MPI Lecture 2: The 'nitty-gritty'

University of BRISTOL

Overview

- Point-to-Point Pic'n'Mix:
  - Synchronous message passing
  - Buffers & 'blocking' routines
  - Non-blocking patterns
- Support for Recurring Patterns of Communication:
  - Message exchange
  - Collective communication
- Reducing Message Traffic:
  - Using MPI derived types
  - Packed data messages

University of BRISTOL

Point-to-Point Pic'n'mix

MPI offers many, many options for point-to-point messages:

MPI_Send()
MPI_Ssend()
MPI_Bsend()
MPI_Isend()
MPI_Rsend()
MPI_Issend()
MPI_Ibsend()
MPI_Irsend()

MPI_Recv()
MPI_IRecv()

and any sending routine can be paired with either receiving routine

Many Concepts Bundled up

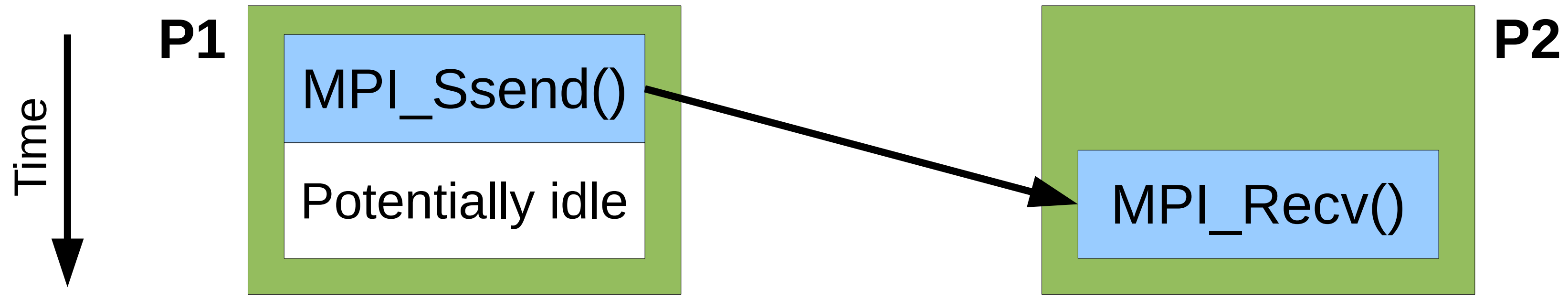
- Blocking vs. Non-blocking
- Buffered or Unbuffered
- Synchronous vs. Asynchronous
- System or User-Buffer
- Safe or Unsafe
- Portable or not Portable

OK, Let's Unravel it All

- Let's start with **`MPI_Ssend(buffer,..)`**.
- This stands for ***synchronous*** send.
- It is ***blocking***, meaning that the call will not return until the function arguments are ***safe for re-use*** in the program.
- It will only return when a matching receive has been posted and data transmission has completed.

`MPI_Ssend()` Pros and Cons

- The call is ***safe*** and also ***portable*** (more of that in a moment).
- But we can anticipate that processes may be ***idle*** waiting for a matching receive to be posted to make the pair.
- Also we can anticipate that ***blocking***, ***synchronous*** communication patterns will be most vulnerable to ***deadlock***.



