NHR4CES NHR for Computational Engineering Science

TECHNISCHE UNIVERSITÄT DARMSTADT

Introduction to parallisation in OpenFOAM

Mohammed Elwardi Fadeli[1,2], Holger Marschall[1] and Christian Hasse[2]

June 02, 2022

[1] Mathematical Modeling and Analysis (MMA)
[2] Simulation of Reactive Thermo Fluid Systems (STFS)
TU Darmstadt

Table of contents

Introduction

The power of parallel workers

Figure 1: Parallel work during F1 Pit stops;

Types of Parallelism

Data Parallelism

Work units execute the same operations on a (distributed) set of data: **domain decomposition**.

Task Parallelism

Work units execute on different control paths, possibly on different data sets: **multi-threading**.

Pipeline Parallelism

Work split between producer and consumer units that are directly connected. each unit executes a single phase of a given task and hands over control to the next one.

Types of Parallelism

Data Parallelism

Work units execute the same operations on a (distributed) set of data: **domain decomposition**.

Task Parallelism

Work units execute on different control paths, possibly on different data sets: **multi-threading**.

Pipeline Parallelism

Work split between producer and consumer units that are directly connected. each unit executes a single phase of a given task and hands over control to the next one.

Domain decomposition in OpenFOAM

simple

Simple geometric decomposition, in which the domain is split into pieces by direction

hierarchical

Same as simple, but the order in which the directional split is done can be specified

metis & scotch

Require no geometric input from the user and attempts to minimize the number of processor boundaries. Weighting for the decomposition between processors can be specified

manual

Allocation of each cell to a particular processor is specified directly.

Domain decomposition in OpenFOAM: Processor boundaries

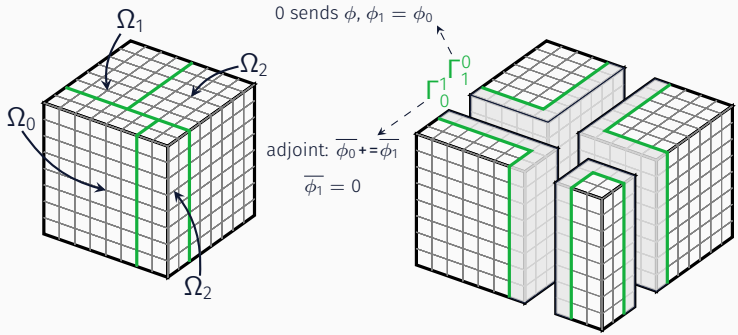


Figure 2: Classical halo approach for inter-processor communication

- Use of a layer of ghost cells to handle comms with neighboring processes → MPI calls not self-adjoint
- Artificial increase in number of computations per process (and does not scale well)

Domain decomposition in OpenFOAM: Processor boundaries

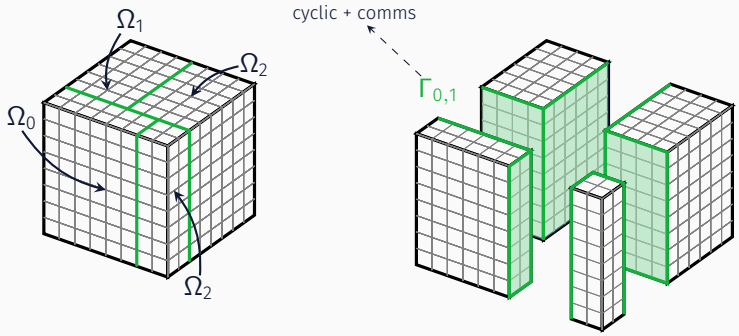


Figure 3: Zero-halo approach for inter-processor communication in OpenFOAM

- Communications accross process boundaries handled as a BC
- MPI calls are self-adjoint; all processes perform the same work at the boundaries

Modes of Parallelism

Distributed Memory

Message Passing Interface (**MPI**): Execute on multiple machines.

Shared Memory

Multi-threading capabilities of programming languages, OpenMP.

Data Streaming

CUDA and OpenCL. Applications are organized into streams (of same-type elements) and kernels (which act on elements of streams).

Is echo MPI-ready?

```
mpirun -n 3 echo Hello World!
```

Hello World!
Hello World!
Hello World!

What about a solver binary?

```
mpirun -n 3 icoFoam
```

This runs on "undecomposed" cases!

MPI with OpenFOAM

Is echo MPI-ready?

```
mpirun -n 3 echo Hello World!
```

Hello World!
Hello World!
Hello World!

