CS420 – Lecture 25

Marc Snir

Fall 2018

How about atomic operations not supported by C++?

Can use Compare-and-Swap to implement such operations.
`atomic.compare_exchange_strong(T& expected, T desired)`

- If the current value of `atomic` is equal to `expected` then this value is replaced by `desired` and the method returns `true`
- Otherwise, it does not update `atomic`, returns in `expected` the value of `atomic`, and returns `false`

`atomic.compare_exchange_weak(T& expected, T desired)` has same semantics, is faster, but may fail spuriously.

Generic algorithm for atomic updates:

- Read `atomic`
- Compute new value of `atomic`
- Attempt to update `atomic` with new value, using CAS
- If successful (there was no other update since last read), DONE; otherwise, start again.

Atomic max

```
...
void atomic_max(volatile std::atomic<int>* counter, int val) {
 int previous = counter.load();
   while(previous < val &&
    !counter->compare_exchange_weak(previous, val)) {}
}
```

Very efficient if conflicts are rare. Need locks or backoff if conflicts are frequent

C++17

- Previous constructs are part of C++11 and are supported by most compilers.
- New constructs have been added in C++17 and may not be fully supported.
- *Parallel versions of the STL algorithms*

```
std::vector<int> v(100000);
...
// sequential sort
std::sort(v.begin(), v.end());

// parallel sort

std::sort(std::execution::par, v.begin(), v.end());
```

STL library methods accept an extra argument, an *execution policy*

std::execution::seq sequential execution (default)

std::execution::par multithreaded execution

std::execution::par_unseq multithreaded execution + vectorization

Implementations may ignore the policies...

Examples

C++ `transform` applies a unary operation to each element of a rangee at returns the results in new range
Or apply binary operation to each pair of elements in two ranges and returns result in new range

```
std::vector<float> X= {...};
std::vector<float> Y(X,size());

std::transform(
 begin(X), end(X), // input range
 begin(Y),         // start of output
 [](float x) {return x*2.0f} // operation (lambda expression)
 );
```

Will execute sequentially `Y=2.0*X`.

```
std::vector<float> X= {...};
std::vector<float> Y(X,size());

std::transform(
std::execution::par,
begin(X), end(X), // input range
begin(Y),         // start of output
[](float x) {return x*2.0f} // operation (lambda expression)
);
```

Does the same, but in parallel

Vector product

```
std::vector<float> X(10000), Y(100000), Z(100000);
float product;
...
std::transform(
  std::execution::par,
  begin(X), end(X), // input range
  begin(Y), begin(Z),         // start of output
  [](float x, y) {return x*y} // operation
);

product = std::reduce(Z.first(), Z.last(), 0.0);
```

Reduce uses a binary tree algorithm, as distinct from `std:accumulate` that preserves order

Handling races

May execute in parallel iterates that conflict
But need to protect them with a lock or otherwise resolve conflicts

Creating vectors of even integers

```
std::vector<int> vec(1000);
std::iota(vec.begin(), vec.end(), 0);
std::vector<int> output;
std::mutex m;

std::for_each(std::execution::par,
  vec.begin(), vec.end(),
  [&output, &m](int& elem) {
   if (elem % 2 == 0) {
  std::lock_guard guard(m);
  output.push_back(elem);
  }
});
```

- Each evaluation of the lambda expression uses (captures) `output` and `m`.
- Each evaluation is protected by the lock
- Code will be correct but **very** slow.

STL Algorithms that can run parallel

adjacent_difference
adjacent_find
all_of
any_of
copy
copy_if
copy_n
count
count_if
equal
exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n
generate
generate_n
includes
inclusive_scan
inner_product
inplace_merge
is_heap
is_heap_until
is_partitioned
is_sorted
is_sorted
is_sorted_until
lexicographical_compare
max_element
merge
min_element
minmax_element
mismatch
move
none_of
nth_element
partial_sort
partial_sort_copy
partition
partition_copy
remove
remove_copy
remove_copy_if
remove_if
replace
unique

replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
sort
stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_fill
uninitialized_fill_n
unique_copy

