2

GLOBAL TOROIDAL CODE INCLUDING X-POINT (GTC-X)

2.1 CRAY SYSTEM OVERVIEW ??

Cray XC40 system is the latest entry to SERC's HPC class of systems. This Cray XC40 is a system that combines the capabilities of Intel's latest Xeon Haswell processors for the CPU cluster and Nvidia's K40 series of GPU cards and Intel Xeon-Phi Processor 7210 for the accelerator cluster connected using Crays's own Aries high speed interconnect on a dragonfly topology with DDN's high performance storage units.

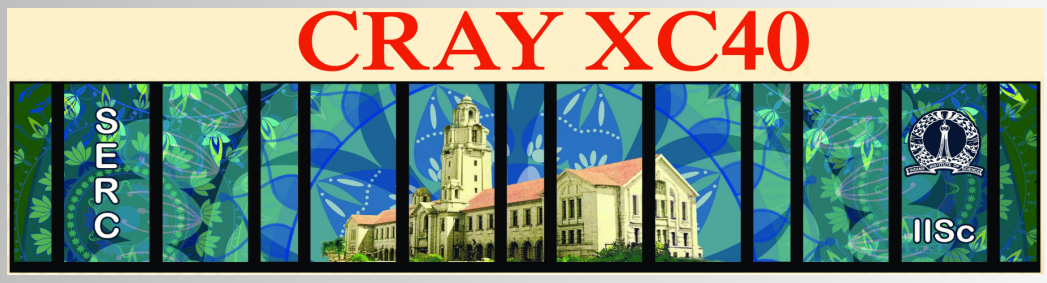


Fig. 26: CRAY at IISc.

CPU only cluster: Intel Haswell 2.5 GHz based CPU cluster with 1376 nodes; each node has 2 CPU sockets with 12 cores each, 128GB RAM and connected using Cray Aries interconnect.

Type of node	Processor Make	No. of CPU cores/ node	RAM / node	Total No. of Nodes	Total cores	Total RAM
Compute nodes	Intel Haswell processor 12 core operating at 2.5 GHz	24	128 GB	1376	33024	172 TB
Intel Xeon phi nodes	Intel IvyBridge processor 12 core operating at 2.4 GHz	12(CPU)+1(Phi 5120D card)	64 GB	48	576(CPU)+2928(MIC cores)	3 TB
GPU nodes	Intel IvyBridge processor 12 core operating at 2.4 GHz	12(CPU)+1(GPU K40 card)	64 GB	44	528(CPU)+ 126720(cuda cores)	2.75 TB
Service/login nodes	Intel SandyBridge processor 8 core operating at 2.6 GHz	8	32 GB	15	240	480 GB
DVS nodes	Intel Haswell processor 12 core operating at 2.5 GHz	24	128 GB	8	192	1 TB

Fig. 27: CRAY XC40 system configuration.

Accelerator based clusters: Two accelerator clusters, one with Nvidia GPU cards (44 nodes) and the other with Intel Xeon-Phi Processor

7210(24 nodes). Each node with Nvidia GPU on the accelerator cluster has Intel IvyBridge 2.4 GHz based single CPU socket with 12 cores. The GPU card is Tesla K40 card with 2880 cores and 12GB device memory. Nodes with Intel Xeon-Phi Processor 7210 are self-hosting nodes, it has 64 cores with 16GB accelerator memory and 96GB DDR4 memory per node.

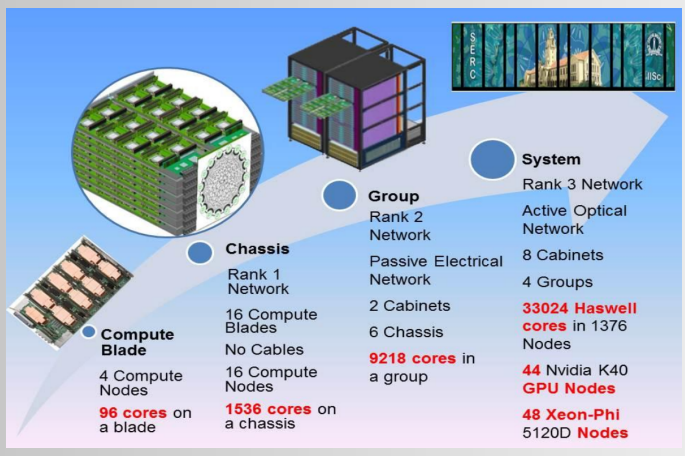


Fig. 28: CRAY XC40 system building blocks.

