One Sided Communications and Remote Memory Access

Note that this lecture covers some more complex topics and you will not be expected to use these techniques in the coursework (though you can try them on the coursework problem if you are feeling enthusiastic)

This is, though, a paradigm that is being increasingly widely used and I therefore thought it important to introduce

Two vs one-sided communications

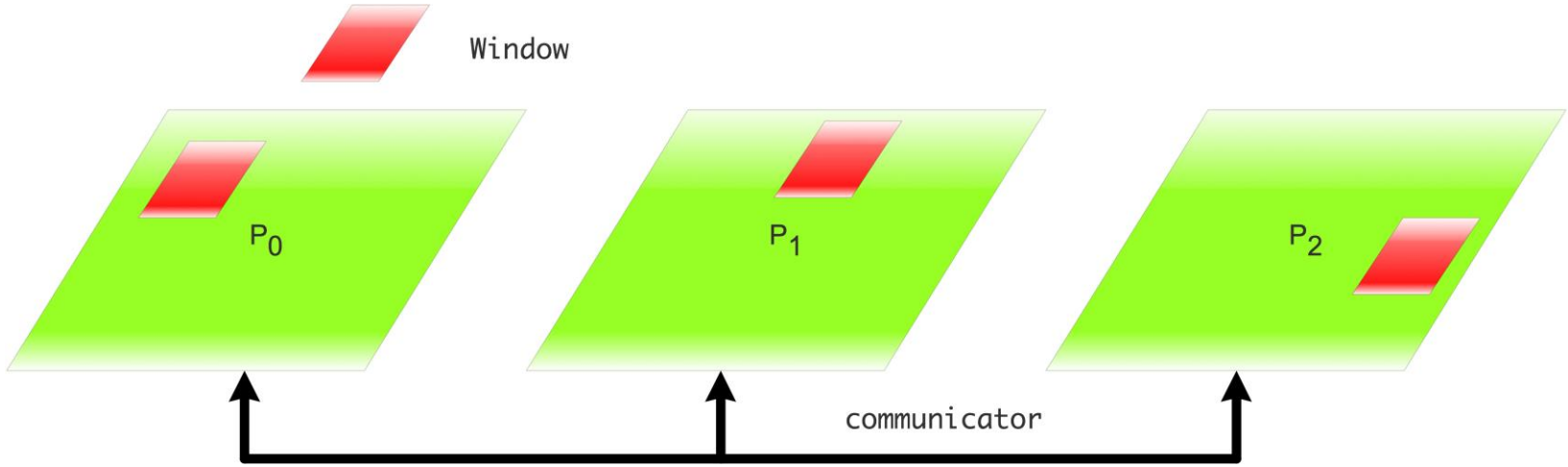
- All the communications we have done thus far in this course are two sided
  - Both the process sending the data and the process receiving the data are actively involved in the communication process
- In OpenMP all of the communications between threads are one-sided
  - The way OpenMP threads communicate is by one thread writing information to a memory location and another thread reading from the location
    - Only the thread doing the reading or the writing is involved in the process – hence one sided
- One sided communications are thus the natural way to communicate in a shared memory system as all processes have access to the same memory
- In distributed memory systems one node's processor cannot, in general, directly write to another node's memory
  - A communication is required
  - …, but it is possible to hide these communications so that it appears to the programmer that you are able to do direct memory writes
  - It is up to the implementation to do this efficiently