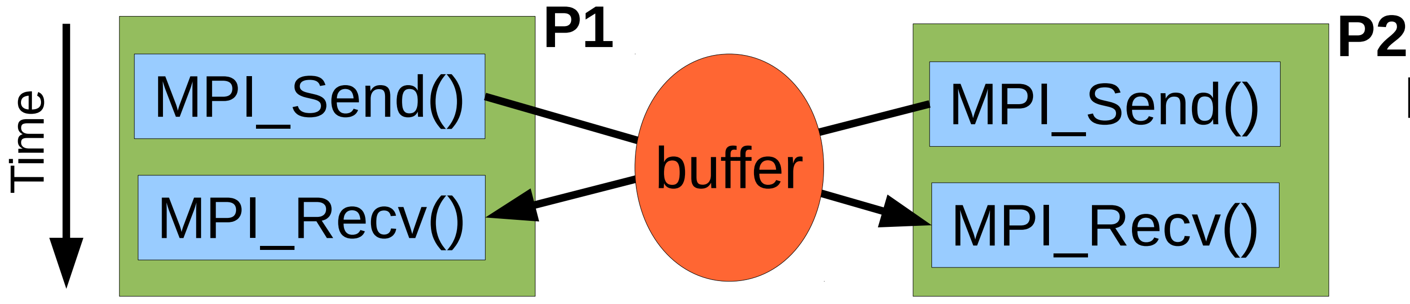
University of BRISTOL

'Standard mode': `MPI_Send()`

- Also ***blocking***, but potentially ***asynchronous***.
- **Potentially**? Why?
  - Some MPI implementations use system buffers, but some do not.
- **If there is a system buffer**, the call will return when the message has been copied from the user-space to the system buffer—i.e. the variable is safe for re-use.
- **Without a system buffer**, **`MPI_Send()`** will behave just like **`MPI_Ssend()`**.

MPI_Send() Pros and Cons

- ***Asynchronous*** communication *may* allow our programs ***run faster***. Why?
- We can use looser communication patterns.
- But copying large messages into a buffer will carry its own ***overheads*** too. **No free lunch**.
- Our programs ***may not be portable*** to other MPI implementations (look at 'deadlock.c'):

