Practical Workbook
CS-428
Parallel and Distributed Computing

Name : _____

Year : _____

Batch : _____

Roll No : _____

Department: _____

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

INTRODUCTION

This workbook has been compiled to assist the conduct of practical classes for CS-428 Parallel and Distributed Computing. Practical work relevant to this course aims at providing students a chance to write parallel algorithms and their programming on shared and distributed memory environments. Parallel and distributed computing is considered to be one of the most exciting technologies to achieve prominence since the invention of computers. The pervasiveness of computing devices containing multicore CPUs and GPUs, including home and office PCs, laptops, and mobile devices, is making even common users dependent on parallel processing. Certainly, it is no longer sufficient for even basic programmers to acquire only the traditional sequential programming skills. The preceding trends point to the need for imparting a broad-based skill set in parallel and distributed computing technology. However, the rapid changes in computing hardware platforms and devices, languages, supporting programming environments, and research advances, poses a challenge both for newcomers and seasoned computer scientists. Programming a parallel and distributed system is not as easy as programming single processor systems. There are many considerations like details of the underlying parallel system, processors interconnection, concurrency, transparency, heterogeneity and selection of appropriate platform which makes the programming of parallel and distributed systems more difficult.

The Course Profile of CS-428 Parallel and Distributed Computing lays down the following Course Learning Outcome:

"Examine existing techniques for parallel & distributed computing (C3, PLO-2)"

"Analyze parallel and distributed computing solutions for their pros and cons (C4, PLO-4)"

Lab sessions 1, 2, 3, 7, 8, 9, 11, 12, 13, and 14 of this workbook have been designed to assist the achievement of the CLO-1. While lab sessions 4, 5, 6, and 10 have been designed for CLO-2. A rubric to evaluate student performance has been provided at the end of the workbook.

Part one of this lab manual is based on shared memory programming. It deals with the programming of both the SIMD and multicore systems respectively. The shared memory programming is done by using openmp programming model. The first lab gives a basic introduction to openmp API. Lab session 2, 3 and 4 deals with openmp work sharing constructs

and performance analysis of real world applications. Lab session 5 and 6 enables the exploitation of vector processing units present in General Purpose Processors (GPPs) by performing SIMD programming using openmp. Part two of this workbook deals with distributed programming. Lab session 7 gives an introduction to MPI programming model. It is followed by lab session 8 which introduces the MPI communication operations. In lab session 9 MPI collective operations are explored whereas lab session 10 discusses the programming of real world applications using MPI. Apart from covering MPI for distributed memory programming, this workbook also covers some other paradigms. Lab session 11 and 12 deal with socket programming in Linux. Java RMI is covered in lab session 13 and implementation of web service using Restful API is discussed in lab session 14.

CONTENTS

Lab Session 1

Acquire basic OpenMP (Open Multi-Processor) Principles

OpenMP Programming Model

OpenMP is a portable and standard Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP attempts to standardize existing practices from several different vendor-specific shared memory environments. OpenMP provides a portable shared memory API across different platforms including DEC, HP, IBM, Intel, Silicon Graphics/Cray, and Sun. The languages supported by OpenMP are FORTRAN, C and C++. Its main emphasis is on performance and scalability.

Shared Memory, Thread Based Parallelism

- OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Fork - Join Model:

- OpenMP uses the fork-join model of parallel execution:

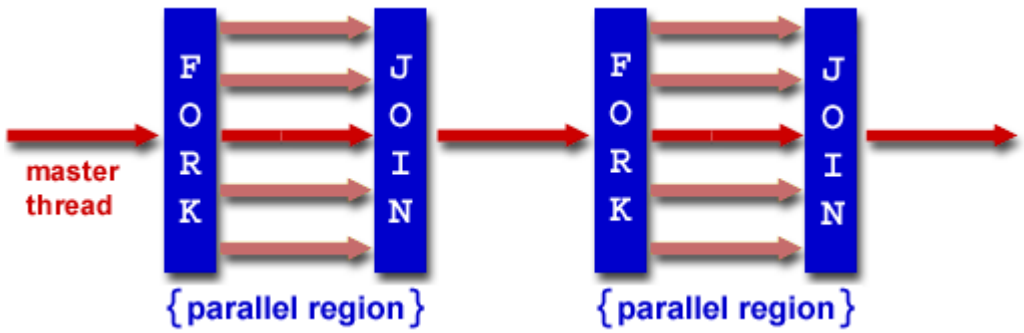


Figure 1.1. Fork and Join Model

- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a ***team*** of parallel threads
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Compiler Directive Based:

- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

Nested Parallelism Support:

- The API provides for the placement of parallel constructs inside of other parallel constructs.
- Implementations may or may not support this feature.

Dynamic Threads:
- The API provides for dynamically altering the number of threads which may used to execute different parallel regions.
- Implementations may or may not support this feature.

Components of OpenMP API
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

Goals of OpenMP
- **Standardization:** Provide a standard among a variety of shared memory architectures/platforms
- **Lean and Mean:** Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:** Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach and also provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:** Supports Fortran (77, 90, and 95), C, and C++

C / C++ - General Code Structure

```
#include <omp.h>
 main (){
       int var1, var2, var3;
       Serial code
        #pragma omp parallel private(var1, var2) shared(var3)
             { Parallel section executed by all threads
                          .
                          .
                          .
                All threads join master thread and disband }
       Resume serial code
         }
```

Important terms for an OpenMP environment

Construct: A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct.

Directive: A C or C++ **#pragma** followed by the **omp** identifier, other text, and a new line. The directive specifies program behavior.

Region: A dynamic extent.

Dynamic Extent: All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

Lexical Extent: Statements lexically contained within a *structured block*.

Structured Block: A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement. A compound statement is a structured block if its execution always begins at the opening **{** and always ends at the closing **}**. An expression statement, selection statement, iteration statement is a structured block if the corresponding

compound statement obtained by enclosing it in **{** and **}**would be a structured block. A jump statement, labeled statement, or declaration statement is not a structured block.

Thread: An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

Master thread: The thread that creates a team when a *parallel region* is entered.

Serial Region: Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

Parallel Region: Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.

Variable: An identifier, optionally qualified by namespace names, that names an object.

Private: A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a threadprivate directive, a private, firstprivate, lastprivate, or reduction clause, or use of the variable as a forloop control variable in a for loop immediately following a for or parallel for directive.

Shared: A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.

Team: One or more threads cooperating in the execution of a construct.

Serialize: To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **omp_in_parallel()** (apart from the effects of any nested parallel constructs).

Barrier: A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

OpenMP Directives Format
OpenMP directives for C/C++ are specified with the pragma preprocessing directive.

#pragma omp directive-name [clause[[,] clause]. . .] new-line

Where:

- **#pragma omp:** Required for all OpenMP C/C++ directives.
- **directive-name:** A valid OpenMP directive. Must appear after the pragma and before any clauses.
- **[clause, ...]:** Optional, Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- **Newline:** Required, Precedes the structured block which is enclosed by this directive.

OpenMP Directives or Constructs

- Parallel Construct
- Work-Sharing Constructs
  - Loop Construct
  - Sections Construct
  - Single Construct
- Data-Sharing, No Wait, and Schedule Clauses
- Barrier Construct
- Critical Construct
- Atomic Construct
- Locks
- Master Construct

Directive Scoping

Static (Lexical) Extent:
- The code textually enclosed between the beginning and the end of a structured block following a directive.
- The static extent of a directives does not span multiple routines or code files.

Orphaned Directive:
- An OpenMP directive that appears independently from another enclosing directive is said to be an orphaned directive. It exists outside of another directive's static (lexical) extent.
- Will span routines and possibly code files

Dynamic Extent:
- The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Parallel Construct

This construct is used to specify the computations that should be executed in parallel. Parts of the program that are not enclosed by a parallel construct will be executed serially. When a thread encounters this construct, a team of threads is created to execute the associated parallel region, which is the code dynamically contained within the parallel construct. But although this construct ensures that computations are performed in parallel, it does not distribute the work of the region among the threads in a team. In fact, if the programmer does not use the appropriate syntax to specify this action, the work will be replicated. At the end of a parallel region, there is an implied *barrier* that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel region.

The thread that encounters the parallel construct becomes the *master* of the new team. Each thread in the team is assigned a unique thread number (also referred to as the "thread id") to identify it. They range from zero (for the master thread) up to one less than the number of threads within the team, and they can be accessed by the programmer. Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution.

Format

```
#pragma omp parallel [clause ...] ……. newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)
{
 structured_block
            }
```

Preparing Visual Studio for OpenMP Programming
- Create a new Visual C++ project, select the Win32 Console Application.
- In the application settings view, uncheck the Precompiled Header option.
- Right click the project, go to Properties, C/C++ Language, turn Open MP Support to Yes.

Program 1

```
#include <omp.h>
main(){
#pragma omp parallel {
```

```
printf("The parallel region is executed by thread %d\n",
omp_get_thread_num());
} /*-- End of parallel region --*/
}/*-- End of Main Program --*/
```

Here, the OpenMP library function **omp_get_thread_num()** is used to obtain the number of each thread executing the parallel region. Each thread will execute all code in the parallel region, so that we should expect each to perform the print statement.. Note that one cannot make any assumptions about the order in which the threads will execute the printf statement. When the code is run again, the order of execution could be different.

Clauses supported by the parallel construct

- if*(scalar-expression)*
- num threads*(integer-expression)*
- private*(list)*
- firstprivate*(list)*
- shared*(list)*
- default(none|shared)
- copyin*(list)*
- reduction*(operator:list)*

Determining the Number of Threads for a parallel Region

When execution encounters a parallel directive, the value of the if clause or num_threads clause (if any) on the directive, the current parallel context, and the values of the nthreads-var, dyn-var, thread-limit-var, max-active-level-var, and nest-var ICVs are used to determine the number of threads to use in the region.

Note that using a variable in an if or num_threads clause expression of a parallel construct causes an implicit reference to the variable in all enclosing constructs. The if clause expression and the num_threads clause expression are evaluated in the context outside of the parallel construct, and no ordering of those evaluations is specified. It is also unspecified whether, in what order, or how many times any side-effects of the evaluation of the num_threads or if clause expressions occur.

Specifying a Fixed Number of Threads

Some programs rely on a fixed, pre-specified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation-defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using omp_set_dynamic and omp_set_num_threads.

Program 2

```
#include <omp.h>
main(){
omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma omp parallel num_threads(10){
printf("Num threads in parallel region=%d\n", omp_get_num_threads());
} }
```

Program 3

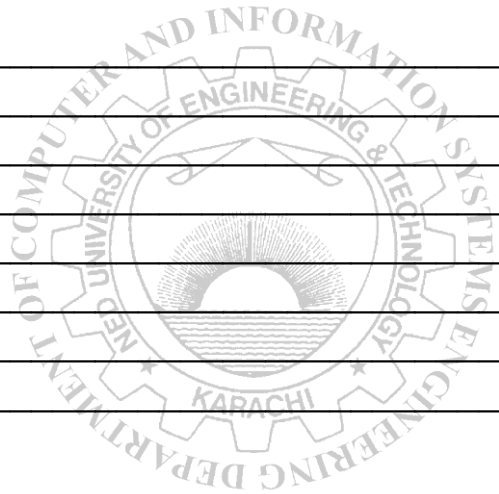
```
#include <omp.h>
int main() {
omp_set_dynamic(1);
omp_set_num_threads(10);
#pragma omp parallel //parallel region 1
printf("Num threads in dynamic region=%d\n", omp_get_num_threads());
```

```
omp_set_dynamic(0);
omp_set_num_threads(10);
#pragma omp parallel //parallel region 2
printf("Num threads in non-dynamic region=%d\n",
omp_get_num_threads());
omp_set_dynamic(1);
#pragma omp parallel //parallel region 3
#pragma omp parallel
printf("Num threads in nesting disabled region=%d\n",
omp_get_num_threads());
omp_set_nested(1);
#pragma omp parallel //parallel region 4
#pragma omp parallel
printf("Num threads in nested region=%d\n", omp_get_num_threads()); }
```

EXERCISE:

1. Code the above example programs and paste the printout of source codes and their outputs.

Output of Program 01:



Output of Program 02:

Output of Program 03:

2. Write an infinite loop within OpenMP parallel directive. Paste the printout of multicore performance observed in task manger during program execution. Also, paste the source code.

Program Code:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 2

Explore the OpenMP Loop Construct

OpenMP Work-Sharing Construct

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed below.

- **#pragma omp for:** Distribute iterations over the threads
- **#pragma omp sections:** Distribute independent work units
- **#pragma omp single:** Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the nowait clause.

The Loop Construct

The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel. At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.

Format:

```
#pragma omp for [clause ...]  newline
                      schedule (type [,chunk])
                      ordered
                      private (list)
                      firstprivate (list)
                      lastprivate (list)
                      shared (list)
                      reduction (operator: list)
                      nowait
   for_loop
```

Example-1: Work-sharing loop

Each thread executes a subset of the total iteration space $i = 0, \ldots, n-1$

```
#include <omp.h>
main()
{
#pragma omp parallel shared(n) private(i)
{
#pragma omp for
for (i=0; i<n; i++)
printf("Thread %d executes loop iteration %d\n",
omp_get_thread_num(),i);
}
}
```

Here we use a parallel directive to define a parallel region and then share its work among threads via the for work-sharing directive: the **#pragma omp for** directive states that iterations of the loop following it will be distributed. Within the loop, we use the OpenMP function omp_get_thread_num(), this time to obtain and print the number of the executing thread in each iteration. Parallel construct that state which data in the region is shared and which is private. Although not strictly needed since this is enforced by the compiler, loop variable i is explicitly declared to be a private variable, which means that each thread will have its own copy of i. its value is also undefined after the loop has finished. Variable n is made shared.

Output from the example which is executed for $n = 9$ and uses four threads.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the parallel construct immediately followed by the work-sharing construct.

Full version Combined construct	Combined construct

`#pragma omp parallel`
`{`
`  #pragma omp for`
`  for-loop`
`}`
	`#pragma omp parallel for`
	`{`
	`  for-loop`
	`}`

Clauses in Loop Construct

- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped.
- Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.
- These constructs provide the ability to control the data environment during execution of parallel constructs.

- They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
- They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.

List of Clauses
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION

PRIVATE Clause

The PRIVATE clause declares variables in its list to be private to each thread.

- **Format:** PRIVATE (list)
- PRIVATE variables behave as follows:
  - A new object of the same type is declared once for each thread in the team
  - All references to the original object are replaced with references to the new object
  - Variables declared PRIVATE are uninitialized for each thread

Example-2: Private Clause – Each thread has a local copy of variables i and a.

```
#pragma omp parallel for private(i,a)
for (i=0; i<n; i++)
{ a = i+1;
  printf("Thread %d has a value of a = %d for i = %d\n",
omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
```

SHARED Clause

The SHARED clause declares variables in its list to be shared among all threads in the team.

- **Format:** SHARED *(list)*
- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

Example-3: Shared Clause – All threads can read from and write to vector a.

```
#pragma omp parallel for shared(a)
for (i=0; i<n; i++)
{
  a[i] += i;
} /*-- End of parallel for --*/
```

DEFAULT Clause

The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope for all variables in the lexical extent of any parallel region.

- The default clause is used to give variables a default data-sharing attribute. Its usage is straightforward. For example, default (shared) assigns the shared attribute to all variables referenced in the construct. This clause is most often used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed.

- If default (none) is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct. Although variables with a predetermined data-sharing attribute need not be listed in one of the clauses, it is strongly recommend that the attribute be explicitly specified for *all* variables in the construct.

- **Format:** DEFAULT (SHARED | NONE)
- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
- The C/C++ OpenMP specification does not include "private" as a possible default. However, actual implementations may provide this option.
- Only one DEFAULT clause can be specified on a PARALLEL directive

Example-4: Default Clause: all variables to be shared, with the exception of a, b, and c.

```
#pragma omp for default(shared) private(a,b,c),
```

FIRSTPRIVATE Clause

The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

- **Format:** FIRSTPRIVATE *(LIST)*
- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct

Example-5: Firstprivate Clause – Each thread has a pre-initialized copy of variable indx. This variable is still private, so threads can update it individually.

```
for(i=0; i<vlen; i++) a[i] = -i-1;
indx = 4;
{
#pragma omp parallel default(none) firstprivate(indx) private(i,TID)
shared(n,a)
{
TID = omp_get_thread_num();
indx += n*TID;
for(i=indx; i<indx+n; i++)
a[i] = TID + 1;
}
}
printf("After the parallel region:\n");
for (i=0; i<vlen; i++)
printf("a[%d] = %d\n",i,a[i]);
```

LASTPRIVATE Clause

The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

- **Format:** LASTPRIVATE *(LIST)*
- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.

Example-6: Lastprivate Clause – This clause makes the sequentially last value of variable a accessible outside the parallel loop.

```
#pragma omp parallel for private(i) lastprivate(a)
for (i=0; i<n; i++)
{
a = i+1;
printf("Thread %d has a value of a = %d for i = %d\n",
omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
printf("Value of a after parallel for: a = %d\n",a);
```

REDUCTION Clause

The REDUCTION clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

- **Format:** REDUCTION *(OPERATOR: LIST)*
- Variables in the list must be named scalar variables. They cannot be array or structure type variables. They must also be declared SHARED in the enclosing context.
- Reduction operations may not be associative for real numbers.
- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:

C / C++
x = x op expr **x = expr op x** (except subtraction) **x binop = expr** **x++** **++x** **x--** **--x**
x is a scalar variable in the list ***expr*** is a scalar expression that does not reference ***x*** ***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, || ***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

Example-7: Reduction Clause- Vector Dot Product. Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (SCHEDULE STATIC). At the end of the parallel loop construct, all threads will add their values of "result" to update the master thread's global copy.

```
#include <omp.h>
main ()
{
      int   i, n, chunk;
      float a[100], b[100], result;
      n = 100;
      chunk = 10;
      result = 0.0;
      for (i=0; i < n; i++)
        {
        a[i] = i * 1.0;
```

```
        b[i] = i * 2.0;
        }
      #pragma omp parallel for  default(shared) private(i)
      schedule(static,chunk) reduction(+:result)
        {
           for (i=0; i < n; i++)
           result = result + (a[i] * b[i]);

      printf("Final result= %f\n",result);
      }
}
```

SCHEDULE Clause
Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC
Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC
Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED
For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

NOWAIT Clause
The nowait clause allows the programmer to fine-tune a program's performance. When we introduced the work-sharing constructs, we mentioned that there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP; in other words, if it is added to a construct, the barrier at the end of the associated construct will be suppressed. When threads reach the end of the construct, they will immediately proceed to perform other work. Note, however, that the barrier at the end of a parallel region cannot be suppressed.

