High-Performance Computing

02616

Large-scale Modelling

Scaling and data-types

`mpi4py`

Scaling – how to

Scaling is an intrinsic component of understanding large scale applications
An application will have to explain it's limitations by means of scaling plots!

https://hpc-wiki.info/hpc/Scaling

Read every week!
Details here should
be 2nd nature!

Scaling: strong vs. weak

- How does the execution time go down for a fixed problem size by increasing the number of PUs?
  - Amdahl's law ⇒ speed-up, i.e. reduce time
  - also known as “strong scaling”
- How much can we increase the problem size by adding more PUs, keeping the execution time approx. constant?
  - Gustafson's law ⇒ scale-up, i.e. increase work
  - also known as “weak scaling”

Strong scaling

Fixed problem size, increase processors

1. Decide on problem size `N`
2. Get execution time (`T`) for 1, 2, 4, 8, …, `NP` processors, same `N`
3. Plot `NP` vs. `T(1)/T(NP)`

A program has a serial fraction *s* and a parallel fraction *p*.
In the ideal world one would assume that:
- *s* is constant, irrespective of `NP`
- *p* is constant, and that the time of the parallel fraction becomes *p*/`NP`

So the execution time is $T(NP) \propto s + p/NP$

Amdahl's law

scaling $\rightarrow$ $$S = \frac{T(1)}{T(NP)} \propto \frac{s+p}{s+p/NP} = \frac{1}{s+p/NP}$$

Weak scaling

Scale the problem and the number of processors by the same factor

1. Decide on initial problem size `N`
2. Get execution time (`T`) for 1, 2, 4, …, `NP` processors, scale `N` with `NP`

   (i.e. problem size gets doubled when doubling the number of processors)
3. Plot `NP` vs. `T(1)/T(NP)`

Serial time for scaled problem

A program has a serial fraction *s* and a parallel fraction *p*.
In the ideal world one have:
- *s* is constant, irrespective of `N`
- *p* is constant, and the total parallel time in the scaled problem is *p*

So the scaled speedup is $S(NP)=\frac{\tau(NP)}{\tau(1)}=\frac{s+p\cdot NP}{s+p}=s+p\cdot NP$