One Sided Communication in MPI

- MPI-3 provides a framework for doing just that
  - Directly access the memory on other processes using one sided communications
  - If those processes are on the same node then this is equivalent to a shared memory system
    - Very efficient because you are simply doing direct memory writes with no communications involved
  - On other nodes there are communications, but they are hidden
- Can do seamless shared and distributed memory computing
  - The commands used are independent of the physical infrastructure even though what is happening “under the hood” does depend on whether the memory is on the same node or not
- Not too difficult to set up and potentially very efficient, especially on shared memory systems, but very easy to make the code inefficient
  - You are not directly responsible for the communications on the other side
    - When writing to remote memory on another node there will be behind the scenes communications on both nodes – i.e. it appears one sided to the programmer, but there are “behind the scenes“ two sided communications
      - On shared memory parts of a system it will be genuinely one sided
      - Code there may seem very fast when testing on your PC, but becomes much slower when running over multiple nodes
  - You need to put in synchronisations (e.g. “fences”) to control who can write to a given piece of memory
    - As with OpenMP, this can result in a lot of process waiting and inefficiencies if done badly – exacerbated by the much longer times required to write data to memory on other nodes compared to OpenMP where the memory is inherently on the same node
  - Note that while one-sided communications can bring many of the benefits of a hybrid OpenMP – MPI code in a single framework it is not always the most efficient thing to do
    - Behind the scenes the communications between nodes behave in a similar way to non-blocking point to point communications
    - This means that they can lack some of the efficiency gains that a true collective communication can provide
  - Correct ordering of the memory access is the key to an efficient code

RMA (Remote Memory Access) via memory “Windows”

- You typically don’t want other processes to have access to all the memory on a given process
  - You only want to give other processes access to the memory it needs to have access to
  - For memory inside the window you need to worry if other processes are accessing it at the same time. Not true of the rest of the memory
- We want to set up a portion of the memory on a given process that can be accessed by the other processes
- Note that if processes are on the same node, then these are simply two windows in the same shared memory space and the communicator is just a memory read/write



RMA (Remote Memory Access) via memory “Windows”

MPI provides a number of ways of creating windows and associating memory with them

- The simplest way to do this is to both allocate the associated memory and create the window using MPI_Win_allocate
  - This is also probably the most efficient as MPI can optimise the location of this memory for ease with which it can written to from other processes
- We can also attach pre-existing memory to a window
  - Either attach this existing memory while creating a window (MPI_Win_create)
  - …or create an empty window (MPI_Win_create_dynamic) and then subsequently attach the memory to the window (MPI_Win_attach)
    - You can also detach this memory from a window (MPI_Win_detach)
  - You can get some of the efficiency of MPI_Win_allocate by using MPI_Alloc_mem when allocating the required memory, though sometimes there is a reason for using memory allocated in other ways
    - E.g. you might want to use the memory stored in a container such as a vector – Note that if you do this you must be VERY careful that you don’t do anything to make the vector reallocate its memory while it is being used for RMA!
    - Note that if you allocate the memory using MPI_Alloc_mem then you should free it using MPI_Free_mem

Put and Get communications

- The standard one-sided MPI communications are MPI_Put and MPI_Get
  - MPI_Put writes data to remote memory
  - MPI_Get read data from remote memory
  - Typically on another process, but these functions can be used to write to the memory window on the current process
    - In most cases it is easier to directly access the memory on the current process, but both option are available
  - As these are one sided the other process involved does not need to do anything for these communications to occur

- Note therefore that an MPI_Put from process i to process j can achieve the same thing as an MPI_Get from process j to i
  - Which of these you actually use for a given communication will depend on what is most efficient and sensible in given application

One-sided communication example

Implementing a scatter using MPI_Put

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

const int chunk_size = 5;

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

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	srand(time(NULL) + id * 10);

	int* send_data = nullptr;
	int* recv_data = nullptr;
	MPI_Win window;

	MPI_Win_allocate(sizeof(int) * chunk_size, sizeof(int), MPI_INFO_NULL,
		MPI_COMM_WORLD, &recv_data, &window);

	if (id == 0)
	{
		send_data = new int[p * chunk_size];
		for (int i = 0; i < p; i++)
		{
			cout << "Data for process " << i << ": ";
			for (int n = 0; n < chunk_size; n++)
			{
				send_data[i * chunk_size + n] = rand() % 1000;
				cout << send_data[i * chunk_size + n] << "\t";
			}
			cout << endl;
		}
	}