Example-8: Nowait clause in C/C++ – The clause ensures that there is no barrier at the end of the loop.

```
#pragma omp for nowait
     for (i=0; i<n; i++)
     {
     ............
     }
```

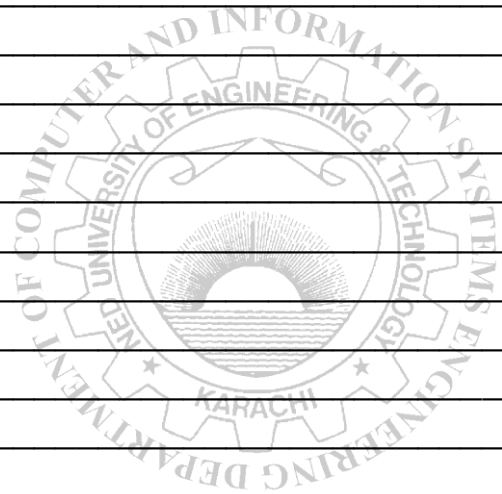
EXERCISE:

1. Code the following parallel program and paste the printout of source code and its output.

```
main() {
int i, sum=0;
#pragma omp parallel private(i) shared(sum){
```

```
#pragma omp for
for (i=1; i<=10; ++i) {
#pragma omp atomic
sum += i; } }
printf("The sum of 1 through 10 is %d\n", sum); }
```

Program and Output:



2. Code the following parallel program and paste the printout of source code and its output.

```
main() {
int n, i;
printf("not using the barrier\n\n");
#pragma omp parallel
{ i=0;
#pragma omp for firstprivate(i) nowait
```

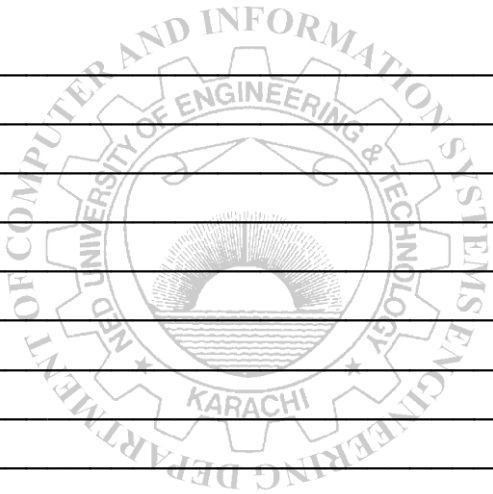
```
for(n=0; n<10; ++n)
{printf("thread# %d says %d, this is my iteration# %d\n\n",
omp_get_thread_num(),n,i);
i++;}
printf("thread# %d says i'm done\n\n",omp_get_thread_num());
}
printf("using the barrier now\n\n");
#pragma omp parallel
{i=0;
#pragma omp for firstprivate(i) lastprivate(i)
for(n=0; n<10; ++n)
{printf("thread# %d says %d, this is my iteration# %d\n\n",
omp_get_thread_num(),n,i);
i++;}
printf("thread# %d says i'm done\n\n",omp_get_thread_num());
#pragma omp single
printf("value of i is %d\n\n",i); } }
```

Program and Output:

3. Code the following parallel program and paste the printout of source code and its output.

```
main() {
#pragma omp parallel
{
#pragma omp single
printf("static shedule:\n");
#pragma omp for schedule(static,5)
for(int n=0; n<23; ++n)
printf("thread# %d says %d\n", omp_get_thread_num(),n);
#pragma omp single
printf("dynamic shedule:\n");
#pragma omp for schedule(dynamic,5)
for(int n=0; n<23; ++n)
printf("thread# %d says %d\n", omp_get_thread_num(),n); } }
```

Program and Output:



4. Code the following parallel program and paste the printout of source code and its output.

```
main (int argc, char *argv[]) {
double a[N]; double sum=0.0; int i,n,tid;
#pragma omp parallel shared(a) private(i)
{tid=omp_get_thread_num();
#pragma omp single
printf("Number of threads = %d\n",omp_get_num_threads());
#pragma omp for
for(i=0;i<N;i++) a[i]=i+1;
#pragma omp for reduction(+:sum)
for(i=0;i<N; i++) sum=sum+a[i]; } // sums of all threads are added up
into one variable
printf("Sum = %2.1f\n",sum);}
```

Program and Output:

5. Code the above example (1 to 8) programs and paste the printout of source codes and their outputs.

Programs and Outputs:

Lab Session 3

Explore the OpenMP Sections and Single Construct

The Sections Construct

The *sections construct* is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads. It consists of two directives: first, **#pragma omp sections:** to indicate the start of the construct and second, **the #pragma omp section:** to mark each distinct section. Each section must be a structured block of code that is independent of the other sections.

At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks. If there are fewer code blocks than threads, the remaining threads will be idle. Note that the assignment of code blocks to threads is implementation-dependent.

Format:

```
#pragma omp sections [clause ...] newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
          {
            #pragma omp section  newline
                 structured_block
            #pragma omp section  newline
               structured_block
            }
```

Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the parallel construct immediately followed by the work-sharing construct.

Full version Combined construct	Combined construct

#pragma omp parallel
{
#pragma omp sections
{
[**#pragma omp section**]
structured block
[**#pragma omp section**
structured block
}
}
	#pragma omp parallel sections
	{
	[**#pragma omp section**]
	structured block
	[**#pragma omp section]**
	structured block
	. . .
	}

Example-1: Parallel Sections
If two or more threads are available, one thread invokes funcA() and another thread calls funcB(). Any other threads are idle.

```
#include <omp.h>
main()
{
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
(void) funcA();
#pragma omp section
(void) funcB();
} /*-- End of sections block --*/
} /*-- End of parallel region --*/
} /*-- End of Main Program --*/
void funcA()
{
printf("In funcA: this section is executed by thread %d\n",
omp_get_thread_num());
}
void funcB()
{
printf("In funcB: this section is executed by thread %d\n",
omp_get_thread_num());
}
```

Output from the example; the code is executed by using two threads.
In funcA: this section is executed by thread 0
In funcB: this section is executed by thread 1

The Single Construct

The *single construct* is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only. It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another. It can also differ for different single constructs within one application. This construct should really be used when we do not care which thread executes this part of the application, as long as the work gets done by exactly one thread. The other threads wait at a barrier until the thread executing the single code block has completed.

Format:
#pragma omp single [clause ...] newline
private (list)
firstprivate (list)
nowait
structured_block

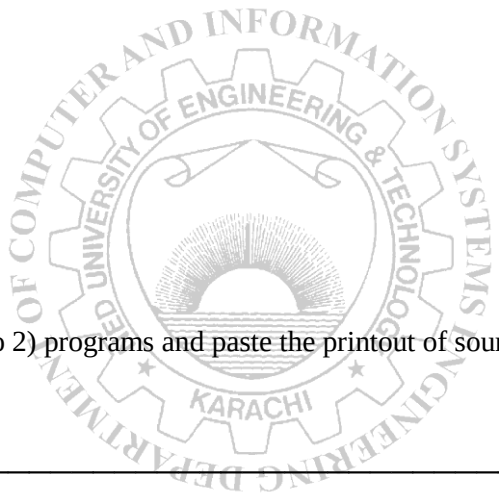
Example-2: Single Construct
Only one thread initializes the shared variable a.

```
#include <omp.h>
main()
{
```

```
#pragma omp parallel shared(a,b) private(i)
{
#pragma omp single
{
a = 10;
printf("Single construct executed by thread %d\n",
omp_get_thread_num());
}
#pragma omp for
for (i=0; i<n; i++)
b[i] = a;
} /*-- End of parallel region --*/
printf("After the parallel region:\n");
for (i=0; i<n; i++)
printf("b[%d] = %d\n",i,b[i]);
}
```

Output from the example, the value of variable *n* is set to 9, and four threads are used.

b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10



EXERCISE:

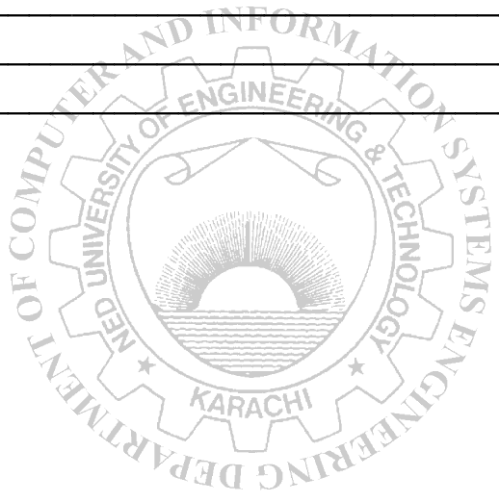
1. Code the above example (1 to 2) programs and paste the printout of source codes and their outputs.

Programs and Outputs:

2. Code the following parallel program and paste the printout of source code and its output.

```
main(int argc, const char **argv)
{
printf("This is the main thread.\n");
omp_set_dynamic(0);
int id;
#pragma omp parallel private(id) num_threads(6)
{
id=omp_get_thread_num();
#pragma omp single
{
printf("Total number of threads:"<<# %d \n", omp_get_num_threads());
printf("In single directive, this is thread# %d \n", id);}
#pragma omp sections
{
#pragma omp section
printf("In section a, this is thread# %d \n", id);
#pragma omp section
printf("In section b, this is thread# %d \n", id);
#pragma omp section
printf("In section c, this is thread# %d \n", id);
}}
printf("Back to the main thread. Goodbye!\n");}
```

Program and Output:


NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI
DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 4

Analyze the performance of OpenMP real Applications

Performance Metrics

The performance of OpenMP programs for the problem size *n* and *p* processors can be analyzed in terms of speedup, efficiency and cost.

Speedup
It measures the relative performance of two systems processing the same problem. It is the ratio of the sequential execution time to the parallel execution time.

$$\psi(n,p) = \frac{sequential\ execution\ time}{parallel\ execution\ time}$$

Ideally, $S(p) = p$, is called perfect speedup, although in practice this is rarely achieved. There are some situations where superlinear speedup is achieved due to memory hierarchy effects.

Efficiency
It is defined as the ratio of speedup to the number of processors. It measures the fraction of time for which a processor is usefully utilized.

$$\varepsilon(n,p) = \frac{sequential\ execution\ time}{processors\ used * parallel\ execution\ time}$$

$$\varepsilon(n,p) = \frac{\psi(n,p)}{processors\ used}$$

Cost
The cost of solving a problem on a parallel system is defined as the product of run time and the number of processors. A cost-optimal parallel system solves a problem with a cost proportional to the execution time of the fastest known sequential algorithm on a single processor.

$$c(n,p) = processors\ used * parallel\ execution\ time$$

Example-1: Matrix Matrix Product

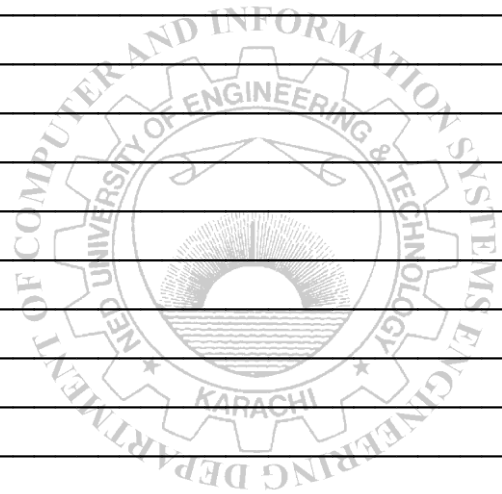
```
//////////////Serial Version//////////////////
void MatMul(double *A, double *B, double *C, int n)
{     int i, j, k;
      for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                  double dot  = 0;
                  for (k = 0; k < n; k++) {
                        dot += A[i*n+k]*B[k*n+j];
                  }
                  C[i*n+j ] = dot;
            }
      }
}
```

```
////////////////Parallel Version/////////////////
void MatMul_omp(double *A, double *B, double *C, int n)
{     #pragma omp parallel
      {     int i, j, k;
            #pragma omp for
            for (i = 0; i < n; i++) {
                  for (j = 0; j < n; j++) {
                        double dot  = 0;
                        for (k = 0; k < n; k++) {
                              dot += A[i*n+k]*B[k*n+j];
                        }
                        C[i*n+j ] = dot;
                  }
            }
      }
}
main() {
int i, n;
double *A, *B, *C, dtime;
n=2;
A = (double*)malloc(sizeof(double)*n*n);
B = (double*)malloc(sizeof(double)*n*n);
C = (double*)malloc(sizeof(double)*n*n);
for(i=0; i<n*n; i++)
{ A[i] = rand()/RAND_MAX; B[i] = rand()/RAND_MAX;}
dtime = omp_get_wtime();
MatMul(A,B,C, n);
dtime = omp_get_wtime() - dtime;
printf("%f\n", dtime);
dtime = omp_get_wtime();
MatMul_omp(A,B,C, n);
dtime = omp_get_wtime() - dtime;
printf("%f\n", dtime);
}
```


EXERCISE:

1. Code the example-1 program for n=2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Execute it with 1, 2, 4, 8, 16, 32 threads. Show the comparison of speedup, efficiency, and cost. Paste the printout of source code and its output.

Program and Output:



2. Consider the following serial code for vector sum of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its OpenMP equivalent. Execute it with 1, 2, 4, 8, 16, 32 threads. Show the comparison of speedup, efficiency, and cost. Paste the printout of source code and its output.

```
for (int i=0; i<N; ++i)
{
c[i] = a[i] + b[i];
}
```

Program and Output:

3. Consider the following serial code for matrix vector product of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its OpenMP equivalent. Execute it with 1, 2, 4, 8, 16, 32 threads. Show the comparison of speedup, efficiency, and cost. Paste the printout of source code and its output.

```
for (i=0; i < N; i++)
{
    for (j=0; j < N; j++)
        C[i] += (A[i][j] * B[i]);
}
```

Program and Output:

4. Consider the following serial code for factorial of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its OpenMP equivalent. Execute it with 1, 2, 4, 8, 16, 32 threads. Show the comparison of speedup, efficiency, and cost. Paste the printout of source code and its output.

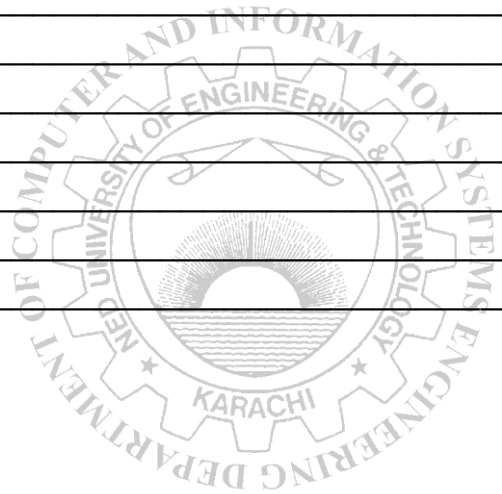
```
for (i=1; i <= N; i++)
{
factorial *= i;
}
```

Program and Output:

5. Consider the following serial code for matrix transpose of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its OpenMP equivalent. Execute it with 1, 2, 4, 8, 16, 32 threads. Show the comparison of speedup, efficiency, and cost. Paste the printout of source code and its output.

```
for(i=0; i<N; i++) {
      for(j=0; j<N; j++) {
            B[j][i] = A[i][j];
      }
}
```

Program and Output:


NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI
DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 5

Explore the SIMD Vectorization

Single Instruction Multiple Data (SIMD)

In computing, a vector processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 1990s, notably the various Cray platforms. The rapid fall in the price-to-performance ratio of conventional microprocessor designs led to the vector supercomputer's demise in the later 1990s. As of 2015 most commodity CPUs implement architectures that feature instructions for a form of vector processing on multiple (vectorized) data sets, typically known as SIMD (Single Instruction, Multiple Data). Common examples include Intel x86's MMX, SSE and AVX instructions, Sparc's VIS extension, PowerPC's AltiVec and MIPS' MSA. Vector processing techniques also operate in video-game console hardware and in graphics accelerators.

Vector Processing

In the vectorised loop, operations are performed using the vector processing unit (VPU). A vector processing unit is similar to an arithmetic unit, in that it has registers on which numbers are loaded, and arithmetic operations are performed in response to instructions. The difference is that the registers on a vector unit are much larger, e.g. 128 bits (16 bytes), 256 bits (32 bytes) or 512 bits (64 bytes). Rather than using this increased size to support higher floating point precision, this larger size is used to pack multiple floating point numbers together, so that multiple arithmetic operations can be performed in response to a single instruction. Vector processing units are built into most modern processors, and typically they come with a fixed size.

Consider a standard scalar loop;

```
for (int i=0; i<size; ++i)
{
    c[i] = a[i] + b[i];
}
```

while the following loop is a vectorised loop,

```
#pragma omp simd
for (int i=0; i<size; ++i)
{
    c[i] = a[i] + b[i];
}
```

The vectorised loop is about four times faster on computer than the standard scalar loop because, in the scalar loop each iteration is performed serially one after another. This fits in with the traditional view that a single core in a computer processor can only perform one arithmetic calculation at a time. However, this view is not strictly correct. A single compute core can actually perform many arithmetic calculations simultaneously. The compute core achieves this by batching the calculations together into groups (called vectors), and performing the entire group of calculations at once using a vector arithmetic instruction.

On X86-64 processors, the vector processing unit is also called the SIMD unit. SIMD stands for “single instruction, multiple data”, referring to the fact that a single vector instruction results in arithmetic

operations being performed simultaneously on multiple numbers (data). The capabilities of the SIMD unit has evolved with each new generation of processors, and has gone through several standards;

- **SSE**:- This stands for "Streaming SIMD Extensions", and became available with the Pentium 3 processor in 1999. The vector registers are 128 bits in size. This means that an SSE-capable processor can perform four 32 bit float operations for every vector instruction, or two 64 bit double operations.
- **SSE2**:- This was a refinement of SSE, available since the Pentium 4 in 2001. The vector registers are 128 bits in size. All X86-64 processors support SSE2.
- SSE3/SSSE3/SSE4:- These are subtle extensions to SSE2 first made available in processors released between 2004-2006, which added more esoteric arithmetic operations.
- **AVX**:- This stands for Advanced Vector eXtensions, and first became available on Intel processors from early 2011. This was a major upgrade over SSE2, as it doubles the size of the vector register from 128 bits to 256 bits. All AVX-capable processors fully support all versions of SSE. AVX-capable processors can perform eight 32 bit float operations for every vector instruction, or four 64 bit double operations.
- **AVX2**:- This is a subtle extension of AVX.
- **AVX-512**:- This is a major upgrade over AVX as it doubles the size of the vector register from 256 bits to 512 bits. This became available in Xeon Phi in 2016, and will become available in new Xeon processors from 2017. AVX-512-capable processors can perform sixteen 32 bit float operations for every vector instruction, or eight 64 bit double operations.