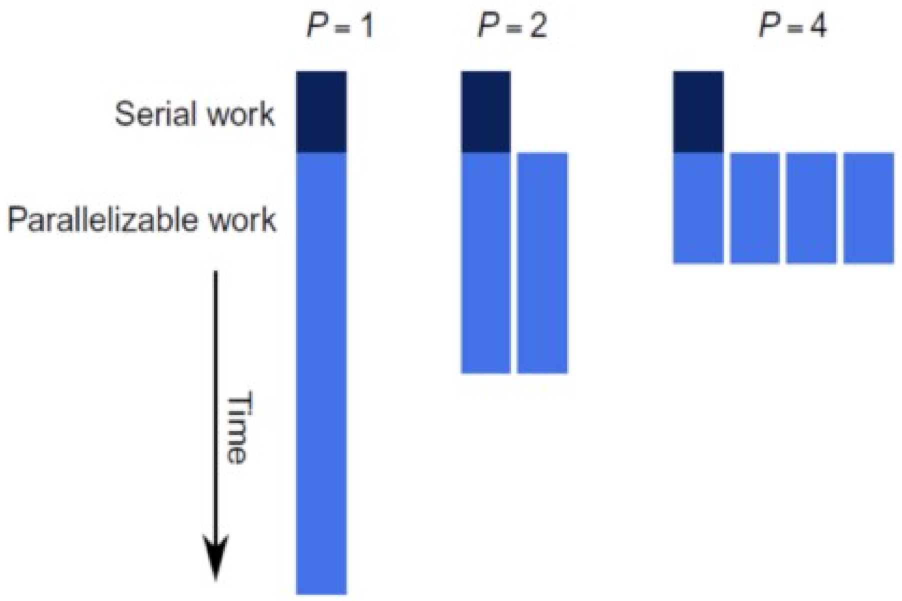
Gustafson's law

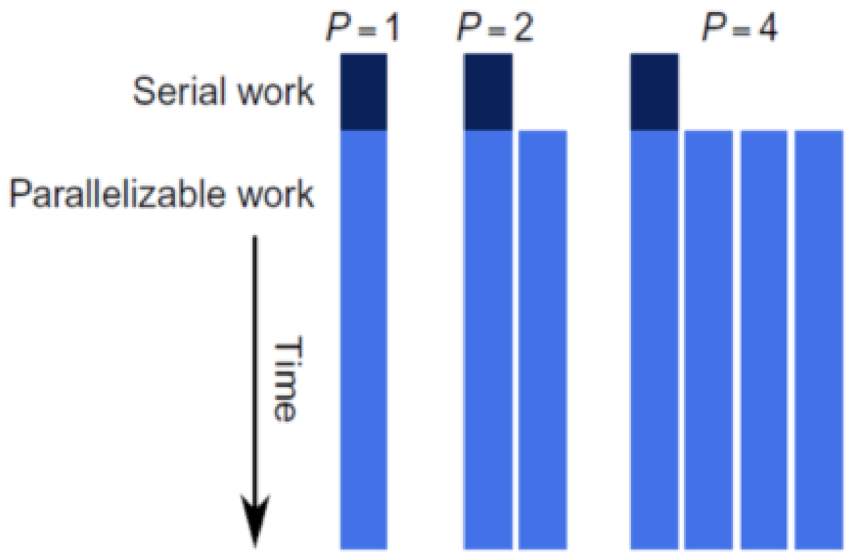
Efficiency $\rightarrow$ $E(NP)=\frac{T(1)}{T(NP)}$