	MPI_Win_fence(0, window);
	if (id == 0)
	{
		for (int i = 0; i < p; i++)
			MPI_Put(&send_data[i * chunk_size], chunk_size, MPI_INT, i, 0, chunk_size,
				MPI_INT, window);
	}
	MPI_Win_fence(0, window);

	cout << "Data on process " << id << ": ";
	for (int n = 0; n < chunk_size; n++)
		cout << recv_data[n] << "\t";
	cout << endl;
	MPI_Win_free(&window);

	delete[] send_data;
	MPI_Finalize();
}
```

One-sided communication example

Implementing a scatter using MPI_Get

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

const int chunk_size = 5;

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

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	srand(time(NULL) + id * 10);

	int* send_data = nullptr;
	int* recv_data = nullptr;
	MPI_Win window;

	if (id == 0)
		MPI_Win_allocate(sizeof(int) * p * chunk_size, sizeof(int), MPI_INFO_NULL,
			MPI_COMM_WORLD, &send_data, &window);
	else
		MPI_Win_create(nullptr, 0, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
			&window);

	recv_data = new int[chunk_size];

	if (id == 0)
	{
		for (int i = 0; i < p; i++)
		{
			cout << "Data for process " << i << ": ";
			for (int n = 0; n < chunk_size; n++)
			{
				send_data[i * chunk_size + n] = rand() % 1000;
				cout << send_data[i * chunk_size + n] << "\t";
			}
			cout << endl;
		}
	}

	MPI_Win_fence(MPI_MODE_NOPRECEDE, window);

	MPI_Get(recv_data, chunk_size, MPI_INT, 0, id * chunk_size, chunk_size, MPI_INT, window);

	MPI_Win_fence(MPI_MODE_NOSUCCEED, window);

	cout << "Data on process " << id << ": ";
	for (int n = 0; n < chunk_size; n++)
		cout << recv_data[n] << "\t";
	cout << endl;
	MPI_Win_free(&window);

	delete[] recv_data;
	MPI_Finalize();
}
```

I have now added attributes to the MPI_Win_fence. These can be used by the background process to help to optimise communications and even let processes through the fence before other processes are finished if it is safe to do so. Putting down an untrue attribute could cause crashes or unexpected behaviour (it is always fine to put 0, but it might be less efficient).

Note that I have created a window on the other processes even though these have no memory associated with them. This is so that I can use the MPI_Win_fence and to set the size of the displacement unit that I am using (the size of an integer in this case).

MPI_Win_allocate

```
int MPI_Win_allocate(MPI_Aint size,
    int disp_unit,
    MPI_Info info,
    MPI_Comm comm,
    void *baseptr,
    MPI_Win * win)
```

- size is the size of the shared memory to be allocated in bytes
- disp_unit is the size in bytes of the basic memory unit in the memory
  - When writing or reading this memory you need not start at the beginning and can specify a location in the memory in multiples of this unit
  - Typically the size of the memory type (e.g. sizeof(int))
  - Could also make it the size of the data for a single communication if all communications are the same size
- info can contain hints as to how the communications are going to be used
  - E.g. no_locks can be used if you are not going to use passive synchronisation (more on this later)
    - Set using MPI_Info_set(info, "no_locks", "true");
  - Can also be MPI_INFO_NULL
- comm is the communicator as usual
- baseptr is a pointer to the memory buffer's pointer
  - E.g. if you want the memory to be stored in int *data, then basptr is &data
- win is the window struct associated with this memory window and which is used to control access to it

MPI_Get

```
int MPI_Get(void *origin_addr,
    int origin_count,
    MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp,
    int target_count,
    MPI_Datatype target_datatype,
    MPI_Win win)
```

- Reads data from remote memory
- origin_addr is the pointer to where the data that has been read off the other process will be stored
  - Note that this need not be shared memory and can be any valid pointer
- origin_count is the number of items of type origin_datatype that is to be read off the other process
- target_rank is the id of the process that you going to read the remote memory off
- target_disp is the displacement in the remote memory from which you are going to be reading from
  - The size of the displacement step is that set by disp_unit
    - If e.g. disp_unit is the size of an integer then target_disp is the number of integers into the remote memory window you will start reading from
- target_count is the number of items of type target_datatype that is to be read off the other process
  - These will typically be the same as origin_count and origin_datatype, but data can be read as one type and saved as another

