Parallel Computing using MPI & Open MP

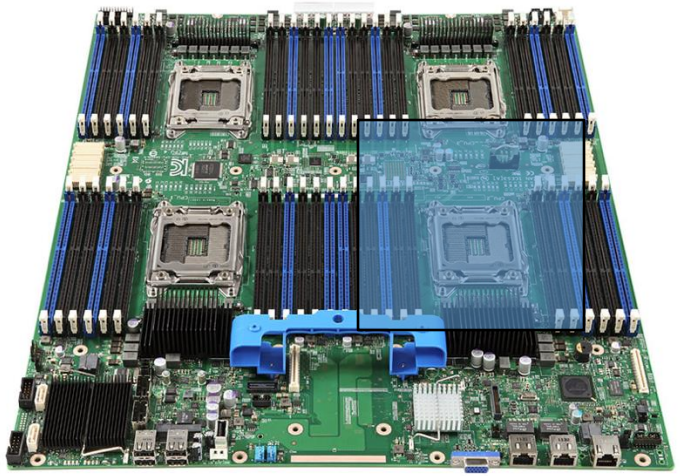
Mahidhar Tatineni
HPC and Data Science Summer Institute
San Diego Supercomputer Center

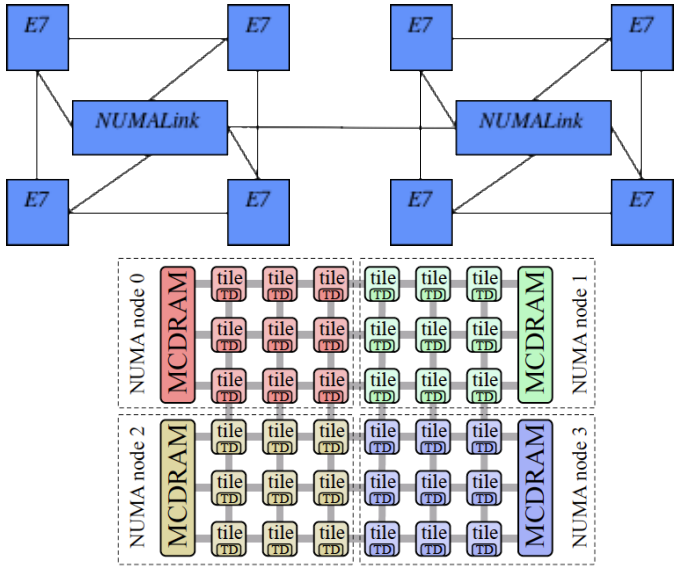
August 5, 2025

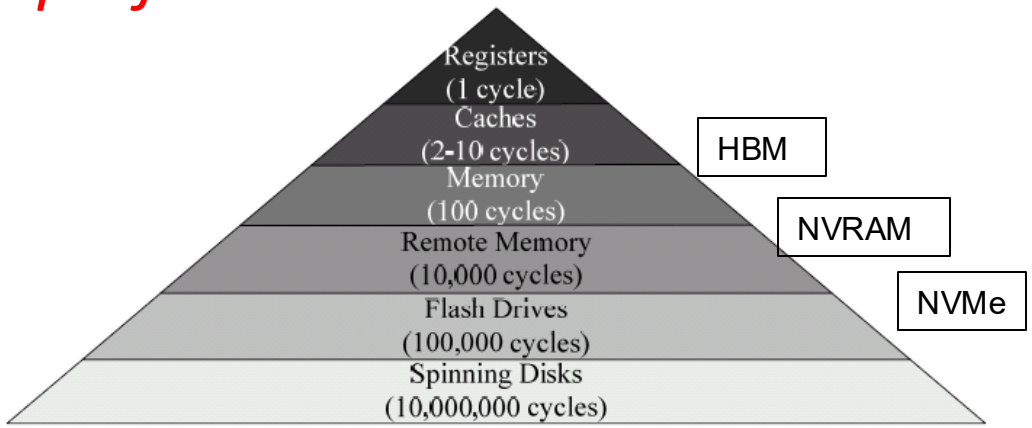
Ref: Past SI tutorials, LLNL tutorials

Current Supercomputer Architectures

- Multi-socket server nodes
  - NUMA
  - Accelerators
- High performance interconnect
  - e.g. InfiniBand
- *Scalable parallel approach needed to achieve performance*

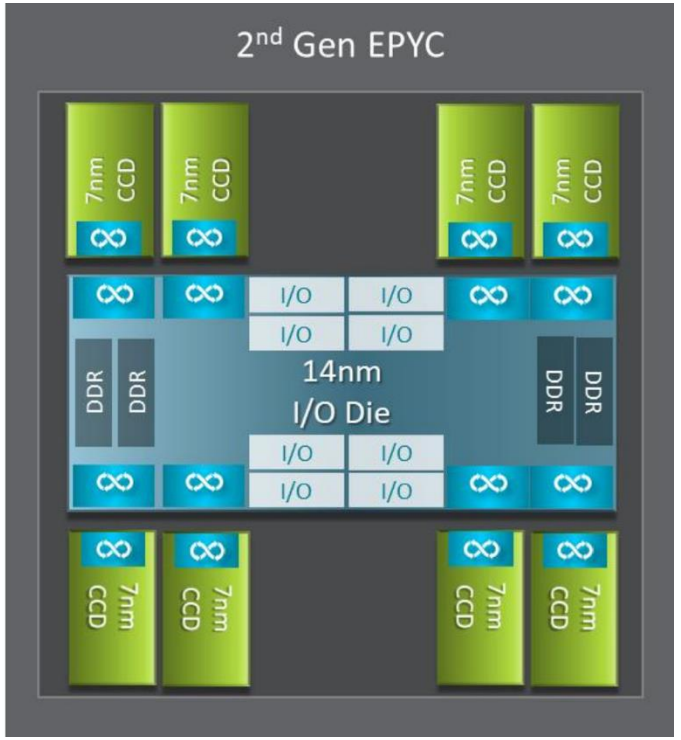






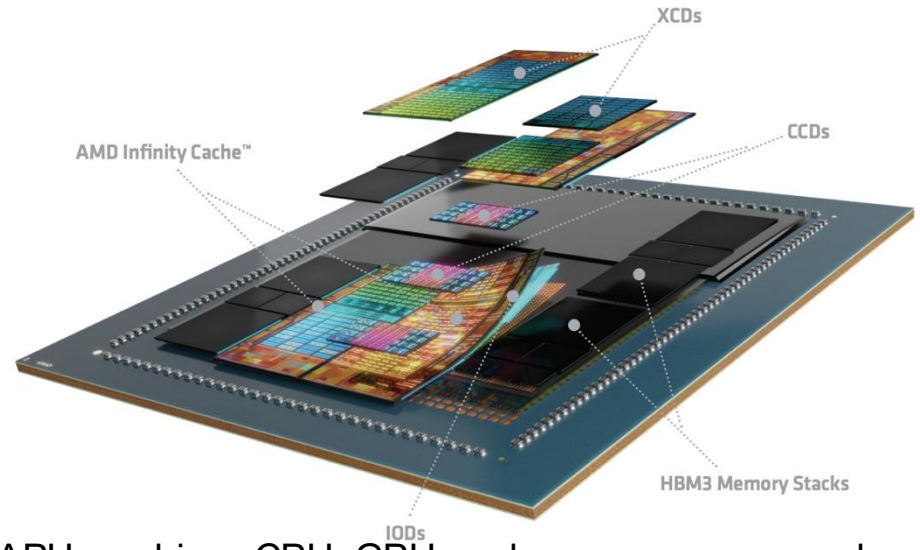
AMD EPYC 7742 Processor Architecture

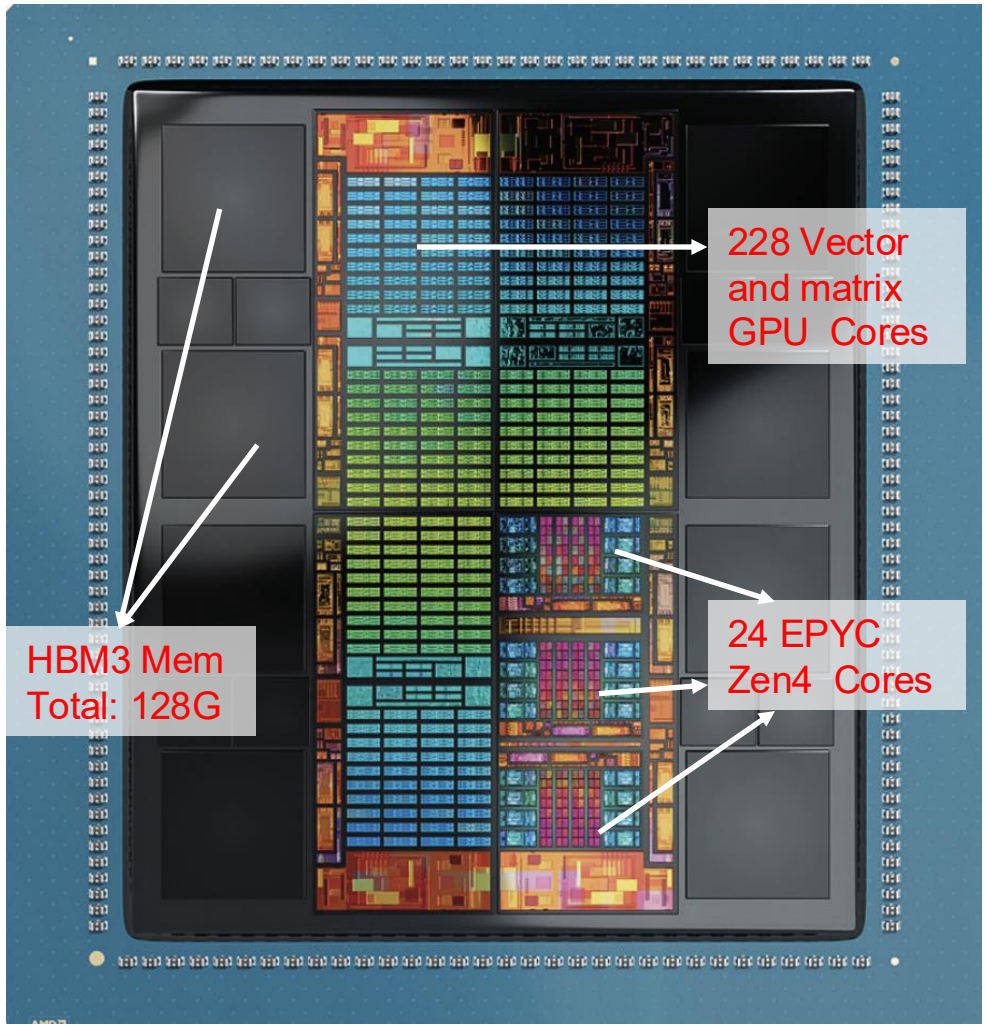
- 8 Core Complex Dies (CCDs).
- CCDs connect to memory, I/O, and each other through the I/O Die.
- 8 memory channels per socket.
- DDR4 memory at 3200MHz.
- PCI Gen4, up to 128 lanes of high speed I/O.
- Memory and I/O can be abstracted into separate quadrants each with 2 DIMM channels and 32 I/O lanes.



Reference: https://developer.amd.com/wp-content/resources/56827-1-0.pdf

Cosmos: AMD MI300A Architecture



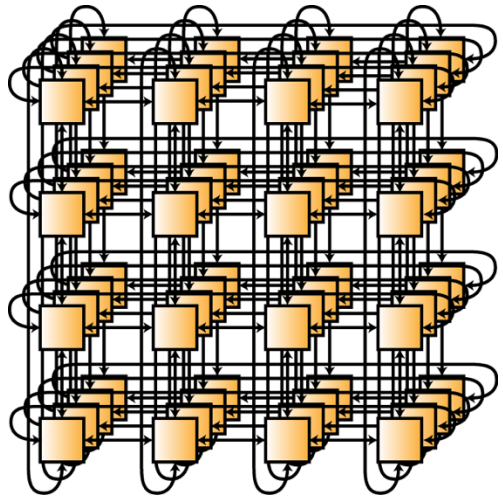


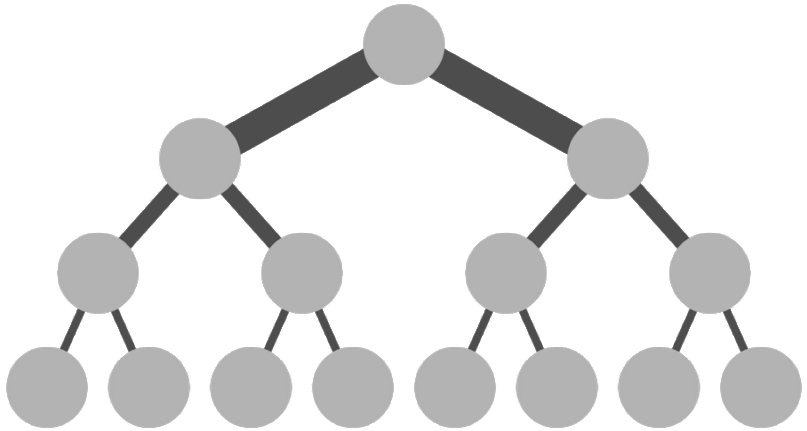
- MI300A APU combines CPU, GPU, and memory on one package. Six accelerated compute dies (XCDs) combined with 3 chiplets with 8 zen4 cores each.
- 128 GB of HBM3 memory shared coherently between CPUs and GPUs. 5.3 TB/s peak throughput.
- 256 MB AMD Infinity Cache (last level) shared between XCDs and CPUs
- Design helps eliminate need to do host/device data copies and eases code development (going towards our goal of democratizing access to accelerators!)

