MODULAR SUPERCOMPUTING ARCHITECTURE

A TUTORIAL

29.03.2022 I UJJWAL SINHA

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

ACKNOWLEDGEMENTS

- **Norbert Eicker**
- **Jacopo De Amicis**
- **Jochen Kreutz**
- **Andreas Herten**
- **Estela Suarez**
- **DEEP-SEA WP1 team**

OUTLINE

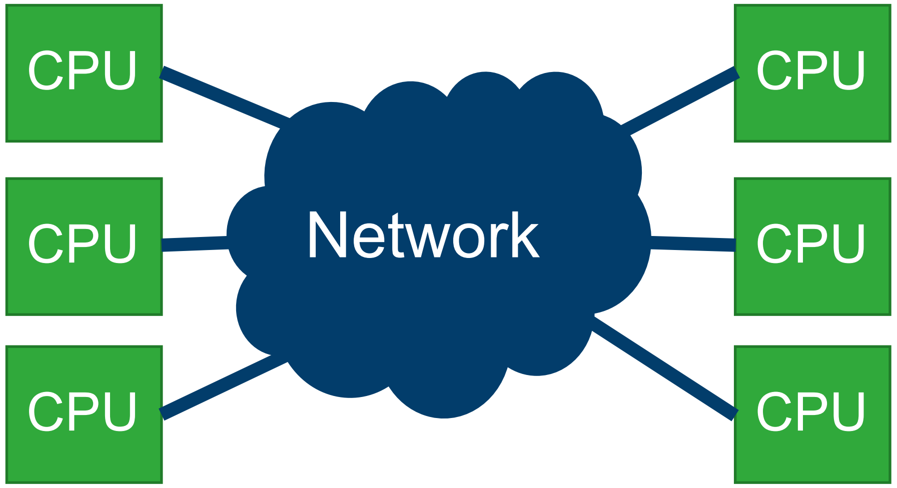
- **Supercomputing Architectures**
  - Importance of supercomputing architectures
  - Homogeneous architectures
  - Heterogeneous monolithic architectures
  - Heterogeneous modular architectures
  - Modular Supercomputing Architecture (MSA)
- **Programming model**
  - Inter-module MPI offloading
  - Parent-Child programming model
  - Submitting heterogenous cross-module jobs
- **Summary**
- **Exercises:**

https://gitlab.jsc.fz-juelich.de/sinha3/modular-supercomputing-architecture-a-tutorial-for-beginners

WHY SUPERCOMPUTING ARCHITECTURES ARE IMPORTANT?

- Reproducing scientific experiments require large scale simulations
- Codes employ diverse algorithms to generate and analyse data
- Results can be quickly obtained if the hardware fits the applications
- Additional constraints related to cost, power consumption, maintenance, and programmability

SUPERCOMPUTING ARCHITECTURES: HOMOGENEOUS



Nodes comprise only CPUs

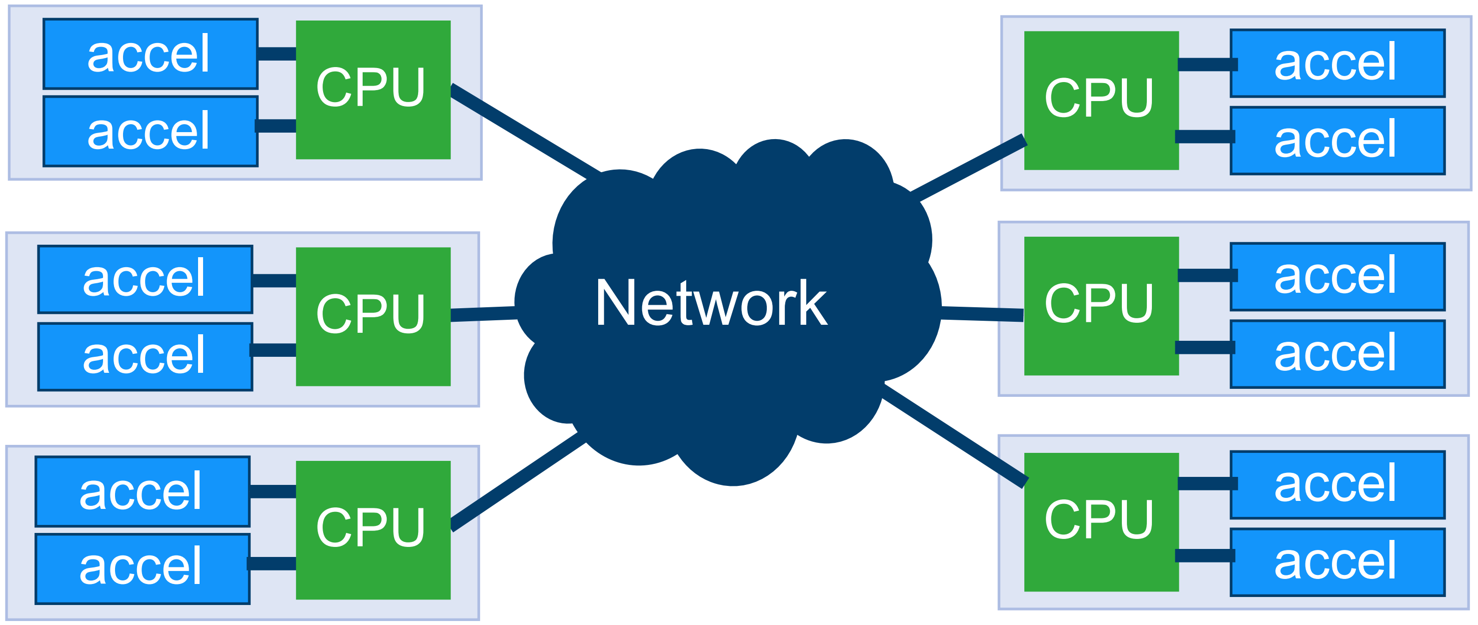
Pros:

- Easy to use
- Very flexible

Cons:

- Power hungry

SUPERCOMPUTING ARCHITECTURES: HETEROGENEOUS MONOLITHIC



- Nodes contain CPUs and accelerators (e.g. GPUs)
- All nodes are equal → Monolithic

