PARALLEL PROGRAMMING...

Parallel Programming: Overview

GOAL

Hardware Architecture

Architecture of a CPU versus GPU

AMD ROCm and CUDA Platforms

GPGPU (General Purpose computation on Graphics Processing Units)

Programming Interface for parallel computing

MPI (Message Passing Interface)

OpenMP (Open Multi-Processing)

Hybrid MPI & OpenMP
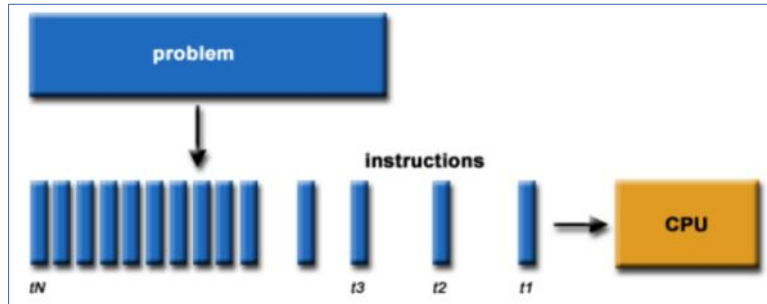
SPECX

Sequential / Parallel Programming

Traditionally, software is based on sequential calculations:

- One problem is the splitting in the instructions.
- These instructions are executed sequentially after each other.
- Executed by a single processor.
- At a time given, only one instruction is executed.

Parallel programming is the use of several calculation resources to solve a problem gives:

- One problem is the cutting into the parts that can put simultaneously.
- Each party is still discovered in instructions
- The instructions of each party are executed in parallel using several processors
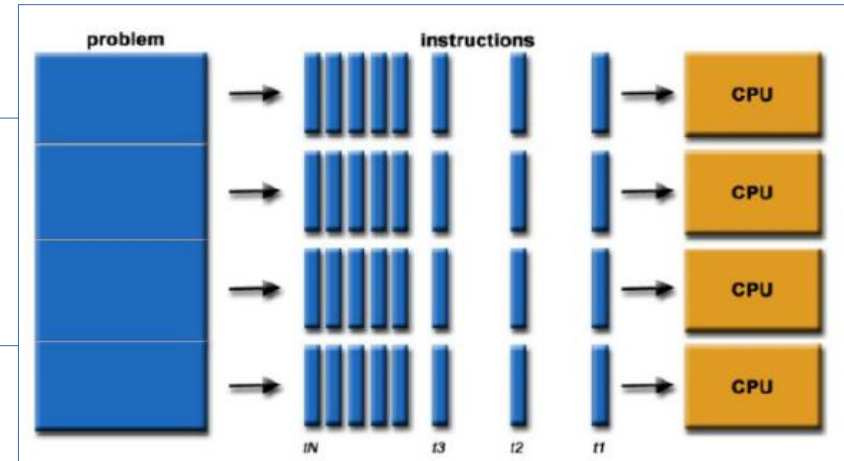
CPU (Central Processing Unit)
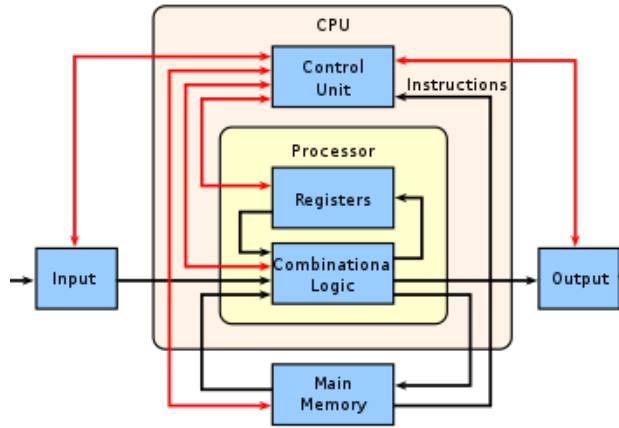
A CPU (Central Processing Unit) is the most important processor in a given computer. It executes instructions of a computer program, such as arithmetic, logic, controlling, and input/output (I/O) operations.

It constitutes the physical heart of the entire computer system; to it is linked various peripheral equipment, including input/output devices and auxiliary storage units.

The CPU and GPU are different in the following ways:

- CPU has less number of cores than GPU, and they run at a lower clock frequency.
- CPU handles various operations like calculating, watching movies, making presentation etc., while GPU aids the CPU by monitoring narrower and more specific tasks, such as graphics rendering.
- CPU does not support the parallel operation of data, while GPU can process multiple data streams simultaneously.
- CPU is slower than GPU and requires more memory !

CPU: Different architectures of the processor

There is a classification of the **different CPU architectures.** Five in number, they are used by programmers depending on the desired results:
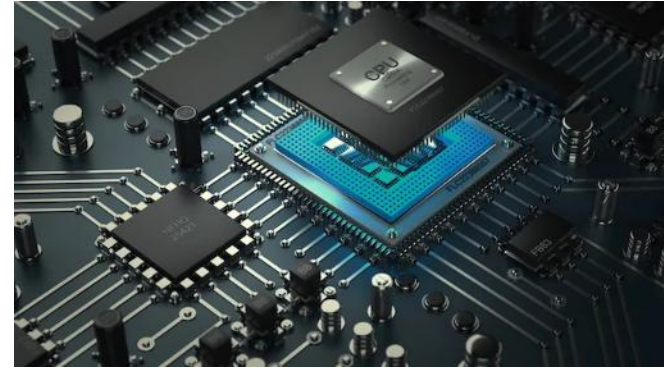
CISC: very complex addressing;

RISC: simpler addressing and instructions performed on a single cycle;
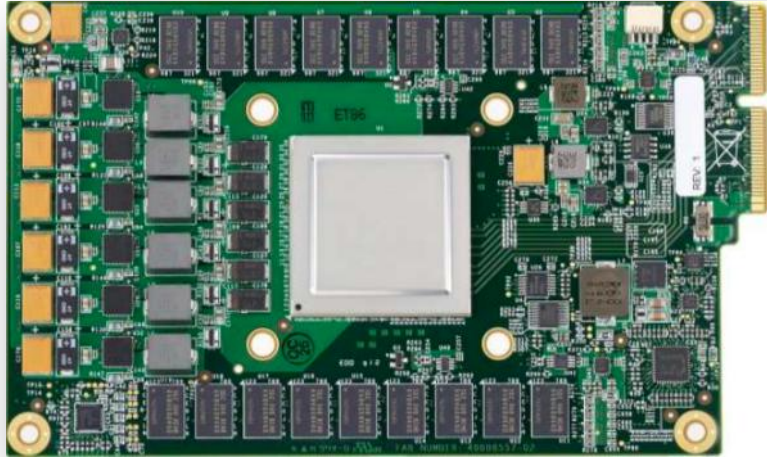
VLIW: long, but simpler instructions;

vectorial: contrary to the processing in number, the instructions are vectorial;

dataflow: data is active unlike other architectures.

GPU (Graphics Processing Unit)

A **GPU** is uses to speed up the process of creating and rendering computer graphics, designed to accelerate graphics and image processing.

It is the most important hardware.

But have later been used for ***non-graphic calculations*** involving embarrassingly parallel problems due to their parallel structure.

CPU vs GPU

CPU — Latency Oriented Cores

GPU — Throughput Oriented Cores
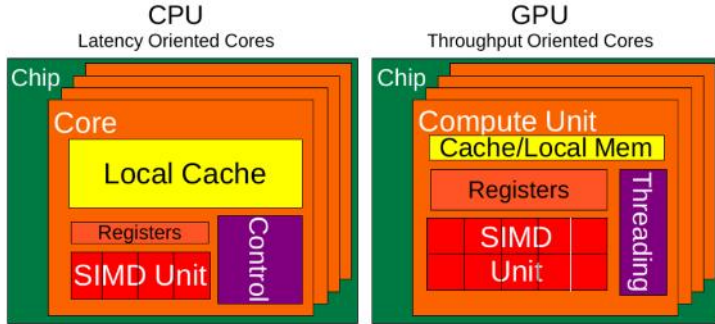
Architecturally, the **CPU** is composed of just a few cores with lots of cache memory that can handle a few software threads at a time.
In contrast, a **GPU** is composed of hundreds of cores that can handle thousands of threads simultaneously.

**GPU** is specialized for compute intensive, highly data parallel computation

- More area is dedicated to processing
- Good for high arithmetic intensity programs with a high ratio between arithmetic operations and memory operations.

CPU
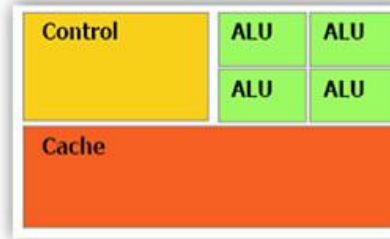
* Low compute density
* Complex control logic
* Large caches (L1$/L2$, etc.)
* Optimized for serial operations
  * Fewer execution units (ALUs)
  * Higher clock speeds
* Shallow pipelines (<30 stages)
* Low Latency Tolerance
* Newer CPUs have more parallelism

GPU
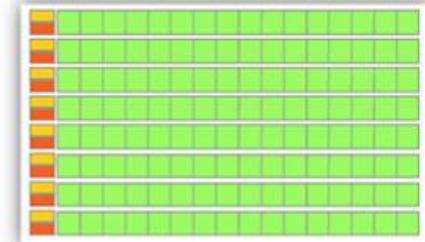
* High compute density
* High Computations per Memory Access
* Built for parallel operations
  * Many parallel execution units (ALUs)
  * Graphics is the best known case of parallelism
* Deep pipelines (hundreds of stages)
* High Throughput
* High Latency Tolerance
* Newer GPUs:
  * Better flow control logic (becoming more CPU-like)
  * Scatter/Gather Memory Access
  * Don't have one-way pipelines anymore

CPU vs GPU Comparison

|	CPU: Latency-oriented design	GPU: Throughput Oriented Design
Clock	High clock frequency	Moderate clock frequency
Caches	Large sizes Converts high latency accesses in memory to low latency accesses in cache	Small caches To maximize memory throughput
Control	Sophisticated control system Branch prediction to reduce latency due to branching Data loading to reduce latency due to data access	Single controlled No branch prediction No data loading
Powerful Arithmetic Logic Unit (ALU)	Reduced operation latency	Numerous, high latency but heavily pipelined for high throughput
Other aspects	Lots of space devoted to caching and control logic. Multi-level caches used to avoid latency Limited number of registers due to fewer active threads Control logic to reorganize execution, provide ILP, and minimize pipeline hangs	Requires a very large number of threads for latency to be tolerable
Beneficial aspects for applications	CPUs for sequential games where latency is critical. CPUs can be 10+X faster than GPUs for sequential code.	GPUs for parallel parts where throughput is critical. GPUs can be 10+X faster than GPUs for parallel code.

GPU (Graphics Processing Unit)

- **GPU** is the chip in computer video cards, PS3, Xbox, etc
  - Designed to realize the 3D graphics pipeline
    - Application →Geometry →Rasterizer→Image
- **GPU** development:
  - Fixed graphics hardware
  - Programmable vertex/pixel shaders
  - **GPGPU**
    - general purpose computation (beyond graphics) using GPU in applications other than 3D graphics
    - GPGPU can be treated as a co-processor for compute intensive tasks
      - With sufficient large bandwidth between CPU and GPU.