MPI_Put

```
int MPI_Get(void *origin_addr,
    int origin_count,
    MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp,
    int target_count,
    MPI_Datatype target_datatype,
    MPI_Win win)
```

- Writes data to remote memory
- Identical to MPI_Get in the form of the parameters
- origin_addr is now the pointer to the data that is to be written to the remote process
  - As with MPI_Get this need not be shared memory and can be any valid pointer
- origin_count is the number of items of type origin_datatype that is to be written to the other process
- target_rank is the id of the process that you going to write the remote memory to
- target_disp is the displacement in the remote memory where the data writing is going to start
- target_count is the number of items of type target_datatype that is to be written to the other process
  - As with MPI_Get ,these will typically be the same as origin_count and origin_datatype

Synchronisation

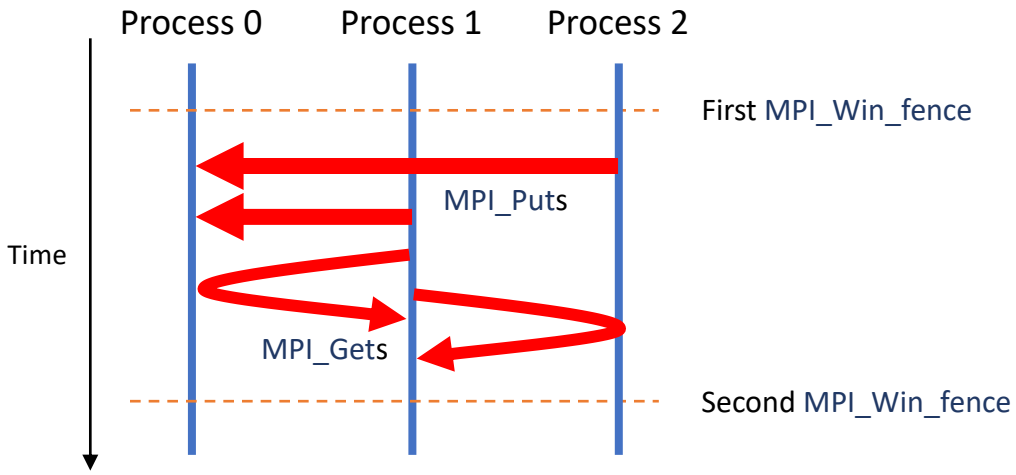
- Even though the communications are one-sided there still needs to be some synchronisation between the processes
  - You can't do multiple MPI_Puts and MPI_Gets to the same shared memory at the same time
    - You have no guarantee of the order of the operations and if they actually occur at the same time you will mangle the memory and get junk out
    - Note that you can do multiple MPI_Puts and MPI_Gets to the same memory window at the same time as long as the actual memory involved does not overlap
  - You also often need to know when the communications are complete so that you can use the data
    - MPI_Puts and MPI_Gets are non-blocking and the actual communications occur in the background. This means that they set up the communications, but the communications are not necessarily complete when the function exits

One-sided MPI communications support three types of synchronisation

- Active synchronisation (Two types)
  - Both the process doing the communication and the process where the data resides are involved in the synchronisation
  - MPI_Win_fence is the simplest form of active synchronisation
    - Blocks all processes
    - Simple to use. Often fine if all processes are involved in a communication cycle ("epoch"), but can otherwise be inefficient
  - Post-Start-Wait-Complete (PSWC) is more powerful if you wish to control individual sets of communications
    - Blocks only those processes involved in a given communication
- Passive synchronisation
  - Only the process that wishes to protect the remotely accessible memory is involved in the synchronisation
    - Potentially simpler code as the synchronisations do not need to be coordinated
    - Potential for lots of inefficiency if done badly as locks will cause other processes to wait for access to the memory – You will have encountered the same issue in OpenMP
  - MPI_Win_lock and MPI_Win_unlock to lock and unlock memory windows

