# HPCL Codes

## 3. OpenMP program to print Hello world with thread ID

#include <omp.h>  
#include <stdio.h>  
int main() {  
 #pragma omp parallel  
 {  
 printf("Hello World from thread %d\n", omp\_get\_thread\_num());  
 }  
 return 0;  
}

## 4. Parallel program to print NAME from 4 cores

#include <omp.h>  
#include <stdio.h>  
int main() {  
 #pragma omp parallel num\_threads(4)  
 {  
 printf("Swarnim Shekhar\n");  
 }  
 return 0;  
}

## 5. OpenMP program demonstrating private clause

#include <omp.h>  
#include <stdio.h>  
int main() {  
 int val = 1234;  
 printf("Initial value of val: %d\n", val);  
 #pragma omp parallel num\_threads(4) firstprivate(val)  
 {  
 printf("Thread %d initial val: %d\n", omp\_get\_thread\_num(), val);  
 val++;  
 printf("Thread %d updated val: %d\n", omp\_get\_thread\_num(), val);  
 }  
 printf("Final value of val: %d\n", val);  
 return 0;  
}

## 6. OpenMP program to demonstrate private clause

#include <omp.h>  
#include <stdio.h>  
int main() {  
 int val = 1234;  
 printf("Initial value of val: %d\n", val);  
 #pragma omp parallel private(val)  
 {  
 val = omp\_get\_thread\_num();  
 printf("Thread %d private val: %d\n", omp\_get\_thread\_num(), val);  
 }  
 printf("Final value of val: %d\n", val);  
 return 0;  
}

## 7. Parallel C program with static scheduling

#include <omp.h>  
#include <stdio.h>  
int main() {  
 int i;  
 int n = 20, chunk = 5;  
 #pragma omp parallel for schedule(static, chunk)  
 for (i = 0; i < n; i++) {  
 printf("Thread %d handling iteration %d\n", omp\_get\_thread\_num(), i);  
 }  
 return 0;  
}

## 8. Parallel C program to print series 2 and 4

#include <omp.h>  
#include <stdio.h>  
int main() {  
 #pragma omp parallel sections  
 {  
 #pragma omp section  
 {  
 printf("2\n");  
 }  
 #pragma omp section  
 {  
 printf("4\n");  
 }  
 }  
 return 0;  
}

## 9. MPI program to print Hello World

#include <mpi.h>  
#include <stdio.h>  
int main(int argc, char\*\* argv) {  
 MPI\_Init(&argc, &argv);  
 printf("Hello World\n");  
 MPI\_Finalize();  
 return 0;  
}

## 10. MPI program to send and receive Hello World

#include <mpi.h>  
#include <stdio.h>  
int main(int argc, char\*\* argv) {  
 int rank, size;  
 char message[20];  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 if (rank == 0) {  
 for (int i = 1; i < size; i++) {  
 MPI\_Recv(message, 20, MPI\_CHAR, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
 printf("Received: %s from process %d\n", message, i);  
 }  
 } else {  
 sprintf(message, "Hello from %d", rank);  
 MPI\_Send(message, 20, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);  
 }  
 MPI\_Finalize();  
 return 0;  
}

## 11. MPI program to find sum of first N integers

#include <mpi.h>  
#include <stdio.h>  
int main(int argc, char\*\* argv) {  
 int rank, size, N = 10000, local\_sum = 0, total\_sum = 0;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 for (int i = rank + 1; i <= N; i += size) {  
 local\_sum += i;  
 }  
 MPI\_Reduce(&local\_sum, &total\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);  
 if (rank == 0) {  
 printf("Sum of first %d numbers is %d\n", N, total\_sum);  
 }  
 MPI\_Finalize();  
 return 0;  
}

