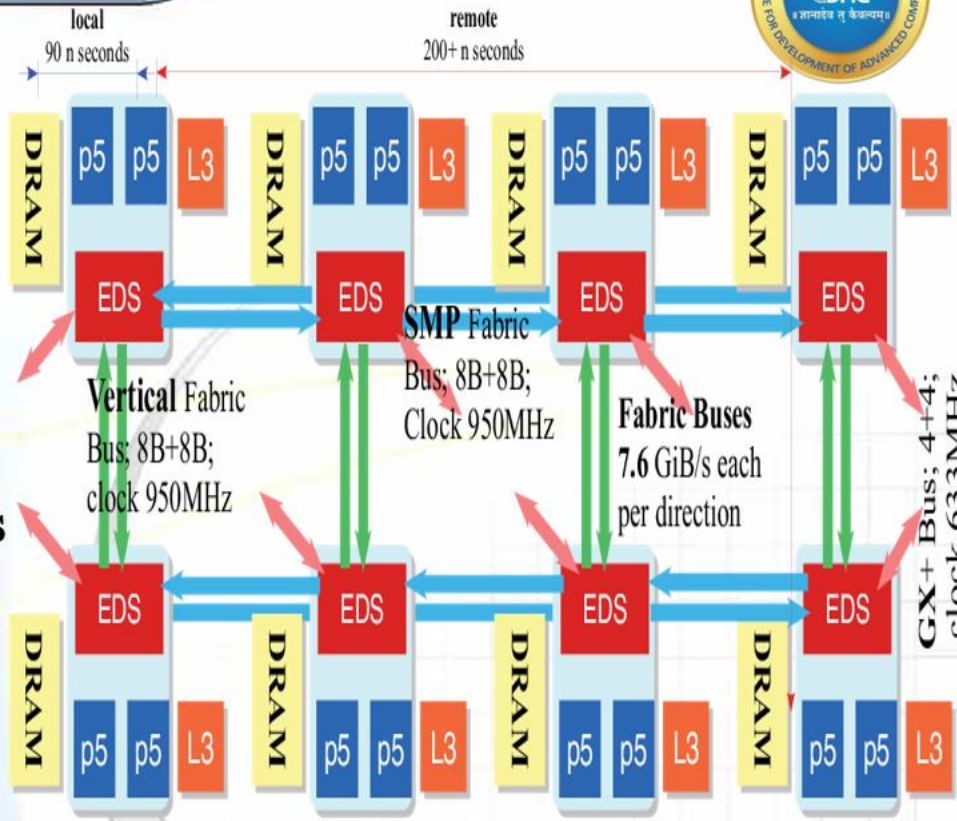
Shared Memory - UMA

- Uniform Memory Access (UMA)
- Example - SMP
- Identical processors
- Equal access & access times to memory
- Sometimes called Cache Coherent UMA (CC-UMA). Cache coherency is accomplished at the hardware level.
- Contention - as more CPUs are added, competition for access to the bus leads to a decline in performance.
- Thus, scalability ~ 32 processors.



Shared Memory - NUMA

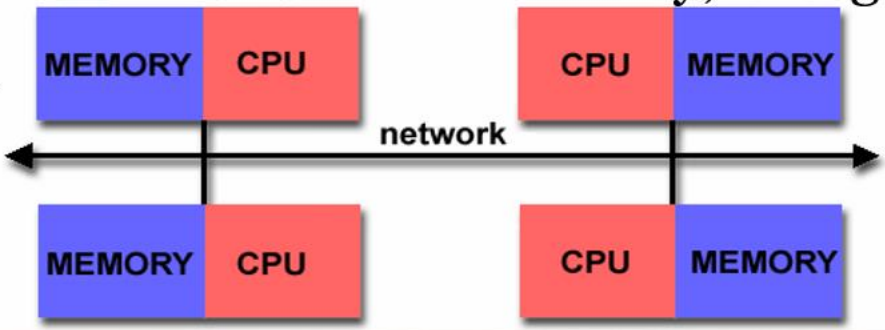
- Non-Uniform Memory Access
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained - CC-NUMA



- Designed to overcome scalability limits of SMPs.
- Can support up to 1024 processors.
- Processors directly attached to a memory module experience lower latency than those attached to "remote" memory modules.