Reference: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf

Network Topologies

- Mesh, Torus, Hypercube
- Tree based
  - Fat-tree
  - Clos
- Dragonfly
- Metrics
  - Bandwidth
  - Diameter, Connectivity
  - Bisection bandwidth





Parallel Computing

- Executing instructions concurrently on physical resources (not time slicing)
  - Multiple tightly coupled resources (e.g. cores) collaboratively solving a single problem
- Benefits
  - Capacity
    - Memory, storage
  - Performance
    - More instructions per unit of time (FLOPS)
    - Data streaming capability
- Cost and Complexity
  - Coordinate tasks and resources
  - Use resources efficiently

Memory, Communication, and Execution Models

- Shared
  - Communication model: shared memory
- Distributed
  - Communication model: exchange messages
- Execution Models
  - Fork-Join (e.g. Thread Level Parallelism)
  - Single Program Multiple Data (SPMD)
- Parallelism enabled by decomposing work
  - Tasks can be executed concurrently
  - Some tasks can have dependencies

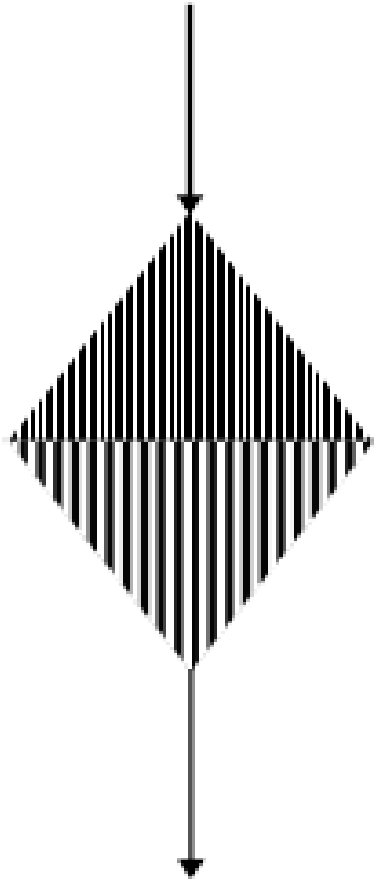
What is OpenMP?

- **High level parallelism abstraction based on thread**
  - Easy to use
  - Suitable to an incremental approach
- **A specification and evolving standard**
  - “a portable, scalable model … for developing portable parallel programs”
  - http://openmp.org
  - GNU, Intel, PGI, etc.
- **A set of**
  - Compiler directives
  - Library routines
  - Environment variables
  - Supports C/C++ and Fortran

```
#pragma omp parallel
{
  ....
}
```

OpenMP Models

- Fork/Join Execution
  - Process starts single threaded (master thread)
  - Forks child threads activated in parallel regions (team)
  - The team synchronizes and threads are disbanded
    - – barrier
  - Overhead is mitigated by reusing threads
  - Master thread continues execution of serial phases
- Work decomposition
  - Programming constructs
  - Scope and compound statements
  - Declarative in loops
  - Mapping to threads can be static or dynamic
  - Barriers and synchronization automatically inserted



Compiler Directives

- Compiler directives is the main mechanism for introducing parallelism. Functionality enabled includes:
  - Spawning a parallel region
  - Diving code among threads
  - Distributing loop iterations over threads
  - Serialization of parts of the code
  - Synchronization of work
- Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

Regions, Loops, Sections etc

#pragma omp parallel *[clause[[,]clause] ...] new-line*
structured-block
clause:
if(*scalar-expression*)
num_threads(*integer-expression*)
default(**shared** | **none**)
private(*list*)
firstprivate(*list*)
shared(*list*)
copyin(*list*)
reduction(*operator*: *list*)

- #pragma omp single/master
- simd
- tasks

#pragma omp for *[clause[[,] clause] ...] new-line for-loops*
clause:
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*operator*: *list*)
schedule(*kind[, chunk_size]*)
collapse(*n*)
ordered
nowait

#pragma omp sections *[clause[[,] clause] ...] new-line*
{
#pragma omp section
structured-block
...
}
clause:
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*operator*: *list*)
nowait

Scope of Variables

- Clauses determine the scope of variables
  - Default: shared (external)
  - Private
    - Also, if declared inside region
  - firstprivate
  - shared
  - lastprivate
  - reductions
  - default
  - ….
- Avoid race conditions!

Data Scope Attribute Clauses

- PRIVATE – variables in the list are private to each thread.
- SHARED – variables in the list are shared between all threads.
- DEFAULT – default scope for all variables in a parallel region.
- FIRSTPRIVATE – variables are private and initialized according to value prior to entry into parallel or work sharing construct.
- LASTPRIVATE – variables are private, the value from the last iteration or section is copied to original variable object.
- Others – COPYIN, COPYPRIVATE
- REDUCTION – reduction on variables in the list

Parallel Region Construct

```
!$OMP PARALLEL [clause ...]
        IF (scalar_logical_expression)
        PRIVATE (list)
        SHARED (list)
        DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
        FIRSTPRIVATE (list)
        REDUCTION (operator: list)
        COPYIN (list)
        NUM_THREADS (scalar-integer-expression)

  code block
!$OMP END PARALLEL
```

Number of Threads

- Number of threads will be determined in the following order of precedence:
  - Evaluation of the IF clause
  - Setting of NUM_THREADS clause
  - omp_set_num_threads() library function
  - OMP_NUM_THREADS environment variable
  - Default – usually ends up being the ***number of cores on the node*** (!)

- The last factor can accidentally lead to oversubscription of nodes in hybrid MPI/OpenMP codes.

Work – Sharing Constructs

Directive format (C version):

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

  for_loop
```

Work-Sharing

- Schedule:
  - Static – Loop iterations are statically divided (chunk or as close to even as possible)
  - Dynamic – Loop iterations are divided in size chunk, and dynamically scheduled among threads. When a thread finishes one chunk it is dynamically assigned another
  - Guided – Similar to dynamic but chunk size is proportionally reduced based on work remaining.
  - Runtime – set at runtime by environment variables
  - Auto – set by compiler or runtime system.

Copy the Examples Directory

cp -r /cm/shared/examples/sdsc/si/2025/PARALLEL $HOME/

Verify:

ls $HOME/PARALLEL

```
[train106@login01 ~]$ cp -r /cm/shared/examples/sdsc/si/2023/PARALLEL $HOME/

[train106@login01 ~]$
[train106@login01 ~]$ ls $HOME/PARALLEL
COLLECTIVES  DOMAIN  MISC  OPENMP  PTOP  run_commands.txt  SIMPLE
[train106@login01 ~]$
```

Simple OpenMP Program – Compute PI

- Find the number of tasks and taskids (omp_get_num_threads, omp_get_thread_num)
- PI is calculated using an integral. The number of intervals used for the integration is fixed at 128000.
- Use OpenMP loop parallelization to divide up the compute work.
- Introduce concept of private and shared variables.
- OpenMP reduction operation used to compute the sum for the final integral.
- Today's OpenMP examples are in: $HOME/PARALLEL/OPENMP
- If you don't see the directory, you can copy it from:
  /cm/shared/examples/sdsc/si/2025/PARALLEL

OpenMP Program to Compute PI

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
int nthreads, tid;
int i, INTERVALS;
double n_1, x, pi = 0.0;

INTERVALS=128000;
/* Fork a team of threads giving them their own copies of
      variables */
#pragma omp parallel private(nthreads, tid)
 {
 /* Obtain thread number */
 tid = omp_get_thread_num();
 printf("Hello from thread = %d\n", tid);

 /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */

  n_1 = 1.0 / (double)INTERVALS;

/* Parallel loop with reduction for calculating PI */
#pragma omp parallel for private(i,x)
shared(n_1,INTERVALS) reduction(+:pi)
  for (i = 0; i < INTERVALS; i++)
    {
    x = n_1 * ((double)i  + 0.5);
    pi += 4.0 / (1.0 + x * x);
    }
    pi *= n_1;
    printf ("Pi = %.12lf\n", pi);
}
```

OpenMP result: Computing Pi

Compile:

module reset

module load gcc/10.2.0

gcc -fopenmp -o pi_openmp pi_openmp.c

Submit: sbatch --res=si25cpu pi_openmp.sb

```
[mahidhar@login02 OPENMP]$ more pi.24508633.exp-1-11.out
Resetting modules to system default. Reseting $MODULEPATH back to system default. All extra directories will be removed
from $MODULEPATH.
Hello from thread = 15
Hello from thread = 14
Hello from thread = 13
Hello from thread = 6
Hello from thread = 1
Hello from thread = 11
Hello from thread = 3
Hello from thread = 0
Hello from thread = 2
Number of threads = 16
Hello from thread = 8
Hello from thread = 4
Hello from thread = 5
Hello from thread = 10
Hello from thread = 7
Hello from thread = 9
Hello from thread = 12
Pi = 3.141592653595
```

More Work-Share Constructs

- SECTIONS directive – enclosed sections are divided among the threads.
- WORKSHARE directive – divides execution of block into units of work, each of which is executed once.
- SINGLE directive – Enclosed code is executed by only one thread.

Synchronization Constructs

- MASTER directive – Specifies region is executed only by the master thread.
- CRITICAL directive – Region of the code that is executed one thread at a time.
- BARRIER directive – synchronize all threads
- TASKWAIT directive – wait for all child tasks to complete
- ATOMIC directive – specific memory location updated atomically (not let all threads write at the same time)

Simple Application using OpenMP: 1-D Heat Equation

- $\partial T/\partial t = \alpha(\partial^2 T/\partial x^2)$; $T(0) = 0$; $T(1) = 0$; $(0 \leq x \leq 1)$

  $T(x,0)$ is known as an initial condition.

- Discretizing for numerical solution we get:

  $T^{(n+1)}_i - T^{(n)}_i = (\alpha \Delta t/\Delta x^2)(T^{(n)}_{i-1} - 2T^{(n)}_i + T^{(n)}_{i+1})$

  (*n* is the index in time and *i* is the index in space)

Fortran OpenMP Code: 1-D Heat Equation

```
PROGRAM HEATEQN
    implicit none
    integer :: iglobal, itime, nthreads
    real*8 :: xalp,delx,delt,pi
    real*8 :: T(0:100,0:10)
    integer:: id
    integer:: OMP_GET_THREAD_NUM,
        OMP_GET_NUM_THREADS
!$OMP PARALLEL SHARED(nthreads)
!$OMP MASTER
    nthreads = omp_get_num_threads()
    write (*,*) 'There are', nthreads, 'threads'
!$OMP END MASTER
!$OMP END PARALLEL
    if (nthreads.ne.3) then
      write(*,*)"Use exactly 3 threads for this case"
      stop
    endif
    delx = 0.1d0
    delt = 1d-4
    xalp = 2.0d0
************* Initial Conditions ******************************
    pi = 4d0*datan(1d0)
    do iglobal = 0, 10
     T(0,iglobal) = dsin(pi*delx*dfloat(iglobal))
    enddo
************* Iterations ***************************************
    do itime = 1 , 3
     write(*,*)"Running Iteration Number ", itime
!$OMP PARALLEL DO PRIVATE(iglobal) SHARED(T,xalp,delx,delt,itime)
     do iglobal = 1, 9
      T(itime,iglobal)=T(itime-1,iglobal)+
    +  xalp*delt/delx/delx*
    +  (T(itime-1,iglobal-1)-2*T(itime-1,iglobal)+T(itime-1,iglobal+1))
     enddo
!$OMP BARRIER
    enddo
    do iglobal = 0, 10
     write(*,*)iglobal,T(3, iglobal)
    enddo
    END
```

OpenMP result: 1-D Heat Equation

Compile:

module reset

module load gcc/10.2.0

gfortran -fopenmp -ffixed-form -o heat_openmp heat_openmp.f90

Submit: sbatch --res=si25cpu heat_openmp.sb

```
[mahidhar@login02 OPENMP]$ more heat.24508716.exp-1-11.out
Resetting modules to system default. Reseting $MODULEPATH back to system default. All extra directories will be removed
from $MODULEPATH.
 There are           3 threads
 Running Iteration Number           1
 Running Iteration Number           2
 Running Iteration Number           3
           0   0.0000000000000000
           1  0.30720562101728494
           2  0.58433981542197655
           3  0.80427475735827125
           4  0.94548168233259799
           5  0.99413827268197230
           6  0.94548168233259799
           7  0.80427475735827125
           8  0.58433981542197666
           9  0.30720562101728505
          10   1.1588922802093023E-310
```

Run Time Library Routines

- Setting and querying number of threads
- Querying thread identifier, team size
- Setting and querying dynamic threads feature
- Querying if in parallel region and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks, nested locks.
- Querying wall clock time and resolution.

Environment Variables

- OMP_SCHEDULE – e.g set to “dynamic”
- OMP_NUM_THREADS
- OMP_DYNAMIC (TRUE or FALSE)
- OMP_PROC_BIND (TRUE or FALSE)
- OMP_NESTED (TRUE of FALSE)
- OMP_STACKSIZE – size of stack for created threads
- OMP_THREAD_LIMIT

General OpenMP Performance Considerations

- Avoid or minimize use of BARRIER, CRITICAL (complete serialization here!), ORDERED regions, and locks. Can use NOWAIT clause to avoid redundant barriers.

- Parallelize at a high level, i.e. maximize the work in the parallel regions to reduce parallelization overhead.

- Use appropriate loop scheduling – static has low synchronization overhead but can be unbalanced, dynamic (and guided) have higher synchronization overheads but can improve load balancing.

- Avoid false sharing (more about it in following slide)!

OpenMP: Offloading to GPUs

- Heterogeneous systems with accelerators/coprocessors supported since OpenMP version 4.0
- Use target construct to offload execution from the host to the target devices
- Typical approach is host driven:
  - Create data environments on devices (GPUs for example)
  - Map data to device data environment
  - Offload OpenMP target regions to the target device and exploit parallelism
  - Transfer data from device to host
  - Destroy data environment on device
- APUs make it simpler – don’t need to move data!