GPU Performance
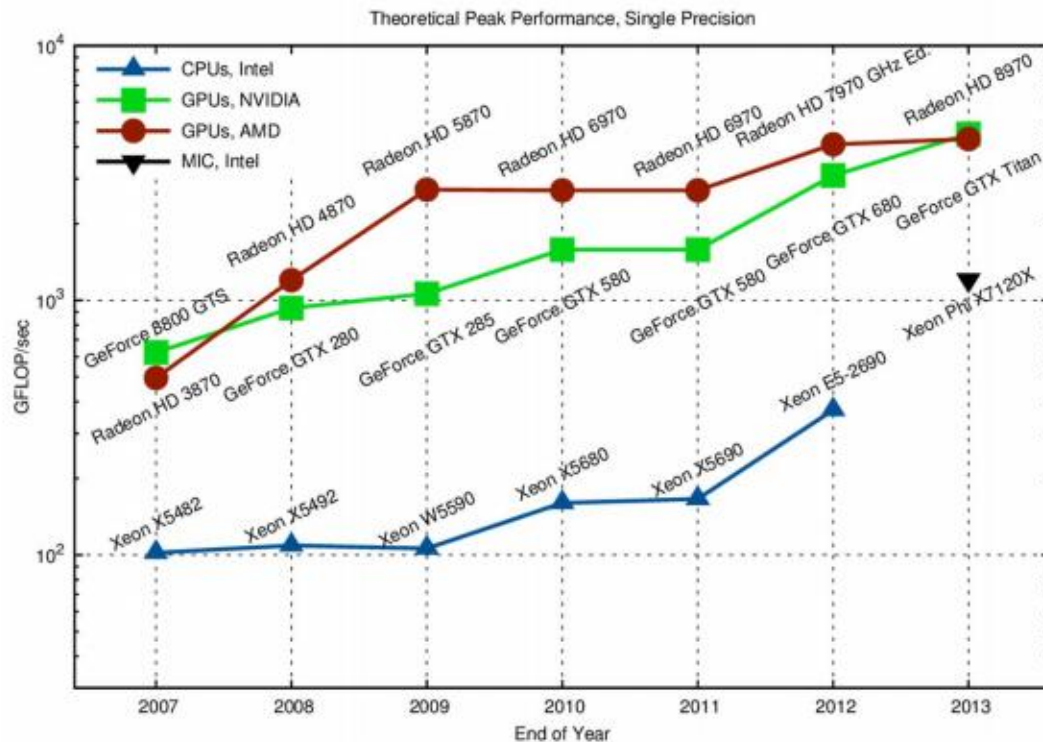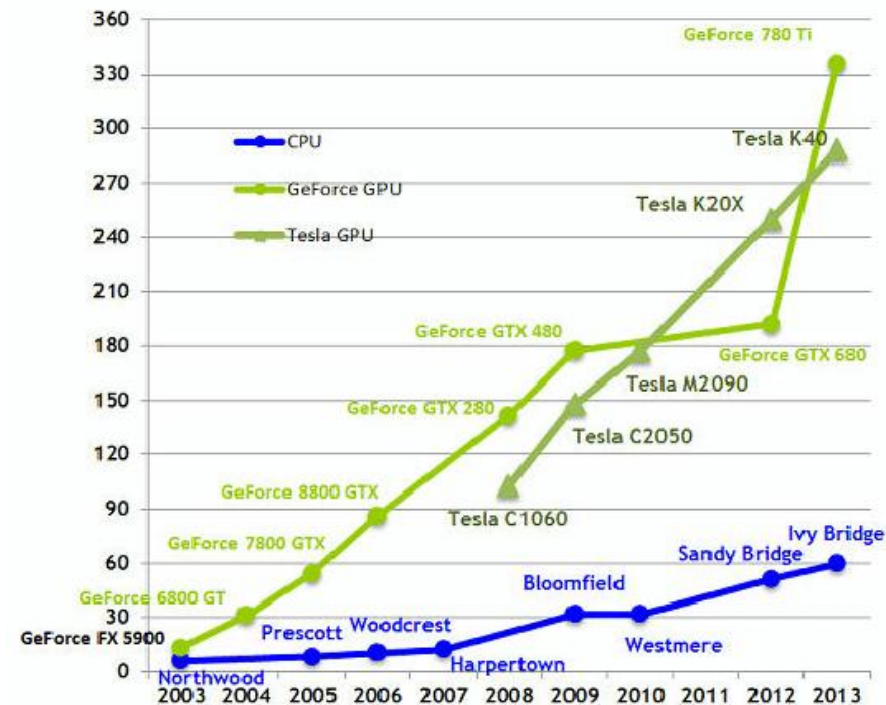
GPU Memory Bandwidth

GPU memory bandwidth is a measure of the data transfer speed between a GPU and the system across a bus, such as PCI Express (PCIe) or Thunderbolt.

Peak Memory Bandwidth Comparison
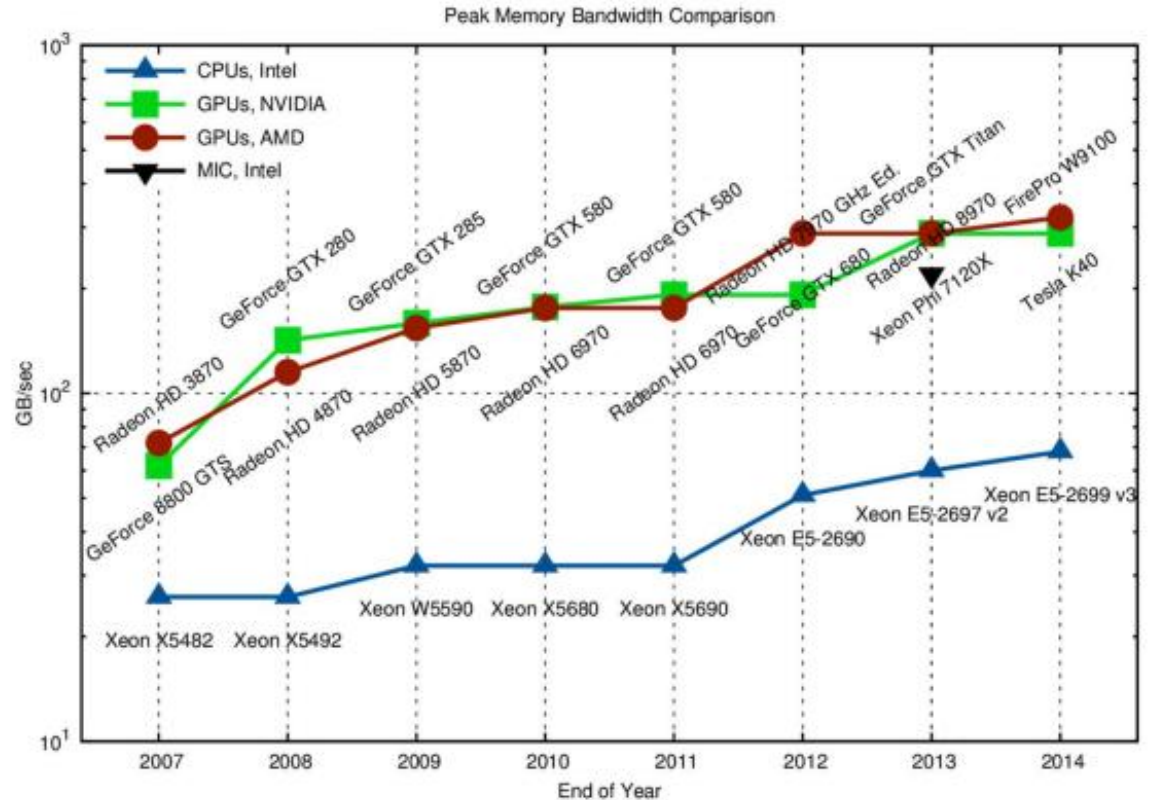
CPU and GPU CONNECTION

Processing flow on CUDA
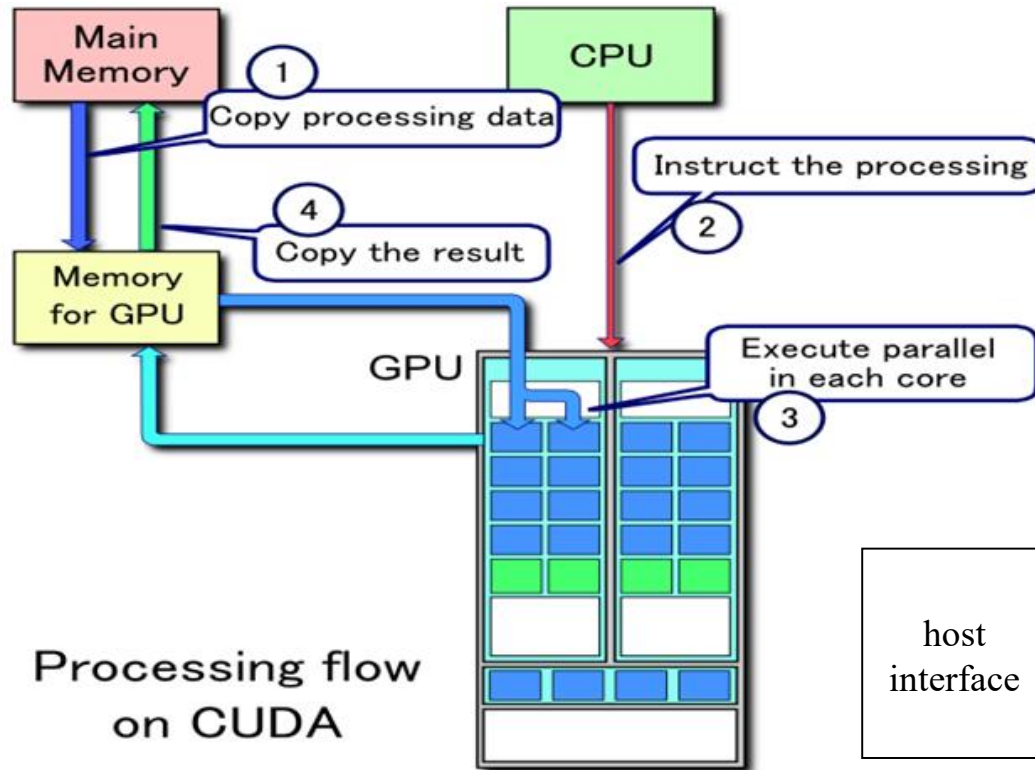
The GPU pipeline

▸ GPU receives geometry information from the CPU as an input and provides a picture as an output

host interface	→	vertex processing	→	triangle setup	→	fragment processing	→	memory interface

Native GPU code: HIP/CUDA

CUDA *from NVIDIA*
- Has been a standard for native GPU code for years
- Extensive set of optimized libraries available
- Custom syntax (extension of C++) supported only by CUDA compilers
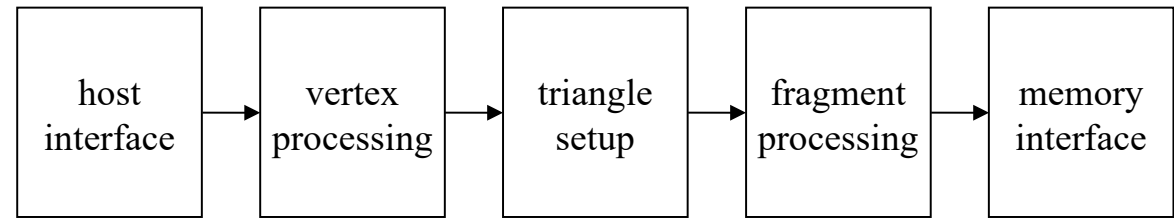- Support only for NVIDIA devices

HIP *(Heterogeneous-computing Interface for Portability) from AMD*
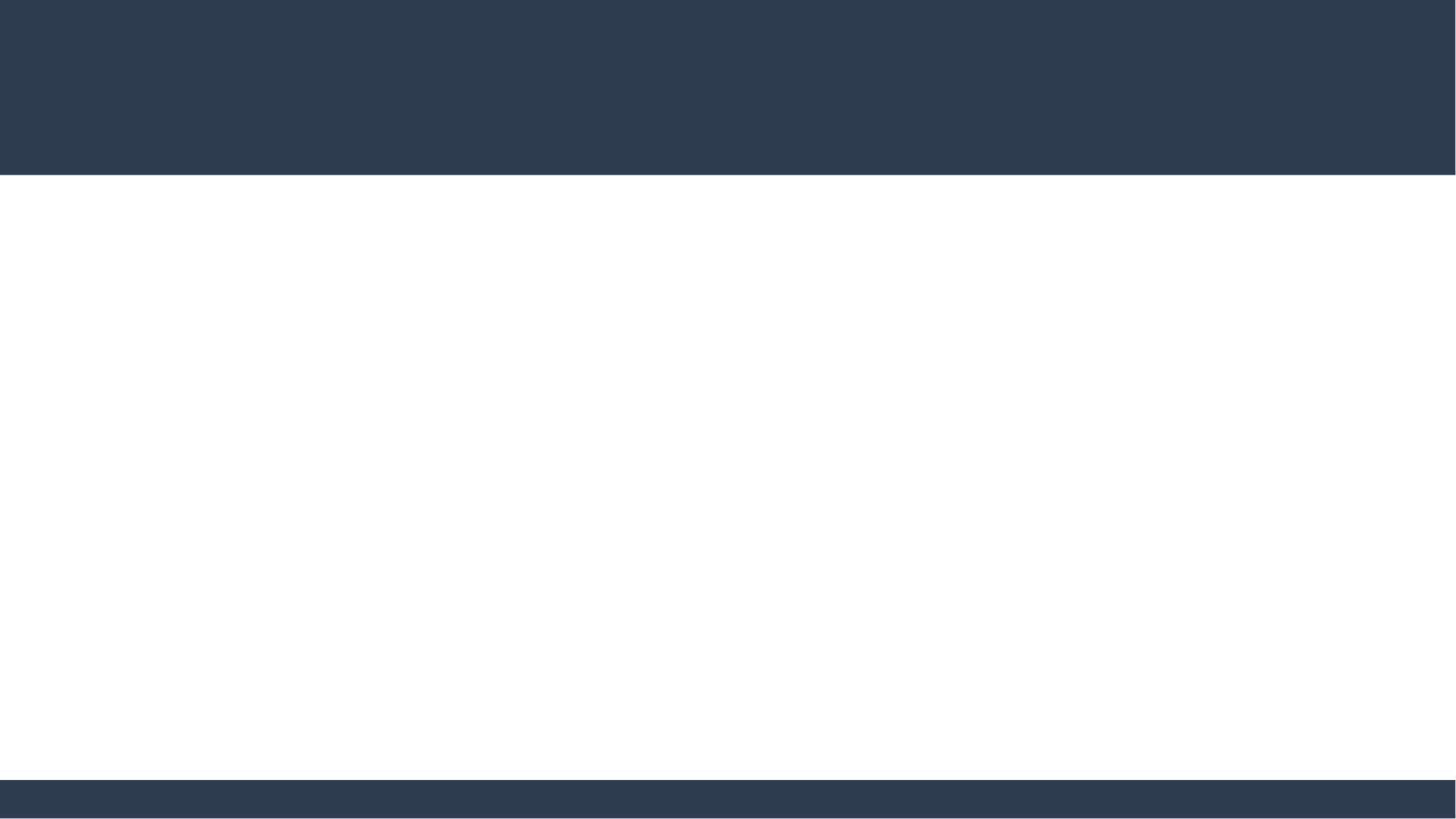- AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
- Standard C++ syntax, uses nvcc/hcc compiler in the background
- Almost a one-on-one clone of CUDA from the user perspective
- Ecosystem is new and developing fast