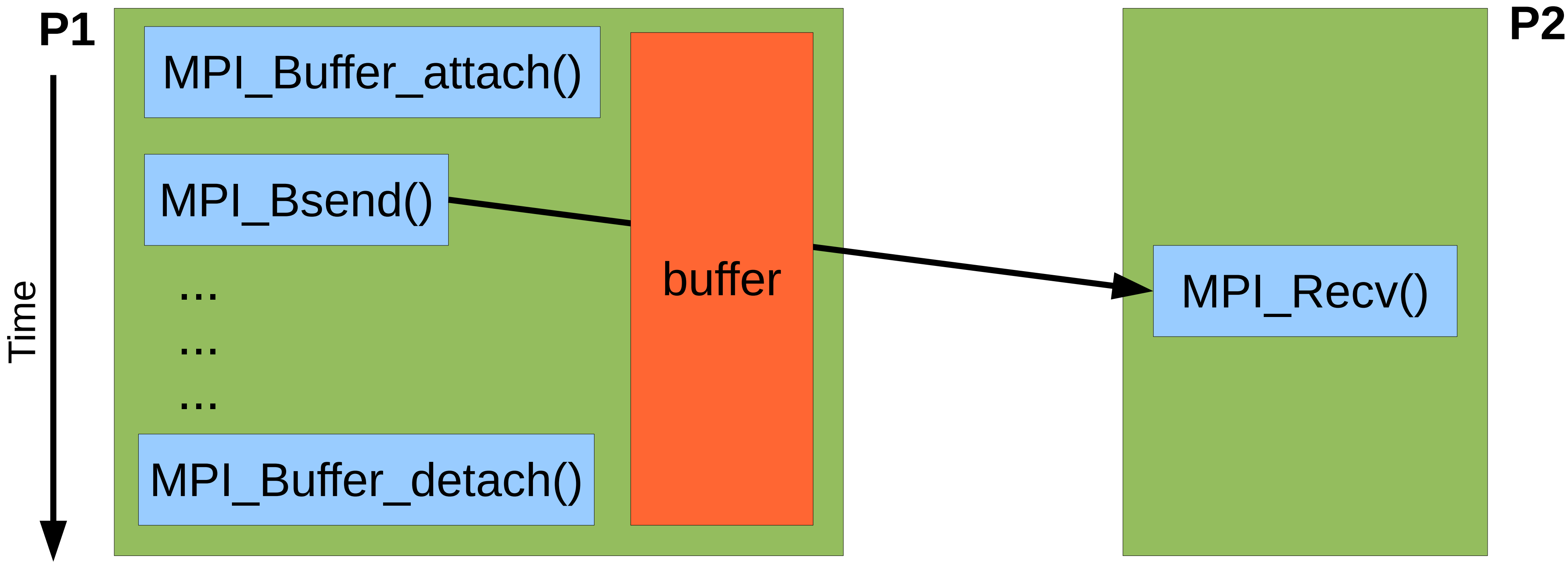


Deadlocks without a system buffer, or if system buffer overfills

University of BRISTOL

User Space Buffers: `MPI_Bsend()`

- A **buffered** (and so **asynchronous**), **blocking** send.
- The buffer is explicitly allocated and managed (attached & detached) by the programmer.



University of BRISTOL

`MPI_Bsend()` Pros and Cons

- **Asynchronous** and **portable,** as user guarantees that buffer is available and of sufficient size.
- But memory allocation is expensive and attaching & detaching will carry **overheads**. You'd better be sure that it's worth it!
- Adds **extra complexity** to your code.