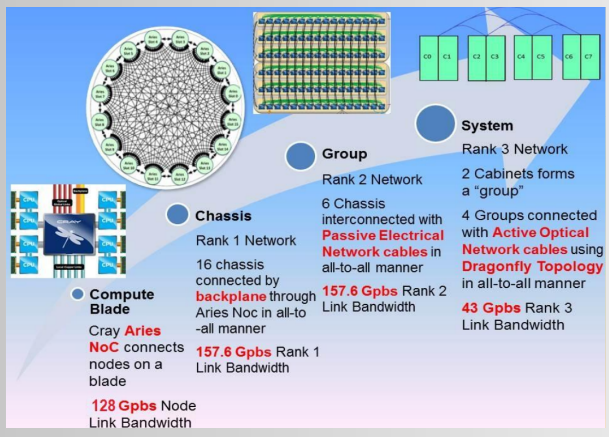


Fig. 29: CRAY XC40 interconnection building blocks.

High Speed Storage: 2 PB usable space provided by high speed DDN storage unit supporting Cray's parallel Lustre filesystem.

No. of cabinets	No. of Enclosures/ cabinet	No. of disks/ enclosure	Total no. of disks	Total Raw Capacity	Total Usable File System Capacity	IOR Performance	
						Read	Write
4	5	48 X 3 TB	960	2.88 PB	2 PB	27.7 GB/sec	32.2 GB/sec

Fig. 30: CRAY XC40 storage configuration.

Software environment: The entire system is built to operate using Cray's customized Linux OS, called Cray Linux Environment, for the cluster supporting architecture specific compilers from Cray as well as Intel and open-source based Gnu compilers. System also hosts architecture specific parallel libraries like OpenMP, MPI, CUDA and Intel Cluster software. Extensive range of parallel Scientific and Mathematical libraries like BLAS, LAPACK, Scalapack, fftw, hdf5, netcdf, PETSc, Trilinos etc. are also available on the system. To facilitate users with parallel program development latest version of the DDT parallel debugger and profiler is enabled for use on the system.

Type of node	Number of Nodes	HPL (sustained)
Compute cluster	1296	901 TeraFlops
GPU cluster	44	52 TeraFlops
Xeon Phi cluster	42	28 TeraFlops

Fig. 31: CRAY XC40 system performance.

Vendor
OEM – Cray Inc
Authorised Seller – Cray Inc, Seattle, WA 98164, USA

2.1.1 *MPI*

- Compiler: To run MPI commands we need an MPI FORTRAN compilers such as mpifort, mpif90, mpif77, etc. Any of the above compilers can be installed in a linux system from their corresponding repository. We compile the code in the same way we would compile a normal FORTRAN code:

```
<COMPILER_NAME> <FILE_NAME>
```

  And the compiler generates the "a.out" file for execution, but to execute this file we need a separate command that also specifies the number of cores and threads to be used.

- Execution: A normal code can be executed as "./a.out". But to execute an MPI code we need to install any of "aprun", "mpirun" or "mpiexec" and the command is

```
<EXECUTION_COMMAND>  -n  <NUM_MPI>  ./a.out
```

- Architecture: In the MPI mode of operation the same code is run in different cores in the system. In general we assign one of the cores as the master and operate the rest of the cores in slave mode, which is specified within the code. The MPI initialization is performed in the code and the number of MPI process or the cores to be used is specified during execution.

  The same code will be run in all the cores. For example, the code 'Test.F90'

```
program Test
   implicit none

   write(*,*)'Hello'
end program test
```

  can be compiled as

```
gfortran Test.F90
```

  and can be run using

```
mpirun -n 8 ./a.out
```

  Here 8 stands for the number of processes or cores to be used to run the code, which can be greater than the number of system cores as well, and the output will be

```
Hello
Hello
Hello
Hello
```

```
Hello
Hello
Hello
Hello
```

Notice that the code itself doesn't have any extra information about the MPI process.

In our applications, there are only two ways in which each processes can be forced to run differently.

