**OpenMP**

1 Multiply two matrices A & B and find the resultant matrix C

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define NRA 62

#define NCA 15

#define NCB 7

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

{

int

tid, nthreads, i, j, k, chunk;

double a[NRA][NCA],

/\* matrix A to be multiplied \*/

b[NCA][NCB],

/\* matrix B to be multiplied \*/

c[NRA][NCB];

/\* result matrix C \*/

/\* number of rows in matrix A \*/

/\* number of columns in matrix A \*/

/\* number of columns in matrix B \*/

chunk = 10;

/\*\*\* Spawn a parallel region explicitly scoping all variables \*\*\*/

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)

{

tid = omp\_get\_thread\_num();

if (tid == 0)

{

nthreads = omp\_get\_num\_threads();

printf("Starting matrix multiple example with %d threads\n",nthreads);

printf("Initializing matrices...\n");

}

/\*\*\* Initialize matrices \*\*\*/

#pragma omp for schedule (static, chunk)

for (i=0; i<NRA; i++)

for (j=0; j<NCA; j++)

a[i][j]= i+j;

#pragma omp for schedule (static, chunk)

for (i=0; i<NCA; i++)

for (j=0; j<NCB; j++)

b[i][j]= i\*j;

#pragma omp for schedule (static, chunk)

for (i=0; i<NRA; i++)

for (j=0; j<NCB; j++)

c[i][j]= 0;

/\*\*\* Do matrix multiply sharing iterations on outer loop \*\*\*/

/\*\*\* Display who does which iterations for demonstration purposes \*\*\*/

printf("Thread %d starting matrix multiply...\n",tid);

#pragma omp for schedule (static, chunk)

for (i=0; i<NRA; i++)

{

printf("Thread=%d did row=%d\n",tid,i);

for(j=0; j<NCB; j++)

for (k=0; k<NCA; k++)

c[i][j] += a[i][k] \* b[k][j];

}

} /\*\*\* End of parallel region \*\*\*/

/\*\*\* Print results \*\*\*/

/\* set loop iteration chunk size \*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Result Matrix:\n");

for (i=0; i<NRA; i++)

{

for (j=0; j<NCB; j++)

printf("%6.2f ", c[i][j]);

printf("\n");

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf ("Done.\n");

}

**2**  **Write a program to show how first private clause works.( Factorial program)**

#include <stdio.h>

#include <malloc.h>

#include <omp.h>

long long factorial(long n)

{

long long i,out;

out = 1;

for (i=1; i<n+1; i++) out \*= i;

return(out);

}

int main(int argc, char \*\*argv)

{

int i,j,threads;

long long \*x;

long long n=12;

/\* Set number of threads equal to argv[1] if present \*/

if (argc > 1)

{

threads = atoi(argv[1]);

if (omp\_get\_dynamic())

{

omp\_set\_dynamic(0);

printf("called omp\_set\_dynamic(0)\n");

}

omp\_set\_num\_threads(threads);

}

printf("%d threads\n",omp\_get\_max\_threads());

x = (long long \*) malloc(n \* sizeof(long));

for (i=0;i<n;i++) x[i]=factorial(i);

j=0;

/\* Is the output the same if the following line is commented out? \*/

#pragma omp parallel for firstprivate(x,j)

for (i=1; i<n; i++)

{

j += i;

x[i] = j\*x[i-1];

}

for (i=0; i<n; i++)

printf("factorial(%2d)=%14lld x[%2d]=%14lld\n",i,factorial(i),i,x[i]);

return 0;

}

**3 Write a program to find the number of processes, number of threads etc**

**(environment information)**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

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

{

int nthreads, tid, procs, maxt, inpar, dynamic, nested;

/\* Start parallel region \*/

#pragma omp parallel private(nthreads, tid)

{

/\* Obtain thread number \*/

tid = omp\_get\_thread\_num();

/\* Only master thread does this \*/

if (tid == 0)

{

printf("Thread %d getting environment info...\n", tid);

/\* Get environment information \*/

procs = omp\_get\_num\_procs();

nthreads = omp\_get\_num\_threads();

maxt = omp\_get\_max\_threads();

inpar = omp\_in\_parallel();

dynamic = omp\_get\_dynamic();

nested = omp\_get\_nested();

