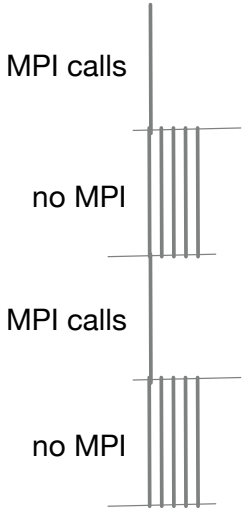
CS420 – Lectures 21

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Two ways of using MPI with OpenMP

1. MPI calls occur only when OpenMP code executes on one thread
2. MPI calls can occur concurrently on multiple threads



MPI+OpenMP

Replace `MPI_Init` with `MPI_Init_thread(required,&provided)`

required: what you ask for.

provided: what you actually get.

`required` can be one of the following:

MPI_THREAD_SINGLE: MPI process is single threaded

MPI_THREAD_FUNNELED: Only the master thread (the thread that initialized MPI) executes MPI calls.

- Used when MPI always called from master thread in OpenMP

MPI_THREAD_SERIALIZED: Multiple threads can call MPI, but the calls ar not concurrent.

MPI_THREAD_MULTIPLE: No constraints on concurrent MPI calls.

- Used when multiple OpenMP threads may call MPI

Master only

```
#pragma omp parallel
{
  // do work
}
// communicate
MPI_Send()
#pragma omp parallel
{
  // do work
}
```

Master only

```
#pragma omp parallel
{
  // do work
}
// communicate
MPI_Send()
#pragma omp parallel
{
  // do work
}
```

- Simple to write and maintain: Clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
- Threads other than the master are idle during MPI calls
- All communicated data passes through the cache where the master thread is executing
- Inter-process and inter-thread communication do not overlap

Funneled

```
#pragma omp parallel
{
  // do work
  #pragma omp barrier
  #pragma omp master
  {
    // communicate
    MPI_Send()
  }
  #pragma omp barrier
  // do work
}
```

- Relatively simple to write and maintain
- Possible for other threads to compute while master is in an MPI call
- Less clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
- All communicated data still passes through the cache where the master thread is executing
- Inter-process and inter-thread communication still do not overlap

Serialized

```
#pragma omp parallel
{
  // do work
  #pragma omp critical
  {
    // communicate
    MPI_Send()
  }
  // do work
}
```

- Easier for other threads to compute while one is in an MPI call
- Can arrange for threads to communicate only their “own" data (i.e. the data they read and write).
- Harder to write and maintain
- More, smaller messages are sent, incurring additional latency overheads
- Need to use tags or communicators to distinguish between messages from or to different threads in the same MPI process.

Serialized

- By default, a call to MPI_Recv by any thread in an MPI process will match an incoming message from the sender
- To distinguish between messages intended for different threads, we can use MPI tags
- Alternatively, different threads can use different MPI communicators

Multiple

```
#pragma omp parallel
{
  // do work
  // communicate
  MPI_Send()
  // do work
}
```

- Messages from different threads can (in theory) overlap: many MPI implementations serialise them internally
- Natural for threads to communicate only their "own" data
- Hard to write and maintain
- Not all MPI implementations support this: loss of portability

MPI+OpenMP

```
MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED
< MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

MPI_Query_thread()
```

One sided

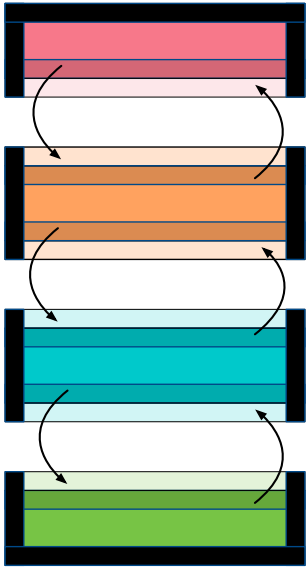
One cause of inefficiency in MPI is the overhead of matching sends to receives.

