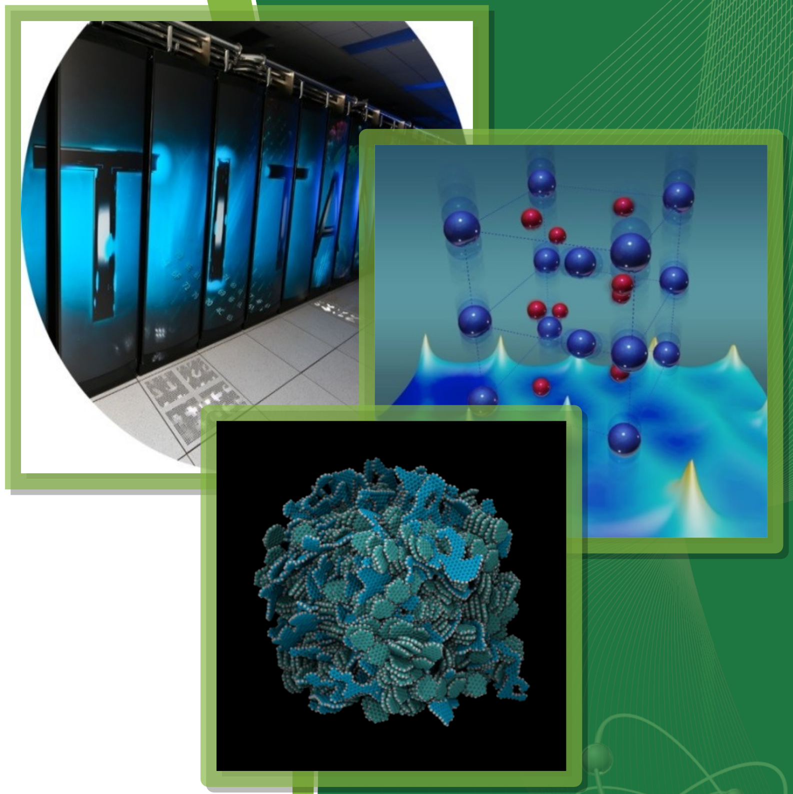
MPI Split type

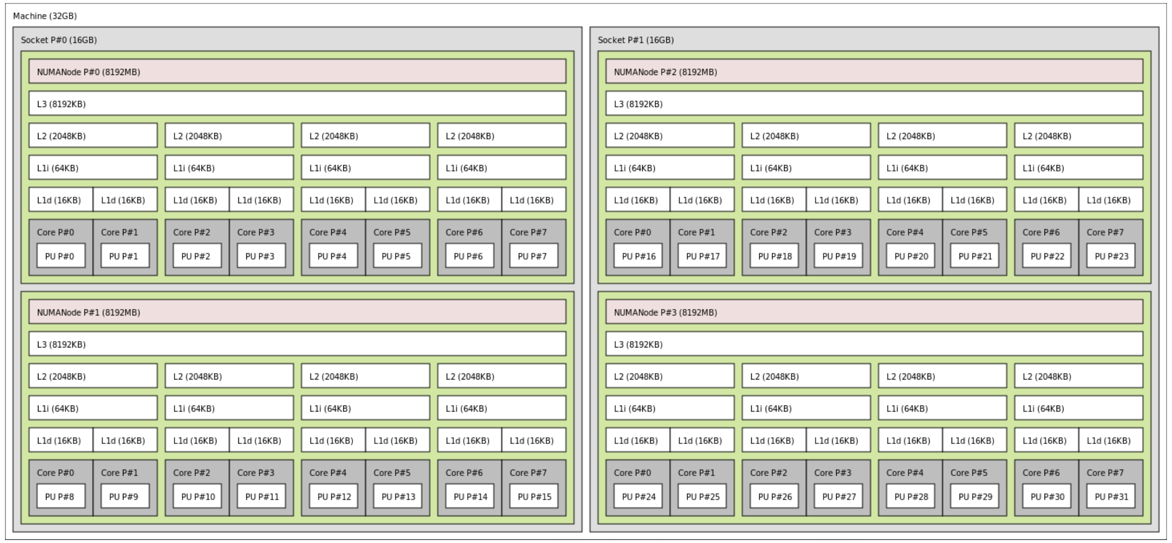


MPI split types – Initial assumptions

- SMP model, i.e., compute nodes is composed of ccNuma/NUMA nodes
- Hierarchical/tree-based architecture
- MPI_COMM_TYPE_SHARED

