XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 10: Parallel data reorganisation

Previous lectures

In the last lecture we saw how to perform **explicit point-to-point communication** in a distributed memory system:

- Send data (*i.e.* an **array** or **sub-array**) from one **process** to another.
- In MPI (where processes are identified by their **rank**):
  - Sending process calls `MPI_Send()`.
  - Receiving process calls `MPI_Recv()`.

- Both **blocking** calls that do not return until resources can be safely modified.
- Can result in **deadlock**, *e.g.* cyclic communication pattern.

This lecture

In this lecture we are going to look at one of the key considerations for the **performance** of distributed memory systems: **Data reorganisation**

- Often necessary for both **shared** and **distributed** memory systems.
- For distributed systems, data reorganisation can result in a significant **parallel overhead**.

- Improved performance using **collective communication routines**.
- Will go through a worked example of a simple **distributed counting algorithm**.

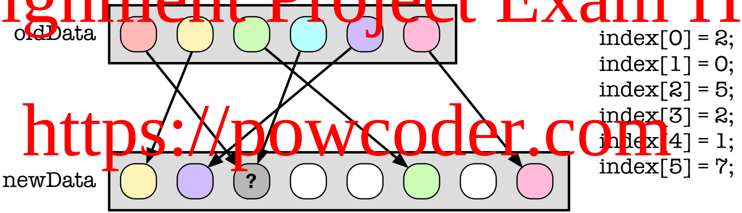
Data reorganisation

Many algorithms require some form of **large-scale data reorganisation**:

- **Sorting**.
- Adding or removing items from a **container** (*i.e.* vector, stack, queue *etc.*) or a **database**.
- **Numerical algorithms**, *i.e.* reordering columns and rows in a matrix.
- **Compression** (*e.g.* `bzip`, `gzip` *etc.*)
- ...

Generalised scatter and gather

General scatter:[1]



In serial code:


```
for( i=0; i<N; i++ )
  newData[ index[i] ] = oldData[i];
```

General **gather** is similar, but indices give **read** locations.

[1] McCool *et al.*, *Structured parallel programming* (Morgan-Kaufman, 2012).

Shared *versus* distributed

Data reorganisation in shared memory systems can lead to a **data race** or **race condition**:

- *e.g.* the **scatter collision** on the previous slide, which arises because `index[0]==index[3]`.
- May require some form of **synchronisation** to resolve, with associated performance penalty.

Although data races are not relevant to distributed memory systems, data reorganisation is very important for **performance**:

> The primary overhead in distributed systems is **communication**, which is a form of data reorganisation

Communication performance

Although most of the overheads in Lecture 4 apply to distributed systems, one typically dominates. The **communication time**.

If the summed times spent for **communicating** and **performing calculations** are t_{comm} and t_{comp} respectively, then[1]

$$t_p = t_{\text{comm}} + t_{\text{comp}}$$

For communication to *not* adversely affect performance, we want the ratio

$$\frac{t_{\text{comm}}}{t_{\text{comp}}}$$

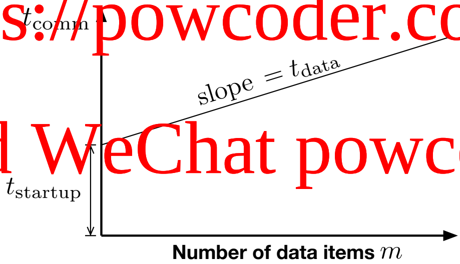
to be **as small as possible**.

[1] Recall from Lecture 4 that t_p is the parallel execution time.

Analysis of t_{comm}

For a single message of size m, a good approximation to t_{comm} is[1]:

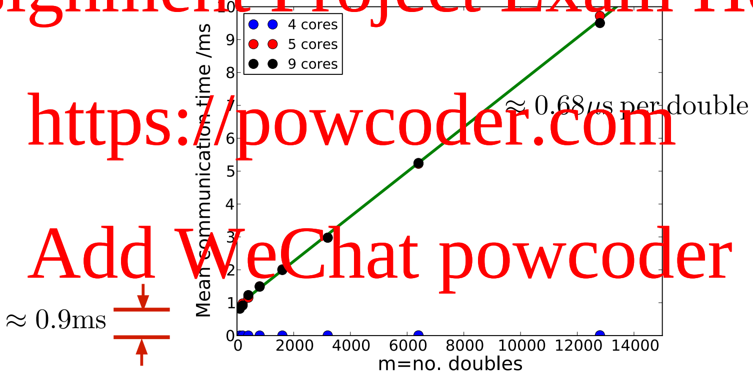
$$t_{\mathrm{comm}} = t_{\mathrm{startup}} + m t_{\mathrm{data}}$$



[1]Wilkinson and Allen, *Parallel programming* 2nd ed. (Pearson, 2005).