- Comparing communicator, source and tag
- Handling wildcard source and wildcard tag
- Matching message in the right order

In may cases, this is superflous, since sender "knows" were the receive buffer is (same receive buffer used again and again)
One sided communication can be used – only one side (sender or receiver) needs to call MPI.
put/get/accumulate

Jacobi- 1D tiling

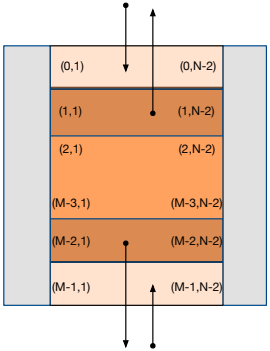
```
...
...
double a[2][M][N];
...
MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,
    rank-1,0,MPI_COMM_WORLD,&req[0]);
MPI_Irecv(&a[1][0][1],N-2,MPI_DOUBLE,
  rank-1,0,MPI_COMM_WORLD,&req[1]);
...
MPI_Isend(&a[1][M-2][1],N-2,MPI_DOUBLE,
    rank+1,0,MPI_COMM_WORLD,&req[2]);
MPI_Irecv(&a[1][M-1][1],N-2,MPI_DOUBLE,
    rank+1,0,MPI_COMM_WORLD,&req[3]);
...
```



With 1-sided communication

```
...
MPI_Win win; /* declare window */

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/* expose array to remote accesses */
MPI_Win_create(a, sizeof(a), sizeof(a[0]), NULL,
     MPI_COMM_WORLD, &win);

for(iter=0; iter<MAX; iter++) {
/* compute first and last row */
 for(j=1; j<N-1; j++)
  a[1-k][1][j] = 0.25*(a[k][0][j]+a[k][2][j]
     +a[k][1][j-1]+a[k][1][j+1]);
 for(j=1; j<N-1; j++)
  a[1-k][M-2][j] = 0.25*(a[k][M-3][j]+a[k][M-1][j]
     +a[k][M-2][j-1]+a[k][M-2][j+1]);
```

```
/* start communication */
 MPI_Win_fence(0, win);

 if (rank>0)  /* send data to neighbor above */
  MPI_Put(&a[k][1][1], N-2, MPI_DOUBLE, rank-1,
      (int)(&a[1-k][M-1][1] -a), N-2, MPI_DOUBLE, win);

 if (rank<size-1) /* send data to neighbor below */
  MPI_Put(&a[k][M-2][1], N-2, MPI_DOUBLE, rank+1,
     (int)(&a[1-k][0][1]-a), N-2, MPI_DOUBLE, win);
/* compute interior */
 for(i=1;i<M-2;i++)
  for(j=1;j<N-2;j++)
   a[1-k][i][j] = 0.25*(a[j][i-1][j]+a[k][i+1][j]
     +a[k][i][j-1]+a[k][i][j+1]);
 MPI_Win_fence(0, win) /* complete communication */
 k=1-k;
```



`MPI_Win_create(base, size, disp_unit, info, comm, win)`

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument
- comm: processes involved
- win: window object returned by the call

`MPI_Win_create(base, size, disp_unit, info, comm, win)`

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument
- comm: processes involved
- win: window object returned by the call

Also

- `MPI_Win_allocate` – allocates new memory when window is created; and
- `MPI_Win_create_dynamic` and `MPI_win_attach` – create zero size window and dynamically attach new memory as needed.

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

target_disp: displacement from start of window to remote buffer

target_count: number of entries in remote buffer

target_datatype: type of each entry in remote buffer

win: window used for communication

`MPI_WIN_FENCE(assert, win)`

assert: information on type of communication

win: window

`MPI_WIN_FENCE(assert, win)`

assert: information on type of communication

win: window

Acts like a barrier: starts and completes a *communication epoch*

Example: Distributed table

- Table too large to fit on one process – it is distributed across all processes
- assume integer indices and double values
- Three operations:
  - `double read_table(int index)`
  - `void write_table(int index, double value)`
  - `void increment_table(int index, double increment)`
- Accesses to table are not bulk synchrounous – each process can access the distributed table independently
- Need to ensure atomicity of accesses (mutual exclusion)

implementation

```
...
#define TABLE_SIZE 10000000000

int local_size; /* local table size */

main() {
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int local_size = (TABLE_SIZE+size-1)/size;
double table[local_size];
MPI_Win_create(table, sizeof(table), sizeof(table[0]), 0,
     MPI_COMM_WORLD, win);
```

