Shared memory parallel systems with OpenMP

Sunday, 9 March 2025 2:09 pm

- System architecture
  - Single core
  - Multi core
- Sequential program execution
  - Cores are kept idle hence waste of resources
- Parallel computing
  - Multiple threads share a single a single address space
  - Changes in one process is reflected to all other processes
  - Implicit communication
    - Data exchange between threads or processes without programmer defining
  - Explicit synchronization
    - Programmer must ensure that computations are syncnhronized to avoid race conditions or inconcistent results

OpenMP (open multi processing)

- Framework for parallel programming in shared memory systems
- Specification for
  - Directives
  - Runtime library routines
  - Environment variables

GOALS OF OpenMP

- Standardization
  - Provides a standard interface across different shared memory architectures
- High level thread programming
  - Makes parallel programming easier compared to low level thread libraries
- Multi-vendor and multi OS support
  - Works across various systems like Unix,Windows etc

Explicit parallelism

- Programmer is responsible for explicitly creating and managing threads

Programmer directed

- Programmer gives high level directives to indicate parallelism and compiler runtime handles thread managements
- e.g.: #pragma omp parallel

SHARED MEMORY PROGRAMMING

- Pthreads
- c++ threads
- openMP

USER INTERFACE MODEL

- Shared memory+ thread based parallelism
- Extends existing language using
  - Compiler directives: used by programmers to define and control paralle regions of code
  - Library calls: functions from openMP library
  - Environment: used to control behaviour of openMP programs

- Programmer defined parallelization

SYNTAX

- Parallelism is highlighted using compiler directives or pragmas
  #pragma omp *construct [clause [clause]…]*
- If we run code without using openMP, then also the code will execute but sequentially

FORK/ JOIN EXECUTION MODEL

- Initially, in an openMP program, a single thread (master thread) runs immediately after startup
- Master thread forks a number of additional threads(team) when entering a parallel region
- Inside a parallel region, master thread and the forked threads execute instruction streams concurrently
  - Each thread has a unique ID
  - Different threads work on different parts of the shared data
  - openMP has compiler directives for dividing work among the threads
- End of parallel region, all the threads are joined/terminated until next parallel region starts

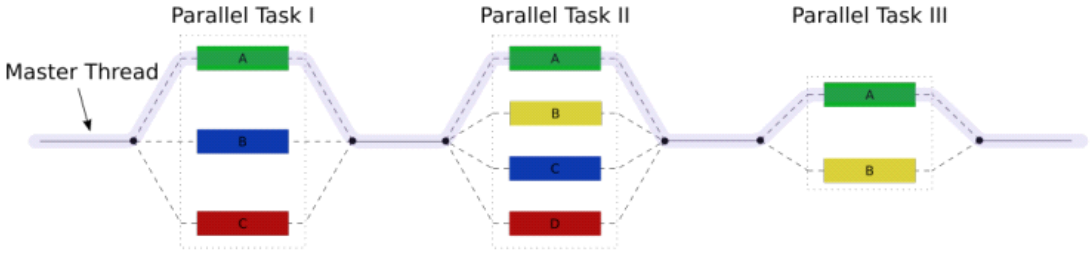
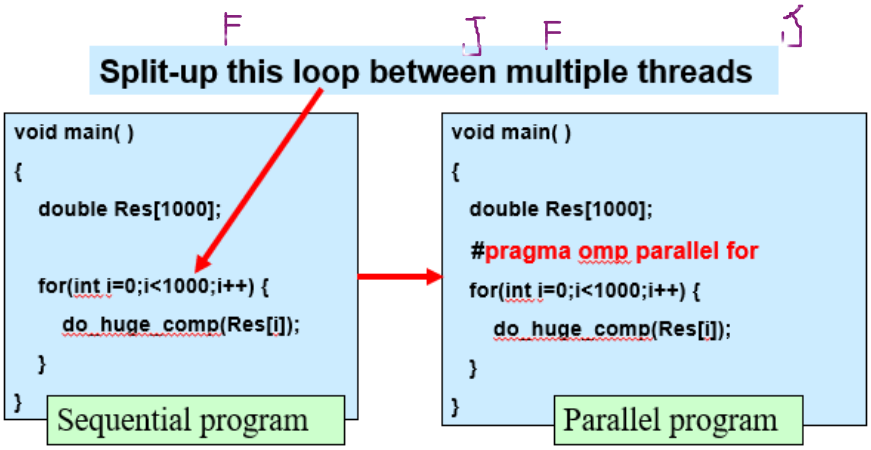