Measurement of t_{comm} from SoC lab machines (in Leeds)

Code on Minerva: `measure_tComm.c`

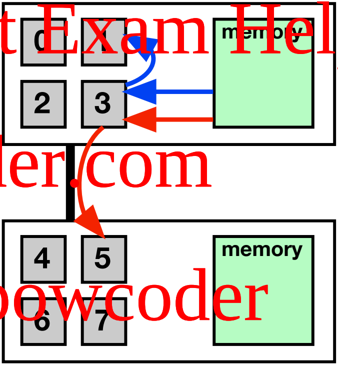


For faster interconnects both t_{startup} and t_{data} about 10 times smaller, but **communication remains the primary overhead**.

Intra- *versus* inter-node communication

Process 3 sends data to process 1, it is **copied within the same machine's memory** *(blue arrows)*.

- **Fast**[1].



If process 3 now sends the *same* data to process 5, it is sent **over the network** *(red arrows)*.

- **Slow**

[1]Could be removed by using one *multi-threaded* process per node *[Lecture 8]*.

Strategies to reduce communication times

This t_{comm} suggests we should **merge** messages when possible:

- For **two** messages of size m and n:

$$t_{\text{comm}} = (t_{\text{startup}} + mt_{\text{data}}) + (t_{\text{startup}} + nt_{\text{data}})$$
$$= 2t_{\text{startup}} + (m+n)t_{\text{data}}$$

- For **one** message of size $m+n$:

$$t_{\text{comm}} = t_{\text{startup}} + (m+n)t_{\text{data}}$$

So we have **saved** t_{startup} in total communication time.

We will see another strategy in Lecture 12.

Collective communication

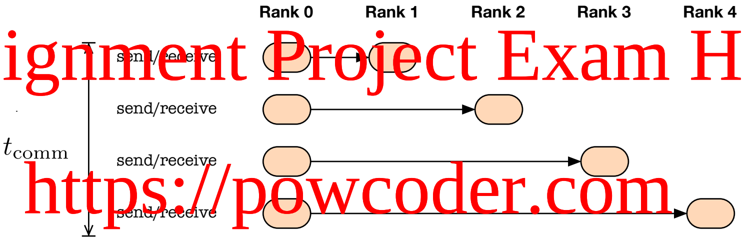
An alternative and often easier way to reduce communication times is to use **collective communication**:

- **All** processes involved in **one** communication.
- Sometimes referred to as **global communication**.

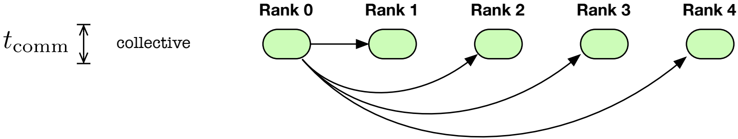
Distributed programming APIs include **optimised** routines for common communication patterns.

- Can **drastically reduce** the communication overhead.
- Implementation varies, but typically **overlap** communications to reduce t_{comm}.

Point-to-point communication:



Collective communication (ideal case):



Common forms of collective communication

Distribution	Type	Meaning
One-to-many	**Broadcast**	**Same** data from one process copied to many
	Scatter	Data from one process **distributed** across many
Many-to-one	**Gather**	Data from many processes **combined** on one process
	Reduction	Data from many processes **accumulated** on one process.

Other variants (*i.e. many-to-many* such as *multi-broadcast*) exist but are less commonly used and not considered here.

We will consider *reduction* next lecture.

Collective communication in MPI

Code on Minerva: `distributedCount.c`

To demonstrate collective communication in MPI we will use a simple worked example: A **distributed count** algorithm.

1. Rank 0 communicates the data size to all other ranks.
2. Rank 0 distributes the data to all ranks.
3. Each rank (including rank 0) counts how many of their local data are below some threshold.
4. All ranks>0 send their counts to rank 0, which determines the total.

Note we assume only rank 0 knows the total data size.

- *e.g.* if rank 0 had loaded the data from a file.

Step 1: Broadcasting: MPI_Bcast()

Sending the variable localSize to all processes can be performed using point-to-point communication:

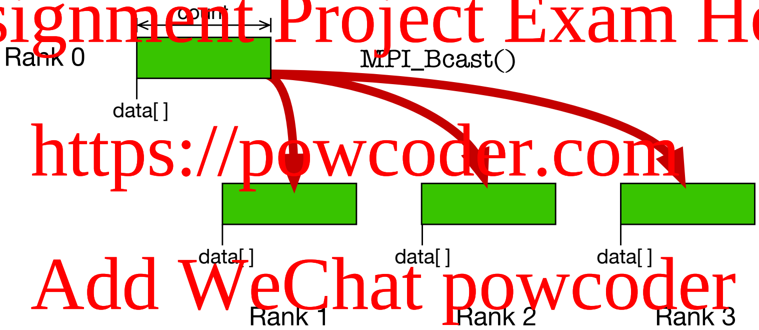
```
if( rank==0 )
  for( p=1; p<numProcs; p++ )
    MPI_Send(&localSize,1,MPI_INT,p,...);
else
  MPI_Recv(&localSize,1,MPI_INT,0,...);
```


The same thing can be achieved using MPI_Bcast():


```
MPI_Bcast(&localSize,1,MPI_INT,0,MPI_COMM_WORLD);
```

- First 3 arguments same as MPI_Send()/MPI_Recv().
- Fourth argument is the rank on which localSize is defined.