AMD and **NVIDIA** offers also a wide set of optimized libraries and tools

GPGPU

General **P**urpose computation

on **G**raphics **P**rocessing **U**nits.
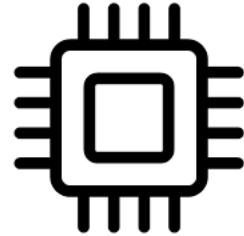
GPGPU (General-Purpose Graphics Processing Unit)

- **GPGPU:** Using graphic hardware for **non-graphic** computations

- Prefect for massive parallel processing on data paralleled applications

- **GPU** acts as an "**accelerator**" to the CPU (heterogeneous system)
  - Most lines of code are executed on the CPU (serial computing)
  - Key computational kernels are executed on the GPU (stream computing)
    - Taking advantage of the large number of cores and high graphics memory bandwidth

- **GPUs** now firmly established in HPC industry
  - Can augment each node of parallel system with GPUs

GPGPU: Programming Considerations

- Standard (CPU) code *will not run on* a GPU unless it is adapted

- Programmer must
  - decompose problem onto the hardware in a specific way (e.g. using a hierarchical thread/grid model in CUDA)
  - Manage data transfers between the separate CPU and GPU memory spaces.
  - Traditional language (C, C++, Fortran etc) not enough, need extensions, directives, or new language.

- Once code is ported to GPU, optimization work is usually required to tailor it to the hardware and achieve good performance

- Many researchers are now successfully exploiting GPUs
  - Across a wide range of application areas

GPGPU (General-Purpose Graphics Processing Unit)
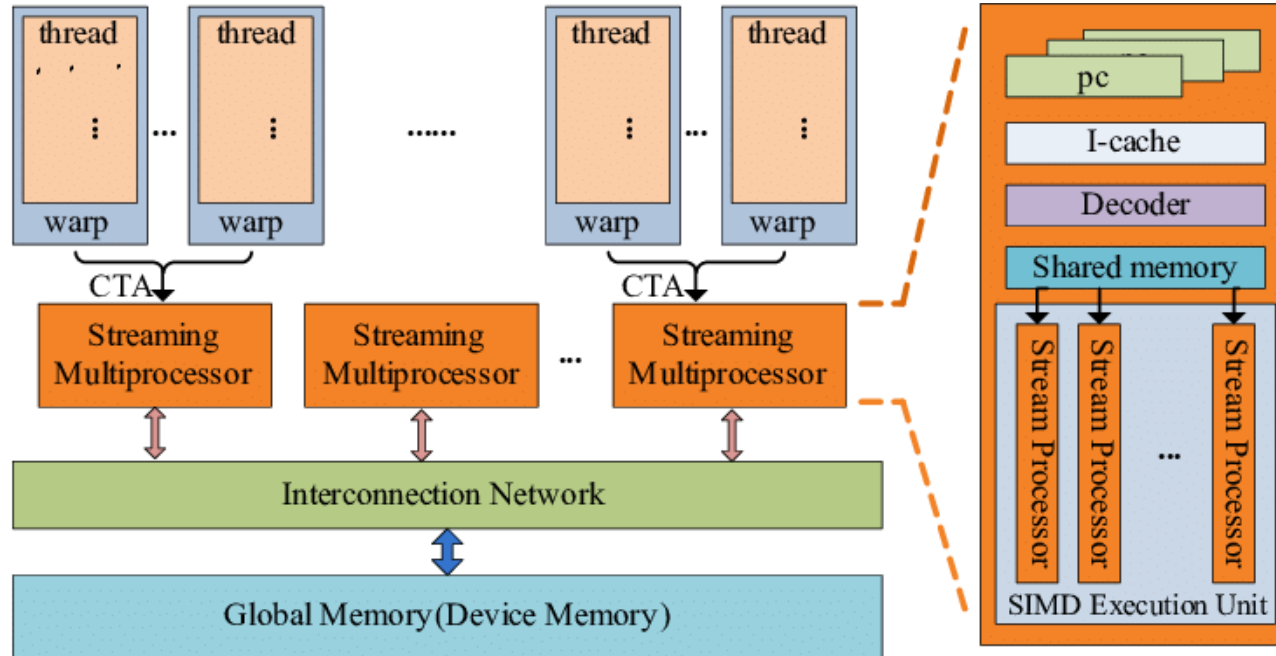
A GPGPU is a GPU that is programmed for purposes beyond graphics processing,

such as performing computations typically conducted by a Central Processing Unit (CPU).

GPGPU (General-Purpose Graphics Processing Unit)

Advantages of GPGPU

➢ GPUs have many more cores than CPUs, which allows them to process large amounts of data in parallel. This can result in significant speedups for some problems, especially those involving matrices, vectors, images or graphics.

➢ GPUs can also handle floating point operations more efficiently than CPUs, which is important for scientific computing and machine learning applications.

➢ GPUs can be used to accelerate various domains such as computer vision, natural language processing, cryptography, bioinformatics, physics simulation, etc.

GPGPU (General-Purpose Graphics Processing Unit)

Disadvantages of GPGPU

➢ GPUs are not suitable for all kinds of problems, especially those that require sequential or branching logic, complex data structures, or synchronization among threads.

➢ GPUs have limited memory and bandwidth compared to CPUs, which can limit the amount of data that can be transferred or processed at once.

➢ GPUs require specialized programming languages and APIs to access their features, which can increase the complexity and learning curve for developers.

GPGPU Programming Languages and APIs

There are several options for programming GPGPU applications:

CUDA: A proprietary platform developed by Nvidia that allows programmers to write C/C++ code that runs directly on Nvidia GPUs. It also provides libraries and tools for various domains such as linear algebra, image processing, deep learning, etc.

OpenCL: An open standard developed by the Khronos Group that supports multiple platforms and devices, including CPUs, GPUs, FPGAs, etc. It defines a C-like language and a runtime API for executing kernels on heterogeneous devices.

DirectCompute: A Microsoft API that is part of DirectX 11 and 12 that enables GPGPU programming on Windows platforms. It supports HLSL shaders and C++ AMP extensions for writing compute kernels.

Metal: An Apple API that provides low-level access to the GPU on iOS and macOS platforms. It supports Swift and Objective-C languages for writing compute shaders.

Vulkan: A cross-platform API developed by the Khronos Group that provides low-level access to the GPU and other devices. It supports SPIR-V as an intermediate language for writing compute shaders.

GPGPU Programming Languages and APIs

There are several options for programming GPGPU applications:

CUDA: A proprietary platform developed by Nvidia that allows programmers to write C/C++ code that runs directly on Nvidia GPUs. It also provides libraries and tools for various domains such as linear algebra, image processing, deep learning, etc.

OpenCL: An open standard developed by the Khronos Group that supports multiple platforms and devices, including CPUs, GPUs, FPGAs, etc. It defines a C-like language and a runtime API for executing kernels on heterogeneous devices.

DirectCompute: A Microsoft API that is part of DirectX 11 and 12 that enables GPGPU programming on Windows platforms. It supports HLSL shaders and C++ AMP extensions for writing compute kernels.
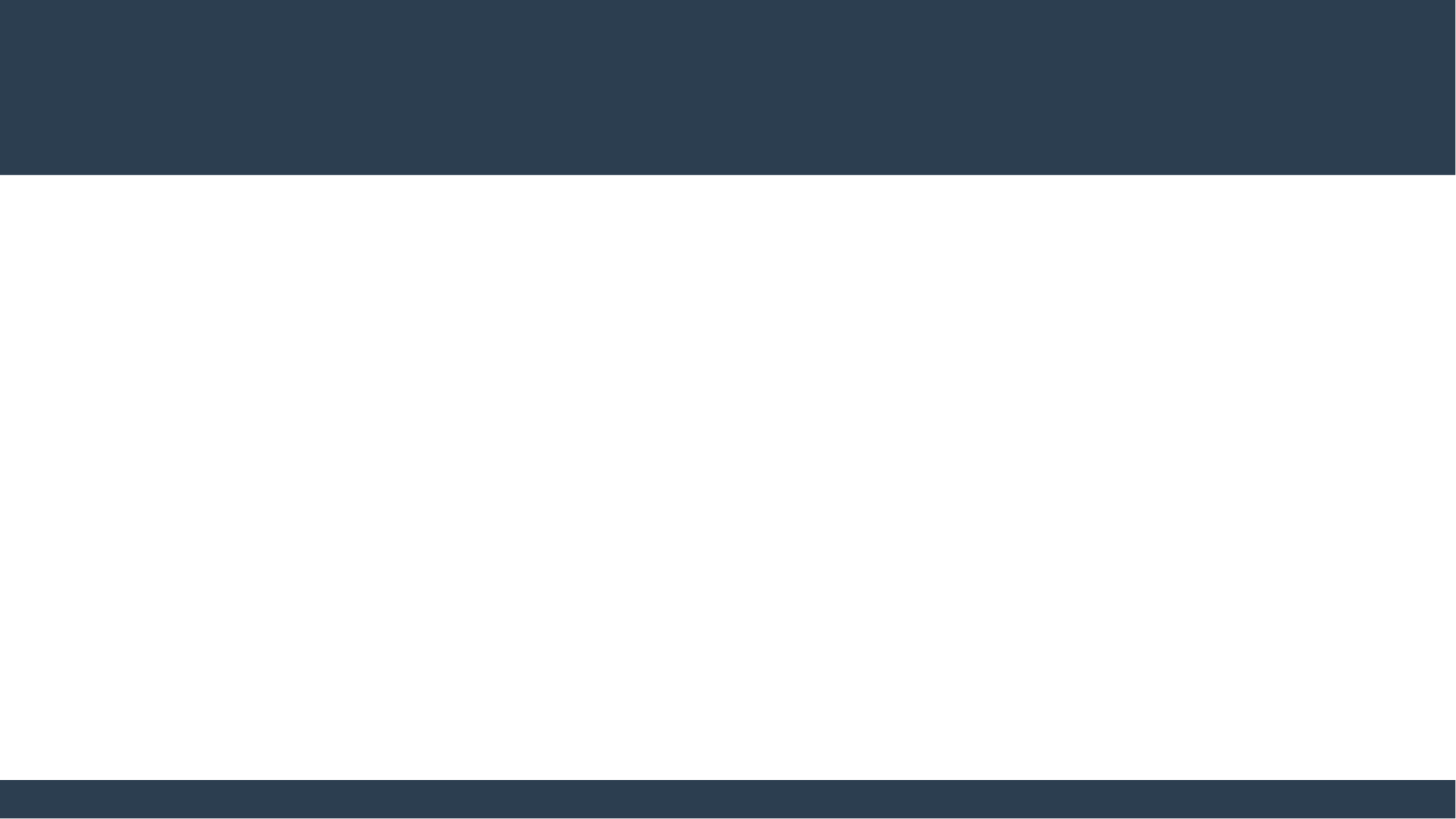
Metal: An Apple API that provides low-level access to the GPU on iOS and macOS platforms. It supports Swift and Objective-C languages for writing compute shaders.