## 12. MPI sum in ring topology

#include <mpi.h>  
#include <stdio.h>  
int main(int argc, char\*\* argv) {  
 int rank, size, n = 10, sum = 0, local;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 local = rank + 1;  
 sum = local;  
 for (int i = 1; i < size; i++) {  
 int recv\_data;  
 MPI\_Send(&sum, 1, MPI\_INT, (rank + 1) % size, 0, MPI\_COMM\_WORLD);  
 MPI\_Recv(&recv\_data, 1, MPI\_INT, (rank - 1 + size) % size, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
 sum = recv\_data;  
 }  
 printf("Rank %d, Total Sum: %d\n", rank, sum);  
 MPI\_Finalize();  
 return 0;  
}

**13. CUDA program for matrix addition**

#include <cuda.h>  
#include <stdio.h>  
  
\_\_global\_\_ void matrixAdd(int\* A, int\* B, int\* C, int n) {  
 int idx = threadIdx.x + blockDim.x \* blockIdx.x;  
 if (idx < n) {  
 C[idx] = A[idx] + B[idx];  
 }  
}  
  
int main() {  
 int n = 16;  
 int size = n \* sizeof(int);  
 int A[n], B[n], C[n];  
 int \*d\_A, \*d\_B, \*d\_C;  
  
 // Initialize arrays  
 for (int i = 0; i < n; i++) {  
 A[i] = i;  
 B[i] = i \* 2;  
 }  
  
 // Allocate device memory  
 cudaMalloc((void\*\*)&d\_A, size);  
 cudaMalloc((void\*\*)&d\_B, size);  
 cudaMalloc((void\*\*)&d\_C, size);  
  
 // Copy data to device  
 cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice);  
  
 // Launch kernel  
 int threadsPerBlock = 8;  
 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;  
 matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, n);  
  
 // Copy result back to host  
 cudaMemcpy(C, d\_C, size, cudaMemcpyDeviceToHost);  
  
 // Print result  
 for (int i = 0; i < n; i++) {  
 printf("C[%d] = %d\n", i, C[i]);  
 }  
  
 // Free device memory  
 cudaFree(d\_A);  
 cudaFree(d\_B);  
 cudaFree(d\_C);  
  
 return 0;  
}

**14. CUDA program for matrix multiplication**

#include <cuda.h>  
#include <stdio.h>  
  
\_\_global\_\_ void matrixMul(int\* A, int\* B, int\* C, int n) {  
 int row = threadIdx.y + blockDim.y \* blockIdx.y;  
 int col = threadIdx.x + blockDim.x \* blockIdx.x;  
 if (row < n && col < n) {  
 int sum = 0;  
 for (int k = 0; k < n; k++) {  
 sum += A[row \* n + k] \* B[k \* n + col];  
 }  
 C[row \* n + col] = sum;  
 }  
}  
  
int main() {  
 int n = 4;  
 int size = n \* n \* sizeof(int);  
 int A[n][n], B[n][n], C[n][n];  
 int \*d\_A, \*d\_B, \*d\_C;  
  
 // Initialize matrices  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++) {  
 A[i][j] = i + j;  
 B[i][j] = i - j;  
 }  
 }  
  
 // Allocate device memory  
 cudaMalloc((void\*\*)&d\_A, size);  
 cudaMalloc((void\*\*)&d\_B, size);  
 cudaMalloc((void\*\*)&d\_C, size);  
  
 // Copy data to device  
 cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice);  
  
 // Launch kernel  
 dim3 threadsPerBlock(2, 2);  
 dim3 blocksPerGrid((n + threadsPerBlock.x - 1) / threadsPerBlock.x,  
 (n + threadsPerBlock.y - 1) / threadsPerBlock.y);  
 matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, n);  
  
 // Copy result back to host  
 cudaMemcpy(C, d\_C, size, cudaMemcpyDeviceToHost);  
  
 // Print result  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++) {  
 printf("C[%d][%d] = %d\n", i, j, C[i][j]);  
 }  
 }  
  
 // Free device memory  
 cudaFree(d\_A);  
 cudaFree(d\_B);  
 cudaFree(d\_C);  
  
 return 0;  
}