Requirement: fetching atomic operations in DO CONCURRENT

Motivation

Atomic operations are an integral part of concurrent algorithms. Currently, Fortran has atomic operations for coarray data, but not DO CONCURRENT regions. The only legal way to access data concurrently within a DO CONCURRENT region is to use the REDUCE locality specifier, but the effect of this is only visible at the end, which means it is impossible to implement classes of algorithms where the results of atomic operations must be known during a DO CONCURRENT region.

One common pattern where fetching atomic operations are used is in the parallel construction of a list. This pattern appears in particle codes, where a parallel agent needs to acquire a unique offset into an array in order to write its data.

```
INTEGER, INTENT(IN) :: maxsize, num, maxtempsize
INTEGER :: offset, i, mysize
REAL :: array(:), temp(:)
ALLOCATE( array(maxsize) , temp(maxtempsize) )
DO CONCURRENT i=1:num
   CALL determine(temp, mysize)
   offset = ATOMIC_FETCH_ADD( offset, mysize )
   array(offset:offset+mysize) = temp(1:mysize)
END DO
```

There is currently no mechanism to write such code in Fortran today, except as a distributed memory algorithm using coarrays.

Proposed Solutions

There are at least three possible options for solving this problem:

1. Relax the restrictions on the existing intrinsic procedure, ATOMIC_FETCH_ADD to allow it to operate on any memory, not just coarray memory.
2. Add a locality specifier that ensures accesses are atomic within a given DO CONCURRENT region.

3. Add a type specifier that prescribes that all accesses to a given variable be atomic.

There are positive and negative aspects to each of these options.

The current use cases and implementations of ATOMIC_FETCH_ADD assume distributed memory is supported and thus may have higher overhead than is appropriate for shared-memory. Furthermore, ensuring that ATOMIC_FETCH_ADD works for both distributed memory and shared memory may slow down the distributed memory implementation.

Using a locality specifier is good in that it means the implementation only needs to generate atomic operations in the scope of a DO CONCURRENT, which may be more efficient than if done across the lifetime of a variable. On the other hand, it may limit our ability to add other forms of shared-memory concurrency in the future.

A type specifier is consistent with the existing art in C11 and C++11 atomics, which means it the implementation is well understood. If limited to certain native types, efficient implementations are straightforward. However, if allowed on arbitrary types, the implementation may be required to do more complicated things, such as lock tables, which may not be desirable.

Prior Art

This feature is supported by a wide range of hardware and software, including, but not limited to:

- Fortran coarrays
- MPI-3 RMA[1]
- OpenSHMEM[2]
- OpenMP[3]
- OpenACC[4]
- C++11[5]
- C11[6]
- Widely used hardware:
    - x86_64 (via the lock prefix)
    - ARMv7 (via load-link-store-conditional)
    - ARMv8.1 (natively[7])
    - PowerPC (via load-link-store-conditional)

[1] https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node272.htm
[2] http://openshmem.org/site/sites/default/site_files/openshmem_specification-1.0.pdf
[3] https://www.openmp.org/spec-html/5.0/openmpsu95.html
[4] https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf
[5] https://en.cppreference.com/w/cpp/atomic
[6] https://en.cppreference.com/w/c/atomic
[7] https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/the-armv8-a-architecture-and-its-ongoing-development

- IBM Power 9 / PowerPC 3.0 (natively[8])
- RISC-V (natively[9])
- CUDA[10]
- PCIe 3.0[11]
- InfiniBand[12]

We note that proposed solution 1 is equivalent to Fortran coarrays and similar to MPI-3 and OpenSHMEM. Proposed solution 2 is somewhat similar to OpenMP and OpenACC, but more restrictive than either. Proposed solution 3 is very similar to C11 and C++11.

[8] https://gcc.gnu.org/onlinedocs/gcc/PowerPC-Atomic-Memory-Operation-Functions.html
[9] https://riscv.org/wp-content/uploads/2016/06/riscv-spec-v2.1.pdf
[10] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
[11] https://old.hotchips.org/wp-content/uploads/hc_archives/hc21/1_sun/HC21.23.1.SystemInterconnectTutorial-Epub/HC21.23.131.Ajanovic-Intel-PCIeGen3.pdf
[12] https://docs.nvidia.com/networking/display/MLNXOFEDv512620/Advanced+Transport