MI300A: APU Programming Model

- Don’t need to explicitly manage memory as in the case of heterogenous architectures
- Can incrementally change your code, starting with most compute intensive areas. Easier to port complex code regions as memory management is not needed
- Can reuse most of the code – unified code base possible in OpenMP offload approach
- Portable – managed memory support is available on other devices

MI300A: APU Programming Model – OpenMP Offload

CPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);

for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

cpu_func(in_h, out_h, M);

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

GPU CODE - OPENMP

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);

for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

#pragma omp target teams map(to:in_h[0:M])
map(tofrom:out_h[0:M])
{ … }

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

APU CODE - OPENMP

```
#pragma omp requires unified_shared_memory
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);

for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

#pragma omp target
{ … }

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

On the APU:

- ~~GPU memory allocation on Device~~
- ~~Explicit memory management between CPU & GPU~~
- Synchronization Barrier. Implicit in case of OpenMP

Reference: https://rocm.blogs.amd.com/software-tools-optimization/mi300a-programming/README.html

What is False Sharing?

- Most modern processors have a cache buffer between slow memory and high-speed registers of the CPU.
- Accessing a memory location causes a “cache line” to be copied into the cache.
- In an OpenMP code two processors may be accessing two different elements in the same cache line. On writes this will lead to “cache line” being marked invalid (because cache coherency is being maintained).
- This will lead to an increase in memory traffic even though the write is to different elements (hence the term false sharing).
- This can have a drastic performance impact if such updates are occurring frequently in a loop.

Detailed info:

https://www.youtube.com/watch?v=CMJXvTF-gJk

False Sharing Example

Code snippet:

```
double global=0.0, local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
 int tid = omp_get_thread_num();
 local[tid] = 0.0;
 #pragma omp for
 for (i = 0; i < N; i++)
 local[tid] += x[i];
 #pragma omp atomic
 global += local[me];
}
```

False Sharing - Solutions

- Three options
  - Compiler directives to align individual variables on cache line boundaries

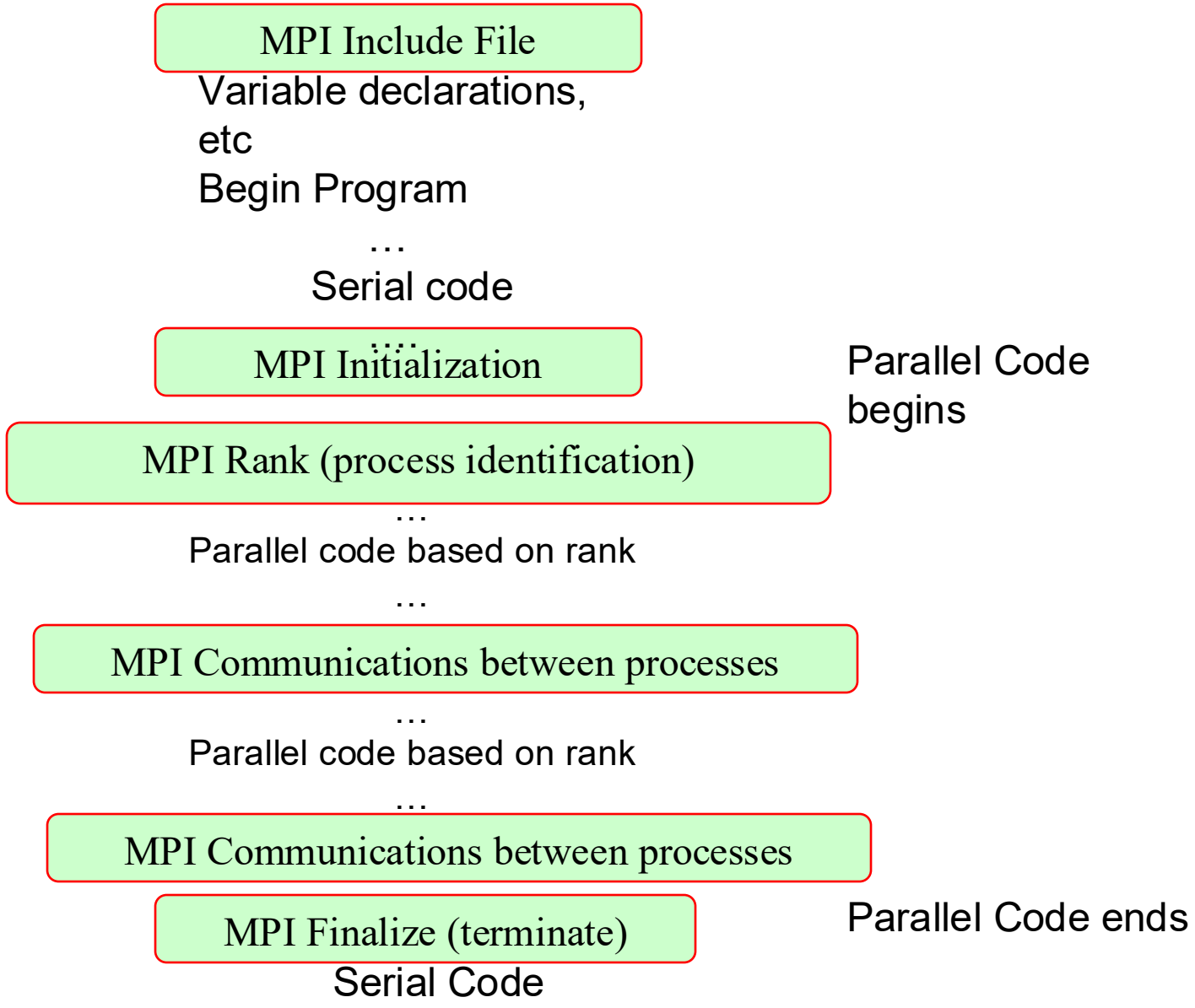
```
__declspec (align(64)) int thread1_global_variable;
__declspec (align(64)) int thread2_global_variable;
```

  - Pad arrays/data structures to make sure array elements begin on cache line boundary.
  - Use thread local copies of data (assuming the copy overhead is small compared to overall run time).

Message Passing Interface (MPI)

- Low level message passing abstraction
  - SPMD execution model + messages
  - Designed for distributed memory. Implemented on hybrid distributed memory/shared memory systems.
- MPI: API specification
  - Portable: de-fact standard for parallel computing, portable, system specific optimizations without changing code interface
  - http://www.mpi-forum.org
  - Several implementations - e.g MVAPICH2, Intel MPI, and OpenMPI – *all 3 are available on Expanse*
  - High performance implementations available virtually on any interconnect and system
  - Point-to-point communication, datatypes, collective operations
  - One-sided communication, Parallel file I/O, Tool support, …

Typical MPI Code Structure



Simple MPI Program – Compute PI

- Initialize MPI (MPI_Init function)
- Find the number of tasks and taskids (MPI_Comm_size, MPI_Comm_rank)
- PI is calculated using an integral. The number of intervals used for the integration is fixed at 128000.
- Computes the sums for a different sections of the intervals in each MPI task.
- At the end of the code, the sums from all the tasks are added together to evaluate the final integral. This is accomplished through a reduction operation (MPI_Reduce function).
- Simple code illustrates decomposition of problem into parallel components.

MPI Program to Compute PI

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

int main(int argc, char *argv[])
 {
 int numprocs, rank;
 int i, iglob, INTERVALS, INTLOC;
 double n_1, x;
 double pi, piloc;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 INTERVALS=128000;
 printf("Hello from MPI task= %d\n", rank);
 MPI_Barrier(MPI_COMM_WORLD);
 if (rank == 0)
  {
 printf("Number of MPI tasks = %d\n", numprocs);
  }

 INTLOC=INTERVALS/numprocs;
 piloc=0.0;
 n_1=1.0/(double)INTERVALS;
 for (i = 0; i < INTLOC; i++)
  {
  iglob = INTLOC*rank+i;
  x = n_1 * ((double)iglob  - 0.5);
  piloc += 4.0 / (1.0 + x * x);
  }

MPI_Reduce(&piloc,&pi,1,MPI_DOUBLE,MPI_SUM,0
,MPI_COMM_WORLD);
 if (rank == 0)
  {
  pi *= n_1;
  printf ("Pi = %.12lf\n", pi);
  }

 MPI_Finalize();
}
```

PI Code : MPI Environment Functions

MPI_Init(&argc, &argv);

Initializes MPI, *must* be called (only once) in every MPI program before any MPI functions.

MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

Returns the total number of tasks in the communicator. MPI uses communicators to define which collections of processes can communicate with each other. The default MPI_COMM_WORLD includes all the processes. User defined communicators are an option.

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

Returns the rank (ID) of the calling MPI process within the communicator.

MPI_Finalize();

Ends the MPI execution environment. No MPI calls after this.!

The other routines in the code are collectives and we will discuss them later in the talk.

Compiling and Running PI Example

cd $HOME/PARALLEL/SIMPLE

Modules: module reset; module load gcc/10.2.0 mvapich2/2.3.7

Compile: mpicc -o pi_mpi.exe pi_mpi.c

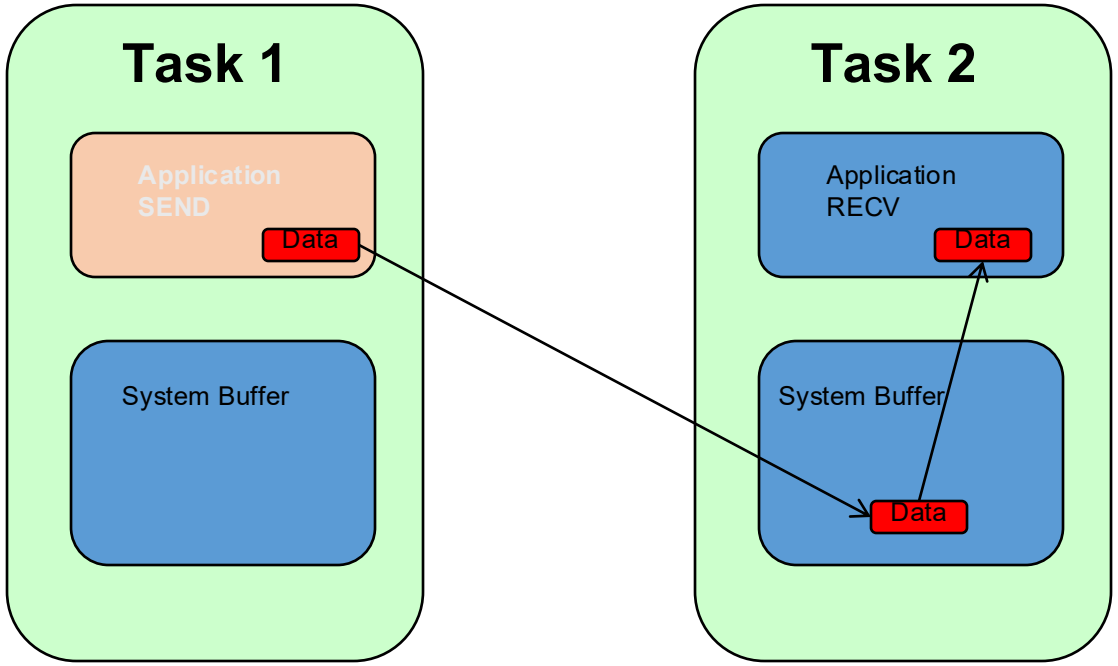
Submit Job: sbatch --res=si25cpu pi_mpi.sb

```
[mahidhar@login02 SIMPLE]$ more pi_mpi.24508779.exp-1-11.out
Resetting modules to system default. Reseting $MODULEPATH back to system default. All extra directories will be removed
from $MODULEPATH.
Hello from MPI task= 13
Hello from MPI task= 14
Hello from MPI task= 15
Hello from MPI task= 12
Hello from MPI task= 3
Hello from MPI task= 1
Hello from MPI task= 2
Hello from MPI task= 4
Hello from MPI task= 5
Hello from MPI task= 6
Hello from MPI task= 7
Hello from MPI task= 8
Hello from MPI task= 9
Hello from MPI task= 10
Hello from MPI task= 11
Hello from MPI task= 0
Number of MPI tasks = 16
Pi = 3.141592653590
```

Point to Point Communication

- Passing data between two, and only two different MPI tasks.
- Typically one task performs a send operation and the other task performs a matching receive.
- MPI Send operations have choices with different synchronization (when does a send complete) and different buffering (where the data resides till it is received) modes.
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides routines to probe status of messages, and "wait" routines.

Buffers



- Buffer space is used for data in transit – whether its waiting for a receive to be ready or if there are multiple sends arriving at the same receiving tasks.
- Typically a system buffer area managed by the MPI library (opaque to the user) is used. Can exist on both sending & receiving side.
- MPI also provides for user managed send buffer.

Blocking MPI Send, Receive Routines

- Blocking send call will return once it is safe for the application buffer (send data) to be reused.
- This can happen as soon as the data is copied into the system (MPI) buffer on receiving process.
- Synchronous if there is confirmation of safe send, and asynchronous otherwise.
- Blocking receive returns once the data is in the application buffer (receive data) and can by used by the application.

Blocking Send, Recv Example (Code Snippet)

```
if(myid == 0) {
      for(i = 0; i < 10; i++) {
        s_buf[i] = i*4.0;
      }
      MPI_Send(s_buf, size, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
  }
  else if(myid == 1) {
      MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &reqstat);
      for (i = 0; i < 10; i++ ){
        printf("r_buf[%d] = %f\n", i, r_buf[i] );
      }
  }
```

Break

Blocking Send, Recv Example

Location:$HOME/PARALLEL/PTOP

Compile: mpicc -o blocking.exe blocking.c

Submit Job: sbatch --res=si25cpu blocking.sb

Output:

r_buf[0] = 0.000000
r_buf[1] = 4.000000
r_buf[2] = 8.000000
r_buf[3] = 12.000000
r_buf[4] = 16.000000
r_buf[5] = 20.000000
r_buf[6] = 24.000000
r_buf[7] = 28.000000
r_buf[8] = 32.000000
r_buf[9] = 36.000000

Deadlocking MPI Tasks

- Take care to sequence blocking send/recvs. Easy to deadlock processes waiting on each other with circular dependencies.
- Can also occur with control errors and unexpected symantics
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
  }
  else if(myid == 1) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    for (i = 0; i < 10; i++ ){
      printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
```

- The MPI_Ssend on both tasks will not complete till the MPI_Recv is posted (which will never happen given the order).

Deadlock Example

- Location: $HOME/PARALLEL/PTOP
- Compile: mpicc -o deadlock.exe deadlock.c
- Submit Job: sbatch --res=si25cpu deadlock.sb
- It should technically finish in less than a second since the data transferred is a few bytes. However, the code deadlocks and hits the wallclock limit (1 minute in the script).