MPI_Win_fence

int MPI_Win_fence(int assert,
MPI_Win win)

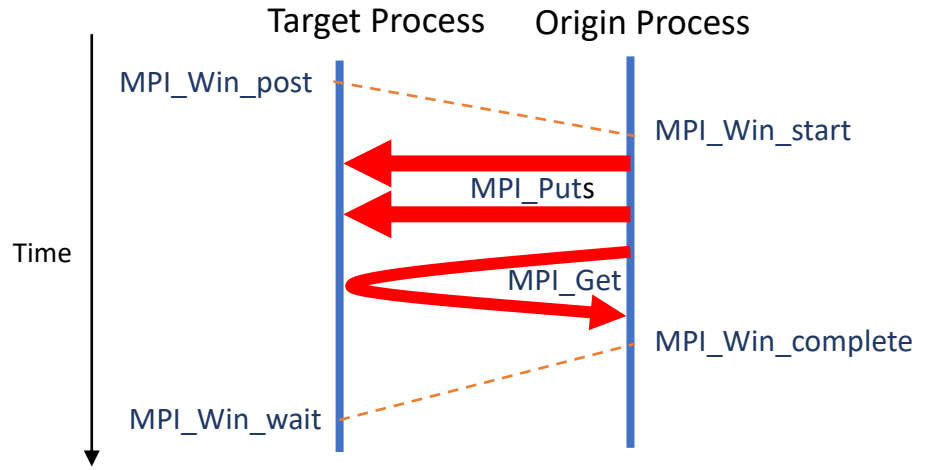


- Used to start and/or end a communication interval
  - Often referred to as a communication epoch
- Has some similarities to a combination of an MPI_Waitall and an MPI_Barrier
  - It waits for all the one-sided communications set up in the preceding epoch to complete
  - Generally only proceeds when all the processes get to this point
- assert tells MPI about the communications and memory writes in either the previous or next epoch to aid optimisation
  - 0 is always a valid assert value
  - There are four different asserts than can be used
    - MPI_MODE_NOSTORE means that the local processor has not changed the shared memory in the proceeding epoch
      - This assert can be different on different processes
    - MPI_MODE_NOPUT means that the shared memory window on the local processor will not be changed by any other process in the next epoch
      - This assert can be different on different processes
    - MPI_MODE_NOPRECEDE means that there should have been no communications in the preceding epoch
      - This assert must be the same on all processes
    - MPI_MODE_NOSUCCEED means that there will be no communications in the subsequent epoch
      - This assert must be the same on all processes
  - Asserts can be combined using an or (|)
    - E.g. MPI_Win_fence(MPI_MODE_NOPUT|MPI_MODE_NOSUCCEED, window); means that there should be no put communications in the previous epoch and no communications at all in the next epoch

Do Worksheet 5 Exercise 1

Post-Start Wait-Complete (PSWC)

- Similar to MPI_Fence in that there are communication epochs
  - Done on the basis of a communication group rather than all the processes
    - Process/es providing the remote memory (the target/s) and process/es either reading or writing the remote memory (the origin/s)
  - The potential efficiency of this synchronisation strategy is from the fact that processes do not need to wait for all the other communications to complete before proceeding
- The target starts a communication epoch using MPI_Win_post and completes the epoch using MPI_Win_wait
- The origin starts the epoch using MPI_Win_start and completes the epoch using MPI_Win_complete
- Note that the MPI_Win_post can be set up in anticipation of the MPI_Win_start and does not block
  - The MPI_Win_start needs the corresponding MPI_Win_post to be in place and will block until this is the case
- MPI_Win_Complete will exit as soon as all the communications set up on that process during the epoch complete
- MPI_Win_wait will exit if MPI_Win_complete has been called on the other process/es otherwise it will wait
- Note that a process can be both a target and an origin at the same time and that these need not involve the same group of processes



PSWC Scatter

- Based on the previous example
  - Everything on this slide is the same as before – no difference in allocating memory and setting up the windows