Methods of Vectorization

There are numerous ways to vectorize the code. Some of them are listed as follows:

1. The easiest way is to use numerical libraries, such as BLAS or Intel's math kernel library (mkl), as these are already fully vectorised.
2. Use the compiler to perform vectorisation of code. Compilers can automatically vectorise simple code. However, like auto-parallelisation, auto-vectorisation has limits, and the compiler can't do it all for you, especially if the code is very object-orientated.
3. Use OpenMP 4.0 SIMD instructions to advise the compiler how to vectorise the code.
4. Use vector intrinsics to work directly with vectors in C++ in the same way that you work with floats or doubles. It is a more direct, but more difficult, way of vectorising the code. Working directly with intrinsics means that the responsibility for vectorising the code is upon programmer. The advantage is that programmer have complete control of what is vectorised and how. This makes it easier to achieve performance portability, as the code should run at a similar speed regardless of the compiler used. However, a major disadvantage is that the intrinsic data types and functions are specific for a particular processor and vectorisation architecture. This means that the gain of performance portability comes with a loss of code portability.

OpenMP SIMD

OpenMP provides a set of compiler directives that are used to provide extra information to a compiler to allow it to automatically vectorise code (typically loops). These are built into the compiler and accessed by using pragmas (via #pragma). OpenMP 4.0 introduced omp simd, accessed via #pragma omp simd as a standard set of hints that can be given to a compiler to encourage it to vectorise code. The addition of #pragma omp simd above a loop is an OpenMP SIMD directive that tells the compiler that it should consider vectorising that loop.

Format:

```
#pragma omp simd [clauses]
```

Key Points:

• Executes iterations of following loop in SIMD chunks.

• Loop is not divided across threads.
• SIMD chunk is set of iterations executed concurrently by SIMD lanes.

NOTE:- This lab will assume that the compilation is performed using the g++ command, via gcc version 5 or above, or clang version 7.3 or above. Moreover, in this lab Linux will be used as an operating system.

Examples

Example # 01:-

```
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    const int size = 512; clock_t start;
    double duration, vector_duration;
    float a[size], b[size];
    for (int i=0; i<size; ++i)
    {
        a[i] = 1.0*(i+1);
        b[i] = 2.5*(i+1);
    }
start = clock();
float total;

    for (int j=0; j<100000; ++j)
    {
        total = 0;

        for (int i=0; i<size; ++i)
        {
            total += a[i] + b[i];
        }
    }
duration = (clock() - start ) / (double) CLOCKS_PER_SEC;
start = clock();
float vec_total;
for (int j=0; j<100000; ++j)
    { vec_total = 0;
        #pragma omp simd
        for (int i=0; i<size; ++i)
        {
            vec_total += a[i] + b[i];
        }
    }
vector_duration = ( clock() - start) / (double) CLOCKS_PER_SEC;
cout << "The standard loop took " << duration << " seconds to complete.
Total is " << total << std::endl;
cout << "The vectorised loop took " << vector_duration << " seconds to
complete. Total is " << vec_total << std::endl;
return 0;
}
```

EXERCISE:

1. Find out the speed, number of cores and size of vector of the processor you are using now (e.g. by using the lscpu command on Linux, and multiplying the number of sockets by the number of cores per socket, and then looking at the file /proc/cpuinfo and observe if sse, sse2, avx or avx2 etc. are included in the set of supported processor flags).

2. Execute Example#01 using the following commands. Note the execution time taken in executing both the standard and vectorized loops.

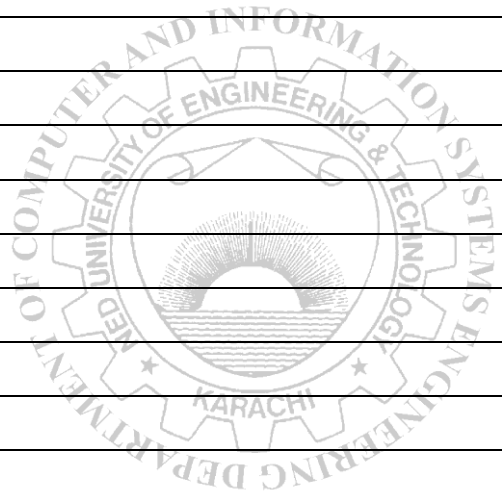
```
$ g++ -O2 --std=c++14 -fopenmp-simd eg1.cpp -Iinclude -o eg1
$ ./eg1
```

Try recompiling eg1.cpp without -fopenmp-simd compiler flag. Note the difference. How fast do the two loops run now? Does this mean that #pragma omp simd has been ignored? Also, paste the print out of source code and output.

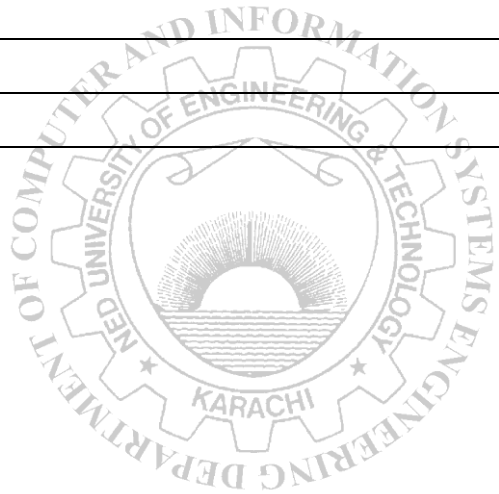
```
$ g++ -O2 --std=c++14 -Iinclude eg1.cpp -o eg1
```

3. Edit eg1.cpp and change it so that it compares a standard multiplication with a vectorised multiplication (i.e. change c[i] = a[i] + b[i]; to c[i] = a[i] * b[i]). Is the vector speed up for multiplication similar to that for addition?

4. Edit eg1.cpp to compare standard division versus vectorised division (c[i] = a[i] / b[i]). How does the vector speed-up of division compare to addition or multiplication?



5. Edit eg1.cpp to change the size of the arrays. Does changing the size affect the speedup of the vectorised loop versus the unvectorised loop? Try with the sizes 1, 2, 4, 128, 256 and 4096.


COMPUTER AND INFORMATION SYSTEMS ENGINEERING DEPARTMENT OF
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 6

Explore the advanced features of SIMD Vectorization

omp simd Features

There are several options that can extend the capability of omp simd.

1. Reduction:

#pragma omp simd reduction(+:var)

Reduction is when you accumulate a value during a loop. For example in the following loop the product of a[i] and b[i] is accumulated via a sum.

```
float total = 0;
for (int i=0; i<n; ++i)
{
    total += a[i]*b[i];
}
```

These reduction loops can be vectorized using pragma omp simd reduction(+:var) where var is the variable which is being accumulated. For example, the above loop would be vectorised as follows:

```
float total = 0;
#pragma omp simd reduction(+:total)
for (int i=0; i<n; ++i)
{
    total += a[i]*b[i];
}
```

2. Vectorised Functions:

#pragma omp declare simd

Vectorising a loop that contains function calls can be challenging. For example, consider this loop that uses a square function to calculate the square of each element of an array;

```
float square( float x )
{
    return x * x;
}
#pragma omp simd
for (int i=0; i<16; ++i)
{
    c[i] = square(a[i]);
}
```

The square function only accepts a single non-vector (scalar) argument. To vectorise the loop, a version of square is provided that accepts a vector argument. The compiler can create a vector version of a function using #pragma omp declare simd. For example;

```
#pragma omp declare simd
float square( float x )
{
    return x * x;
}
```

This would tell the compiler to create both scalar float and vector float versions of the square function. The vector float function can then be called from within a vectorised simd loop, i.e.

```
#pragma omp simd
for (int i=0; i<16; ++i)
{
    c[i] = square(a[i]);
```

```
}
```

3. Nested loops:

#pragma omp simd collapse(n)

The compiler can vectorise nested loops by using the collapse(n) option. This tells the compiler to try to vectorise the next n loops. For example, let's consider vectorising the double-loop needed to divide matrix a by matrix b;

```
for (int i=0; i<10; ++i)
{
    for (int j=0; j<10; ++j)
    {
        c[i][j] = a[i][j] / b[i][j];
    }
}
```

The outer loop could be vectorized by adding #pragma omp simd to that loop;

```
#pragma omp simd
for (int i=0; i<10; ++i)
{
    for (int j=0; j<10; ++j)
    {
        c[i][j] = a[i][j] / b[i][j];
    }
}
```

The inner loop could be vectorised by putting #pragma omd simd on that loop;

```
for (int i=0; i<10; ++i)
{
    #pragma omp simd
    for (int j=0; j<10; ++j)
    {
        c[i][j] = a[i][j] / b[i][j];
    }
}
```

To vectorise both loops together, we need to tell the compiler to collapse them together by using the collapse option;

```
#pragma omp simd collapse(2)
for (int i=0; i<10; ++i)
{
    for (int j=0; j<10; ++j)
    {
        c[i][j] = a[i][j] / b[i][j];
    }
}
```

The collapse option tells the compiler to collapse together the following two loops. The compiler does this by internally transforming these nested loops into a single collapsed loop. The loop of ten iterations of i, each with their own ten iterations of j are collapsed into a single loop of 100 iterations of i,j. This single loop is then vectorised.

Examples

Example # 2a:-

```
float red_total;
    for (int j=0; j<100000; ++j)
    {red_total = 0;
        #pragma omp simd reduction(+:red_total)
        for (int i=0; i<size; ++i)
```

```
        {
            red_total += a[i] + b[i];
        }
    }
```

Example # 2b:-

```
float simple_function(float a, float b)
{
    float x = a * a;
    float y = b * b;
    return (x-y) / (x+y);
}
#pragma omp declare simd
float vector_function(float a, float b)
{
    float x = a * a;
    float y = b * b;
    return (x-y) / (x+y);
}
int main(int argc, char **argv)
{
     for (int j=0; j<100000; ++j)
    {
        #pragma omp simd
        for (int i=0; i<size; ++i)
        {
            c[i] = simple_function(a[i], b[i]);
        }
    }
    for (int j=0; j<100000; ++j)
    {
        #pragma omp simd
        for (int i=0; i<size; ++i)
        {
            c[i] = vector_function(a[i], b[i]);
        }
    }
}
```

Example # 2c:-

```
    for (int k=0; k<10000; ++k)
    {
        #pragma omp simd
        for (int i=0; i<size; ++i)
        {
            for (int j=0; j<size; ++j)
            {
                c[i][j] = a[i][j] / b[i][j];
            }
        }
    }
for (int k=0; k<10000; ++k)
```

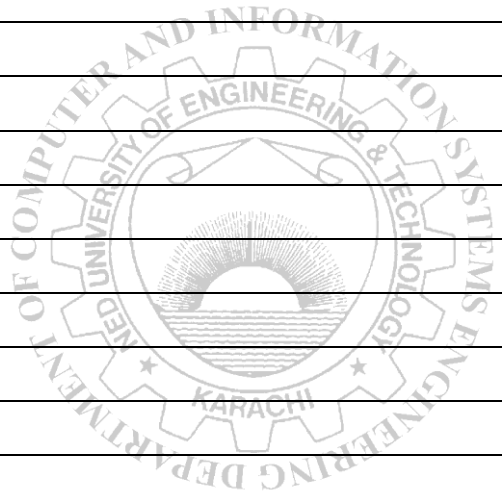
```
    {
        for (int i=0; i<size; ++i)
        {
            #pragma omp simd
            for (int j=0; j<size; ++j)
            {
                c[i][j] = a[i][j] / b[i][j];
            }
        }
    }
for (int k=0; k<10000; ++k)
    {
        #pragma omp simd collapse(2)
        for (int i=0; i<size; ++i)
        {
            for (int j=0; j<size; ++j)
            {
                c[i][j] = a[i][j] / b[i][j];
            }
        }
    }
```

EXERCISE:

1. Edit eg1.cpp to add reduction clause shown in example2a. Save the file with name eg2a.cpp. Compile and run this code for different values of size. How does the speed-up from using vectorisation and vectorised reduction change? Also, paste the print out of source code and output.

2. Edit eg1.cpp to add vectorised functions shown in example2b. Save the file with name eg2b.cpp. Compile and run this code. What is the speed-up from calling the vector function compared to the scalar function? Note that any speed-up may be masked by the compiler automatically inlining the function call. Try recompiling the function telling the compiler not to inline, using the following command. What is the speed-up of the vector function versus the scalar function? What is the difference in speed between the inlined and non-inlined program? Also, paste the print out of source code and output.

```
$ g++ -O2 --std=c++14 -fopenmp-simd -fno-inline -Iinclude function.cpp -o function
```



3. Edit eg1.cpp to add nested loops mentioned in example2c. Compile and run this code. This code will compare the unvectorised double-loop against the double-loop where only the outer loop is vectorised,

against the double loop where only the inner loop is vectorised, and where collapse(2) is used to vectorise both loops together. Run this code for different values of size to compare the speed-up (if any) of vectorising the double loop in these different ways. Do you see any speed-up from using collapse(2)? Which method of vectorising a double loop reliably performs best? Also, paste the print out of source code and output.

Lab Session 7

Acquire basic MPI (Message Passing Interface) Principles

MPI - Message Passing Interface

The Message Passing Interface or MPI is a standard for message passing that has been developed by a consortium consisting of representatives from research laboratories, universities, and industry. The first version MPI-l was standardized in 1994, and the second version MPI-2 was developed in 1997. MPI is an explicit message passing paradigm where tasks communicate with each other by sending messages.

The two main objectives of MPI are portability and high performance. The MPI environment consists of an MPI library that provides a rich set of functions numbering in the hundreds. MPI defines the concept of communicators which combine message context and task group to provide message security. Intra-communicators allow safe message passing within a group of tasks, and intercommunicates allow safe message passing between two groups of tasks. MPI provides many different flavors of both blocking and non-blocking point to point communication primitives, and has support for structured buffers and derived data types. It also provides many different types of collective communication routines for communication, between tasks belonging to a group. Other functions include those for application-oriented task topologies, profiling, and environmental query and control functions. MPI-2 also adds dynamic spawning of MPI tasks to this impressive list of functions.

Key Points:

- MPI is a library, not a language.
- MPI is a specification , not a particular implementation
- MPI addresses the message passing model.

Implementation of MPI: MPICH

MPICH is one of the complete implementation of the MPI specification, designed to be both portable and efficient. The ``CH'' in MPICH stands for ``Chameleon,'' symbol of adaptability to one's environment and thus of portability. Chameleons are fast, and from the beginning a secondary goal was to give up as little efficiency as possible for the portability.
MPICH is a unified source distribution, supporting most flavors of Unix and recent versions of Windows. In additional, binary distributions are available for Windows platforms.

Structure of MPI Program:

```
#include <mpi.h>
int main(int argc, char ** argv)
      //Serial Code
      {
          MPI_Init(&argc,&argv);
               //Parallel Code
          MPI_Finalize();

      //Serial Code
      }
```

A simple MPI program contains a main program in which parallel code of program is placed between MPI_Init and MPI_Finalize.

- **MPI_Init**
  It is used initializes the parallel code segment. Always use to declare the start of the parallel code segment.

```
int MPI_Init( int* argc ptr /* in/out */ ,char** argv ptr[ ] /* in/out */)
```

or simply

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

- **MPI Finalize**
  It is used to declare the end of the parallel code segment. It is important to note that it takes no arguments.

```
int MPI Finalize(void)
```

or simply

```
MPI_Finalize()
```

Key Points:

- Must include mpi.h by introducing its header #include<mpi.h>. This provides us with the function declarations for all MPI functions.
- A program must have a beginning and an ending. The beginning is in the form of an *MPI_Init()* call, which indicates to the operating system that this is an MPI program and allows the OS to do any necessary initialization. The ending is in the form of an *MPI_Finalize()* call, which indicates to the OS that "clean-up" with respect to MPI can commence.
- If the program is embarrassingly parallel, then the operations done between the MPI initialization and finalization involve no communication.

Predefined Variable Types in MPI

MPI DATA TYPE	***C DATA TYPE***
MPI_CHAR	Signed Char
MPI_SHORT	Singed Short Int
(Cont.)	
MPI_INT	Signed Int
MPI_LONG	Singed Long Int
MPI_UNSIGNED_CHAR	Unsigned Char
MPI_UNSIGNED_SHORT	Unsigned Short Int
MPI_UNSIGNED	Unsigned Int
MPI_UNSIGNED_LONG	Unsigned Long Int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long Double
MPI_BYTE	----------
MPI_PACKED	----------

Example-1: First MPI Program:

```
#include <iostream.h>
#include <mpi.h>
int main(int argc, char ** argv)
      {
            MPI_Init(&argc,&argv);
            cout << "Hello World!" << endl;
            MPI_Finalize();
      }
```

On compile and running of the above program, a collection of "Hello World!" messages will be printed to your screen equal to the number of processes on which you ran the program despite there is only one print statement.

StarHPC Virtual Machine
StarHPC provides a virtual machine image (Linux OS) configured for parallel programming in both OpenMP and OpenMPI technologies. StarHPC can be used with Virtual Box, VMware player, etc. to quickly get started with MPI and OpenMP programming.

The StarHPC virtual machine comes with the following as shown in Figure 7.1.

- OpenMPI and OpenMP compilers, headers, etc.
- Eclipse Parallel Tools Platform 1.1
- Photran Development Tools (Fortran in Eclipse)

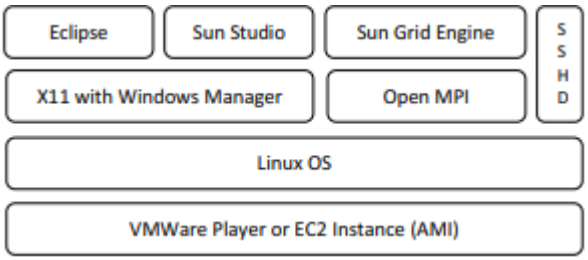


Figure 7.1. Software packaged in StarHPC VM

Compilation and Execution of a Program

For Compilation on Linux terminal, ***mpicc - o*** *{executable file name} {file name with c extension}*
For Execution on Linux terminal, ***mpirun –np*** *{ number of process} {executable file name}*

Determining the Number of Processors and their IDs

There are two important commands very commonly used in MPI:

- **MPI Comm rank:** It provides you with your process identification or rank
  (Which is an integer ranging from 0 to P − 1, where P is the number of processes on which are running),

```
int MPI_Comm_rank(MPI Comm comm /* in */,int* result /* out */)
```

or simply

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank)
```

- **MPI Comm size:** It provides you with the total number of processes that have been allocated.

