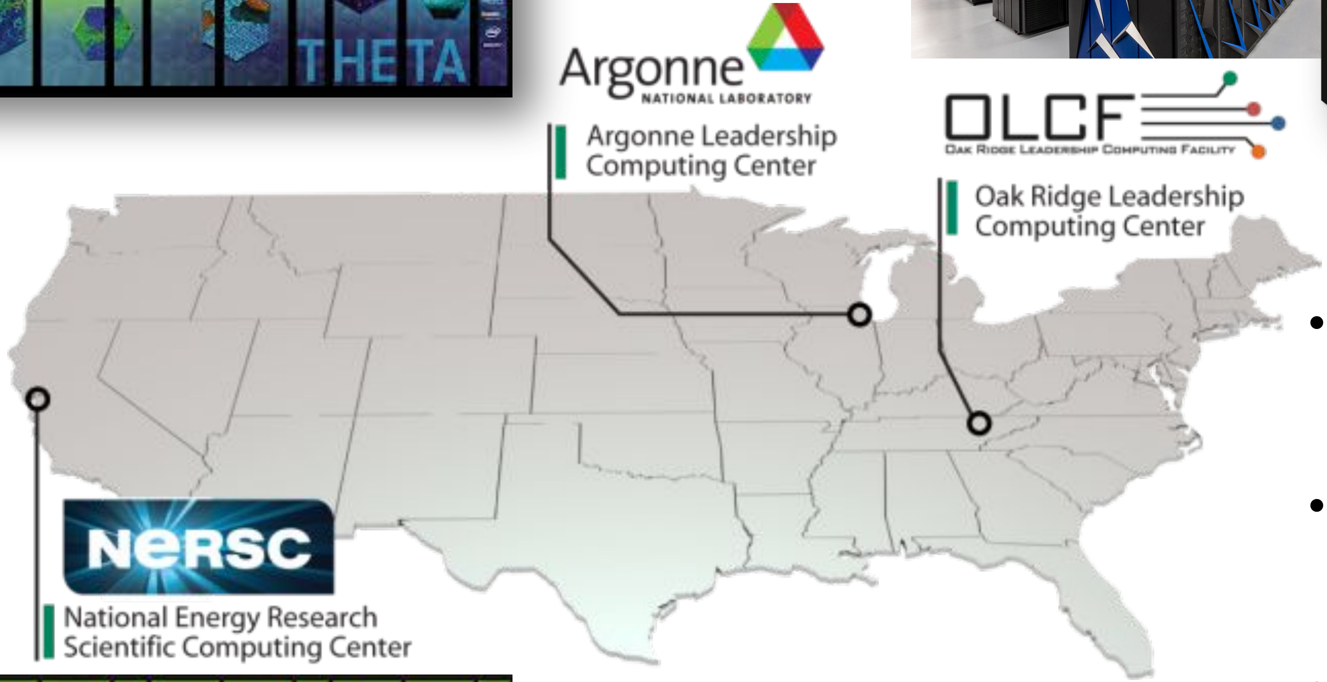
Parallel Programming Models

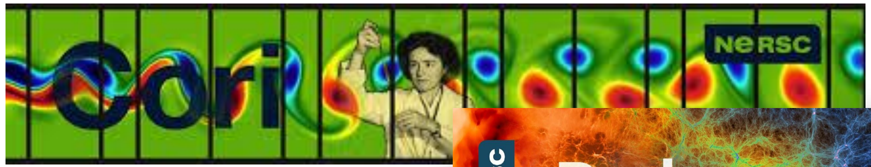
OAK RIDGE National Laboratory
U.S. DEPARTMENT OF ENERGY
Hewlett Packard Enterprise
AMD
CRAY
FRONTIER

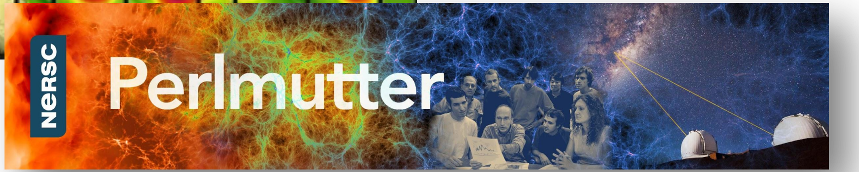
Suzanne Parete-Koon
Oak Ridge Leadership Computing Facility
Oak Ridge National Lab.

Time (all times EDT)	Topic	Presenter
1:00 PM	Welcome Back	Suzanne Parete-Koon, HPC Engineer
1:02 PM	What is HPC	Trey White, Computational Scientist
1:15 PM	Odo Overview	Subil Abraham, HPC Engineer
1:30 PM	Hands-on Session 1 Workflow and Job Launcher	
2:00 PM	Parallel Programming Models and Certificate requirements	Suzanne Parete-Koon, HPC Engineer
2:55 PM	Hands-on Session 2 Parallel Programming models	
3:00	Intro to Machine Learning	Michael Sandoval, HPC Engineer
3:15	Hands-on Session 3 Python and Machine Learning exercises	
4:00	Final Check-in and Questions	All

We are one of the DOE's Office of Science computational user facilities


Argonne NATIONAL LABORATORY
Argonne Leadership Computing Center
OLCF OAK RIDGE LEADERSHIP COMPUTING FACILITY
Oak Ridge Leadership Computing Center
NERSC
National Energy Research Scientific Computing Center





- DOE is leader in open high-performance computing
- Provide the world's most powerful computational tools for open science
- Access is free to researchers who publish
- Boost US competitiveness
- Attract the best and brightest researchers

Certificate Instructions

See: https://github.com/olcf/hands-on-with-frontier/tree/master/challenges

Summary:
Each challenge is a stand-alone, self-guided tutorial that includes a README.md file to walk you through the content. The first challenge is to complete Access_Frontier_and_Clone_Repo, which will show you how to clone this repository.

