# STANDARD LANGUAGES

N-WAYS GPU BOOTCAMP

# STANDARD LANGUAGES

## What to expect?

- C++ , Fortran ISO standard brief
- C++ std::par , Fortran DO-Concurrent API
- Known limitations

# BRIEF HISTORY

- Historically, accelerating your code with GPUs has not been possible in Standard C++/Fortran without using language extensions or additional libraries:
    - CUDA C++ requires the use of __host__ and __device__ attributes on functions and the <<<>>> syntax for GPU kernel launches.
    - OpenACC uses #pragmas to control GPU acceleration
- What if you could take your Standard C++ or Fortran code and accelerate on a GPU?

# QUICK BACKGROUND

## C++ STL Containers

- One driving feature of C++ are its templates and the STL library. C++11 is further pushing these ideas and shows no sign of slowing.
- C++ templates are probably most widely used through the STL containers.
  - std::vector, std::string, std::map, std::list, etc...
- Besides the OO features and convenience, these containers are designed to rise-above basic C pointers, providing more safety from memory violations, while maintaining the bare-metal performance.
- For example std::vector … the vector template is designed to replace C's arrays.

```
std::vector<int> my_ints(4, 100);   // four ints with value 100
```

# STD::PAR

## What is std::par?

- Use standard C++ constructs to make code run parallel on heterogeneous hardware
- C++11 introduced a memory model, concurrent execution model, and concurrency library, providing a standard way to take advantage of multicore processors
- The C++17 Standard introduced higher-level parallelism features that allow users to request parallelization of Standard Library algorithms.

## Advantage:

- No language extensions, pragmas, directives, or non-standard libraries
- Write Standard C++, which is portable to other compilers and systems
- Compiler automatically accelerates code with high-performance NVIDIA GPUs and hence less time

# STD::PAR

## Parallelism in Standard C++

- Parallelism is expressed by adding an execution policy as the first parameter to any algorithm that supports execution policies
- Most of the existing Standard C++ algorithms were enhanced to support execution policies

Execution policies can be applied to most standard algorithms

| | | |
|---|---|---|
| • std::execution::seq | = sequential: | Sequential execution. No parallelism is allowed. |
| • std::execution::par | = parallel: | Parallel execution on one or more threads. |
| • std::execution::par_unseq | = parallel + vectorized: | Parallel execution on one or more threads, with each thread possibly vectorized. |

# C++17 PARALLEL ALGORITHMS

## Example

C++98: std::sort(c.begin(), c.end());

C++17: std::sort(std::execution::par, c.begin(), c.end());

## BUILD AND RUN THE CODE

# NVIDIA HPC SDK

- Comprehensive suite of compilers, libraries, and tools used to GPU accelerate HPC modeling and simulation application
- The NVIDIA HPC SDK includes the new NVIDIA HPC C++ compiler, NVC++. NVC++ supports C++17, C++ Standard Parallelism (stdpar) for CPU and GPU
- NVC++ can compile Standard C++ algorithms with the parallel execution policies std::execution::par for execution on NVIDIA GPUs.
- An NVC++ command-line option, -stdpar, is used to enable GPU-accelerated C++ Parallel Algorithms

```
nvc++ -stdpar program.cpp -o program
```

# RDF

## Pseudo Code

```
for (int frame=0;frame<nconf;frame++){

    for(int id1=0;id1<numatm;id1++)
    {
        for(int id2=0;id2<numatm;id2++)
        {
            dx=d_x[]-d_x[];
            dy=d_y[]-d_y[];
            dz=d_z[]-d_z[];
            r=sqrtf(dx*dx+dy*dy+dz*dz);

            if (r<cut) {
                ig2=(int)(r/del);
                d_g2[ig2] = d_g2[ig2] +1 ;
            }
        }
    }
}
```

- Across Frames
- Find Distance
- Reduction

# STEP 1

## Replace **for** with **std::for_each**

```
std::for_each (InputIterator first, InputIterator last, Function fn)
```

| | |
|---|---|
| start_iter : | The beginning position from where function operations have to be executed. |
| last_iter : | The ending position until which the function must be executed. |
| fnc/obj_fnc : | The 3rd argument is a function or an object function whose operation(s) would be applied to each element. |