What about a solver binary?

```
mpirun -n 3 icoFoam
```

This runs on "undecomposed" cases!

But the solver is linked to libmpi!

```
ldd $(which icoFoam)
```

... libmpi.so ...

MPI with OpenFOAM

Is echo MPI-ready?

```
mpirun -n 3 echo Hello World!
```

Hello World!
Hello World!
Hello World!

What about a solver binary?

```
mpirun -n 3 icoFoam
```

This runs on "undecomposed" cases!

But the solver is linked to libmpi!

```
ldd $(which icoFoam)
```

... libmpi.so ...

Alright we get it now

```
mpirun -n 3 icoFoam -parallel
```

Needs a decomposed case

MPI with OpenFOAM: Parallel mode

Anatomy of MPI programs

```
#include <mpi.h>
void main (int argc, char *argv[])
{
  int np, rank, err;
  err = MPI_Init(&argc, &argv) ;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  // Do parallel communications
  err = MPI_Finalize() ;
}
```

MPI with OpenFOAM: Parallel mode

Anatomy of MPI programs

```
#include <mpi.h>
void main (int argc, char *argv[])
{
  int np, rank, err;
  err = MPI_Init(&argc, &argv) ;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  // Do parallel communications
  err = MPI_Finalize() ;
}
```

How solver programs look

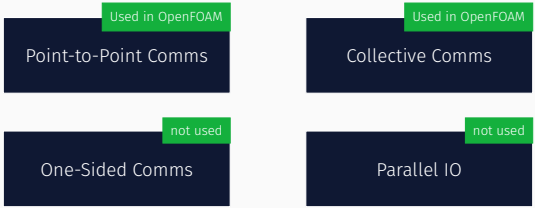
```
#include "fvCFD.H"
void main (int argc, char *argv[])
{
  #include "setRootCase.H"
  // Defines an argList object,
  // which has a ParRunControl member
  // If -parallel is passed in:
  // MPI_Init called in its ctor
  // MPI_Finalize called in its dtor

  // Time is constructed with
  // <case>/processor<procID> paths
}
```

You don't have to know MPI API to parallelise OpenFOAM code! But you need the concepts.

Objectives

1. Have a basic understanding of Parallel programming with MPI in OpenFOAM Code.
2. Be able to send basic custom classes around using MPI.
3. Be aware of some of the common issues around MPI comms.
4. Aquire enough knowledge to learn more on your own
    - Directly from OpenFOAM's code
    - MPI in general

Communication types in MPI



We'll be focusing on the communications OpenFOAM wraps!

Point-to-Point communication

Communicators and ranks

There may be many processes talking!

MPI Communicators

Objects defining which processes can communicate; Processes are refered to by their **ranks**

- `MPI_COMM_FOAM` in the Foundation version and Foam Extend 5
- `MPI_COMM_WORLD` (All processes) elsewhere
- Size: `Pstream::nProcs()`

MPI rank

Process Identifier (an integer).

- `Pstream::myProcNo()` returns the active process's ID.

P2P comms

- MPI defines its own **Data Types**
- OpenFOAM gets around it using **parallel streams**
  - OpenFOAM hands over a stream-representation of your data to MPI calls
  - MPI passes the information in those streams around