Figure 11.1: The fork-join model

- openMP is usually used to parallelize loops
  - Firstly finding out most time consuming loops
  - Then splitting them amongst threads



DIRECTIVES

- Creates parallel regions
- Dividing blocks of code among threads
- Distributes loop iterations
  #pragma omp *directive-name [clause [clause]…]*
  #pragma omp parallel private(var)

VALID CLAUSES

- If(expression): when if condition is met
- Num_threads(N): specifies number of threads

- Private(list): variables are private to each thread
- Shared(list): variables shared among all threads
- Reduction(operator:list): reduces value across threads

CONSTRUCTS

1. Parallel regions: creates parallel execution blocs
2. Work sharing: splitting workload across threads
3. Data environment: shared/private data
4. Synchronization: thread safety
5. Runtime functions/environment variables: controlling openMP behaviour

***PARALLEL* DIRECTIVE**

- Parallel region is a block of code that will be executed by multiple threads
- If a program is serial initially and it finds a PARALLEL directive, a team of threads is created and main thread becomes the master thread which was previously the serial execution thread
- Master thread id is 0

HOW MANY THREADS?

- Determined by following factors in order of precedence
  - If clause
  -
```
#pragma omp parallel if (scalar_expression)
```
  - Num_threads clause
```
#pragma omp parallel if(np>1) num_threads(np)
{
    …
}
```
  - Omp_set_num_threads() functions
```
#define TOTAL_THREADS 8
int main( )
{
    omp_set_num_threads(TOTAL_THREADS);
    #pragma omp parallel
    {
        . . .
    }
. . .
```
  - Omp_num_threads environment variable
```
$ export OMP_NUM_THREADS=4
$ echo $OMP_NUM_THREADS
```
  - Number of CPU cores
- Threads are numbered from 0 to n-1

SHARED AND PRIVATE DATA

- Shared
  - Accessible by all threads
  - Default is shared
  - Final value will be updated by the last thread leaving the region

    - Race condition may occur
- Private
    - each thread gets its own copy
    - Separate 'stack memory' for each thread data
    - Use *firstprivate or lastprivate* to override
    - Firstprivate: initialized with the value the variable had before entering the construct
    - Lastprivate: used in for loops, thread that executes in last iteration updates the value

WORK SHARING CONSTRUCTS

- Distribute task amongst threads within the same team
- Used for loops
- *Schedule* clause describes how iterations of the loops are divided among threads
- Static
    - fixed chunk size assigned round robin
    - Causes load imabalance as some threads are kept idle for a long time
- Dynamic
    - assigns chunks as thread finishes
    - More overhead is created
- Guided
    - Chunks shrink over time
    - If chunksize is not specified it decreases to 1, otherwise to chunksize

CRITICAL SECTION PROBLEM

- Ensures mutual exclusion when modifying shared variables
- One thread in the critical section at a time

SYNCHORNIZATION CONSTRUCTS

- *Critical* directive specifies a region of code that must be executed by only one thread at a time
- If another thread tries to attempt to execute in CS, it will be blocked until previous thread is released from CS
  #pragma omp critical [name]
- *Master* directive specifies a region that only master thread can execute and no other thread
  #pragma omp master
- *Barrier* directive means all thread must reach this point before moving forward
  #pragma omp barrier

REDUCTION (Data sharing attribute clause)

- Performs a reduction operation on the variables that appear in the list
- Private copy for each list variable is created and initialized
- Used for accumulating results
  #pragma omp operator: list
  operator can be +, -, *, &&, ||, max, min ...

MPI Basics

Friday, 4 April 2025 2:38 pm

Distributed systems

- Software components on networked computers corrdinate actions by message passing to achieve a common goal
- e.g:online retail system

History of MPI

- Before MPI, parallel computing lacked a standard, leading to incompatible implementations

Message passing model

- Separate processes execute on different computers and communicate by sending and receiving messages
- A process is a program counter and address space; may have multiple threads

Types of parallel computing models

- Data parallel
  - Same operation on multiple data items
- Task parallel
  - Different operations on different data
- SPMD
  - Single program,multiple data not synchronized at the instruction level

Message passing programming paradigm

- How each processor runs an instance of the same program but operates on different data with the same name which are local to each process
- Communicate via message passing
- e.g: parallel quicksort algorithm sort parts of an array but communicate their results at the end

MPI fundamentals

- Communicator (MPI_COMM_WORLD) defines a group of processes
- Each process have a different rank that is also an ID of the process
- Same process might have different ranks in different communicators
- Starts with 0 and ends with size-1