Cannot use fence, since each process can call the 3 functions at any point in time.

```
double read_table(int index) {
double value;
int target_rank= index%size;
int  target_disp= index/size;
MPI_Win_lock(MPI_LOCK_SHARED, target_rank, 0, win);
MPI_Get(&value, 1, MPI_DOUBLE, target_rank, target_disp, 1, MPI_DOUBLE, win);
MPI_Win_unlock(target_rank, win);
return value;

void write_table(int index, double value) {
int target_rank= index%size;
int  target_disp= index/size;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target_rank, 0, win):
MPI_Win_put(&value, 1, MPI_DOUBLE, target_rank, target_disp, 1, MPI_DOUBLE,
    win);
MPI_Win_unlock(rank, win);
}
```

```
void increment_table(int index, double value) {
int rank= index%rank;
int  local_index= index/rank;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win):
MPI_Accumulate(&value, 1, MPI_DOUBLE, rank, local_index, 1, MPI_DOUBLE,
    MPI_SUM, win);
MPI_Win_unlock(rank, win);
}
```

- `MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)` – same arguments as `MPI_Put`.
- `MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)` – same arguments as `MPI_Put`, with the addition of the operation argument.
- `MPI_Win_lock(lock_type, rank, assertion, win)`
- `MPI_Win_unlock(rank, win)`

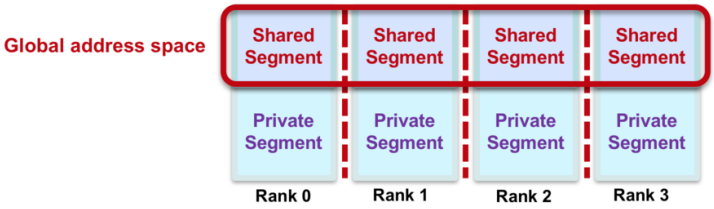
The following slides are for reference only (not for the exam).

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());
```

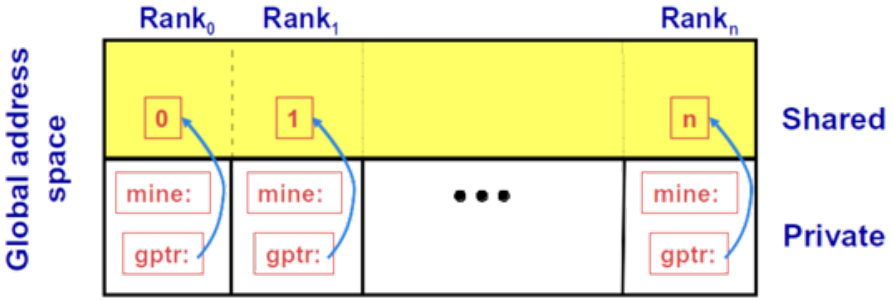
Allocates an integer variable in the local shared segment, initializes it to local rank and returns a global pointer pointing to this integer.

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.

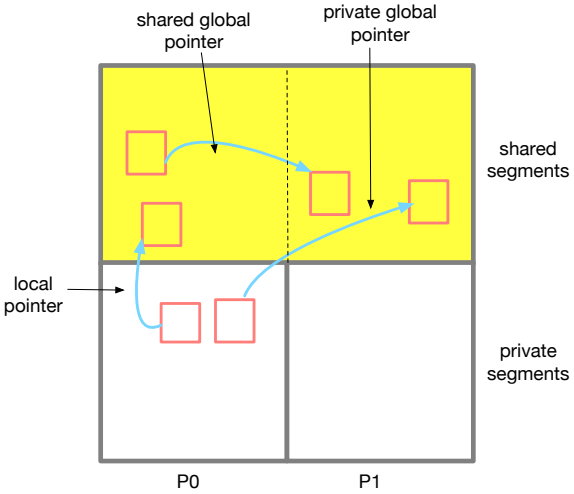

Rank0
Rank1
Rankn
Global address space
0
1
n
Shared
mine:
gptr:
mine:
gptr:
mine:
gptr:
Private

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 global pointer pointing to any component.