- Different MPI implementations vary in rules around attach and detach calls, so potential **portability problems**.

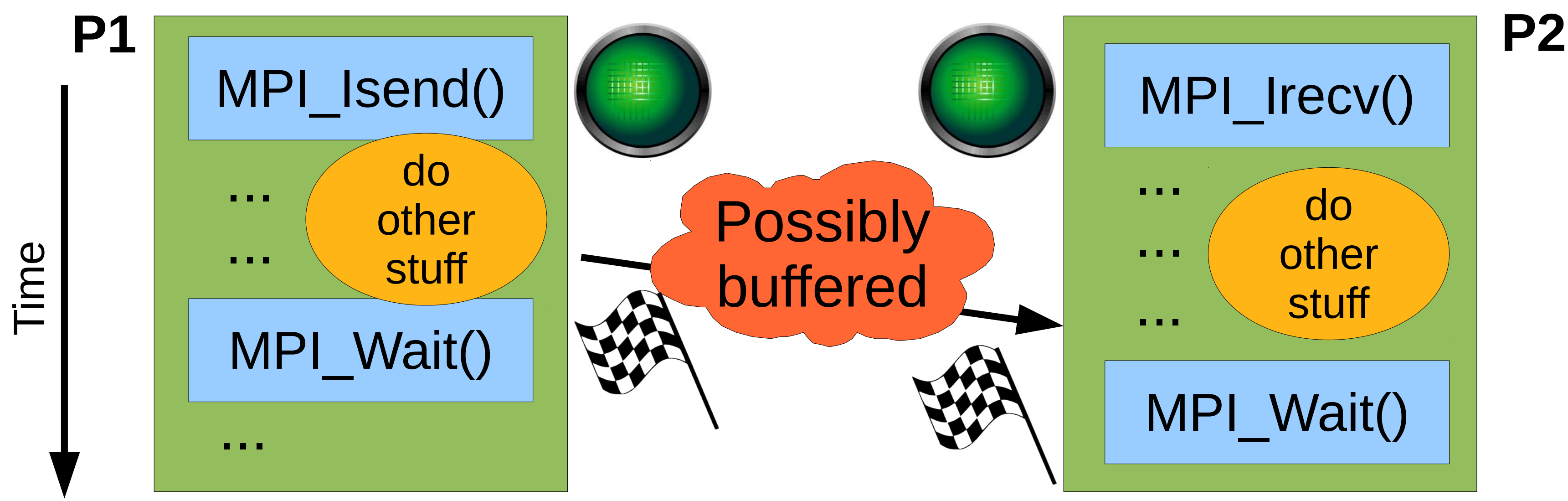
Non-blocking **`MPI_Isend()`**

- ..and counterpart **`MPI_Irecv()`**.
- Is **non-blocking** and so returns (almost) immediately.
- Communication is only **requested** at that point.
- Must test or wait for completion later in your code (via extra request argument).
- Function arguments will not be safe for re-use until the communication is complete.
- 'Standard mode' behaviour applies:
  - i.e. there are also explicitly buffered and synchronous (non-buffered) versions.