MPI library

int MPI_Init(int *argc, char ***argv)

- Initializes MPI environment
- Called by *main thread*

int MPI_Finalize( )

- Must be called at the end of computation by *main thread*
- Performs clean up tasks

MPI_Comm( )

- Group of processes that communicate

MPI_COMM_WORLD( )

- Root communicator

int MPI_Comm_size (MPI_Comm comm, int *size)

- Number of processes

int MPI_Comm_rank(MPI_Comm comm, int *rank)

- Index of the calling process

MPI_Get_processor_name(processor_name,&name_len)

- Gets the actual name of the processor on which process is executing its job

Point to point communcation

- Messages are sent from a source process to a destination process, using ranks for identification

- Communication takes place within a communicator
- To send a message, sender provides a rank of the process and a unique tag to identify message, and type of data
- A message can be received by a given tag(not necessarily), type of data(rough guess, upper bound)
- Sending buffer gets blocked while its copying its data to sysbuf and receiving buffer gets blocked when recvbuf gets valid data from sysbug

```
MPI_Send(void* data, int count, MPI_Datatype
type, int dest, int tag, MPI_Comm comm)
```

- Data: pointer to data
- Count: number of elements or size
- Type: data type
- Dest: destination process rank
- Tag: identifier
- Comm: communicator

```
MPI_Recv(void* data,int count,MPI_Datatype type, int
source,int tag,MPI_Comm comm,MPI_Status* status)
```

- Data: pointer to data
- Count: number of elements
- Type: data type
- source: source process
- Tag: identifier
- Comm: communicator
- Status: sender, tag,message size (MPI_STATUS_IGNORE used if no additional information is required)

```
Int MPI_Get_count(MPI_Status* status,MPI_Datatype
type,int* count)
```

- Count: number of elements that were received

Non blocking point to point communication

- Sendbuf is not blocked when copying data to sysbuf
- And recvbuf also not blocked when getting data from sysbuf
- MPI_WAIT uses request as an argument and block until communication is complete
- MPI_TEST uses request as an argument and checks for completion (non-blocking)
  - No deadlocks

*MPI_Probe(int **source**, int **tag**, MPI_Comm **comm**,*
MPI_Status **status**)*

```
MPI_WAIT (request, status)
MPI_TEST (request, flag, status)
```

MPI collective communication

Saturday, 5 April 2025 2:45 am

Collective communications

Involves exchange of data among multiple processes in a parallel computing environment

- Broadcast
- Scatter
- Gather
- Reduce

Reductions

- Sum,max

These operations must be executed by all processes

These operations do not return control until all processes have completed their communication

1. **Broadcast**
   a. One process to many other processes
   b. Root process sends data to all processes in a communicator

   MPI_Bcast(void buf, int **count**,MPI_Datatype datatype,int root, MPI_Comm **comm**)*

   - Buf: address of send/receive
   - Count: number of elements
   - Dtype: data type
   - Root: sender
   - The senders data is copied to all processes including itself
2. **Scatter**
   a. Divides the data from the single source and sends different chunks of the data to different processes

   MPI_Scatter(void sendbuf, int send**count**,MPI_Datatype datatype,void* recvbuf, int recvcount,MPI_Datatype datatype,int root, MPI_Comm **comm**)*

   - Sendcount: number of elements sent to each process
   - Recvcount: number of elements to be received by a process
   - Each process receives a portion of the original data array
   - Useful for splitting large datasets
3. Gather
   a. Inverse of scatter
   b. Collects data from multiple processes and gathers it into one process, typically the root process

MPI_Gather(void sendbuf, int send**count**,MPI_Datatype datatype,void* recvbuf, int recvcount,MPI_Datatype datatype,int root, MPI_Comm **comm**)*

- Sendcount: number of elements sent to root process
- Recvcount: number of elements to be received by each process
- Root process stores data in order by the process number of the senders