To complete the requirements for your certificate, pick any 7 of the challenges in any of the sections below. (High School students need only 3). For all challenges other than the first one, which we will do together, you will turn in your work by entering the path to one of the challenge's output file in this google sheet as directed by your instructor.

Time to work on Getting Started Challenges

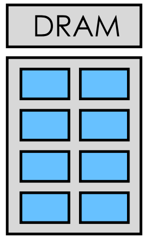
You have 30 minutes. Raise your virtual hand when you are done with two or more of the following challenges :

1. Basic_Unix_Vim (Only do this one if you have no experience with Unix or a text editor)
2. Basic_Workflow
3. Srun_Job_Launcher
4. Password_in_a_Haystack

Parallel Programming Models

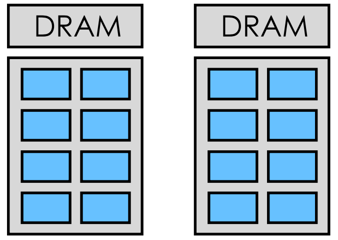
A =

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



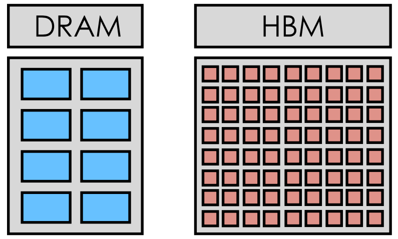
Shared-memory models

- E.g., OpenMP
  - All process threads can access same memory (single-node)



Distributed-memory models

- E.g., Message Passing Interface (MPI)
  - All processes (i.e., MPI ranks) have access to their own memory (multi-node)



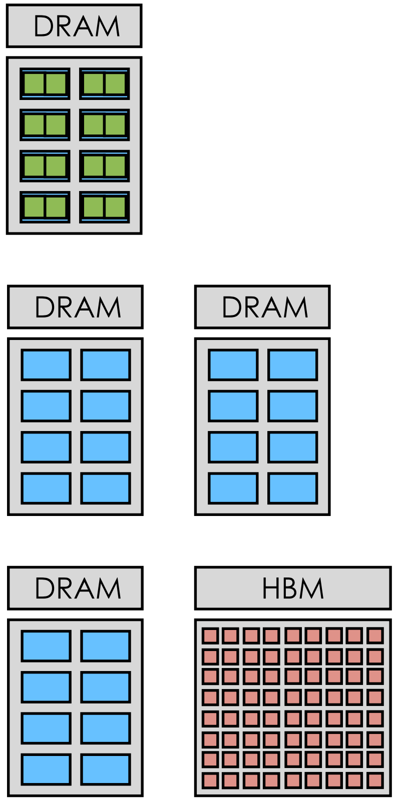
GPU

- CUDA, HIP
- OpenACC, OpenMP offload (directive-based models)
- Kokkos (portability)