```
[mahidhar@login02 PTOP]$ more deadlock.24508842.exp-1-11.out
Resetting modules to system default. Reseting $MODULEPATH back to system default. All extra directories will be removed
from $MODULEPATH.
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 24508842.0 ON exp-1-11 CANCELLED AT 2023-08-10T11:44:38 DUE TO TIME LIMIT ***
slurmstepd: error: *** JOB 24508842 ON exp-1-11 CANCELLED AT 2023-08-10T11:44:38 DUE TO TIME LIMIT ***
[mahidhar@login02 PTOP]$
```

Deadlock Example – Simple Fix

- Change the order on one of processes!
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
  }
  else if(myid == 1) {
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    for (i = 0; i < 10; i++ ){
      printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
```

- Now the MPI_Ssend on task 0 will complete since the corresponding MPI_Recv is posted first on task 1. (qsub deadlock-fix1.cmd)
- We will look at Non-Blocking options next.

Deadlock Example (Fix 1)

- Location: $HOME/PARALLEL/PTOP
- Compile: mpicc -o deadlock-fix1.exe deadlock-fix1.c
- Submit Job: sbatch --res=si25cpu deadlock-fix1.sb
- Fix works!

$ **more deadlock-fix1.out**

r_buf[0] = 0.000000
r_buf[1] = 4.000000
r_buf[2] = 8.000000
r_buf[3] = 12.000000
r_buf[4] = 16.000000
r_buf[5] = 20.000000
r_buf[6] = 24.000000
r_buf[7] = 28.000000
r_buf[8] = 32.000000
r_buf[9] = 36.000000

Non-Blocking MPI Send, Receive Routines

- Non-Blocking MPI Send, Receive routines return before there is any confirmation of receives or completion of the actual message copying operation.
- The routines simply put in the request to perform the operation.
- MPI wait routines can be used to check status and block till the operation is complete and it is safe to modify/use the information in the application buffer.
- This non-blocking approaches allows computations (that don't depend on this data in transit) to continue while the communication operations are in progress. This allows for hiding the communication time with useful work and hence improves parallel efficiency.

Non-Blocking Send, Recv Example

- Example uses MPI_Isend, MPI_Irecv, MPI_Wait
- Code snippet:

```
if(myid == source){
        s_buf=1024;
        MPI_Isend(&s_buf,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
   }
   if(myid == destination {
        MPI_Irecv(&r_buf,count,MPI_INT,source,tag,MPI_COMM_WORLD,&request);
   }
   MPI_Wait(&request,&status);
```

- Compile & Run:

mpicc -o nonblocking.exe nonblocking.c

sbatch --res=si25cpu nonblocking.sb

Sample output:

processor 0 sent 1024

processor 1 got 1024

Deadlock Example – Non-Blocking Option

- Change the order on one of processes!
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Isend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD, &request);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
  }
  else if(myid == 1) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    for (i = 0; i < 10; i++ ){
      printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
```

- Now the MPI_Ssend on task 0 will complete since the corresponding MPI_Recv is posted first on task 1. (qsub deadlock-fix1.cmd)
- We will look at Non-Blocking options next.

Deadlock Example (Fix 2)

- Location: $HOME/PARALLEL/PTOP
- Compile: mpicc -o deadlock-fix2-nb.exe deadlock-fix2-nb.c
- Submit Job: sbatch --res=si25cpu deadlock-fix2-nb.sb
- Fix works!

$ **more deadlock-fix2-nb.out**

r_buf[0] = 0.000000
r_buf[1] = 4.000000
r_buf[2] = 8.000000
r_buf[3] = 12.000000
r_buf[4] = 16.000000
r_buf[5] = 20.000000
r_buf[6] = 24.000000
r_buf[7] = 28.000000
r_buf[8] = 32.000000
r_buf[9] = 36.000000

Collective MPI Routines

- Synchronization Routines: All processes in group/communicator wait till they get synchronized.
- Data Movement: Send/Receive data from all processes. E.g. Broadcast, Scatter, Gather, AlltoAll.
- Collective Computation (reductions): Perform reduction operations (min, max, add, multiply, etc.) on data obtained from all processes.
- Collective Computation and Data Movement combined (Hybrid).

Examples for Collectives

- Location

  $HOME/PARALLEL/COLLECTIVES

- Switch compilers

  module reset; module load intel/19.1.3.304; module load openmpi/4.1.3

Synchronization Example

- Our simple PI program had a synchronization example.
- Code Snippet:

```
printf("Hello from MPI task= %d\n", rank);
MPI_Barrier(MPI_COMM_WORLD);
if (rank == 0)
  {
  printf("Number of MPI tasks = %d\n", numprocs);
  }
```

- All tasks will wait till they are synchronized at this point.

Broadcast Example

- Code Snippet (All collectives examples in $HOME/PARALLEL/COLLECTIVES):
  - if(myid .eq. source)then
  - do i=1,count
  - buffer(i)=i
  - enddo
  - endif
  - **Call MPI_Bcast(buffer, count, MPI_INTEGER,source,&**
  - **MPI_COMM_WORLD,ierr)**
- Compile:
  - **mpif90 -o bcast.exe bcast.f90**
- Run:
  - **sbatch --res=si25cpu bcast.sb**
- Output:

```
processor      1 got      1      2      3      4
processor      0 got      1      2      3      4
processor      2 got      1      2      3      4
processor      3 got      1      2      3      4
```

Reduction Example

- Code Snippet:

```
myidp1 = myid+1
call MPI_Reduce(myidp1,ifactorial,1,MPI_INTEGER,MPI_PROD,root,MPI_COMM_WORLD,ierr)
if (myid.eq.root) then
    write(*,*)numprocs,"! = ",ifactorial
endif
```

- Compile:

```
mpif90 -o factorial.exe factorial.f90
```

- Run:

```
sbatch --res=si25cpu factorial.sb
```

- **Output:**

```
8 ! =       40320
```

MPI_Allreduce example

- Code Snippet:

```
imaxloc=IRAND(myid)
call MPI_ALLREDUCE(imaxloc,imax,1,MPI_INTEGER,MPI_MAX,MPI_COMM_WORLD,
mpi_err)
if (imax.eq.imaxloc) then
    write(*,*)"Max=",imax,"on task",myid
endif
```

- Compile:

  mpif90 -o allreduce.exe allreduce.f90

- Run:

  sbatch --res=si25cpu allreduce.sb

- Output:

  Max= 337897 on task 7

Data Types

C Data Types		FORTRAN Data Types
MPI_CHAR	MPI_C_DOUBLE_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_BOOL	MPI_INTEGER1
MPI_INT	MPI_LOGICAL	MPI_INTEGER2
MPI_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_INT8_T	MPI_REAL
MPI_LONG_LONG	MPI_INT16_T	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT32_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT64_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_UINT8_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_UINT16_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT32_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT64_T	MPI_LOGICAL
MPI_DOUBLE	MPI_BYTE	MPI_BYTE
MPI_LONG_DOUBLE	MPI_PACKED	MPI_PACKED
MPI_C_COMPLEX		
MPI_C_FLOAT_COMPLEX		

MPI Reduction Operations

NAME	OPERATION
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Decomposition and Mapping

- Data and work decomposition
  - Map partitioned domain to processes
- Mapping
  - Processes/ranks topology
  - System/Domain/Data
- How to share data?
  - Exchange messages and replicate data
- Load imbalance
  - What if the system is not regular?
  - Is work proportional to size of partitions?

Simple Application using MPI: 1-D Heat Equation

- $\partial T/\partial t = \alpha(\partial^2 T/\partial x^2)$; $T(0) = 0$; $T(1) = 0$; $(0 \leq x \leq 1)$

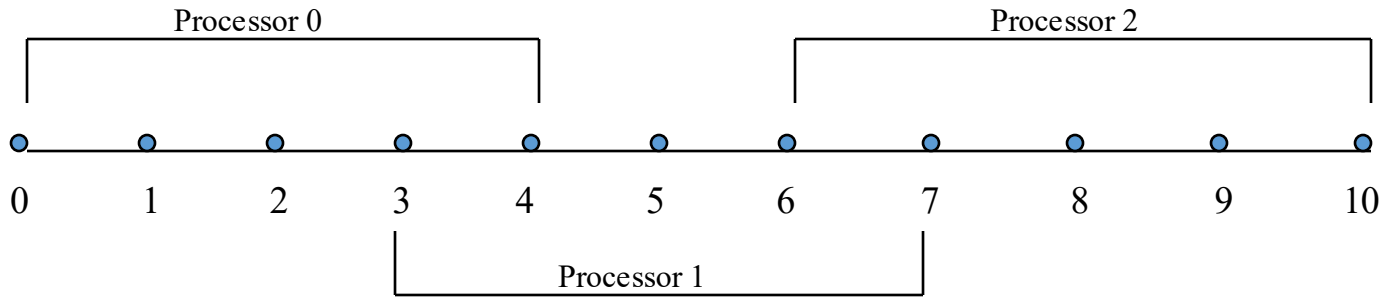
  T(x,0) is know as an initial condition.

- Discretizing for numerical solution we get:

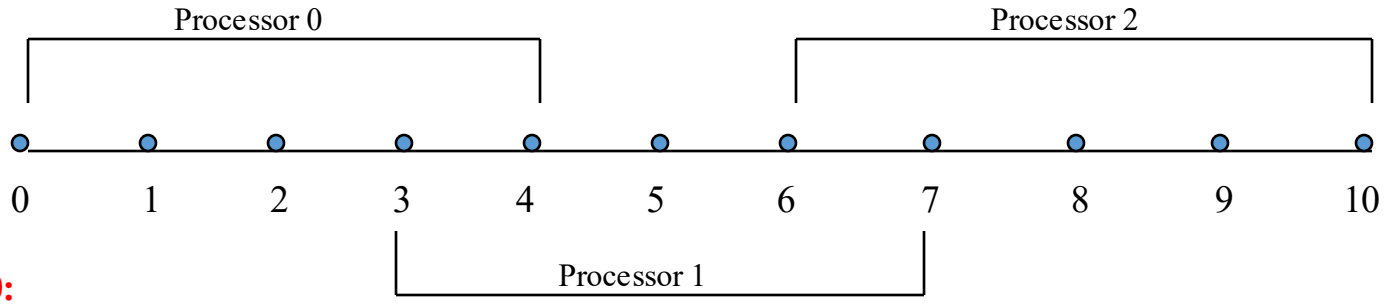
  $T^{(n+1)}_i - T^{(n)}_i = (\alpha \Delta t/\Delta x^2)(T^{(n)}_{i-1} - 2T^{(n)}_i + T^{(n)}_{i+1})$

  (*n* is the index in time and *i* is the index in space)

- In this example we solve the problem using 11 points and we distribute this problem over exactly 3 processors (for easy demo) shown graphically below:



Simple Application using MPI: 1-D Heat Equation



Processor 0:

Local Data Index : ilocal = 0 , 1, 2, 3, 4

Global Data Index: iglobal = 0, 1, 2, 3, 4

Solve the equation at (1,2,3)

Data Exchange: **Get 4 from processor 1; Send 3 to processor 1**

Processor 1:

Local Data Index : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 3, 4, 5, 6, 7

Solve the equation at (4,5,6)

Data Exchange: **Get 3 from processor 0; Get 7 from processor 2; Send 4 to processor 0; Send 6 to processor 2**

Processor 2:

Local Data Index : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 6, 7, 8, 9, 10

Solve the equation at (7,8,9)

Data Exchange: **Get 6 from processor 1; Send 7 to processor 1**

FORTRAN MPI CODE: 1-D Heat Equation

```
PROGRAM HEATEQN
  implicit none
  include "mpif.h"
  integer :: iglobal, ilocal, itime
  integer :: ierr, nnodes, my_id
  integer :: dest, from, status(MPI_STATUS_SIZE),tag
  integer :: msg_size
  real*8 :: xalp,delx,delt,pi
  real*8 :: T(0:100,0:5), TG(0:10)
  CHARACTER(20) :: FILEN

  delx = 0.1d0
  delt = 1d-4
  xalp = 2.0d0

  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nnodes, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)

  if (nnodes.ne.3) then
    if (my_id.eq.0) then
     print *, "This test needs exactly 3 tasks"
    endif
```

```
 print *, "Process ", my_id, "of", nnodes ,"has started"
!************ Initial Conditions
*********************************
      pi = 4d0*datan(1d0)
      do ilocal = 0, 4
        iglobal = 3*my_id+ilocal
        T(0,ilocal) = dsin(pi*delx*dfloat(iglobal))
      enddo
      write(*,*)"Processor", my_id, "has finished setting
     + initial conditions"
!************ Iterations ************************************
      do itime = 1 , 3
       if (my_id.eq.0) then
         write(*,*)"Running Iteration Number ", itime
       endif
       do ilocal = 1, 3
         T(itime,ilocal)=T(itime-1,ilocal)+
     +   xalp*delt/delx/delx*
     +   (T(itime-1,ilocal-1)-2*T(itime-1,ilocal)+T(itime-
1,ilocal+1))
       enddo
       if (my_id.eq.0) then
         write(*,*)"Sending and receiving overlap points"
         dest = 1
```

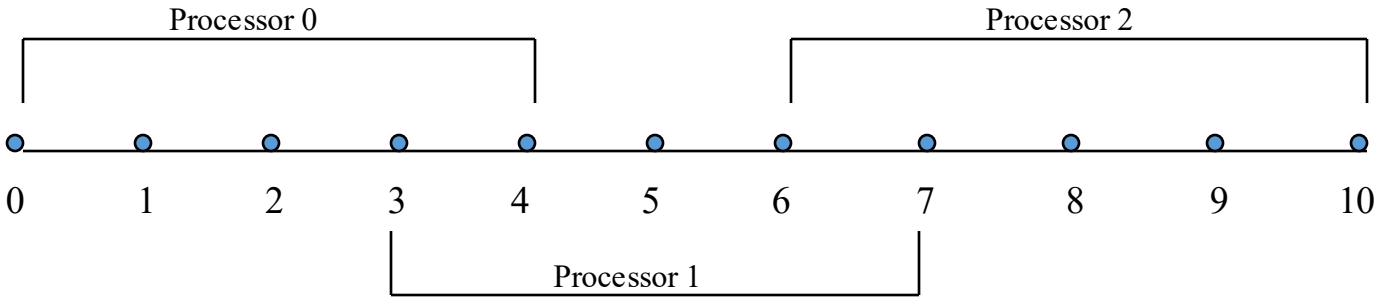
Fortran MPI Code: 1-D Heat Equation (Contd.)