Vulkan: A cross-platform API developed by the Khronos Group that provides low-level access to the GPU and other devices. It supports SPIR-V as an intermediate language for writing compute shaders.

ROCm: is an Advanced Micro Devices (AMD) software stack for graphics processing unit (GPU) programming and spans several domains: GPGPU, HPC ,heterogeneous computing

Programming interface for parallel computing

병렬 컴퓨팅을 위한 프로그래밍
인터페이스

Programming Interface

MPI (Message Passing Interface) and

OpenMP (Open Multi-Processing)

Programming interface MPI, OpenMP

MPI, OpenMP two complementary parallelization models.

MPI (Message Passing Interface) is a multi-process model whose mode of communication between the processes is **explicit** (communication management is the responsibility of the user).

Generally used on multiprocessor machines with distributed memory.

It is a library for passing messages between processes without sharing.
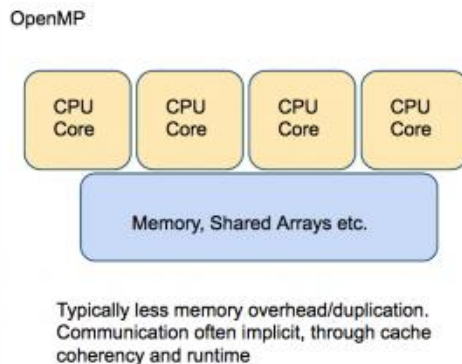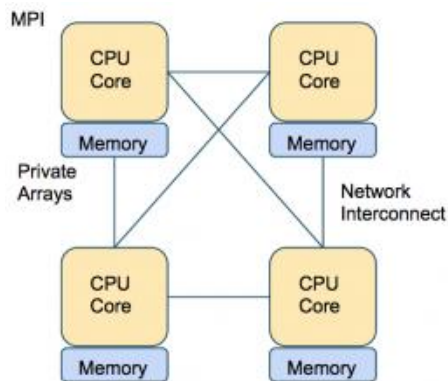
OpenMP (Open Multi-Processing) is a multitasking model whose mode of communication between tasks is implicit (the management of communications is the responsibility of the compiler).

OpenMP is used on shared-memory multiprocessor machines. It focuses on shared memory paradigms. It is a language extension for expressing data-parallel operations (usually parallelized arrays over loops).

MPI vs OpenMP

✅ **MPI**	**OpenMP**
Portable to a distributed and shared memory machine. Scale beyond a node No data placement issues	Easy to implement parallelism Implicit communications Low latency, high bandwidth Dynamic Load Balancing
❌ **MPI**	**OpenMP**
Explicit communication High latency, low bandwidth Difficult load balancing	Only on nodes or shared memory machines Scale on Node Data placement problem

MPI (Message Passing Interface)

MPI (Message Passing Interface)

MPI is a library of subroutines (in Fortran,C)

MPI allows the coordination of a program running as multiple processes in a distributed-memory environment.

MPI is flexible enough to also be used in a shared-memory environment.

MPI programs can be used and compiled on a wide variety of single platforms or (homogeneous or heterogeneous) clusters of computers over a network.

MPI library is standardized, should work (without further changes!) on any machine on which the MPI library is installed.

MPI: Basic Environment

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

Initializes MPI environment
Must be called in every MPI program
Must be first MPI call
Can be used to pass command line arguments to all

```
MPI_Finalize()
```

Terminates MPI environment
Last MPI function call

MPI: Basic Environment

```
MPI_Comm_rank(comm, &rank)
```

Returns the rank of the calling MPI process
Within the communicator, comm
MPI_COMM_WORLD is set during Init(...)
Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

Returns the total number of processes
Within the communicator, comm

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI: Point-to-Point Communication

`MPI_Send(&buf, count, datatype, dest, tag, comm)`

Send a message
Returns only after buffer is free for reuse (Blocking)

`MPI_Recv(&buf, count, datatype, source, tag, comm, &status)`

Received a message
Returns only when the data is avaible
Blocking

`MPI_SendRecv(...)`

Two way communication
Blocking
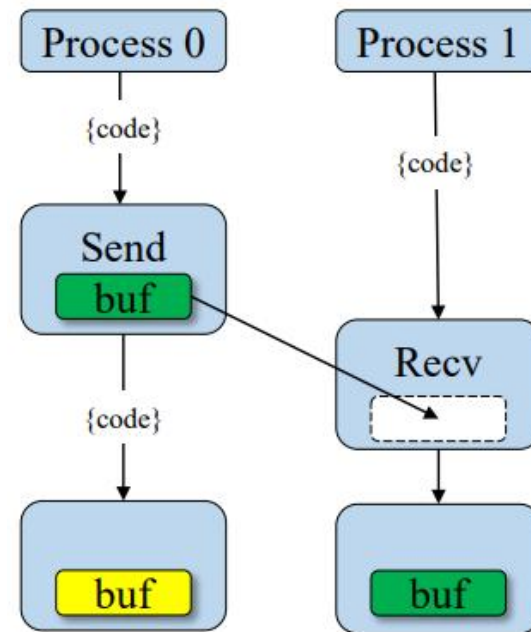
MPI: Point-to-Point Communication

Blocking
- Only returns after completed
    - Received: data has arrived and ready to use
    - Send: safe to reuse sent buffer
- Be aware of deadlocks
- Tip: use when possible

Non-Blocking
- returns immediately
    - Unsafe to modify buffers until operation is known to be complete
- Allows computation and communication to overlap
- Tip: Use only when needed

MPI: Deadlock

Blocking calls can resluts in deadlock

One process is waiting for message that will never arrive
Only option is to abort the interrupt/kill the code (CTRL-c)
Might not always deadlock - depends on size of system buffer
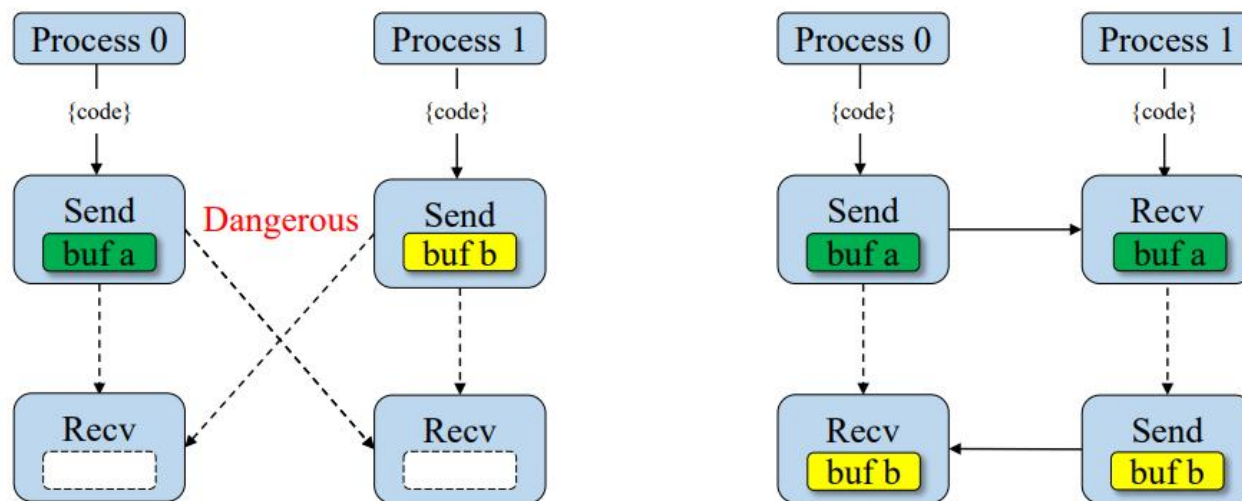
MPI: Coolective Communication (BCast)

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

Broadcast a message from the root process to all other processes.
Useful when reading in input parameters from file.
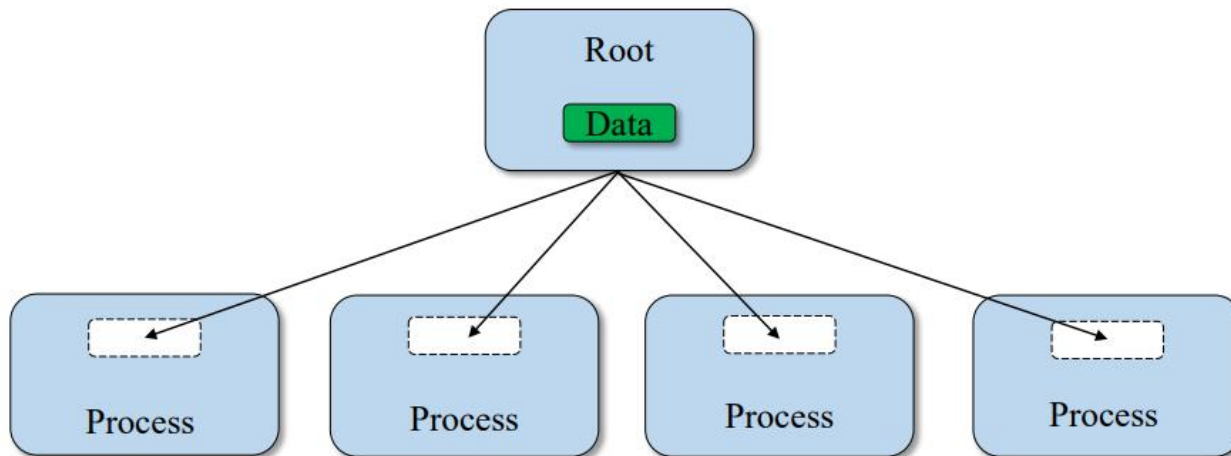
MPI: Collective Communication (Scatter)

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

Sends individual messages from the root process to all other processes.
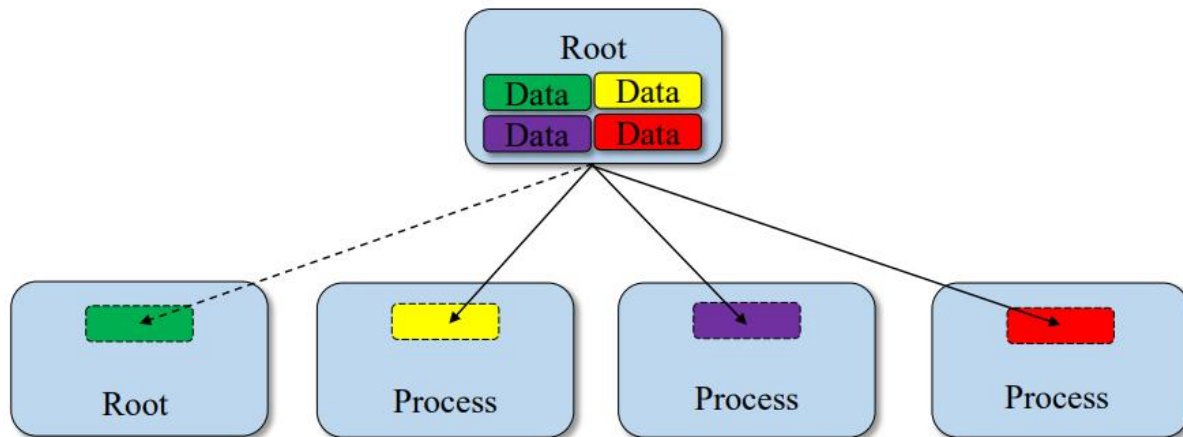
MPI: Collective Communication (Gather)

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

Opposite of Scatter.
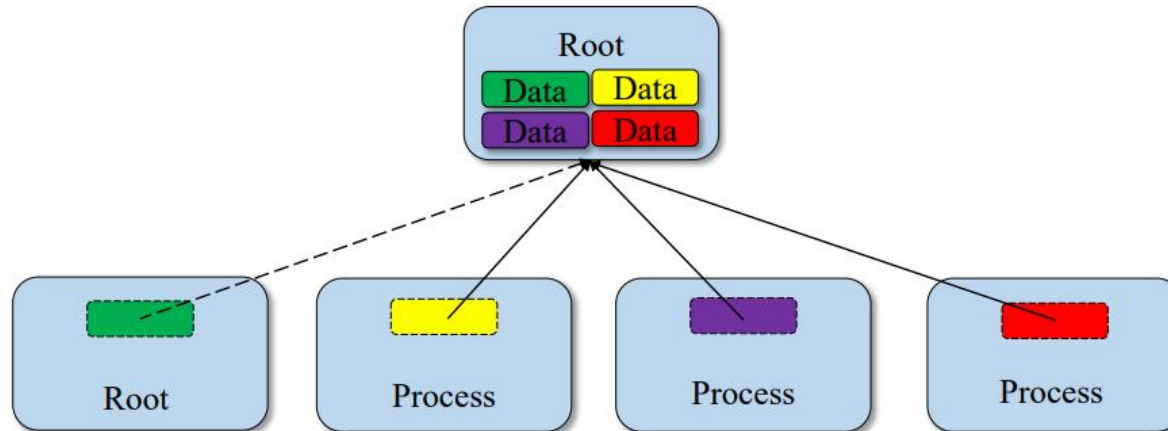
MPI: Collective Communication (Reduce)

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

Applies reduction operation on data from all processes.
Puts results on root process.