Distributed Memory Architecture

- Processors have their **own local memory**. So operates independently.
- **No** concept of **global address space** across all processors.
- Changes in local memory have no effect on memory of other processors. Hence, **cache coherency** does **not apply**.
- **Data access** between processors is **defined by programmer** ; explicitly define how and when data is communicated. Synchronization between tasks is also programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.



Evolution of Super Computers

Supercomputer

- 1961 -- IBM 7030, 1.2 MIPS
- 1964 -- CDC 6600 , 6 MFLOPS
- 1970s – Cray-1 , a vector processor , 160 MFLOPS
- 1985 – Cray-2, 4 processor liquid cooling , 1.9 GFLOPS
- 1996 – ASCI Red , Intel, Sandia National Labs, 1.3 TFLOPS

Listing of TOP500 from 1993 is available on top500.org

Evolution of Supercomputers

Mflop/s	10^6 flop/sec
Gflop/s	10^9 flop/sec
Tflop/s	10^{12} flop/sec
Pflop/s	10^{15} flop/sec

2015

2013

PARAM YUVA II

2008

PARAM YUVA

2005

GARUDA (NKN)

2002-03

PARAM PADMA

1998

PARAM 10000

Platform for User community to interact/ collaborate

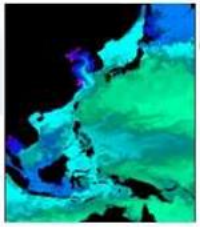
1994

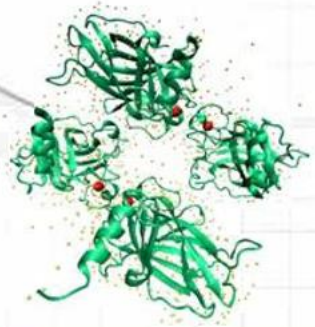
PARAM 9000

1991

PARAM 8000

Technology Denial





World Top5 Supercomputers

Rank	Site	System	Cores	Rmax (PFlop)	RPeak (PFlop)
1	Frontier , DOE/SC/Oak Ridge National Laboratory United States	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE	8,699,904	1,194.00	1,679.82
2	Aurora, DOE/SC/Argonne National Laboratory United States	HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel	4,742,808	585.34	1,059.33
3	Eagle, Microsoft Azure United States	Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR	1,123,200	561.20	846.84
4.	A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu	A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu	7,630,848	442.01	537.21
5.	LUMI, EuroHPC/CSC Finland	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE	2,220,288	309.10	428.70

Indian Top5 Supercomputers

Rank	Site	System	Cores/ nodes	Rmax (TFlop)	RPeak (TFlop)
1 [90]	Airawat-PSAI, C-DAC Pune	NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHZ, NVIDIA A100, INFINIBAND HDR2	81344	8500	13176
2 [163]	PARAM Siddhi-AI, C-DAC,Pune	NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband (OEM:ATOS under NSM initiative, Bidder)	41664/ 236	4619	5267.14
3 [201]	Pratyush, Indian Institute of Tropical Meteorology, Pune	Cray XC-40 class system with 3315 CPU-only (Intel Xeon Broadwell E5-2695 v4 CPU) nodes with Cray Linux environment as OS, and connected by Cray Aries interconnect	119232/ - -	3763.9	4006.19
4. [354]	Mihir, NCMRWF, Noida	Intel Xeon Broadwell E5-2695 v4 CPU with Cray Linux environment connected by Cray Aries interconnect OEM:Cray, Bidder:Cray	83592/ 1152	2570.4	2808.7
5.	PARAM Pravega, IISc, Bangalore	Intel Xeon Cascade Lake processors,NVIDIA Tesla V100 with NVLink, Mellanox HDR interconnect. OEM:Atos, Bidder:Atos	29952/ 624	1702	2565

What is Parallel Computing?

Serial Computing

If 1 person takes 15 hours to complete the job; then

How long will it take if 15 people work together

PARALLEL Processin or Parallelism!

- Saves total time taken
- Solve larger problems

Parallelism is a necessity of today's computing!!

Terminologies

Parallel Processing

- Processing multiple tasks simultaneously on multiple processors is called parallel processing.

Parallel Programming

- Software methodology used to implement parallel processing.