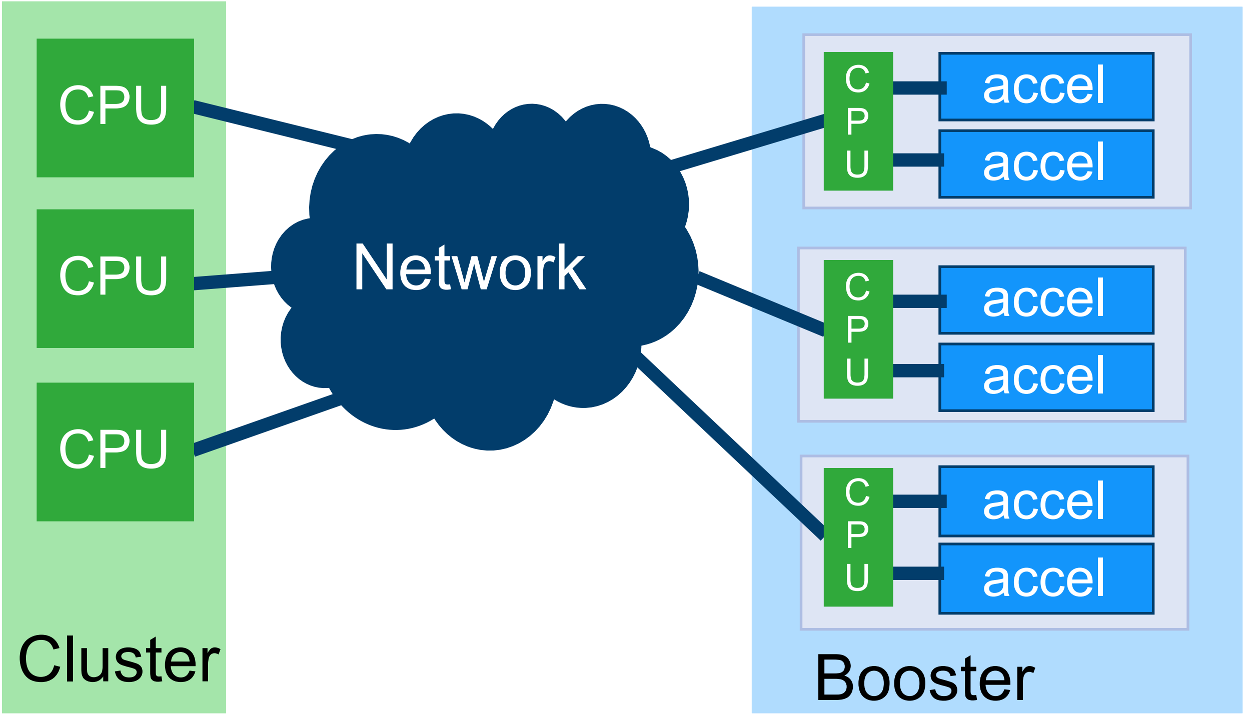
Pros:

- Energy efficient
- Easy management

Cons:

- Static assignment of accelerators to CPUs
- Difficulty to efficiently share resources

SUPERCOMPUTING ARCHITECTURES: HETEROGENEOUS MODULAR



- All nodes within one module are equal
- Different modules have different configurations → Modular

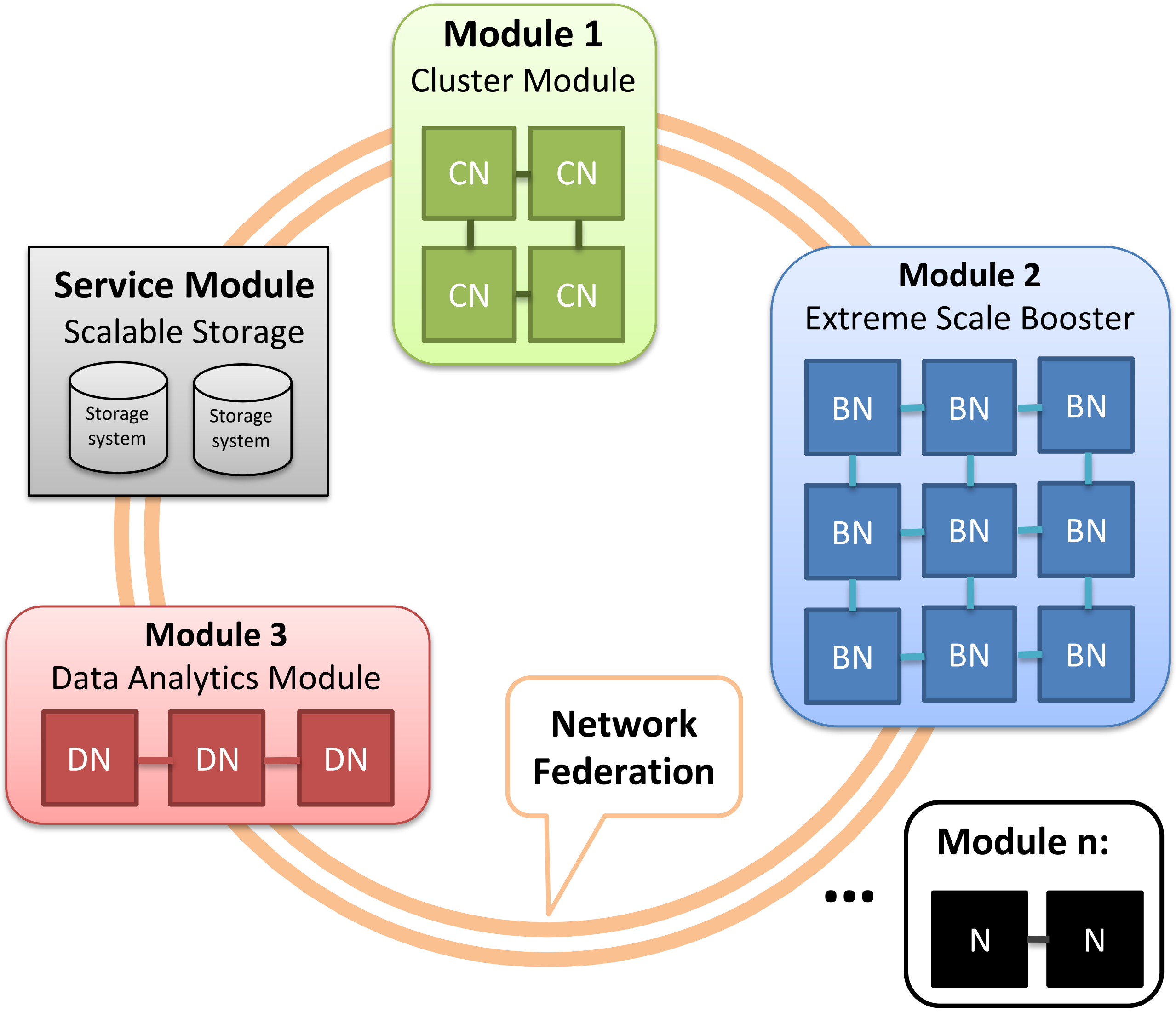
Pros:

- Energy efficient
- Better scalability
- High flexibility
- Dynamic resource assignment

Cons:

- Complexity

MODULAR SUPERCOMPUTING ARCHITECTURE (MSA)



Cluster Module:

Run codes requiring high single thread performance

Extreme Scale Booster:

Run highly scalable codes

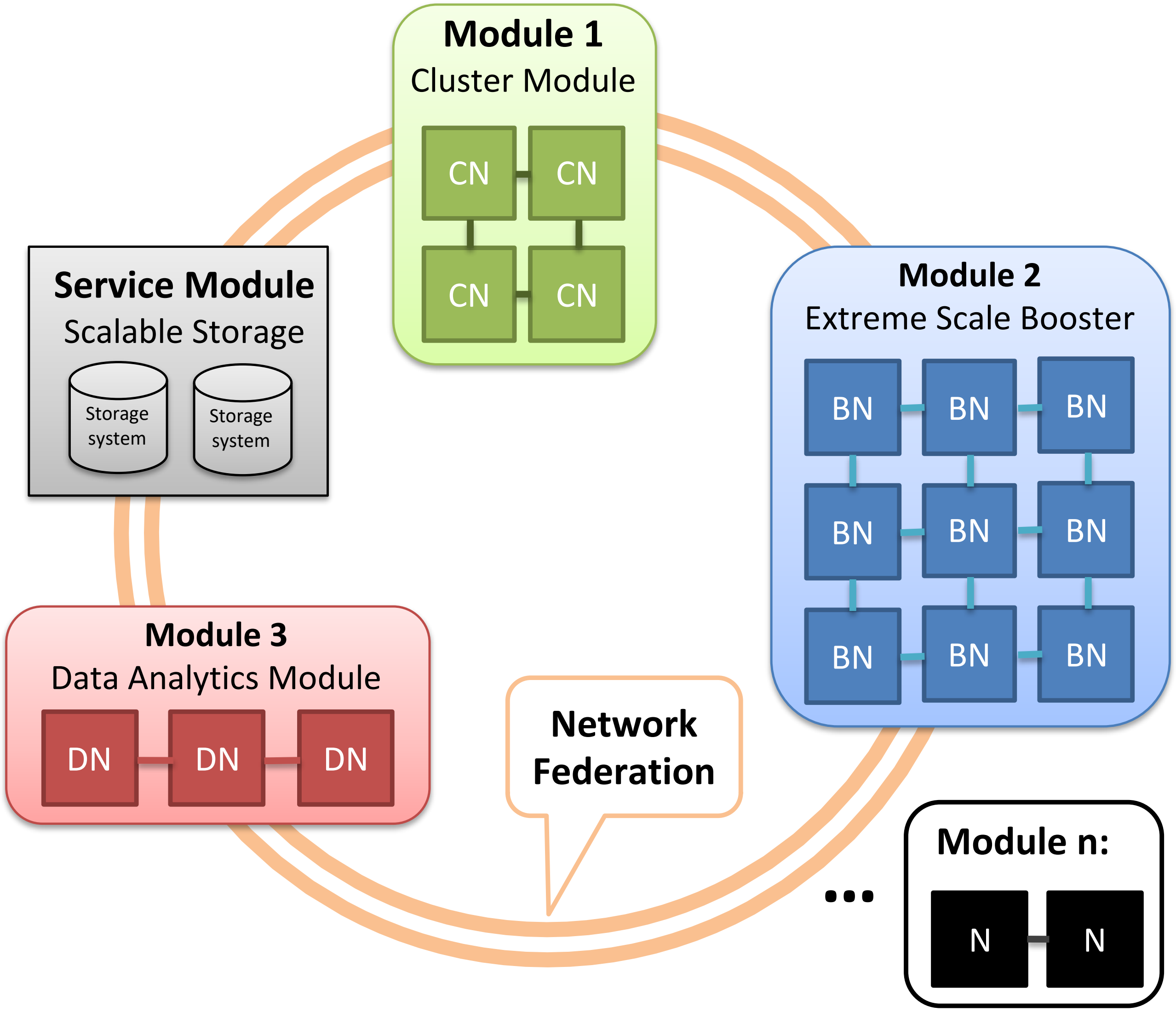
Data Analytics Module:

Support High Performance Data Analysis (HPDA) requirements

Service Module:

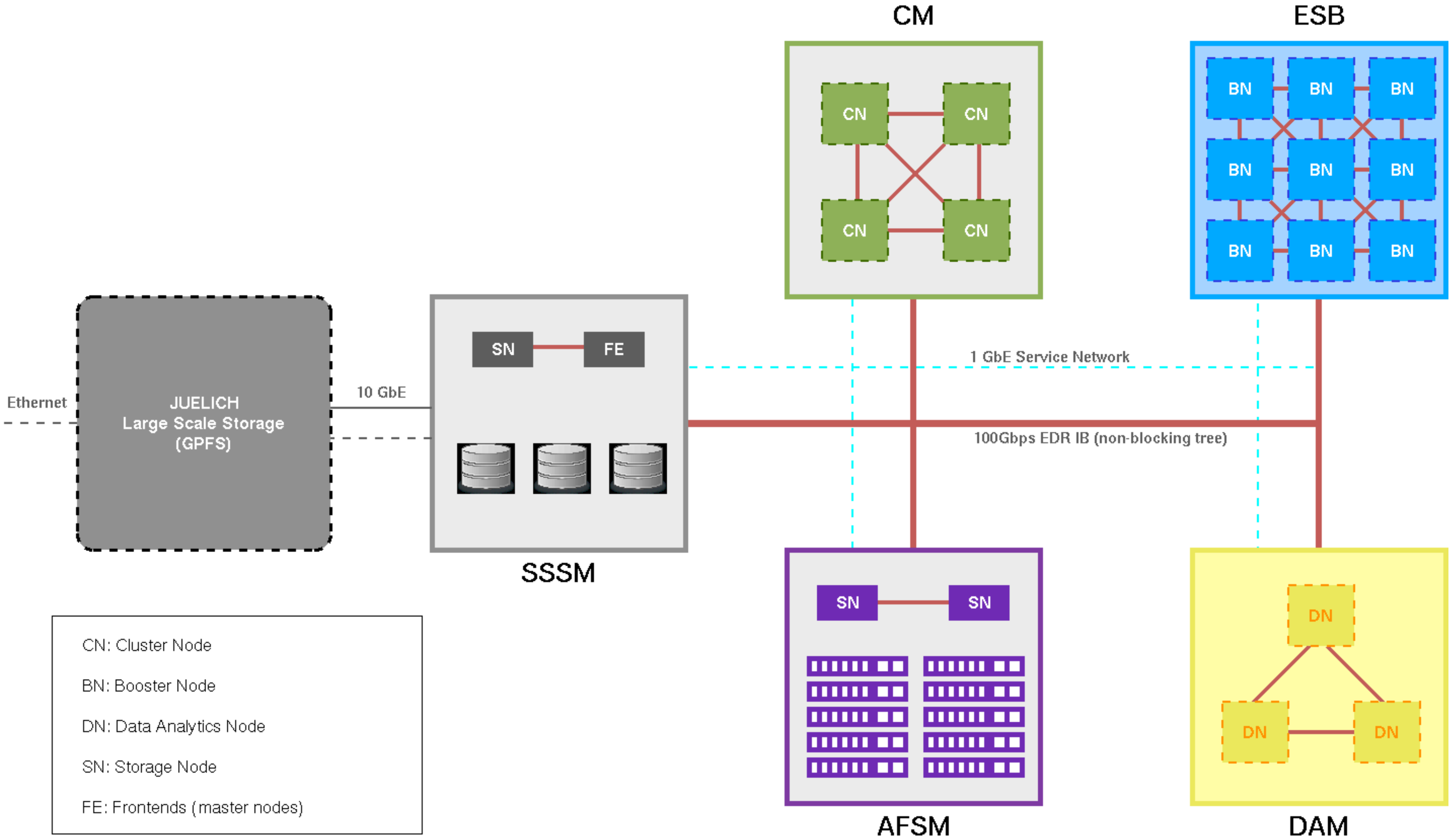
Provide the prototype with required scalable storage

MODULAR SUPERCOMPUTING ARCHITECTURE (MSA)



- Enable codes to take advantage of highly scalable systems and improve energy efficiency and scalability
- No constraints on the combination of nodes
- Resources are reserved and allocated dynamically
- Booster is a massively parallel system on its own such that it can fit highly scalable codes
- Applications can simultaneously run on cluster as booster

DEEP-EST MODULAR SUPERCOMPUTER (PROTOTYPE SYSTEM)