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD
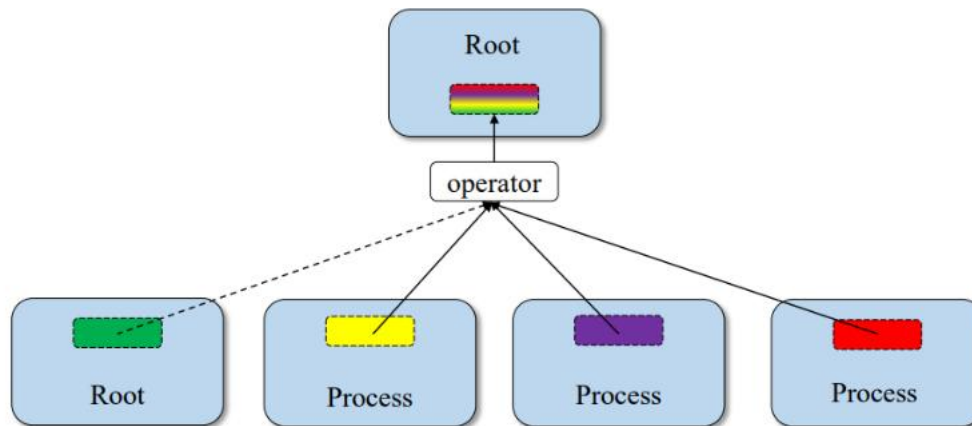
MPI: Collective Communication (Allreduce)

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

Applies reduction operation on data from all processes.
Store results on all processes.
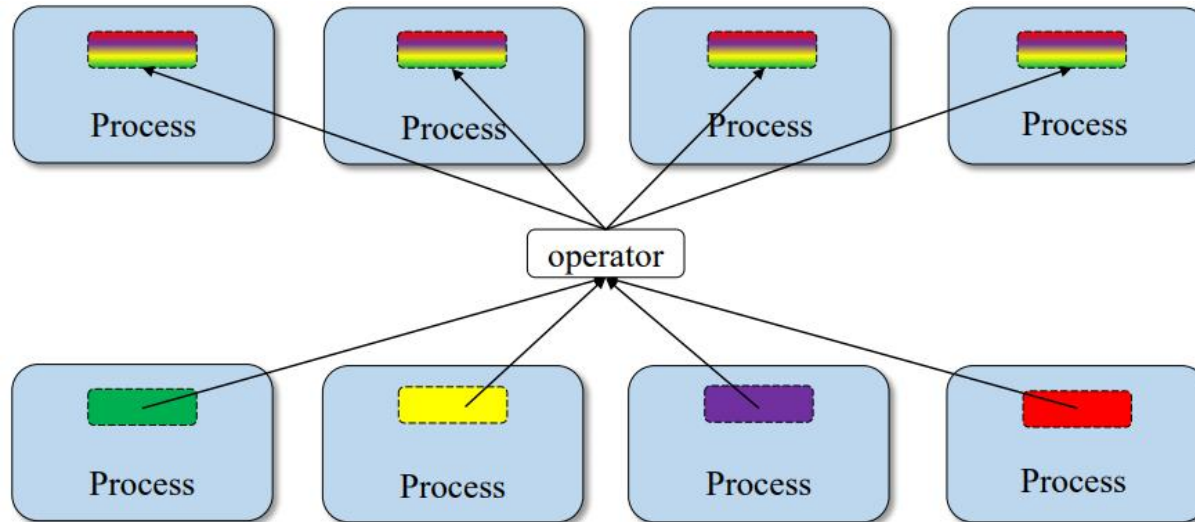
MPI: Collective Communication (Barrier)

```
MPI_Barrier(comm)
```

Process synchronization (blocking).
All processes forced to wait for each other.
Use only where necessary.
Will reduce parallelism.

MPI: keywords

1 environment
• MPI Init: Initialization of the MPI environment
• MPI Comm rank: Rank of the process
• MPI Comm size: Number of processes
• MPI Finalize: Deactivation of the MPI environment
• MPI Abort: Stopping of an MPI program
• MPI Wtime: Time taking

2 Point-to-point communications
• MPI Send: Send message
• MPI Isend: Non-blocking message sending
• MPI Recv: Message received
• MPI Irecv: Non-blocking message reception
• MPI Sendrecv and MPI Sendrecv replace: Sending and receiving messages
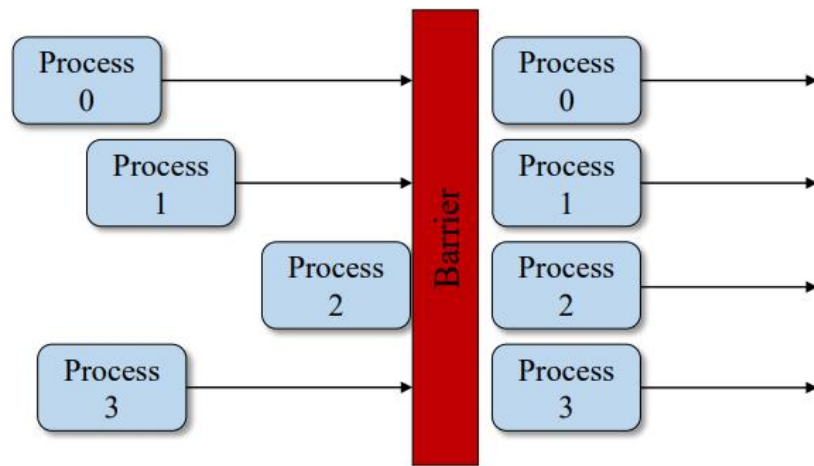• MPI Wait: Waiting for the end of a non-blocking communication
• MPI Wait all: Wait for the end of all non-blocking communications

3 Collective communications
• MPI Bcast: General broadcast
• MPI Scatter: Selective spread
• MPI Gather and MPI Allgather: Collecting
• MPI Alltoall: Collection and distribution
• MPI Reduce and MPI Allreduce: Reduction
• MPI Barrier: Global synchronization

4 Derived Types
• MPI Contiguous type: Contiguous types
• MPI Type vector and MPI Type create hvector: Types with a con-standing
• MPI Type indexed: Variable pitch types
• MPI Type create subarray: Sub-array types
• MPI Type create struct: H and erogenous types
• MPI Type commit: Type commit
• MPI Type get extent: Recover the extent
• MPI Type create resized: Change of scope
• MPI Type size: Size of a type
• MPI Type free: Release of a type

MPI: Keywords

5 Communicator
- MPI Comm split: Partitioning of a communicator
- MPI Dims create: Distribution of processes
- MPI Cart create: Creation of a Cartésian topology
- MPI Cart rank: Rank of a process in the Cartésian topology
- MPI Cart coordinates: Coordinates of a process in the Cart esian topology
- MPI Cart shift: Rank of the neighbors in the Cartésian topology
- MPI Comm free: Release of a communicator

6 MPI-IO
- MPI File open: Opening a file
- MPI File set view: Changing the view
- MPI File close: Closing a file

6.1 Explicit addresses
- MPI File read at: Reading
- MPI File read at all: Collective reading
- MPI File write at: Writing

6.2 Individual pointers
- MPI File read: Reading
- MPI File read all: collective reading
- MPI File write: Writing
- MPI File write all: collective writing
- MPI File seek: Pointer positioning

6.3 Shared pointers
- MPI File read shared: Read
- MPI File read ordered: Collective reading
- MPI File seek shared: Pointer positioning

7.0 Symbolic constants
- MPI COMM WORLD, MPI SUCCESS
- MPI STATUS IGNORE, MPI PROC NULL
- MPI INTEGER, MPI REAL, MPI DOUBLE PRECISION
- MPI ORDER FORTRAN, MPI ORDER C
- MPI MODE CREATE,MPI MODE RONLY,MPI MODE WRONLY

MPI: Program Basics

| Include MPI Header File | `#include <mpi.h>` |

Start of Program

(Non-interacting Code)

```
int main (int argc, char *argv[])
{
```

| Initialize MPI | `MPI_Init(&argc, &argv);` |

```
.
.       // Run parallel code
.
```

| Run Parallel Code & Pass Messages |

| End MPI Environment | `MPI_Finalize();` // End MPI Envir |

(Non-interacting Code)

End of Program

```
return 0;
}
```

MPI: Example

```c
#include <mpi.h>
#include <stdio.h>


int main (int argc, char *argv[]) {

  int rank, size;

  MPI_Init (&argc, &argv);   //initialize MPI library

  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

  //do something
  printf ("Hello World from rank %d\n", rank);
  if (rank == 0) printf("MPI World size = %d processes\n", size);

  MPI_Finalize(); //MPI cleanup

  return 0;
}
```

- 4 processes

```
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 2
Hello World from rank 1
```

- Code ran on each process independently
- MPI Processes have *private* variables
- Processes can be on completely different machines

COMPILING an MPI Program

➤ **Compiling a program** for MPI is almost just like compiling a regular C or C++ program

   ➤ The C compiler is **mpicc** and the **C++** compiler is **mpic++**.

      ➤ For example, to compile **MyProg.c** you would use a command like

         ➤ **mpicc** - O2 -o **MyProg MyProg . c**

OpenMP (Open Multi-Processing)

OpenMP (Open Multi-Processing)

OpenMP is a programming interface for parallel computing on shared memory architecture.

- **It allows you to manage:**

  the creation of light processes
  the sharing of work between these lightweight processes
  synchronizations (explicit or implicit) between all light processes
  the status of the variables (private or shared).

OpenMP (Open Multi-Processing)

- **Shared memory model**
  - ➢ Threads communicate by accessing shared variables

  - **The sharing is defined syntactically**
    - ➢ Any variable that is seen by two or more threads is shared
    - ➢ Any variable that is seen by one thread only is private

    - **Race conditions possible**
      - ➢ Use synchronization to protect from conflicts
      - ➢ Change how data is stored to minimize the synchronization

OpenMP (Open Multi-Processing)

- **Multicore CPUs are everywhere:**
  - ➤ Servers with over 100 cores today and more
  - ➤ Even smartphone CPUs have 8 cores