```
int MPI_Comm_size( MPI Comm comm /* in */,int*  size /* out */)
```

or simply

```
MPI_Comm_size(MPI_COMM_WORLD,&mysize)
```

The argument ***comm*** is called the communicator, and it essentially is a designation for a collection of processes which can communicate with each other. MPI has functionality to allow you to specify varies communicators (differing collections of processes); however, generally ***MPI_COMM_WORLD,*** which is predefined within MPI and consists of all the processes initiated when a parallel program, is used.

Example-2:

```
#include <iostream.h>
#include <mpi.h>
int main(int argc, char ** argv)
      {
      int mynode, totalnodes;
            MPI_Init(&argc,&argv);
                  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
                  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
                  cout << "Hello world from process " << mynode;
                  cout << " of " << totalnodes << endl;
            MPI_Finalize();
      }
```

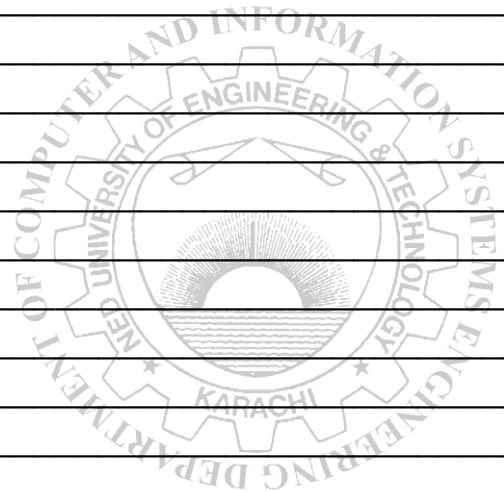
When run with four processes, the screen output may look like:
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 2 of 4
Hello world from process 1 of 4

Key Point:
The output to the screen may not be ordered correctly since all processes are trying to write to the screen at the same time, and the operating system has to decide on an ordering. However, the thing to notice is that each process called out with its process identification number and the total number of MPI processes of which it was a part.

Exercises:

1. Code the above example (1 to 2) programs and paste the printout of source codes and their outputs.

Programs and Outputs:



2. Write another program that has the same objective but it prints the name of node also using MPI_Get_processor_name function. Paste the source code and output of program.

Program and Output:

3. Write a program that prints "*I am even*" for nodes whom rank is divisible by two and print "I am odd" for the other odd ranking nodes. Paste the source code and output of program.

Program and Output:

Lab Session 8

Explore the communication between MPI processes

MPI Processes Communication

It is important to observe that when a program running with MPI, all processes use the same compiled binary, and hence all processes are running the exact same code. What in an MPI distinguishes a parallel program running on P processors from the serial version of the code running on P processors? Two things distinguish the parallel program:

- Each process uses its process rank to determine what part of the algorithm instructions are meant for it.
- Processes communicate with each other in order to accomplish the final task.

Even though each process receives an identical copy of the instructions to be executed, this does not imply that all processes will execute the same instructions. Because each process is able to obtain its process rank (using MPI_Comm_rank).It can determine which part of the code it is supposed to run. This is accomplished through the use of IF statements. Code that is meant to be run by one particular process should be enclosed within an IF statement, which verifies the process identification number of the process. If the code is not placed within IF statements specific to a particular id, then the code will be executed by all processes.

The second point, communicating between processes; MPI communication can be summed up in the concept of sending and receiving messages. Sending and receiving is done with the following two functions: MPI Send and MPI Recv.

- **MPI_Send**

```
int MPI_Send( void* message /* in */, int count /* in */, MPI Datatype
 datatype /* in */, int dest /* in */, int tag /* in */, MPI Comm comm
                          /* in */ )
```

- **MPI_Recv**

```
int MPI_Recv( void* message /* out */, int count /* in */, MPI Datatype
datatype /* in */, int source /* in */, int tag /* in */, MPI Comm comm
                 /* in */, MPI Status* status /* out */)
```

Argument Lists

- **message** - starting address of the send/recv buffer.
- **count** - number of elements in the send/recv buffer.
- **datatype** - data type of the elements in the send buffer.
- **source** - process rank to send the data.
- **dest** - process rank to receive the data.
- **tag** - message tag.
- **comm** - communicator.
- **status** - status object.

Example-1:

The following program demonstrate the use of send/receive function in which sender is initialized as node two (2) whereas receiver is assigned as node four (4). The following program requires that it should be accommodated on five (5) nodes otherwise the sender and receiver should be initialized to suitable ranks.

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

int main(int argc, char ** argv
{
      int mynode, totalnodes;
      int datasize; // number of data units to be sent/recv
      int sender=2; // process number of the sending process
      int receiver=4; // process number of the receiving process
      int tag; // integer message tag
      MPI_Status status; // variable to contain status information
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
      MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
      // Determine datasize
      double * databuffer = new double[datasize];
      // Fill in sender, receiver, tag on sender/receiver processes,
      // and fill in databuffer on the sender process.
      if(mynode==sender)
                  MPI_Send(databuffer,datasize,MPI_DOUBLE,receiver,
      tag,MPI_COMM_WORLD);
      if(mynode==receiver)
         MPI_Recv(databuffer,datasize,MPI_DOUBLE,sender,tag,
      MPI_COMM_WORLD,&status);
      // Send/Recv complete
     MPI_Finalize();
}
```

Key Points:

- In general, the *message* array for both the sender and receiver should be of the same type and both of same size at least *datasize*.
- In most cases the *sendtype* and *recvtype* are identical.
- The tag can be any integer between 0-32767.
- *MPI Recv* may use for the tag the wildcard *MPI ANY TAG*. This allows an MPI Recv to receive from a send using any tag.
- *MPI Send* cannot use the wildcard *MPI ANY TAG*. A special tag must be specified.
- *MPI Recv* may use for the source the wildcard *MPI ANY SOURCE*. This allows an MPI Recv to receive from a send from any source.
- MPI Send must specify the process rank of the destination. No wildcard exists.

Example-2: Calculate the sum of given numbers in parallel

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<mpi.h>

int main(int argc, char ** argv)
{
      int mynode, totalnodes;
```

```
      int sum,startval,endval,accum;
      MPI_Status status;
     MPI_Init(argc,argv);
      MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
      MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
            sum = 0;
            startval = 1000*mynode/totalnodes+1;
            endval = 1000*(mynode+1)/totalnodes;
            for(int i=startval;i<=endval;i=i+1)
            sum = sum + i;
            if(mynode!=0)
            MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
            else
            for(int j=1;j<totalnodes;j=j+1)
            {
             MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
            sum = sum + accum;
            }
            if(mynode == 0)
             cout << "The sum from 1 to 1000 is: " <<  sum <<   endl;
            MPI_Finalize();
}
```

The second portion of this lab will focus on more information about sending and receiving in MPI like sending of arrays and simultaneous send and receive

Key Points

- Whenever you send and receive data, MPI assumes that you have provided non overlapping positions in memory. As discussed in the previous lab session, *MPI_COMM_WORLD* is referred to as a **communicator**. In general, a communicator is a collection of processes that can send messages to each other. *MPI_COMM_WORLD* is pre-defined in all implementations of MPI, and it consists of all MPI processes running after the initial execution of the program.

- In the send/receive, we are required to use a *tag*. The tag variable is used to distinguish upon receipt between two messages sent by the same process.

- The order of sending does not necessarily guarantee the order of receiving. Tags are used to distinguish between messages. MPI allows the tag *MPI_ANY_TAG* which can be used by *MPI_Recv* to accept any valid tag from a sender but you *cannot* use *MPI_ANY_TAG* in the *MPI_Send* command.

- Similar to the *MPI_ ANY_ TAG* wildcard for tags, there is also an *MPI_ANY_SOURCE* wildcard that can also be used by *MPI_Recv*. By using it in an *MPI_Recv*, a process is ready to receive from any sending process. Again, you *cannot* use *MPI_ ANY_ SOURCE* in the *MPI_ Send* command. There is no wildcard for sender destinations.

- When you pass an array to *MPI_ Send/MPI_Recv*, it need not have exactly the number of items to be sent – *it must have greater than or equal to the number of items to be sent*. Suppose, for example, that you had an array of 100 items, but you only wanted to send the first ten items, you can do so by passing the array to *MPI_Send* and only stating that ten items are to be sent.

Example-3:
In the following MPI code, array on each process is created, initialize it on process 0. Once the array has been initialized on process 0, then it is sent out to each process.

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

int main(int argc, char * argv[])
{
      int i;
      int nitems = 10;
      int mynode, totalnodes;
      MPI_Status status;
      double * array;

      MPI_Init(&argc,&argv);
         MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
         MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
         array = new double[nitems];
         if(mynode == 0)
         {
         for(i=0;i<nitems;i++)
         array[i] = (double) i;
         }
         if(mynode==0)
         for(i=1;i<totalnodes;i++)
          MPI_Send(array,nitems,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
          else
          MPI_Recv(array,nitems,MPI_DOUBLE,0,1,MPI_COMM_WORLD,
          &status);
          for(i=0;i<nitems;i++)
             {
              cout << "Processor " << mynode;
              cout << ": array[" << i << "] = " << array[i] <<  endl;
             }
          MPI_Finalize();
}
```

Key Points:

- *An array is* created, on each process, using dynamic memory allocation.
- On process 0 only (i.e., mynode == 0), an array is initialized to contain the ascending index values.
- On process 0, program proceeds with (totalnodes-1) calls to *MPI Send*.
- On all other processes other than 0, MPI_*Recv is* called *to* receive the sent message.
- On each individual process, the results are printed of the sending/receiving pair.

Simultaneous Send and Receive, MPI_Sendrecv:

The subroutine MPI_Sendrecv exchanges messages with another process. A send-receive operation is useful for avoiding some kinds of unsafe interaction patterns and for implementing remote procedure calls. A message sent by a send-receive operation can be received by MPI_Recv and a send-receive operation can receive a message sent by an MPI_Send.
MPI_Sendrecv(&data_to_send, send_count, send_type, destination_ID, send_tag, &received_data, receive_count, receive_type, sender_ID, receive_tag, comm, &status)

Argument Lists

- **data_to_send**: variable of a C type that corresponds to the MPI send_type supplied below

- **send_count:** number of data elements to send (int)
- **send_type:** datatype of elements to send (one of the MPI datatype handles)
- **destination_ID**: process ID of the destination (int)
- **send_tag**: send tag (int)
- **received_data**: variable of a C type that corresponds to the MPI receive_type supplied below
- **receive_count**: number of data elements to receive (int)
- **receive_type:** datatype of elements to receive (one of the MPI datatype handles)
- **sender_ID**: process ID of the sender (int)
- **receive_tag**: receive tag (int)
- **comm**: communicator (handle)
- **status**: status object (MPI_Status)

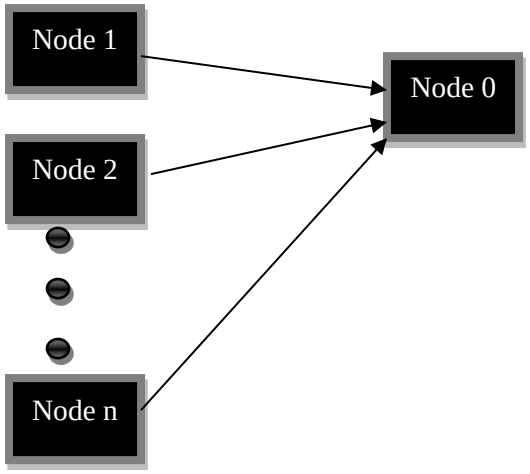
It should be noted in above stated arguments that they contain all the arguments that were declared in send and receive functions separately in the previously.

Exercise:

1. Code the example programs (1 to 3). Execute with 1, 2, and 4 processes. Paste the source codes and their outputs.

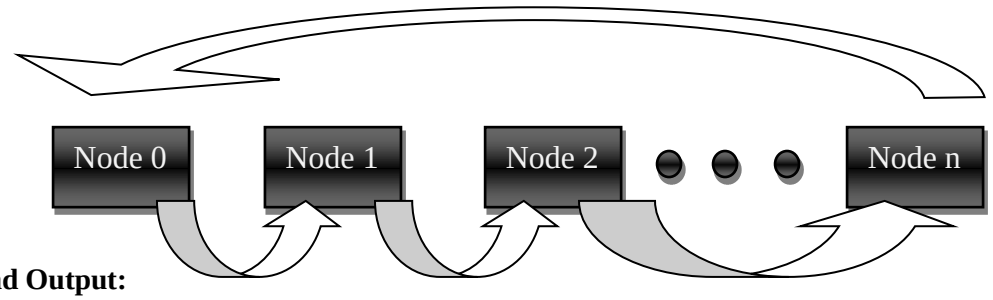
Programs and Outputs:

2. 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. Paste the printout of source code and output.



Program and Output:

3. 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. Paste the printout of source code and output.

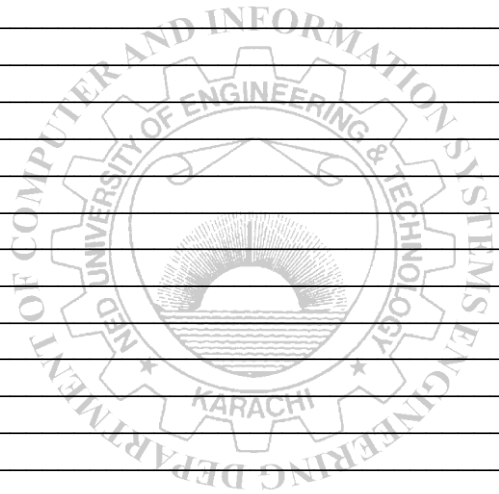


Program and Output:

4. Write a program to calculate prefix sum S_n 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.

Paste the printout of source code and output.

Program and Output:


COMPUTER AND INFORMATION SYSTEMS ENGINEERING DEPARTMENT OF
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 9

Explore the MPI collective operations

Collective Operations

MPI_Send and MPI_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operations are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission. MPI supports three classes of collective operations:

- Synchronization,
- Data Movement, and
- Collective Computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

Synchronization

The MPI_Barrier function can be used to synchronize a group of processes. To synchronize a group of processes, each one must call MPI_Barrier when it has reached a point where it can go no further until it knows that all its partners have reached the same point. Once a process has called MPI_Barrier, it will be blocked until all processes in the group have also called MPI_Barrier.

- **MPI Barrier**

```
int MPI_Barrier( MPI_Comm comm /* in */  )
```

Argument Lists

- *comm* – communicator

Example of Usage

```
int mynode, totalnodes;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
MPI_Barrier(MPI_COMM_WORLD);
// At this stage, all processes are synchronized
```

Key Point

- This command is a useful tool to help insure synchronization between processes. For example, you may want all processes to wait until one particular process has read in data from disk. Each process would call *MPI_Barrier* in the place in the program where the synchronization is required.

Collective data movement

There are several routines for performing collective data distribution tasks:

- **MPI_Bcast**, The subroutine MPI_Bcast sends a message from one process to all processes in a communicator.
- **MPI_Gather, MPI_Gatherv**, Gather data from participating processes into a single structure
- **MPI_Scatter, MPI_Scatterv,** Break a structure into portions and distribute those portions to other processes
- **MPI_Allgather, MPI_Allgatherv**, Gather data from different processes into a single structure that is then sent to all participants (Gather-to-all)
- **MPI_Alltoall, MPI_Alltoallv**, Gather data and then scatter it to all participants (All-to-all scatter/gather)

The routines with "V" suffixes move variable-sized blocks of data.

MPI_Bcast

- The subroutine MPI_Bcast sends a message from one process to all processes in a communicator.
- In a program all processes must execute a call to MPI_BCAST. There is no separate MPI call to receive a broadcast.

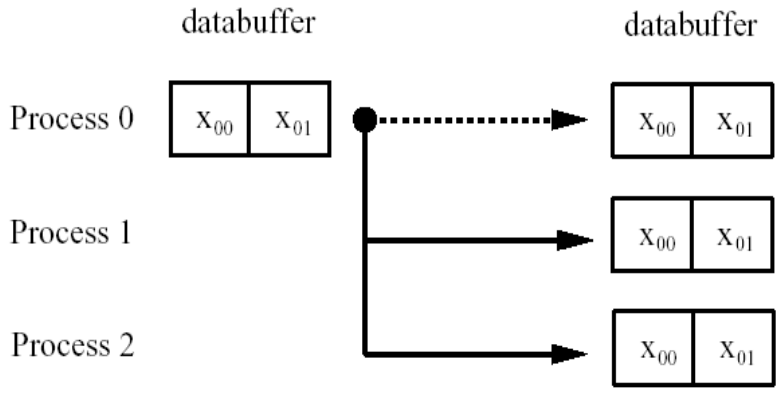


Figure 9.1. MPI Bcast schematic demonstrating a broadcast of two data objects from process zero to all other processes.

```
int MPI Bcast( void* buffer /* in/out */, int count /* in */, MPI Datatype
datatype /* in */, int root /* in */, MPI Comm comm /* in */)
```

Argument List

- **buffer** - starting address of the send buffer.
- **count** - number of elements in the send buffer.
- **datatype** - data type of the elements in the send buffer.
- **root** - rank of the process broadcasting its data.
- **comm** - communicator.

MPI_Bcast broadcasts a message from the process with rank "root" to all other processes of the group.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units to be broadcast
int root; // process which is broadcasting its data

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

```
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

// Determine datasize and root
double * databuffer = new double[datasize];
// Fill in databuffer array with data to be broadcast

MPI_Bcast(databuffer,datasize,MPI_DOUBLE,root,MPI_COMM_WORLD);

// At this point, every process has received into the
// databuffer array the data from process root
```

Key Point

- Each process will make an identical call of the *MPI Bcast* function. On the broadcasting (root) process, the *buffer* array contains the data to be broadcast. At the conclusion of the call, all processes have obtained a copy of the contents of the *buffer* array from process root.

MPI_Scatter:

MPI_Scatter is one of the most frequently used functions of MPI Programming. Break a structure into portions and distribute those portions to other processes. Suppose you are going to distribute an array elements equally to all other nodes in the cluster by decomposing the main array into its sub segments which are then distributed to the nodes for parallel computation of array segments on different cluster nodes.

```
int MPI_Scatter
      (
     void *send_data,
     int send_count,
     MPI_Datatype send_type,
     void  *receive_data,
     int receive_count,
     MPI_Datatype receive_type,
     int sending_process_ID,
     MPI_Comm comm.
     )
```

MPI_Gather

- Gather data from participating processes into a single structure
- Synopsis:

```
#include "mpi.h"
int MPI_Gather
     (
     void *sendbuf,
      int sendcnt,
      MPI_Datatype sendtype,
      void *recvbuf,
      int recvcount,
      MPI_Datatype recvtype,
      int root,
      MPI_Comm comm
      )
```

  - Input Parameters:
    - **sendbuf:** starting address of send buffer
    - **sendcount:** number of elements in send buffer

- **sendtype:** data type of send buffer elements
- **recvcount:** number of elements for any single receive (significant only at root)
- **recvtype:** data type of recv buffer elements (significant only at root)
- **root:** rank of receiving process
- **comm:** communicator

- Output Parameter:
  - recvbuf: address of receive buffer (significant only at root)

Collective Computation Routines

Collective computation is similar to collective data movement with the additional feature that data may be modified as it is moved. The following routines can be used for collective computation.

