PES COLLEGE OF ENGINEERG MANDYA

Computer Architecture and Parallel Programming(Integrated)

LAB MANUAL

(P22CS604)

Prepared By,

Prof. Shivashankar S K, Asst. Professor

Prof. Poojitha N, Asst. Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Syllabus

1. Write a OpenMp program (bubble sort) to illustrate
   a) Data hazard
   b) Eliminating data hazard
2. Write a OpenMp program to illustrate data dependency (Binary Search).
3. Write a OpenMp program to explore Loop Unrolling mechanism (Fibonacci numbers).
4. Write a OpenMp program to illustrate tomasulo's algorithm.
5. Write an OpenMp program which performs C=A+B & D=A-B in separate blocks/sections where A,B,C& D are arrays.
6. Write an OpenMp program to add all the elements of two arrays A & B each of size 1000 and store their sum in a variable using reduction clause.
7. Write an OpenMp program to multiply two matrices A & B and find the resultant matrix C.
8. Write an OpenMp program to show how thread private clause works.
9. Write an OpenMp program to show how first private clause works (Factorial program).
10. Write an OpenMP program to find prime numbers (split).

1. **Write a OpenMp program (bubble sort) to illustrate**
    a) **Data hazard**
    b) **Eliminating data hazard**

<u>a) To illustrate the data hazard</u>

```
#include <stdio.h>

#include <omp.h>

#define ARRAY_SIZE 25

void bubble_sort(int arr[], int n)

{

        for (int i = 0; i < n - 1; ++i)

        {

        // Last i elements are already sorted

                for (int j = 0; j < n - i - 1; ++j)

                {

                        if (arr[j] > arr[j + 1])

                        {

                         // Swap arr[j] and arr[j+1]

                                int temp = arr[j];

                                arr[j] = arr[j + 1];

                                arr[j + 1] = temp;

                        }

                }

        }

}
```

```
int main()
{
	int arr[ARRAY_SIZE];
	#pragma omp parallel for
	for (int i = 0; i < ARRAY_SIZE; ++i)
	{
		arr[i]=random()%100;
	}
	printf("\n the array elements are:\n");
	for(i=0; i<ARRAY_SIZE; i++)
	printf("%d  ",arr[i]);
	// Sort the array using bubble sort in parallel
	#pragma omp parallel shared(arr)
	{
		 bubble_sort(arr, ARRAY_SIZE);
	}
	printf("Sorted array:\n");
	for (int i = 0; i < ARRAY_SIZE; ++i)
	{
		printf("%d   ", arr[i]);
	}
	printf("\n");
	return 0;
}
```

Explanation:

- **bubble_sort Function**: Implements the bubble sort algorithm. It compares adjacent elements and swaps them if they are in the wrong order.
- **Parallel Region**: The sorting operation (bubble_sort) is placed inside a parallel region using *#pragma omp parallel shared(arr)*. This allows multiple threads to concurrently execute the sorting algorithm on the same array arr.

Data Hazard:

In this example, a data hazard occurs because:

1. **Shared Access to Array arr**: Each thread operates on the shared array arr without any synchronization mechanism (atomic, critical, etc.).
2. **Race Conditions**: Threads may simultaneously read and write to elements of arr, potentially causing race conditions where incorrect results (unsorted arrays) can occur due to unordered updates.

Sample Output:

The array elelments are:

83 86 77 15 93 35 86 92 49 21

Sorted array:

15 21 21 35 49 77 35 86 92 93

b) To eliminate the data hazard

```
void bubble_sort(int arr[], int n)
{
        #pragma omp critical
                for (int i = 0; i < n - 1; ++i)
                {
                // Last i elements are already sorted
                        for (int j = 0; j < n - i - 1; ++j)
                        {
                                if (arr[j] > arr[j + 1])
```

```
{
// Swap arr[j] and arr[j+1]
    int temp = arr[j];
    arr[j] = arr[j + 1];
    arr[j + 1] = temp;
}
        }
    }
  }
}
```

Mitigating Data Hazards:

To fix the data hazard issue, you can introduce synchronization mechanisms such as:

- Using *#pragma omp critical* to create a critical section where only one thread can access the array at a time.
- Implementing a parallel sorting algorithm that handles synchronization explicitly, such as parallel merge sort or parallel quicksort, which manage shared data more safely.