University of BRISTOL

`MPI_Isend()` Pros and Cons

- Best chance yet to overlap computation and communication. (This manoeuvre is potentially key for good scaling..)
- But code is more complex and we run the risk of introducing some hard to find bugs. So you'd better be sure that it's worth it..



University of BRISTOL

There's more!

- To explore at your own leisure..;)
- Notably:
  - '**Ready**' send: **`MPI_Rsend()`** (recv already posted)
  - **Persistent** communications: **`MPI_Send_init()`** & **`MPI_Start()`**.
- Example programs for point-to-point are given in examples 2-4 in the MPI practical.

University of BRISTOL

Support for Recurring Patterns

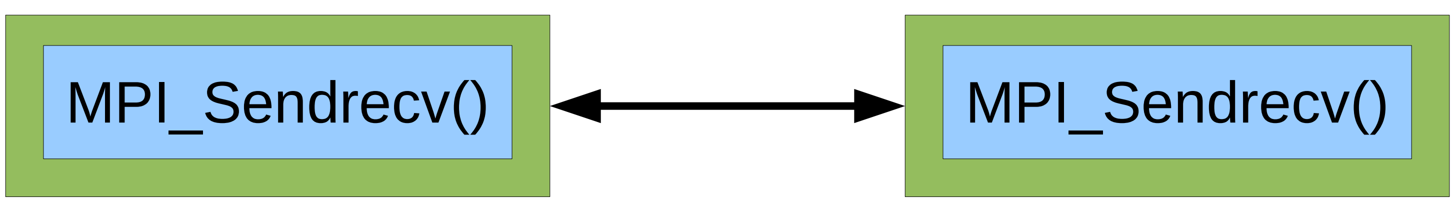
- We've seen many options for point-to-point communications.
- In principle, point-to-point patterns are all that we need to write any program.
- However, we see some compound patterns of communication cropping up again and again.
- MPI is obliging enough to provide routines for these which save us effort and make our programs faster.

University of BRISTOL

For example: Message Exchange

```
int MPI_Sendrecv(void*         sendbuf,
                 int           sendcount,
                 MPI_Datatype  sendtype,
                 int           dest,
                 int           sendtag,
                 void*         recvbuf,
                 int           recvcount,
                 MPI_Datatype  recvtype,
                 int           source,
                 int           recvtag,
                 MPI_Comm      comm,
                 MPI_Status*   status)
```

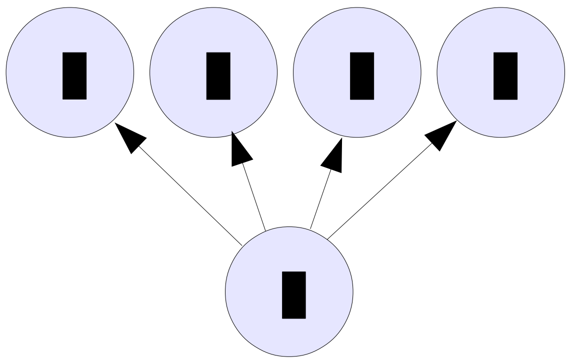
Provide buffers for both send and recv operations