Parallel Computing

- Term that encompasses all the technologies used in running multiple tasks simultaneously on multiple processors

Parallel Programming

Classification

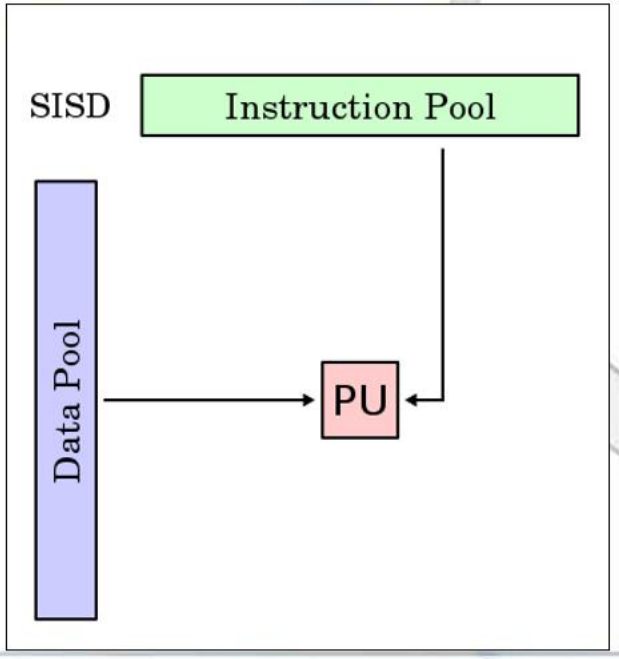
Flynn's Taxonomy

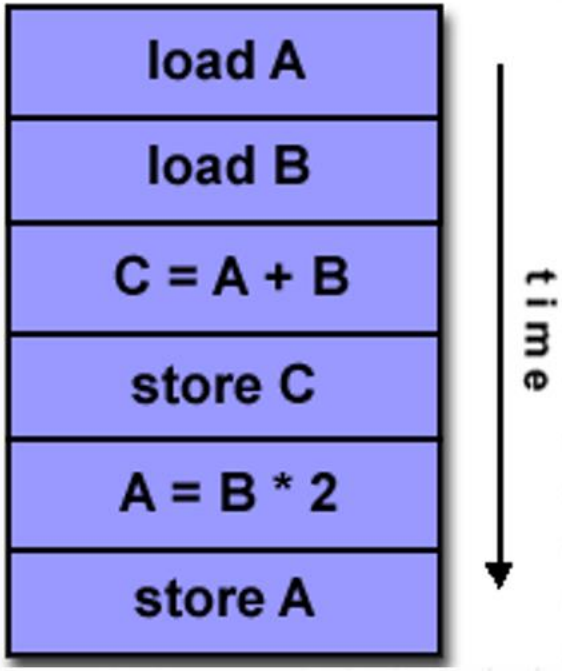
Instruction Streams \ Data Streams	
S I S D Single Instruction, Single Data	S I M D Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	M I M D Multiple Instruction, Multiple Data