/\* Print environment information \*/

printf("Number of processors = %d\n", procs);

printf("Number of threads = %d\n", nthreads);

printf("Max threads = %d\n", maxt);

printf("In parallel? = %d\n", inpar);

printf("Dynamic threads enabled? = %d\n", dynamic);

printf("Nested parallelism supported? = %d\n", nested);

}

} /\* Done \*/

}

**4 Write a program to find the largest element in an array (usage of locks)**

#include <stdio.h>

#include <omp.h>

#include<stdlib.h>

#define MINUS\_INFINITY -9999

#define MAXIMUM\_VALUE 65535

/\* Main Program \*/

main()

{

int \*array, i, Noofelements, cur\_max, current\_value;

omp\_lock\_t MAXLOCK;

printf("Enter the number of elements\n");

scanf("%d", &Noofelements);

if (Noofelements <= 0) {

printf("The array elements cannot be stored\n");

exit(1);

}

/\* Dynamic Memory Allocation \*/

array = (int \*) malloc(sizeof(int) \* Noofelements);

/\* Allocating Random Number To Array Elements \*/

srand(MAXIMUM\_VALUE);

for (i = 0; i < Noofelements; i++)

array[i] = rand();

if (Noofelements == 1) {

printf("The Largest Element In The Array Is %d", array[0]);

exit(1);

}

/\* Initializing The Lock \*/

printf("The locking is going to start\n");

omp\_set\_num\_threads(8);

omp\_init\_lock(&MAXLOCK);

cur\_max = MINUS\_INFINITY;

printf("the lock s initialized\n");

/\* OpenMP Parallel For Directive And Lock Functions \*/

#pragma omp parallel for

for (i = 0; i < Noofelements; i = i + 1) {

if (array[i] > cur\_max) {

omp\_set\_lock(&MAXLOCK);

if (array[i] > cur\_max)

cur\_max = array[i];

omp\_unset\_lock(&MAXLOCK);

}

}

/\* Destroying The Lock \*/

omp\_destroy\_lock(&MAXLOCK);

/\* Serial Calculation \*/

current\_value = array[0];

for (i = 1; i < Noofelements; i++)

if (array[i] > current\_value)

current\_value = array[i];

printf("The Array Elements Are \n");

for (i = 0; i < Noofelements; i++)

printf("\t%d", array[i]);

/\* Checking For Output Validity \*/

if (current\_value == cur\_max)

printf("\nThe Max Value Is Same For Serial And Using Parallel OpenMP Directive\n");

else {

printf("\nThe Max Value Is Not Same In Serial And Using Parallel OpenMP Directive\n");

exit(1);

}

/\* Freeing Allocated Memory \*/

free(array);

printf("\nThe Largest Number Of The Array Is %d\n", cur\_max);

}

**5 There are two arrays A and B write a program that has two blocks: one for**

**generating array C = A+B and another array D = A+B, such that work in blocks will**

**be done by different threads.**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define N 50

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

{

int i, nthreads, tid;

float a[N], b[N], c[N], d[N];

/\* Some initializations \*/

for (i=0; i<N; i++) {

a[i] = i \* 1.5;

b[i] = i + 22.35;

c[i] = d[i] = 0.0;

}

#pragma omp parallel shared (a,b,c,d,nthreads) private(i,tid)

{

tid = omp\_get\_thread\_num();

if (tid == 0)

{

nthreads = omp\_get\_num\_threads();

printf("Number of threads = %d\n", nthreads);

}

printf("Thread %d starting...\n",tid);

#pragma omp sections nowait

{

#pragma omp section

{

printf("Thread %d doing section 1\n",tid);

for (i=0; i<N; i++)

{

c[i] = a[i] + b[i];

printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);

}

}

#pragma omp section

{

printf("Thread %d doing section 2\n",tid);

for (i=0; i<N; i++)

{

d[i] = a[i] \* b[i];

printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);

}

}

} /\* end of sections \*/

printf("Thread %d done.\n",tid);