```
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

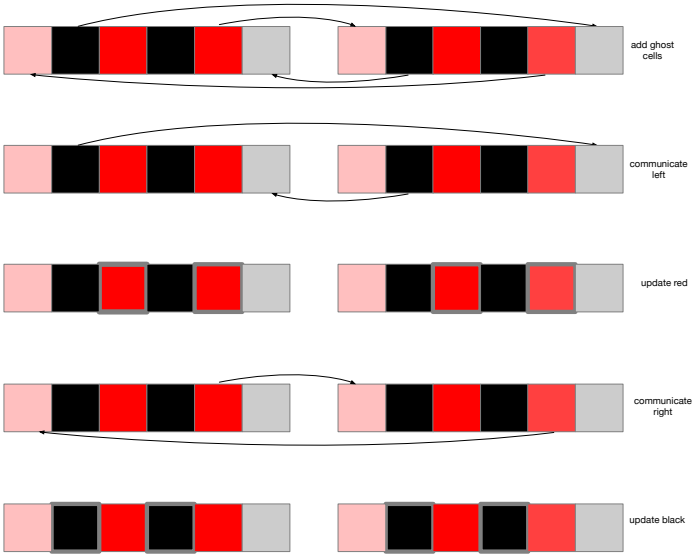
We assume cyclic array

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();
```

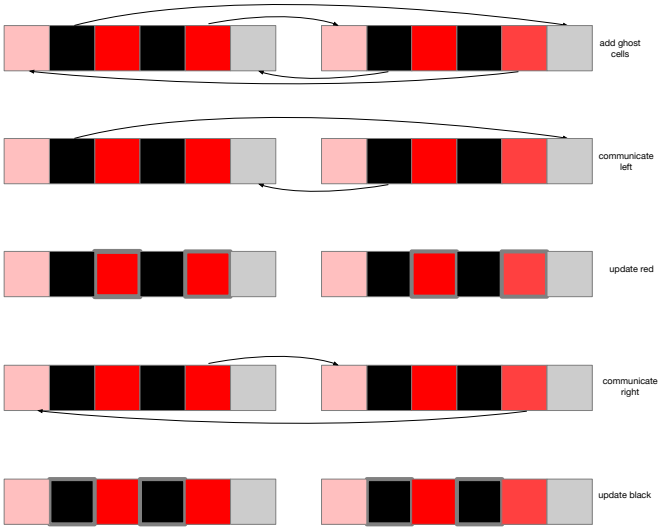
```
// initializes random number generator ("Mersenne Twister" with 19937 bits)
 std::mt19937_64 rgen(upcxx::rank_me() + 1);
// fill with uniformly distributed random values
 for (int i = 1; i < n_local - 1; i++)
  u[i] = 0.5 + rgen() % MAX_VAL;

// 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

Charm++

- Basic Object is *Chare*: Contains data and external methods that can be called by other chares; the methods do not return values.
- Chares are distributed across compute nodes; there are typically multiple chares per node, and they can be migrated.

Charm++ Hello World

Header file (hello.h)

```
class Hello : public CBase_Hello {
  public:
  Hello(); // C++ constructor

  void sayHi(int from); // Remotely invocable "entry method"
};
```

Charm++ Interface file (hello.ci)

```
module hello {
  array [1D] Hello {
    entry Hello();
    entry void sayHi(int);
  };
};
```

Source file (hello.cpp)

```
#include "hello.decl.h"
#include "hello.h"

extern CProxy_Main mainProxy;
extern int numElements;

Hello::Hello() {
  // No member variables to initialize in this simple example
}
```

```
void Hello::sayHi(int from) {

  // Have this chare object say hello to the user.
  CkPrintf("Hello␣from␣chare␣#␣%d␣on␣processor␣%d␣(told␣by␣%d)\n",
  thisIndex, CkMyPe(), from);

  // Tell the next chare object in this array of chare objects
  // to also say hello. If this is the last chare object in
  // the array of chare objects, then tell the main chare
  // object to exit the program.
  if (thisIndex < (numElements - 1)) {
    thisProxy[thisIndex + 1].sayHi(thisIndex);
  } else {
    mainProxy.done();
  }
}

#include "hello.def.h"
```