1. SISD

UNIVAC

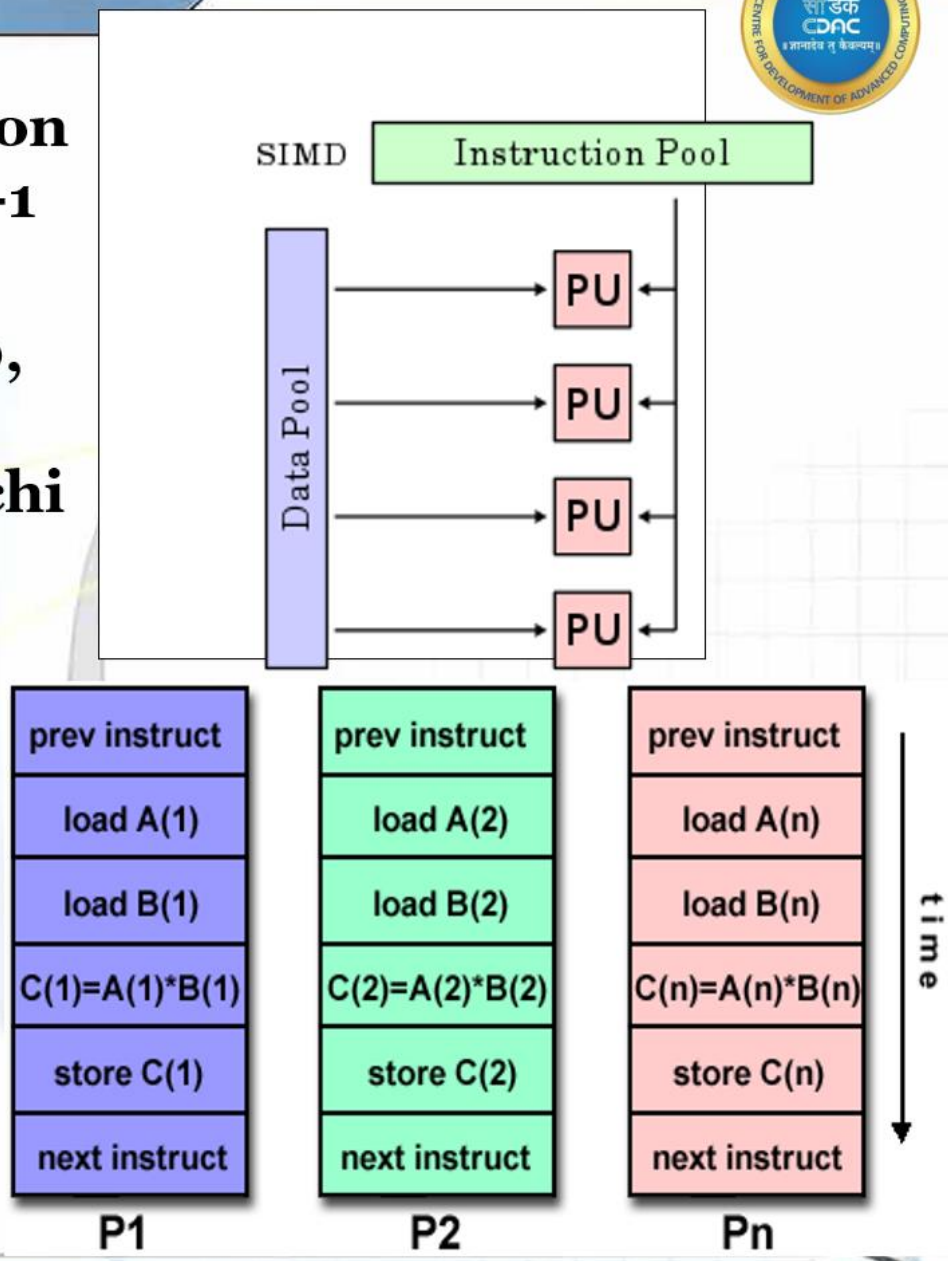
IBM 360





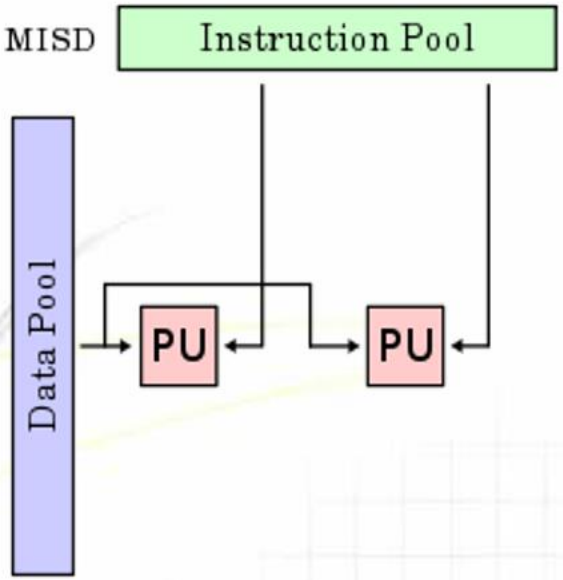
2. SIMD

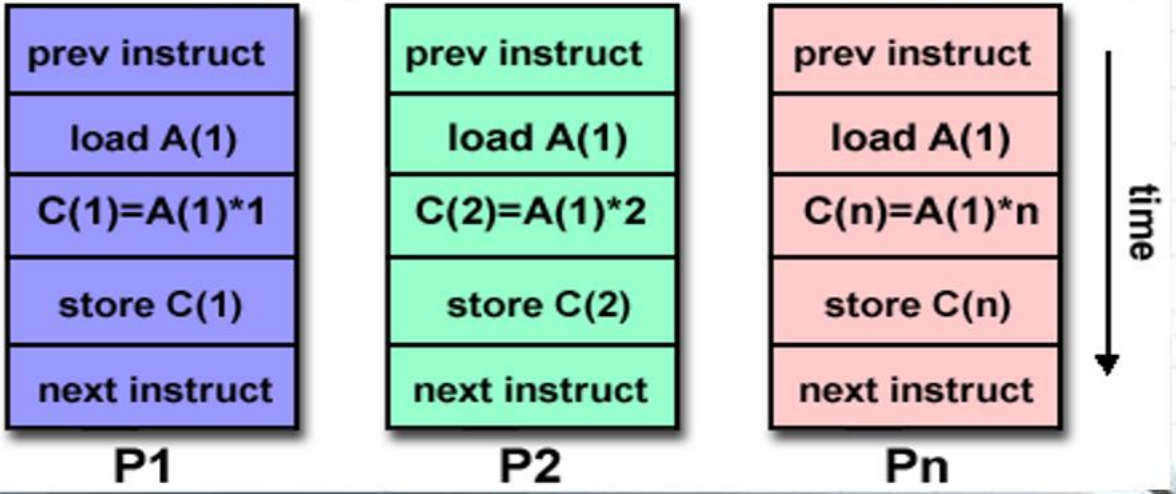
- **Processor Arrays: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV**
- **Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10**
