1 OpenMP

OpenMP is an API that supports multi-platform shared-memory multiprocessing,it uses threads to run code in parallel.

How to run/compile a program

- `# gcc -fopenmp <SOURCE_CODE> -o <OUTPUT>`
- `# ./<OUTPUT>`

Indicating the number of threads

- `omp_set_num_threads(NUM_THRADS);`
- `# pragma omp parallel num_threads(NUM_THRADS)`
- `export OMP_NUM_THREADS = NUM_THRADS;`

private/shared variables

By default variables are shared across all threads,sometimes we want each thread to have its own variable (copy of that variable),a solution that doesn't require creating an array / explicitly defining a variable for each thread can be achieved by marking that variable as private.

```
int prv = 0,pub = 0;

printf("Thread id,Private variable,Shared Variable\n");

#pragma omp parallel num_threads(NUM_THREADS) private(prv) shared(pub)
{
   int thread_id = omp_get_thread_num();
   prv = 0;

   prv += thread_id;
   sleep(1); // just so the difference is obvious
   pub += thread_id;

   printf("%d,%d,%d\n",thread_id,prv,pub);
}
```

For loops

Divides the number of iterations in a loop across threads,for example if NUM_THREADS is 4 and NUM_ITERS is 8 each thread will be assigned exactly tow iterations (default behaviour).

```
#pragma omp parallel num_threads(NUM_THREADS)
{
   #pragma omp for
   for (int i = 0; i < NUM_ITERS; i++) {
      int id = omp_get_thread_num();
      printf("Thread id : %d, Loop id : %d\n", id, i);
   }
}
```

scheduling

scheduling defines how loops iterations are assigned to threads.

- **static scheduling** :
  - Behaviour : Divides the iterations of the loop into equal-sized chunks, with each chunk assigned to a thread statically at compile time. The optional chunk parameter specifies the size of each chunk.
  - Syntax :

```
#pragma omp for schedule(static [,chunk])
```

- **dynamic scheduling** :
  - Behaviour : Divides the iterations of the loop into chunks of size chunk, and assigns these chunks dynamically to threads as threads become available to do work.
  - Syntax :

```
#pragma omp for schedule(dynamic [,chunk])
```

- **guided scheduling** :
  - Behaviour : Similar to dynamic scheduling, but the size of the chunks decreases dynamically over time. Initially, larger chunks are assigned, which helps reduce scheduling overhead, but as the loop progresses, the chunk size decreases to balance the workload among threads.
  - Syntax :

```
#pragma omp for guided(static [,chunk])
```

- **runtime scheduling** :
  - Behaviour : Allows the scheduling policy to be determined at runtime using the OMP_SCHEDULE environment variable or the omp_set_schedule function. This provides flexibility in choosing the scheduling policy without recompiling the code.
  - Syntax :

```
#pragma omp for schedule(runtime)
```

Reduction Clauses and Directives

The reduction clauses are used to combine the results obtained from each thread based on an operator, its general syntax is : #pragma omp for reduction(<op>:<var_name>), the first argument is the operator (independent of the operator used in the loop), and the second argument is the variable to store the result.

<op> can be one of the following : $\{+, -, *, \&, |, \hat{}\ \&\&, ||, max, min\}$

for example here's below a program that computes the sum of an array :

```
int NUM_ITERS = 2 * NUM_THREADS;
int arr[ARRAY_SIZE];
int total = 0;

for (int i = 0;i < ARRAY_SIZE;i++) {
   arr[i] = i;
}

#pragma omp parallel num_threads(4)
{
   #pragma omp for reduction(+:total)
```

```
      for (int i = 0; i < ARRAY_SIZE; i++) {
         total += arr[i];
      }
   }

   printf("Total = %d\n", total);
```

synchronization

- critical: Ensures that a specific block of code is executed by only one thread at a time. all threads will eventually execute the code in this section but not in the same time.
- single: Specifies that a block of code should be executed by only one thread, but it doesn't specify which thread.
- master: like single but the block is ensured to be executed by the master (main program,thread with ID 0).
- barrier: Synchronizes all threads in a parallel region. Each thread waits at the barrier until all threads have reached it, then they all proceed.
- atomic: Specifies that a specific operation is atomic,it is used for indivisible operations like an increment.

sections

sections are used when a problem can be divided into independent sections that can be executed in parallel for example : $a = f1(b, c) + f2(d, e)$,can be divided into : $a1 = f1(a, b); a2 = f2(d, e); a = a1 + a2$.

```
   int a = 0;
   int b = 1,c = 2,d = 3,e = 4;
   int a1,a2;

   #pragma omp parallel sections
   {

      #pragma omp section
      {
         a1 = f1(b,c);
      }

      #pragma omp section
      {
         a2 = f2(d,e);
      }

   }

   a = a1 + a2;

   printf("a = %d\n",a);
```

2 MPI

MPI is a standardized and portable message-passing standard designed to function on parallel computing architectures,and it uses processes and distributed memory.