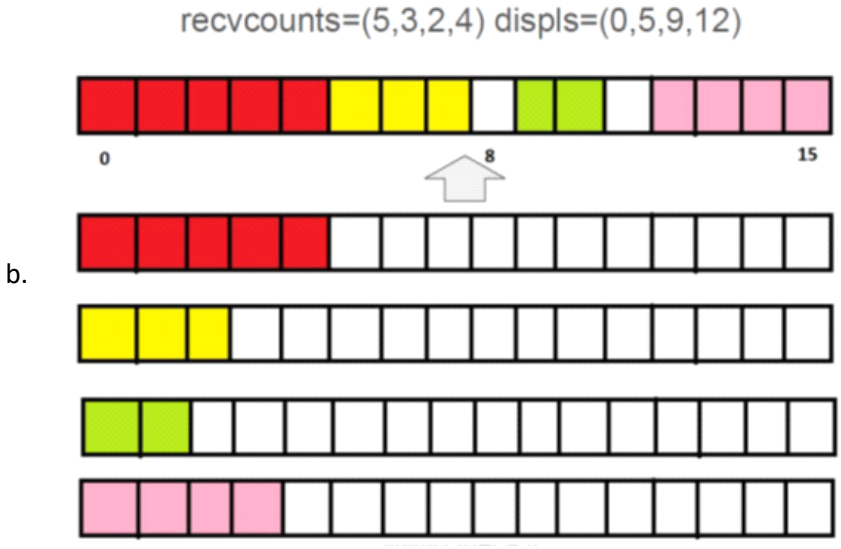
4. **MPI_Scatterv**
   a. Each process may receive a different number of elements
   b. Used when different processes need different ammounts of data

MPI_Scatterv(const void sendbuf, const int send**counts[]**,const int displs[],MPI_Datatype sendtype,void* recvbuf, int recvcount,MPI_Datatype datatype,int root, MPI_Comm **comm**)*

- Sendbuf: address of send buffer
- Sendcount: integer array specifying number of elements to send to each process
- Displs: integer array. (i) Specifies displacement
- Recvcount: number of elements to be received by each process
- Recvtype: datatype of receiver buffer
- Root: rank of sending process

5. MPI_Gatherv
   a. Root process can receive variable sized data from each process

b.



MPI_Gatherv(const void sendbuf, int*

*send**counts**,MPI_Datatype sendtype,void* recvbuf, const int recvcount[],const int displs[],MPI_Datatype datatype,int root, MPI_Comm **comm**)*

- Sendbuf: address of send buffer
- Sendcount: number of elements in sendbuf
- Displs: integer array. (i) Specifies displacement relative to recvbuf
- Recvcount: integer array containing number of elements that are to be received from each process
- Recvtype: datatype of receiver buffer
- Root: rank of receiving process

6. **Allgather and Allgatherv**
   - **MPI_Allgather** and **MPI_Allgatherv** are similar to **MPI_Gather** and **MPI_Gatherv**, but the results are made available to all processes in the communicator, not just the root.
   - This ensures that all processes receive the data gathered from other processes, which is useful for sharing results across all processes.
7. **Alltoall and Alltoallv**
   - **MPI_Alltoall** performs a process-to-process communication, where each process sends data to every other process in the communicator.
   - **MPI_Alltoallv** allows different data sizes to be sent to different processes, much like **MPI_Scatterv** and **MPI_Gatherv**.
   - This operation is useful for complex data redistributions like matrix transposition.

SYNCHRONIZATION

1. **Barrier**
   a. **MPI_Barrier** is a synchronization operation where all processes in the communicator are blocked until every process has reached the barrier
   b. All processes are synchronized before continuing which is useful in parallel algorithms that require global coordination before moving forward
   c. If all the processes in a communicator do not call barrier, then the program is a deadlock

Int MPI_Barrier(MPI_COMM COMM)

REDUCTION

- Involves combining data from multiple processes into a smaller set, often for summing numbers or taking max value
- e.g.: taking sum of array or product of array
- The result is only available at the root process or at all processes
- Scalar reduction involves reducing individual scalar values across processes
- Array reduction reduces elements in an array across processes, performing operation element wise
- Count,op, root have to be equal in all processes

Int MPI_Reduce(void sbuf, void* rbuf,int*

count, MPI_Datatype dtype,MPI_Op op, int root,MPI_COMM COMM)

- sbuf: address of send buffer
- rbuf: address of receive buffer
- Count: number of elements in send buffer
- Op: operation

MPI_Allreduce

- **MPI_Allreduce** is similar to **MPI_Reduce**, but it makes the result available to all processes, not just the root process.
- This is useful when all processes need the result of the reduction operation, for example, in iterative algorithms where the result is used in subsequent steps

Practice tasks

Sunday, 6 April 2025 11:34 pm

**The following program calculates the sum of numbers from 1 to 1000 in a parallel
fashion while executing on all the cluster nodes and providing the result at the end on
only one node. It should be noted that the print statement for the sum is only executed
on the node that is ranked zero (0) otherwise the statement would be printed as much
time as the number of nodes in the cluster. #include<iostream.h>**

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

int main(int argc, char **argv)
{
   int mynode, totalnodes;
   int sum = 0, startval, endval, accum;
   MPI_Status status;