Sample Output:

The array elelments are:

83 86 77 15 93 35 86 92 49 21

Sorted array:

15 21 35 49 77 83 86 86 92 93

2. **Write a OpenMp program to illustrate data dependency (Binary Search).**

```
#include <stdio.h>

#include <omp.h>

#define ARRAY_SIZE 100

#define TARGET 42

int binary_search(int arr[], int left, int right, int target)

{

    while (left <= right)

    {

         int mid = left + (right - left) / 2;

        if (arr[mid] == target)          // Check if target is present at mid

            return mid;

        if (arr[mid] < target)          // If target greater, ignore left half

            left = mid + 1;

        else                            // If target is smaller, ignore right half

            right = mid - 1;

     }

    return -1;      // Target element is not present in the array

}

int main()

{

     int arr[ARRAY_SIZE];

    int i, result = -1;

    for (i = 0; i < ARRAY_SIZE; ++i)        // Initialize sorted array

    {

        arr[i] = i;

     }
```

```c
    // Parallel region for binary search
    #pragma omp parallel shared(arr, result)
    {
        int tid = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        int chunk_size = ARRAY_SIZE / num_threads;    // Calculate bounds for each thread
        int left = tid * chunk_size;
        int right = (tid == num_threads - 1) ? ARRAY_SIZE - 1 : (tid + 1) * chunk_size - 1;
        // Perform binary search within thread's bounds
        int local_result = binary_search(arr, left, right, TARGET);
        // Use reduction to find global result (if found)
        #pragma omp critical
        {
            if (local_result != -1 && result == -1)
                result = local_result;
        }
    }
    // Output the result
    if (result != -1)
        printf("Element %d found at index %d.\n", TARGET, result);
    else
        printf("Element %d not found in the array.\n", TARGET);
return 0;
}
```

Explanation:

1. Binary Search Function (`binary_search()`):

   - This function performs a binary search on a sorted array `arr[]` within the indices `left` to `right` to find the `target` element.

   - It returns the index of `target` if found, or `-1` if not found.

2. Main Function:

- ✓ Array Initialization: Initializes an array `arr[]` of size `ARRAY_SIZE` with sorted values (`arr[i] = i`).
- ✓ OpenMP Parallel Region:
  - `#pragma omp parallel shared(arr, result)`: This directive starts a parallel region. `shared(arr, result)` specifies that variables `arr[]` and `result` are shared among all threads.
  - `tid = omp_get_thread_num()` and `num_threads = omp_get_num_threads()`: These functions retrieve the current thread ID (`tid`) and the total number of threads (`num_threads`).
  - `chunk_size = ARRAY_SIZE / num_threads`: Calculates the chunk size for each thread based on the total array size and number of threads.
  - `left = tid * chunk_size` and `right = (tid == num_threads - 1) ? ARRAY_SIZE - 1 : (tid + 1) * chunk_size - 1`: Computes the bounds (`left` and `right`) for the binary search within each thread.
- ✓ Binary Search Within Each Thread:
  - Each thread executes `binary_search()` function with its calculated `left` and `right` bounds to search for `TARGET`.
- ✓ Critical Section (`#pragma omp critical`):
  - `#pragma omp critical`: Ensures that only one thread at a time can execute the critical section block.
  - Checks if `local_result` (result from the current thread) is valid (`local_result != -1`) and if `result` has not been set yet (`result == -1`).

- Updates `result` with `local_result` if the conditions are met.

✓ Output the Result:

- After all threads complete their binary search, the main thread checks if `result` is still `-1` (indicating `TARGET` was not found) or holds the index where `TARGET` was found.
- Prints the appropriate message based on whether `TARGET` was found or not.

Key Points:

- Parallelization Strategy: Each thread independently performs a binary search on its assigned portion of the array using OpenMP's `#pragma omp parallel` directive and `#pragma omp sections`.

- Data Sharing: `arr[]` and `result` are shared among threads. `result` is updated using a critical section to avoid race conditions.

- Efficiency Considerations: Parallelizing binary search can improve performance, especially for large arrays and a significant number of threads. However, the overhead of thread management and synchronization (such as critical sections) should be considered.