How to run/compile a program

- `# mpicc <SOURCE_CODE> -o <OUTPUT>`
- `# mpirun -np <NUMBER_OF_PROCESSES> ./<OUTPUT>`

The genrale structure of an MPI program

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

// your code

MPI_Finalize();
```

Number of processes

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Process ID

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Message Passing

```
MPI_Send(BUFFER,BUFF_SIZE,TYPE,DST_ID,TAG,COMMUNICATOR);
MPI_Recv(BUFFER,BUFF_SIZE,TYPE,SRC_ID,TAG,COMMUNICATOR,STATUS);
```

- Buffer : a pointer to the data to send to / receive from the other process.
- Number of items in the buffer.
- Type : the buffer's data type.
- SRC_ID / DST_ID : the id of sender / receiver.
- Tag : a message type,to differenciate messages.
- COMMUNICATOR : a static group of processes that can communicate with each other,MPI_COMM_WORLD is a default group that contains all the processes.
- STATUS : contains additional informations.

Broadcast messages

```
MPI_Bcast(BUFF,SIZE,TYPE,ROOT,COMMUNICATOR)
```

Here ROOT is the rank of broadcast root (sender).

Scatter & Gather

Scatter : data division. Gather : results aggregation.

```
MPI_Scatter(BUFF,SEND_COUNT,TYPE,RECV_BUFF,RECV_COUNT,TYPE,ROOT,COMMUNICATOR)

MPI_Gather(BUFF,SEND_COUNT,TYPE,RECV_BUFF,RECV_COUNT,TYPE,ROOT,COMMUNICATOR)

MPI_Reducer(BUFF,SEND_COUNT,TYPE,RECV_BUFF,RECV_COUNT,TYPE,OPERATION,ROOT,COMMUNICATOR)
```

Synchronisation

```
MPI_Barrier(COMMUNICATOR)
```

Data Types

- MPI_SHORT short int
- MPI_INT int
- MPI_LONG long int
- MPI_LONG_LONG long long int
- MPI_UNSIGNED_CHAR unsigned char
- MPI_UNSIGNED_SHORT unsigned short int
- MPI_UNSIGNED unsigned int
- MPI_UNSIGNED_LONG unsigned long int

Example : Ordered Hello-World

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

/**
 * printing hello world while ensuring the order
*/

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

   int size, rank;
   char dummy;

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

   // If it's not the first process
   // It has to receive a message from
   // the previous process
   if (rank != 0) {
      int src = (rank + size - 1) % size;
      MPI_Recv(&dummy,1,MPI_CHAR,src,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   }

   // Notify the next process
   printf("Process %d is saying Hello World.\n", rank);

   // The last process has no process to notify
   if (rank != size - 1) {
      MPI_Send(&dummy,1,MPI_CHAR,(rank + 1) % size,0,MPI_COMM_WORLD);
   }

   MPI_Finalize();
```

Cluster setup

Master

1. Update and Upgrade the System :
   `# sudo apt update`
   `# sudo apt upgrade`
2. Install Build Essentials and MPI :
   `# sudo apt install build-essential`
   `# sudo apt install mpi-default-dev`
3. Set Up SSH for Passwordless Login :
   `# ssh-keygen -t rsa`
   `# sudo apt install openssh-server`
   `# ssh-copy-id esisba@172.16.0.11`
   `# ssh esisba@172.16.0.11`
4. Set Up NFS Server :
   `# sudo apt install nfs-server`
   `# sudo mkdir /mirror echo "/mirror *(rw,sync)" | sudo tee -a /etc/exports`
   `# sudo service nfs-kernel-server restart`
   `# sudo chown esisba /mirror`
5. Configure MPI to Use Worker Nodes
   `# nano machines` with content : ‘172.16.0.11‘.
6. Run MPI Program on Multiple Nodes :
   `# mpicc -o a.out main.c`
   `# mpirun -n 2 -machinefile machines ./a.out`

Slave

1. Update and Upgrade the System
2. Install Build Essentials and MPI
3. Install SSH Server
4. Set Up NFS Client :
   `# sudo apt install nfs-client`
   `# sudo mkdir /mirror`
   `# sudo mount hpcm-VM:/mirror /mirror MPI`
5. Cluster Setup with NFS on Virtual Machines
6. echo ”hpcm:/mirror /mirror nfs” — sudo tee -a /etc/fstab