CN: Cluster Node

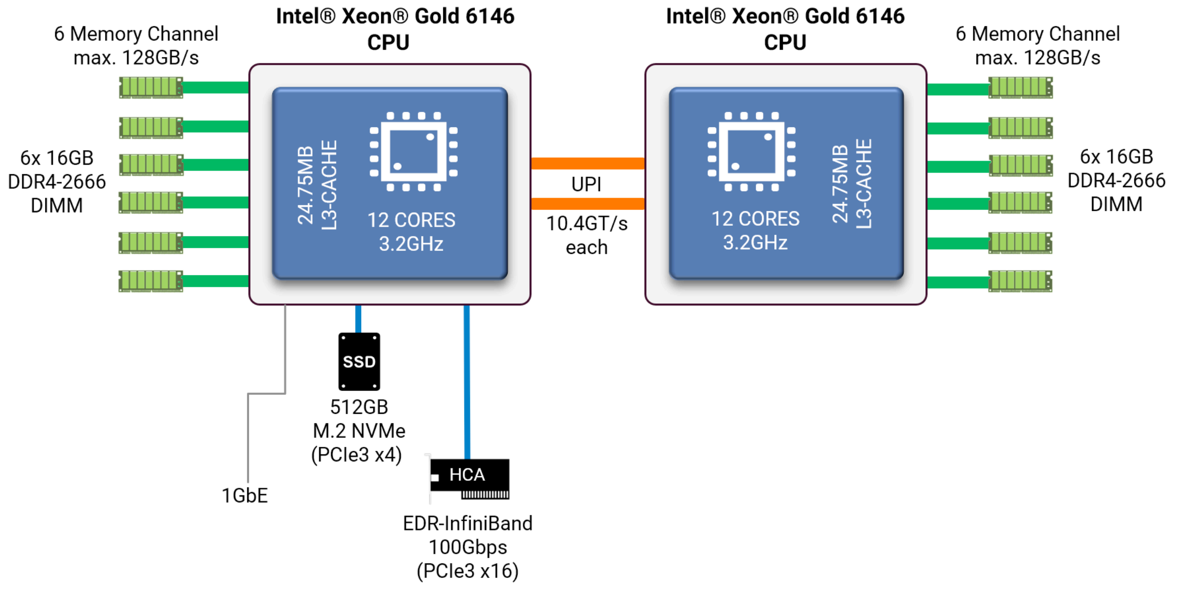
BN: Booster Node

DN: Data Analytics Node

SN: Storage Node

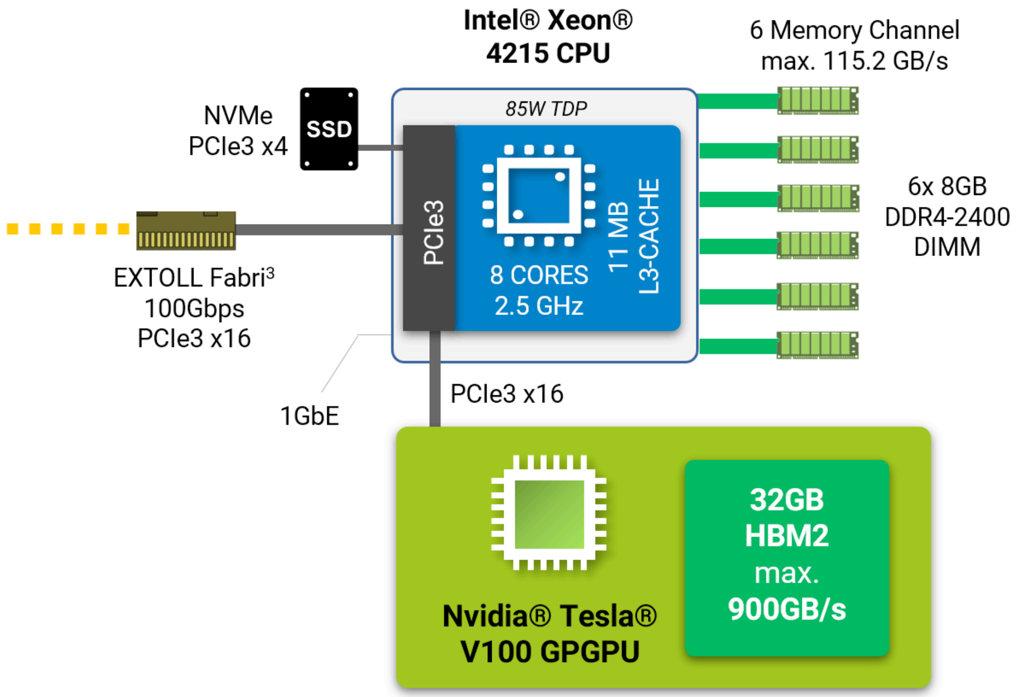
FE: Frontends (master nodes)

DEEP-EST MODULAR SUPERCOMPUTER (PROTOTYPE SYSTEM)



Cluster [50 nodes]: dp-cn[01-50]:

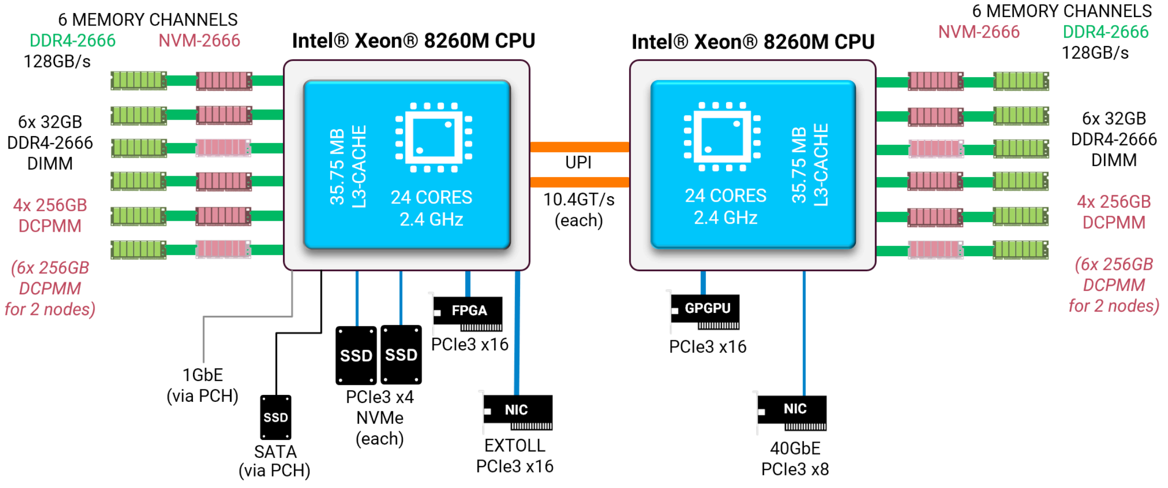
- 2 Intel Xeon 'Skylake' Gold 6146 (12 cores (24 threads), 3.2GHz)
- 192 GB RAM
- 1 x 400GB NVMe SSD
- network: InfiniBand EDR (100 Gb/s)



Extreme Scale Booster [75 nodes]: dp-esb[01-75]