  - **Multithreading, natural programming model**
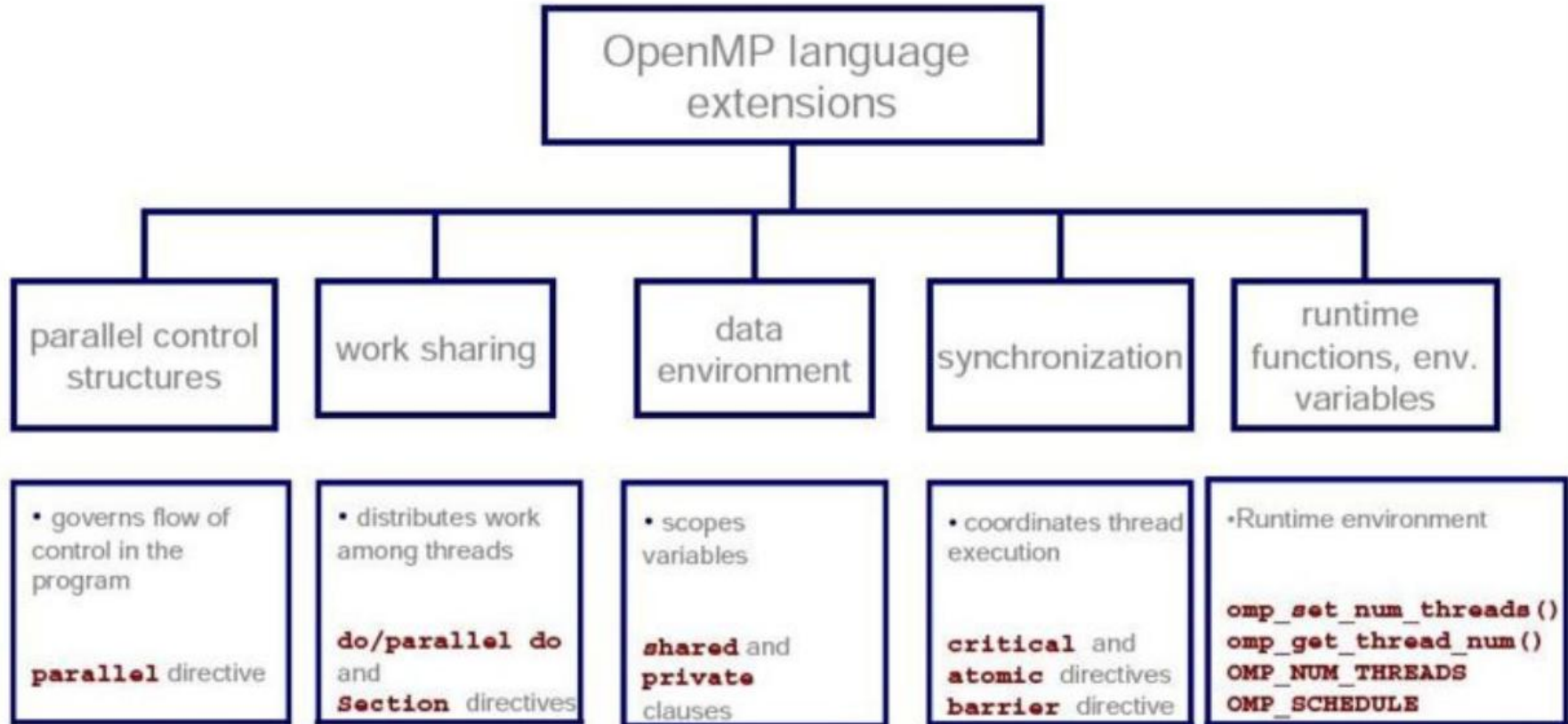   - ➤ All processors share the same memory
    - ➤ Threads in a process see same address space
    - ➤ Many shared-memory algorithms developed

    - **Multithreading is hard**
      - ➤ Lots of expertise necessary
      - ➤ Deadlocks and race conditions
      - ➤ Non-deterministic behavior makes it hard to debug

OpenMP: Architecture

OpenMP language extensions

parallel control structures	work sharing	data environment	synchronization	runtime functions, env. variables
• governs flow of control in the program **parallel** directive	• distributes work among threads **do/parallel do** and **Section** directives	• scopes variables **shared** and **private** clauses	• coordinates thread execution **critical** and **atomic** directives **barrier** directive	•Runtime environment **omp_set_num_threads()** **omp_get_thread_num()** **OMP_NUM_THREADS** **OMP_SCHEDULE**

OpenMP: Terminology and behavior

- **OpenMP Team** = Master + Worker

- **Parallel Region** is a block of code executed by all threads simultaneously (has implicit barrier)
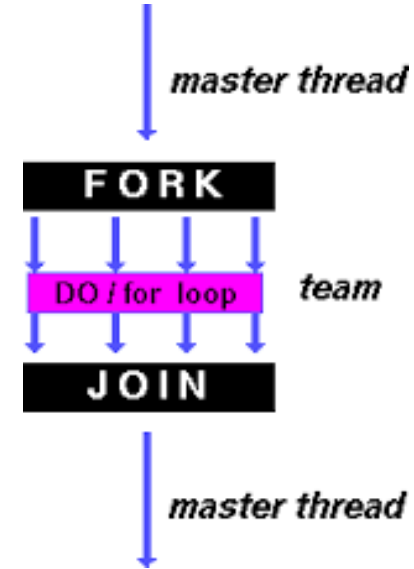 - The master thread always has thread id 0
  - Parallel regions can be nested
  - If clause can be used to guard the parallel region
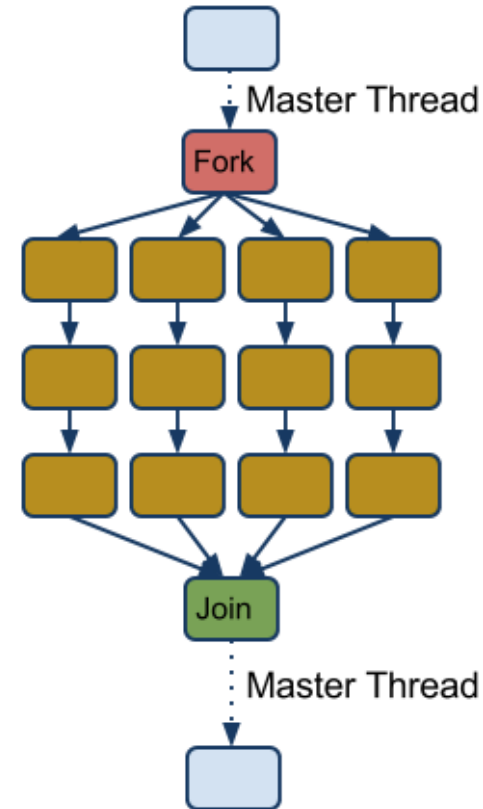
OpenMP: Terminology and behavior

- A **Work-Sharing construct** divides the execution of the enclosed code region among the members of the team. (Loop, Section etc.)

OpenMP: Preprocessor Directives

Preprocessor directives tell the compiler what to do.
Always start with #
You have already seen one:

```
#include <stdio.h>
```

OpenMP directives tell the compiler to add machine code for parallel execution of the following block.

```
#pragma omp parallel
```

OpenMP: Some OpenMP Subroutines

```
int omp_get_max_threads()
```

Return max possible

```
int omp_get_num_threads()
```

Returns number of treads in current team \\

```
int omp_get_thread_num()
```

Returns tread id of calling thread
Between () and omp_get_num_threads-1

OpenMP: Process vs. Thread

- MPI = Process, OpenMP = Thread

- Program start with a single process

- Process have their own (private) memory space

- A process can create one or more threads

- Threads created by a process share its memory space
  - Read and write to same memory adresses
  - Share same process ids and file descriptors

- Each thread has a unique counter and stack pointer
  - A tread can have private starage on the stack

OpenMP: Example

```c
#include <omp.h>   //<-- necessary header file for OpenMP API
#include <stdio.h>

int main(int argc, char *argv[]){

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP: Constructs

Parallel region

      Thread creates team, and becvomes master (id 0)

      All threads run code after

      Barrier at end of parallel section

```
#pragma omp parallel [clause ...]
                if (scalar_expression)
                private (list)
                shared (list)
                default (shared | none)
                firstprivate (list)
                lastprivate (list)
                reduction (operator: list)
                num_threads (integer)


    structured_block                          (not a complete list)
```

OpenMP Parallel Clauses

`#pragma omp parallel if (scalar_expression)`

Only execute in parallel.
Otherwise serial.

`#pragma omp parallel private (list)`

Data local to thread.
Value are not guaranteed to be defined on exit (even if defined before)
No storage associated with original object
Use firstprivate and/or lastprivate clause to override

`#pragma omp parallel firstprivate (list)`

Variables in list are private.
Initialized with the value the variable had before entering the construct.

`#pragma omp parallel for lastprivate (list)`

Only in for loops
Variables in list are private.
The thread that executes the sequentially last iteration updates the value of the variables in the list.

OpenMP Parallel Clause 3

```
#pragma omp shared (list)
```

Data is accessible by all threads in team.
All threads access samme address space.

Improperly scoped variables are big source of OMP bugs
- Shared when should be private
- Race condition

```
#pragma omp default (shared | none)
```

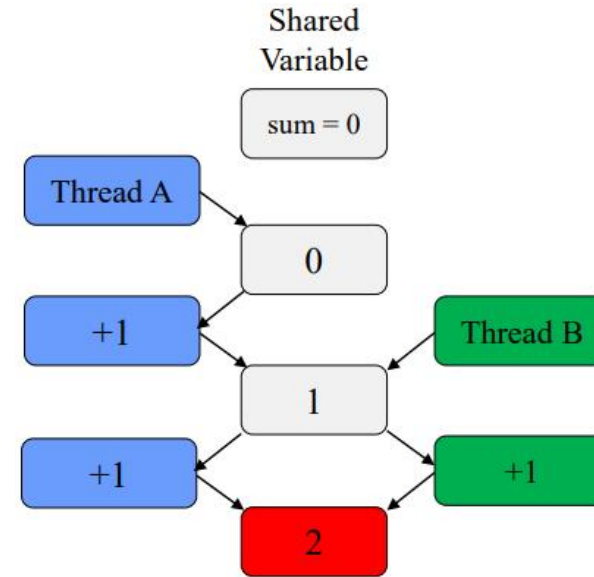Tip: Safest is to use default (none) and declare by hand.

OpenMP: Caution Race Condition

When multible threads simultaneously read/write
Multiple OMP solutions
- Reduction
- Atomic
- Critical

```
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```
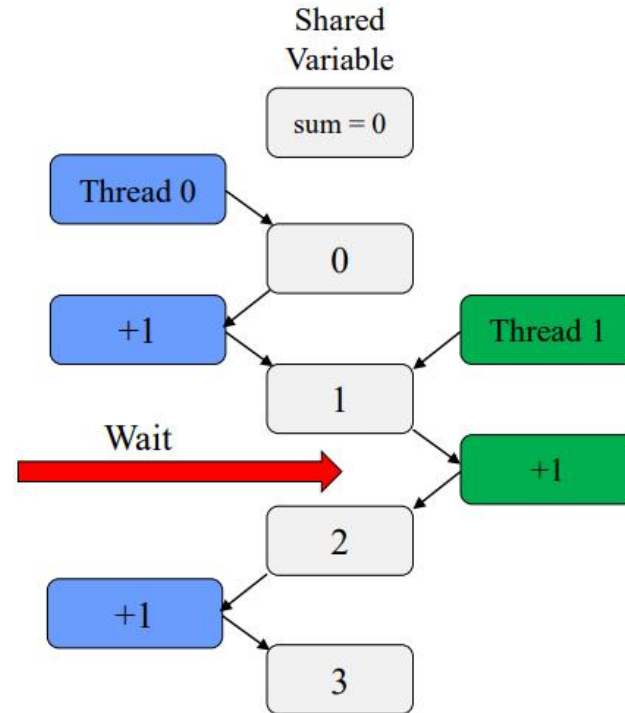

Should be 3!

OpenMP: Critical Section

One solution: use critical
Only one tread at a time can execute
 a critical section

```
#pragma omp critical
    {
        sum += i;
    }
```
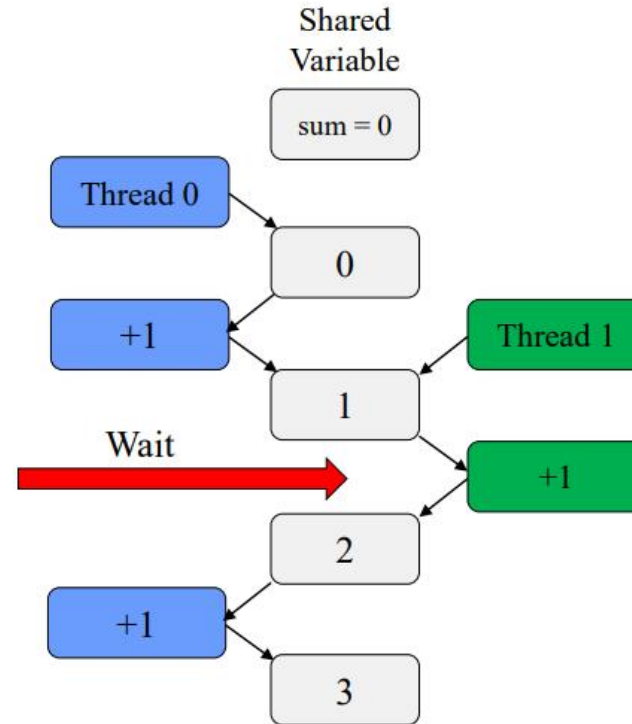
Downside ?
SLOOOOWWW
Overhead and serialization

OpenMP: Atomic

Atomics like "mini" critical
Only one line
      Certain limitations

```
#pragma omp atomic
    sum += i;
```