3. Write a OpenMp program to explore Loop Unrolling mechanism (Fibonacci numbers)

Loop unrolling in OpenMP can be applied to Fibonacci number calculation to potentially improve performance by reducing loop control overhead. Here's a simple example of how you can use the unroll` clause in OpenMP to unroll the loop for Fibonacci number generation:

```
#include <stdio.h>

#include <omp.h>

#define N 20

int main()

 {
```

```
int fib[N];

// Initialize the first two Fibonacci numbers
fib[0] = 0;
fib[1] = 1;

// Sequential computation of Fibonacci numbers
for (int i = 2; i < N; i++) {
    fib[i] = fib[i-1] + fib[i-2];
}

printf("Fibonacci sequence (sequential):\n");
for (int i = 0; i < N; i++) {
    printf("%d ", fib[i]);
}
printf("\n");

// Parallel computation of Fibonacci numbers with loop unrolling using OpenMP
#pragma omp parallel for unroll
for (int i = 2; i < N; i++) {
    fib[i] = fib[i-1] + fib[i-2];
```

```
    }

    printf("Fibonacci sequence (parallel with loop unrolling):\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", fib[i]);
    }
    printf("\n");
    return 0;
}
```

Explanation:

1. **Initialization:**

   - `fib[N]` array is used to store Fibonacci numbers up to the N-th number.

2. **Sequential Computation:**

   - Calculates Fibonacci numbers sequentially using a simple for loop.

   - The first two Fibonacci numbers (`fib[0]` and `fib[1]`) are initialized to 0 and 1 respectively.

   - The subsequent Fibonacci numbers are calculated using the formula `fib[i] = fib[i-1] + fib[i-2]`.

3. **Parallel Computation with Loop Unrolling:**

  - Uses OpenMP directive `#pragma omp parallel for unroll` to parallelize the computation of Fibonacci numbers with loop unrolling.

  - The `unroll` clause instructs OpenMP to unroll the loop iterations to reduce loop overhead and potentially improve performance.

  - Each thread will compute its assigned iterations of the Fibonacci sequence in parallel.

4. **Output:**

  - Prints the Fibonacci sequence computed sequentially and in parallel with loop unrolling.

Notes:

- **OpenMP `unroll` Clause:**

  - The `unroll` clause in `#pragma omp parallel for unroll` instructs the OpenMP compiler to unroll the loop iterations.

  - Loop unrolling may improve performance by allowing for better instruction pipelining and utilization of hardware resources.

- **Thread Safety:**

  - In this example, thread safety is maintained automatically by OpenMP since each thread works on its own portion of the `fib` array without explicit shared data modifications.

- **Performance Considerations:**

- The effectiveness of loop unrolling depends on factors such as the specific problem size (`N`), the number of iterations, and the hardware architecture.

- Experiment with different values of `N` to observe how loop unrolling affects performance.

This example demonstrates how to use the `unroll` clause in OpenMP to explore loop unrolling for calculating Fibonacci numbers, illustrating potential performance improvements through parallel execution with reduced loop overhead. Adjustments to the unrolling factor can be made based on specific performance requirements and hardware characteristics.

4. Write a OpenMp program to illustrate tomasulo's algorithm.

Implementing Tomasulo's algorithm, which is primarily used in pipelined processors for out-of-order execution with register renaming and reservation stations, is complex and typically involves simulating a processor's execution stages. However, I can provide a simplified example that demonstrates the essence of Tomasulo's algorithm using OpenMP to simulate multiple functional units executing operations concurrently.