# STEP 2

Pass execution policy as **std::execution::par**

```
for_each (std::execution::par , InputIterator first, InputIterator last, Function fn)
```

Execution policy as the first parameter will dictate that the loop body will run in parallel across threads

# STEP 3

## Change indexing to use counting::iterator

```
std::for_each(std::execution::par,
              thrust::counting_iterator<unsigned int>(0u),  thrust::counting_iterator<unsigned int>(numatm*numatm)
```

```
std::vector<unsigned int> indices(numatm * numatm);
std::generate(indices.begin(), indices.end(), [n = 0]() mutable { return n++; });

std::for_each(std::execution::par,
              indices.begin(), indices.end(),
```

- Counting Iterator helps in filling up a vector with the numbers zero through N
- In our case from 0 to number of atoms squared
- GPU will be using **Thrust** library for counting iterator for GPU
    - High-Level Parallel Algorithms Library
    - Parallel Analog of the C++ Standard Template Library (STL)

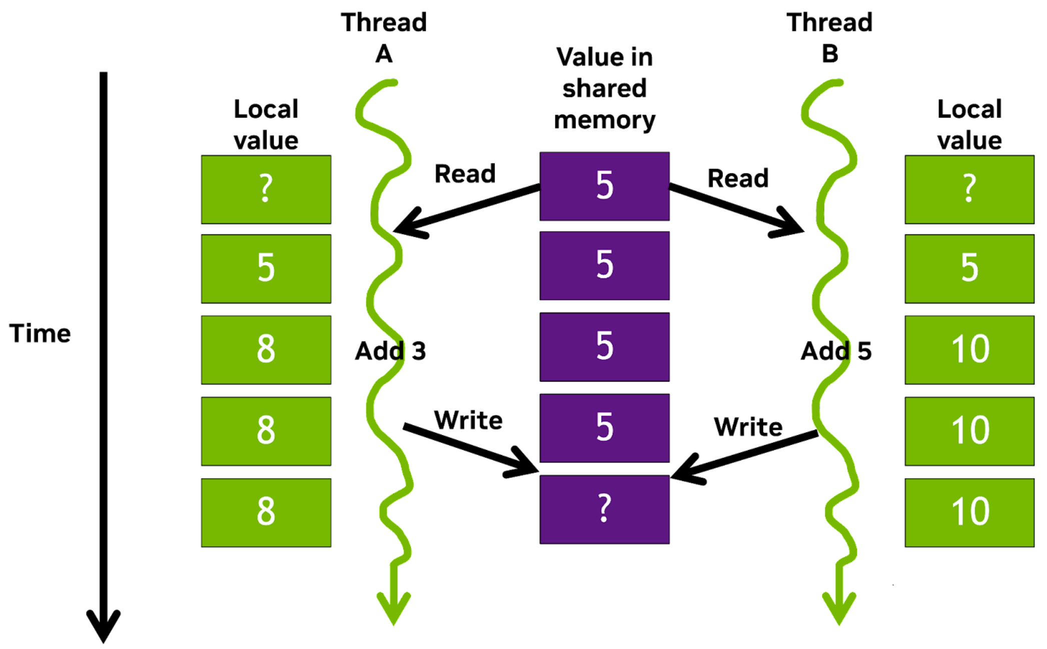
# STEP 4

## Use atomic to avoid a data race condition

```
std::atomic<int>* h_g2 = new std::atomic<int>[nbin];
```

```
do_stuff()
{
    ...
    // Execute in parallel
    for (int i = 0 ; i < 200000000 ; ++ i)
    {  runningTotal += myNum; }
    ...
}
```

Since the variable runningTotal is shared – i.e. multiple threads can access it simultaneously - w can get a data race