```
 msg_size = 1
      call
MPI_SEND(T(itime,3),msg_size,MPI_DOUBLE_PRECISION,dest,
     +              tag,MPI_COMM_WORLD,ierr)
      endif
      if (my_id.eq.1) then
        from = 0
        dest = 2
        msg_size = 1
        call
MPI_SEND(T(itime,3),msg_size,MPI_DOUBLE_PRECISION,dest,
     +              tag,MPI_COMM_WORLD,ierr)
        call
MPI_RECV(T(itime,0),msg_size,MPI_DOUBLE_PRECISION,from,
     +              tag,MPI_COMM_WORLD,status,ierr)
      endif
      if (my_id.eq.2) then
        from = 1
        dest = 1
        msg_size = 1
        call
MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
     +              tag,MPI_COMM_WORLD,ierr)
        call
MPI_RECV(T(itime,0),msg_size,MPI_DOUBLE_PRECISION,from,
     +              tag,MPI_COMM_WORLD,status,ierr)
      endif
      if (my_id.eq.1) then
        from = 2
        dest = 0
        msg_size = 1
        call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +              tag,MPI_COMM_WORLD,status,ierr)
        call MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
     +              tag,MPI_COMM_WORLD,ierr)
      endif
      if (my_id.eq.0) then
        from = 1
        msg_size = 1
        call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +              tag,MPI_COMM_WORLD,status,ierr)
      endif
     enddo

     if (my_id.eq.0) then
       write(*,*)"SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:"
     endif
     FILEN = 'data'//char(my_id+48)//'.dat'
     open (5, file=FILEN)
     write(5,*)"Processor ",my_id
     do ilocal = 0 , 4
      iglobal = 3*my_id + ilocal
      write(5,*)"ilocal=",ilocal,";iglobal=",iglobal,";T=",T(3,ilocal)
     enddo
     close(5)
     call MPI_FINALIZE(ierr)

     END
```

Simple Application using MPI: 1-D Heat Equation



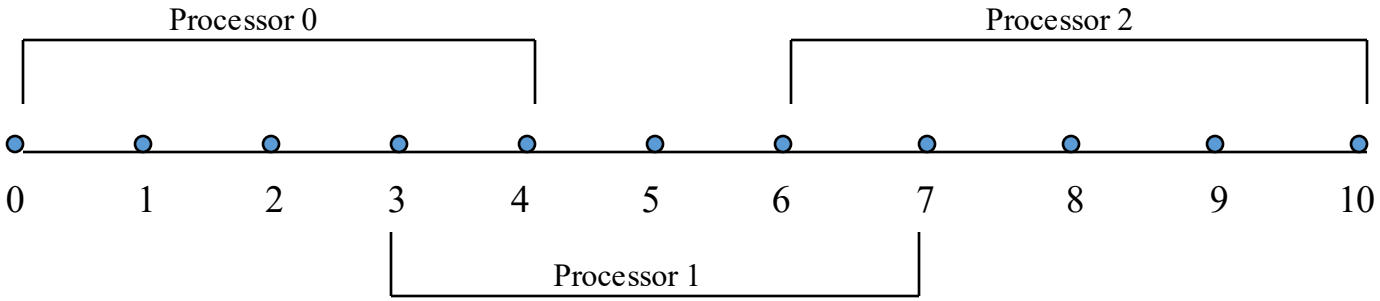
•Compilation

```
module reset
module load gcc/10.2.0 mvapich2/2.3.7
mpif90 -ffixed-form heat_mpi.f90 -o heat_mpi.exe
```

•Run Job:

```
sbatch --res=si25cpu heat_mpi.sb
```

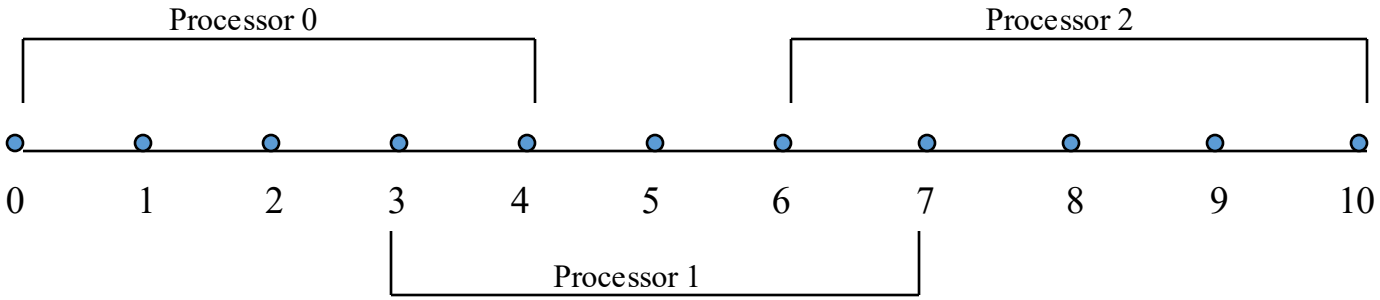
Simple Application using MPI: 1-D Heat Equation



OUTPUT FROM SAMPLE PROGRAM

```
Process  0 of 3 has started
 Processor 0 has finished                setting initial conditions
 Process  1 of 3 has started
 Processor 1 has finished                setting initial conditions
 Process  2 of 3 has started
 Processor 2 has finished                setting initial conditions
 Running Iteration Number  1
 Sending and receiving overlap points
 Running Iteration Number  2
 Sending and receiving overlap points
 Running Iteration Number  3
 Sending and receiving overlap points
 SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:
```

Simple Application using MPI: 1-D Heat Equation




```
% more data0.dat
 Processor  0
 ilocal= 0 ;iglobal= 0 ;T=
0.00000000000000000E+00
 ilocal= 1 ;iglobal= 1 ;T= 0.30720562101728499 1
 ilocal= 2 ;iglobal= 2 ;T= 0.58433981542197654 9
 ilocal= 3 ;iglobal= 3 ;T= 0.80427475735827125 3
 ilocal= 4 ;iglobal= 4 ;T= 0.94548168233259788 4
```

```
% more data2.dat
 Processor  2
 ilocal= 0 ;iglobal= 6 ;T= 0.94548168233259799 5
 ilocal= 1 ;iglobal= 7 ;T= 0.80427475735827125 3
 ilocal= 2 ;iglobal= 8 ;T= 0.58433981542197666 0
 ilocal= 3 ;iglobal= 9 ;T= 0.30720562101728510 2
 ilocal= 4 ;iglobal= 10 ;T=
0.00000000000000000E+00
```

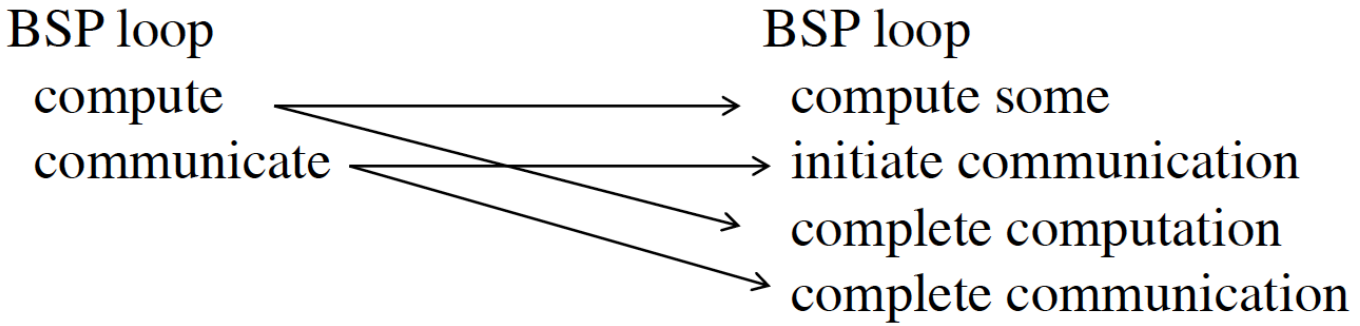
```
% more data1.dat
 Processor  1
 ilocal= 0 ;iglobal= 3 ;T= 0.80427475735827125 3
 ilocal= 1 ;iglobal= 4 ;T= 0.94548168233259788 4
 ilocal= 2 ;iglobal= 5 ;T= 0.99413827268197230 1
 ilocal= 3 ;iglobal= 6 ;T= 0.94548168233259799 5
 ilocal= 4 ;iglobal= 7 ;T= 0.80427475735827125 3
```

Performance Considerations

- Overlap communication with computation
  - Use non-blocking primitives
  - Hide communication cost
  - Split-phase programming
- Minimize surface-to-volume ratio
  - Ghost cell exchange
- Avoid communication
  - Even at the cost of some more computation
  - Example: double size of ghost cell and communicate every other time step
  - Communication avoiding algorithms

Asynchronous Communication

- Overlap communication w/ computation
  - High performance interconnects can offload communication tasks from CPU to adapter
- Condition
  - No data dependencies on transfer
- Split-pha



MPI – Profiling, Tracing Tools

- Several options available. On Expanse we have mpiP and TAU installed.
- Useful when you are trying to isolate performance issues.
- Tools can give you info on how much time is being spent in communication. The levels of detail vary with each tool.
- In general identify scaling bottlenecks and try to overlap communication with computation where possible.

mpiP example

- Location: $HOME/PARALLEL/MISC
- Modules:

  module reset; module load gcc/10.2.0 mvapich2/2.3.7 mpip/3.5
- Compile (compile_profile.readme.txt):

  mpif90 -ffixed-form -g -o heat_mpi_profile.exe heat_mpi.f90 -L$MPIPHOME/lib -lmpiP
- Executable already exists. Just submit

  sbatch --res=si25cpu heat_mpi_profile.sb
- Once the job runs you get a .mpiP file.

mpiP output

```
@ mpiP
@ Command : /home/mahidhar/PARALLEL/MISC/./heat_mpi_profile.exe
@ Version                  : 3.4.1
@ MPIP Build date          : Feb 26 2021, 06:50:01
@ Start time               : 2021 08 04 22:18:31
@ Stop time                : 2021 08 04 22:18:31
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 74358
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 exp-1-01
@ MPI Task Assignment      : 1 exp-1-01
@ MPI Task Assignment      : 2 exp-1-01
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0     0.0241   0.000554     2.30
   1     0.0242   0.000716     2.96
   2     0.0242   0.000657     2.72
```

mpiP Output

```
   *       0.0724    0.00193     2.66
---------------------------------------------------------------------------
@--- Callsites: 8 -----------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line Parent_Funct             MPI_Call
  1   0 0x408ac2                [unknown]                 Recv
  2   0 0x4087db                [unknown]                 Send
  3   0 0x40897b                [unknown]                 Recv
  4   0 0x408924                [unknown]                 Send
  5   0 0x408a50                [unknown]                 Send
  6   0 0x408855                [unknown]                 Send
  7   0 0x4089f9                [unknown]                 Recv
  8   0 0x4088ac                [unknown]                 Recv
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Recv                    8      0.611    0.84   31.73    0.00
Recv                    3      0.583    0.81   30.25    0.00
Recv                    1      0.492    0.68   25.55    0.00
Send                    6      0.082    0.11    4.25    0.00
Send                    4     0.0739    0.10    3.84    0.00
Send                    2     0.0615    0.08    3.19    0.00
```

mpiP output

```
Send                    5     0.0174    0.02    0.91    0.00
Recv                    7    0.00552    0.01    0.29    0.00
---------------------------------------------------------------------------
@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
---------------------------------------------------------------------------
Call                 Site       Count      Total       Avrg  Sent%
Send                    2           3         24          8  25.00
Send                    4           3         24          8  25.00
Send                    5           3         24          8  25.00
Send                    6           3         24          8  25.00
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 8 ----------------------
---------------------------------------------------------------------------
Name                 Site Rank  Count      Max     Mean      Min   App%   MPI%
Recv                    1    0      3    0.487    0.164  0.00253   2.05  88.90
Recv                    1    *      3    0.487    0.164  0.00253   0.68  25.55

Recv                    3    2      3    0.539    0.194   0.0113   2.41  88.75
Recv                    3    *      3    0.539    0.194   0.0113   0.81  30.25