- **MPI_Reduce:** Perform a reduction operation.
- **MPI_Allreduce:** Perform a reduction leaving the result in all participating processes
- **MPI_Reduce_scatter:** Perform a reduction and then scatter the result
- **MPI_Scan:** Perform a reduction leaving partial results (computed up to the point of a process's involvement in the reduction tree traversal) in each participating process. (parallel prefix)

Collective computation built-in operations

Many of the MPI collective computation routines take both built-in and user-defined combination functions. The built-in functions are:

Table 9.1 Collective Computation Operations

Operation handle	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical Exclusive OR
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise Exclusive OR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

MPI_Reduce:

MPI_Reduce apply some operation to some operand in every participating process. For example, add an integer residing in every process together and put the result in a process specified in the MPI_Reduce argument list. The subroutine MPI_Reduce combines data from all processes in a communicator using one of several reduction operations to produce a single result that appears in a specified target process.

When processes are ready to share information with other processes as part of a data reduction, all of the participating processes execute a call to MPI_Reduce, which uses local data to calculate each process's

portion of the reduction operation and communicates the local result to other processes as necessary. Only the target_process_ID receives the final result.

```
int MPI_Reduce(
           void* operand /* in */,
           void* result /* out */,
           int count /* in */,
           MPI Datatype datatype /* in */,
           MPI Op operator /* in */,
           int root /* in */,
           MPI Comm comm /* in */
           )
```

Argument List

- **operand** - starting address of the send bu.er.
- **result** - starting address of the receive bu.er.
- **count** - number of elements in the send bu.er.
- **datatype** - data type of the elements in the send/receive bu.er.
- **operator** - reduction operation to be executed.
- **root** - rank of the root process obtaining the result.
- **comm** - communicator.

Example of Usage

The given code receives data on only the root node (rank=0) and passes null in the receive data argument of all other nodes

```
int mynode, totalnodes;
int datasize; // number of data units over which
// reduction should occur
int root; // process to which reduction will occur

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

// Determine datasize and root
double * senddata = new double[datasize];
double * recvdata = NULL;

if(mynode == root)
recvdata = new double[datasize];
// Fill in senddata on all processes

MPI_Reduce(senddata,recvdata,datasize,MPI_DOUBLE,MPI_SUM,
root,MPI_COMM_WORLD);

// At this stage, the process root contains the result of the
reduction (in this   case MPI_SUM) in the recvdata array
```

Key Points

- The recvdata array only needs to be allocated on the process of rank root (since root is the only processor receiving data). All other processes may pass NULL in the place of the recvdata argument.

- Both the senddata array and the recvdata array must be of the same data type. Both arrays should contain at least datasize elements.

MPI_Allreduce*:* Perform a reduction leaving the result in all participating processes

```
int MPI Allreduce(
          void* operand /* in */,
          void* result /* out */,
          int count /* in */,
          MPI Datatype datatype /* in */,
          MPI Op operator /* in */,
          MPI Comm comm /* in */
          )
```

Argument List

- **operand** - starting address of the send buffer.
- **result** - starting address of the receive buffer.
- **count** - number of elements in the send/receive buffer.
- **datatype** - data type of the elements in the send/receive buffer.
- **operator** - reduction operation to be executed.
- **comm** - communicator.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units over which
// reduction should occur

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

// Determine datasize and root
double * senddata = new double[datasize];
double * recvdata = new double[datasize];
// Fill in senddata on all processes

MPI_Allreduce(senddata,recvdata,datasize,MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
// At this stage, all processes contains the result  of the
reduction (in this case MPI_SUM) in the recvdata array
```

Remarks

- In this case, the *recvdata* array needs to be allocated on all processes since all processes will be receiving the result of the reduction.
- Both the *senddata* array and the *recvdata* array must be of the same data type. Both arrays should contain at least *datasize* elements.

MPI_Scan:

- MPI_Scan: Computes the scan (partial reductions) of data on a collection of processes

  int MPI_Scan (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Input Parameters
  - sendbuf: starting address of send buffer
  - count: number of elements in input buffer
  - datatype: data type of elements of input buffer
  - op: operation
  - comm: communicator

- Output Parameter:
  recvbuf: starting address of receive buffer

Exercise:

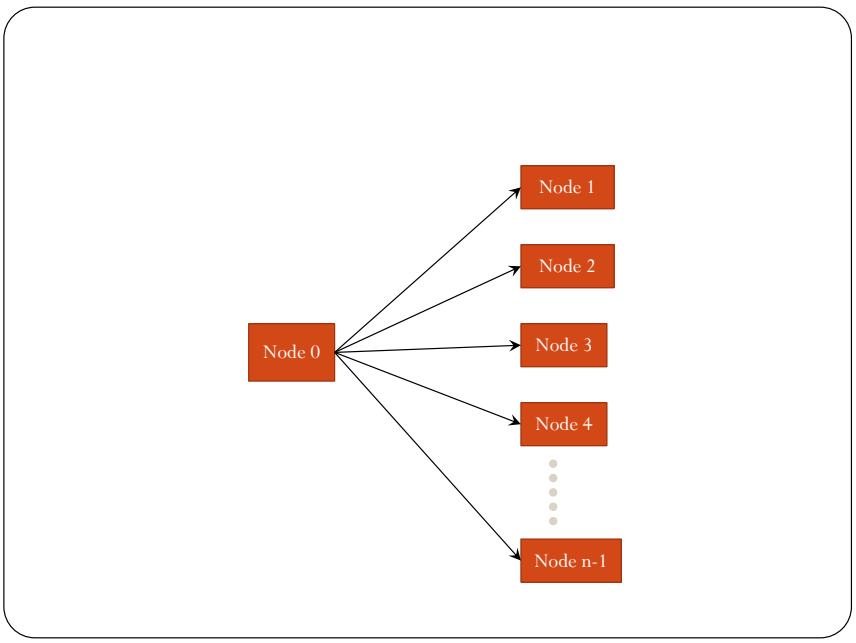
1. Write a program to spread the iterations of a loop over multiple processes using MPI_Barrier. Paste the printout of source code and output.

Program and Output:

2. Write a program to send messages between processes after making sure that all processes are alive and running using MPI_Barrier. Paste the printout of source code and output.

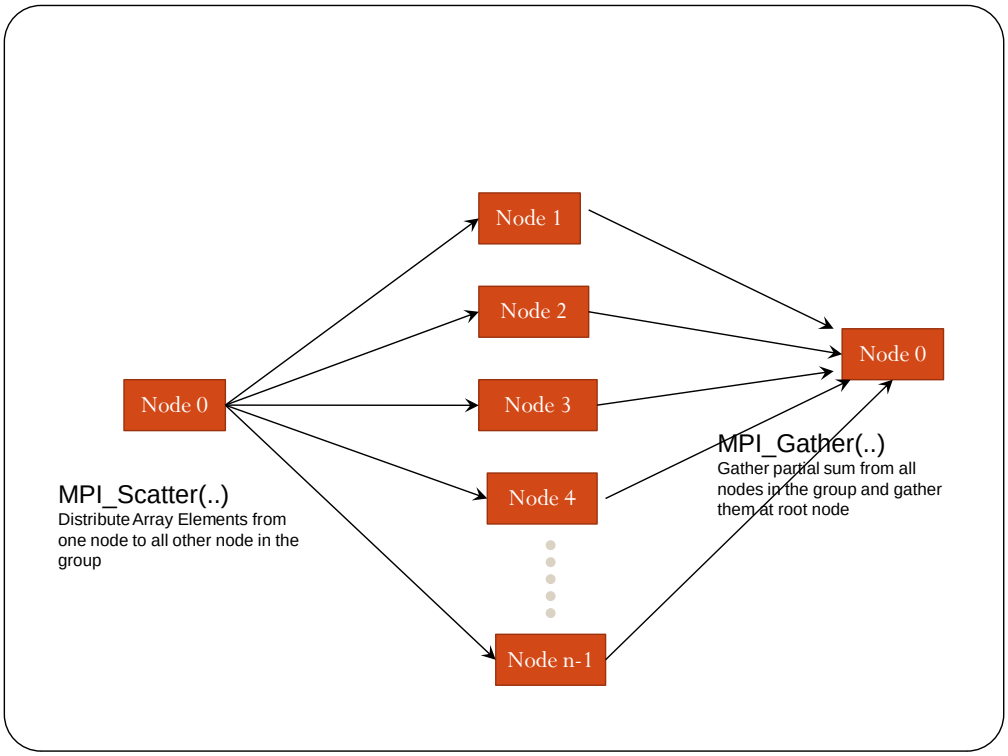
Program and Output:

3. Write a program that broadcasts a number from one process to all others by using MPI_Bcast. Paste the printout of source code and output.



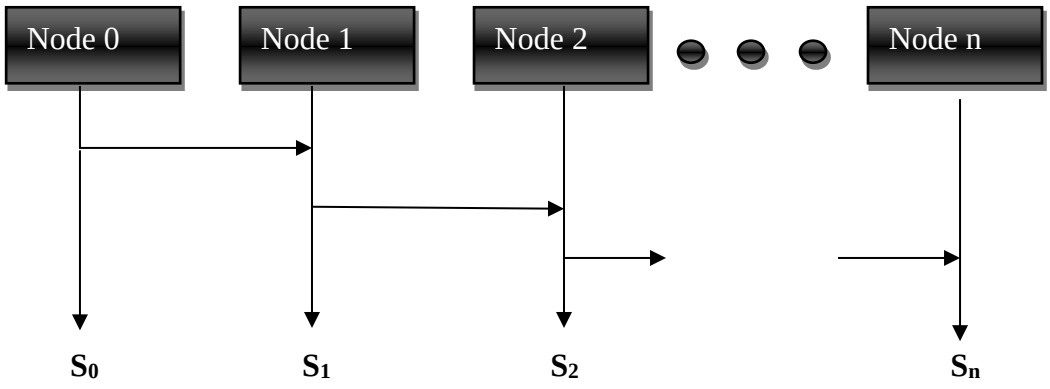
Program and Output:

4. Break up a long vector into sub-vectors of equal length. Distribute sub-vectors to processes. Let the processes to compute the partial sums. Collect the partial sums from the processes and add them at root node using collective computation operations. Paste the printout of source code and output.



Program and Output:

5. Write a program to calculate prefix sum of ‘n’ numbers on ‘n’ processes. Use MPI_Scan to address this problem. Paste the printout of source code and output. The above problem can be best stated with the help of following figure.



Program and Output:

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 10

Analyze the performance of MPI real Applications

Example-1: Matrix Matrix Product

```
#define SIZE 2              /* Size of matrices */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
double start_time, end_time;
void fill_matrix(int m[SIZE][SIZE])
{
  static int n=0;
  int i, j;
  for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      m[i][j] = n++;
}
void print_matrix(int m[SIZE][SIZE])
{
  int i, j = 0;
  for (i=0; i<SIZE; i++) {
    printf("\n\t| ");
    for (j=0; j<SIZE; j++)
      printf("%2d ", m[i][j]);
    printf("|");
  }
}
main(int argc, char *argv[])
{
  int myrank, P, from, to, i, j, k;
  int tag = 666;      /* any value will do */
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);   /* who am i */
  MPI_Comm_size(MPI_COMM_WORLD, &P); /* number of processors */
/* Just to use the simple variants of MPI_Gather and MPI_Scatter we */
/* impose that SIZE is divisible by P. By using the vector versions, */
/* (MPI_Gatherv and MPI_Scatterv) it is easy to drop this restriction.
*/

  if (SIZE%P!=0) {
    if (myrank==0) printf("Matrix size not divisible by number of
processors\n");
    MPI_Finalize();
    exit(-1);
  }
from = myrank * SIZE/P;
to = (myrank+1) * SIZE/P;
/* Process 0 fills the input matrices and broadcasts them to the rest
*/
/* (actually, only the relevant stripe of A is sent to each process) */
if (myrank==0) {
    fill_matrix(A);
```

```
    fill_matrix(B);
  }
start_time = MPI_Wtime();
MPI_Bcast (B, SIZE*SIZE, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (A, SIZE*SIZE/P, MPI_INT, A[from], SIZE*SIZE/P, MPI_INT, 0,
MPI_COMM_WORLD);
printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);
for (i=from; i<to; i++)
 for (j=0; j<SIZE; j++) {
   C[i][j]=0;
    for (k=0; k<SIZE; k++)
     C[i][j] += A[i][k]*B[k][j];
    }
MPI_Gather (C[from], SIZE*SIZE/P, MPI_INT, C, SIZE*SIZE/P, MPI_INT, 0,
MPI_COMM_WORLD);
end_time = MPI_Wtime();
 if (myrank==0) {
    printf("\n\n");
    print_matrix(A);
    printf("\n\n\t       * \n");
    print_matrix(B);
    printf("\n\n\t       = \n");
    print_matrix(C);
    printf("\n\n");
printf("Execution time is: %f\n", end_time-start_time);
  }
MPI_Finalize();
}
```

EXERCISE:

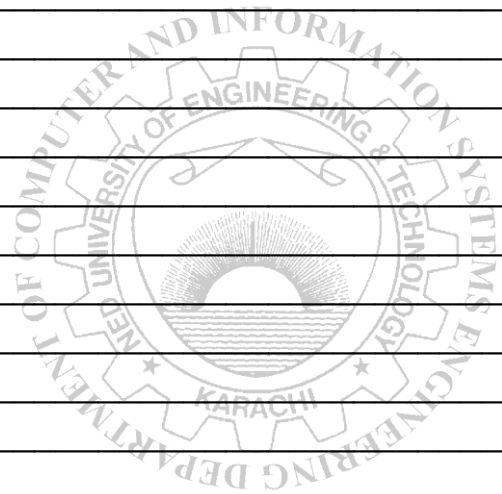
1. Code the above example program for SIZE=2,4,8,16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Execute it with 1, 2, and 4 processes. Compare the speedup, efficiency, and cost. Paste the printout of source code and output.

Program and Output:

2. Consider the following serial code for vector sum of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its MPI equivalent. Execute it with 1, 2, and 4 processes. Compare the speedup, efficiency, and cost. Paste the printout of source code and output.

```
for (int i=0; i<N; ++i)
{
c[i] = a[i] + b[i];
}
```

Program and Output:



3. Consider the following serial code for matrix vector product of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its MPI equivalent. Execute it with 1, 2, and 4 processes. Compare the speedup, efficiency, and cost. Paste the printout of source code and output.

```
for (i=0; i < N; i++)
{
    for (j=0; j < N; j++)
        C[i] += (A[i][j] * B[i]);  }
```

Program and Output:

4. Consider the following serial code for factorial of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its MPI equivalent. Execute it with 1, 2, and 4 processes. Compare the speedup, efficiency, and cost. Paste the printout of source code and output.

```
for (i=1; i <= N; i++)
{
factorial *= i;
}
```

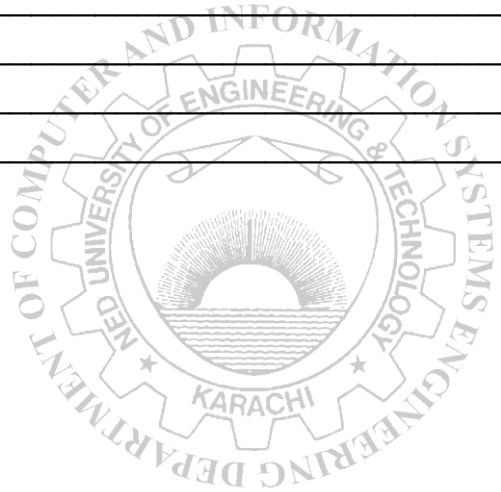
Program and Output:

5. Consider the following serial code for matrix transpose of N=16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Write its MPI equivalent. Execute it with 1, 2, and 4 processes. Compare the speedup, efficiency, and cost. Paste the printout of source code and output.