## STEP 5

### Put the function body inside a **Lambda**

```
std::for_each(std::execution::par,
            thrust::counting_iterator<unsigned int>(0u),  thrust::counting_iterator<unsigned int>(numatm*numatm)
[...](unsigned int index)
            {
                  for(int id2=0;id2<numatm;id2++)
                  {
                        dx=d_x[]-d_x[];
                        dy=d_y[]-d_y[];
                        dz=d_z[]-d_z[];
                        r=sqrtf(dx*dx+dy*dy+dz*dz);

                        if (r<cut) {
                              ig2=(int)(r/del);
                              ++d_g2[ig2];
                        }
                  }
            }
      )
```

- Lambda : convenient way of defining an anonymous function

# STEP 6

## Compile the code for multicore and for GPU

```
std::atomic<int>* d_g2 = new std::atomic<int>[nbin];

std::for_each(std::execution::par, thrust::counting_iterator<unsigned int>(0u),
                                   thrust::counting_iterator<unsigned int>(numatm*numatm),
        [...](unsigned int index)
                {
                        for(int id2=0;id2<numatm;id2++)
                        {
                                ...

                        }
                }
        )
```

```
nvc++ -stdpar=multicore program.cpp -o program
nvc++ -stdpar=gpu program.cpp -o program
```

## KNOWN LIMITATIONS

# LIMITATIONS

## Heap Only

- Limitation: All pointers used in parallel algorithms must point to the heap

```
std::array<int, 1024> a = ...;

std::sort(std::execution::par, a.begin(), a.end()); // Fails, array stored on the stack
```

- Solution: declare STL types dynamically. Also, can pass by value to the lambda which will copy the data to the GPU

```
std::vector v = ...;
std::sort(std::execution::par, v.begin(), v.end()); // OK, vector allocates on heap
          ...
std::transform(std::execution::par, x, x + N, y, y, [&](float xi, float yi){ return a * xi + yi; }); // Nope
std::transform(std::execution::par, x, x + N, y, y, [=](float xi, float yi){ return a * xi + yi; }); // Yep
```

# OTHER LIMITATIONS

- GPU code does not have access to the operating system or pre-compiled standard library
- Usually works:
  - template classes and functions
  - inlined functions
  - math functions
- Usually doesn't work:
  - non-template library functions
  - OS functions

# LIMITATIONS

## FUNCTION POINTERS

- Limitation:  Don’t pass function pointers to algorithms that will run on the GPU

```
void square(int& x) { x = x * x; }

std::for_each(std::execution::par, v.begin(), v.end(),  &square); // Fails: uses raw function pointer
```

- Solution: Use function objects or lambdas instead

```
struct square {

        void operator()(int& x) const { x = x * x; }

};

std::for_each std::execution::par  v.begin(), v.end(), square()); // OK, function object

std::for_each(std::execution::par, v.begin(), v.end(), [](int& x) { x = x * x; } ); // OK, lambda
```

## FORTRAN

OpenACC
More Science, Less Programming

OPEN HACKATHONS

# FORTRAN

## DO CONCURRENT :: ISO Standard Fortran

- ISO Standard Fortran 2008 introduced the DO CONCURRENT construct to allow you to express loop-level parallelism, one of the various mechanisms for expressing parallelism directly in the Fortran language
- HPC SDK 20.11 release of the NVIDIA HPC SDK, the included NVFORTRAN compiler automatically accelerates DO CONCURRENT

```fortran
subroutine saxpy(x,y,n,a)
  real :: a, x(:), y(:)
  integer :: n, i
  do i = 1, n
    y(i) = a*x(i)+y(i)
  enddo
end subroutine saxpy
```

```fortran
subroutine saxpy(x,y,n,a)
  real :: a, x(:), y(:)
  integer :: n, i
  do concurrent (i = 1: n)
    y(i) = a*x(i)+y(i)
  enddo
end subroutine saxp
```

```
nvfortran –stdpar=gpu,multicore program.f90 -o program
```