Recv                    7    1      3  0.00317  0.00184  0.00115   0.02   0.77
Recv                    7    *      3  0.00317  0.00184  0.00115   0.01   0.29
```

More Complex routines

- Derived Data Types
- User defined reduction functions
- Groups/communicator management
- Parallel I/O
- One Sided Communication Routines (RDMA)
- MPI-3 Standard has over 400 routines(!).

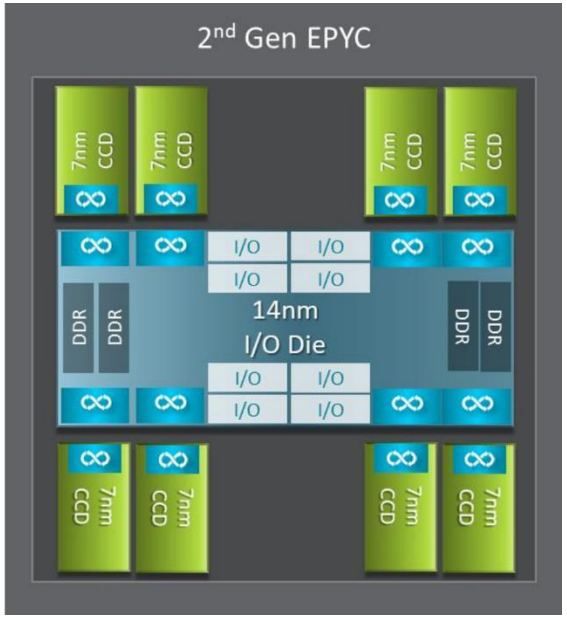
Hybrid MPI/OpenMP Jobs and SLURM Usage on Expanse

Ref: ibrun scripts developed by Manu Shantharam at SDSC

module load sdsc
(puts ibrun in your path)

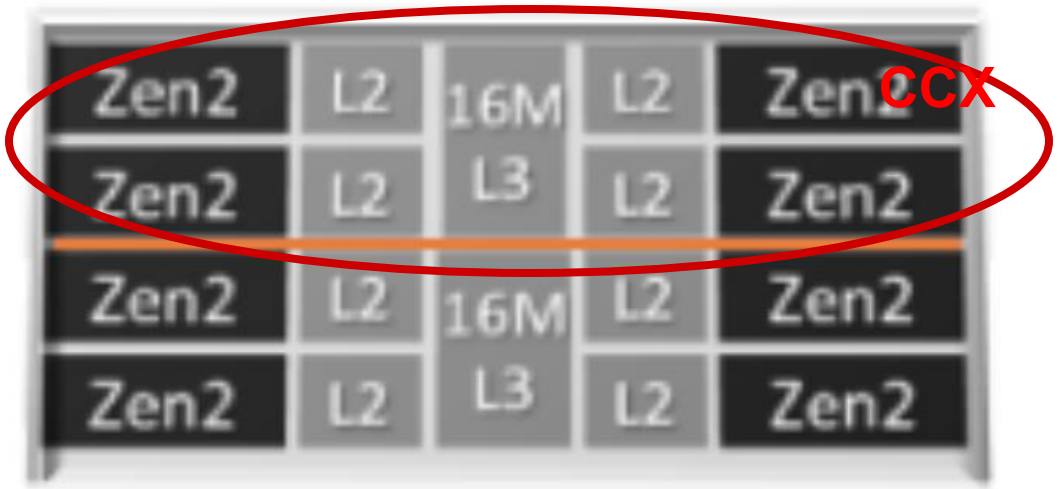
AMD EPYC 7742 Processor Architecture

- 8 Core Complex Dies (CCDs).
- CCDs connect to memory, I/O, and each other through the I/O Die.
- 8 memory channels per socket.
- DDR4 memory at 3200MHz.
- PCI Gen4, up to 128 lanes of high speed I/O.
- Memory and I/O can be abstracted into separate quadrants each with 2 DIMM channels and 32 I/O lanes.



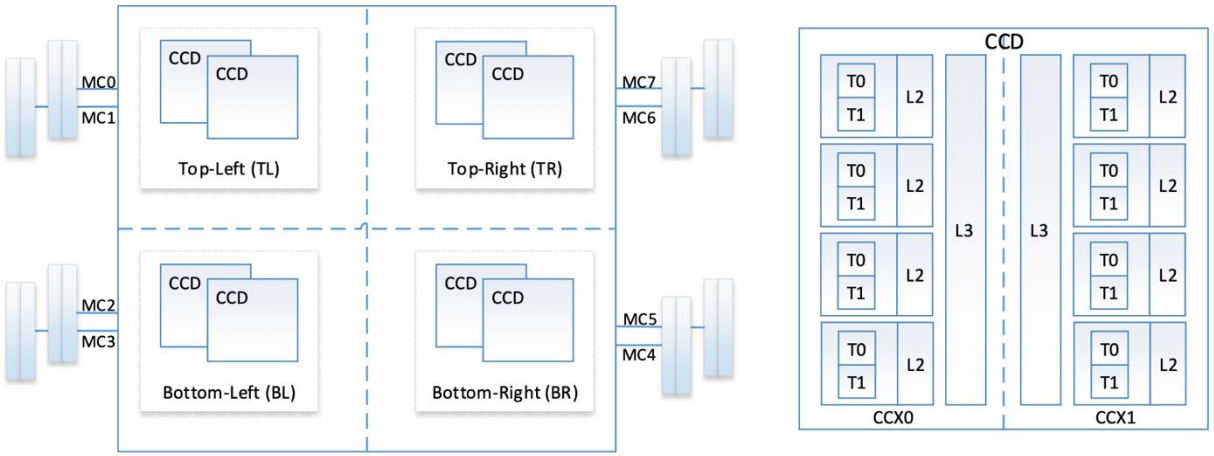
AMD EPYC 7742 Processor: Core Complex Die (CCD)

- 2 Core Complexes (CCXs) per CCD
- 4 Zen2 cores in each CCX shared a 16M L3 cache. Total of 16 x 16 = 256MB L3 cache.
- Each core includes a private 512KB L2 cache.



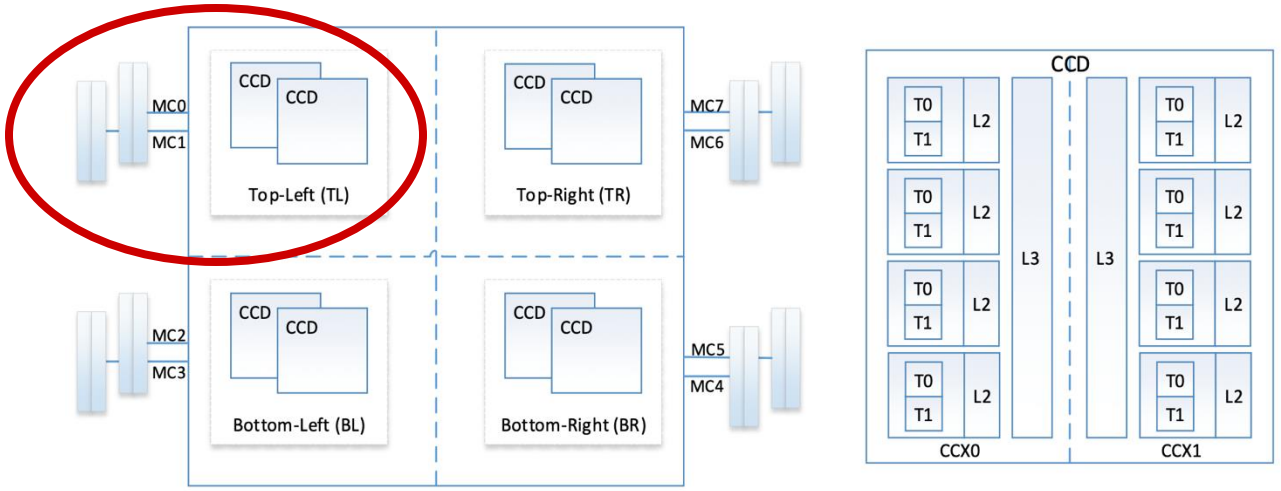
AMD EPYC 7742 Processor : NUMA Nodes Per Socket

- The four logical quadrants allow the processor to be partitioned into different NUMA domains. Options set in BIOS.
- Domains are designated as NUMA per socket (NPS).
- **NPS4:** Four NUMA domains per socket is the typical HPC configuration.



NPS4 Configuration

- The processor is partitioned into four NUMA domains.
- Each logical quadrant is a NUMA domain.
- Memory is interleaved across the two memory channels
- PCIe devices will be local to one of four NUMA domains (the IO die that has the PCIe root for the device)
- ***This is the typical HPC configuration*** as workload is NUMA aware, ranks and memory can be pinned to cores and NUMA nodes.



AMD EPYC: Optimization/Usage Guidelines

- Processor is x86_64
  - Supports AVX2 instruction set
  - Multiple separate L3 caches - 16 on 64-core CPUs. Thread migration affects cache locality
- Make sure the threads stay close to their cache
  - Pinning can make a big impact on performance
  - Need to use at least 2 cores on CCD to maximize cache
- Typically, hybrid approach works better
  - One MPI rank/L3 cache and then OpenMP threads on each core

Using MPI options

- All MPI implementations have affinity options.

- Example OpenMPI run command:

```
mpirun -np 32 --mca pml ucx --mca osc ucx --map-by l3cache xhpl
```

- Example Intel MPI setup:

```
export OMP_NUM_THREADS=16
mpirun -genv I_MPI_PIN_DOMAIN=omp:compact ./hello_hybrid
```

- Can also combine with application pinning options. For example, for NAMD:

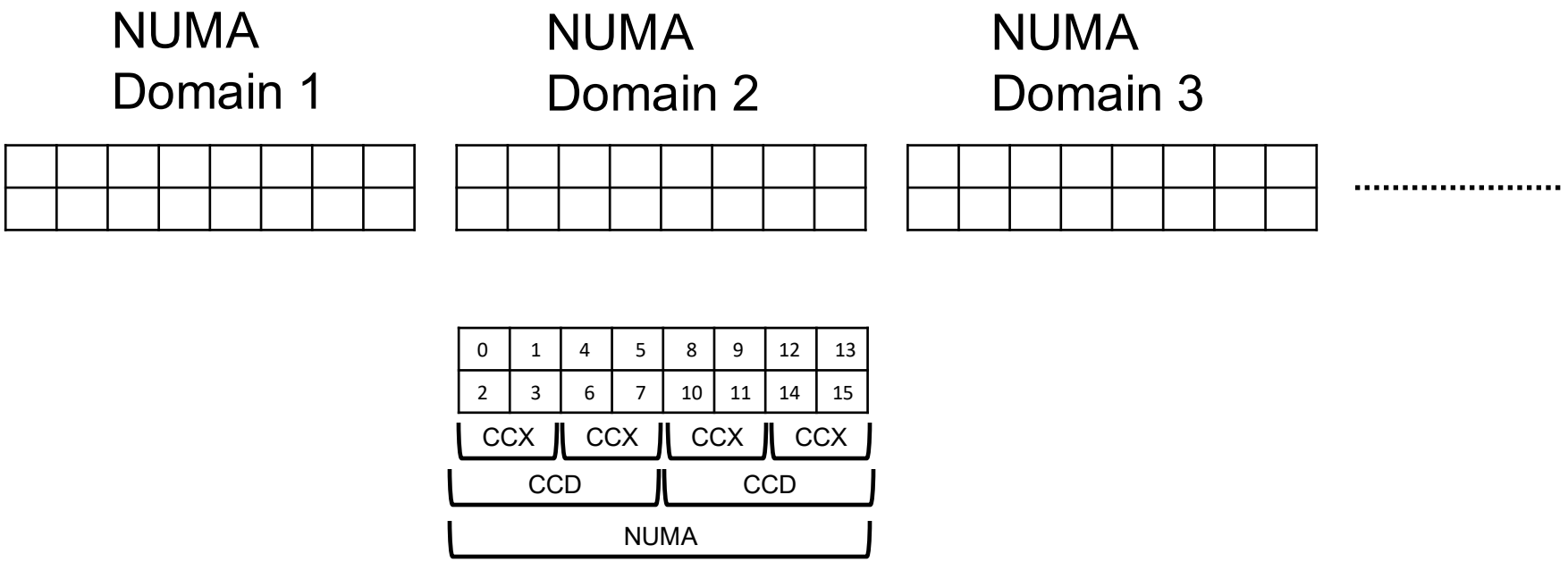
```
mpirun -np 8 --map-by ppr:4:node namd2 +setcpuaffinity +ppn 31
+commap 0,32,64,96 +pemap 1-31,33-63,65-95,97-127 stmv.namd
```

ibrun and affinity options

- Basic usage
  - ibrun ./executable <executable_opions>
- With affinity
  - ibrun affinity <hints> ./executable <executable_opions>
- Affinity options
  - **scatter**: scatters the ranks across all numa domains in a cyclic manner
  - **scatter-ccd**: scatters the ranks across all AMD CCD domains in a cyclic manner
  - **scatter-ccx**: scatters the ranks across all AMD CCX domains in a cyclic manner
  - **scatter blk <blk_size>**: scatters the ranks across all numa domains in a cyclic manner, but with 'blk_size' (1-16) consecutive ranks packed into a single numa domain
  - **scatter-ccd blk <blk_size>**: scatters the ranks across AMD CCD domains in a cyclic manner, but with 'blk_size' (1-8) consecutive ranks packed into a single CCD domain
  - **scatter-ccx blk <blk_size>**: scatters the ranks across AMD CCX domains in a cyclic manner, but with 'blk_size' (1-4) consecutive ranks packed into a single CCX domain

NOTE: valid blk_sizes depend on the **cpus-per-task a**nd the **domain type** (numa, CCD, CCX). 'blk' is optional and is set to '1' by default

Guide for Layout Diagrams (for upcoming slides)



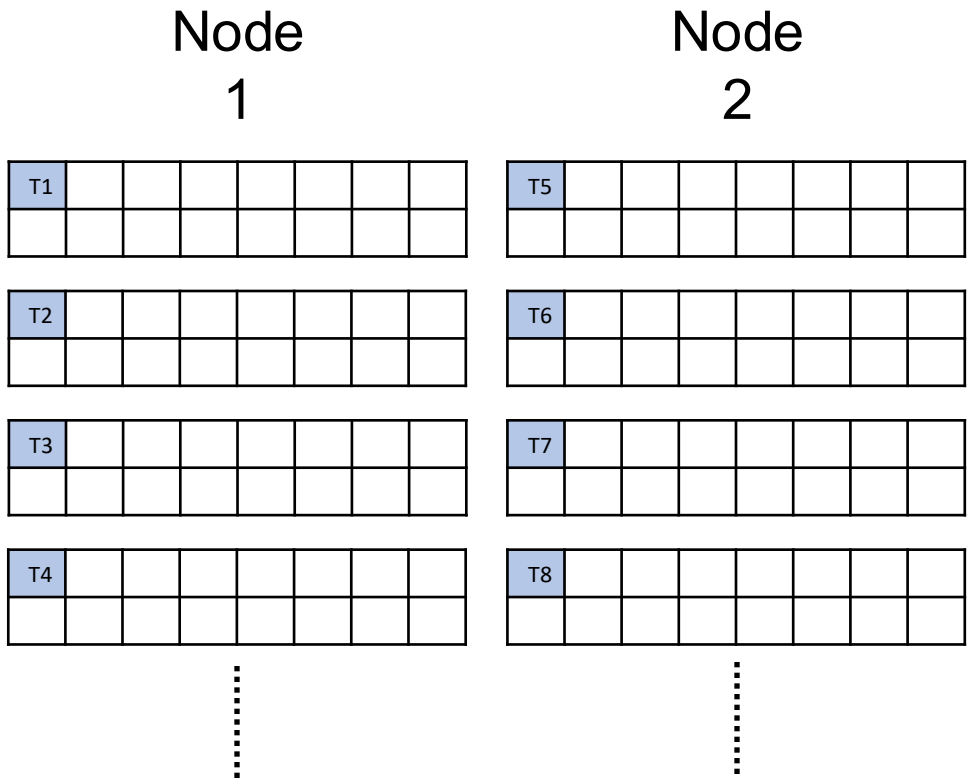
Example ibrun options and layouts

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -N 2
#SBATCH -A <ACCT>
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=4
#SBATCH -t 00:20:00

### Expanse modules
module reset
module load cpu/0.15.4
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
```