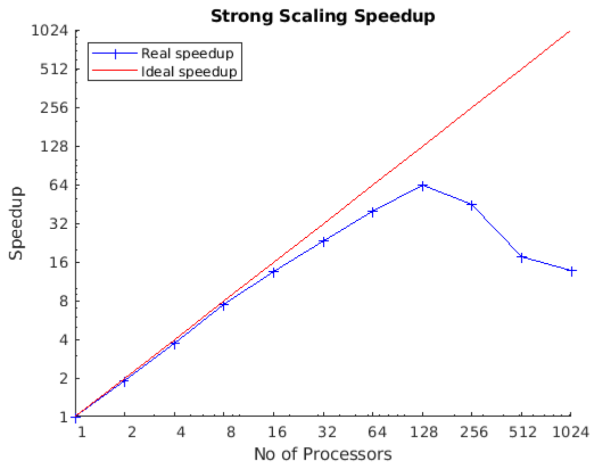
Amdahl's vs Gustafson's law

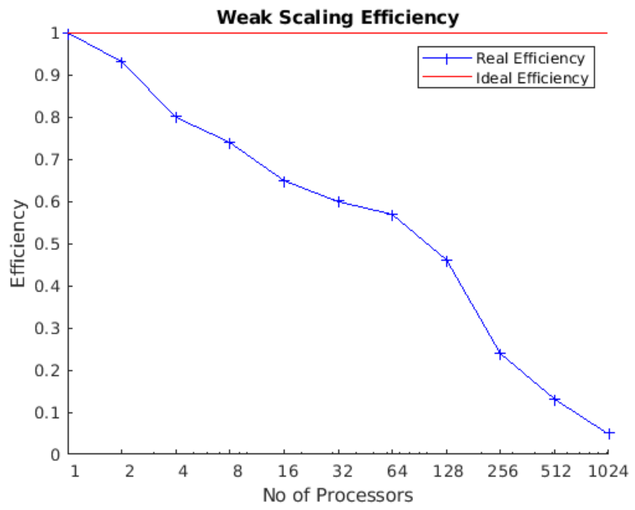
Amdahl: fixed work

Gustafson: fixed work/PU









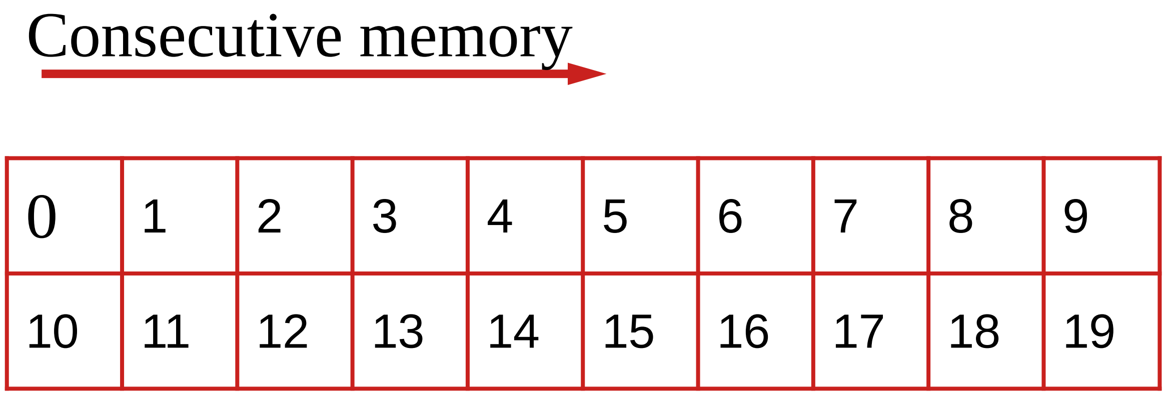
Amdahl's vs Gustafson's Law

- Amdahl's law
  - Theoretical performance of an application with a ***fixed amount of parallel work*** given a particular number of Processing Units (PUs)
- Gustafson’s Law:
  - Theoretical performance of an application with a ***fixed amount of parallel work per PU*** given a particular number of PUs

Custom data-types in MPI

… on to something completely different …

Custom data-types – contiguity

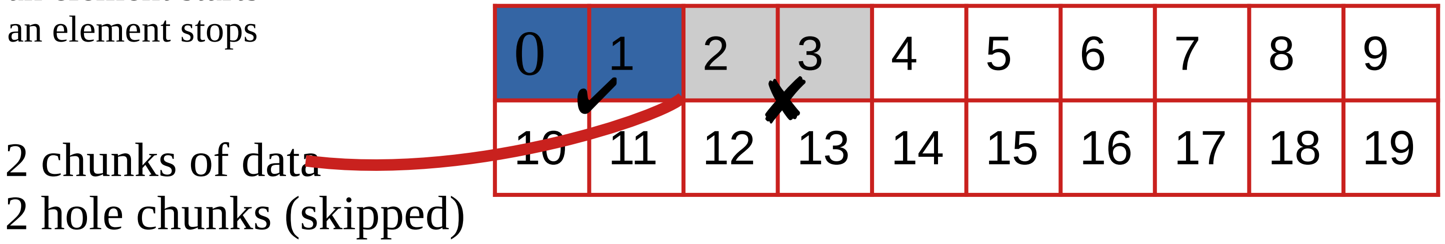


`MPI.Send([buf, N, dtype], …)`

Sends *N* consecutive elements, each of a byte size determined by `dtype`. (`N dtype.itemsize`)

Recall week 4 where we sent a complex array with a float data type, resulting in only sending half of the complex number!

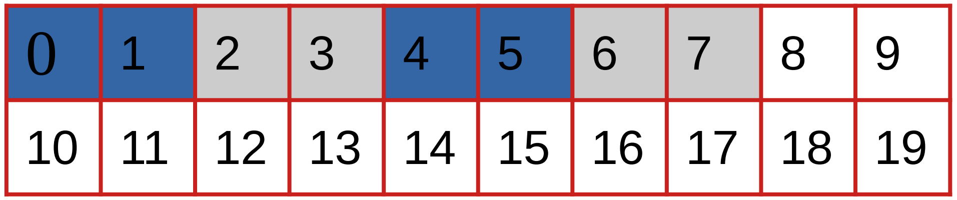
Boiling it down: a data-type tells MPI several details:
• An initial byte offset from the memory segment a user passes
• How many bytes a single element really is (float64 == 8 bytes)
• Where an element starts
• Where an element stops



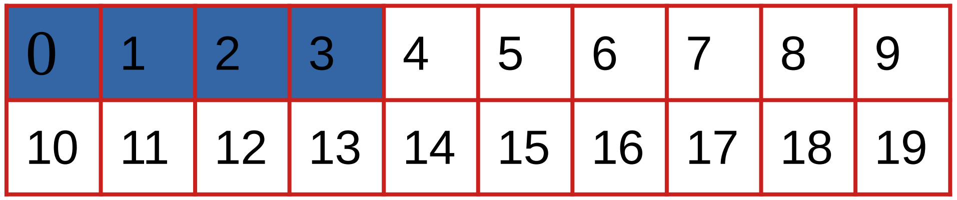
Custom data-types – contiguity

Recall – sender and receiver need not use same data-type

Sending rank :
2 of (2*double, 2*hole)



Receiving rank:
4 of (1*double) or
2 of (2*double) or
1 of (4*double)



It is just a byte-stream!