Hardware controlled
Less overhead tha critical

OpenMP Reduction

```
#pragma omp reduction (operator:variable)
```

Avoids race condition

Reduce variable must be shared

Makes variable private, then performs operator at end of loop

operator cannot be overloaded (c++)

> One of: +,*,-,/ (and &,^,|,&&,||)
>
> OpenMP 3.1: added min and max for c/c++

```c
#include <omp.h>
#include <stdio.h>

int main() {

  int i;
  const int N = 1000;
  int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
  for (i=0; i<N; i++) {
    sum += i;
  }

  printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

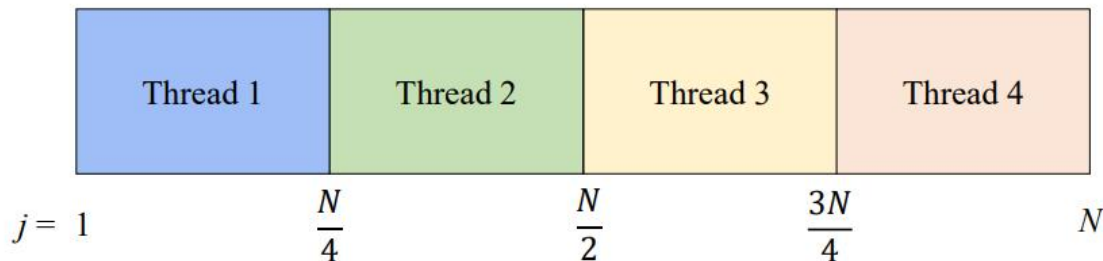# OpenMP: Scheduling omp for

How does a loop get split up ?
In MPI, we have to do it manually
If you do not tell what to do, the compiler decides
Usually compiler chooses "static" - chunks of N/p

```
#pragma omp parallel for default(shared) private(j)
  for (j=0; j<N; j++) {
        ... // some work here
}
```
Unspecified schedule

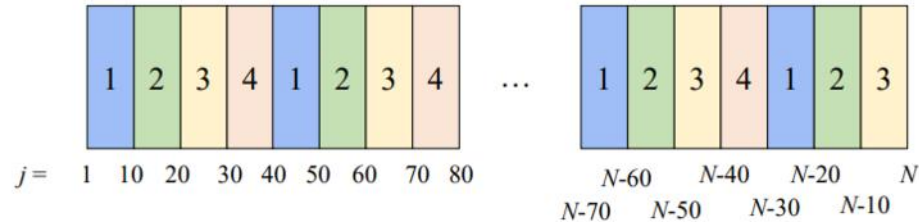| Thread 1 | Thread 2 | Thread 3 | Thread 4 |

$j = 1$ $\dfrac{N}{4}$ $\dfrac{N}{2}$ $\dfrac{3N}{4}$ N

You can tell the compiler what size chunks to take.

```
#pragma omp parallel for default(shared) private(j) schedule(static,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```


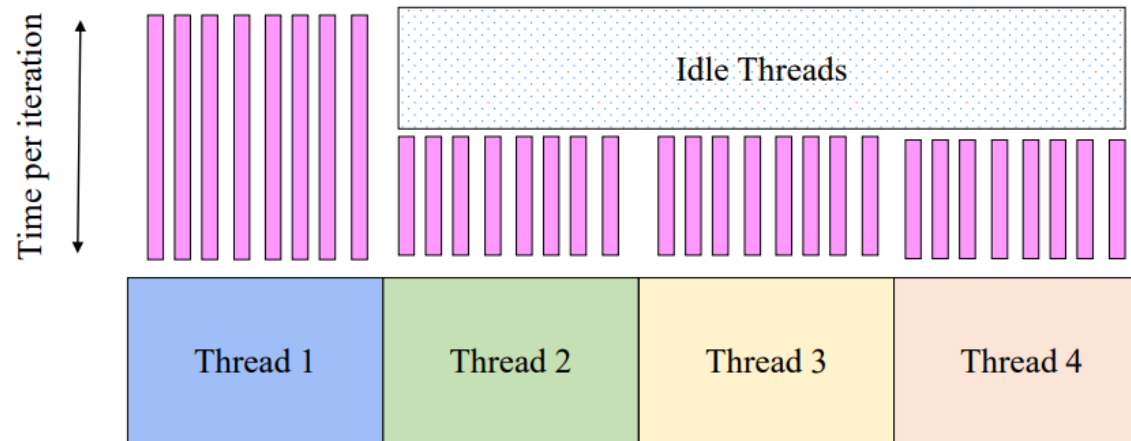Keeps assigning chunks until done.
Chunk size that is not a multiple of the loop will results in thread with uneven numbers.

OpenMP: Problem with Static Scheduling

What happens if loop iterations do not take the same amount of time ?
Load imbalance

OpenMP: Dynamic Scheduling

Chunks are assigned on the fly, as threads become available
When a thread finishes on chunk, it is assigned another

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

Caveat Emptor: higer overhead than static!

OpenMP omp for Scheduling Recap

```
#pragma omp parallel for schedule(type [,size])
```

Scheduling types

Static

-Chucks of specified size assigned round-robin

Dynamic

-Chunks of specified size are assigned when thread finishes previous chunk

Guided

-Like dynamic, but chunks are exponentially decreasing
-Chunk will not be smaller than specified size

Runtime

-Type and chunk determined at runtime via environment variables

OpenMP: API

- API for library calls that perform useful functions
- Must include "omp:h"
- Will not compile without OpenMP compiler support

```c
#include <omp.h>  //<-- necessary header file for OpenMP API
#include <stdio.h>

int main(int argc, char *argv[]){

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP: API

```
void omp_set_num_threads(int num_threads)
```

Sets number of treads used in next parallel section
Overrides OMP_NUM_THREADS environment variable
Positive integer

```
int omp_get_max_threads()
```

Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

Returns number of threads currently in team

```
int omp_get_thread_num()
```

Returns thread id of calling thread
Between 0 and omp_get_num_threads-1

```
double omp_get_wtime()
```

Returns number of secons since some poin
Use inpais time=(t2-t1)

OpenMP: Performance Tips

- Avoid serialization!
- Avoid using `#pragma omp parallel for` before each loop
  - Can have significant overhead
    - Thread creation and scheduling is NOT free!!
  - Try for broader parallelism
    - One `#pragma omp parallel`, multiple `#pragma omp for`
  - Always try to parallelize the outer most loop
- Use reduction whenever possible
- Minimize I/O
- Minimize `critical`
  - Use `atomic` instead of `critical` where possible
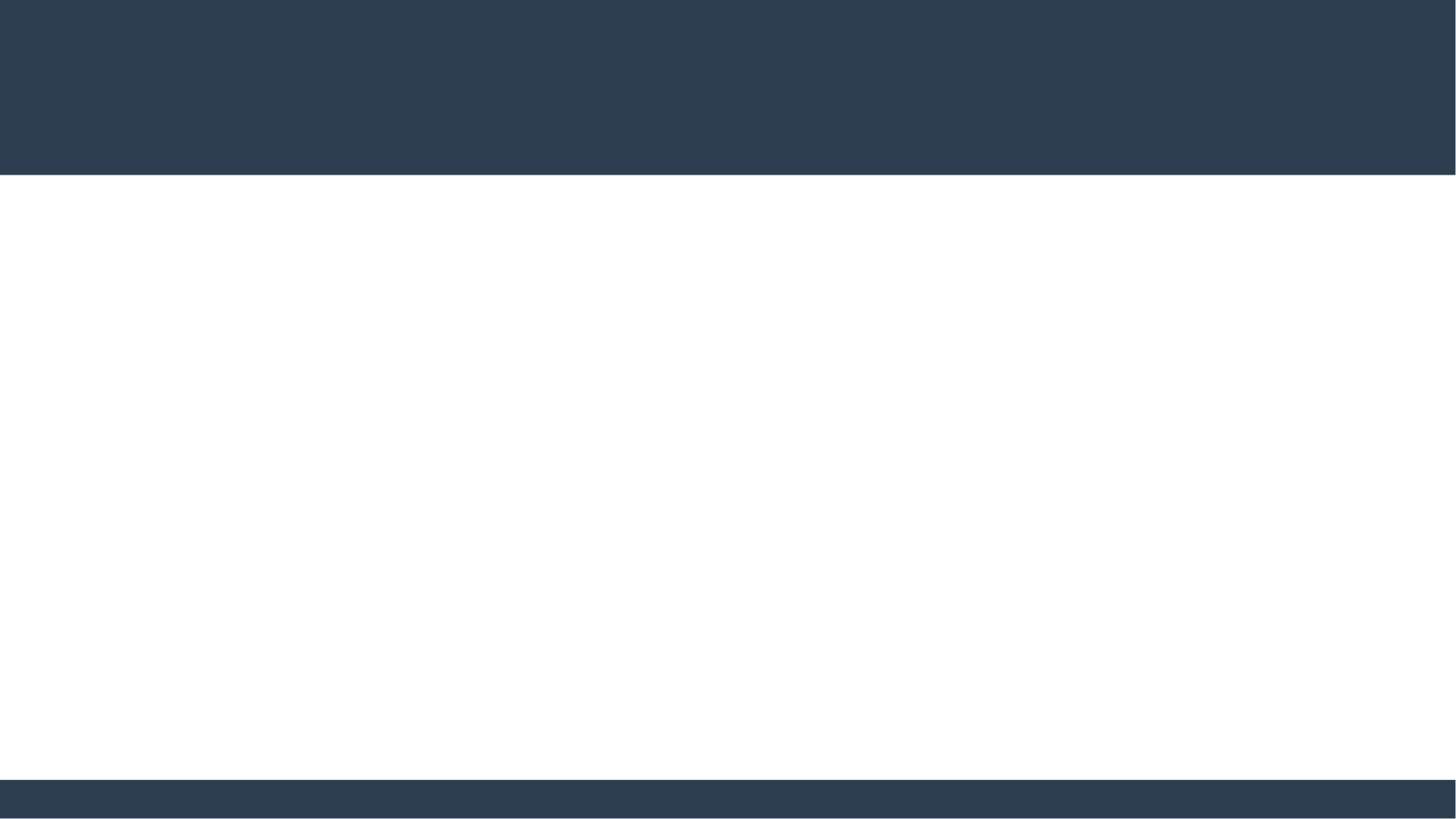
Hybrid OpenMP-MPI

Hybrid MPI and OpenMP

Hybrid application programs using **MPI + OpenMP** are now common place on large HPC systems.

There are basically two main motivations:

1. Reduced memory footprint, both in the application and in the MPI library (eg communication buffers).

2. Improved performance, especially at high core counts where pure MPI scalability runs out.

A common hybrid approach
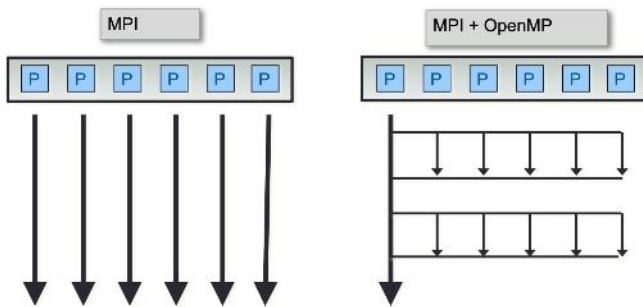
- From dequential code, alongside MPI first, then try adding OpenMP

- From MPI code, add OpenMP

- From OpenMP code, treat as serial code

- The simplest and least error-prone method is to use MPI outside the parallel region and allow only the master thread to communicate between MPI tasks.

- Could use MPI in parallel region with thread-safe MPI.

Hybrid MPI and OpenMP

- Two-level Parallelization
  - ➤ Mimics hardware layout of cluster
  - ➤ Only place this really make sense
  - ➤ MPI between nodes
  - ➤ OpenMP within shared-memory nodes

- Why ?
  - ➤ Saves memory by not duplicating data
  - ➤ Minimize interconnect communication by only having 1 MPI process per node

- Careful of MPI calls within OpenMP block
- ➤ Safest to do MPI calls outside (but not required)

  Obviously requires some thought!

Hybrid Programming

In hybrid programming each process can have multiple threads executing simultaneously
All threads within a process share all MPI objects Communicators, requests, etc.

MPI defines 4 levels of thread safety:

➢ **MPI_THREAD_SINGLE** : One thread exists in program

➢ **MPI_THREAD_FUNNELED** : Multithreaded but only the master thread can make MPI calls Master is one that calls MPI_Init_thread()

➢ **MPI_THREAD_SERIALIZED:** Multithreaded, but only one thread can make MPI calls at a time

➢ **MPI_THREAD_MULTIPLE:** Multithreaded and any thread can make MPI calls at any time. Use MPI_Init_thread instead of MPI_Init if more than single thread

Hybrid Programming

Safest (easiest) to use MPI_THREAD_FUNNLED

- Fits nicely with most OpenMP models
- ➢   - Expensive loops parallelized with OpenMP
- ➢   -Communication and MPI calls between loops
- Eliminates need for true "thread-safe" MPI

- Parallel scaling efficiency may be limited (Amdahl's law) by MPI_THREAD_FUNNLED approach

- Moving to MPI_THREAD_MULTIPLE does come at a performance price (and programming challenge)

Hybrid Programming Example