In this example, we'll simulate a processor with multiple functional units (like adders and multipliers) and reservation stations where instructions (add, multiply, etc.) are dispatched for execution. We'll focus on scheduling instructions and updating register values, mimicking the dynamic execution of Tomasulo's algorithm.

```c
#include <stdio.h>

#include <omp.h>

#define NUM_UNITS 2

#define NUM_INSTRUCTIONS 6


typedef struct {

    char type;      // 'A' for Add, 'M' for Multiply

    int src1;       // Source operand 1

    int src2;       // Source operand 2

    int dest;       // Destination register

} Instruction;


typedef struct {

    char type;      // 'A' for Add, 'M' for Multiply

    int op1;        // Operand 1

    int op2;        // Operand 2

    int result;     // Result of operation

    int status;     // 0 if ready, 1 if executing

} FunctionalUnit;


Instruction instructions[NUM_INSTRUCTIONS] = {

    {'A', 2, 3, 1}, // Add R1 = R2 + R3

    {'M', 4, 5, 2}, // Multiply R2 = R4 * R5

    {'A', 1, 2, 3}, // Add R3 = R1 + R2

    {'M', 3, 3, 4}, // Multiply R4 = R3 * R3
```

```
    {'A', 5, 4, 5}, // Add R5 = R5 + R4
    {'M', 2, 2, 6}  // Multiply R6 = R2 * R2
};

FunctionalUnit units[NUM_UNITS];

void execute_instruction(int unit_id, Instruction inst) {
    // Simulate execution delay
    for (int i = 0; i < 100000000; ++i); // Delay loop (simulating execution time)

    // Perform the operation based on instruction type
    if (inst.type == 'A') {
        units[unit_id].result = inst.src1 + inst.src2;
    } else if (inst.type == 'M') {
        units[unit_id].result = inst.src1 * inst.src2;
    }

    // Mark the unit as ready
    units[unit_id].status = 0;
}

int main() {
    // Initialize functional units
    for (int i = 0; i < NUM_UNITS; ++i) {
        units[i].status = 0; // Initialize all units as idle
```

```
}

// Dispatch instructions to functional units (simulating Tomasulo's algorithm)
#pragma omp parallel for num_threads(NUM_INSTRUCTIONS)
for (int i = 0; i < NUM_INSTRUCTIONS; ++i) {
    // Find a free functional unit
    int unit_id = -1;
    #pragma omp critical
    {
        for (int j = 0; j < NUM_UNITS; ++j) {
            if (units[j].status == 0) {
                unit_id = j;
                units[j].status = 1; // Mark the unit as executing
                break;
            }
        }
    }

    if (unit_id != -1) {
        // Execute the instruction
        execute_instruction(unit_id, instructions[i]);
        printf("Executed instruction %d: Result = %d\n", i + 1, units[unit_id].result);
    } else {
        printf("No free functional unit available for instruction %d\n", i + 1);
    }
```

```
    }

    return 0;
}
```

Explanation:

1. Structures:

- `Instruction`: Represents an instruction with its type ('A' for Add, 'M' for Multiply), source operands (`src1`, `src2`), and destination register (`dest`).

- `FunctionalUnit`: Represents a functional unit (e.g., adder, multiplier) with its type ('A' for Add, 'M' for Multiply), operands (`op1`, `op2`), `result` of the operation, and `status` (0 if ready, 1 if executing).

2. Instructions:

- An array of `Instruction` structures representing a sequence of instructions to be executed.

3. Functional Units:

- An array of `FunctionalUnit` structures representing multiple functional units (e.g., adders and multipliers).

4. Execution Function (`execute_instruction`):

- Simulates the execution of an instruction by a functional unit. It calculates the result based on the instruction type and marks the unit as ready (`status = 0`) after execution.

5. Main Function:

- Initializes the functional units and then dispatches instructions to them using OpenMP for parallel execution.

- Each instruction is executed by finding a free functional unit, executing the instruction in parallel (simulating Tomasulo's algorithm's out-of-order execution), and printing the result.

Notes:

- ✓ Simulation Limitations:
  - This example simplifies Tomasulo's algorithm and does not include advanced features such as issue queues, register renaming, and dependency checking.
  - The execution delay loop (`for (int i = 0; i < 100000000; ++i);`) simulates the execution time of each instruction and can be adjusted for different simulation speeds.

- ✓ OpenMP Usage:
  - #pragma omp parallel for num_threads(NUM_INSTRUCTIONS)`: Distributes the loop iterations (instructions) across multiple threads. Each thread executes one instruction at a time, mimicking the concurrent execution of instructions in a pipelined processor.

- ✓ Thread Safety:
  - #pragma omp critical`: Ensures that only one thread at a time can access the loop to find a free functional unit, ensuring thread safety in accessing and updating shared resources (`units` array).

5. Write an OpenMp program which performs C=A+B & D=A-B in separate blocks/sections where A,B,C& D are arrays.

```
#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 */
```

<u>Output</u>

Number of threads = 4

Thread 0 starting...

Thread 0 doing section 1

Thread 0: c[0]= 22.350000

Thread 0: c[1]= 24.850000

Thread 0: c[2]= 27.350000

Thread 0: c[3]= 29.850000

Thread 0: c[4]= 32.349998

Thread 0: c[5]= 34.849998

Thread 0: c[6]= 37.349998

Thread 0: c[7]= 39.849998

Thread 0: c[8]= 42.349998

Thread 0: c[9]= 44.849998

Thread 3 starting...