Broadcasting: Schematic



Note that using '`&variable`' for the data argument 'fools' MPI into thinking the variable `count` is an array of size 1.

Common pitfall - careful!

When using collective communication, it is important to realise that **all** processes are involved.

- **Must** be called by **all** ranks.

This will fail:

```c
if( rank==0 ) MPI_Bcast(...);
```


- `MPI_Bcast()` does not return until called by **all ranks**.
- Ranks >0 do **not** call `MPI_Bcast()` in the example.
- Rank 0 will wait forever - **deadlock**.

The name *broadcast* is misleading as it suggests only **sending** is involved, whereas in fact it also includes the **receiving**.

Step 2: Scattering: MPI_Scatter()

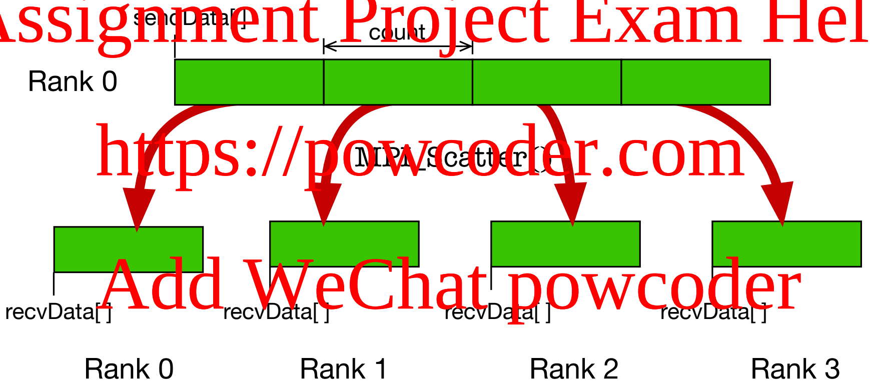
Need to break up an array into equal sized chunks and send one chunk to each process *[cf. vector addition last lecture]*:


```
if( rank==0 )
  for( p=1; p<numProcs; p++ )
    MPI_Send(&data[p*localSize],localSize,...,p,...);
else
  MPI_Recv(localData,localSize,...,0,...);
```

This can be replaced with a single call:


```
MPI_Scatter(
  globalData,localSize,MPI_INT, // Sent from
  localData ,localSize,MPI_INT, // Received to
  0, MPI_COMM_WORLD             // Source rank 0
);
```

Scattering: Schematic



Note also copies to `recvData[]` on rank 0.

Step 4: Gathering: MPI_Gather()

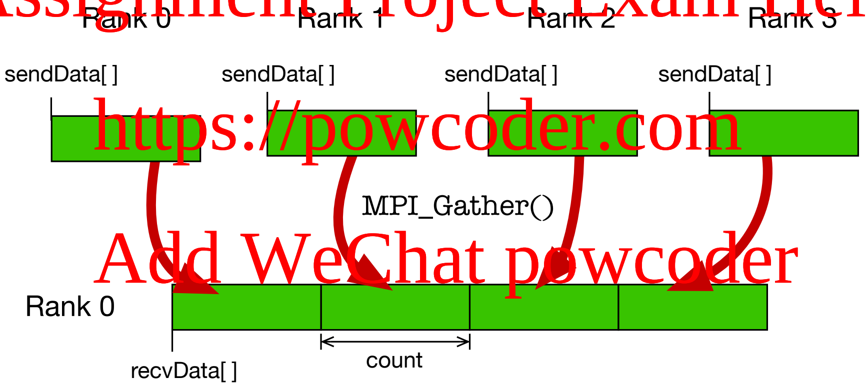
Gathering is the opposite of scattering:

```
MPI_Gather(
  &count   , 1, MPI_INT,      // Sent from
  partials, 1, MPI_INT,       // Received to
  0, MPI_COMM_WORLD           // Destination rank 0
);
```


This gathers all local counts into the array `partials[numProcs]`, from which the total can be counted. As with scattering:

- Data is ordered **by rank**.
- There is no tag.
- The data size is the **local** size, both times.
- Can **in principle** use different data sizes or types for sending and receiving, but not recommended.

Gathering: Schematic



Summary and next lecture

Today we have looked at **data reorganisation** and **collective communication**

- Generalised scatter can cause **collisions** in shared memory systems.

- Common communication patterns in distributed memory systems can be handled **efficiently** by specialised routines.
  - **Broadcasting** (*e.g.* `MPI_Bcast`).
  - **Scattering** (*e.g.* `MPI_Scatter`).
  - **Gathering** (*e.g.* `MPI_Gather`).

In fact, the last stage of our example involved data reorganisation **and calculation**.

- This **reduction** is the subject of the next lecture.