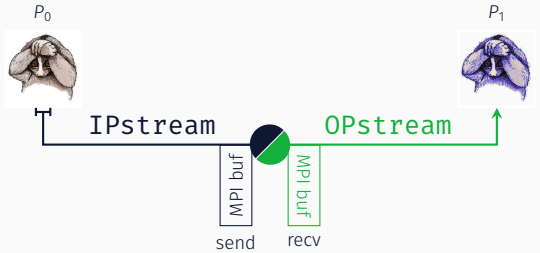


Figure 4: Communication between two processes in OpenFOAM

P2P comms: A first example

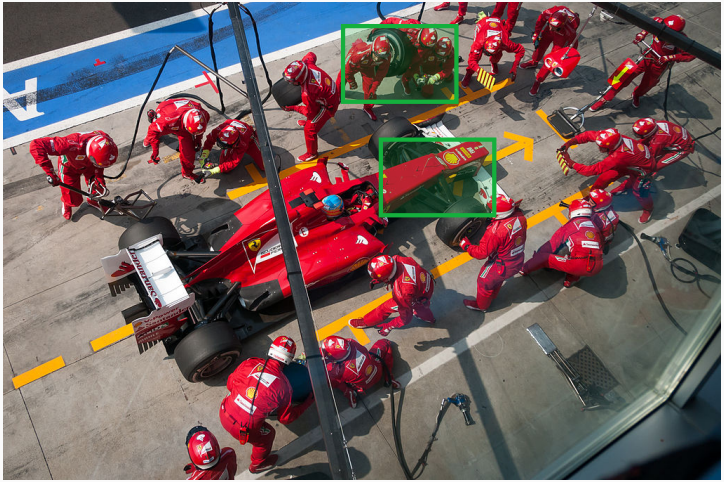


Figure 1: Parallel work during F1 Pit stops;

P2P comms: A first example

- `Pstream` class provides the interface needed for communication
- Each ”send” must be matched with a ”recieve”

Slaves talk to master

```
if (Pstream::master())
{
    // Receive lst on master
    for
    (
        int slave=Pstream::firstSlave();
        slave<=Pstream::lastSlave();
        slave++
    )
    {
        labelList lst;
        IPstream fromSlave (Pstream::commsTypes::blocking, slave);
        fromSlave >> lst; // Then do something with lst
    }
} else {
    // Send lst to master
    OPstream toMaster (Pstream::commsTypes::blocking, Pstream::masterNo());
    toMaster << localLst;
}
```

P2P Blocking comms

`Pstream::commsTypes::blocking` (or just `Pstream::blocking` in Foam Extend) defines properties for the MPI call which is executed by the constructed stream.

- Does a ”local blocking send”, i.e. acts on a local buffer
- No matching receive available yet? Block until the message is copied into the buffer
- Returns when the send buffer is safe to be reused.
- Blocking recieve only returns when the receive buffer has the expected data.
- Use this if you want to be on the safe side.
- But, It may result in deadlocks

P2P Blocking comms

`Pstream::commsTypes::scheduled` (or just `Pstream::scheduled` in Foam Extend) lets MPI pick the best course of action (in terms of performance and memory). This may also depend on the MPI implementation.

- Does a ”standard send”, Either:
  1. The message is directly put in the recieve buffer.
  2. Data is buffered (similar to ‘blocking‘).
  3. Block until a receive shows up.
- Has higher chances of causing deadllocks

A Deadlock happens when a process is waiting for a message that never reaches it.

P2P Blocking comms: Deadlocks

- Either a matching send or a recieve is missing (Definitely a deadlock).
- A send-recieve cycle (Incorrect usage or order of send/recieve calls).

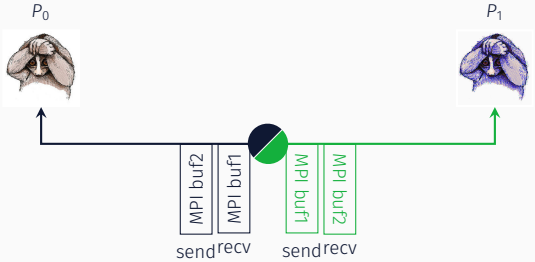


Figure 5: Deadlock possibility due to a 2-processes send-recieve cycle (Kind of depends on MPI implementation used!).

P2P Non-Blocking comms

`Pstream::commsTypes::nonBlocking` (or just `Pstream::nonBlocking` in Foam Extend) does not wait until buffers are safe to re-use.

- Returns immediately.
- The program must wait for the operation to complete (`Pstream::waitRequests`).
- It's a form of piepline parallelism; i.e. Overlaps computation and communication.

- Avoids Deadlocks
- Minimizes idle time for MPI processes
- Helps skip unnecessary synchronisation

P2P Non-Blocking comms: An example

Communicate with a neighboring processor

```
// Code for the Foundation version and ESI
PstreamBuffers pBufs (Pstream::commsTypes::nonBlocking);
// Send
forAll(procPatches, patchi)
{
  UOPstream toNeighb(procPatches[patchi].neighbProcNo(), pBufs);
  toNeighb << patchInfo;
}
pBufs.finishedSends();  // <- Calls Pstream::waitRequests
// Receive
forAll(procPatches, patchi)
{
  UIPstream fromNb(procPatches[patchi].neighbProcNo(), pBufs);
  Map<T> nbrPatchInfo(fromNb);
}
```