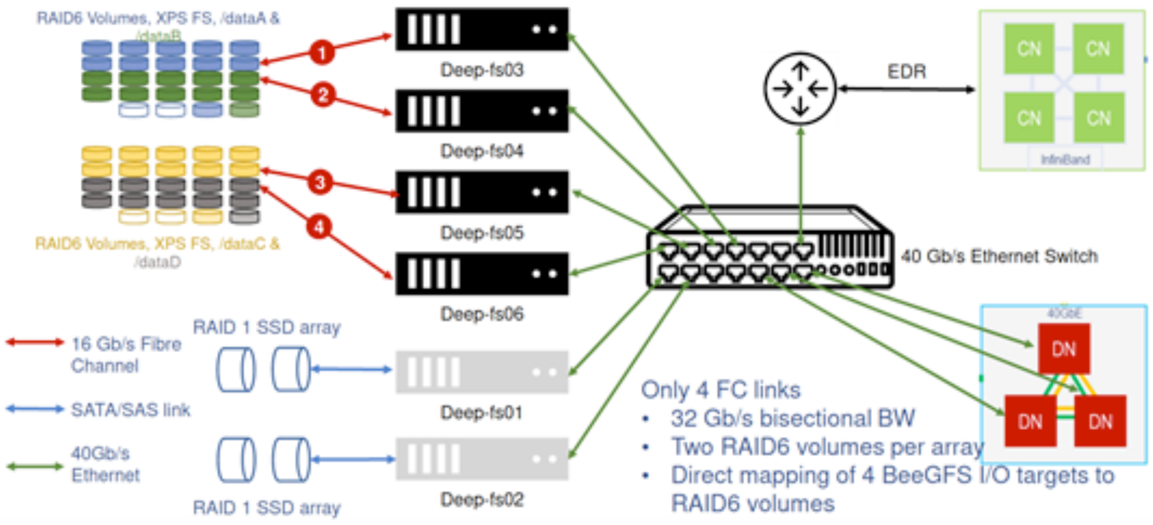
- 1 x Intel Xeon 'Cascade Lake' Silver 4215 CPU @ 2.50GHz
- 1 x Nvidia V100 Tesla GPU (32 GB HBM2)
- 48 GB RAM
- 1 x 512 GB SSD
- network: IB EDR (100 Gb/s)

DEEP-EST MODULAR SUPERCOMPUTER (PROTOTYPE SYSTEM)



Data Analytics Module [16 nodes]: dp-dam[01-16]

- 2 x Intel Xeon 'Cascade Lake' Platinum 8260M CPU @ 2.40GHz
- dp-dam[01-08]: 1 x Nvidia V100 Tesla GPU (32 GB HBM2)
- dp-dam[09-12]: 2 x Nvidia V100 Tesla GPU (32 GB HBM2)
- dp-dam[13-16]: 2 x Intel STRATIX10 FPGA (32 GB DDR4)
- 384 GB RAM + 3 TB non-volatile memory
- 2 x 1.5 TB Intel Optane SSD (1 for local scratch, 1 for BeeOND)
- 1 x 240 GB SSD (for boot and OS)
- network: IB EDR (100 Gb/s)



Only 4 FC links

- 32 Gb/s bisectional BW
- Two RAID6 volumes per array
- Direct mapping of 4 BeeGFS I/O targets to RAID6 volumes

- SSSM [6 servers]: dp-fs[01-06]:
  - 2 Intel Xeon Silver 4114 (20 cores, 2.2 GHz)
  - 96 GB RAM
  - 2 x 240 GB SSD
  - (additional 2 x 480 GB SSD in dp-fs[01-02] for metadata)
  - network: IB EDR (100 Gb/s)
- SSSM [2 EUROstor ES-6600 RAID enclosures]: dp-raid[01-02]:
  - 24 x 8 TB SAS Nearline
  - 2 x 16 Gb FC connector

PROGRAMMING MODEL

- The Dynamical Exascale Entry Platform (DEEP) projects have contributed significantly towards the development of a programming environment that maximally reduces the effort of porting applications to the new platform

- De-facto standard HPC programming model:
  - MPI
  - MPI + OpenMP
  - MPI + OpenAcc/CUDA
  - MPI + OpenMPI + OpenAcc/CUDA

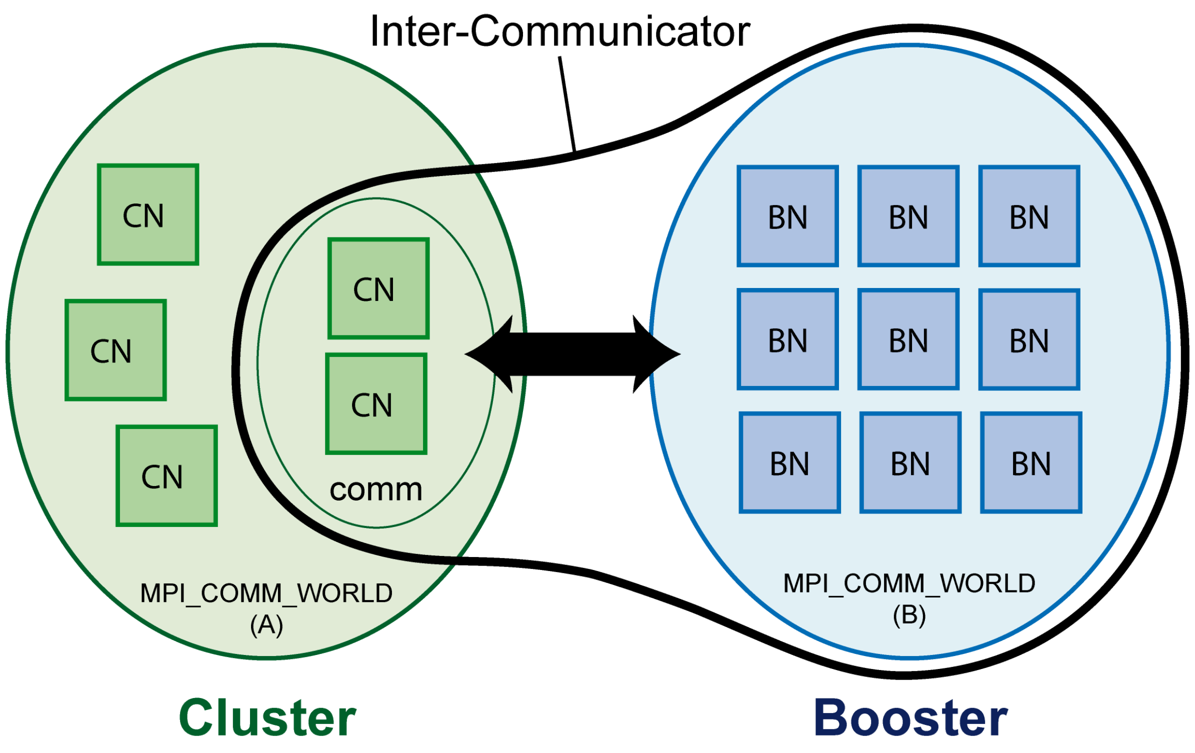
INTER-MODULE MPI OFFLOADING

- To allow the different parts of an application to run simultaneously on the cluster and booster and establish a communication between them, MPI-2 has a “Spawn” functionality called `MPI_Comm_spawn`

```
MPI_Comm_spawn(“./executable“, MPI_ARGV_NULL, maxprocs, MPI_INFO_NULL, int root,
MPI_COMM comm, MPI_Comm *intercomm, int array_of_errcodes[])
```

- `MPI_Comm_spawn` tries to start “maxprocs” identical copies of the MPI program specified by “./executable”, establishing communication with them and returning an inter-communicator.