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

const int chunk_size = 5;

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

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	srand(time(NULL) + id * 10);

	int* send_data = nullptr;
	int* recv_data = nullptr;
	MPI_Win window;
```

```
if (id == 0)
	MPI_Win_allocate(sizeof(int) * p * chunk_size, sizeof(int), MPI_INFO_NULL,
MPI_COMM_WORLD, &send_data, &window);
else
	MPI_Win_create(nullptr, 0, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
&window);

recv_data = new int[chunk_size];


if (id == 0)
{
	for (int i = 0; i < p; i++)
	{
		cout << "Data for process " << i << ": ";
		for (int n = 0; n < chunk_size; n++)
		{
			send_data[i * chunk_size + n] = rand() % 1000;
			cout << send_data[i * chunk_size + n] << "\t";
		}
		cout << endl;
	}
}
```

PSWC Scatter

```
MPI_Comm_group(MPI_COMM_WORLD, &comm_group);

if (id != 0)
{
        group_ids = new int;
        group_ids[0] = 0;
        MPI_Group_incl(comm_group, 1, group_ids, &group);
        //These are the origin processes
        MPI_Win_start(group, 0, window);
        MPI_Get(recv_data, chunk_size, MPI_INT, 0, id * chunk_size, chunk_size, MPI_INT, window);
        MPI_Win_complete(window);
}
else
{
        group_ids = new int[p];
        for (int n = 1; n < p; n++)
                group_ids[n - 1] = n;
        MPI_Group_incl(comm_group, p - 1, group_ids, &group);

        MPI_Win_post(group, 0, window);
        for (int n = 0; n < chunk_size; n++)
                recv_data[n] = send_data[n];
        MPI_Win_wait(window);
}
```

Setup groups of processes involved in the communication. On the origin processes the group consists of all the targets (only process zero in this case)

On the target process the group consists of all the origins (every process except zero in this case)

```
cout << "Data on process " << id << ": ";
for (int n = 0; n < chunk_size; n++)
        cout << recv_data[n] << "\t";
cout << endl;

MPI_Win_free(&window);

delete[] recv_data;
delete[] group_ids;

MPI_Finalize();
}
```

MPI_Comm_group, MPI_Group_incl

```
int MPI_Comm_group(MPI_Comm comm,
    MPI_Group *group)

int MPI_Group_incl(MPI_Group group,
    int n,
    const int *ranks,
    MPI_Group *newgroup)
```

- Used to set up a group of processes to be involved in PSWC communications
- MPI_Comm_group sets up an initial group based on a communicator, comm

- MPI_Group_incl adds a list of processes to include within a group
- group is the original group on which the resultant newgroup will be based
- n is the number of remote processes (ranks) to add to the group
- ranks is an array of ids of the remote processes involved in a communication epoch on a given processor
  - These can be different on every process and should reflect either the processes that are expect to communicate with the current process (if the process is the target) or those that are going to be communicate with (if the process is the origin)

MPI_Win_start, MPI_Win_post

```
int MPI_Win_start(MPI_Group group,
        int assert,
        MPI_Win win)