Thread 3 doing section 2

Thread 3: d[0]= 0.000000

Thread 3: d[1]= 35.025002

Thread 3: d[2]= 73.050003

Thread 3: d[3]= 114.075005

Thread 3: d[4]= 158.100006

Thread 3: d[5]= 205.125000

Thread 3: d[6]= 255.150009

Thread 3: d[7]= 308.175018

Thread 3: d[8]= 364.200012

Thread 3: d[9]= 423.225006

Thread 3: d[10]= 485.249969

Thread 2 starting...

Thread 0: c[10]= 47.349998

Thread 1 starting...

Thread 1 done.

Thread 2 done.

Thread 0: c[11]= 49.849998

Thread 0: c[12]= 52.349998

Thread 0: c[13]= 54.849998

Thread 0: c[14]= 57.349998

Thread 0: c[15]= 59.849998

Thread 0: c[16]= 62.349998

Thread 0: c[17]= 64.849998

Thread 0: c[18]= 67.349998

Thread 0: c[19]= 69.849998

Thread 0: c[20]= 72.349998

Thread 0: c[21]= 74.849998

Thread 0: c[22]= 77.349998

Thread 0: c[23]= 79.849998

Thread 0: c[24]= 82.349998

Thread 3: d[11]= 550.274963

Thread 3: d[12]= 618.299988

Thread 3: d[13]= 689.324951

Thread 3: d[14]= 763.349976

Thread 0: c[25]= 84.849998

Thread 0: c[26]= 87.349998

Thread 0: c[27]= 89.849998

Thread 0: c[28]= 92.349998

Thread 0: c[29]= 94.849998

Thread 0: c[30]= 97.349998

Thread 0: c[31]= 99.849998

Thread 0: c[32]= 102.349998

Thread 0: c[33]= 104.849998

Thread 0: c[34]= 107.349998

Thread 0: c[35]= 109.849998

Thread 0: c[36]= 112.349998

Thread 0: c[37]= 114.849998

Thread 0: c[38]= 117.349998

Thread 0: c[39]= 119.849998

Thread 0: c[40]= 122.349998

Thread 0: c[41]= 124.849998

Thread 0: c[42]= 127.349998

Thread 0: c[43]= 129.850006

Thread 0: c[44]= 132.350006

Thread 0: c[45]= 134.850006

Thread 0: c[46]= 137.350006

Thread 0: c[47]= 139.850006

Thread 0: c[48]= 142.350006

Thread 0: c[49]= 144.850006

Thread 3: d[15]= 840.374939

Thread 3: d[16]= 920.399963

Thread 3: d[17]= 1003.424988

Thread 3: d[18]= 1089.449951

Thread 3: d[19]= 1178.474976

Thread 3: d[20]= 1270.500000

Thread 3: d[21]= 1365.524902

Thread 3: d[22]= 1463.549927

Thread 3: d[23]= 1564.574951

Thread 3: d[24]= 1668.599976

Thread 3: d[25]= 1775.625000

Thread 0 done.

Thread 3: d[26]= 1885.649902

Thread 3: d[27]= 1998.674927

Thread 3: d[28]= 2114.699951

Thread 3: d[29]= 2233.724854

Thread 3: d[30]= 2355.750000

Thread 3: d[31]= 2480.774902

Thread 3: d[32]= 2608.799805

Thread 3: d[33]= 2739.824951

Thread 3: d[34]= 2873.849854

Thread 3: d[35]= 3010.875000

Thread 3: d[36]= 3150.899902

Thread 3: d[37]= 3293.924805

Thread 3: d[38]= 3439.949951

Thread 3: d[39]= 3588.974854

Thread 3: d[40]= 3741.000000

Thread 3: d[41]= 3896.024902

Thread 3: d[42]= 4054.049805

Thread 3: d[43]= 4215.074707

Thread 3: d[44]= 4379.100098

Thread 3: d[45]= 4546.125000

Thread 3: d[46]= 4716.149902

Thread 3: d[47]= 4889.174805

Thread 3: d[48]= 5065.199707

Thread 3: d[49]= 5244.225098

Thread 3 done.