} /\* end of parallel section \*/

}

**6 Write a program to find prime numbers ( split )**

#include <stdio.h>

#include <omp.h>

#define N 100000000

#define TRUE 1

#define FALSE 0

int main(int argc, char \*\*argv )

{

char host[80];

int \*a;

int i, k, threads, pcount;

double t1, t2;

int found;

/\* Set number of threads equal to argv[1] if present \*/

if (argc > 1)

{

threads = atoi(argv[1]);

if (omp\_get\_dynamic())

{

omp\_set\_dynamic(0);

printf("called omp\_set\_dynamic(0)\n");

}

omp\_set\_num\_threads(threads);

}

printf("%d threads max\n",omp\_get\_max\_threads());

a = (int \*) malloc((N+1) \* sizeof(int));

// 1. create a list of natural numbers 2, 3, 4, ... none of which is marked.

for (i=2;i<=N;i++) a[i] = 1;

// 2. Set k = 2, the first unmarked number on the list.

k = 2;

t1 = omp\_get\_wtime();

// 3. Repeat

#pragma omp parallel firstprivate(k) private(i,found)

while (k\*k <= N)

{

// a. Mark all multiples of k between k^2 and N

#pragma omp for

for (i=k\*k; i<=N; i+=k) a[i] = 0;

// b. Find the smallest number greater than k that is unmarked

// and set k to this new value until k^2 > N

found = FALSE;

for (i=k+1;!found;i++)

{

if (a[i]){ k = i; found = TRUE; }

}

}

t2 = omp\_get\_wtime();

printf("%.2f seconds\n",t2-t1);

// 4. The unmarked numbers are primes

pcount = 0;

for (i=2;i<=N;i++)

{

if( a[i] )

{

pcount++;

//printf("%d\n",i);

}

}

printf("%d primes between 0 and %d\n",pcount,N);

}

**7 . Add two arrays A & B each of 1000 to generate an array C using reduction clause**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

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

{

int i, n;

float a[1000], b[1000], sum;

/\* Some initializations \*/

n = 1000;

for (i=0; i < n; i++)

a[i] = b[i] = i \* 1.0;

sum = 0.0;

#pragma omp parallel for reduction(+:sum)

for (i=0; i < n; i++)

sum = sum + (a[i] \* b[i]);

printf(" Sum = %f\n",sum);

}

**8 Write a program to find the largest element in an array (Check again) critical section**

#include <stdio.h>

#include <omp.h>

#include<stdlib.h>

#define MAXIMUM 65536

/\* Main Program \*/

main()

{

int \*array, i, Noofelements, cur\_max, current\_value;

printf("Enter the number of elements\n");

scanf("%d", &Noofelements);

if (Noofelements <= 0) {

printf("The array elements cannot be stored\n");

exit(1);

}

/\* Dynamic Memory Allocation \*/

array = (int \*) malloc(sizeof(int) \* Noofelements);

\*array, i, Noofelements, cur\_max, current\_value;

/\* Allocating Random Number Values To The Elements Of An Array \*/

srand(MAXIMUM);

for (i = 0; i < Noofelements; i++)

array[i] = rand();

if (Noofelements == 1) {

printf("The Largest Number In The Array is %d", array[0]);

exit(1);

}

/\* OpenMP Parallel For Directive And Critical Section \*/

cur\_max = 0;

omp\_set\_num\_threads(8);

#pragma omp parallel for

for (i = 0; i < Noofelements; i = i + 1) {

if (array[i] > cur\_max)

#pragma omp critical

if (array[i] > cur\_max)

cur\_max = array[i];

}

/\* Serial Calculation \*/

current\_value = array[0];

for (i = 1; i < Noofelements; i++)

if (array[i] > current\_value)

current\_value = array[i];

printf("The Input Array Elements Are \n");

for (i = 0; i < Noofelements; i++)

printf("\t%d", array[i]);

printf("\n");

/\* Checking For Output Validity \*/

if (current\_value == cur\_max)

printf("\nThe Max Value Is Same From Serial And Parallel OpenMP Directive\n");

else {

printf("\nThe Max Value Is Not Same In Serial And Parallel OpenMP Directive\n");

exit(1);