INTER-MODULE MPI OFFLOADING

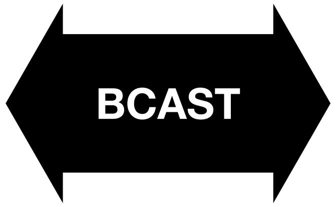


- Parents call `MPI_Comm_spawn` and create and inter-communicator between parents and children
- Children processes call their own `MPI_Init` and create `MPI_Comm_world`
- Children call `MPI_comm_get_parent()` to obtain the inter-communicator

PARENT-CHILD PROGRAMMING MODEL

Parent

```
int main (int argc, char *argv[])
{
   int  numtasks, taskid, len;
   char hostname[MPI_MAX_PROCESSOR_NAME];

// Child communicator
   MPI_Comm child;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Get_processor_name(hostname, &len);

// spawn errors
   int spawnError[numtasks];

// Spawn num_children child process for each process
   MPI_Comm_spawn("./slave", MPI_ARGV_NULL,
num_children, MPI_INFO_NULL, 0, MPI_COMM_SELF, &child,
spawnError);

   int myid;
   int message1, message2;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   MPI_Bcast(&myid,1,MPI_INT,MPI_ROOT,child);
   MPI_Finalize();

}
```



Child

```
int main (int argc, char *argv[])
{
   int  numtasks, taskid, len, size;
   char hostname[MPI_MAX_PROCESSOR_NAME];

// Obtain an intercommunicator to the parent MPI job
   MPI_Comm parent;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Get_processor_name(hostname, &len);
   MPI_Comm_get_parent(&parent);

// Get child rank
   int myid;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   if (parent != MPI_COMM_NULL) {
     int parent_id;

     MPI_Bcast(&parent_id, 1, MPI_INT,0, parent);

     printf("Child %d of parent %d \n", myid, parent_id);

   }

   MPI_Finalize();

}
```

PARENT-CHILD PROGRAMMING MODEL

Parent

```
int main (int argc, char *argv[])
{
   int   i, numtasks, taskid, len;
   char hostname[MPI_MAX_PROCESSOR_NAME];
   int work[4] = {1,2,3,4};
   int data[4] = {10,20,30,40};
   MPI_Status status;

// Child communicator
   MPI_Comm child;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Get_processor_name(hostname, &len);

// spawn errors
   int spawnError[numtasks];

// Spawn num_children child process for each process
   MPI_Comm_spawn("./slave", MPI_ARGV_NULL, num_children,
MPI_INFO_NULL, 0, MPI_COMM_SELF, &child, spawnError);

   int myid;
   int message1, message2;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   MPI_Send(work + myid, 1, MPI_INT, 0, 1, child);
   MPI_Send(data + myid, 1, MPI_INT, 1, 1, child);

   MPI_Recv(&message1, 1, MPI_INT, 0, 1, child, &status);
   MPI_Recv(&message2, 1, MPI_INT, 1, 1, child, &status);

   printf("The first message received by task %d is %d \n", myid,
message1);
   printf("The second message received by task %d is %d \n", myid,
message2);

   MPI_Finalize();

}
```

SEND/RECV

Child

```
int main (int argc, char *argv[])
{
   int   numtasks, taskid, len, size;
   char hostname[MPI_MAX_PROCESSOR_NAME];
   MPI_Status status;

// Obtain an intercommunicator to the parent MPI job
   MPI_Comm parent;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Get_processor_name(hostname, &len);
   MPI_Comm_get_parent(&parent);

// Get child rank
   int myid;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
// Check if this process is a spawned one and if so get parent CPU rank
   if (parent != MPI_COMM_NULL) {
     int parent_id;
     int message, result;

     MPI_Recv(&message, 1, MPI_INT, 0, 1, parent, &status);

     result = message * 2;

     MPI_Send(&result, 1, MPI_INT, 0, 1, parent);

   }

   MPI_Finalize();

}
```

PARENT-CHILD PROGRAMMING MODEL: CLUSTER-BOOSTER

Parent

```
   int* restrict const A =
(int*)malloc(array_size*sizeof(int));
   int* restrict const B =
(int*)malloc(array_size*sizeof(int));

// spawn errors
   int spawnError[numtasks];

// Spawn num_children child process for each process
   MPI_Comm_spawn("./slave", MPI_ARGV_NULL, num_children,
MPI_INFO_NULL, 0, MPI_COMM_SELF, &child, spawnError);

   int myid;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   for (i=0; i<array_size; i++){
           A[i] = i;
   }
   MPI_Send(A, array_size, MPI_INT, 0, 1, child);

   MPI_Recv(B, array_size, MPI_INT, 0, 1, child, &status);

   for (i=0; i<array_size; i++){
           printf("B[%d]=%d\n", i, B[i]);
   }

   MPI_Finalize();

   free(A);
   free(B);
   return 0;
```

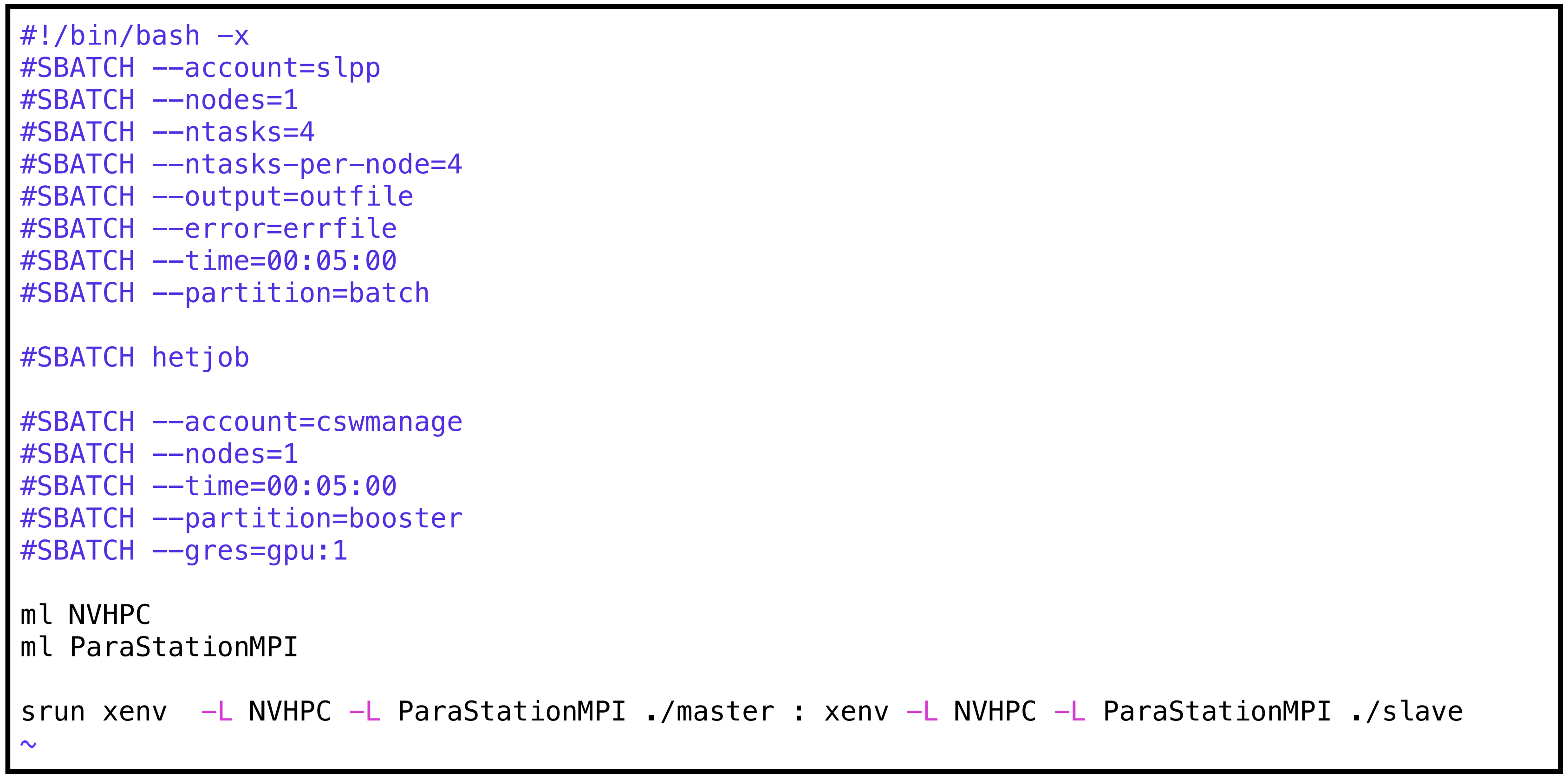
Child

```
   int* restrict const A =
(int*)malloc(array_size*sizeof(int));
   int* restrict const B =
(int*)malloc(array_size*sizeof(int));

// Get child rank
   int myid;

   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
// Check if this process is a spawned one and if so get
parent CPU rank
   if (parent != MPI_COMM_NULL) {
//     int parent_id;
//     int message, result;

     MPI_Recv(A, array_size, MPI_INT, 0, 1, parent,
&status);

#pragma acc init
#pragma acc parallel loop

     for (i=0; i<array_size; i++){
             B[i] = A[i] * A[i];
     }

     MPI_Send(B, array_size, MPI_INT, 0, 1, parent);

   }

   MPI_Finalize();

   free(A);
   free(B);
   return 0;
```