1. Using random number generator: If the code calls and makes use of random numbers than the output will change accordingly. For example, the code

```
program Test

   implicit none
   real :: r(1)

   CALL RANDOM_NUMBER(r)
   write(*,*)r

end program test
```

   and when compiling and running as above the output was

```
                    0.279739141
                    0.915168107
                    0.751220167
                    0.416352272
                    0.591016054
                    0.454270780
                    0.662734151
                    0.574757993
```

2. Initializing and using the process ID: The MPI process can be initialized and the process ID can be accessed using the following code

```
PROGRAM Test_MPI
   include 'mpif.h'

   integer ProcessID, NumProcess, ierror

   call MPI_INIT(ierror)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, NumProcess,
       ierror)
   call MPI_COMM_RANK(MPI_COMM_WORLD, ProcessID, ierror
       )

   write(*,*)'ProcessID = ',ProcessID, 'NumProcess =',
       NumProcess
```

```
12     call MPI_FINALIZE(ierror)
   END PROGRAM
```

The above code when compiled with "mpif90" and executed with "mpirun" we get

```
                      ProcessID = 0 NumProcess = 8
 2                    ProcessID = 1 NumProcess = 8
                      ProcessID = 2 NumProcess = 8
                      ProcessID = 3 NumProcess = 8
                      ProcessID = 4 NumProcess = 8
                      ProcessID = 6 NumProcess = 8
 7                    ProcessID = 7 NumProcess = 8
                      ProcessID = 5 NumProcess = 8
```

Notice that the order of "ProcessID" output is random, as the different processes may take different times for execution. Also note that the ProcessID starts from 0.

3. Assigning a "Master": Once we have acess to ProcessID we can make one of the processes as the master, using an IF statement. Also, it is a good practice to only allow the MASTER to read and write files, so that there is no unnecessary clash for read-write acess rights. To ensure that all the process have access to the read files data the MASTER can broadcast the data as below

```
   PROGRAM Test_MPI_BC
 2 include 'mpif.h'

   integer ProcessID, NumProcess, ierror
   real :: r2d(3,2),r1d(6)

 7 call MPI_INIT(ierror)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, NumProcess, ierror)
   call MPI_COMM_RANK(MPI_COMM_WORLD, ProcessID, ierror)

   r2d = 0.0
12
   if( ProcessID==0 )  then
      open (1, file = 'data.dat', status = 'old')

      do i = 1,3
17       read(1,*) r2d(i,1), r2d(i,2)
      end do
   endif

   write(*,*)'ProcessID:',ProcessID,'Before value',r2d
22
   r1d = RESHAPE( r2d , (/6/) )

   call MPI_BCAST(r1d,6,MPI_REAL,0,MPI_COMM_WORLD,ierror)
```