```
for(i=0; i<N; i++) {
      for(j=0; j<N; j++) {
            B[j][i] = A[i][j];
```

```
            }
        }
```

Program and Output:



DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 11

Explore Socket Programming in Linux Environment

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines.

System calls for sockets

These are the system calls involving sockets:

a) *socket*—Creates a socket

```
#include <sys/socket.h>
sockfd = socket(intprotocol_family, intsocket_type, int protocol);
```

The protocol modules are grouped into protocol families like AF_INET, AF_IPX, AF_PACKET and socket types like SOCK_STREAM or SOCK_DGRAM.

Protocol Families	
Name	Purpose
AF_UNIX, **AF_LOCAL**	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols

SOCK_STREAM provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported. SOCK_DGRAM supports datagrams (connectionless, unreliable messages of a fixed maximum length).

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner.

On success, a file descriptor for the new socket is returned. On error, -1 is returned, and *errno* is set appropriately.

b) *close*—Destroys a socket

```
#include <unistd.h>
int close(intfd);
```

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

close() returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

c) *connect*—Creates a connection between two sockets

```
#include <sys/socket.h>
int connect(intsockfd, conststructsockaddr *addr, socklen_taddrlen);
```

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd.

The sockaddr structure is the basic structure for all system calls and functions that deal with socket addresses. All pointers to other socket address structures are often cast to pointerstosockaddrbeforeuseinvarious functions and system calls:

```
#include <netinet/in.h>
structsockaddr {
      unsignedshortsa_family;       // address family, AF_xxx
      charsa_data[14];        // 14 bytes of protocol address
};
```

The Unix domain socket address structure is:

```
#include <sys/un.h>
structsockaddr_un{
      shortsun_family; //AF_UNIX
      charsun_PATH[108];      // path name
};
```

If the socket sockfd is of type SOCK_DGRAM then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to the socket that is bound to the address specified by addr. Generally, connection-based protocol sockets may successfully connect() only once; connectionless protocol sockets may use connect() multiple times to change their association.

If the connection or binding succeeds, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

d) *bind*—Labels a socket with an address

```
#include <sys/socket.h>
int bind(intsockfd, conststructsockaddr *addr,socklen_taddrlen);
```

When a socket is created with socket(), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified to by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called "assigning a name to a socket". It is normally necessary to assign a local address using bind() before a SOCK_STREAM socket may receive connections.

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

e) *listen*—Configures a socket to accept conditions

```
#include <sys/socket.h>
int listen(intsockfd, int backlog);
```

listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept(). The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

f) *accept*—Accepts a connection and creates a new socket for the connection

```
#include <sys/socket.h>

int accept(intsockfd, structsockaddr *addr, socklen_t *addrlen);
```

The accept() system call is used with connection-based socket types (SOCK_STREAM). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call. The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.

The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address. The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK. In order to be notified of incoming connections on a socket, you can use select() or poll(). A readable event will be delivered when a new connection is attempted and you may then call accept() to get a socket for that connection.

On success, this returns a nonnegative integer that is a descriptor for the accepted socket. On error, -1 is returned, and *errno* is set appropriately.

Making a server

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the *socket()* system call.
2. Bind the socket to an address using the *bind()* system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the *listen()* system call.
4. Accept a connection with the *accept()* system call. This call typically blocks until a client connects with the server.
5. Send and receive data.

Data isn't read and written directly via the server socket; instead, each time a program accepts anew connection, Linux creates a separate socket to use in transferring data over that connection.

Making a client

The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the *socket()* system call.
2. Connect the socket to the address of the server using the *connect()* system call.
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the *read()* and *write()* system calls.

Local Sockets

Sockets connecting processes on the same computer can use the local namespace represented by the synonyms AF_LOCAL and AF_UNIX. These are called local sockets or UNIX-domain sockets. Their socket addresses, specified by filenames, are used only when creating connections.

The socket's name is specified in structsockaddr_un. We must set the sun_family field to AF_LOCAL, indicating that this is a local namespace.

The sun_path field specifies the filename to use and may be, at most, 108 bytes long. Any filename can be used, but the process must have directory write permissions, which permit adding files to the directory. To connect to a socket, a process must have read permission for the file. Even though different computers may share the same file system, only processes running on the same computer can communicate with local namespace sockets.

The only permissible protocol for the local namespace is 0.Because it resides in a file system, a local socket is listed as a file.

Compilation and Execution

Use the following format for compiling the codes:

```
gcc sourcefile.c –o outputfile
```

As a result you should get two output files. Run the output files in two separate terminals, making sure you pass the socket filename as argument to the unix server at the command line.

CODING

un_server.c

```
//The pathname of the socket address is passed as an argument.
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/un.h>
#include <stdio.h>
void error(const char *)
{
perror(msg);
exit(0);
}
int main(intargc, char *argv[])
{
intsockfd, newsockfd, servlen, n;
socklen_tclilen;
structsockaddr_uncli_addr, serv_addr;
charbuf[80];
if ((sockfd = socket(AF_UNIX,SOCK_STREAM,0)) < 0)
       error("creating socket");
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, argv[1]);
servlen=strlen(serv_addr.sun_path)                                    +
sizeof(serv_addr.sun_family);
if(bind(sockfd,(structsockaddr *)&serv_addr,servlen)<0)
       error("binding socket");
listen(sockfd,5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,(structsockaddr)&cli_addr,&clilen);
if (newsockfd< 0)
       error("accepting");
n=read(newsockfd,buf,80);
printf("A connection has been established\n");
write(1,buf,n);
write(newsockfd,"I got your message\n",19);
close(newsockfd);
close(sockfd);
return 0;
}
```

un_client.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
void error(const char *)
{
perror(msg);
exit(0);
}
int main(intargc, char *argv[])
{
intsockfd, servlen,n;
structsockaddr_unserv_addr;
char buffer[82];
bzero((char *)&serv_addr,sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, argv[1]);
servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);
if ((sockfd = socket(AF_UNIX, SOCK_STREAM,0)) < 0)
       error("Creating socket");
if (connect(sockfd, (structsockaddr*)&serv_addr, servlen)< 0)
       error("Connecting");
printf("Please enter your message: ");
bzero(buffer,82);
fgets(buffer,80,stdin);
write(sockfd,buffer,strlen(buffer));
n=read(sockfd,buffer,80);
printf("The return message was\n");
write(1,buffer,n);
close(sockfd);
return 0;
}
```

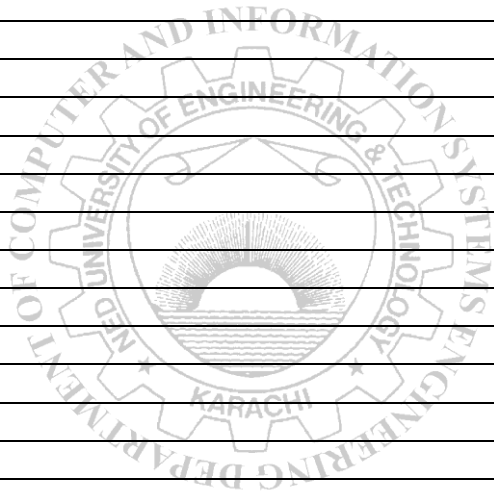
EXERCISES

a. Implement the above server and client on your linux machine and write atleast two sample inputs and outputs.

b. What does 1 represent in the line `write(1,buffer,n)`? What would 0 or 2 represent in the same place?

c. Explain the following:

i. `void perror(const char *s);`
ii. `void bzero(void *s, size_t n);`
iii. `ssize_t read(intfd, void *buf, size_t count);`
iv. `ssize_t write(intfd, const void *buf, size_t count);`
v. `socklen_t`


COMPUTER AND INFORMATION SYSTEMS ENGINEERING DEPARTMENT OF
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 12

Explore the Socket Programming in Linux over the Network

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

Here, we focus on sockets set up over a network, enabling communication between two different machines. The protocol families for such sockets are AF_INET and AF_INET6.

The socket address structure for IPv4 sockets is:

```
structsockaddr_in {
      shortsin_family;        // e.g. AF_INET, AF_INET6
      unsigned shortsin_port;       // e.g. htons(3490)
      structin_addrsin_addr;        // see structin_addr, below
      charsin_zero[8];        // zero this if you want to
};
structin_addr {
      unsigned long s_addr; // load with inet_pton()
};
```

The socket address structure for IPv6 sockets is:

```
struct sockaddr_in6 {
      u_int16_tsin6_family;         // AF_INET6
      u_int16_tsin6_port;           // port number, Network Byte Order
      u_int32_tsin6_flowinfo;             // IPv6 flow information
      struct in6_addrsin6_addr;     // IPv6 address
      u_int32_tsin6_scope_id;             // Scope ID
};
struct in6_addr {
      unsigned chars6_addr[16];     // load with inet_pton()
};
```

Pointers to both the above structures have to be type casted to sockaddr structure pointers. The sockaddr structure has already been described in the previous lab dealing with local sockets.

A function that requires mention is:

```
#include <netdb.h>
structhostent *gethostbyname(const char *name);
```

The gethostbyname() function returns a structure of type hostent for the given host name. Here name is either a hostname, or an IPv4 address in standard dot notation, or an IPv6 address in colon (and possibly dot) notation. The hostent structure is defined as follows:

```
structhostent {
      char*h_name;                  //official name of host
      char **h_aliases;                   //alias list
      inth_addrtype;                //host address type
      inth_length;                  //length of address
      char **h_addr_list;           //list of addresses
}
#define h_addrh_addr_list[0]        //for backward compatibility
```

Another function that you may use is:

```
#include <netinet/in.h>
uint16_thtons(uint16_t hostshort);
```

The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.

Compilation and execution shall be done on two separate machines.

ALGORITHMS

Algorithm A1: Setting up a simple server in the internet domain using TCP

The port number is passed as an argument at the command line.

1. If the port number has not been passed as argument by the user, report "missing port number" and exit.
2. Create a stream socket of AF_INET family with TCP protocol.
3. Initialize structure serv_addr, of type sockaddr_in, to zero.
4. Prepare the elements of serv_addr in the following manner:
   a. Assign AF_INET to sin_family.
   b. Convert the port number from ascii to integer.
   c. Use htons(port_no) to fill in sin_port.
   d. Assign INADDR_ANY to s_addr in sin_addr structure in serv_addr.
5. Bind the socket with the serv_addr structure.
6. Listen for connections to this socket.
7. Accept a connection to this socket. (A new socket will be established)
8. Initialize an array buffer of adequate size to zero.
9. Read from the new socket into the buffer array, and display its contents.
10. Write to the new socket: "I got your message".
11. Close the new socket.
12. Close the original listening socket.

Algorithm A2: Setting up a simple client in the internet domain using TCP

The server's hostname and server port number are passed as arguments at the command line.

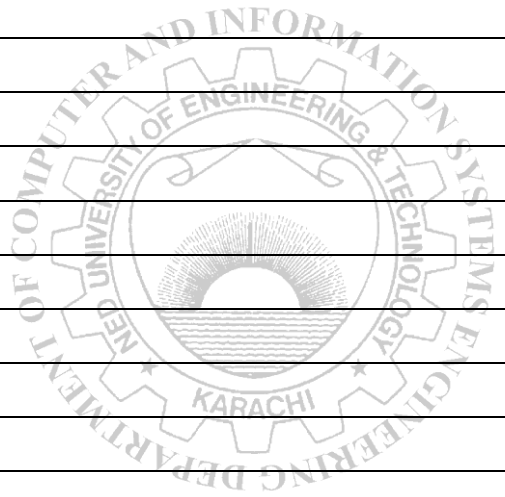
1. If the server's hostname has not been passed as argument by the user, report "missing hostname" and exit.
2. If the port number has not been passed as argument by the user, report "missing port number" and exit.
3. Create a stream socket of AF_INET family with TCP protocol.
4. Initialize structure serv_addr, of type sockaddr_in, to zero.
5. Prepare the elements of serv_addr in the following manner:
   a. Assign AF_INET to sin_family.
   b. Convert the port number from ascii to integer. Use htons(port_no) to fill in sin_port.
   c. Use gethostbyname(server_hostname) to get a hostent structure for the server.
   d. Copy the server's address from the h_addr element of the hostent structure returned to s_addr in sin_addr structure in serv_addr.
6. Connect to the server using the serv_addr structure and the socket established in step 3.
7. Initialize an array buffer of adequate size to zero.
8. Prompt the user for input.
9. Store the user's input in the buffer array.
10. Write the buffer array to the socket.
11. Reinitialize the array buffer to zero.

12. Read from the socket into the buffer array, and display its contents.
13. Close the socket.

EXERCISES

a. Implement the given server algorithm in C for your Linux machine.

b. Implement the given client algorithm in C for your Linux machine.


DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

c. Why do we assign INADDR_ANY to s_addr in sin_addr structure in serv_addr?

Lab Session 13

Explore the Java Remote Method Invocation (RMI)

Remote Method Invocations (RMI) facilitates object function calls between Java Virtual Machines (JVMs). JVMs can be located on separate computers - yet one JVM can invoke methods belonging to an object stored in another JVM. Methods can even pass objects that a foreign virtual machine has never encountered before, allowing dynamic loading of new classes as required. This is a powerful feature.

Consider the following scenario:

- Developer A writes a service that performs some useful function. He regularly updates this service, adding new features and improving existing ones.
- Developer B wishes to use the service provided by Developer A. However, it's inconvenient for A to supply B with an update every time.

Java RMI provides a very easy solution! Since RMI can dynamically load new classes, Developer B can let RMI handle updates automatically for him. Developer A places the new classes in a web directory, where RMI can fetch the new updates as they are required.

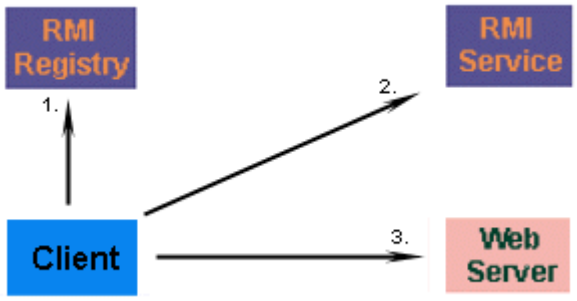


Figure 13.1. Connections made when client uses RMI

Figure 13.1 shows the connections made by the client when using RMI. Firstly, the client must contact an RMI registry, and request the name of the service. Developer B won't know the exact location of the RMI service, but he knows enough to contact Developer A's registry. This will point him in the direction of the service he wants to call.

Developer A's service changes regularly, so Developer B doesn't have a copy of the class. Not to worry, because the client automatically fetches the new subclass from a webserver where the two developers share classes. The new class is loaded into memory, and the client is ready to use the new class. This happens transparently for Developer B - no extra code need to be written to fetch the class.

Following steps are performed in order to implement any Java RMI service:

- Define the remote interface
- Implement the server

- Implement the client
- Compile the source files
- Start the Java RMI registry, server, and client

The diagram illustrates the process of setting up and accessing a remote object.

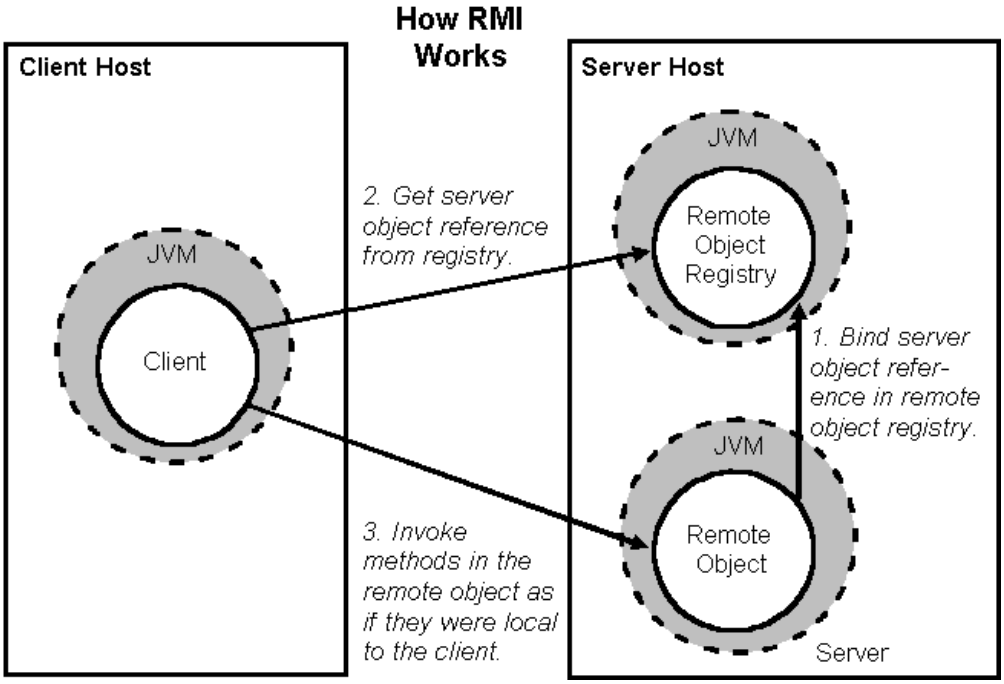


Figure 13.2. Accessing a Remote Object

The RMI Layered System

The RMI system consists of three layers:

- The stub/skeleton layer - client-side stubs (proxies) and server-side skeletons
- The remote reference layer - remote reference behavior (e.g. invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.

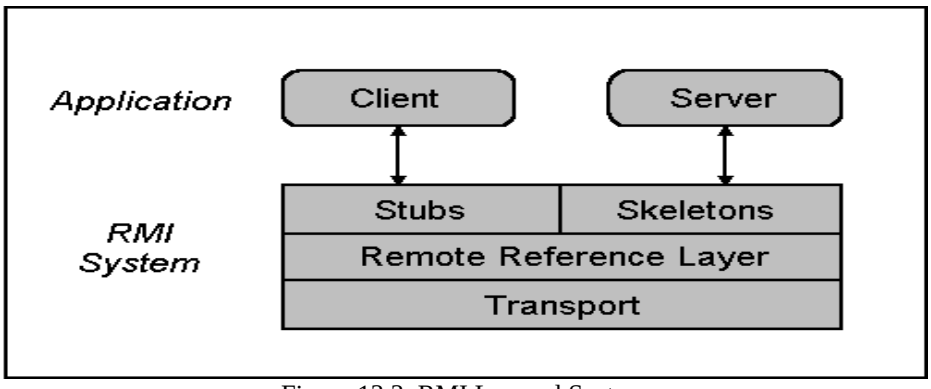


Figure 13.3. RMI Layered System

A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics-whether the server is a single object or is a replicated object requiring communications with multiple locations.

The transport is responsible for connection set-up, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to be done before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up-call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer and transport on the server side, and then up through the transport, remote reference layer and stub on the client side.

The RMI package only includes UnicastRemoteObject, which is a class in the RMI package that is extended by the client interface object to allow point-to-point communication. If other strategies are desired, they would have to be developed.

Java RMI

There are 6 steps involved in writing the RMI program as follows.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

RMI Example

In the following example shown in figure 13.4, all the 6 steps will be followed to create and run the RMI application. The client application needs only two files, remote interface and client application. In the RMI application, both client and server interact with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.