   // Initialize MPI
   MPI_Init(&argc, &argv)
   MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
   MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

   // Get timestamp for the start of the parallel computation
   double start_time = MPI_Wtime();

   // Divide the range 1 to 1000 across all nodes
   startval = 1000 * mynode / totalnodes + 1;
   endval = 1000 * (mynode + 1) / totalnodes;

   // Compute local sum
   for (int i = startval; i <= endval; i++) {
      sum += i;
   }

   // If the node is not the root, send its local sum to node 0
   if (mynode != 0) {
        MPI_SEND(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD)
   } else {
      // If the node is root, gather the partial sums from all other nodes
      int total_sum = sum;
      for (int j = 1; j < totalnodes; j++) {
```

```
            MPI_Recv(&accum, 1, MPI_INT, j, 1, MPI_COMM_WORLD, &status);
            total_sum += accum;
        }

        // Print the final sum
        printf("The sum from 1 to 1000 is: %d\n", total_sum);
    }

    // Get timestamp for the end of the parallel computation
    double end_time = MPI_Wtime();
    double execution_time = end_time - start_time;

    // Print execution time at each node
    if (mynode == 0) {
        printf("Execution Time on Node %d: %f seconds\n", mynode, execution_time);
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Suppose you are in a scenario where you have to transmit an array buffer from all other nodes to one node by using send/ receive functions that are used for intra- process synchronous communication. The figure below demonstrates the required functionality of the program.

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

#define ARRAY_SIZE 10

int main(int argc, char *argv[]) {
    int rank, size;
    int sendbuf[ARRAY_SIZE];  // Buffer to send data
    int recvbuf[ARRAY_SIZE * 3];  // Buffer to receive the complete array (assuming 3 nodes)

    MPI_Init(&argc, &argv); // Initialize MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get rank of the process
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get total number of processes

    if (size < 2) {
        printf("At least two processes are required\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
```

```
    }

    // Initialize sendbuf with some values (e.g., rank value)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sendbuf[i] = rank * ARRAY_SIZE + i;  // Fill with unique values based on rank
    }

    if (rank == 0) {
        // Root process receives data from all other processes
        for (int i = 1; i < size; i++) {
            MPI_Recv(recvbuf + i * ARRAY_SIZE, ARRAY_SIZE, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }

        // Print received array at root
        printf("Root process (rank 0) received the following array:\n");
        for (int i = 0; i < size * ARRAY_SIZE; i++) {
            printf("%d ", recvbuf[i]);
        }
        printf("\n");

    } else {
        // Other processes send their data to the root process
        MPI_Send(sendbuf, ARRAY_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();  // Finalize MPI environment
    return 0;
}
```

Write a program in which every node receives from its left node and sends message to its right node simultaneously as depicted in the following figure

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

#define TAG 0

int main(int argc, char **argv) {
    int rank, size;
    int data = 0, received_data = 0;

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get rank of the process
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get total number of processes

    // Each node initializes its data
    data = rank + 1;  // Assigning a value to data (e.g., rank + 1 for simplicity)

    // The node receives data from its left neighbor (if it's not the first node)
    if (rank != 0) {
        MPI_Recv(&received_data, 1, MPI_INT, rank - 1, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Node %d received data %d from node %d\n", rank, received_data, rank - 1);
    }

    // The node sends data to its right neighbor (if it's not the last node)
    if (rank != size - 1) {
        MPI_Send(&data, 1, MPI_INT, rank + 1, TAG, MPI_COMM_WORLD);
        printf("Node %d sent data %d to node %d\n", rank, data, rank + 1);
    }

    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

2. Write a program to calculate prefix sum Sn of ‘n’ numbers on ‘n’ processes

• Each node has two variables ‘a’ and ‘b’. • Initially a=node id

• Each node sends ‘a’ to the other node. • ‘b’ is a variable that receives a sent from another node.

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

int main(int argc, char *argv[]) {
    int rank, size;
    int a, b;  // Variables for each process (a: send, b: receive)

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get rank of the process
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get the total number of processes

    // Each process initializes its own 'a' with its rank (or ID)
    a = rank;  // 'a' is initialized with the node ID

    // Initially, b is set to 0 in all processes (for receiving the value from other processes)
    b = 0;

    // Send 'a' to the next process, and receive 'a' from the previous process
    if (rank > 0) {
```

```
    MPI_Recv(&b, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  // Receive 'a' from previous node
  }

  // Update 'a' with the received 'b' (prefix sum logic)
  a = a + b;

  // Send the updated value of 'a' to the next node
  if (rank < size - 1) {
    MPI_Send(&a, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);  // Send 'a' to the next node
  }

  // The last process (rank size-1) will print the final result
  if (rank == size - 1) {
    printf("The prefix sum at node %d is: %d\n", rank, a);
  }

  // Finalize MPI
  MPI_Finalize();
  return 0;
}
```

Write a MPI parallel program that calculates the value of PI using integral method.