- **Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.**



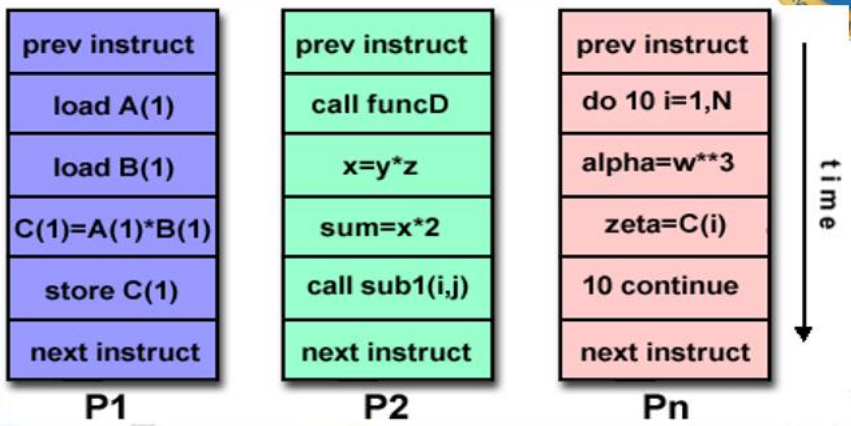
3. MISD

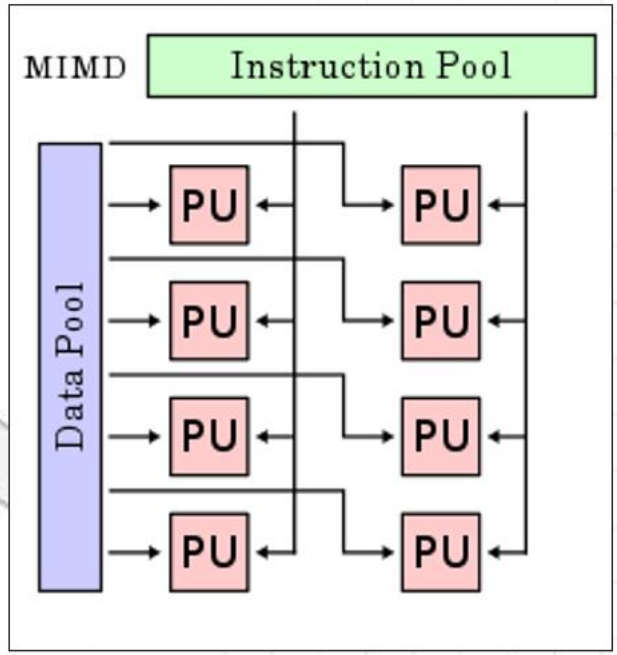
- **Some conceivable uses might be:**
  - **multiple frequency filters operating on a single signal stream**
  - **multiple cryptography algorithms attempting to crack a single coded message.**