“this type splits the communicator into subcommunicators, each of which can create a shared memory region”

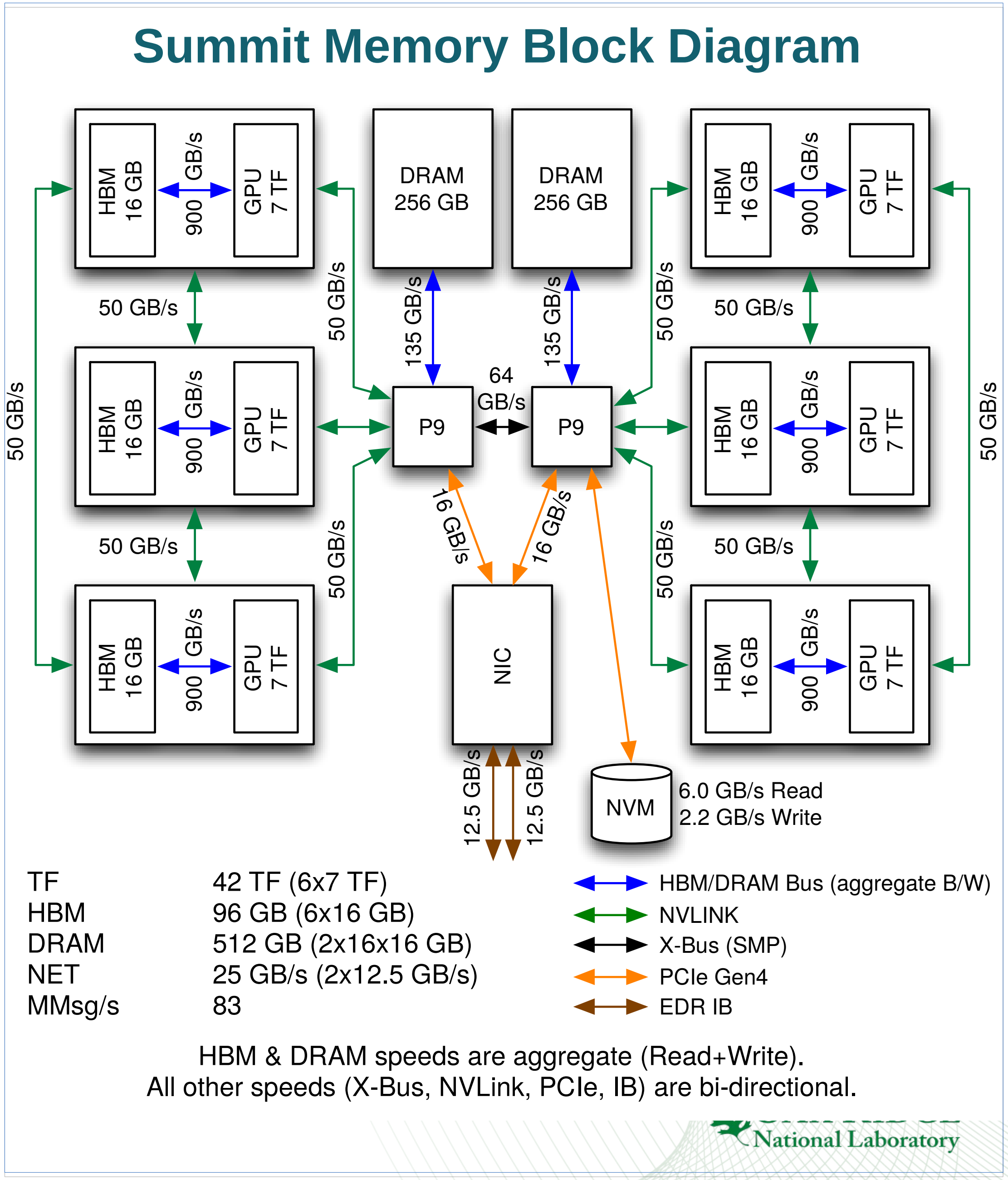
- Already creating problems, e.g., SGI shared memory systems



Source: https://en.wikipedia.org/wiki/Non-uniform_memory_access#/media/File:Hwloc.png

Split types – New challenges

- Hardware architectures are becoming more complex
  - More devices
  - Not hierarchical anymore
- Unique address space
  - Paging used to move pages between devices
  - MPI_COMM_TYPE_SHARED returns the entire compute node
- Assuming a hierarchical split is therefore confusing and complex
- The key point with new architectures is the support of many difference *devices*