}

/\* Freeing Allocated Memory \*/

printf("\n");

free(array);

printf("\nThe Largest Number In The Given Array Is %d\n", cur\_max);

}

**9 Write a program to show how thread private clause works.**

#include <omp.h>

#include<stdio.h>

int a, b, i, tid;

float x;

#pragma omp threadprivate(a, x)

main () {

/\* Explicitly turn off dynamic threads \*/

omp\_set\_dynamic(0);

printf("1st Parallel Region:\n");

#pragma omp parallel private(b,tid)

{

tid = omp\_get\_thread\_num();

a = tid;

b = tid;

x = 1.1 \* tid +1.0;

printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);

} /\* end of parallel section \*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Master thread doing serial work here\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("2nd Parallel Region:\n");

#pragma omp parallel private(tid)

{

tid = omp\_get\_thread\_num();

printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);

} /\* end of parallel section \*/

}

**10 Addition of two array A & B to get array C (Check again) scheduling**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define CHUNKSIZE 10

#define N 100

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

{

int nthreads, tid, i, chunk;

float a[N], b[N], c[N];

/\* Some initializations \*/

for (i=0; i < N; i++)

a[i] = b[i] = i \* 1.0;

chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)

{

tid = omp\_get\_thread\_num();

if (tid == 0)

{

nthreads = omp\_get\_num\_threads();

printf("Number of threads = %d\n", nthreads);

}

printf("Thread %d starting...\n",tid);

#pragma omp for schedule (dynamic, chunk)

for (i=0; i<N; i++)

{

c[i] = a[i] + b[i];

printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);

}

} /\* end of parallel section \*/

}

**CUDA**

**Q1) Write a CUDA program for adding two vectors.**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void vecAdd(double \*a, double \*b, double \*c, int n)

{

// Get our global thread ID

int id = blockIdx.x\*blockDim.x+threadIdx.x;

// Make sure we do not go out of bounds

if (id < n)

c[id] = a[id] + b[id];

}

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

{

// Size of vectors

int n = 100;

// Host input vectors

double \*h\_a;

double \*h\_b;

//Host output vector

double \*h\_c;

// Device input vectors

double \*d\_a;

double \*d\_b;

//Device output vector

double \*d\_c;

// Size, in bytes, of each vector

size\_t bytes = n\*sizeof(double);

// Allocate memory for each vector on host

h\_a = (double\*)malloc(bytes);

h\_b = (double\*)malloc(bytes);

h\_c = (double\*)malloc(bytes);

// Allocate memory for each vector on GPU

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

int i;

// Initialize vectors on host

for( i = 0; i < n; i++ ) {

h\_a[i] = i;

h\_b[i] = i;

}

// Copy host vectors to device

cudaMemcpy( d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy( d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block

blockSize = 1024;

// Number of thread blocks in grid

gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel

vecAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

// Copy array back to host

cudaMemcpy( h\_c, d\_c, bytes, cudaMemcpyDeviceToHost );

// Sum up vector c and print result divided by n, this should equal 1 within error

double sum = 0;

for(i=0; i<n; i++)

printf(" %f + %f =%f\n",h\_a[i],h\_b[i],h\_c[i]);

//printf("final result: %f\n", sum/(double)n);

// Release device memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

// Release host memory

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

**Q2) Write a CUDA program to demonstrate different types of memory.**

// Using different memory spaces in CUDA

#include <stdio.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* using local memory \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// a \_\_device\_\_ or \_\_global\_\_ function runs on the GPU

\_\_global\_\_ void use\_local\_memory\_GPU(float in)

{

float f; // variable "f" is in local memory and private to each thread

f = in; // parameter "in" is in local memory and private to each thread

// ... real code would presumably do other stuff here ...