4. MIMD



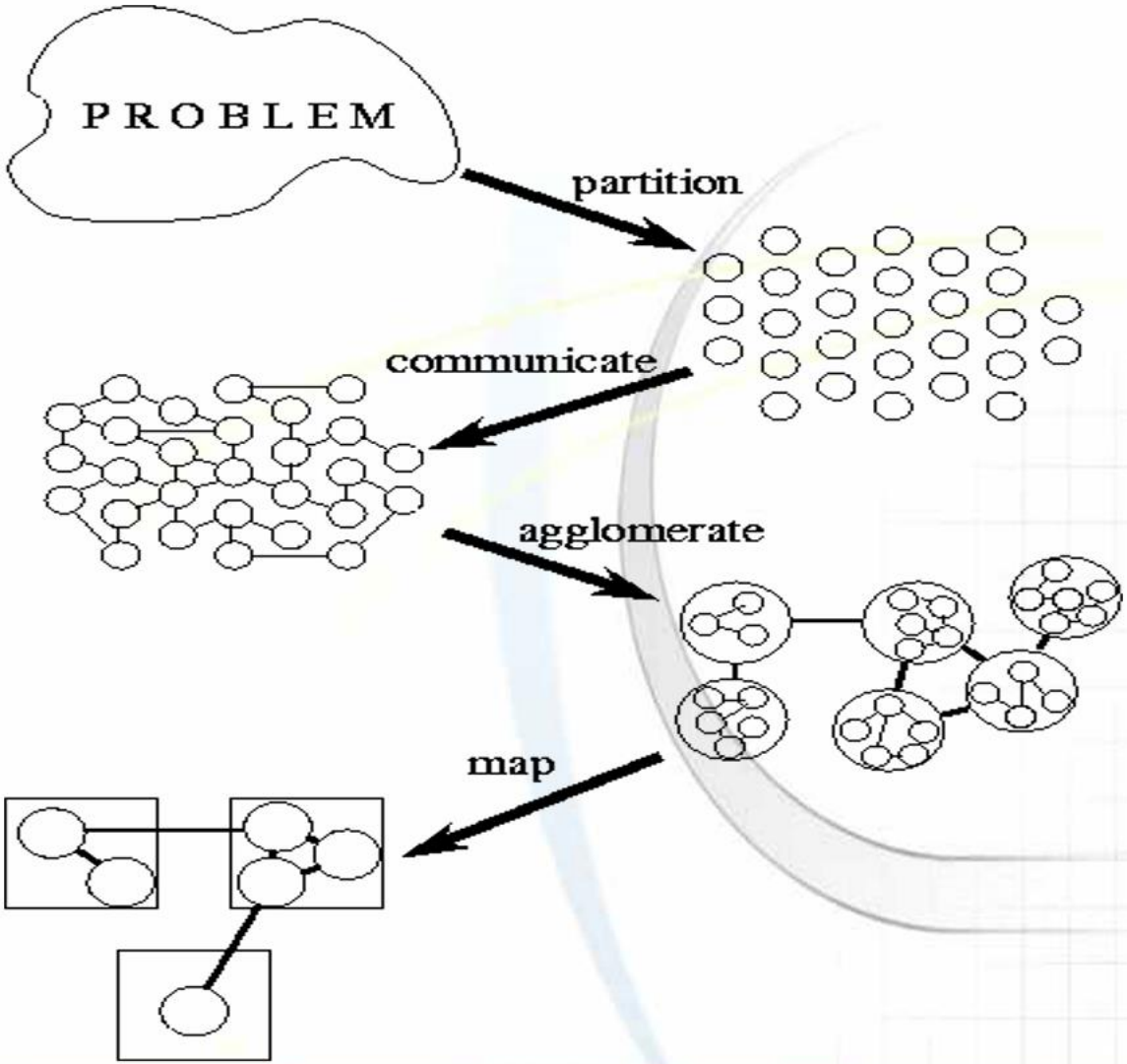


Most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.

Parallel Programming

Program Design

Program Design



Partitioning:

- divide the computation to be performed and the data operated on by the computation into small tasks.
- The focus here should be on identifying tasks that can be executed in parallel.

Communication:

- determine what communication needs to be carried out among the tasks identified in the previous step.

Agglomeration or aggregation:

- combine tasks and communications identified in the first step into larger tasks.

  For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

Mapping:

- assign the composite tasks identified in the previous step to processes/threads.