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

double f(double x) {
  return 4.0 / (1.0 + x * x);  // The function 4 / (1 + x^2)
}

int main(int argc, char *argv[]) {
  int rank, size;
  int n;  // Number of intervals
  double local_sum = 0.0, total_sum = 0.0;
  double start_time, end_time;

  // Initialize MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Master process (rank 0) asks for the number of intervals and broadcasts it to other processes
  if (rank == 0) {
    printf("Enter the number of intervals: ");
    scanf("%d", &n);
  }
```

```
    // Broadcast the number of intervals to all processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Start the timer to measure execution time
    start_time = MPI_Wtime();

    // Calculate the local sum for each process
    int i;
    double x;
    for (i = rank; i < n; i += size) {
        x = -0.5 + (i + 0.5) / n;  // x_i = -1/2 + (i + 0.5) / n
        local_sum += f(x);  // Add the value of f(x) to local sum
    }

    // Use MPI_Reduce to gather the local sums and sum them up to get the total sum
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // Root process calculates and prints the result
    if (rank == 0) {
        double pi_approximation = total_sum / n;  // The approximation of pi
        printf("The approximated value of PI is: %.15f\n", pi_approximation);

        // Stop the timer and print execution time
        end_time = MPI_Wtime();
        printf("Execution Time: %.6f seconds\n", end_time - start_time);
    }

    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

cuda

Tuesday, 13 May 2025 3:53 pm

Accelerator

- Hardware device or a software program with a main function of enchancing the overall pefromance of the computer

- Use GPU to parallelize
- GPU peak performance much higher than CPU
- Used for graphics, scientific

3 ways to accelerate applications

- Libraries
  - Pre optimized GPU accelerated libraries as an alternative
  - NVIDIA libraries
  - CUDA accelerated libraries can hide a lot of the architectural details of the GPU
- openACC directives
  - Automatically parallelize compute intensive loops
  - Perform custom computations that are not yet available in the library
  - They are directives that identify parallel regions of the code to accelerate
  - It automatically accelerates these regions without requiring changes to the underlying code
  - If there is any compiler without any support for openACC, will ignore the directives as if they were code comments
- Programming language
  - Use language that you already know to implement

3 ways to CUDA-accelerated applications

- Substitute library call with equivalent CUDA library
- manage data locality; data needs to be moved from the GPU to some library
- Rebuild and link the code

CUDA (computer unfied device architecture)

- A software program that helps computer run faster
- CUDA is embedded in NVIDIA graphic cards
- Host
  - Pointers point to CPU and its memory
- Device
  - Pointers point to GPU and its memory

```
__global__ void mykernel(void)
```