GPU part

HETEROGENEOUS AND CROSS-MODULE JOBS

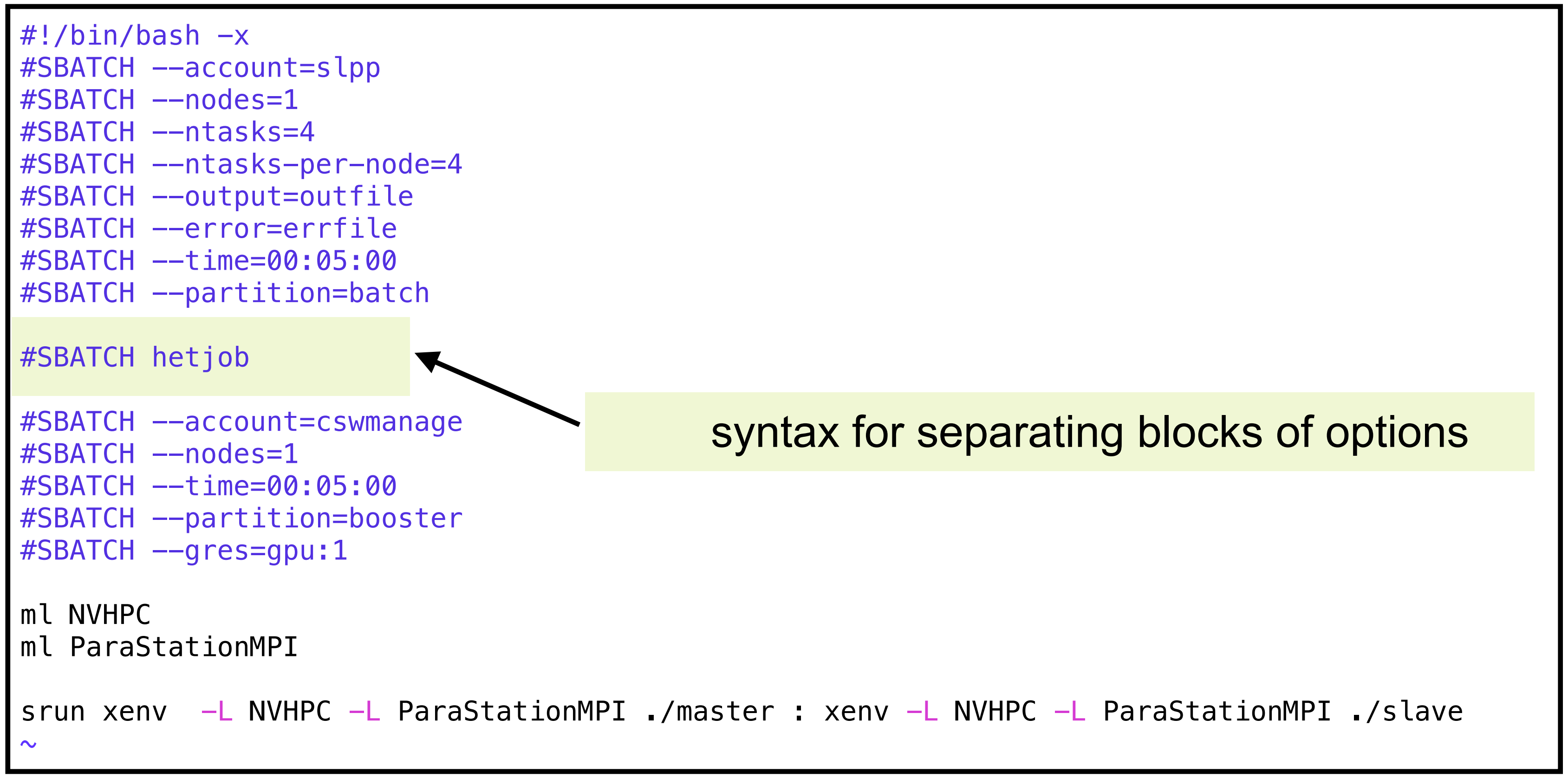
- Slurm 17.11 supports heterogenous jobs
- A heterogenous job consists of several job components, all of which have individual job options



```
#!/bin/bash -x
#SBATCH --account=slpp
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --output=outfile
#SBATCH --error=errfile
#SBATCH --time=00:05:00
#SBATCH --partition=batch

#SBATCH hetjob

#SBATCH --account=cswmanage
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:1

ml NVHPC
ml ParaStationMPI

srun xenv  -L NVHPC -L ParaStationMPI ./master : xenv -L NVHPC -L ParaStationMPI ./slave
~
```

HETEROGENEOUS AND CROSS-MODULE JOBS

- Slurm 17.11 supports heterogenous jobs
- A heterogenous job consists of several job components, all of which have individual job options