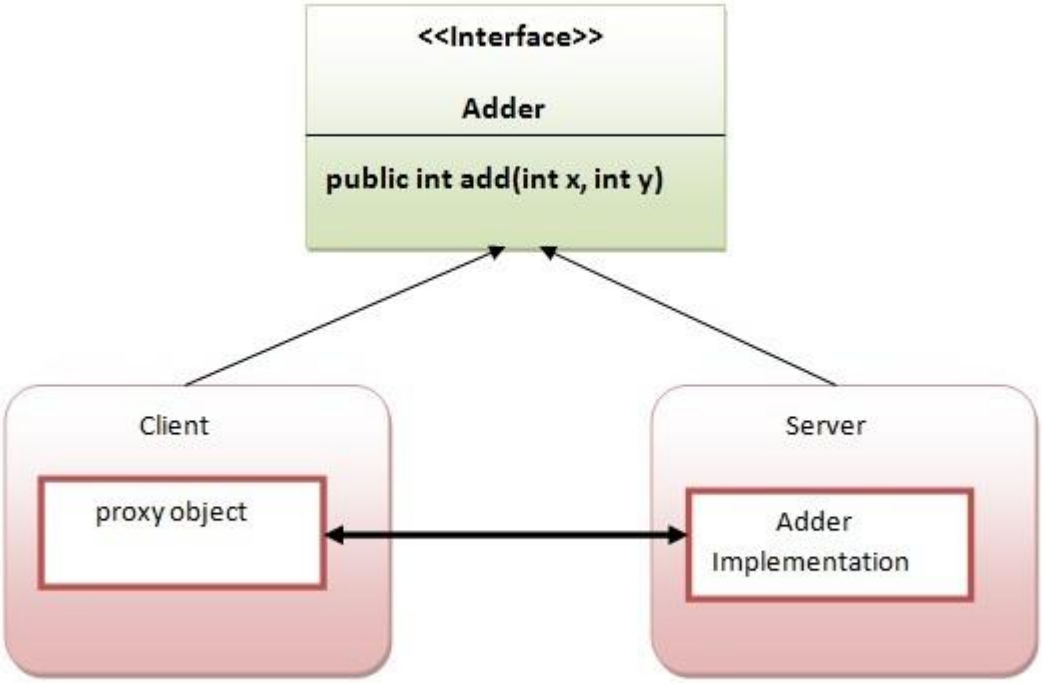


Figure 13.4. RMI Adder Example

1. Create the Remote Interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;
public interface Adder extends Remote{
public int add(int x,int y)throws RemoteException;
}
```

2. Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to either extend the UnicastRemoteObject class or use the exportObject() method of the UnicastRemoteObject class. In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adde
r{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

3. **Create the stub and skeleton objects using the rmic tool.**

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4. Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5. Create and run the server application

The rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It returns the reference of the remote object.
public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It binds the remote object with the given name.
public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;	It destroys the remote object which is bound with the given name.
public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;	It binds the remote object to the new name.
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;	It returns an array of the names of the remote objects bound in the registry.

In this example, we are binding the remote object by the name 'lab'.

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/lab",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/lab");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

Running RMI Application:
Following steps are followed to execute RMI applications.

1. Compile all the java files
2. javac *.java
3. Create stub and skeleton object by rmic tool
4. rmic AdderRemote
5. Start RMI registry in one command prompt
6. rmiregistry 5000
7. Start the server in another command prompt
8. java MyServer
9. Start the client application in another command prompt
10. java MyClient

Example 1 Source Code:

Adder.java

```
import java.rmi.*;
public interface Adder extends Remote{
public int add(int x,int y)throws RemoteException;}
```

AdderRemote.java

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder{
AdderRemote()throws RemoteException{
super();}
public int add(int x,int y){return x+y;}}
```

MyClient.java

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/lab");
System.out.println(stub.add(34,4));
}catch(Exception e){System.out.println(e);}
}}
```

MyServer.java

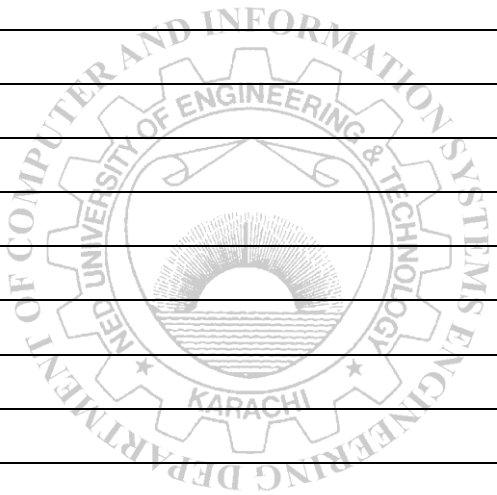
```
import java.rmi.*;
```

```
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/lab",stub);
}catch(Exception e){System.out.println(e);}
}}
```

EXERCISES

1. Implement the adder program using RMI as mentioned in Example1. Execute the program using all the steps mentioned above. Also, paste print out of code and output.

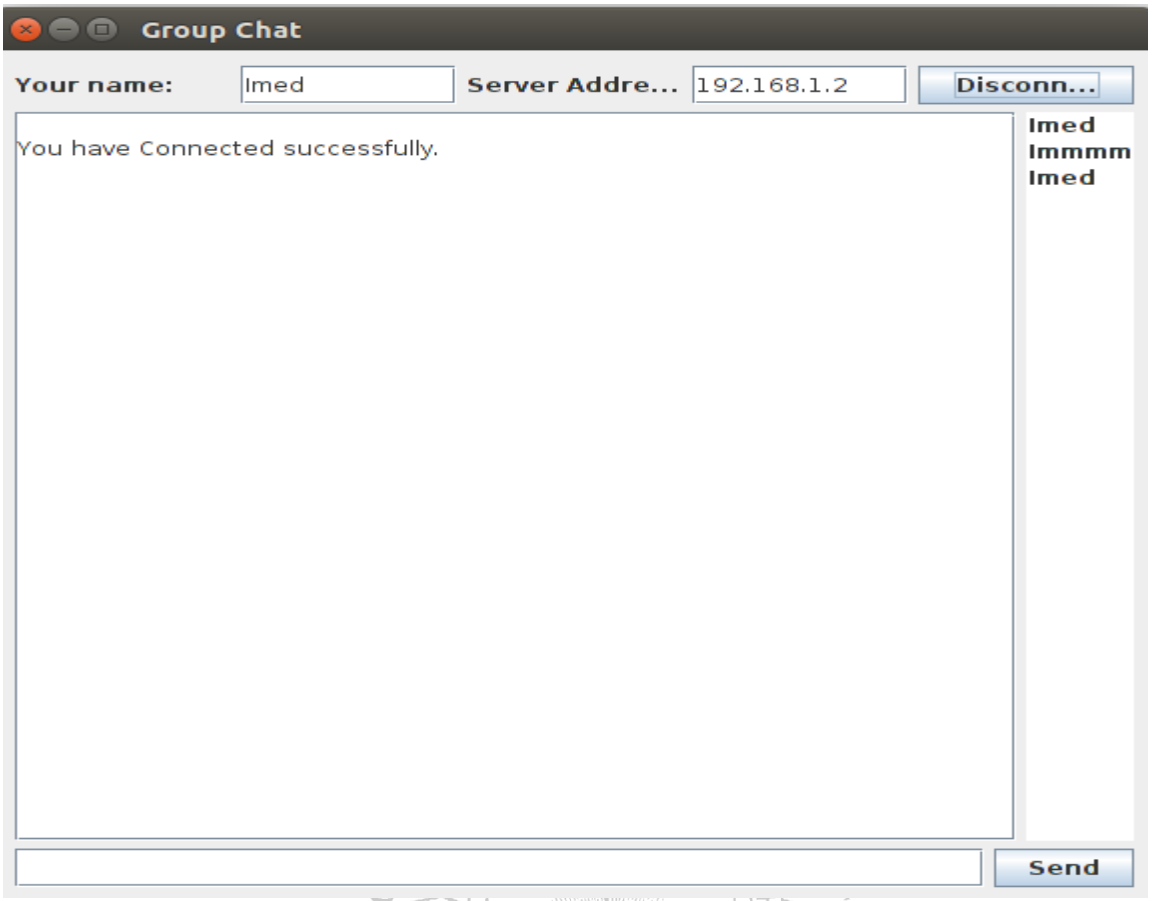
2. Implement the same RMI program (Example 1) involving two machines connected via LAN. Highlight the major changes in the source code.



3. Implement the multi-function calculator using RMI. Also, paste print out of code and output.

4. Implement a Group Chat example using *Java RMI* with a Graphical User Interface. Students are allowed to use any Integrated Development Environment (e.g. Eclipse or Net Beans) for the ease in development.
   This task must accomplish the following tasks namely:
   - Implementation of Server Side with RMI registry
   - Implementation of Client Side
   - Correct execution of both Server Side & Client Side
   - Paste the print out of source code and output.

Paste the screen shot of your final Group Chat Application.



Group Chat Application using Java RMI

Lab Session 14

Explore Web Services using Restful API

Web Service

Web service is the communication platform between two different or same platform applications that allows using their web method. Different or same platform application means that a web service can be created in any language, such as Java or other languages and that the language web service can be used in a .Net based application and also a .Net web service or in another application to exchange the information. The World Wide Web Consortium (W3C) has defined the web services. Web Services are the message-based design frequently found on the Web and in enterprise software. The Web of Services is based on technologies such as HTTP, XML, SOAP, WSDL, SPARQL, and others. Some of the major advantages include reusability, interoperability, loose coupling, ease of integration and deploy-ability. For example, a Web service may be offered in a business to business scenario whereby Company A provides a currency conversion Web service and Company B, in turn, uses this Web service to provide the currency conversion functionality to its customers. The Web service offered by Company A can also be used by Company C in a different capacity.

Web services is a broad term that represents all the technologies used to transmit data across a network by using standard Internet protocols, typically HyperText Transfer Protocol (HTTP). An eXtensible Markup Language (XML) format is used to represent the data, which is why Web services are sometimes known as *XML Web services.* An individual Web service can be considered as a piece of software that performs a specific task (also known as a *function*), and makes that task available by exposing a set of operations that can be performed (known as *methods* or *Web methods*) with the task. Additionally, each of the methods exposes a set of variables that can accept data passed into the method. These variables are known as *parameters* or *properties*. Together, the properties and methods refer to a Web service's *interface*. For example, Company A creates a Web service that provides currency rate functionality, which may expose a method called GetRate. Company B is then able to pass a parameter called CountryCode into the GetRate method. The GetRate method takes the CountryCode parameter, looks up the appropriate currency rate in a database, and returns the rate back to the program that requested it.

In this example, which database did Company A use to access the currency rate information? What was the name of the database server? What communication mechanisms and security mechanisms were used to access the database server? The answer to all of these questions is, "It doesn't matter." The beauty of a Web service is the concept of *encapsulation*. Encapsulation allows the complexity of retrieving the actual currency rate to be completely self-contained within the company that created the Web service (Company A). The only thing that Company B knows is that they called a Web service to get a currency rate and it was given to them.

Web services are made possible by placing the programs, or applications, on a Web server, such as Microsoft Internet Information Server (IIS). Because the application resides on a Web server, it can be called, or *invoked*, from any other computer on the network by using HTTP. The Web service provides seamless distributed computing across the entire network, as long as both sides know how to use a Web service.

One major advantage of invoking or creating Web services over HTTP is that if the Web server is on the Internet, the network administrators on both ends of the data transmission don't have to open any additional ports in their firewalls. All transmission of data is sent across port 80 (typically) by using HTTP. Port 80 is always open in a firewall because it is the same port used to browse the Internet. The fact that the network administrators don't need to open additional ports means that you face virtually no additional security risk in using Web services. Another major advantage in Web services is that (because

Web services conform to open standards) a Web service written on one platform (such as the Microsoft platform) can call another Web service written on another platform (such as Linux). Because of their innate flexibility, Web services make the notion of *software as a service* a real possibility. And because Web services provide integration between two systems, *software as a service* refers to the possibility of not having to install software on workstations or servers, but rather, being able to use it from across the Internet.

RESTful Web Services

RESTful web services are services that are built to work best on the web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induces desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains architecture to client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourage RESTful applications to be simple, lightweight, and have high performance. RESTful web services typically map the four main HTTP methods to the operations they perform: create, retrieve, update, and delete. The following table shows a mapping of HTTP methods to the operations they perform.

Table 14.1 Mapping HTTP Methods to Operations Performed

HTTP Method	Operations Performed
GET	Get a resource
POST	Create a resource and other operations, as it has no defined semantics
PUT	Create or update a resource
DELETE	Delete a resource

Typically, a RESTful Web service should define the following aspects:

- The base/root URI for the Web service such as http://host/<appcontext>/resources.
- The MIME type of the response data supported, which are JSON/XML/ATOM and so on.
- The set of operations supported by the service. (for example, POST, GET, PUT or DELETE).

Java Jersey API

Developing RESTful Web services that seamlessly support exposing your data in a variety of representation, media types and abstract away the low-level details of the client-server communication is not an easy task without a good toolkit. In order to simplify development of RESTful Web services and their clients in Java, a standard and portable JAX-RS API has been designed. Jersey RESTful Web Services framework is open source, production quality, and framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides its own API that extends the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Jersey also exposes numerous extension SPIs so that developers may extend Jersey to best suit their needs. Jersey contains three major parts.

- Core Server: By providing annotations and APIs standardized in JSR 311, one can develop a RESTful Web service in a very intuitive way.
- Core Client: The Jersey client API helps to easily communicate with REST services.
- Integration: Jersey also provides libraries that can easily integrate with Spring, Guice, Apache Abdera, and so on.

Example 1:
Configuration of Java Jersey for developing Restful Web Services

The configuration of Java Jersey involves the following steps.
Install NetBeans and Tomcat

NetBeans
Download the J2EE versions of NetBeans, which includes GlassFish and Tomcat web servers. In this lab use Tomcat will be used.

1. Open the install then when you get to the *Installation Type* screen, click on the **Customize** button.
2. Select the *Apache Tomcat* package
3. Click **Install** to finish.

Configure Tomcat
Tomcat is configured in order to run and deploy web services. To run web services we need a Tomcat user with manager-script privileges. The easiest way to do that is let NetBeans do that for us by deleting the Tomcat server when NetBEans is installed, then adding it back in.

1. Run NetBeans.
2. When NetBeans comes up, Right click on the Servers > Apache Tomcat or TomEE in the Services tab.
3. Select the Apache Tomcat or TomEE server.
4. Click on the Remove Server button.
5. Click on the Add Server.. button.
6. Select Apache Tomcat or TomEE from the list of servers in the Choose Server box.
7. Set the server Name to Apache Tomcat.
8. Click on Next.
9. In the Add Server Instance panel, click on Browse to find your Tomcat server directory.
10. Select the Tomcat directory and click on Open.
11. For both Username and Password enter tomcat to make things simple. For production systems you'll want a better password than that.
12. Set the Server Port to 8082 for this example.
13. Make sure the Create user if it does not exist box is checked.
14. Click the Finish button.
15. Back in the Servers panel click the Close button.
16. Open up $HOME\conf\tomcat-users.xml file. $HOME is the directory where you installed Tomcat.
17. Add manager-gui to the list of roles for the tomcat user.

Test NetBeans-Tomcat Integration
It is required to test whether Tomcat can now be started within NetBeans.

1. Right click **Servers > Apache Tomcat > Start** in the NetBeans **Services** tab.
2. Open http://localhost:8082 in a browser. That action should open the main Apache Tomcat screen.
3. Click on the **Manager App** button. Then enter tomcat for the user name and password. That action should open the **Tomcat Web Application Manager** screen, an example of which is shown in figure 14.1.

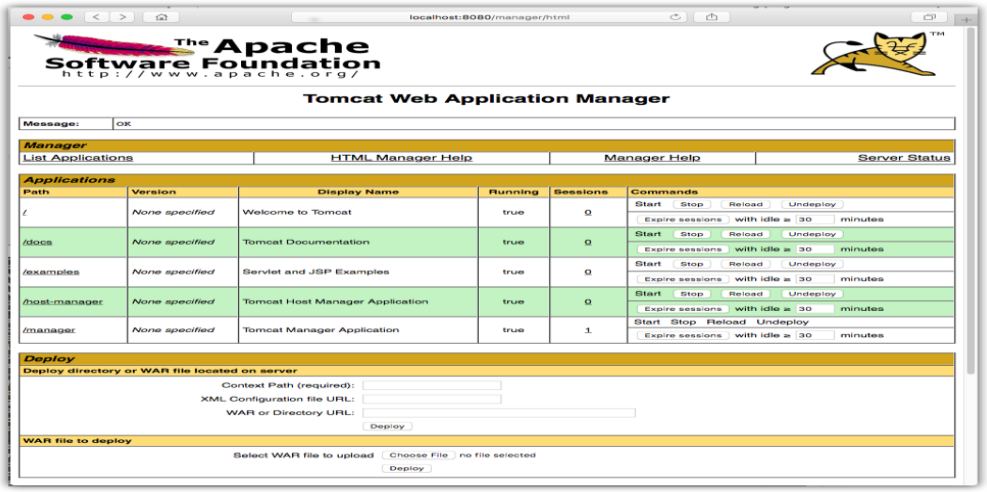


Figure 14.1. Tomcat Web Application Manager

Create RESTful Web Services

Create a Web Application Project

In order to use Jersey API in the ***J2EE versions of NetBeans***, We are not required to add Jersey jar files externally as they are added by default. After configuring the development environment, you first need to build a base web application to which you will add a RESTful services class and methods later.

1. Select **File > Open > New Project…**
2. Select the **Java Web** category and Web Application project, then click Next.
3. Enter the project name as *TASK1*.
4. Set the directory location of the project, then click **Next**. This action sets the top level REST path variable in the *context.xml* file.
5. In the **Server and Settings** screen, select **Apache Tomcat** in the **Server** field, then click **Next**.
6. We will not add any additional application frameworks, so click **Finish** in the **Frameworks** screen. You now have a simple web application that will load the index.html page. To run the application, click on the green triangle under the NetBeans menu bar, select **Run > Project** from the menu bar or press **F6.**

GET Service Handler

Now that we have a basic web application, we will add a RESTful service that responds to an HTTP GET request.

1. Right click on **TASK1** in the **Projects** tab.
2. Select **New…** then select **RESTful Web Services from Patterns…**
3. Select the **Simple Root Resource** in the **Select Pattern** screen as depicted in figure 14.2.
4. Then click **Next**.

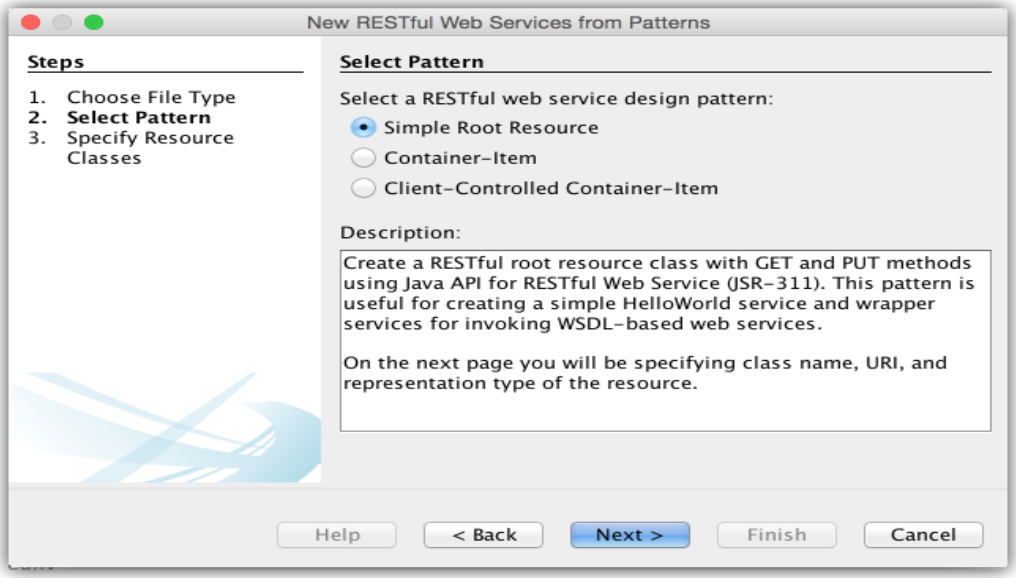


Figure 14.2. Select Pattern Screen

5. In the Specify Resource screen set the name of the Java package that will contain your service class. The name can be anything, but it is usually a domain name in reverse order. In this example we'll use com.example.
6. Set the service Path to service and the service Class Name to ServiceResource.
7. Set the MIME type for the GET service to text/html for the simple GET service then click on Finish.

You will now see a section in your project called RESTful Web Services. If you open this section you can see the service class with GET and PUT handler methods that have been created. Replace the contents of the getHtml() with the return statement as shown below. Note that the content type of the return string is specified by the @Produces("text/html") Jersey annotation.