Parallel Programming Models

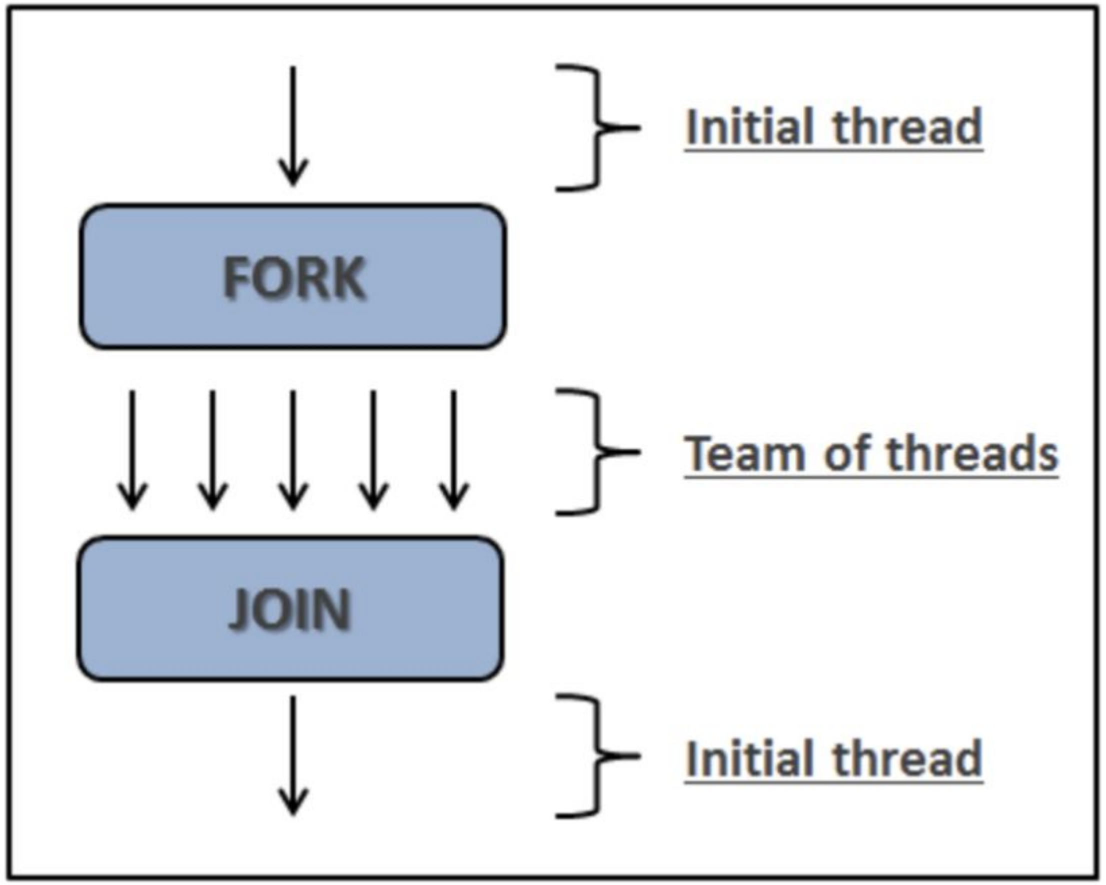
A =

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



Shared-memory models

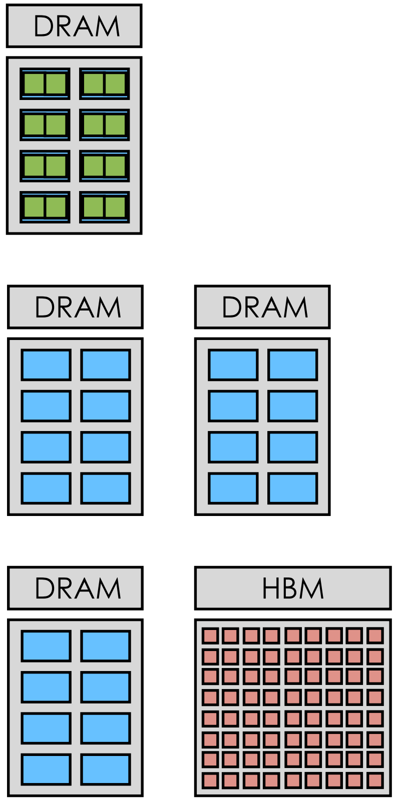
- E.g., OpenMP
- [illegible]



Parallel Programming Models

A =

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



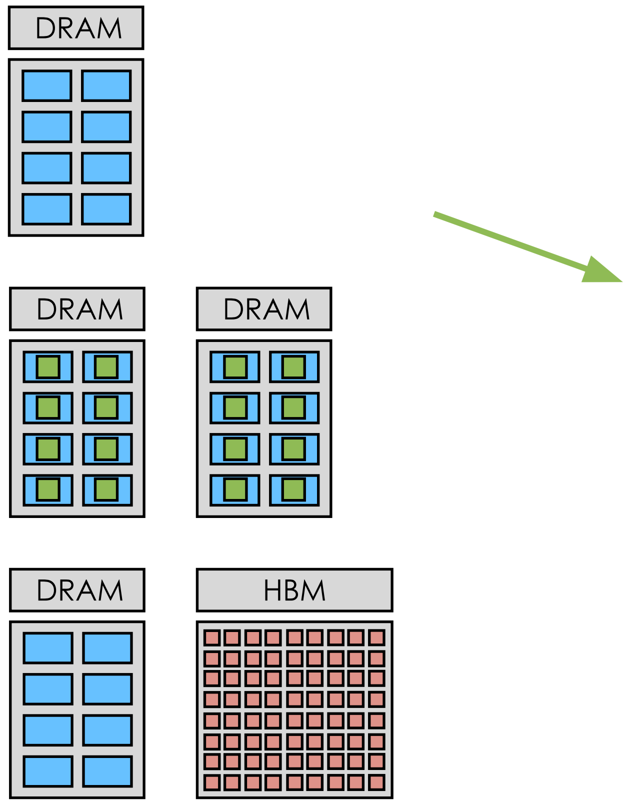
Shared-memory models

- E.g., OpenMP

```
#pragma omp parallel default(none) shared(A, B)
    {
        #pragma omp for
         for(int i=0; i<N; i++){
           B[i] = A[i]*A[i]
         }
    }
```

Parallel Programming Models

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



Shared-memory models

- E.g., OpenMP
    - All process threads can access same memory (single-node)

Distributed-memory models

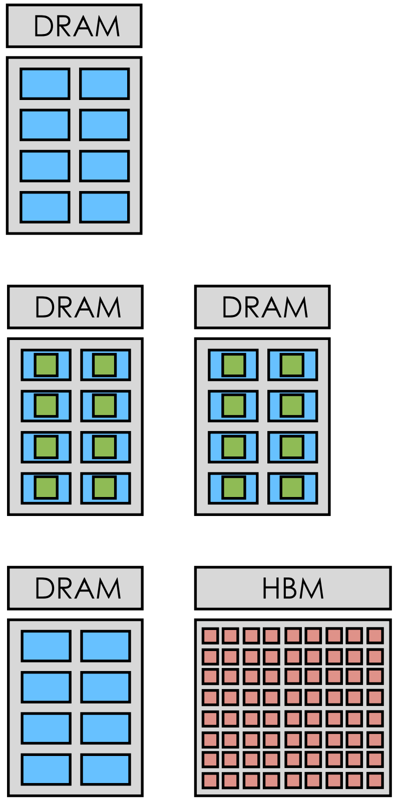
- E.g., Message Passing Interface (MPI)
    - All processes (i.e., MPI ranks) have access to their own memory (multi-node)

GPU

- CUDA, HIP
- OpenACC, OpenMP offload (directive-based models)
- Kokkos (portability)

Parallel Programming Models

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



. . .