A few new STL algorithms

for_each similar to for_each except returns void

for_each_n applies a function object to the first n elements of a sequence

reduce similar to accumulate, except out of order execution to allow parallelism

transform_reduce transforms the input elements using a unary operation, then reduces the output out of order

exclusive_scan parallel version of partial_sum, excludes the i-th input element from the i-th sum, out of order execution to allow parallelism

inclusive_scan parallel version of partial_sum, includes the i-th input element in the i-th sum, out of order execution to allow parallelism

transform_exclusive_scan applies a functor, then calculates exclusive scan

transform_inclusive_scan applies a functor, then calculates inclusive scan

Example

```
std::vector<int> v={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto sumTransformed = std::transform_reduce(
  std::execution::par,
  v.begin(), v.end(), 0,
  std::plus<int>{},
   []( const int& i) { return i * 2; }
  );
```

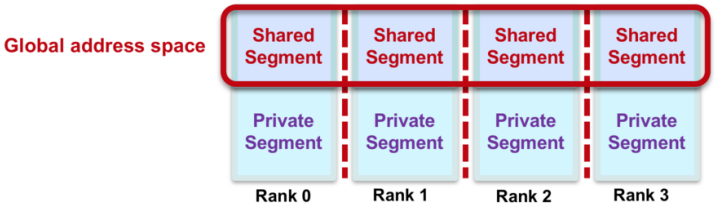
sum is 110

Partitioned Global Adress Space: UPC++

- Basic idea: Treat distributed memory system as if it had shared memory
- Use *global pointers*: `global_pointer = [process,address]`.
- `store` is replaced by `put`
- `load` is replaced by `get`
- Done by compiler (UPC, X10,...) or done (in UPC++) by overloading pointers.

Problems

- get from remote memory takes $\times 10$ longer than from local memory
  - Programmer has to be aware of where variables are kept
- If each pointer is a global pointer then all references take longer
  - Need to distinguish between regular pointers and global pointers
- There are no global coherent caches
  - User needs to "cache" explicitly, by copying data from remote memory to local memory, and back



- Private segments can be accessed only by local process, using regular pointers
- Global segments can be accessed all processes, using global pointers.
- Each process executes the same program, as in MPI

Hello World

```
#include <iostream>
#include <upcxx/upcxx.hpp>
using namespace std;

int main(int argc, char *argv[]) {

 upcxx::init();
 cout << "Hello␣world␣from␣process␣" << upcxx::rank_me() <<
    "␣out␣of␣" << upcxx::rank_n() << "␣processes" << endl;
 upcxx::finalize();
 return 0;
}
```

Global memory allocation

```
upcxx::global_ptr<int> gptr = upcxx::new_<int>(upcxx::rank_me());
```

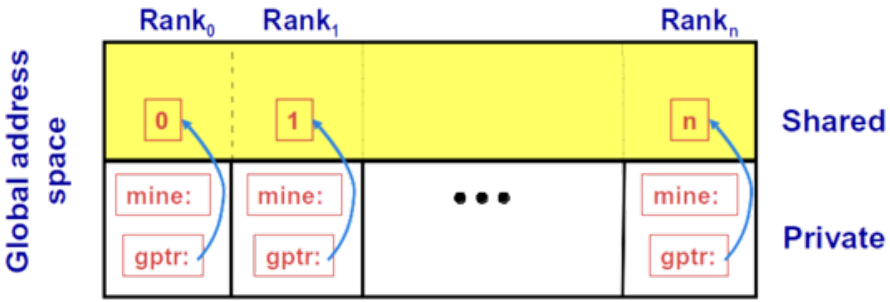
Allocates an integer variable in the local shared segment, initializes it to local rank and returns a global pointer pointing to this integer.

Global memory allocated with `upcxx::new_` can be freed with `upcxx::destroy_`.

`upcxx::new_array` method can be used to allocate 1D-arrays

```
upcxx::global_ptr<int> x_arr = upcxx::new_array<int>(10);
```

An array of length 10 is allocated in the shared memory segment at each process.