ibrun ./hy-gcc-openmpi.exe

(same as srun –n 8 ./hy-gcc-openmpi.exe)

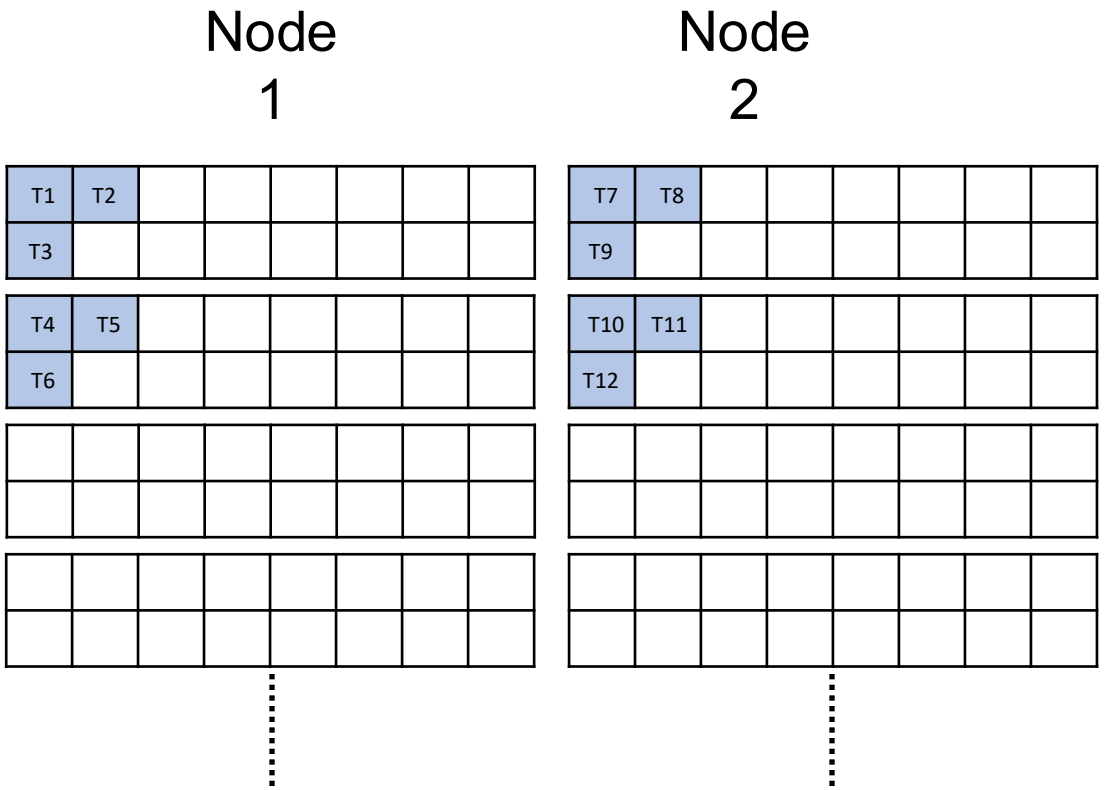


Example ibrun options and layouts

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -N 2
#SBATCH -A <ACCT>
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=6
#SBATCH -t 00:20:00

### Expanse modules
module reset
module load cpu/0.15.4
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
```

ibrun affinity scatter blk 3 ./hy-gcc-openmpi.exe

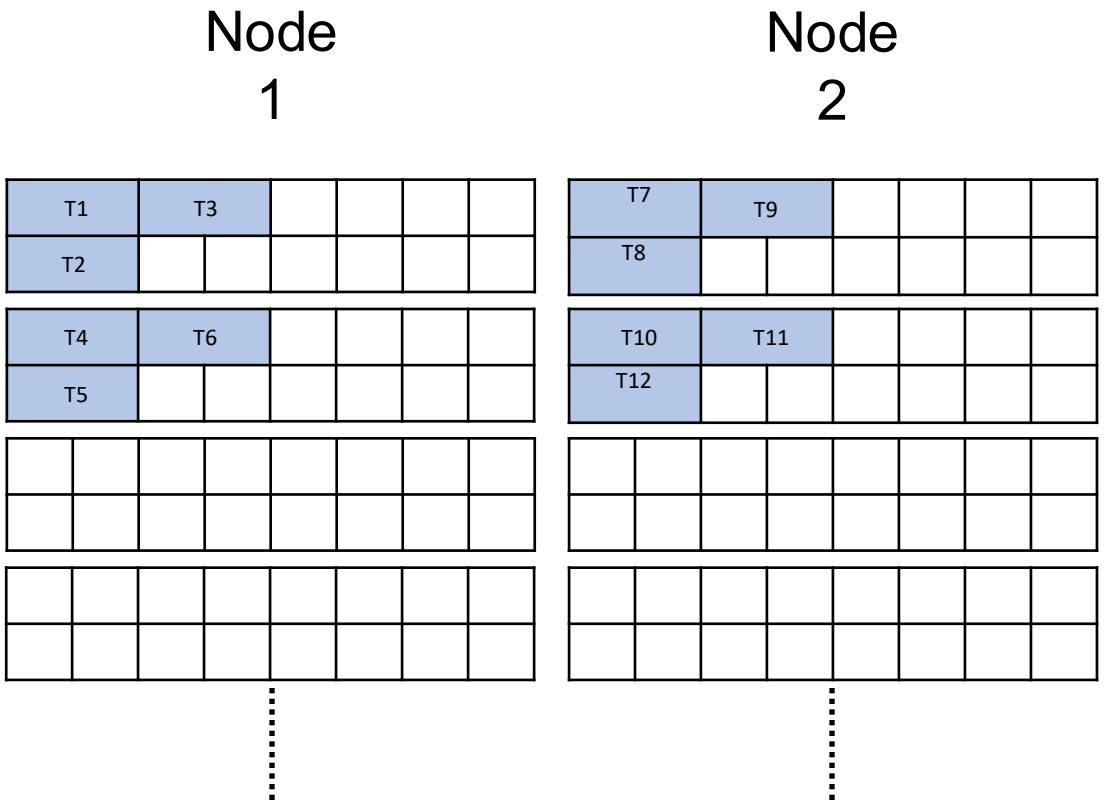


Example ibrun options and layouts

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -N 2
#SBATCH -A <ACCT>
#SBATCH --cpus-per-task=2
#SBATCH --ntasks-per-node=6
#SBATCH -t 00:20:00

### Expanse modules
module reset
module load cpu/0.15.4
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
```

ibrun affinity scatter blk 3 ./hy-gcc-openmpi.exe

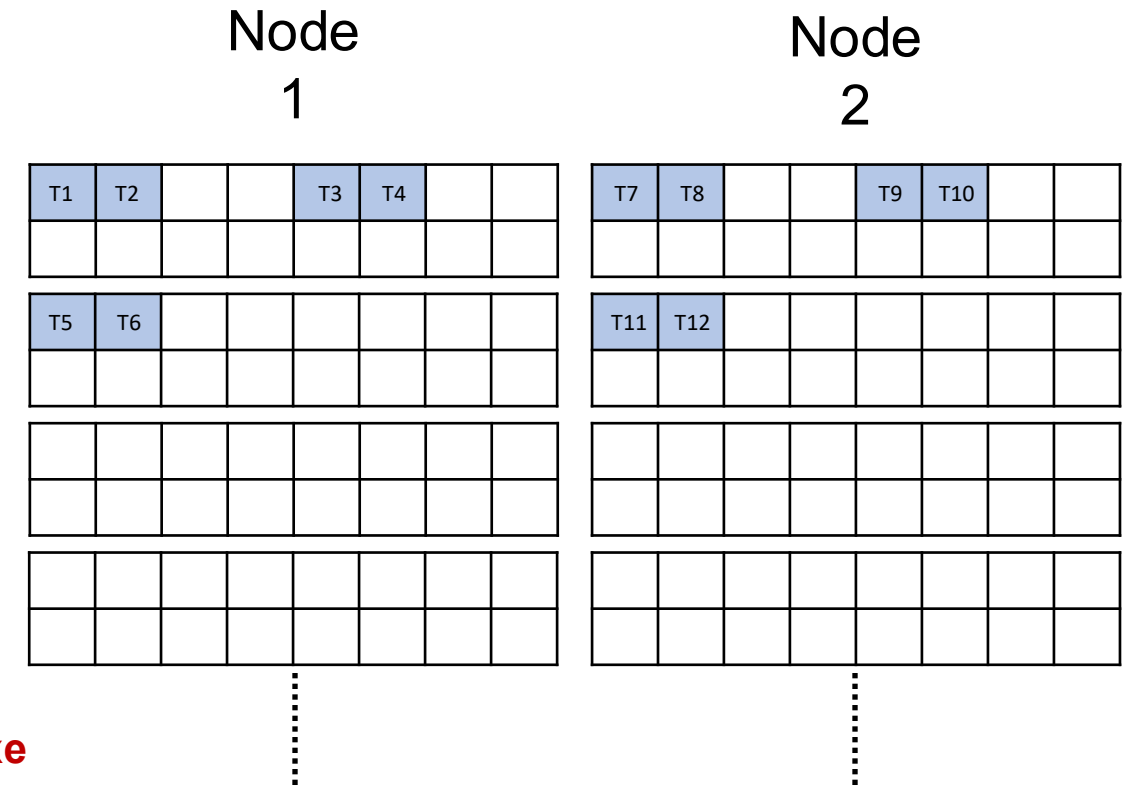


Example ibrun options and layouts

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -N 2
#SBATCH -A <ACCT>
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=6
#SBATCH -t 00:20:00