MPI_Init()

#sets up communication

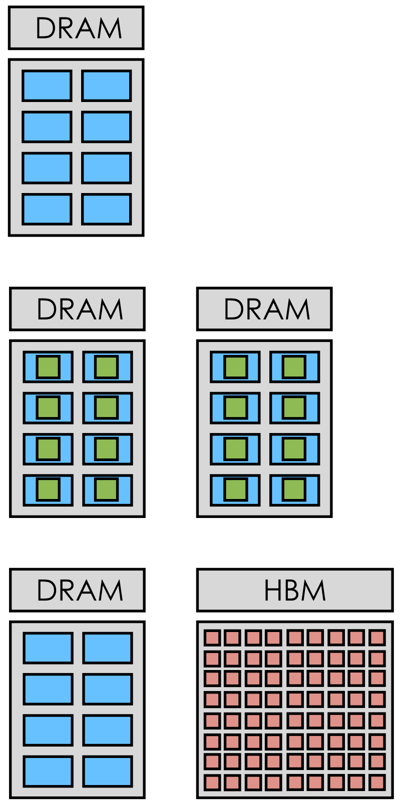
Code that you want to run on many processors

MPI_Finalize

. . .

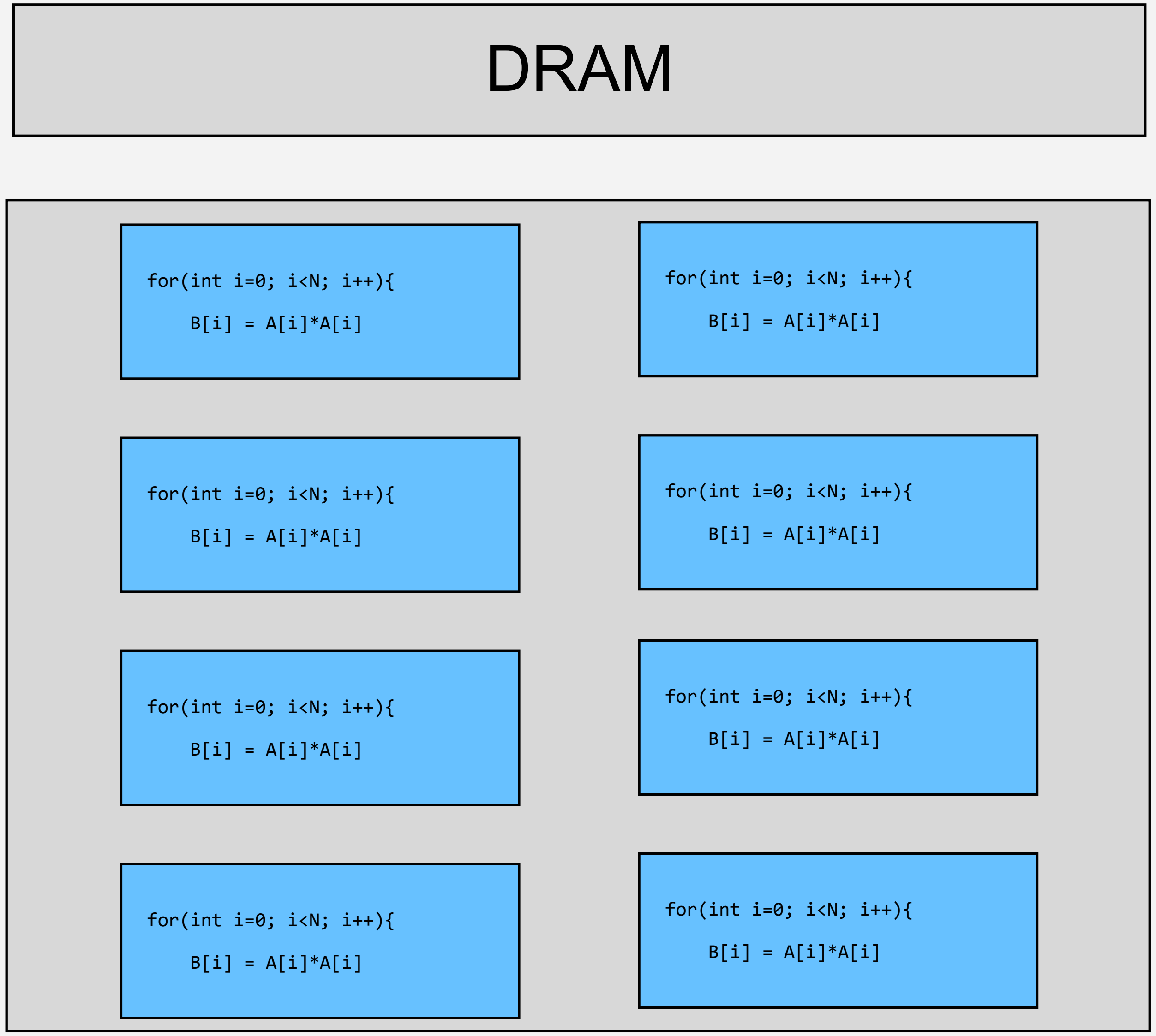
Parallel Programming Models

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



Putting that loop
In an MPI region
will not by itself
divide the iterations
among the processes.

The programmer
needs to supply the
logic for how to
distribute the work.



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

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

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

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

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

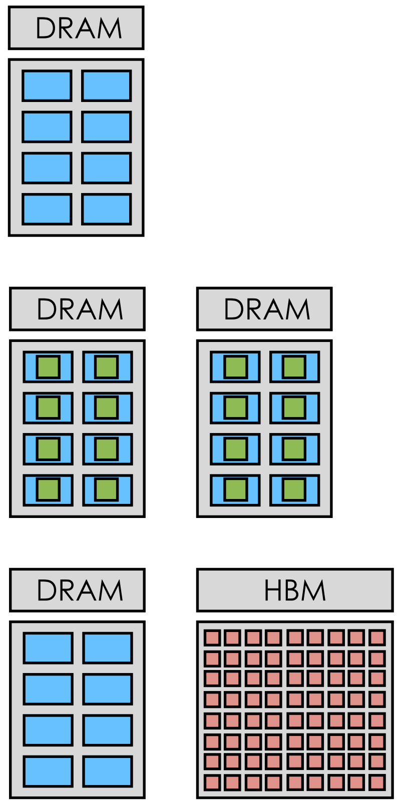
```
for(int i=0; i<N; i++){
    B[i] = A[i]*A[i]
```