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* using global memory \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// a \_\_global\_\_ function runs on the GPU & can be called from host

\_\_global\_\_ void use\_global\_memory\_GPU(float \*array)

{

// "array" is a pointer into global memory on the device

array[threadIdx.x] = 2.0f \* (float) threadIdx.x;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* using shared memory \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// (for clarity, hardcoding 128 threads/elements and omitting out-of-bounds checks)

\_\_global\_\_ void use\_shared\_memory\_GPU(float \*array)

{

// local variables, private to each thread

int i, index = threadIdx.x;

float average, sum = 0.0f;

// \_\_shared\_\_ variables are visible to all threads in the thread block

// and have the same lifetime as the thread block

\_\_shared\_\_ float sh\_arr[128];

// copy data from "array" in global memory to sh\_arr in shared memory.

// here, each thread is responsible for copying a single element.

sh\_arr[index] = array[index];

\_\_syncthreads(); // ensure all the writes to shared memory have completed

// now, sh\_arr is fully populated. Let's find the average of all previous elements

for (i=0; i<index; i++) { sum += sh\_arr[i]; }

average = sum / (index + 1.0f);

printf("Thread id = %d\t Average = %f\n",index,average);

// if array[index] is greater than the average of array[0..index-1], replace with average.

// since array[] is in global memory, this change will be seen by the host (and potentially

// other thread blocks, if any)

if (array[index] > average) { array[index] = average; }

// the following code has NO EFFECT: it modifies shared memory, but

// the resulting modified data is never copied back to global memory

// and vanishes when the thread block completes

sh\_arr[index] = 3.14;

}

int main(int argc, char \*\*argv)

{

/\*

\* First, call a kernel that shows using local memory

\*/

use\_local\_memory\_GPU<<<1, 128>>>(2.0f);

/\*

\* Next, call a kernel that shows using global memory

\*/

float h\_arr[128]; // convention: h\_ variables live on host

float \*d\_arr; // convention: d\_ variables live on device (GPU global mem)

// allocate global memory on the device, place result in "d\_arr"

cudaMalloc((void \*\*) &d\_arr, sizeof(float) \* 128);

// now copy data from host memory "h\_arr" to device memory "d\_arr"

cudaMemcpy((void \*)d\_arr, (void \*)h\_arr, sizeof(float) \* 128, cudaMemcpyHostToDevice);

// launch the kernel (1 block of 128 threads)

use\_global\_memory\_GPU<<<1, 128>>>(d\_arr); // modifies the contents of array at d\_arr

// copy the modified array back to the host, overwriting contents of h\_arr

cudaMemcpy((void \*)h\_arr, (void \*)d\_arr, sizeof(float) \* 128, cudaMemcpyDeviceToHost);

// ... do other stuff ...

/\*

\* Next, call a kernel that shows using shared memory

\*/

// as before, pass in a pointer to data in global memory

use\_shared\_memory\_GPU<<<1, 128>>>(d\_arr);

// copy the modified array back to the host

cudaMemcpy((void \*)h\_arr, (void \*)d\_arr, sizeof(float) \* 128, cudaMemcpyHostToDevice);

// ... do other stuff ...

// force the printf()s to flush

cudaDeviceSynchronize();

return 0;

}

**Q3) Write a CUDA program to print the message “Hello World” and demonstrate threads by varying BLOCK\_WIDTH to different sizes.**

#include <stdio.h>

#define NUM\_BLOCKS 32

#define BLOCK\_WIDTH 1

\_\_global\_\_ void hello()

{

printf("Hello world! I'm a thread in block %d\n", blockIdx.x);

}

int main(int argc,char \*\*argv)

{

// launch the kernel

hello<<<NUM\_BLOCKS, BLOCK\_WIDTH>>>();

// force the printf()s to flush

cudaDeviceSynchronize();

printf("That's all!\n");

return 0;

}

**Q4) Write a CUDA program to print the message “Hello World” and demonstrate threads by varying ThreadId to different sizes.**

#include <stdio.h>

#define NUM\_BLOCKS 1

#define BLOCK\_WIDTH 512

\_\_global\_\_ void hello()

{

printf("Hello world! I'm thread %d\n", threadIdx.x);

}

int main(int argc,char \*\*argv)

{

// launch the kernel

hello<<<NUM\_BLOCKS, BLOCK\_WIDTH>>>();

// force the printf()s to flush

cudaDeviceSynchronize();

printf("That's all!\n");

return 0;