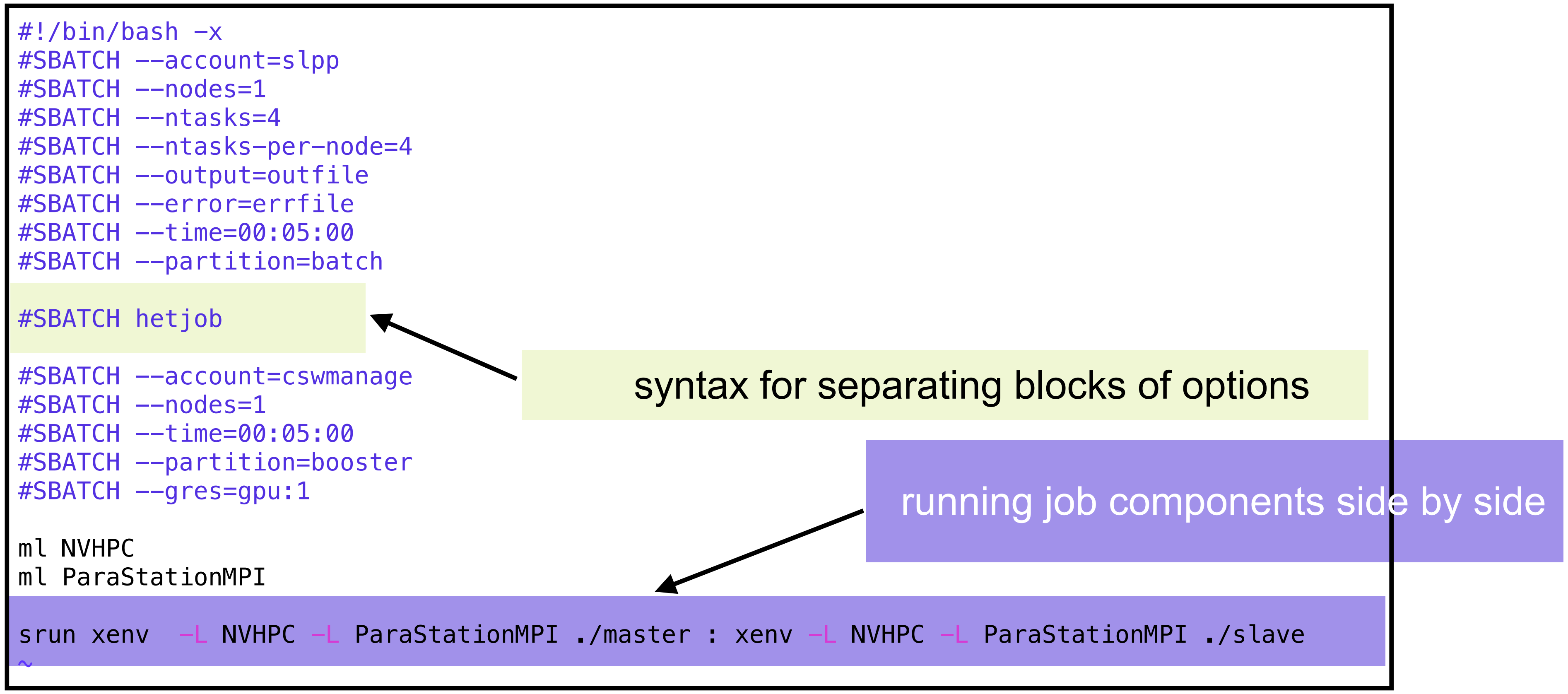


```
#!/bin/bash -x
#SBATCH --account=slpp
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --output=outfile
#SBATCH --error=errfile
#SBATCH --time=00:05:00
#SBATCH --partition=batch

#SBATCH hetjob

#SBATCH --account=cswmanage
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:1

ml NVHPC
ml ParaStationMPI

srun xenv  -L NVHPC -L ParaStationMPI ./master : xenv -L NVHPC -L ParaStationMPI ./slave
~
```

syntax for separating blocks of options

HETEROGENEOUS AND CROSS-MODULE JOBS

- Slurm 17.11 supports heterogenous jobs
- A heterogenous job consists of several job components, all of which have individual job options



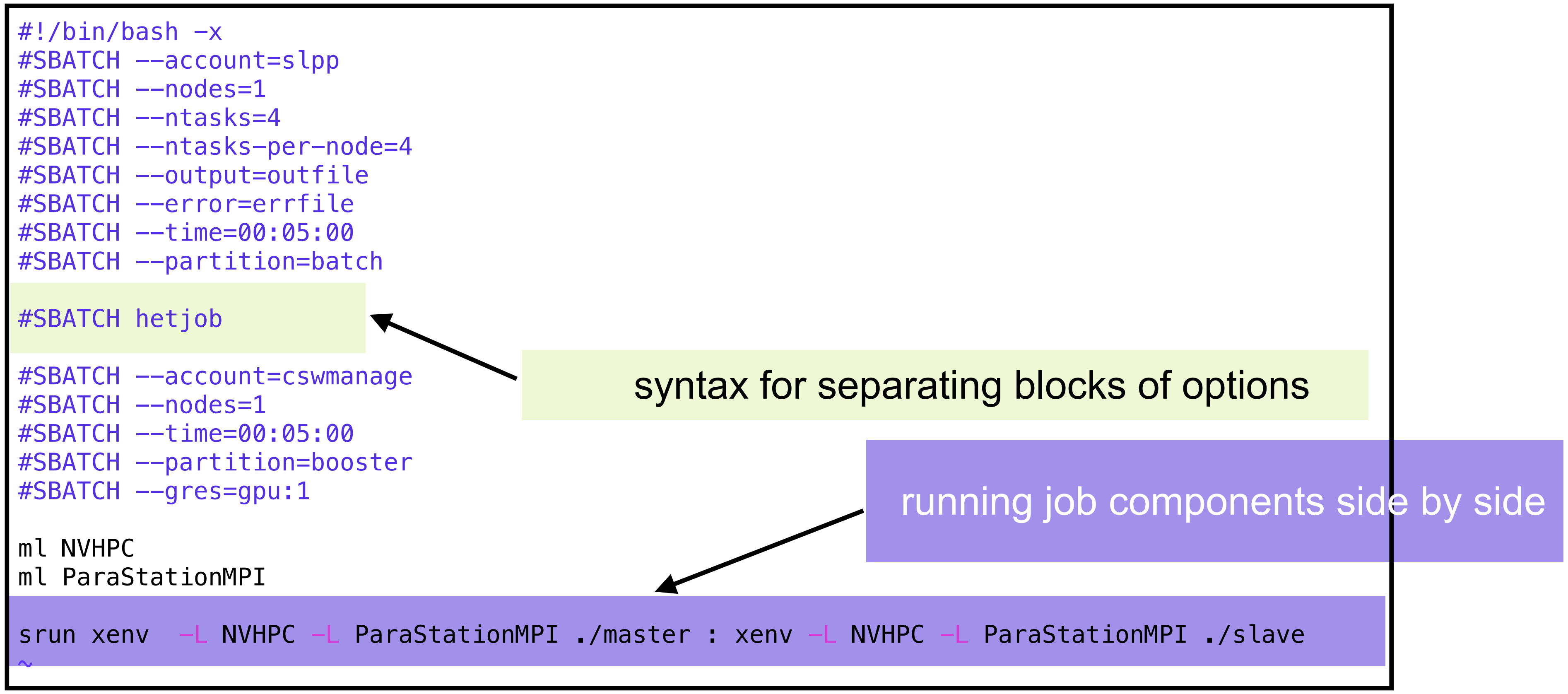
```
#!/bin/bash -x
#SBATCH --account=slpp
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --output=outfile
#SBATCH --error=errfile
#SBATCH --time=00:05:00
#SBATCH --partition=batch

#SBATCH hetjob

#SBATCH --account=cswmanage
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:1

ml NVHPC
ml ParaStationMPI

srun xenv  -L NVHPC -L ParaStationMPI ./master : xenv -L NVHPC -L ParaStationMPI ./slave
~
```

syntax for separating blocks of options

running job components side by side

HETEROGENEOUS AND CROSS-MODULE JOBS

- Slurm 17.11 supports heterogenous jobs
- A heterogenous job consists of several job components, all of which have individual job options



```
#!/bin/bash -x
#SBATCH --account=slpp
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --output=outfile
#SBATCH --error=errfile
#SBATCH --time=00:05:00
#SBATCH --partition=batch

#SBATCH hetjob

#SBATCH --account=cswmanage
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:1

ml NVHPC
ml ParaStationMPI

srun xenv  -L NVHPC -L ParaStationMPI ./master : xenv -L NVHPC -L ParaStationMPI ./slave
~
```

syntax for separating blocks of options

running job components side by side

`xenv` is a node-local tool which knows the correct software stack for the nodes and where to locate the appropriate modules for it.

SUMMARY

- The MSA generalises the idea of segregating heterogeneous resources into individual, interconnected compute modules
- MSA has advantages in terms of flexibility and is suitable for diverse application requirements
- Applications with partially scalable parts can run the scalable parts on the booster and the less scalable parts can profit from the cluster.
- Communication between the cluster and the booster can be established with an inter-communicator created using `MPI_Comm_spawn`
- SLURM allows to run heterogenous jobs simultaneously on different modules