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

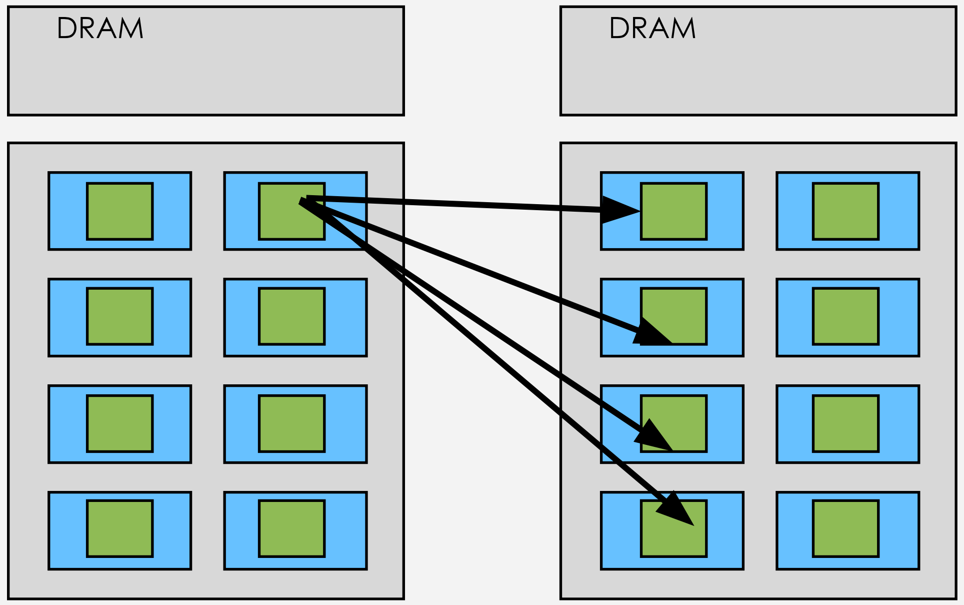
```
for(int i=0; i<N; i++){
    B[i] = A[i]*A[i]
```

Parallel Programming Models

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

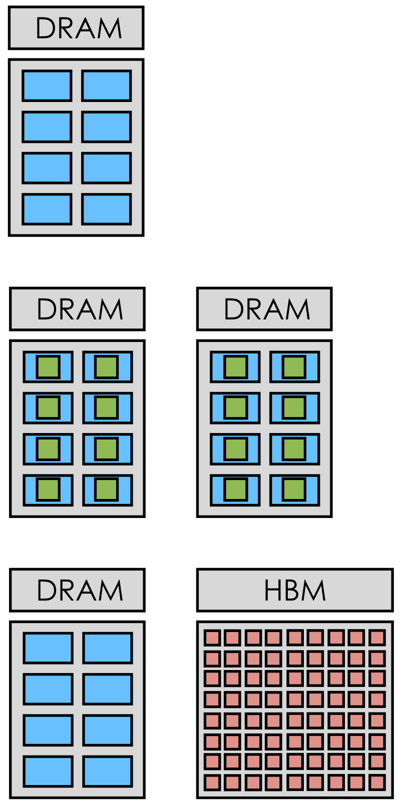


Processor can communicate collectively, for example, by broadcasting data from one processor to many.