int MPI_Win_post(MPI_Group group,
        int assert,
        MPI_Win win)
```

- MPI_Win_start and MPI_Win_post are the processes that start a PSWC epoch on the origin and target respectively
  - They also both have the same parameters
- group contains the list of processes involved in the communications on this process
  - Outgoing communications on MPI_Win_start
  - Incoming communications on MPI_Win_post
- assert contains information for the function that will help to optimise the performance. The available assert are different on the target and origin
  - On both 0 is a valid assert
  - MPI_Win_start

    MPI_MODE_NOCHECK
    - MPI_Win_start does not check if the corresponding MPI_Win_post is available and waiting for a communication. This saves on an additional handshake communication, but the code structure must ensure that the MPI_Win_post is actually available or else the code is likely to hang or crash
  - MPI_Win_post

    MPI_MODE_NOCHECK
    - MPI_Win_post does not check if the corresponding MPI_Win_start is available

    MPI_MODE_NOSTORE
    - The local processor isn't going to modify its shared memory window during this epoch

    MPI_MODE_NOPUT
    - The shared memory on the current processor is not going to modified by puts from other processes during this epoch
- win is the memory window associated with this set of communications

MPI_Win_complete, MPI_Win_wait

int MPI_Win_complete(MPI_Win win)

int MPI_Win_wait(MPI_Win win)

- MPI_Win_complete and MPI_Win_wait are the processes that end a PSWC epoch on the origin and target respectively
  - They also both have the same parameter
- win is the memory window associated with this set of communications

Do Worksheet 5 Exercise 2

Passive synchronisation using MPI_Win_lock and MPI_Win_unlock

- MPI_Win_lock and MPI_Win_unlock have similarities to omp_set_lock and omp_unset_lock
  - Generalized from a shared memory paradigm to something that will work across both shared and distributed memory
  - In MPI reading and/or writing to a shared memory window by the MPI processes are non-blocking and so the MPI_Win_unlock will wait until the communications are complete to continue
    - It actually ends an epoch, not just releasing a window for use by other processes
  - Note that you do need to put both the RMA operations (e.g. MPI_Put and MPI_Get) as well as the direct writes to shared memory into lock/unlock sections
- Because the actual results of communications are not guaranteed to be available even if a communication completes, doing multiple operations on the same data within an epoch requires care
  - E.g. You might want to get data (MPI_Get), manipulate it and place it back in the original location (MPI_Put) without releasing the lock and giving other processes the chance to manipulate the data
    - Note that we will look at atomic communications for doing just this, but it will not help with more complex manipulation
  - We can ensure that the communications have completed without ending an epoch by, for example, using MPI_Win_flush

Maintaining a counter using passive synchronisation

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

const int max_count = 100;

unsigned long long fibonacci(int num)
{
	unsigned long long next = 1;
	unsigned long long val[2] = { 1, 1 };
	for (int cnt = 0; cnt < num - 2; cnt++)
	{
		next = val[0] + val[1];
		val[0] = val[1];
		val[1] = next;
	}

	return next;
}
```

```
int main(int argc, char* argv[])
{
	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	srand(time(NULL) + id * 10);

	int* counter = nullptr;
	MPI_Win window;

	int current_count= -1;

	if (id == 0)
		MPI_Win_allocate(sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
			&counter, &window);
	else
		MPI_Win_create(nullptr, 0, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &window);

	if (id ==0)
		*counter = 0;

	//ensures that the system is set before the task counting commences
	MPI_Win_fence(0, window);
```

Maintaining a counter using passive synchronisation

```
    do
    {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, window);
        //read the current count off the memory on process zero
        MPI_Get(&current_count, 1, MPI_INT, 0, 0, 1, MPI_INT, window);

        int increment = 1;
        MPI_Accumulate(&increment, 1, MPI_INT, 0, 0, 1, MPI_INT, MPI_SUM, window);
        MPI_Win_unlock(0, window);

        if (current_count < max_count)
        {
            int num = rand() % 100;
            cout << "Process " << id << " is did task " << current_count << " and the " <<
                num << "th Fibonacci number is " << fibonacci(num) << endl;
        }
    } while (current_count < max_count);

    //fence to ensure that the other process get to read the counter before process 0 exits
    MPI_Win_fence(0, window);
    MPI_Win_free(&window);

    MPI_Finalize();
}
```

Lock the memory so that it is safe for reading and writing

MPI_Accumulate carries out an operation between the input data and the data on the remote memory and stores the answer in the remote memory

Unlock the memory so that other processes can access it. Note that this also ensures that the communications are complete. current_count is only guaranteed to have its correct value after this point (unless an MPI_Win_flush is called)

MPI_Win_lock, MPI_Win_unlock

```
int MPI_Win_lock(int lock_type,
        int rank,
        int assert,
        MPI_Win win)

int MPI_Win_unlock(int rank,
        MPI_Win win)
```