}

**Q5)Write a CUDA program to multiply two matrices**

// Multiply two matrices A \* B = C

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

//Thread block size

#define BLOCK\_SIZE 3

#define WA 3

// Matrix A width

#define HA 3

// Matrix A height

#define WB 3

// Matrix B width

#define HB WA

// Matrix B height

#define WC WB

// Matrix C width

#define HC HA

// Matrix C height

//Allocates a matrix with random float entries.

void randomInit(float \* data ,int size)

{

for(int i = 0; i < size; ++i)

//data[i] = rand() / (float) RAND\_MAX;

data[i] = i;

}

// CUDA Kernel

\_\_global\_\_ void matrixMul(float\* C,float\* A,float\* B,int wA,int wB)

{

// 2D Thread ID

int tx = threadIdx.x;

int ty = threadIdx.y;

// value stores the element that is computed by the thread

float value = 0;

for(int i = 0; i < wA; ++i)

{

float elementA = A[ty \* wA + i];

float elementB = B[i \* wB + tx];

value += elementA \* elementB;

}

// Write the matrix to device memory each

// thread writes one element

C[ty \* wA + tx] = value;

}

// Program main

int main(int argc ,char\*\* argv)

{

// set seed for rand()

srand(2006);

// 1. allocate host memory for matrices A and B

unsigned int size\_A = WA \* HA;

unsigned int mem\_size\_A =sizeof(float) \* size\_A;

float\* h\_A = (float\*) malloc(mem\_size\_A);

unsigned int size\_B = WB \* HB;

unsigned int mem\_size\_B =sizeof(float) \* size\_B;

float \* h\_B = (float\*) malloc(mem\_size\_B);

// 2. initialize host memory

randomInit(h\_A, size\_A);

randomInit(h\_B, size\_B);

// 3. print out A and B

printf("\n\nMatrix A\n");

for(int i = 0; i < size\_A; i++)

{

printf("%f ", h\_A[i]);

if(((i + 1) % WA) == 0)

printf("\n");

}

printf("\n\nMatrix B\n");

for(int i = 0; i < size\_B; i++)

{

printf

("%f ", h\_B[i]);

if(((i + 1) % WB) == 0)

printf("\n");

}

// 4. allocate host memory for the result C

unsigned int size\_C = WC \* HC;

unsigned int mem\_size\_C =sizeof(float) \* size\_C;

float \* h\_C = (float \*) malloc(mem\_size\_C);

// 8. allocate device memory

float\* d\_A;

float\* d\_B;

cudaMalloc((void\*\*) &d\_A, mem\_size\_A);

cudaMalloc((void\*\*) &d\_B, mem\_size\_B);

//9. copy host memory to device

cudaMemcpy(d\_A, h\_A,mem\_size\_A ,cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B,mem\_size\_B ,cudaMemcpyHostToDevice);

// 10. allocate device memory for the result

float\* d\_C;

cudaMalloc((void\*\*) &d\_C, mem\_size\_C);

// 5. perform the calculation

// setup execution parameters

dim3 threads(BLOCK\_SIZE , BLOCK\_SIZE);

dim3 grid(WC / threads.x, HC / threads.y);

// execute the kernel

matrixMul<<< grid , threads >>>(d\_C, d\_A,d\_B, WA, WB);

// 11. copy result from device to host

cudaMemcpy(h\_C, d\_C, mem\_size\_C ,cudaMemcpyDeviceToHost);

// 6. print out the results

printf("\n\n Matrix C ( Results ) \n ");

for(int i = 0;i<size\_C; i ++){

printf("%f",h\_C[i]);

if(((i+ 1) % WC) == 0)

printf("\n");

}

printf("\n");

// 7.clean up memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

free(h\_A);

free(h\_B);

free(h\_C);

}

**MPI**

1. Write a MPI program where each processor sends an integer number and its rank to the master processor, where the master gathers all the information and prints the data accordingly
2. Write an MPI program where the master processor broadcasts a message “HELLO MSRIT” to the remaining processors using broadcast system call.
3. Write a MPI program to find sum of 'n' integers on 'p' processors using point-to-point communication libraries call
4. Write a MPI program to calculate and print the value of PI.
5. Write a MPI program to send the message from a process whose rank=3 to all other remaining processes.
6. Write a MPI program where each processor send a string and its rank to the master processor, where the master gathers all the information and prints the data accordingly
7. Write a MPI program scatter the information to different processes.