Overlapping communication and computation

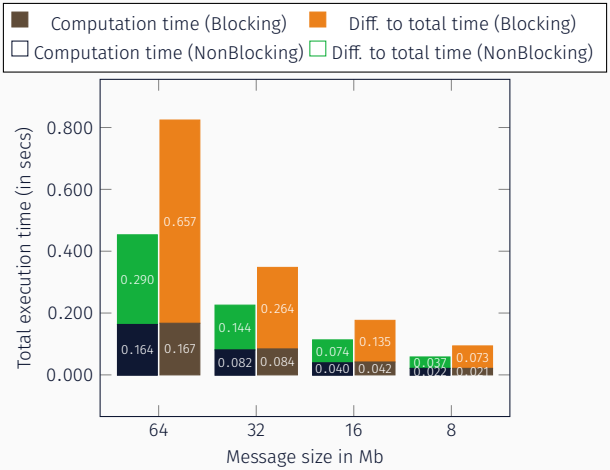
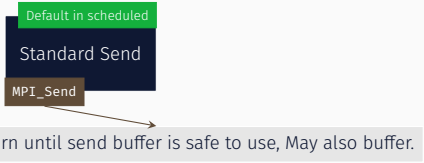
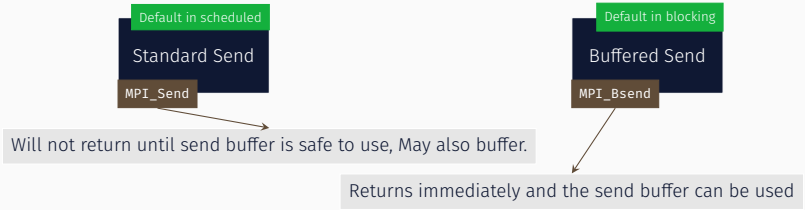


Figure 6: Effect of message size on overlapping communication and computation (4 processors, OpenMPI 4, OpenFOAM 8); Benchmark inspired from [2]

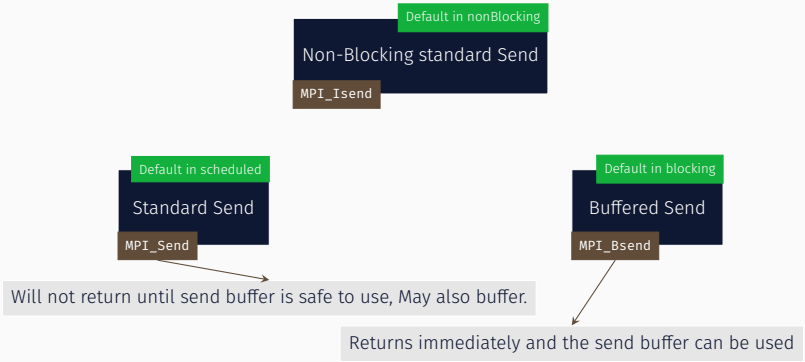
MPI send modes used in OpenFOAM code



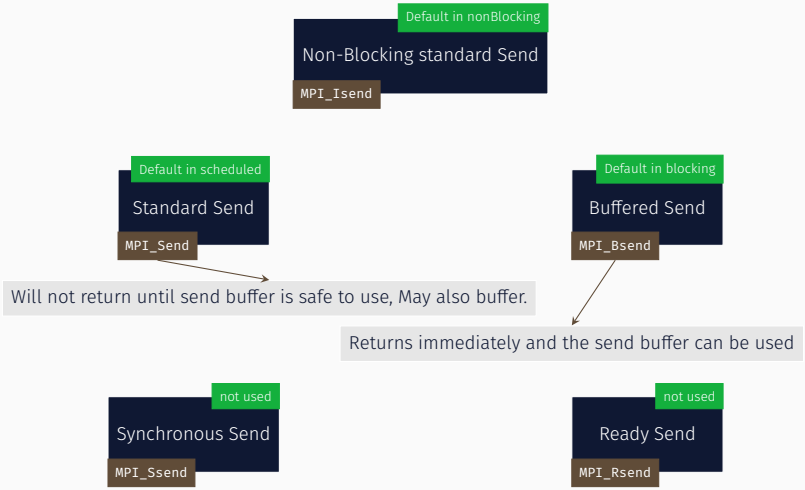
MPI send modes used in OpenFOAM code



MPI send modes used in OpenFOAM code



MPI send modes used in OpenFOAM code



Collective communication

Collective comms

When Two or more processes talk to each other.