Custom data-types – contiguity

Data-type

Number of contiguous elements

```
Datatype.Create_contiguous(int count)
```

When you have created your data-type; tell MPI to *use* it!

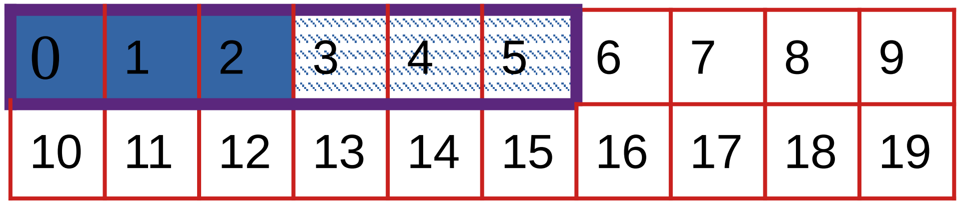
```
Datatype.Commit() # enables it to be used by MPI
Datatype.Free() # remove it from MPI
```

Easy to forget!

Simplest when packing data.

Used: Always sending N of chunks of specific size

E.g. a coordinate system (xyz).



```
dt = MPI.DOUBLE.Create_contiguous(3)
dt.Commit() # necessary!
Comm.Send([xyz, 2, dt])
```

Custom data-types – contiguity

Data-type

Number of contiguous elements in each block

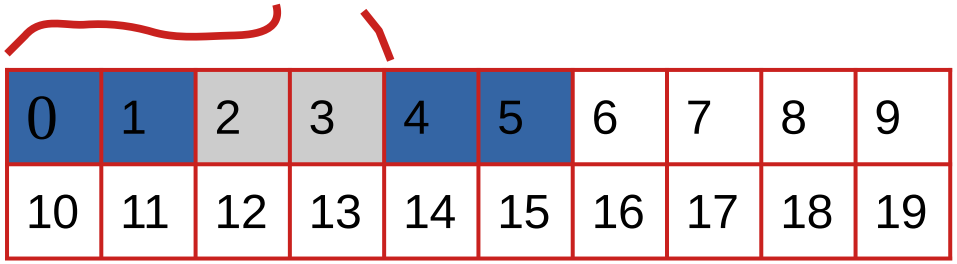
```
Datatype.Create_indexed(blocklengths, displacements)
```

Displacement of `DataType's` size for each block

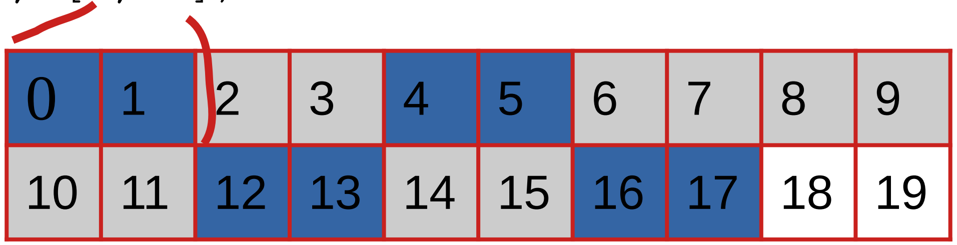
```
Datatype.Commit() # enables it to be used by MPI
Datatype.Free() # remove it from MPI
```

Easy to forget!

```
dt1 = MPI.DOUBLE.Create_indexed([2, 2], [0, 4])
```



```
dt2 = dt1.Create_indexed([1, 1], [0, 2])
```



Custom data-types – contiguity

Number of contiguous elements in each block

Displacement (in BYTES) for each block

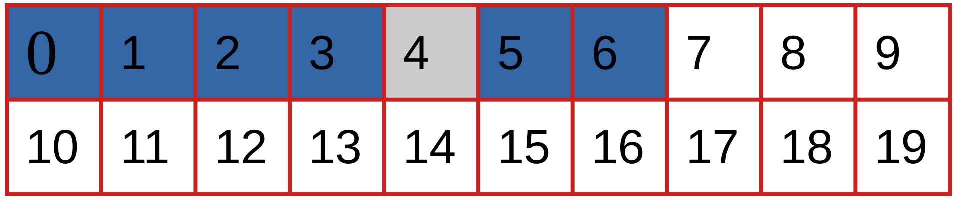
```
MPI.Datatype.Create_struct(blocklengths, displacements,
                           dtypes)
```