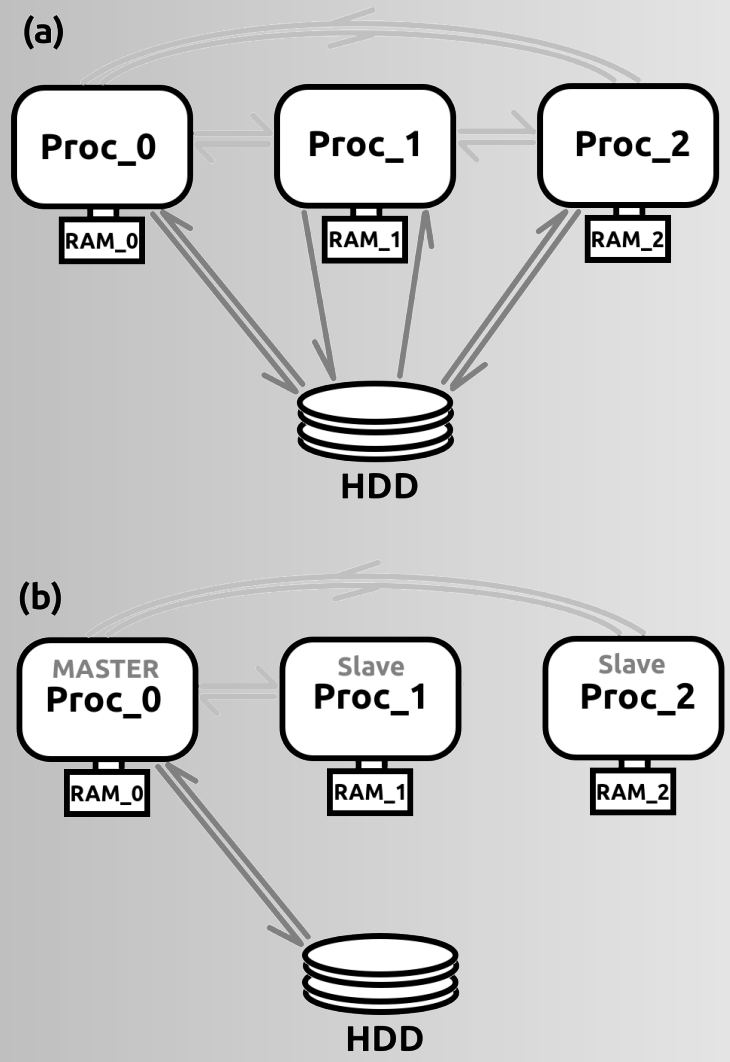


Fig. 32: A schematic of the interaction between the different processes in (a) all interacting, (b) Master-Slave interacting mode.

```
27  r2d = RESHAPE( r1d , (/3,2/) )

    write(*,*)'ProcessID:',ProcessID,'After value',r2d

    call MPI_FINALIZE(ierror)
32  END PROGRAM
```

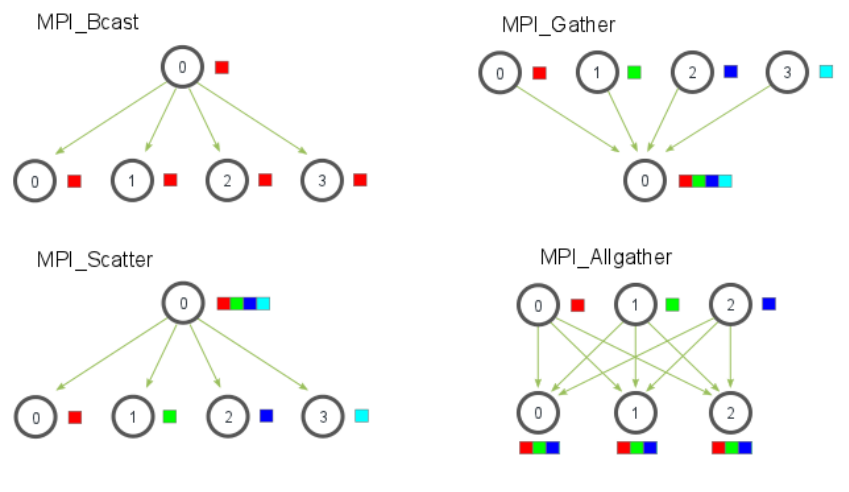
Now if the 'data.dat' is

```
    0.1 0.2
    0.3 0.4
 3  0.5 0.6
```

then the output when the above code is executed with 2 processes is

```
   ProcessID: 0 Before value  0.1      0.3      0.5
   0.2      0.4      0.6 ProcessID: 1 Before value  0.0
   0.0      0.0      0.0      0.0      0.0
2   ProcessID: 0 After value  0.1      0.3      0.5
       0.2      0.4      0.6
    ProcessID: 1 After value  0.1      0.3      0.5
       0.2      0.4      0.6
```

4. **Gathering and scattering data:**




```
1  PROGRAM scatter_mpi
       include 'mpif.h'

       integer process_Rank, size_Of_Cluster, ierror,
           message_Item
       integer scattered_Data
6      integer, dimension(4) :: distro_Array
       distro_Array = (/39, 72, 129, 42/)

       call MPI_INIT(ierror)
       call MPI_COMM_SIZE(MPI_COMM_WORLD, size_Of_Cluster,
           ierror)
11     call MPI_COMM_RANK(MPI_COMM_WORLD, process_Rank,
           ierror)
       call MPI_Scatter(distro_Array, 1, MPI_INT,
           scattered_Data, 1, MPI_INT, 0, MPI_COMM_WORLD,
           ierror);

       print *, "Process ", process_Rank, "received: ",
           scattered_Data
       call MPI_FINALIZE(ierror)
16
   END PROGRAM
```

And the corresponding output when run on 4 MPI's is

```
                Process 1 received: 39
                Process 0 received: 72
3               Process 3 received: 129
                Process 2 received: 42
```

5. **Send and receive data:**
6. **Partition the MPI's:**

- Inititialization

2.1.2 *Vectorization*

The L2 cache is faster than the RAM and acts as an intermediary between the CPU and the RAM. The CPU searches for the variable within the cache before checking in the RAM. So if the often used variables are stored in the cache it helps reduce the access time. Furthermore, since the data is loaded into the cache in a block format, vectorized storage of data helps reduce the RAM access.