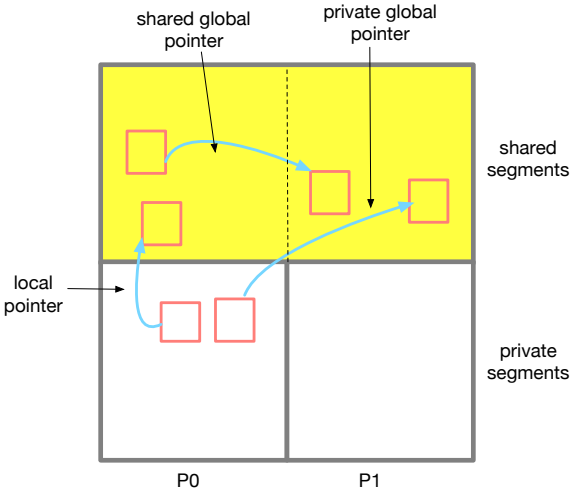

Rank0
Rank1
Rankn
Global address space
0
1
n
Shared
mine:
gptr:
mine:
gptr:
mine:
gptr:
Private

Downcasting pointers

A global pointer that points to the local shared memory segment can be cast to a regular pointer.

```
upcxx::global_ptr<int> x_arr = upcxx::new_array<int>(10);
int *local_ptr = x_arr.local();
local_ptr[i] = ... // work with local ptr
```

A local pointer cannot be cast to a global pointer.


shared global pointer
private global pointer
shared segments
local pointer
private segments
P0
P1

Collective allocations

Previous allocators are local:

```
if (upcxx::rank_me() == 0)
  upcxx::global_ptr<int> x_arr = upcxx::new_array<int>(10);
```

will allocate an array only in the shared memory segment of process 0; the returned pointer is available only in the private memory of process zero.
Collective allocators are called collectively by all processes and return at each process a global reference that can used to obtain a globval pointer pointing to any compnent.

```
upcxx::dist_object<upcxx::global_ptr<double>>
    u_g(upcxx::new_array<double>(10));
```

Allocates an array of length 10 in each shared memory segment

1D red-black iteration

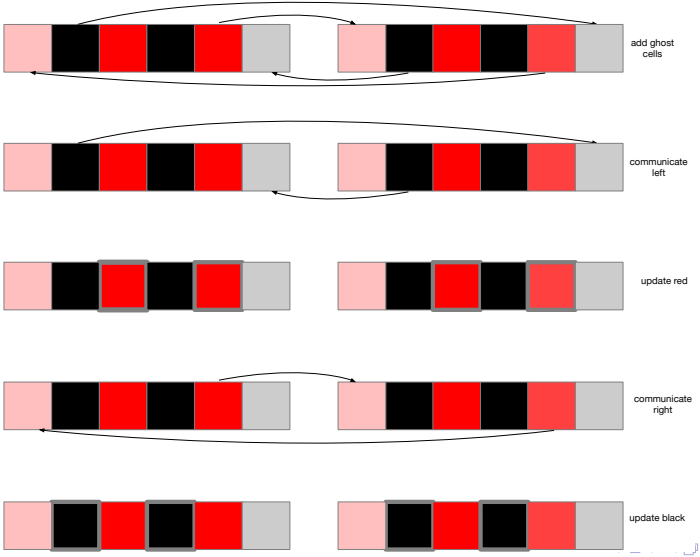
We assume cyclic array

Example: Sequential code

```
for (int step = 0; step < max_steps; step++) {
 for (int i = step % 2; i < n ; i += 2)
  u[i] = 0.5*(u[(i - 1)%n] + u[(i + 1])%n);
}
```

Parallel code

We assume vector length is a multiple of $2p$, p number of processes.



UPC++ parallel code

```
int main(int argc, char **argv) {

 upcxx::init();
//  initialize parameters - simple test case
 const long N = 1000;
 const long MAX_ITER = N * N * 2;
 const int MAX_VAL = 100;
//  get the bounds for the local panel, assuming 2*num_procs
//  divides N evenly
 int block = N / upcxx::rank_n();
// plus two for ghost cells
 int n_local = block + 2;

// set up the distributed object
 upcxx::dist_object<upcxx::global_ptr<double>>
    u_g(upcxx::new_array<double>(n_local));
// downcast to a regular C++ pointer -- points to start of local array
 double *u = u_g->local();
```