Dtypes defining extent through `blocklengths`

```
dt1 = MPI.Datatype.Create_struct([2, 2], [0, 20], [MPI.DOUBLE, MPI.INT])
```

- use 2 doubles (16 bytes)
- jump to 20 byte
- use 2 integers (8 bytes)

An element is 4 bytes



Custom data-types – holes

Lower bound (in BYTES)

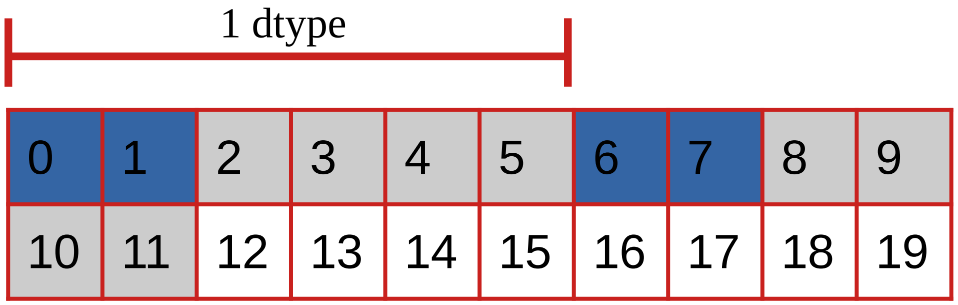
```
MPI.Datatype.Create_resized(lb, ub)
```

Upper bound of the data-types extent

```
dt = MPI.INT.Create_contiguous(2).Create_resized(0, 24)
```

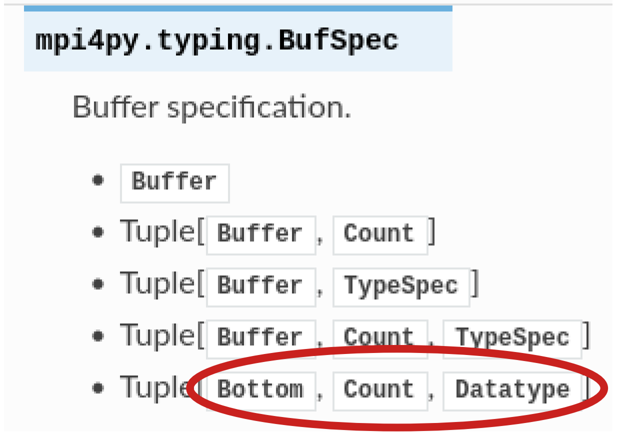
- use 2 integers (8 bytes)
- the data-type stops at 24 byte, next element starts at 25 bytes

An element is 4 bytes



Send/Recv custom data-types

They behave as the normal dtypes, but mpi4py does not auto calculate the number of dtypes it will send, so always use the extended bufferspec.



mpi4py.typing.BufSpec

Buffer specification.

- `Buffer`
- Tuple[`Buffer` , `Count`]
- Tuple[`Buffer` , `TypeSpec`]
- Tuple[`Buffer` , `Count` , `TypeSpec`]
- Tuple[`Bottom` , `Count` , `Datatype`]

```
from mpi4py import MPI
C = MPI.COMM_WORLD
xyz = np.empty([N, 3])
dt = MPI.DOUBLE.Create_contiguous(3)
```

- `C.Send(xyz, 1)`
- `C.Send((a, N, dt), 1)`

Recall `Get_count`, can *also* be used with custom data-types!

```
C.Recv(buf, …, status=status)
assert status.Get_count(dt) == N
assert status.Get_elements(dt) == N * 3
```

Count of dtype

Count of *basic* elements in dtype

Custom data-types

It is easy to create complicated data-types, customization is huge!

- Use them as much as you can, they will generally save you a temporary buffer when you do simple send/recv.

- They are local, so there is very little overhead

- Use `status.Get_count` to get count of data-types

- Use them for *boundary conditions* where a “halo” region needs to be communicated.

- They can also describe *sub-arrays* (see `MPI_Type_create_subarray`) (not covered! However, can be really useful when doing parallel IO in next weeks topic)

- They can reduce some complexity when you are sending constant sub-divisions i.e. coordinates (`N` * 3 => dtype of length 3, send `N`)