Program Execution (All three):20 marks

Write up : 5 Marks

Program Modification : 5 Marks

**MPI solutions**

**1.Write a MPI program where each processor sends an integer number and its rank to the master processor, where the master gathers all the information and prints the data accordingly**

#include <stdio.h>

#include <mpi.h>

void main(int argc, char \*argv[])

{

int rank,size;

double param[6],mine;

int sndcnt,rcvcnt;

int i;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

sndcnt=1;

mine=23.0+rank;

if(rank==0) rcvcnt=1;

MPI\_Gather(&mine,sndcnt,MPI\_DOUBLE,param,rcvcnt,MPI\_DOUBLE,0,MPI\_COMM\_WORLD);

if(rank==0)

for(i=0;i<size;++i)

{

printf("PE:%d param[%d] is %f \n",rank,i,param[i]);

printf(" %d %d \n",rank,i);

}

MPI\_Finalize();

}

**2. Write an MPI program where the master processor broadcasts a message “HELLO MSRIT” to the remaining processors using broadcast system call.**

#include <stdio.h>  
#include <string.h>  
#include "mpi.h"  
int main (int argc, char \*argv[])  
{   
int rank;  
char msg[20];  
MPI\_Init (&argc, &argv);  
MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank);  
  
 if (rank == 0) strcpy(msg,"HELLO MSRIT");  
 MPI\_Bcast (msg, 20, MPI\_CHAR, 0, MPI\_COMM\_WORLD);  
 printf ("[%d] msg = %s\n", rank, msg);  
 // Wait for every process to reach this code  
 MPI\_Barrier (MPI\_COMM\_WORLD);  
 MPI\_Finalize();  
 return 0;  
}

**3.Write a MPI program to find sum of 'n' integers on 'p' processors using point-to-point communication libraries call**

#include <stdio.h>  
#include "mpi.h"  
  
int main(int argc,char \*argv[])  
{  
 int iproc;  
 int MyRank, Numprocs, Root = 0;  
 int value, sum = 0;  
 int Source, Source\_tag;  
 int Destination, Destination\_tag;  
 MPI\_Status status;  
  
 MPI\_Init(&argc,&argv);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD,&Numprocs);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);  
  
 if(MyRank == Root){  
  
 for(iproc = 1 ; iproc < Numprocs ; iproc++){  
 Source = iproc;  
 Source\_tag = 0;  
  
 MPI\_Recv(&value, 1, MPI\_INT, Source, Source\_tag,   
 MPI\_COMM\_WORLD, &status);  
 sum = sum + value;  
 }  
 printf("MyRank = %d, SUM = %d\n", MyRank, sum);  
 }  
 else{  
 Destination = 0;  
 Destination\_tag = 0;  
  
 MPI\_Send(&MyRank, 1, MPI\_INT, Destination, Destination\_tag,   
 MPI\_COMM\_WORLD);  
 }  
  
 MPI\_Finalize();  
  
}

**4.Write a MPI program to calculate and print the value of PI.**

#include <stdio.h>

#include <math.h>

#include "mpi.h"

double func(double x)

{

return (4.0 / (1.0 + x\*x));

}

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

{

int NoInterval, interval;

int MyRank, Numprocs, Root = 0;

double mypi, pi, h, sum, x;

double PI25DT = 3.141592653589793238462643;

/\*....MPI initialisation....\*/

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&Numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);

if(MyRank == Root){

printf("\nEnter the number of intervals : ");

scanf("%d",&NoInterval);

}

/\*....Broadcast the number of subintervals to each processor....\*/

MPI\_Bcast(&NoInterval, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if(NoInterval <= 0){

if(MyRank == Root)

printf("Invalid Value for Number of Intervals .....\n");

MPI\_Finalize();

exit(-1);