****
***

6. Write an OpenMp program to add all the elements of two arrays A & B each of size 1000 and store their sum in a variable 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);

}
```

Output

Sum = 332833152.000000

****

7. Write an OpenMp program to 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");

}
```

Output

```
Starting matrix multiple example with 4 threads

Initializing matrices...

Thread 0 starting matrix multiply...

Thread 3 starting matrix multiply...

Thread=3 did row=30

Thread=3 did row=31

Thread=3 did row=32

Thread=3 did row=33

Thread=3 did row=34

Thread=3 did row=35

Thread=0 did row=0

Thread 2 starting matrix multiply...

Thread 1 starting matrix multiply...

Thread=3 did row=36
```

Thread=3 did row=37

Thread=3 did row=38

Thread=2 did row=20

Thread=1 did row=10

Thread=0 did row=1

Thread=0 did row=2

Thread=0 did row=3

Thread=2 did row=21

Thread=0 did row=4

Thread=2 did row=22

Thread=3 did row=39

Thread=1 did row=11

Thread=0 did row=5

Thread=1 did row=12

Thread=1 did row=13

Thread=1 did row=14

Thread=1 did row=15

Thread=1 did row=16

Thread=1 did row=17

Thread=1 did row=18

Thread=1 did row=19

Thread=1 did row=50

Thread=1 did row=51

Thread=1 did row=52

Thread=1 did row=53

Thread=1 did row=54

Thread=1 did row=55

Thread=1 did row=56

Thread=1 did row=57

Thread=1 did row=58

Thread=1 did row=59

Thread=2 did row=23

Thread=2 did row=24

Thread=2 did row=25

Thread=0 did row=6

Thread=0 did row=7

Thread=0 did row=8

Thread=0 did row=9

Thread=2 did row=26

Thread=2 did row=27

Thread=2 did row=28

Thread=2 did row=29

Thread=2 did row=60

Thread=2 did row=61

Thread=0 did row=40

Thread=0 did row=41

Thread=0 did row=42

Thread=0 did row=43

Thread=0 did row=44

Thread=0 did row=45

Thread=0 did row=46

Thread=0 did row=47

Thread=0 did row=48

Thread=0 did row=49

****

Result Matrix:

0.00 1015.00 2030.00 3045.00 4060.00 5075.00 6090.00

0.00 1120.00 2240.00 3360.00 4480.00 5600.00 6720.00

0.00 1225.00 2450.00 3675.00 4900.00 6125.00 7350.00

0.00 1330.00 2660.00 3990.00 5320.00 6650.00 7980.00

0.00 1435.00 2870.00 4305.00 5740.00 7175.00 8610.00

0.00 1540.00 3080.00 4620.00 6160.00 7700.00 9240.00

0.00 1645.00 3290.00 4935.00 6580.00 8225.00 9870.00

0.00 1750.00 3500.00 5250.00 7000.00 8750.00 10500.00

0.00 1855.00 3710.00 5565.00 7420.00 9275.00 11130.00

0.00 1960.00 3920.00 5880.00 7840.00 9800.00 11760.00

0.00 2065.00 4130.00 6195.00 8260.00 10325.00 12390.00

0.00 2170.00 4340.00 6510.00 8680.00 10850.00 13020.00

0.00 2275.00 4550.00 6825.00 9100.00 11375.00 13650.00

0.00 2380.00 4760.00 7140.00 9520.00 11900.00 14280.00

0.00 2485.00 4970.00 7455.00 9940.00 12425.00 14910.00

0.00 2590.00 5180.00 7770.00 10360.00 12950.00 15540.00

0.00 2695.00 5390.00 8085.00 10780.00 13475.00 16170.00

0.00 2800.00 5600.00 8400.00 11200.00 14000.00 16800.00

0.00 2905.00 5810.00 8715.00 11620.00 14525.00 17430.00

0.00 3010.00 6020.00 9030.00 12040.00 15050.00 18060.00

0.00 3115.00 6230.00 9345.00 12460.00 15575.00 18690.00

0.00 3220.00 6440.00 9660.00 12880.00 16100.00 19320.00

0.00 3325.00 6650.00 9975.00 13300.00 16625.00 19950.00

0.00 3430.00 6860.00 10290.00 13720.00 17150.00 20580.00

0.00 3535.00 7070.00 10605.00 14140.00 17675.00 21210.00

0.00 3640.00 7280.00 10920.00 14560.00 18200.00 21840.00

0.00 3745.00 7490.00 11235.00 14980.00 18725.00 22470.00

0.00 3850.00 7700.00 11550.00 15400.00 19250.00 23100.00

0.00 3955.00 7910.00 11865.00 15820.00 19775.00 23730.00

0.00 4060.00 8120.00 12180.00 16240.00 20300.00 24360.00

0.00 4165.00 8330.00 12495.00 16660.00 20825.00 24990.00

0.00 4270.00 8540.00 12810.00 17080.00 21350.00 25620.00

0.00 4375.00 8750.00 13125.00 17500.00 21875.00 26250.00

0.00	4480.00	8960.00	13440.00	17920.00	22400.00	26880.00
0.00	4585.00	9170.00	13755.00	18340.00	22925.00	27510.00
0.00	4690.00	9380.00	14070.00	18760.00	23450.00	28140.00
0.00	4795.00	9590.00	14385.00	19180.00	23975.00	28770.00
0.00	4900.00	9800.00	14700.00	19600.00	24500.00	29400.00
0.00	5005.00	10010.00	15015.00	20020.00	25025.00	30030.00
0.00	5110.00	10220.00	15330.00	20440.00	25550.00	30660.00
0.00	5215.00	10430.00	15645.00	20860.00	26075.00	31290.00
0.00	5320.00	10640.00	15960.00	21280.00	26600.00	31920.00
0.00	5425.00	10850.00	16275.00	21700.00	27125.00	32550.00
0.00	5530.00	11060.00	16590.00	22120.00	27650.00	33180.00
0.00	5635.00	11270.00	16905.00	22540.00	28175.00	33810.00
0.00	5740.00	11480.00	17220.00	22960.00	28700.00	34440.00
0.00	5845.00	11690.00	17535.00	23380.00	29225.00	35070.00
0.00	5950.00	11900.00	17850.00	23800.00	29750.00	35700.00
0.00	6055.00	12110.00	18165.00	24220.00	30275.00	36330.00
0.00	6160.00	12320.00	18480.00	24640.00	30800.00	36960.00
0.00	6265.00	12530.00	18795.00	25060.00	31325.00	37590.00
0.00	6370.00	12740.00	19110.00	25480.00	31850.00	38220.00
0.00	6475.00	12950.00	19425.00	25900.00	32375.00	38850.00

0.00 6580.00 13160.00 19740.00 26320.00 32900.00 39480.00

0.00 6685.00 13370.00 20055.00 26740.00 33425.00 40110.00

0.00 6790.00 13580.00 20370.00 27160.00 33950.00 40740.00

0.00 6895.00 13790.00 20685.00 27580.00 34475.00 41370.00

0.00 7000.00 14000.00 21000.00 28000.00 35000.00 42000.00

0.00 7105.00 14210.00 21315.00 28420.00 35525.00 42630.00

0.00 7210.00 14420.00 21630.00 28840.00 36050.00 43260.00

0.00 7315.00 14630.00 21945.00 29260.00 36575.00 43890.00

0.00 7420.00 14840.00 22260.00 29680.00 37100.00 44520.00

****

8. Write an OpenMp 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 */


}
```

Output

1st Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 1: a,b,x= 1 1 2.100000

Thread 2: a,b,x= 2 2 3.200000

Thread 3: a,b,x= 3 3 4.300000

****

Master thread doing serial work here

****

2nd Parallel Region:

Thread 3: a,b,x= 3 0 4.300000

Thread 2: a,b,x= 2 0 3.200000

Thread 1: a,b,x= 1 0 2.100000

Thread 0: a,b,x= 0 0 1.000000

****

9. Write an OpenMp 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;
}
```

Output

4 threads

```
factorial( 0)=              1 x[ 0]=              1
factorial( 1)=              1 x[ 1]=              1
factorial( 2)=              2 x[ 2]=              3
factorial( 3)=              6 x[ 3]=             18
factorial( 4)=             24 x[ 4]=             72
factorial( 5)=            120 x[ 5]=            648
factorial( 6)=            720 x[ 6]=           9720
factorial( 7)=           5040 x[ 7]=           5040
factorial( 8)=          40320 x[ 8]=          75600
factorial( 9)=         362880 x[ 9]=        1814400
factorial(10)=        3628800 x[10]=        3628800
factorial(11)=       39916800 x[11]=       76204800
```

****
***

10. Write an OpenMP 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);


}
```

Output

4 threads max

5.11 seconds

5761455 primes between 0 and 100000000

****