- **All processes** call the same function with the same set of arguments.
- Although MPI-2 has non-blocking collective communications, OpenFOAM uses only the blocking variants.
- NOT a simple wrapper around P2P comms.
- OpenFOAM puts their interface in **static public methods** of `Pstream` class.
  - Major differences in the API accross forks: (ESI and Foundation version) vs Foam Extend.

Collective comms

- Most collective algorithms are $log(nProcs)$
- Gather (all-to-one), Scatter (one-to-all), All-to-All variants of all-to-one ones.
- OpenFOAM does not use all-to-one ”reduce”. What OpenFOAM calls a ”reduce” is Gather+Scatter.
- MPI has also a ”Broadcast” and ”Barrier” but these are not used in OpenFOAM.

Collective comms: Gather (All-to-one)

Check how something is distributed over processors

```
bool v = false;
if (Pstream::master()){ v = something(); } // <- must do on master
Pstream::gather(v, orOp<bool>());  // <- root process gathers
```

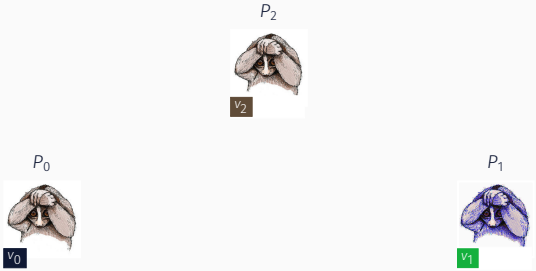


Figure 7: An example OpenFOAM gather operation (More like a MPI-reduce)

Collective comms: Gather (All-to-one)

Check how something is distributed over processors

```
bool v = false;
if (Pstream::master()){ v = something(); } // <- must do on master
Pstream::gather(v, orOp<bool>());  // <- root process gathers
```

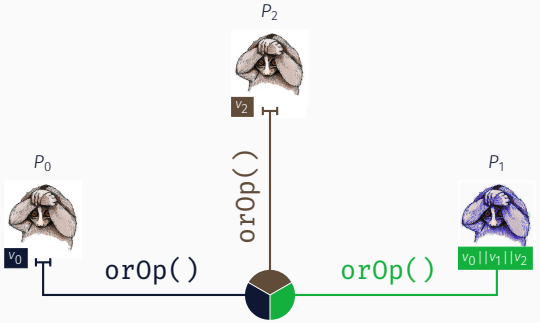


Figure 7: An example OpenFOAM gather operation (More like a MPI-reduce)

Collective comms: Gather (All-to-one)

Check how something is distributed over processors (List-like)

```
List<bool> localLst(Pstream::nProcs(), false);
localLst[Pstream::myProcNo()] = something();
Pstream::gatherList(localLst); // <- root process gathers
```

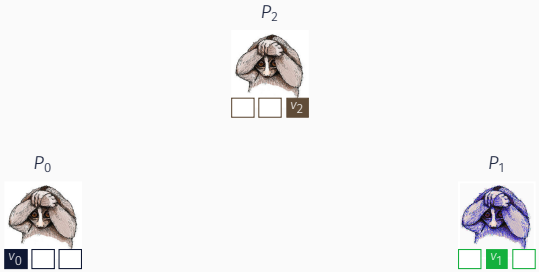


Figure 8: Example OpenFOAM gather operation on list items

Collective comms: Gather (All-to-one)

Check how something is distributed over processors (List-like)

```
List<bool> localLst(Pstream::nProcs(), false);
localLst[Pstream::myProcNo()] = something();
Pstream::gatherList(localLst); // <- root process gathers
```

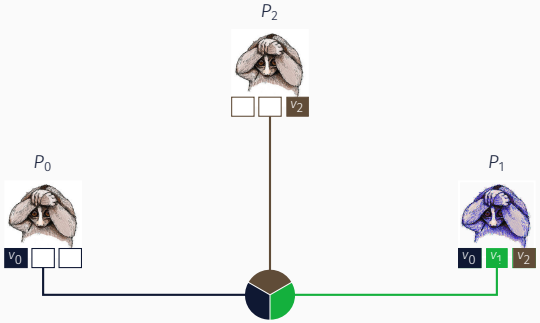


Figure 8: Example OpenFOAM gather operation on list items

Collective comms: Scatter (One-to-all)

Make processes know about something

```
bool v = false;
if (Pstream::master()){ v = something(); } // <- must do on master
Pstream::scatter(v);  // <- root process scatters
```