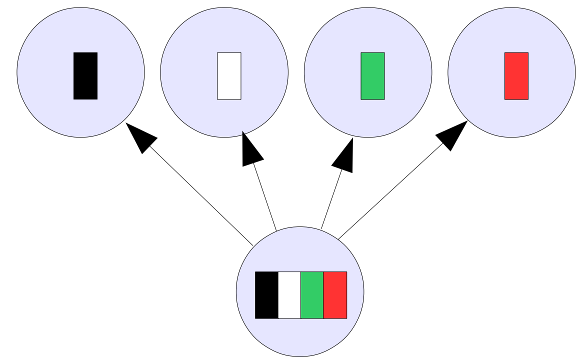
University of BRISTOL

Collective Communications

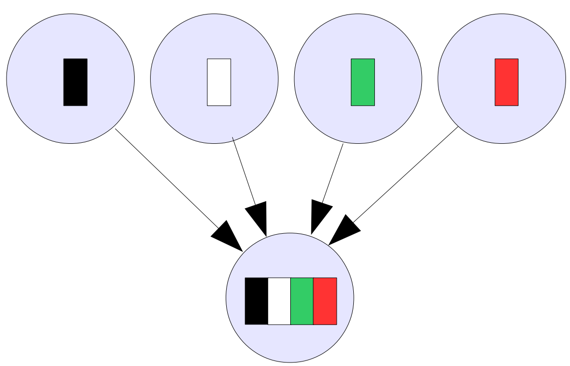
broadcast



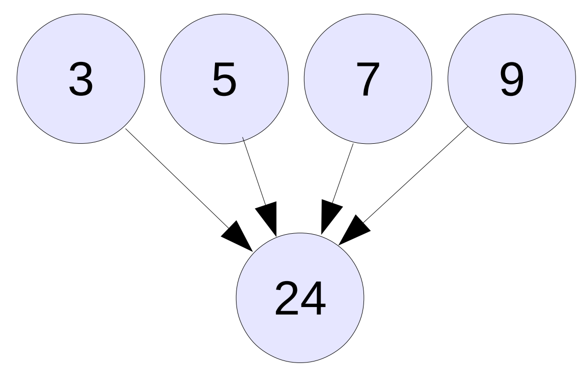
scatter



gather

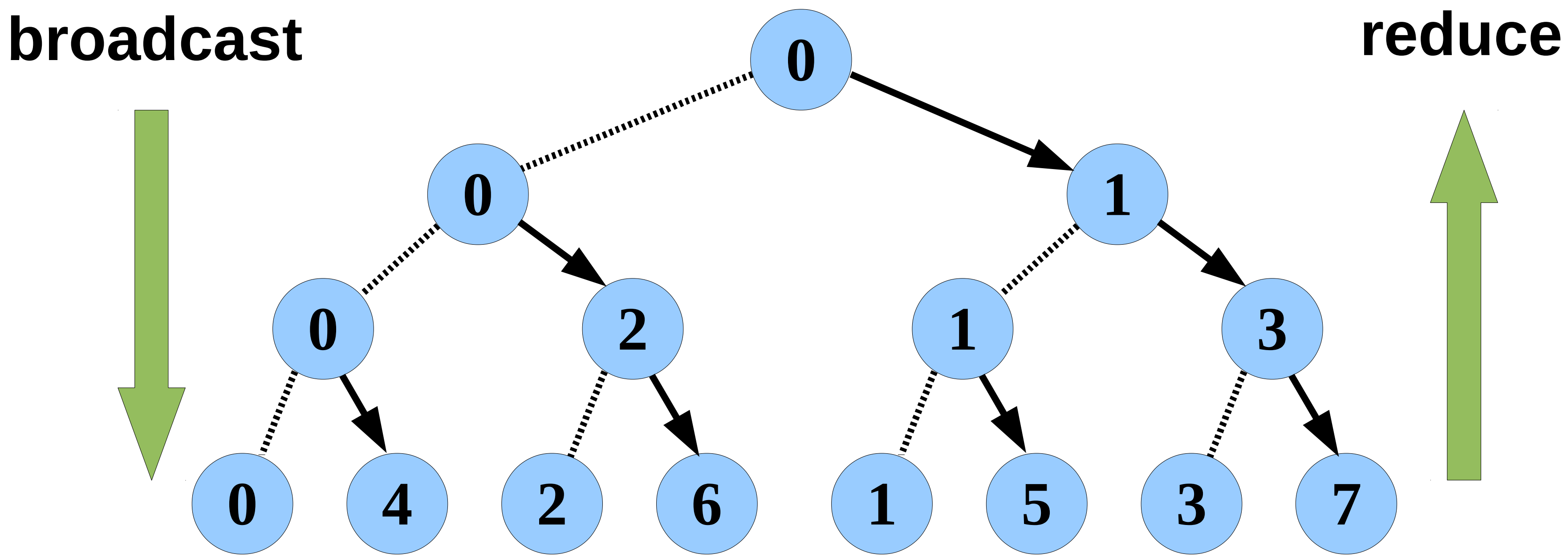


reduction



University of BRISTOL

Tree-Based Load Balancing



- Time taken: 3 units rather than 7, for an 8 process cohort.
- **Big benefits** of $O(\log_2 N)$ vs. $O(N)$ as the **cohort size increases**, $\log_2(1024) = 10$. A saving of a factor of 100!
- With system buffering, collective comms are asynchronous, but are a point of synchronisation if no system buffer is available.
- Usage will be shown in example 5 of the practical.
- **Benefits** are **amplified for higher-latency/lower-bandwidth** networks

University of BRISTOL

Collective Comms: There's more..

- MPI offers several more options for collective communication.
- These include:
  - **MPI_Gatherv()**
  - **MPI_Alltoall()**
  - **MPI_Allreduce()**
  - **MPI_Allgather()**
  - **MPI_Reduce_scatter()**
- Should they come in handy for you..

Message Traffic & Latency

- Given that message latencies and communication overheads always exist, it is sensible to try to reduce message traffic.
- One way to do this is to combine information of different datatypes into a single message.
- MPI gives us two options for this:
  - Using MPI derived types.
  - Packed data messages.
- However, beware of keeping a process waiting for data as it will lead to idle time. Potential benefits depend on context..