Split type – Proposition – Concept of device

- Designed to leverage the Session proposal
- A new MPI handle: MPI_Device

"A device is a hardware components on the compute node that can be used to perform any computation and/or communication, i.e., host a MPI process/thread. A device is identified through a unique URI set by the runtime and the process set running on the default device can be identified through http://default_device"

Note: the URI includes a compute node identifier so it is possible to check which devices are "local"

Split type – Proposition – Concept of *default* device

- Default device

  "The default device is the device hosting the operating system (e.g., Linux kernel), which enables the access to other local compute devices (e.g., accelerators). The runtime in charge of identifying the default device and can be the logical grouping on actual devices (e.g., all local processors, in opposition to accelerators)"

Split types – Proposition – Process sets and devices

- Associate MPI_COMM_TYPE_SHARED with a predefined process set named *mpi://shared_mem*
- MPI_COMM_TYPE_DEFAULT_DEVICE (URI: mpi://DEFAULT_DEV as pset name)

  “This type splits the communicator into subcommunicators, each of which grouping MPI processes running on the default device. MPI processes running on the default device can be referred through mpi://MPI_DEFAULT_DEV”

- MPI_DEVICES_GET_PSET ([IN] MPI_Session, [IN] MPI_Pset_names, [OUT] *MPI_Pset)

  “This collective operation shall returns the process set(s) involved in the group of MPI processes.”

  Note: it is possible to get the device where the MPI process is running by using MPI_HW_DEVICES_GET and a group with only mpi://SELF

Split types – Proposition – Process sets and devices (2)

- MPI_PSET_GET_DEVICES ([IN] MPI_Session, [IN] MPI_PSET, [OUT] MPI_DEVICE[])

“This function shall return the set of devices where the MPI process of the process set are running.”

- MPI_HW_NEIGHBOR_DEVICES ([IN] MPI_Session, [IN] MPI_DEVICE, [OUT] MPI_DEVICE[])

“This functions shall return the list of devices that are directly linked (hardware link) with the specified device. If no device is directly linked to it, mpi://NULL shall be returned and the function shall return MPI_SUCCESS.”

Note: can be used to find neighbors on the network, especially if the runtime can use Netloc

Split types – Proposition – Split for hierarchical devices

- MPI_COMM_HW_SPLIT ([IN] MPI_COMM, [OUT] MPI_COMM)

“This function shall split the communicator into subcommunicators, each of which based on a shared resource (e.g., cache) of a single device. If the communicator spawns multiple devices or if the device is not hierarchical, the function shall fail”

Note: need to precisely describe what happens in the multiple device case

Example – Optimization of Cartesian topologies

- Get all local CPUs to later do a Cartesian optimization based on Rolph’s proposal:

```
MPI_Pset *mpi_default_pset;
MPI_Comm dev_comm, *subcomms;

MPI_HW_DEVICES_GET(“http://default_device”, mpi_default_dev_pset);
MPI_Group_from_session_pset (my_session, mpi_default_dev_pset, &default_dev_gp);
MPI_Comm_create_from_group (default_dev_gp, “default_dev_gp”, MPI_INFO_NULL, MPI_ERRORS_RETURN, &dev_comm);
MPI_COMM_HW_SPLIT (dev_comm, &subcomms);
(continue with Cartesian optimizations…)
```

Example – Network neighbors

```
MPI_Session my_session;
MPI_Device *devs, *neighbor_devs;
MPI_Group compute_node_gp, net_neighbors_group;


MPI_Session_init (&flags_in, MPI_INFO_NULL, MPI_ERRORS_RETURN, &my_session);
MPI_PSET_GET_DEVICES (mysession, “mpi://shared_mem”, &devs);
MPI_HW_NEIGHBOR_DEVICES (mysession, devs, &neighbor_devs);
MPI_DEVICES_GET_PSET (mysession, neighbor_devs, &pset_net_neighbors);
MPI_Group_from_session_pset (mysession, pset_net_neighbors, &net_neighbors_group);
MPI_Comm_create_from_group (net_neighbors_group, “network_neighbors”, MPI_INFO_NULL,
MPI_ERRORS_RETURN, &net_neighbors_comm);
```

Conclusion

- By adding the concept of device
  - We can do hierarchical splits when it makes sense
  - Support architectures that are not hierarchical
- Minimum extension to the standard
- Backward compatible (if we keep MPI_COMM_TYPE_SHARED)
- Similar to device concept from OpenMP, which will ease the development of MPI+X applications