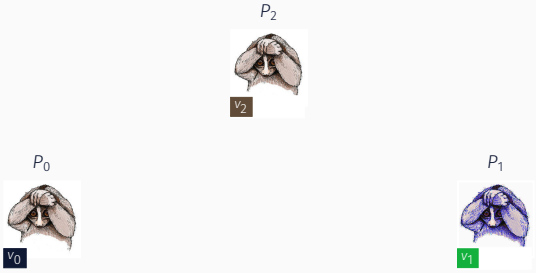


Figure 9: An example OpenFOAM scatter operation (More like a MPI-Bcast)

Collective comms: Scatter (One-to-all)

Make processes know about something

```
bool v = false;
if (Pstream::master()){ v = something(); } // <- must do on master
Pstream::scatter(v);  // <- root process scatters
```

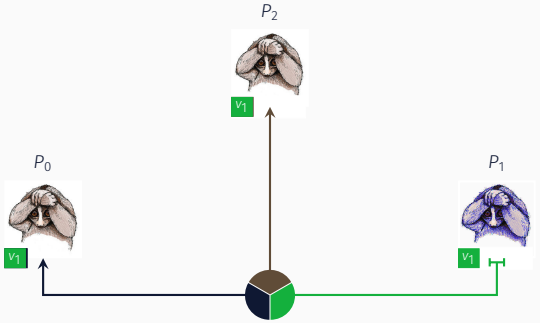


Figure 9: An example OpenFOAM scatter operation (More like a MPI-Bcast)

Collective comms: Scatter (One-to-all)

Make processes know about something (List-like)

```
List<bool> localLst(Pstream::nProcs(), false);
if (Pstream::master()){ forAll(localLst, ei) { localLst[ei] = something(); } }
Pstream::scatterList(localLst); // <- root process scatters
```

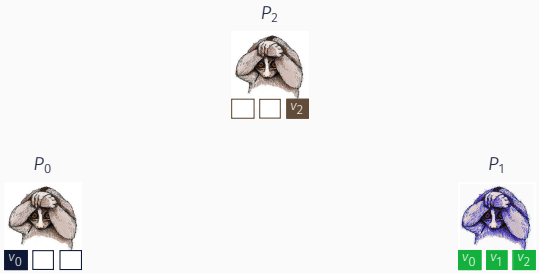


Figure 10: Example OpenFOAM scatter operation on list items

Collective comms: Scatter (One-to-all)

Make processes know about something (List-like)

```
List<bool> localLst(Pstream::nProcs(), false);
if (Pstream::master()){ forAll(localLst, ei) { localLst[ei] = something(); } }
Pstream::scatterList(localLst); // <- root process scatters
```

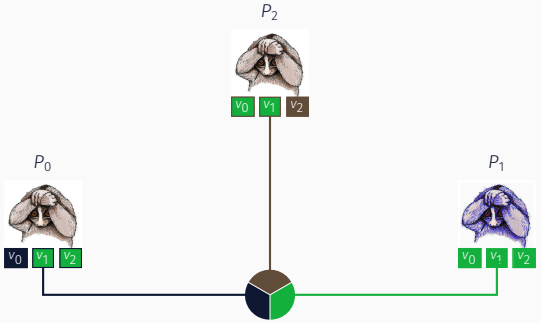


Figure 10: Example OpenFOAM scatter operation on list items

Collective comms: Reduce (All-to-All)

Do something with a var on all processors (eg. sum them up)

```
// Second arg: a binary operation function (functors); see ops.H
Foam::reduce(localVar, sumOp<decltype(localVar)>());
localVar = Foam::returnReduce(nonVoidCall(), sumOp<decltype(localVar)>());
```

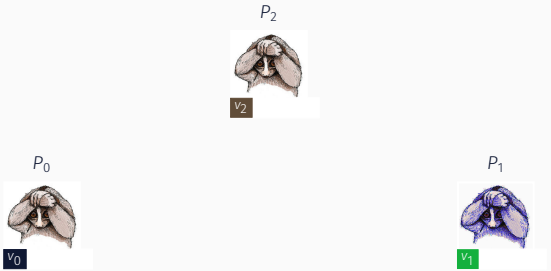


Figure 11: An example OpenFOAM reduce operation (MPI-Allreduce)

Collective comms: Reduce (All-to-All)

Do something with a var on all processors (eg. sum them up)

```
// Second arg: a binary operation function (functors); see ops.H
Foam::reduce(localVar, sumOp<decltype(localVar)>());
localVar = Foam::returnReduce(nonVoidCall(), sumOp<decltype(localVar)>());
```