- Used to lock and unlock memory windows in passive synchronisation
  - MPI_Win_unlock also ensures that the communications are complete
- lock_type sets how strict a lock is being applied
  - MPI_LOCK_EXCLUSIVE stops all other process from reading and writing the memory
  - MPI_LOCK_SHARED is usually used if you wish to lock memory for writing but allow reading or if each process is writing to different memory
    - An MPI_LOCK_SHARED will block an MPI_LOCK_EXCLUSIVE, but not another MPI_LOCK_SHARED
    - It is also useful on its own due to the end of the epoch causing the communications to complete
- rank is the id of the process on which the memory is to be locked/unlocked
- assert contains info about the communications involved that can help with optimisation.
  - 0 is again a valid number
  - MPI_MODE_NOCHECK – The code doesn't actually check if the memory is already locked. Again saves a handshake communication, but relies on their being no way that other processes could have locked the memory at that point in the code
- win is the window involved in the communications

MPI_Win_flush, MPI_Win_flush_all and MPI_Win_flush_local

```
int MPI_Win_flush(int rank,
      MPI_Win win)

int MPI_Win_flush_all(MPI_Win win)

int MPI_Win_flush_local(int rank,
      MPI_Win win)
```

- MPI_Win_flush completes all the RMA communications initiated on the current process and accessing the remote memory on the process with id rank
  - This will ensure that the communication is complete with the final values stored on both the local and the remote process
- MPI_Win_flush_all is similar to MPI_Win_flush, but completes all the RMA communications initiated on the current process to all other process
- MPI_Win_flush_local ensures that the local side of a communication has completed.
  - This allows send buffers to be reused or deallocated, but does not guarantee that the full communication has completed
- win is the window involved in the communications

- Note that allowing the behind-the-scenes communications to buffer and delay the transfer of data can make the system more efficient
  - Only force a communication with a flush if you actually need the data or the buffers

Do Worksheet 5 Exercise 3

Atomic communications

- An atomic communication is one that be safely done to a piece of memory at the same time as another process is potentially doing a similar communication
- Note that atomic communications still occur in the background
  - There is no guarantee on the order in which the communications will occur
  - You still need to wait for a synchronisation event to ensure a valid result from the communications is actually stored
- There are a number of available atomic communications, but probably the most useful is MPI_Accumulate
  - This communication reads in data from the current process as well as from remote memory, combines the values using a specified operation and stores the result in the remote memory

MPI_Accumulate

```
int MPI_Accumulate(const void *origin_addr,
    int origin_count,
    MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp,
    int target_count,
    MPI_Datatype target_datatype,
    MPI_Op op,
    MPI_Win win)
```

- The parameters in MPI_Accumulate are virtually the same as those found in MPI_Get and MPI_Put
  - MPI_Accumulate behaves like a get followed by a calculation and then followed by a put of the data back into the original locating
  - In practice it will only actually involve one communication as it would be inefficient to bring the data back to the originating process to do the operation
    - It will be implemented like a put, but where a calculation is carried out on the remote process to combine the existing and the new data
  - Unlike MPI_Put and MPI_get, MPI_Accumulate is atomic, which means that multiple accumulates to the same memory location can be carried out within the same epoch
    - The result must not depend upon the order of the accumulate operations as those are not guaranteed
- op is the only parameter in MPI_Accumulate that does not appear in MPI_Put or MPI_get
  - The op options are the same as those available to MPI_Reduce

Do Worksheet 5 Exercise 4

Final words

- We have only touched the surface of the functionality of RMA and one-sided communications using MPI
- While it will be the most efficient way of solving some problems, it is not a silver bullet for all problems
  - There are problems where conventional two-sided communications will be most efficient
    - E.g. problems that are best solved using collective communications or ones with unusual or changing memory patterns
- It does, though, provide a framework for doing seamless distributed and shared memory parallel computing
- Much of the efficiency of the method is dependent on the implementation of the standard and the architecture of the systems on which they run
  - Implementations will continue to improve and as the standard becomes more popular, the architecture of HPC systems is likely to adapt to better suite RMA based programs