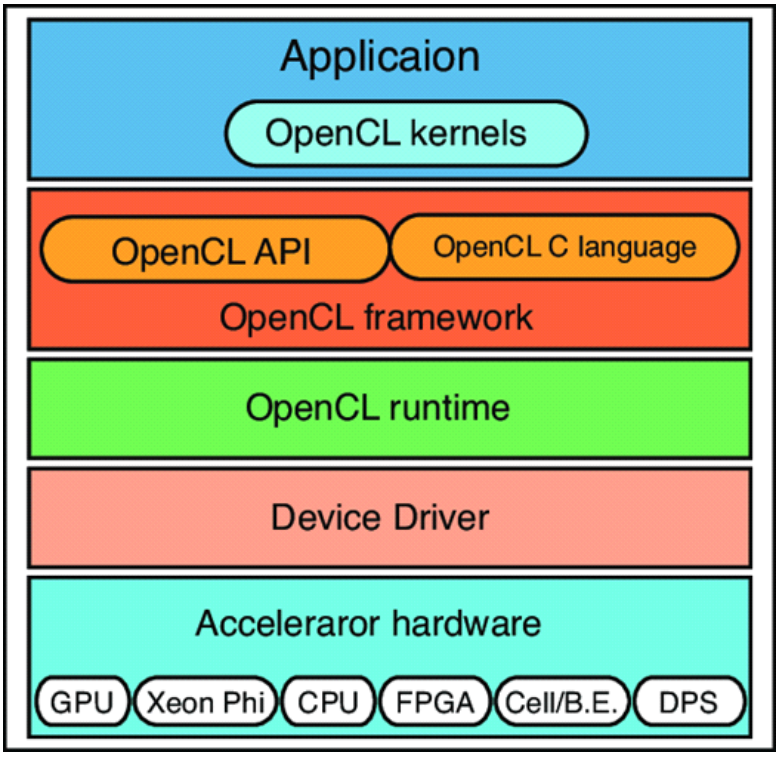
- It is called from the host code and is an initializer, but runs on the device
- Nvcc separates source code from host and device
- Device functions are processes by NVIDIA compiler
- ```
  mykernel<<<n,1>>>();
  ```
- This is the host call to device code
- For handling device memory
  - cudaMalloc(), cudaFree(), cudaMemcpy()
- GPU is about parallelism so instead of running code once, make it n times
  ```

- Each parallel invocation of add() is referred to as a **block**
- A block can be split into parallel threads
- `mykernel<<<1,n>>>();`
- Create blocks and assign 1 thread to each
- `int index = threadIdx.x + blockIdx.x * M;`
- Kernel launches are asynchronous
- cudaMemcpy() blocks the CPU until the copy is complete
- Multiple threads can share a device
- A single thread can manage multiple devices

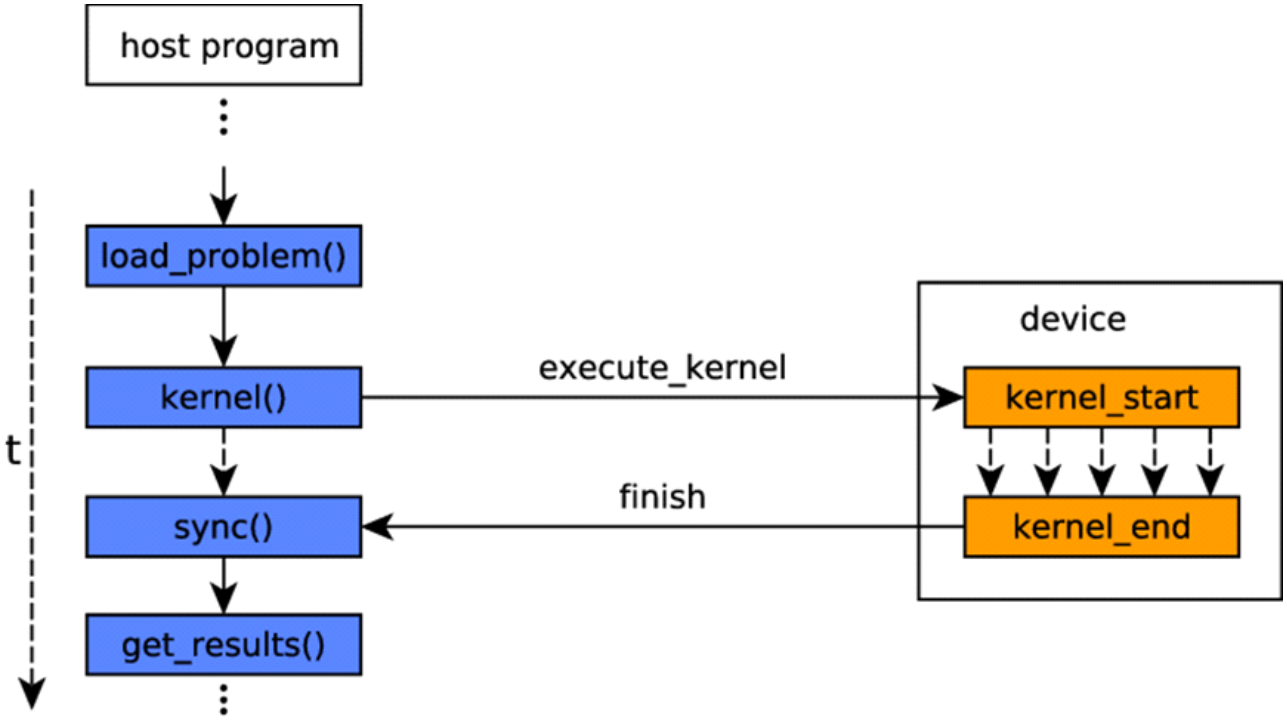
Opencl

Thursday, 15 May 2025 1:06 am

- For heterogenous parallel computing systems



- 0 means CPU and 1 means GPU



- CPU is treated as one OpenCL device
- Each GPU is a separate OPENcl device
- A thread is executed by a core
- A block of threads is executed by streaming multiprocessor
- A kernel grid consists of multiple thread blocks is executed by complete GPU unit
- Private memory
    - Per work-item
- Local memory
    - Shared within a work group
- Global/constant memory
    - Visible to all work groups
    - Is read only
- Host memory
    - Available on CPU only

UNDERSTANDING THE HOST PROGRAM

1. Define platform
    a. Creating a context and queue
    b. Context is the environment in which the kernel will execute
    c. The context includes one or more devices, device memory, one or more command queues
    d. All commands are submitted through a command queue
    e. Each command queue points to a single device within a context
    f. In order queues
        i. They are completed in the order as they appear in the host program
        ii. They are synchronized, will execute in the same sequence as they were added
    g. Out of order queues
        i. Can be executed/completed in any order based on the availability of resources nd the devices scheduling
        ii. Non blocking
        iii. It executes independent tasks concurrently and donot have any dependencies