### Expanse modules
module reset
module load cpu/0.15.4
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
```

ibrun affinity scatter-ccd blk 2 ./hy-gcc-openmpi.exe

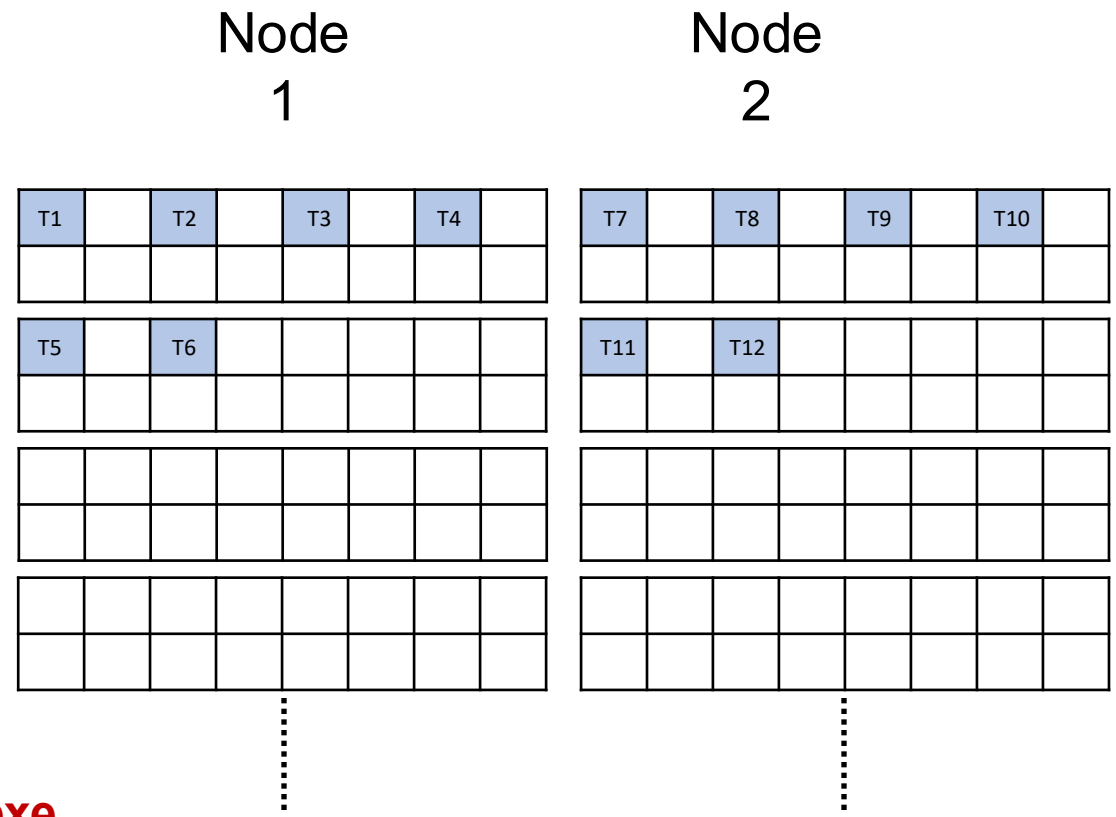


Example ibrun options and layouts

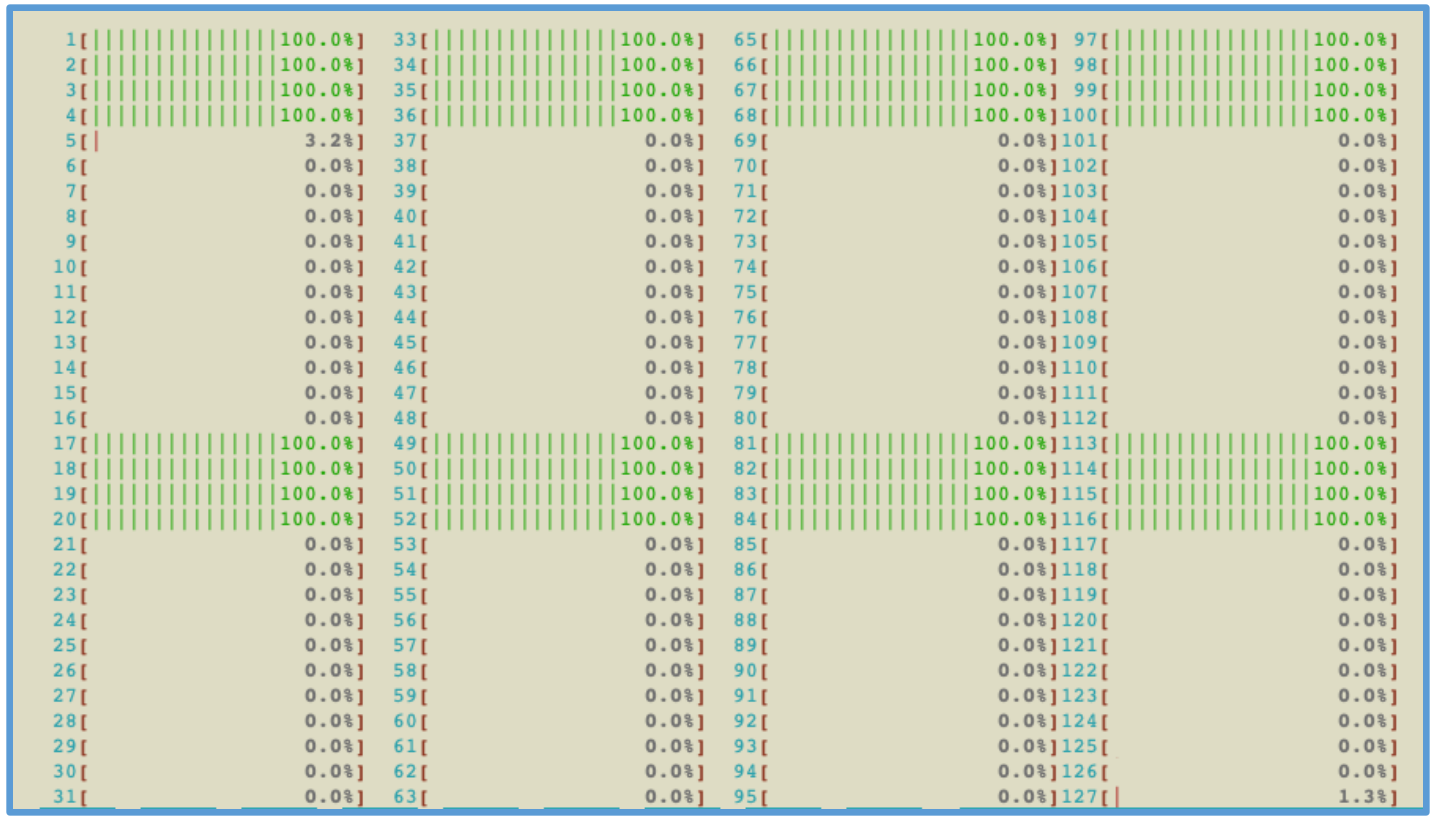
```
#!/bin/bash
#SBATCH -p compute
#SBATCH -N 2
#SBATCH -A <ACCT>
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=6
#SBATCH -t 00:20:00

### Expanse modules
module reset
module load cpu/0.15.4
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
```

ibrun affinity scatter-ccx ./hy-gcc-openmpi.exe



Snapshot of task layout with wrong layout

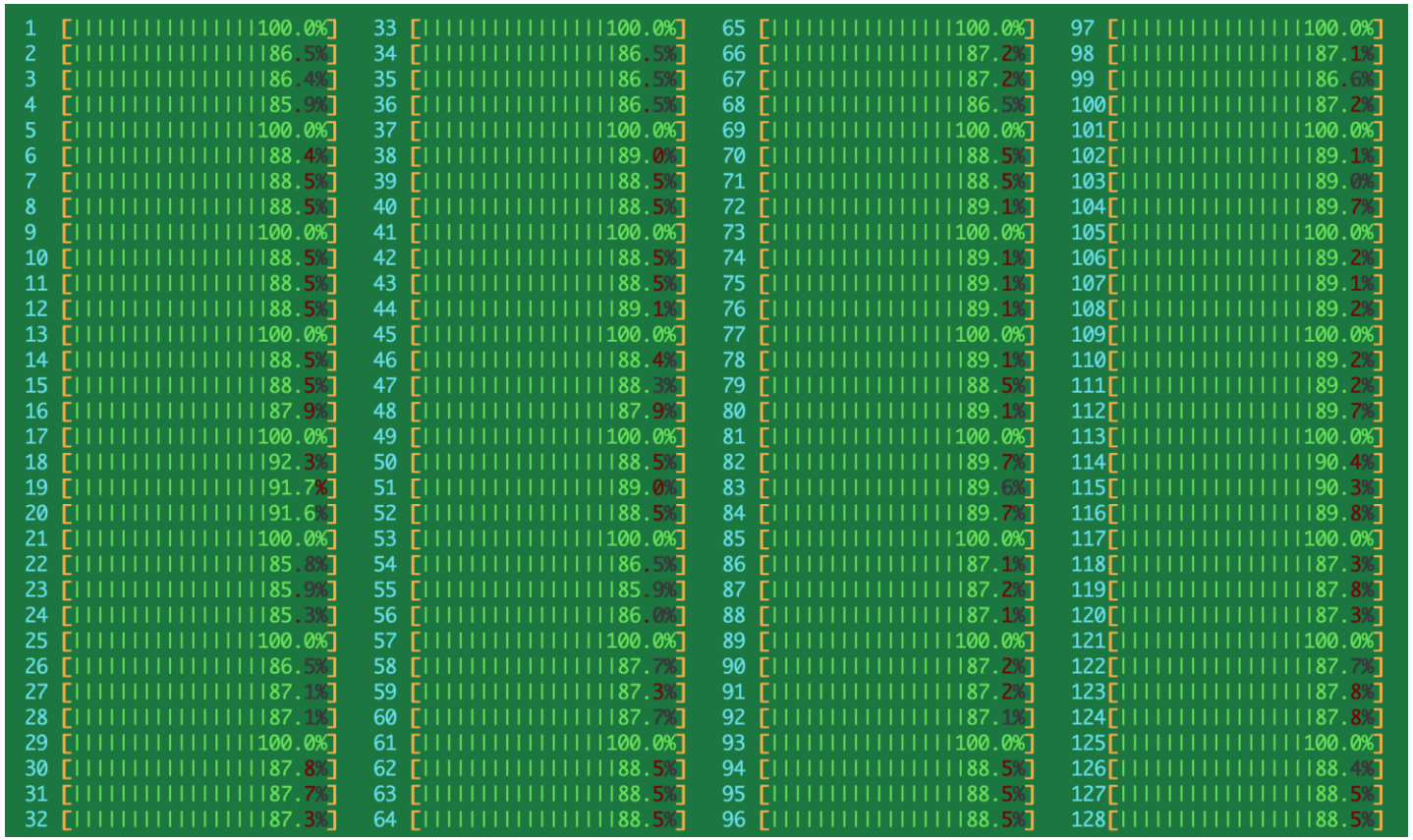



```
3296452 xhpl            mahidhar    3205092 R 75.4 00:00:59  19
3296452 xhpl            mahidhar    3205092 S  9.3 00:00:06  19
3296452 xhpl            mahidhar    3205092 S  9.3 00:00:06  19
3296452 xhpl            mahidhar    3205092 S  9.3 00:00:06  19
3296453 xhpl            mahidhar    3205092 R 75.7 00:00:59  35
3296453 xhpl            mahidhar    3205092 S  9.2 00:00:06  35
3296453 xhpl            mahidhar    3205092 S  9.2 00:00:06  35
3296453 xhpl            mahidhar    3205092 S  9.2 00:00:05  35
3296454 xhpl            mahidhar    3205092 R 75.5 00:00:59  51
```

Snapshot of task layout with scatter-ccx option

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=4

ibrun affinity scatter-ccx $XHPL
```



Snapshot of task layout with scatter-ccx option

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=4

ibrun affinity scatter-ccx $XHPL
```

```
82667 xhpl            mahidhar    2526976 R 98.4 00:02:10   0
82667 xhpl            mahidhar    2526976 S  0.0 00:00:00   0
82667 xhpl            mahidhar    2526976 S  0.8 00:00:01   0
82667 xhpl            mahidhar    2526976 R 87.9 00:01:49   1
82667 xhpl            mahidhar    2526976 R 87.9 00:01:49   2
82667 xhpl            mahidhar    2526976 R 87.9 00:01:49   3
82668 xhpl            mahidhar    2527544 R 98.4 00:02:09  40
82668 xhpl            mahidhar    2527544 S  0.0 00:00:00  40
82668 xhpl            mahidhar    2527544 S  0.9 00:00:01  40
82668 xhpl            mahidhar    2527544 R 88.3 00:01:49  41
82668 xhpl            mahidhar    2527544 R 88.3 00:01:49  42
82668 xhpl            mahidhar    2527544 R 88.3 00:01:49  43
82669 xhpl            mahidhar    2527532 R 98.3 00:02:09   4
82669 xhpl            mahidhar    2527532 S  0.0 00:00:00   4
82669 xhpl            mahidhar    2527532 S  0.7 00:00:00   4
82669 xhpl            mahidhar    2527532 R 87.7 00:01:48   5
82669 xhpl            mahidhar    2527532 R 87.7 00:01:48   6
82669 xhpl            mahidhar    2527532 R 87.7 00:01:48   7
```

slurm-aff-prod script for MPI/Pthreads codes

- The **slurm-aff-prod** script is used to bind Pthreads after MPI job launch.
- Can be used by wrapping binary based on name.

```
#!/bin/bash
#SBATCH -p shared
#SBATCH -N 1
#SBATCH --ntasks-per-node=10
#SBATCH --cpus-per-task=4
#SBATCH --mem=77G
#SBATCH -t 00:10:00
#SBATCH -J A76BE.GTRGAMMA.mpi10pt4NautoMRExfa
#SBATCH -o A76BE.GTRGAMMA.mpi10pt4NautoMRExfa.%j.%N.out
#SBATCH -e A76BE.GTRGAMMA.mpi10pt4NautoMRExfa.%j.%N.err
#SBATCH -A use300

export NP=$SLURM_TASKS_PER_NODE
export THREADS=$SLURM_CPUS_PER_TASK

rm RAx*

export AFFINITY_INFO=0
export AFFINITY_DEBUG=0

srun --mpi=pmi2 -n $NP ./raxmlHPC-HYBRID_8.2.12_expanse -s ./A76BE.txt -n A76BE.GTRGAMMA.mpi10pt4NautoMRExfa -m GTRGAMMA -N a
utoMRE -p 12345 -x 12345 -f a
```

```
#!/bin/sh

# This is for running on EXPANSE

#source $HOME/.bashrc

module reset
module load sdsc
module load gcc/10.2.0
module load openmpi/4.0.4
module load raxml/8.2.12
module load slurm

EXE='raxmlHPC-HYBRID-AVX'
slurm-aff-prod-test $EXE &

echo "running:"
echo " $EXE -T ${THREADS} $*"

$EXE -T ${THREADS} $*
```

Summary of Binding Options on Expanse

- AMD Processor on Expanse has 4 NUMA domains with 16 cores each.
- 8 Core Complex Dies (CCDs) per processor, with 2 Core Complexes (CCXs) per CCD. Four cores in a CCX share L3 cache.
- For hybrid MPI/OpenMP and MPI/Pthreads codes it is important to lay out tasks correctly and binding is important for performance.
- **ibrun, affinity, and slurm-aff-prod** scripts available to make it easier to lay out and bind tasks.
- Tools are being updated so feedback is encouraged!

MPI and OpenMP References

- Excellent tutorials from LLNL (some of today's content was derived from here):
  - https://hpc-tutorials.llnl.gov/mpi/
  - https://hpc.llnl.gov/sites/default/files/DavidCronkSlides.pdf
  - https://hpc-tutorials.llnl.gov/openmp
- MPI for Python:
  - https://mpi4py.readthedocs.io/en/stable/
- OpenMPI User Guide:
  - https://www.open-mpi.org/doc/current/
- MVAPICH2 User Guide:
  - http://mvapich.cse.ohio-state.edu/userguide/