```
package com.example;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PUT;
import javax.ws.rs.POST;
@Path("service")
public class ServiceResource {
    @Context
    private UriInfo context;
    public ServiceResource() {
    }
    @GET
           @Produces("text/html")
    public String getHtml() {
                   return "<h1>TASK ONE!</h1>";
    }
    @PUT
           @Consumes("text/html")
    public void putHtml(String content) {
    }
```

```
    @POST
@Consumes("application/x-www-form-urlencoded")
@Produces("text/plain")
public String postHandler(String content) {
    return content;
}
}
```

Next we need to set the path to the web service so it will be automatically invoked when we test it with the default browser.

8. In the Projects tab right click on the TASK1 project then select Properties.
9. In the Categories section select Run.
10. When we test the web service the URL to the GET service will be http://localhost:8082/TASK1/webresources/service.
11. Recall that the TASK1 part of the URL is defined in the path attribute in the context.xml file which you can find in the configuration files of the project.
12. The webresources part of the path is set in the @javax.ws.rs.ApplicationPath("webresources") annotation in the ApplicationConfig.java file that NetBeans created.
13. The service part is defined by the @Path annotation in the ServiceResource.java file. Enter /webresources/service in the Relative URL field then click OK.
14. There are two ways to execute this web service.
15. First method is to right click project TASK1 and select Test RESTful Web Services. Select Web Test Client in Project and click OK.
16. This will automatically open web browser as shown in Figure 14.3.
17. Second option to execute is to simply run the project by F6. And type the following url in web browser.

    http://localhost:8082/TASK1/webresources/service

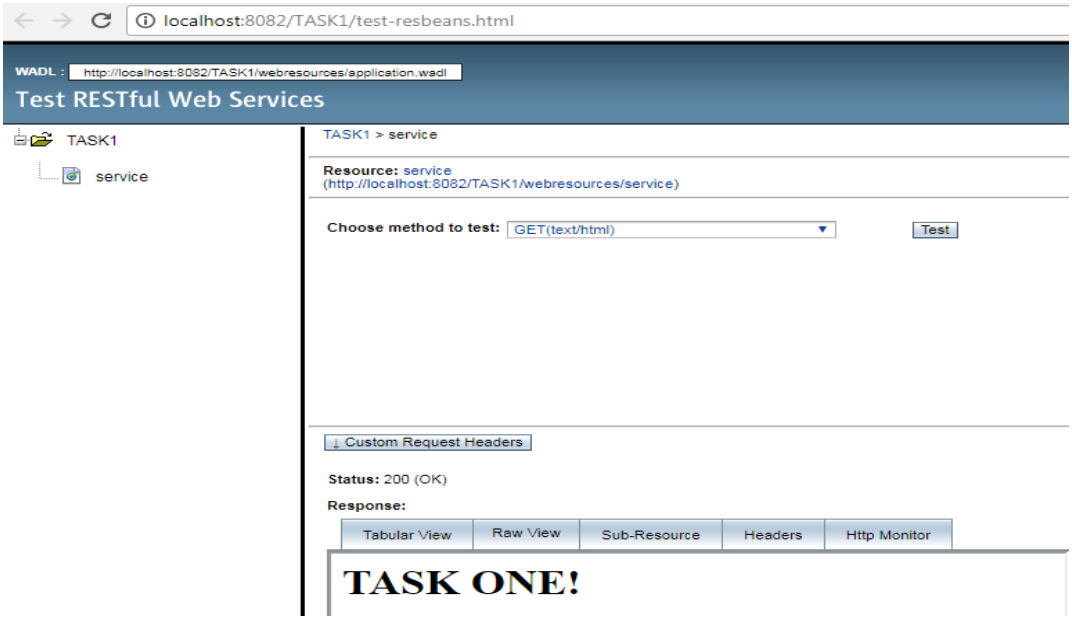


Figure 14.3. Execution by Method 1

Jersey Web Service Annotations

Jersey provides a set of Java annotations that can be used to define the web service structure.

- @Path – Defines the URL relative path to the given resource. We saw in the previous section that the complete URL to our service was defined by fields in the context.xml and web.xml files plus the @Path field. You can change the relative path of any service by modifying these three fields. For example if you wanted to simplify the GET service URL to be http://localhost:8082/TASK1/service you would set the url-pattern field in web.xml to /*.
- @Context – Defines parameters that can be extracted from the request.
- @GET – Identifes the method in the service class that will handle GET requests.
- @Produces – Specifies the MIME type of the response that is returned to the client.
- @Consumes – Specifies the content type that the service will accept as input.
- @PUT – Identifies the method that will handle PUT requests.
- @DELETE – Identifies the method that will handle DELETE requests.
- @Consumes – Specifies the MIME type that the method immediately following the annotation will accept from the client. It is possible to define multiple MIME types that will each be handled by a separate method. The GET service example does not have any arguments so the @Consumesannotation was not needed.
- @QueryParam – Denotes a field that will be extracted from the URL in a GET request after the ? character.
- @DefaultValue – Denotes the value that will be used to fill a variable in a @QueryParam list that is missing.
- @PathParam – Denotes a field that will be extracted from a field in the URL path.

Example 2:

Create a web service which will convert temperature in Celsius to Fahrenheit and temperature in Fahrenheit to Celsius. Use the methodology defined above to create the new project. Add two source files namely CtoFService.java and FtoCService.java.

CtoFService.java:

```
package com.example;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
@Path("/ctofservice")
public class CtoFService {
      @GET
      @Produces("application/xml")
      public String convertCtoF() {
            Double fahrenheit;
            Double celsius = 36.8;
            fahrenheit = ((celsius * 9) / 5) + 32;
            String result = "@Produces(\"application/xml\") Output:
\n\nC to F Converter Output: \n\n" + fahrenheit;
            return "<ctofservice>" + "<celsius>" + celsius +
"</celsius>" + "<ctofoutput>" + result + "</ctofoutput>" +
"</ctofservice>";
      }
      @Path("{c}")
      @GET
      @Produces("application/xml")
      public String convertCtoFfromInput(@PathParam("c") Double c) {
            Double fahrenheit;
            Double celsius = c;
```

```
            fahrenheit = ((celsius * 9) / 5) + 32;
            String result = "@Produces(\"application/xml\") Output:
\n\nC to F Converter Output: \n\n" + fahrenheit;
            return "<ctofservice>" + "<celsius>" + celsius +
"</celsius>" + "<ctofoutput>" + result + "</ctofoutput>" +
"</ctofservice>";
      }
}
```

FtoCService.java.

In order to execute this an external jar "json-20131018.jar" is added in path to include json functions.

```
package com.example;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;
import org.json.JSONException;
import org.json.JSONObject;
@Path("/ftocservice")
public class FtoCService {
        @GET
        @Produces("application/json")
        public Response convertFtoC() throws JSONException {
            JSONObject jsonObject = new JSONObject();
            Double fahrenheit = 98.24;
            Double celsius;
            celsius = (fahrenheit - 32)*5/9;
            jsonObject.put("F Value", fahrenheit);
            jsonObject.put("C Value", celsius);
            String result = "@Produces(\"application/json\") Output:
\n\nF to C Converter Output: \n\n" + jsonObject;
            return Response.status(200).entity(result).build();
        }
        @Path("{f}")
        @GET
        @Produces("application/json")
        public Response convertFtoCfromInput(@PathParam("f") float f)
throws JSONException {
            JSONObject jsonObject = new JSONObject();
            float celsius;
            celsius =  (f - 32)*5/9;
            jsonObject.put("F Value", f);
            jsonObject.put("C Value", celsius);
            String result = "@Produces(\"application/json\") Output:
\n\nF to C Converter Output: \n\n" + jsonObject;
            return Response.status(200).entity(result).build();
        }
}
```

- Make sure that addRestResourceClasses function of ApplicationConfig.java contains,

```
resources.add(com.example.CtoFService.class);
resources.add(com.example.FtoCService.class);
```

- Now execute the project using both the methods described above.
- **Method 1 Output is follows:**

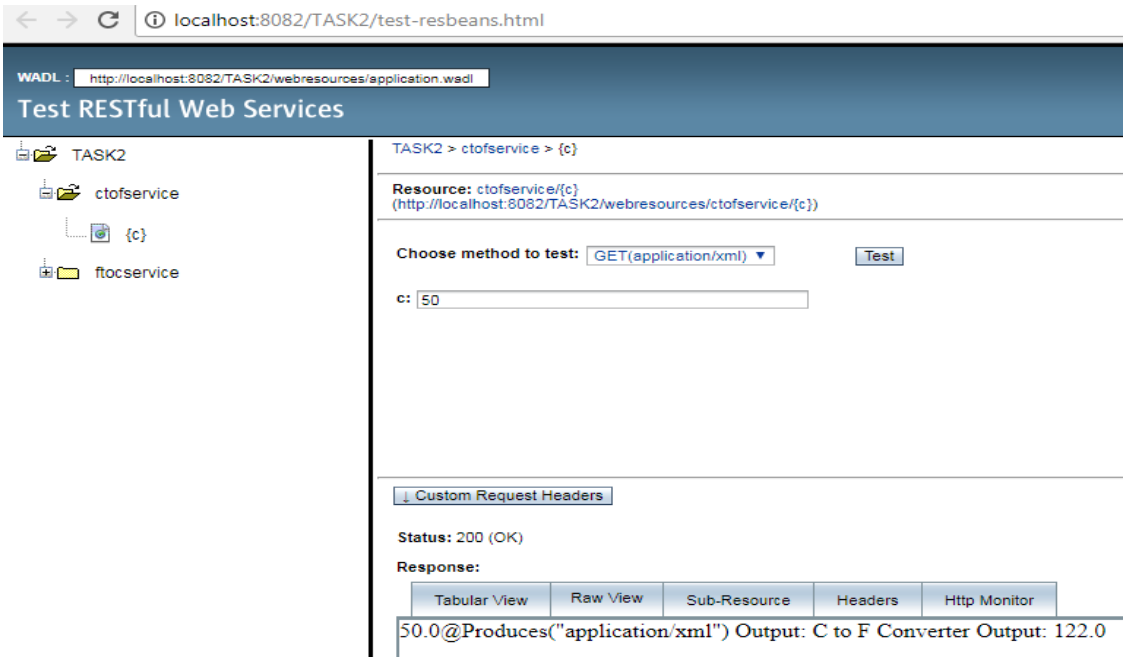


Figure 14.4. Method 1 Output

- **Method 2 Output is follows:**

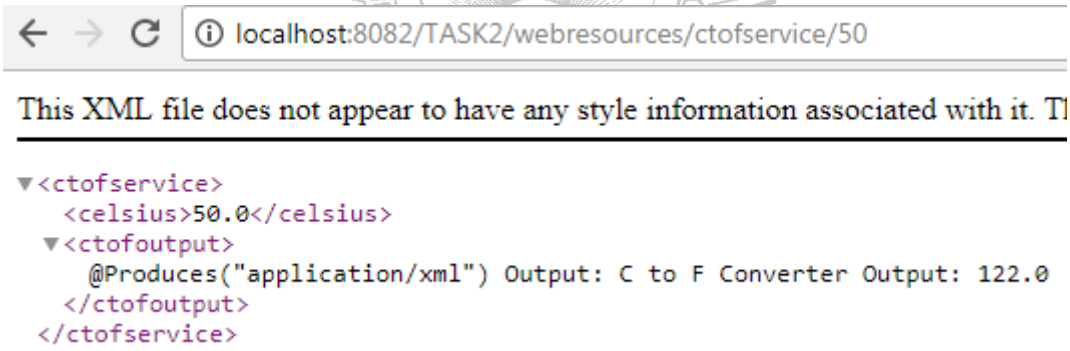


Figure 14.5. Method 2 Output

Example 3:

Design a web service for multi-function calculator. Follow the above described method for project creation. Add a new class calculator.java and add the following code in it.

```
package com.example;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("calculator service")
```

```
public class Calculator {
     String returnMsg;
      @GET
       public String Calculate() {
           returnMsg = "Calculator Web Service ";
                return returnMsg;
          }
        @GET
        @Path("/{param}")
        public String getMsg(@PathParam("param") String msg) {
                returnMsg = "Jersey say : Hello " + msg;
                return returnMsg;
        }
@GET
@Path("/add/{a}/{b}")
        @Produces(MediaType.TEXT_PLAIN)
public String addPlainText(@PathParam("a") int a, @PathParam("b")
int b) {
      returnMsg = "Addition of " +a+ " and " +b+ " is : " + (a + b);
                return returnMsg;
        }
        @GET
@Path("/sub/{a}/{b}")
@Produces(MediaType.TEXT_PLAIN)
public String subPlainText(@PathParam("a") int a, @PathParam("b")
int b) {
   returnMsg = "Subtraction of " +a+ " and " +b+ " is : " + (a - b);
                return returnMsg;
        }
        @GET
@Path("/mul/{a}/{b}")
@Produces(MediaType.TEXT_PLAIN)
public String mulPlainText(@PathParam("a") int a, @PathParam("b")
int b) {
returnMsg = "Multiplication of " +a+ " and " +b+ " is : " + (a * b);
return returnMsg;
}
@GET
@Path("/div/{a}/{b}")
@Produces(MediaType.TEXT_PLAIN)
public String divPlainText(@PathParam("a") int a, @PathParam("b")
int b) {
if(b == 0){
returnMsg="Division by 0 is not permitted";
return returnMsg;
                }
returnMsg= "Division of " +a+ " and " +b+ " is : " + (a / b) + " and
the Remainder is : " + (a%b);
return returnMsg;
}
        @GET
@Path("/sqr/{a}")
@Produces(MediaType.TEXT_PLAIN)
public String sqrPlainText(@PathParam("a") int a) {
returnMsg = "Square of "+a+ " is : " + (a*a) + "";
```

```
return returnMsg;
        }
        @GET
@Path("/cube/{a}")
@Produces(MediaType.TEXT_PLAIN)
public String cubePlainText(@PathParam("a") int a) {
returnMsg = "Cube of " + a + " is : " + (a*a*a) + "";
return returnMsg;
        }
}
```

Execute the project with both the methods.

Method 1 Output:

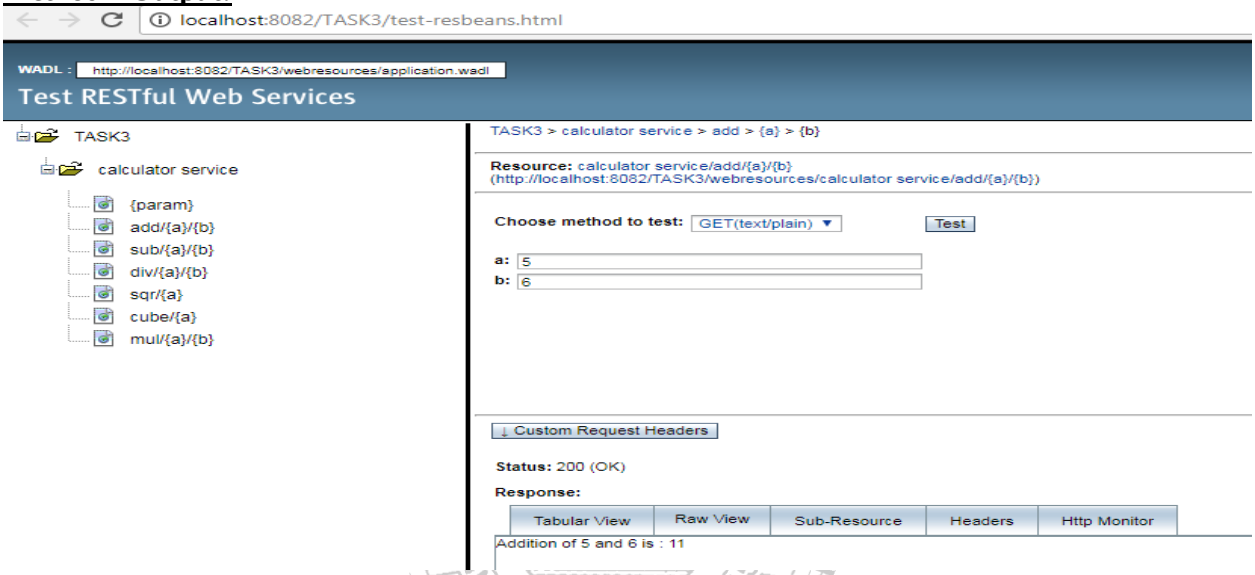


Figure 14.5. Method 1 Output

Method 2 Output:

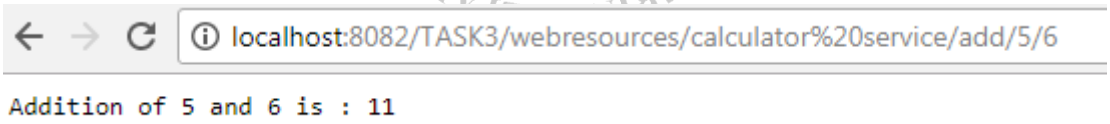


Figure 14.6. Method 2 Output

Exercises

Implement the above example (1 to 3) programs and paste their outputs.

Outputs:

_____

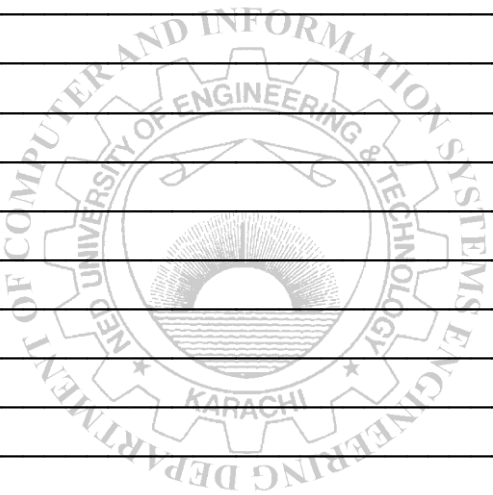
_____

_____

_____

_____

_____


NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI
DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING

2. Create RESTful Web Service from a database. Use the sample database of Java DB. The web service is required to display the customer table in XML format. Attach print out of code and output. Use the following link for reference.

https://netbeans.org/kb/docs/websvc/rest.html