```
cl::Context
context(CL_DEVICE_TYPE_CPU/GPU/ACCELERATOR/DEFAULT);

cl::CommandQueue
queue(context);
```

2. Create and build the program

```
cl::Program program(context,
KernelSource, true );
```

- Define source code for the kernel program either as a string literal or read it from a file for large applications
- A program object comprises of
    - Context

- Program source
- Target devices

- Buffers are declared on the host as an object type

- **cl::Buffer**

- **clCreateBuffer()**
  - Arrays in host memory hold your original host-side data
  - Create a device side buffer (d_a) and assign read only memory to hold the host array (h_a) and copy it into device memory

3. Setup memory objects
   a. For each input vector, you need a memory object
   b. Then define openCL device buffers
   c. In device memory, we have 2 kinds of memory object
      i. Buffer object: linear
      ii. Image project: 2d-3d region of memory and can only be accessed with read and write function
   d. In the parameters; true means read only whereas false means read/write
   e. Submit the command to copy the device buffer back to host memory in array of host

Creates a buffer object.

```
cl_mem clCreateBuffer(cl_context context,
                      cl_mem_flags flags,
                      size_t size,
                      void *host_ptr,
                      cl_int *errcode_ret)
```

4. Define kernel

a. `cl::Kernel kernel(program, "vecadd");`

   b. Call the kernel as a function in your host code to enqueue the kernel

5. Submit commands
   a. For kernel launches, specify global and local dimensions

- Include key header files
  - #include <CL/cl.hpp>

INTRODUCTION TO OPENCL KERNEL PROGRAMMING

- ___kernel declares a function to make it visible to host code
- ___global, ____local,____constant, ____private declared with an address space qualifier
- Replace loops with kernel function

- **Traditional loops**

```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

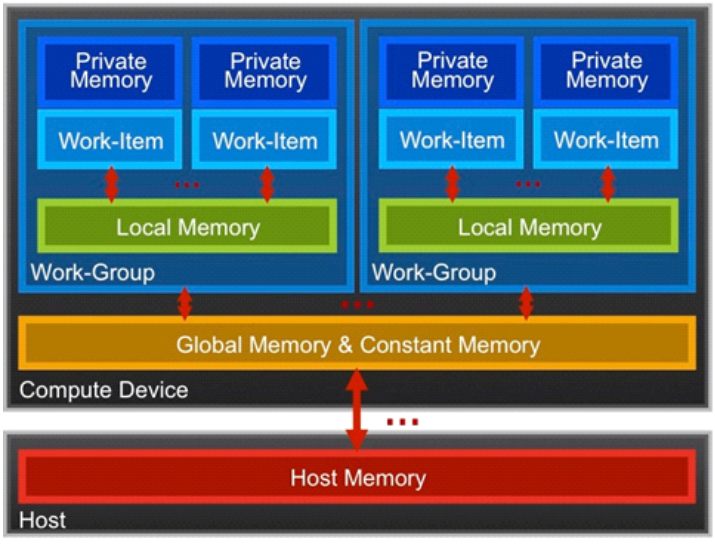
- **OpenCL**

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id]  = a[id] * b[id];
}
// execute over n work-items
```

- ND range is defined by two parameters
  - Global size in each dimensions
  - Local size in each dimension
- Ndrange is a concept to specify the number of work items and their distribution across a compute device. It is used to define the size and shape of the problem that needs to be solved
- Barriers
  - All work items within a work group must execute the barrier function before any work item can continue
- Memory fences
  - Ordering between memory operations
- Pointers to function are not allowed but points to pointers allowed within a kernel

THREAD MAPPING

-



- Responsible for moving data from host->global->local and back