  This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Parallel Programming

Program Paradigms

Paradigms

- On a single machine with shared memory
  - OpenMP 3 and lesser
  - Pthreads
- On machines connected via network and have no shared memory
  - MPI
  - PGAS
- Accelerators, offload tasks from CPU to it
  - CUDA
  - OpenACC /OpenMP 4
- Heterogenous
  - SYCL programming (implementations like HIP / OneAPI)

OpenMP

```
#include <omp.h>
#define N     1000

main ()
{
int i, chunk=100;
float a[N], b[N], c[N];

for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

Compilation

- gcc -fopenmp pgm.c
- icc -openmp pgm.c
- pgcc –mp pgm.c

MPI

```
#define MASTER 0        //One process will take care of initialization
#define ARRAY_SIZE 1000    //Size of arrays that will be added together.
 int main (int argc, char *argv[]){
            int * a, * b, * c;
            int total_proc,rank,n_per_proc,n=ARRAY_SIZE,i;
            MPI_Status status
            MPI_Init (&argc, &argv);
            MPI_Comm_size (MPI_COMM_WORLD, &total_proc);
            MPI_Comm_rank (MPI_COMM_WORLD,&rank);
            int * ap,* bp,* cp;
            if (rank == MASTER)  {
                        a = (int *) malloc(sizeof(int)*n);
                        b = (int *) malloc(sizeof(int)*n);
                        c = (int *) malloc(sizeof(int)*n);
                        for(i=0;i<n;i++){
                                    a[i] = i;
                                    b[i] = i;
            }}
            n_per_proc = n/total_proc;
            ap = (int *) malloc(sizeof(int)*n_per_proc);
            bp = (int *) malloc(sizeof(int)*n_per_proc);
            cp = (int *) malloc(sizeof(int)*n_per_proc);
            MPI_Scatter(a, n_per_proc, MPI_INT, ap, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);
            MPI_Scatter(b, n_per_proc, MPI_INT, bp, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);
            for(i=0;i<n_per_proc;i++)
                        cp[i] = ap[i]+bp[i];
            MPI_Gather(cp, n_per_proc, MPI_INT, c, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);
            if(rank == MASTER)  {
              int good = 1;
              for(i=0;i<n;i++) {
               if (c[i] != a[i] + b[i]) {
                        printf("problem at index %lld\n", i);
                        good = 0;
                        break;
              }}if (good) { printf ("Values correct!\n");
            }}if (rank == MASTER)
                        free(a);  free(b); free(c);
            free(ap);  free(bp); free(cp);
            MPI_Finalize();
}
```

❑Compilation

mpicc pgm.c

OpenACC

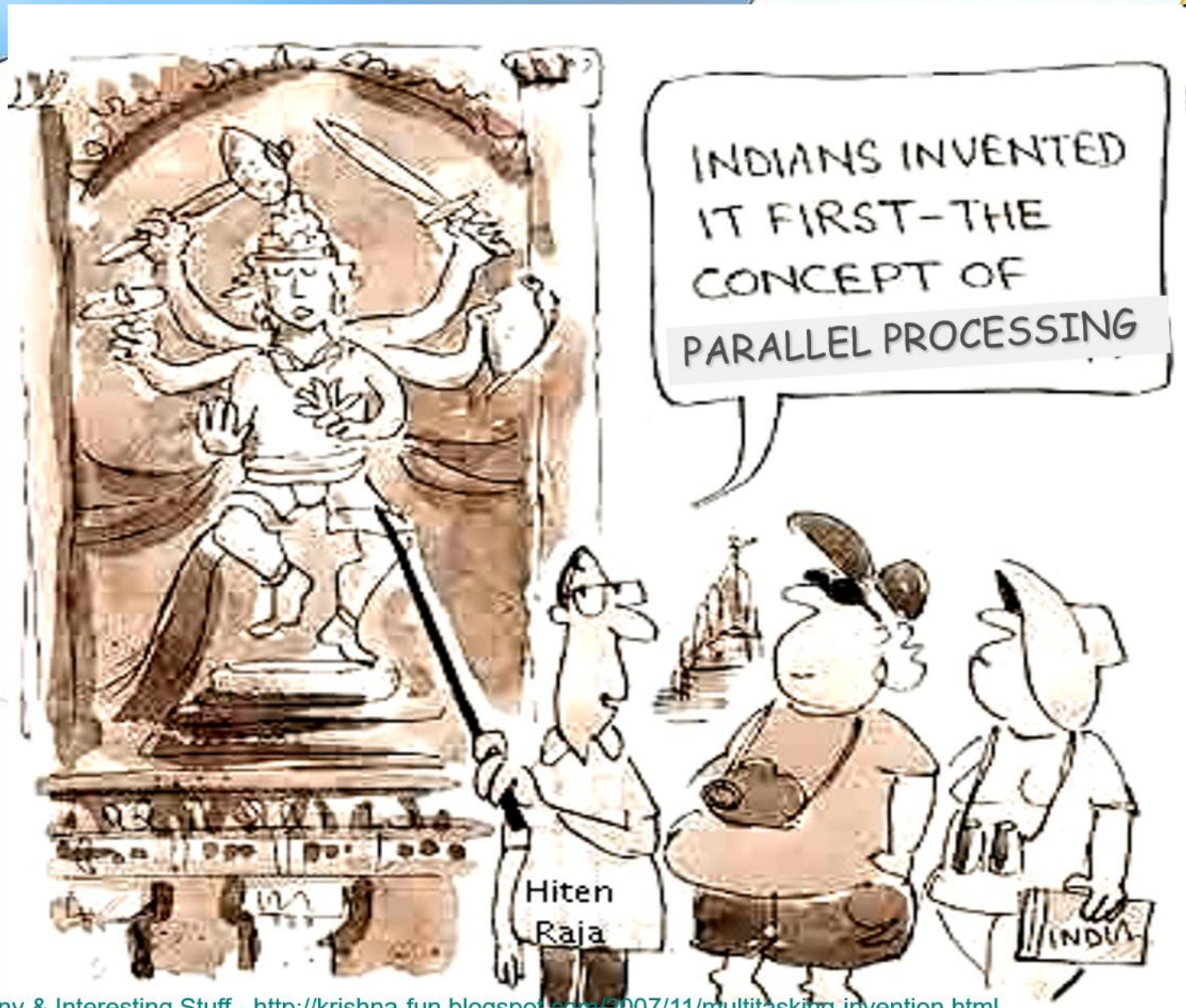
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main( int argc, char* argv[] )
{
   int n = 10000, i;
   double *restrict a, *restrict b;
   double *restrict c;
   size_t bytes = n*sizeof(double);
   a = (double*)malloc(bytes);
   b = (double*)malloc(bytes);
   c = (double*)malloc(bytes);
   for(i=0; i<n; i++) {
      a[i] = sin(i)*sin(i);
      b[i] = cos(i)*cos(i);
   }
  #pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
   for(i=0; i<n; i++) {
      c[i] = a[i] + b[i];
   }
   double sum = 0.0;
   for(i=0; i<n; i++) {
      sum += c[i];
   }
   sum = sum/n;
   printf("final result: %f\n", sum);
    free(a, b, c);
}
```