```
Program hybrid
call MPI_INIT (ierr)
call MPI_COMM_RANK (...)
call MPI_COMM_SIZE (...)
... some computation and MPI
communication
... start OpenMP within node
!$OMP PARALLEL DO PRIVATE(i)
!$OMP&              SHARED(n)
  do i=1,n
     ... computation
  enddo
!$OMP END PARALLEL DO
... some computation and MPI
communication
call MPI_FINALIZE (ierr)
end
```

Start with MPI Initialization

Create OMP parallel regions with MPI task (Process):

 - Sereal Regionds are the master head or MPI task .

 - MPI rank is know to all thread.

Call MPI library in serial and parallel regionds.

Finalize MPI

Hybrid Programming Example
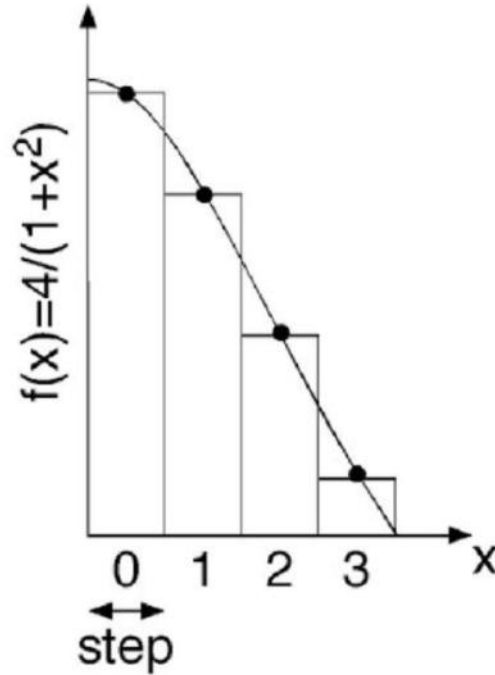
- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta \ (i = 0,\ldots,N-1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```


$f(x)=4/(1+x^2)$

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>                  /* MPI header file */
#include <omp.h>                  /* OpenMP header file */
#define NUM_STEPS 100000000
#define MAX_THREADS 4

int main(int argc, char *argv[]) {
  int nprocs, myid;
  int tid, nthreads, nbin;
  double start_time, end_time;
  double pi, Psum=0.0, sum[MAX_THREADS]={0.0};
  double step = 1.0/(double) NUM_STEPS;

  /* initialize for MPI */
  MPI_Init(&argc, &argv);         /* starts MPI */
  /* get number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  /* get this process's number (ranges from 0 to nprocs - 1) */
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  nbin= NUM_STEPS/nprocs;
```

Hybrid Programming Example

```c
#pragma omp parallel private(tid)
{
    int i;
    double x;
    nthreads=omp_get_num_threads();
    tid=omp_get_thread_num();
    for (i=nbin*myid+tid; i < nbin*(myid+1); i+= nthreads) { /* changed*/
        x = (i+0.5)*step;
        sum[tid] += 4.0/(1.0+x*x);
    }
}
for(tid=0; tid<nthreads; tid++)   /*sum by each mpi process*/
    Psum += sum[tid]*step;

MPI_Reduce(&Psum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

if (myid == 0) {
    printf("parallel program results with %d processes:\n", nprocs);
    printf("pi = %g  (%17.15f)\n",pi, pi);
}
MPI_Finalize();

return 0;
}
```
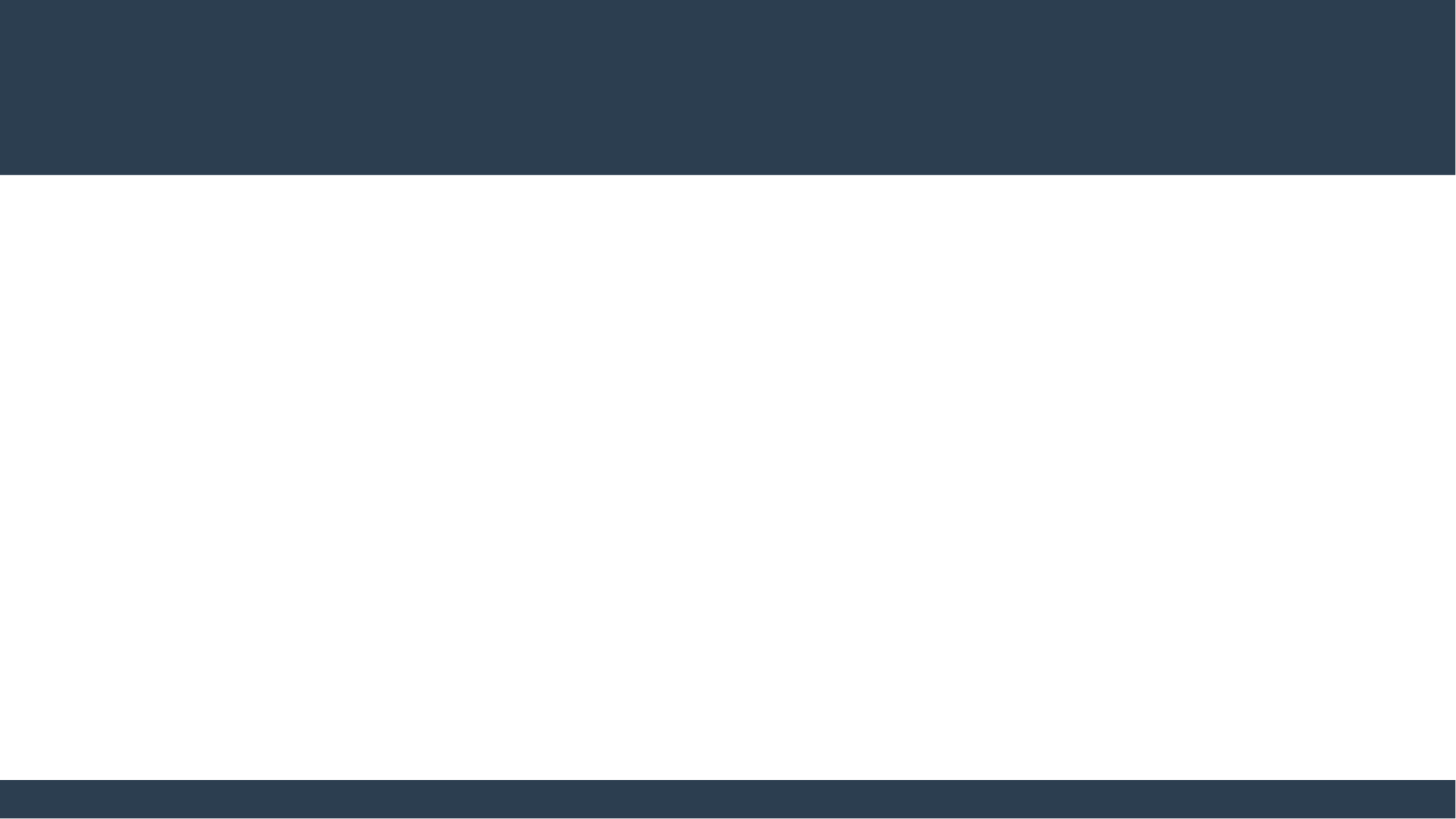
Results

- **MPI**
  MPI uses 8 processes:
  pi = 3.14159 (3.141592653589828)

- **OpenMP**
  OpenMP uses 8 threads:
  pi = 3.14159 (3.141592653589882)

- **Hybrid**
  mpi process 0 uses 4 threads
  mpi process 1 uses 4 threads
  mpi process 1 sum is 1.287 (1.287002217586605)
  mpi process 0 sum is 1.85459 (1.854590436003132)
  Total MPI processes are 2
  pi = 3.14159 (3.141592653589738)

SPECX

양

SPECX

SPECX

- Shares many similarities with StarPU.

- Written in modern C++ (20).

- Task-based execution system.

- Able to also support speculative execution, which is the ability to execute tasks ahead of time if others are unsure about changing the data.

StarPU

- StarPU is a task scheduling library for hybrid architectures.

- Design systems in which applications are distributed across the machine, feeding all available resources into parallel tasks.

- Optimized heterogeneous scheduling, cluster communication, data transfers and replication between main memory and discrete memories

SPECX

Workflow

Execution interface: Provides functionality for creating tasks, task graphs and generating traces. Can be used to specify speculation model.

Data Dependency Interface: Forms a collection of objects that can be used to express data dependencies. Also provides wrapper objects that can be used to specify whether a given callable should be considered CPU or GPU code.

Task visualization interface: Specifies the ways to interact with the task object.

SPECX
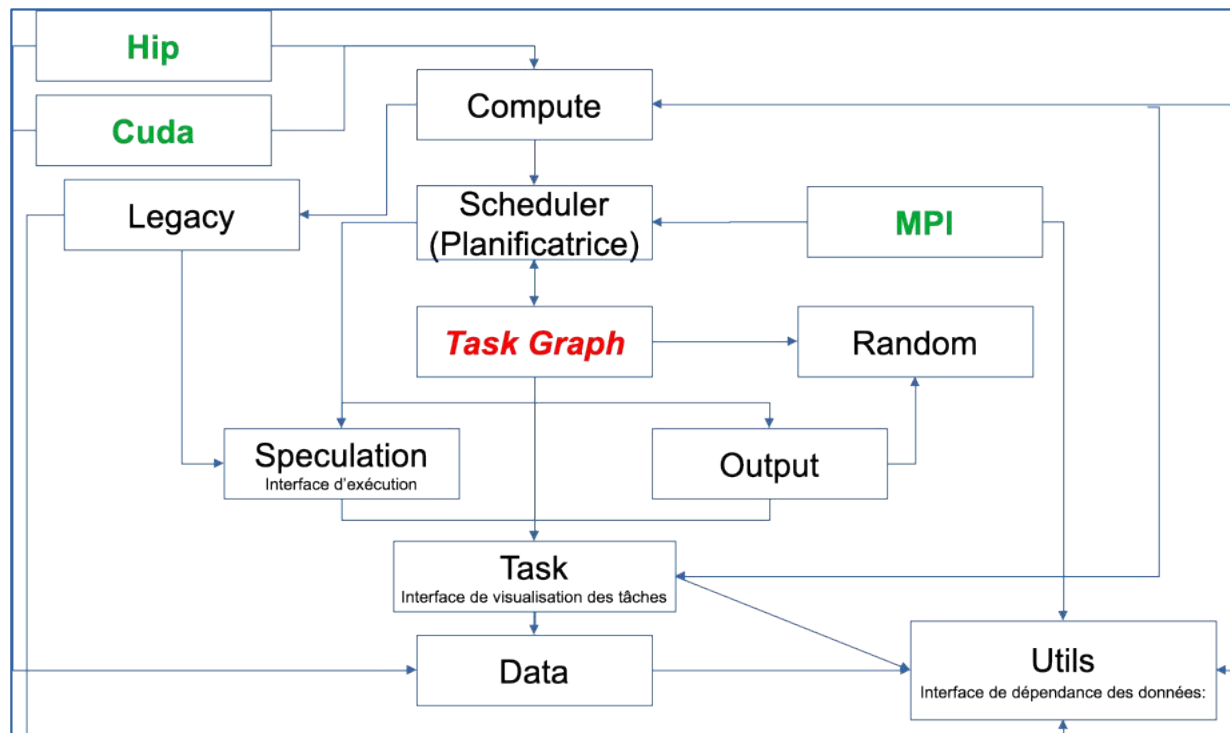
Future developments

- The main objective is to reduce the calculation times,

- To manage the use of the different calculation resources, the different typical workloads, in particular in the case of multicore machines equipped with several acceleration machines.

- **Plan to separate thread management from execution.** To change the prototype of the predicate, to be able to consider additional data or different to make the decision.

- Develop decision graphs to optimize available hybrid resources (CPU, GPU, GPGPU, TPU,...) to increase computational speed for given problems.

- To provide effective and high -performance tools to the user.

Thank you for your attention !