}

h = 1.0 / (double)NoInterval;

sum = 0.0;

for(interval = MyRank + 1; interval <= NoInterval; interval += Numprocs){

x = h \* ((double)interval - 0.5);

sum += func(x);

}

mypi = h \* sum;

/\*....Collect the areas calculated in P0....\*/

MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, Root, MPI\_COMM\_WORLD);

if(MyRank == Root){

printf("pi is approximately %.16f, Error is %.16f\n",

pi, fabs(pi - PI25DT));

}

MPI\_Finalize();

}

**5.Write a MPI program to send the message from a process whose rank=3 to all other remaining processes.**

#include <stdio.h>

#include <mpi.h>

#include <string.h>

#define BUFFER\_SIZE 32

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

{

int MyRank,Numprocs, Destination, iproc;

int tag = 0;

int Root = 3, temp = 1;

char Message[BUFFER\_SIZE];

MPI\_Init(&argc,&argv);

MPI\_Status status;

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&Numprocs);

/\* print host name, and send message from process with rank 0 to all other processes \*/

if(MyRank == 3) {

system("hostname");

strcpy(Message, "Hello India");

for (temp=0; temp<Numprocs&&temp!=3;temp++)

{

MPI\_Send(Message, BUFFER\_SIZE, MPI\_CHAR, temp, tag,MPI\_COMM\_WORLD);

}

}

else {

system("hostname");

MPI\_Recv(Message, BUFFER\_SIZE, MPI\_CHAR, Root, tag,MPI\_COMM\_WORLD, &status);

printf("\n%s in process with rank %d from Process with rank %d\n", Message,MyRank,Root);

}

MPI\_Finalize();

}

output:

PC-IBM-NVIDIA-SAP

PC-IBM-NVIDIA-SAP

PC-IBM-NVIDIA-SAP

Hello India in process with rank 2 from Process with rank 3

Hello India in process with rank 0 from Process with rank 3

PC-IBM-NVIDIA-SAP

Hello India in process with rank 1 from Process with rank 3

**6.Write a MPI program where each processor send a string and its rank to the master processor, where the master gathers all the information and prints the data accordingly**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <mpi.h>

void main(int argc, char \*argv[])

{

int rank,size;

char param[10][10];

char mine[10];

char srank[10];

int sndcnt,rcvcnt;

int i;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

sndcnt=10;

strcpy(mine,"Hello ");

//srank = itoa(rank,srank,10);

sprintf(srank, "%d", rank);

strcat(mine,srank);

if(rank==0) rcvcnt=10;

MPI\_Gather(&mine,sndcnt,MPI\_CHAR,param,rcvcnt,MPI\_CHAR,0,MPI\_COMM\_WORLD);

if(rank==0)

for(i=0;i<size;++i)

{

printf("PE:%d param[%d] is %s \n",rank,i,param[i]);

printf(" %d %d \n",rank,i);

}

MPI\_Finalize();

}

output

ibmsap@PC-IBM-NVIDIA-SAP:~$ mpirun -np 4 ./p6

PE:0 param[0] is Hello 0

0 0

PE:0 param[1] is Hello 1

0 1

PE:0 param[2] is Hello 2

0 2

PE:0 param[3] is Hello 3

0 3

**7.Write a MPI program scatter the information to different processes.**

#include <stdio.h>

#include "mpi.h"

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

{ int data[] = {1, 2, 3, 4, 5, 6, 7}; // Size must be >= #processors

int rank, i = -1;

MPI\_Init (&argc, &argv);

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank);

MPI\_Scatter ((void \*)data, 1, MPI\_INT,

(void \*)&i , 1, MPI\_INT,

0, MPI\_COMM\_WORLD);

printf ("[%d] Received i = %d\n", rank, i);

MPI\_Finalize();

return 0;

}

output:

ibmsap@PC-IBM-NVIDIA-SAP:~$ mpirun -np 4 ./p6

PE:0 param[0] is Hello 0

0 0

PE:0 param[1] is Hello 1

0 1

PE:0 param[2] is Hello 2

0 2

PE:0 param[3] is Hello 3

0 3