- **Compilation**
  - **pgcc –acc pgm.c**

Heterogeneous programming-SYCL

```
using namespace sycl;
size_t array_size = 10000;
void VectorAdd(queue &q, const int *a, const int *b, int *sum, size_t size) {
 range<1> num_items{size};
 auto e = q.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
 e.wait();
}
void InitializeArray(int *a, size_t size) {
 for (size_t i = 0; i < size; i++) a[i] = i;
}
int main(int argc, char* argv[]) {
 if (argc > 1) array_size = std::stoi(argv[1]);
 auto d_selector{default_selector_v};
 try {
  queue q(d_selector, exception_handler);
  std::cout << "Running on device: "
        << q.get_device().get_info<info::device::name>() << "\n";
  std::cout << "Vector size: " << array_size << "\n";
  int *a = malloc_shared<int>(array_size, q);
  int *b = malloc_shared<int>(array_size, q);
  int *sum_sequential = malloc_shared<int>(array_size, q);
  int *sum_parallel = malloc_shared<int>(array_size, q);
  InitializeArray(a, array_size);
  InitializeArray(b, array_size);
  for (size_t i = 0; i < array_size; i++) sum_sequential[i] = a[i] + b[i];
  VectorAdd(q, a, b, sum_parallel, array_size);
  int indices[]{0, 1, 2, (static_cast<int>(array_size) - 1)};
  constexpr size_t indices_size = sizeof(indices) / sizeof(int);
  free(a, b, q, sum_sequential, sum_parallel);
 } catch (exception const &e) {
  std::cout << "An exception is caught while adding two vectors.\n";
  std::terminate();
 } std::cout << "Vector add successfully completed on device.\n";
}
```

icx pgm.cpp



Courtesy: Funny & Interesting Stuff - http://krishna-fun.blogspot.com/2007/11/multitasking-invention.html

Applying Advanced Computing for Human Advancement

Thank you!