# FORTRAN

## Nested Loop Parallelism

- Nested loops are a common code pattern encountered in HPC applications
- It is straightforward to write such patterns with a single DO CONCURRENT statement, as in the following example

```
do i=2, n-1
    do j=2, m-1
        a(i,j) = w0 * b(i,j)
    enddo
enddo
```

```
do concurrent(i=2 : n-1, j=2 : m-1)
    a(i,j) = w0 * b(i,j)
enddo
```

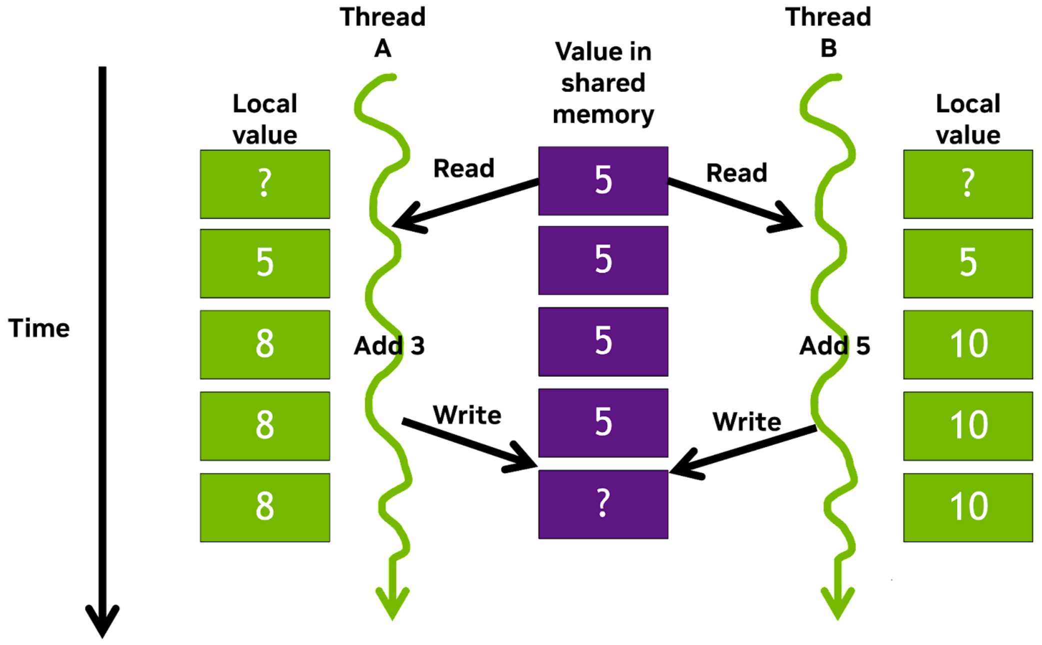
# ATOMIC: Limitation

## Use OpenACC **atomic** to avoid a data race condition

```
!$acc atomic
g(ind) = g(ind)+1.0d0
```

```
do_stuff()
{
    ...
    // Execute in parallel
    for (int i = 0 ; i < 200000000 ; ++ i)
    {  runningTotal += myNum; }
    ...
}
```

- Do-Concurrent implementation of HPC SDK currently does not support atomic constructs
- We use OpenACC to prevent data race



# STEPS

## Compile for Multicore and GPU

```
do iconf=1,nframes

      do concurrent(i=1 : natoms, j=1:natoms)
            dx=x(iconf,i)-x(iconf,j)
            dy=y(iconf,i)-y(iconf,j)
            dz=z(iconf,i)-z(iconf,j)

              …
            r=dsqrt(dx**2+dy**2+dz**2)
            if(r<cut)then
               !$acc atomic
               g(ind)=g(ind)+1.0d0
            endif
      enddo
enddo
```

- Do Concurrent
- Find Distance
- Atomic Increment

```
nvfortran -stdpar=gpu,multicore program.f90 -o program
```

## REFERENCES

https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/

https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9770-c++17-parallel-algorithms-for-nvidia-gpus-with-pgi-c++.pdf

## Acknowledgment

Learn more at

WWW.OPENHACKATHONS.ORG

OpenACC
More Science, Less Programming

OPEN
HACKATHONS