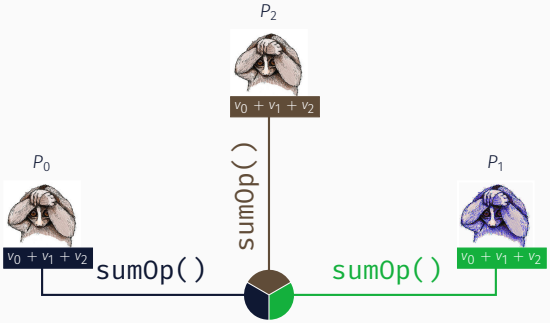


Figure 11: An example OpenFOAM reduce operation (MPI-Allreduce)

Oh, there is a reduce here!

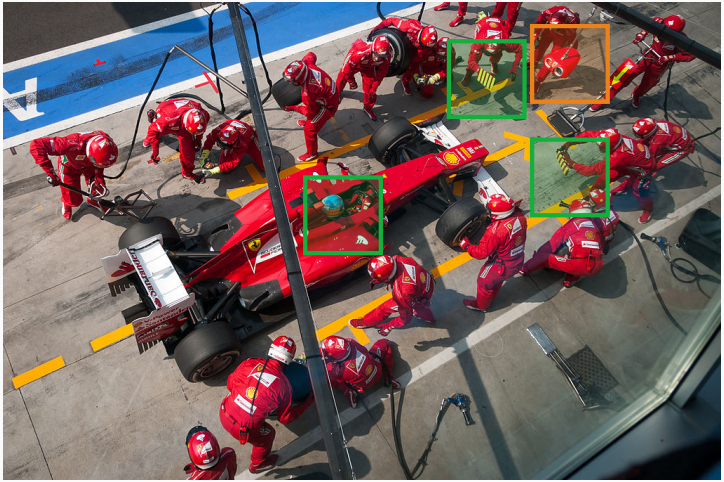


Figure 1: Parallel work during F1 Pit stops;

Collective comms: Issues

You can still fall for endless loops if you're not careful!

Infinite loops due to early returns

```
void refineMesh(fvMesh& mesh, const label& globalNCells)
{
  label currentNCells = 0;
  do
  {
    // Perform calculations on all processors
    currentNCells += addCells(mesh);
    // On some condition, a processor should not continue, and
    // returns control to the caller
    if (Pstream::myProcNo() == 1) return; // <-- oops, can't do this
    // !!! who's going to reduce this!
    reduce(currentNCells, sumOp<label>());
  } while (currentNCells < globalNCells);
  return;
}
```

What if one of them just walks away mid-op?

Figure 1: Parallel work during F1 Pit stops;

How do I send my own data?

A graph to hold neighboring processors

Say we have something like this:

A directed graph of nodes

```
struct Edge {
    label destination = -1;
    scalar weight = 0.0;
};
using Graph = List<List<Edge>>;
// Try to push an edge from master to all procs
Edge ej;
if (Pstream::master())
{
    ej.destination = 16;
    ej.weight = 5.2;
}
```

Note: Edge does not have the requirements to be in a List yet

A graph to hold neighboring processors

First, does this compile?

Push an edge from master to All

```
Pstream::scatter(ej);
```

error: No match for `operator<<(OPstream&,Edge&)`
error: No match for `operator>>(IPstream&,Edge&)`

So, Edges can't be communicated as MPI messages, fix it:

Stream Operators for Edge class

```
Ostream&  operator<<(Ostream& os, Edge& e) {
    os << e.destination << " " << e.weight;
    return os;
}
Istream&  operator>>(Istream& is, Edge& e) {
    is >> e.destination;
    is >> e.weight;
    return is;
}
```

A graph to hold neighboring processors

Try again:

Gathering info about graph nodes

```
Pstream::gatherList(g);
Pstream::scatterList(g);

// Check graph edges on all processes
Pout << g << endl;
```

Compiles, and works as expected

A Better way: Make Edge a child of one of the OpenFOAM classes

Better ways to define an Edge

```
struct Edge : public Tuple2<label, scalar> {};
// That's it, Edge is now fully MPI-ready
```

Application examples & advanced topics

Solving PDEs over decomposed domains (P2P comms)

General Transport Equation for a physical transport property

$$\partial_t \phi + \nabla \cdot (\phi \mathbf{u}) - \nabla \cdot (\Gamma \nabla \phi) = S_\phi(\phi)$$