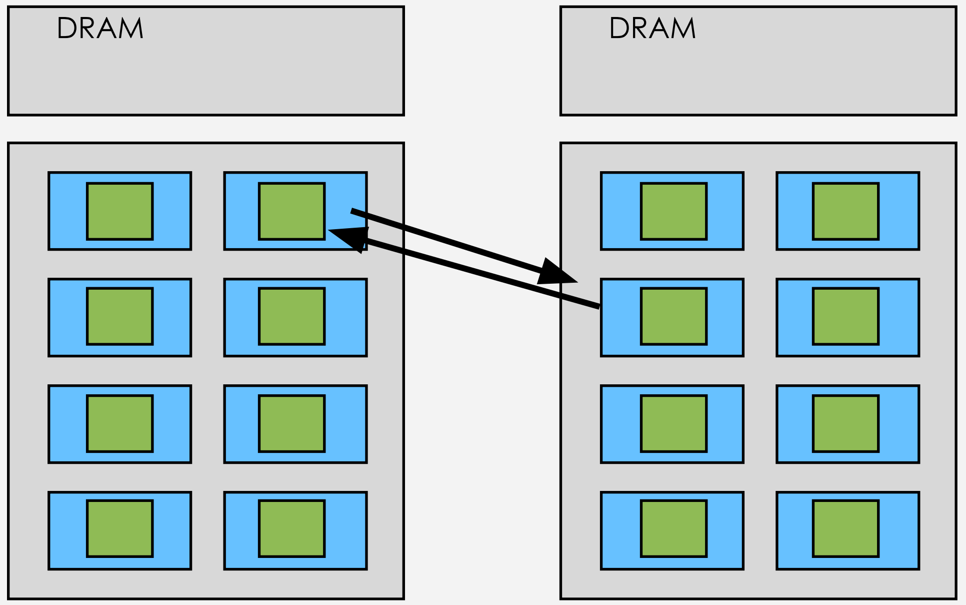


Parallel Programming Models

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

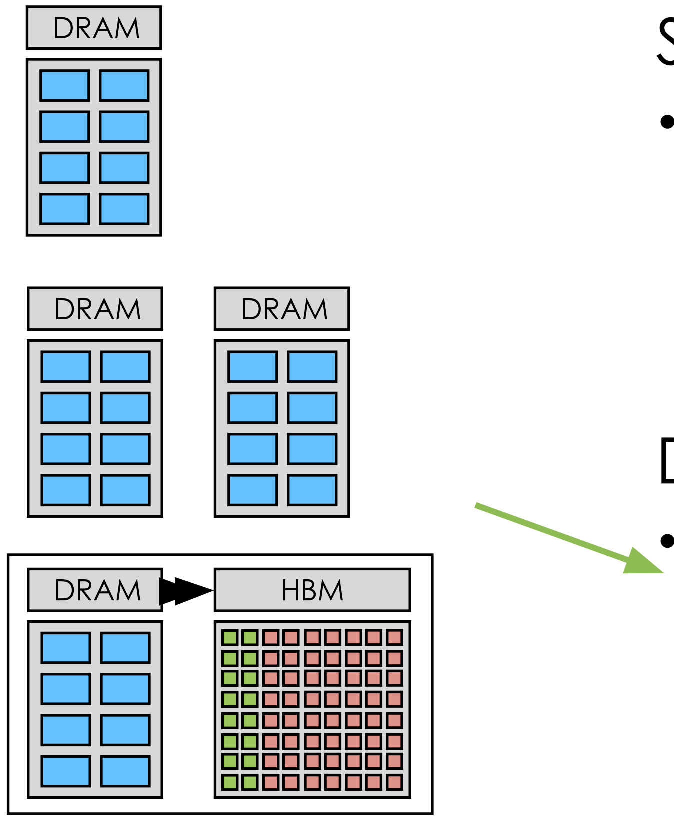


Communication of data can flow from one process to another in a point to point communication.



Parallel Programming Models

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



Shared-memory models

- E.g., OpenMP
    - All process threads can access same memory (single-node)

Distributed-memory models

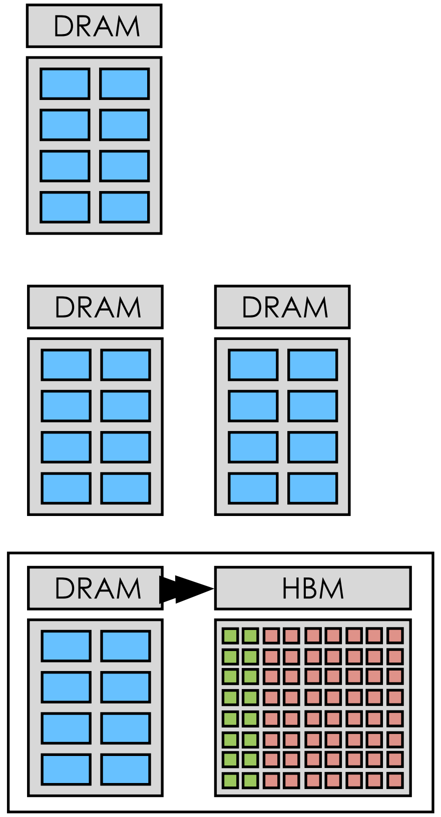
- E.g., Message Passing Interface (MPI)
    - All processes (i.e., MPI ranks) have access to their own memory (multi-node)

GPU

- CUDA, HIP
- OpenACC, OpenMP offload (directive-based models)
- Kokkos (portability)

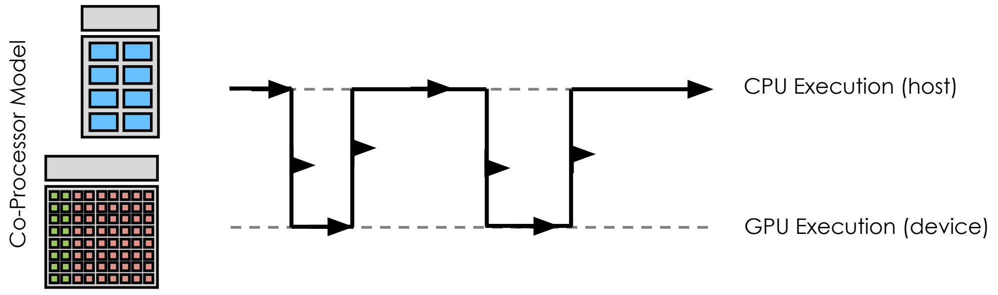
Parallel Programming Models

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



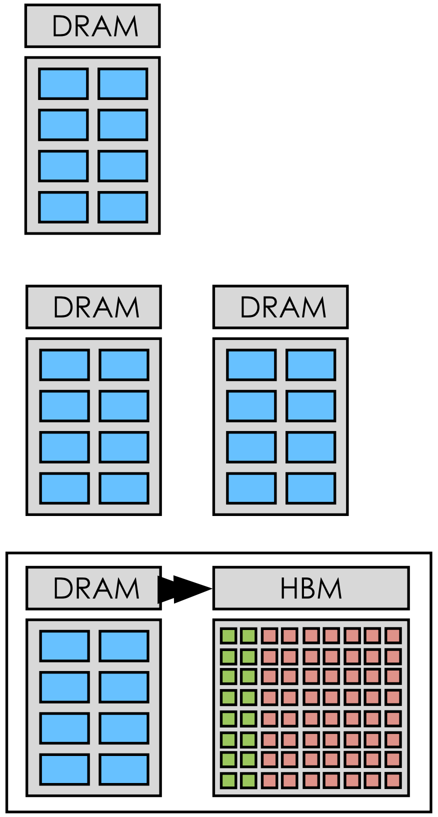
GPU Programming Models

- With GPU programming, work is split between the CPU (host) and GPU (device).
- Mostly bottleneck regions of applications are ported to the GPU for optimization.
- Applications are initially profiled to determine compute intensive regions which are then offloaded to the device.