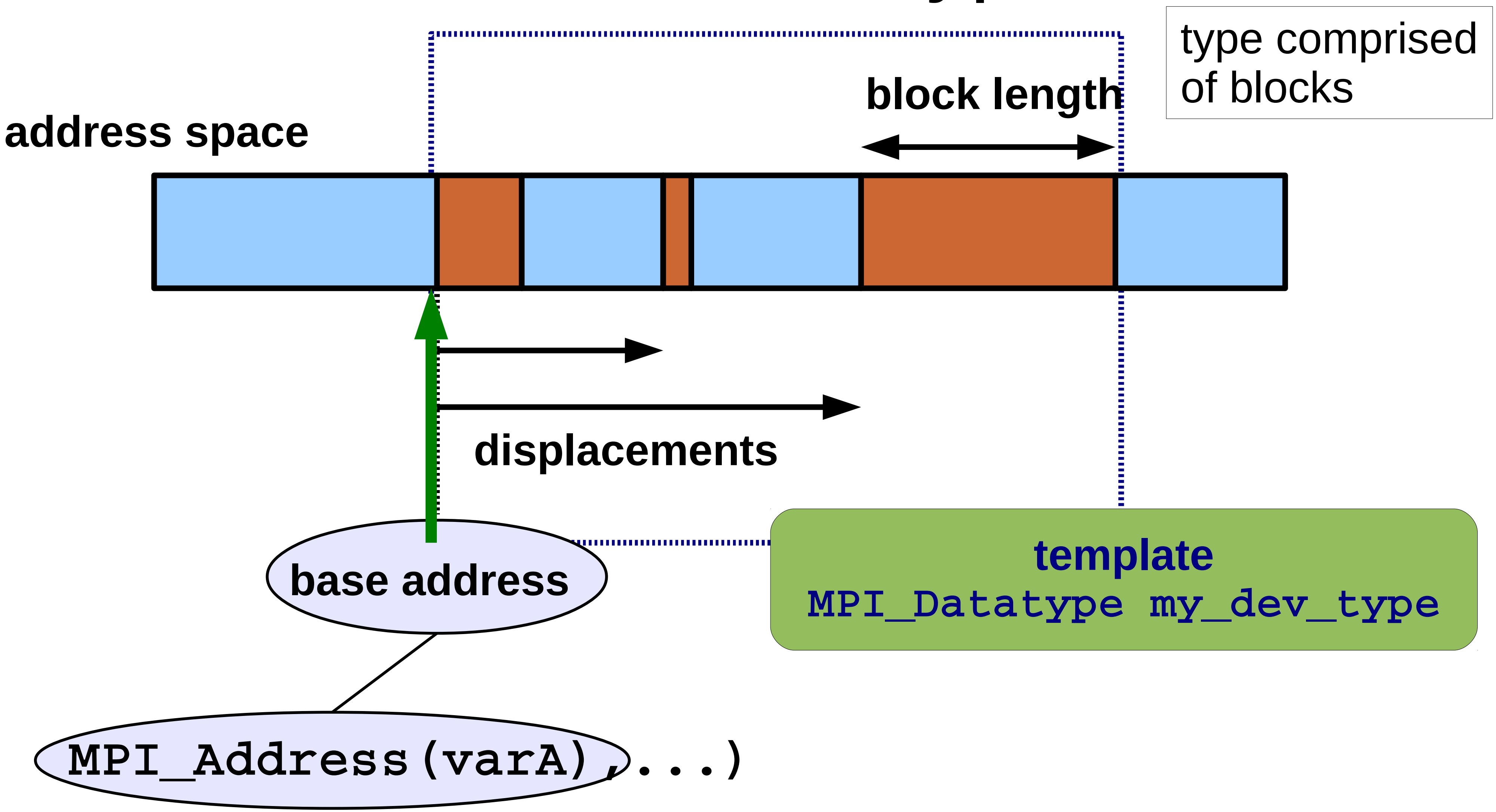
University of BRISTOL

MPI Derived Types

- C derived types are not guaranteed to be stored as contiguous elements in memory.
- Thus we cannot send them as messages.
- The MPI solution is to construct a sequence of pairs: $\{(t_0,d_0),(t_1,d_1),..,(t_{n-1},d_{n-1})\}$, where:
  - t_x is the datatype,
  - d_x is the displacement in bytes from a base address

University of BRISTOL

MPI Derived Types



See example 6

Example 6

```
block_lengths[0] = block_lengths[2] = 1;
block_lengths[1] = STRLEN;

typelist[0] = MPI_FLOAT;
typelist[1] = MPI_CHAR;
typelist[2] = MPI_INT;

displacements[0] = 0;

MPI_Address(&station_freq, &base_address);
MPI_Address(&station_name, &address);
displacements[1] = address - base_address;

MPI_Address(&station_preset_num, &address);
displacements[2] = address - base_address;

MPI_Type_struct(NTYPES, block_lengths, displacements,
             typelist, &my_dev_type);

MPI_Type_commit(&my_dev_type);
```

MPI Derived Types

- Block lengths allow for elements to be in arrays.
- Note: to ensure portability, addresses are stored in the type **`MPI_Aint`** and we call **`MPI_Address()`** to determine the address of an element. (i.e., don't use C's '&'.)
- We pass the base address as the message and the derived type as the datatype to the sending routine and the MPI library constructs the sequence of bytes for us.

Packed Data Messages

- Another option is to fill a buffer (array) with copies of variables using calls **`MPI_Pack()`**.
  - Mixed types and multiples allowed.
- The constructed buffer is contiguous in memory and so can be transmitted as a regular message.
- The buffer can be unpacked at the receiving end with analogous calls to **`MPI_unpack()`**.
- Since this approach involves (many) copies, using a derived type may be more efficient.

MPI2: Recap

- Many point-to-point options. Aspects to consider include **blocking**, **buffering** and (**a**)**synchronous** communication.
- **Buffering** raises **portability** issues.
- **Collective communication** functions can be very convenient and also highly efficient.
- MPI offers approaches to send **compound messages**, which can save on network traffic and ameliorate cumulative latency costs.