Discretized form (Finite Volume notation)

$$[\![\partial_t[\phi]]\!] + [\![\nabla \cdot \left(F[\phi]_{f(F,S,\gamma)}\right)]\!] - [\![\nabla \cdot \left(\Gamma_f \nabla[\phi]\right)]\!] = [\![S_l[\phi]]\!].$$

- Receive neighbour values from neighbouring processor.
- Send face cell values from local domain to neighouring processor
- Interpolate to processor patch faces

Adaptive Mesh Refinement on polyhedral meshes

1. Refine each processor's part of the mesh, but we need to keep the global cell count under a certain value:

Reduce nAddCells or nTotalAddCells?

```
label nAddCells = 0;
label nIters = 0;
label nTotalAddCells = 0;
do
{
    nAddCells = 0;
    if (edgeBasedConsistency_)
    {
        nAddCells += edgeConsistentRefinement(refineCell);
    }
    nAddCells += faceConsistentRefinement(refineCell);
    reduce(nAddCells, sumOp<label>());
    ++nIters;
    nTotalAddCells += nAddCells;
} while (nAddCells > 0);
```

Adaptive Mesh Refinement on polyhedral meshes

2. To decide on whether to refine cells at processor boundaries, we need cell levels from the other side:

Reduce nAddCells or nTotalAddCells?

```
// Code extracted from Foam Extend 4.1
labelList ownLevel(nFaces - nInternalFaces);
forAll (ownLevel, i)
{
    const label& own = owner[i + nInternalFaces];
    ownLevel[i] = updateOwner();
}

// Swap boundary face lists (coupled boundary update)
syncTools::swapBoundaryFaceList(mesh_, ownLevel, false);

// Note: now the ownLevel list actually contains the neighbouring level
// (from the other side), use alias (reference) for clarity from now on
const labelList& neiLevel = ownLevel;
```

Advanced topics

The need for Load Balancing in AMR settings

- AMR operations tend to unbalance cell count distribution accross processors
- Using Blocking comms means more idle process time
  - Non-Blocking are not a solution.
  - Spending some time on rebalancing the mesh is.
- Naturally, load balancing itself involves parallel communication!

Licensing

Questions?

Compile and link against MPI implementations

Compiler wrappers are your best friends!

Grab correct compiler/linker flags

```
mpic++ --showme:compiler
mpic++ --showme:linker
```

```
-I/usr/lib/x86_64-linux-gnu/openmpi/include/openmpi
...
-pthread -L/usr/lib/x86_64-linux-gnu/openmpi/lib
...
```

OpenFOAM environment autmatically figures things out for you:

Typical Make/options file for the ESI fork

```
include $(GENERAL_RULES)/mpi-rules
EXE_INC = $(PFLAGS) $(PINC) ...
LIB_LIBS = $(PLIBS) ...
```

Miscellaneous

- MPI standards: Blocking send can be used with a Non-blocking receive, and vice-versa
- But OpenFOAM wrapping makes it work ”non-trivial”
- You can still use MPI API directly, eg. if you need one-sided communication.
- Overlapping computation and communication for non-blocking calls is implemented on the MPI side, so, put your computations after the recieve call.

Sources and further reading i

[1] C. Augustine. *Introduction to Parallel Programming with MPI and OpenMP*. Source of the great 'pit stops' analogy. Oct. 2018. URL: https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf.

[2] Fabio Baruffa. *Improve MPI Performance by Hiding Latency*. July 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/overlap-computation-communication-hpc-applications.html.

Supported by NH4CES

Sources and further reading ii

[3] Pavanakumar Mohanamuraly, Jan Christian Huckelheim, and Jens-Dominik Mueller. “Hybrid Parallelisation of an Algorithmically Differentiated Adjoint Solver”. In: *Proceedings of the VII European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2016)*. Institute of Structural Analysis and Antiseismic Research School of Civil Engineering National Technical University of Athens (NTUA) Greece, 2016. DOI: 10.7712/100016.1884.10290. URL: https://doi.org/10.7712/100016.1884.10290.

[4] B. Steinbusch. *Introduction to Parallel Programming with MPI and OpenMP*. Mar. 2021. URL: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/mpi/mpi-openmp-handouts.pdf?__blob=publicationFile.

[5] EuroCC National Competence Center Sweden. *Intermediate MPI*. May 2022. URL: https://enccs.github.io/intermediate-mpi/.