Parallel Programming Models

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



Frontier Support for Parallel Programming Models

Parallel Programming Model	Frontier Support	GNU Compiler	Cray Compiler	AMD Compiler
OpenMP	✓	✓	✓	✓
MPI	✓	✓	✓	✓
CUDA	x	x	x	x
OpenMP Offload	✓	✓	✓	✓
OpenACC	✓	✓	✓ *	x
HIP	✓	x	✓	✓
Kokkos	✓	✓	✓	✓

Challenges to work on in the next 30 minutes:

Parallel Programming Models (in C)

6. OpenMP_Basics
7. MPI_Basics
8. OpenMP_Offload (Frontier only)
9. GPU_Matrix_Multiply (Frontier only)

Raise your virtual hand when you have finished 1 or more of the following challenges.

Challenges to work on in the next 30 minutes:

Python exercies and Machine Learning

10. Python_Conda_Basics (Required before attemping any of the Python challenges below)
11. Python_Pytorch_Basics
12. Python_Galaxy_Evolution

Raise your virtual hand when you have finished 1 or more of the following challenges.

Questions?

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Questions?

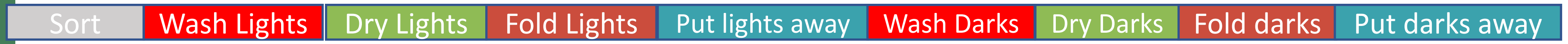
EXTRA SLIDES BELOW THIS POINT.

High Performance Computing

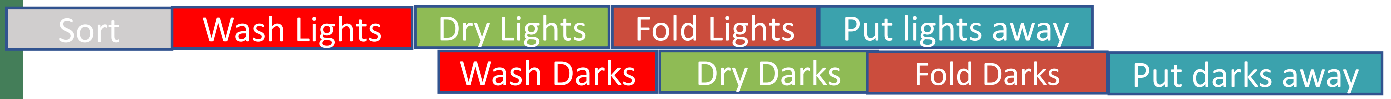
High Performance Computing (HPC) is about doing work efficiently in parallel.

Profiling and Optimizing your Laundry Day Workflow

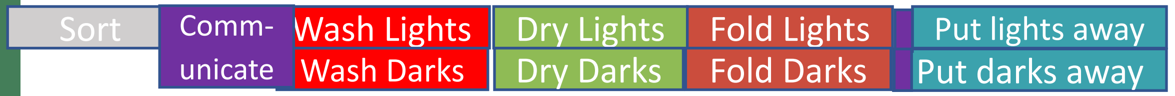
Just you and one washer and one dryer



Optimizing just you and one washer and one dryer

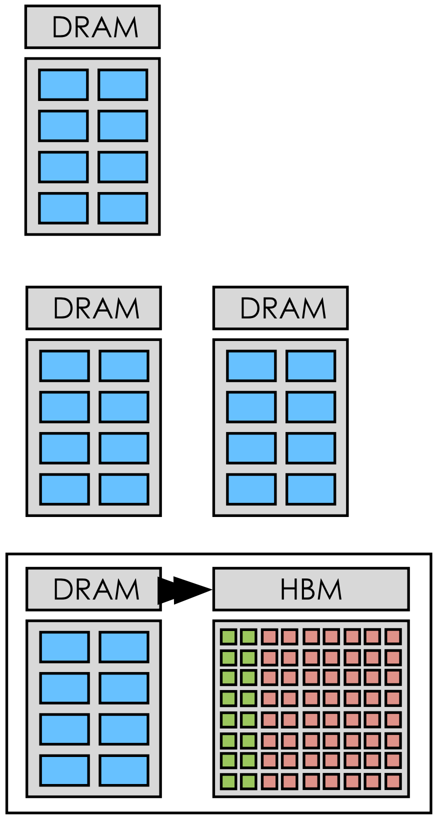


You your housemate, 2 washers, and 2 dryers



Parallel Programming Models

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



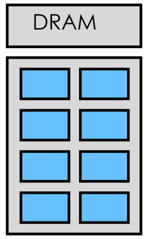
GPU Programming Models: CUDA

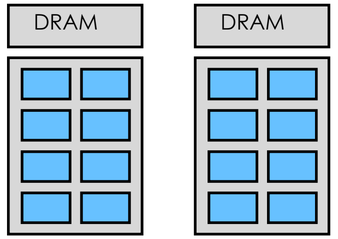
- `__global__` is a CUDA C++ keyword which indicates a function that
  - Runs on the device
  - Is called from the host code or other device code.

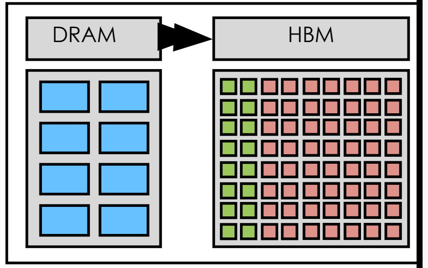
```
__global__ void mykernel (void)  {

    }
```

Parallel Programming Models: CUDA

```
for(int i=0; i<N; i++){

    B[i] = A[i]*A[i]
}
```