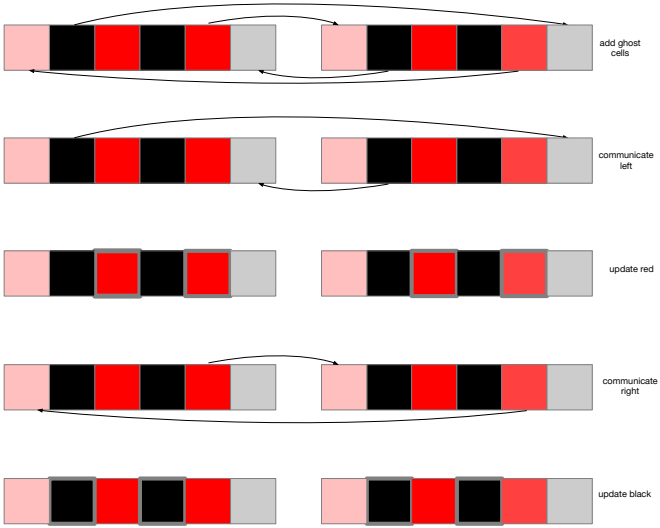
```
// initializes random number generator ("Mersenne Twister" with 19937 bits)
 std::mt19937_64 rgen(upcxx::rank_me() + 1);
// fill with uniformly distributed random values
 for (int i = 1; i < n_local - 1; i++)
  u[i] = 0.5 + rgen() % MAX_VAL;

// get access to left and right neighbor
 upcxx::global_ptr<double> uL = nullptr, uR = nullptr;
// retrieve global pointers for arrays at left and right neighbors
 uL = u_g.fetch((upcxx::rank_me()-1)%upcxx::rank_n()).wait();
 uR = u_g.fetch((upcxx::rank_me() + 1)%upcxx::rank_n()).wait();

 upcxx::barrier();
```

```
// iteratively solve
 for (int step = 0; step < MAX_ITER; step++) {
// alternate between red and black
  int start = step % 2;
// get the values for the ghost cells
  if (!start)
   u[0] = upcxx::rget(uL + block).wait;
  else
   u[n_local - 1] = upcxx::rget(uR + 1).wait();
// compute updates
  for (int i = start + 1; i < n_local - 1; i += 2)
   u[i] = 0.5*(u[i - 1] + u[i + 1]);
// wait until all processes have finished calculations
  upcxx::barrier();
  }
 }
 upcxx::finalize();
 return 0;
}
```

`fetch(rank)`
Asynchronously retrieves a copy of the instance of the distributed object at the specified process.
`fetch(rank).wait` also blocks until the operation is complete

`rget(global_pointer)`
starts an asynchronous get from the indicated address
`rget(glogal_pointer).wait`
also waits for its completion

Overlap computation and communication



- compute leftmost (black) cell
- start sending it to the left
- compute (black) interior
- complete communication
- compute rightmost (red) cell
- start sending it
- compute (red) interior
- complete communication

UPC++ code

```
// iteratively solve
upcxx::future<> fut;
for (int step = 0; step < MAX_ITER; step++) {
// alternate between red and black
 int start = step % 2;
 if (!start) {
  u[1]=0.5*(u[0]+u[2]);
// remote put with future to test completion
  upcxx::rput(u[0], uL + block, operation_cx::as_future(fut))
  }
 else {
  u[block] = 0.5*u[block-1]+u[block+1]);
  upcxx::rput(u[block], uR+1, operation_cx::as_future(fut))
  }
```

```
// update interior
 for (int i = start + 1; i < n_local - 1; i += 2)
    u[i] = 0.5*(u[i - 1] + u[i + 1]);
// complete remote put;
while (!fut.ready()) upcxx::progress();
// wait until all processes have finished calculations
upcxx::barrier();
% }}
upcxx::finalize();
return 0;
 }
```

- An asynchronous operation can be completed by a future or a procedure call.
- "Completion" may mean completion at the source process (`upcxx::source_cx::as_future()`) or completion of the operation at both source and destination (`upcxx::operation_cx::as_future()`)
- `fut.ready()` tests the future is satisfied
- `progress()` makes sure async operation make progress

Atomics

First, create *domain* that specifies what atomic operations can be applied to a variable

```
atomic_domain <int64_t > ad_i64 ({ atomic_op::load,
  atomic_op::store, atomic_op:: fetch_add });
```

Next, can perform atomic operations from this set

```
global_ptr <int64_t> x = new_ <int64_t>(0);
// fetch&add returns future
future <int64_t > f = ad_i64.fetch_add(x, 2, std::memory_order_relaxed);
// wait for future completion
int64_t res = f.wait ();
```