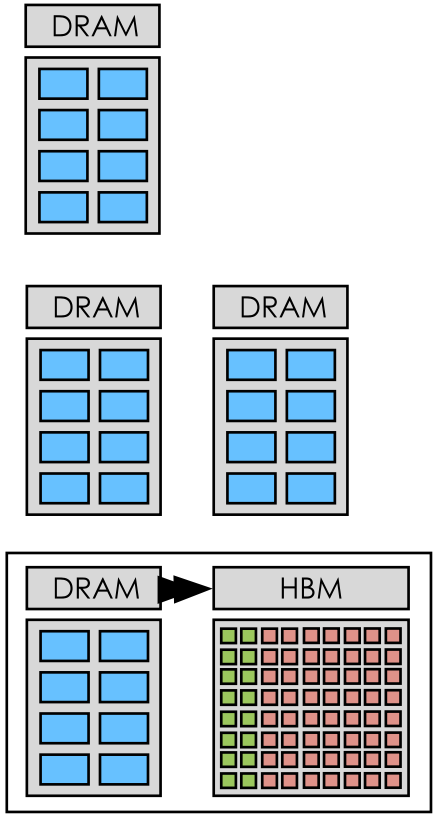




```
__global__ void mykernel (void)  {

         }
         #include <stdio.h>

         __global__
         void saxpy(int n, float a, float *x, float *y)
         {
           int i = blockIdx.x*blockDim.x + threadIdx.x;
           if (i < n) y[i] = a*x[i] + y[i];
         }

         int main(void)
         {
           int N = 1<<20;
           float *x, *y, *d_x, *d_y;
           x = (float*)malloc(N*sizeof(float));
           y = (float*)malloc(N*sizeof(float));

           cudaMalloc(&d_x, N*sizeof(float));
           cudaMalloc(&d_y, N*sizeof(float));

           for (int i = 0; i < N; i++) {
             x[i] = 1.0f;
             y[i] = 2.0f;
           }

           cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
           cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

           // Perform SAXPY on 1M elements
           saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

           cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

           float maxError = 0.0f;
           for (int i = 0; i < N; i++)
             maxError = max(maxError, abs(y[i]-4.0f));
           printf("Max error: %f\n", maxError);

           cudaFree(d_x);
           cudaFree(d_y);
           free(x);
           free(y);
         }
```

Parallel Programming Models

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



GPU Programming Models: OpenACC

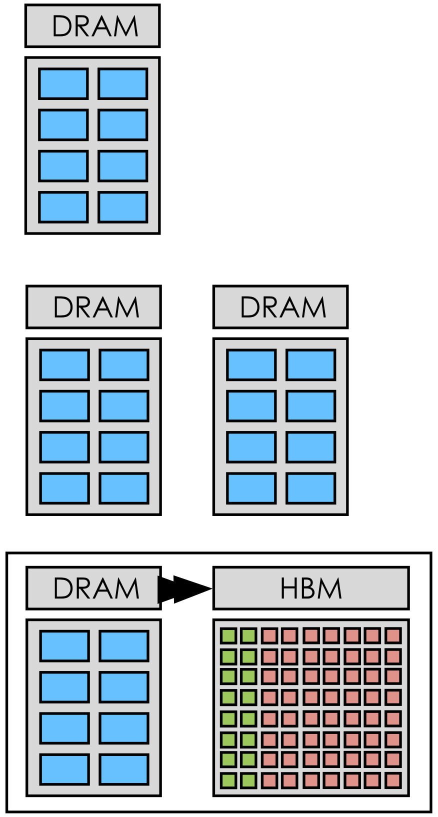
- **OpenACC is a directive based parallel programming model for GPU offloading**
- **The directive `!$acc parallel loop` indicates the device code which instructs the compiler to offload to GPU**

```
!$acc parallel loop gang default(present)
 do i=1,N

    B(i) = A(i)*A(i)

  end do
```

Parallel Programming Models

A =

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



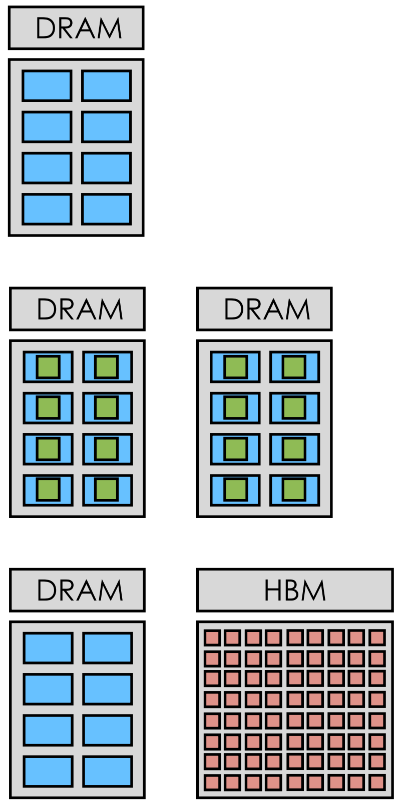
GPU Programming Models: OpenMP Offload

- **OpenMP offload is another one of directive based parallel programming models for GPU offloading.**
- **It is similar to the OpenACC offloading paradigm.**
- **As in OpenACC, the `!$omp target teams distribute parallel do` construct indicates the region of code to offload to the device.**

```
!$omp target teams distribute parallel do

do i=1,N

     B(i) = A(i)*A(i)

end do
!$omp end target teams distribute parallel do
```

Parallel Programming Models

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



There are 4 things to know about MPI

1. It is a standard for a message passing library. The standard describes how each library call needs to function. The functions, among other things, allow you to move data between processors with different pools of memory.
2. It sets up processes on each processing element (CPU) and gives them an identifying rank, so messages with data can be sent between them.
3. All processes have their own memory and data.
4. MPI executes the same code on all processes. So for example, if you put a loop that multiplies two vectors inside an MPI region *as is*, it will multiply *all* of the elements of those two vectors in each process. If you want each processor to handle one specific part of that loop, you have to provide the logic in the code for how the iterations of the loop are divided between the processes.