Parallelization in OpenFOAM for HPC Deployment

Hands-on activities

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

April, 2023

[1] Mathematical Modeling and Analysis (MMA)
[2] Simulation of Reactive Thermo Fluid Systems (STFS)
Energy Conversion Group, NHR4CES - TU Darmstadt

Introductory activities

Activity 01: Blocking P2P comms - A first look

- parallelClass class has a lst_ member, initiated differently on each process.
- Your task is to append lst_ from all **slave** processes (1, 2 and 3) into a single list object on master.
  - → Using blocking P2P communication between master and a slave each time.
  - → Append the lists in the order of process IDs.
  - → Modify exercises/parallelClass.C file.

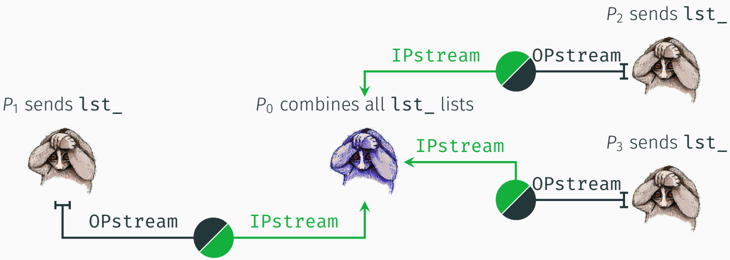

Figure 1: Blocking P2P with master

Activity 02: Collective comms - A first look

- `parallelClass::isPrime(int)` is a primitive method to check if its argument is a prime number.
  - → But it's serial code! Running it in parallel will **duplicate work** on all processes
  - → Parallelizing it should yield performance gains

1. Domain Decomposition
   - → This is done already if your data is mesh-related (mesh itself, fields, ... etc)
   - → We need to "decompose" the search range `[3, sqrt(n)+1]` into `nProcs` ranges.
   - → `parallelClass::next(int n, int& i)` increments `i` to the next number in process-controlled range

Figure 2: Provided trivial decomposition of the range $[3, \sqrt{n} + 1]$

Activity 02: Collective comms - A first look

2. Blocking P2P comms:
   - → Transform `parallelClass::isPrime(int)` so that it acts on the corresponding range on each of the 4 processes.
   - → All processes **must** decide if it's a prime number, based on results from all other process.
   - → Think: collective comms. Although P2P comms also would work!
3. An alternate decomposition:
   - → Each process is responsible for possible divisors that are `nProcs` apart.
   - → Implement this behavior in `parallelClass::next(int n, int& i)`

Figure 3: A new example decomposition of the range $[3, \sqrt{n} + 1]$

Activity 02: Collective comms - A first look

Is there room for improvement?

4. Decomposition effect
   - → Different decompositions result in different loads on processes
   - → In blocking comms, balanced decompositions are the most efficient
   - → But keep in mind; if it takes too much time to decompose "dynamically" - it might backfire!
5. Optimizing the code
   - → A process can `break` out of loop and return if it finds a viable divisor.
   - → But can't stop calculations on all other processes.
   - → Hence, apparently, not worth the trouble.
   - → **But,** if all other processes also return early, we can gain some CPU time.
   - → Maybe test only on **prime numbers** less than $\sqrt{n} + 1$?
   - → The whole parallelization idea was premature optimization, huh!

Activity 03: Blocking P2P comms are not good for your health

- Code in `parallelClass::run()` tries to perform 2 blocking P2P communication operations between two processes.
  - → When they hang; processes are killed by `timeout`.
- Your task is to modify the `run` member method so the code no longer hangs.
  - → Pay attention to the order of the send/receive ops!
  - → The Actual sending happens on `*Pstream` **objects' destruction**
  - → Do you think your solution would withstand MPI implementation swapping? (OpenMPI, Intel MPI, MPICH ... etc)

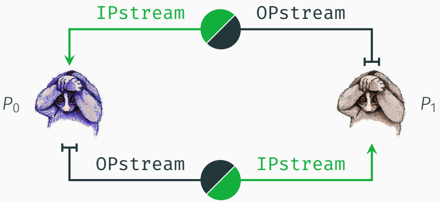

Figure 4: 2 Blocking P2P communication operations at the same time

Activity 04: Non-Blocking P2P comms for swapping ops

- Each process communicates a list object to its **neighbour**
  - → Neighbouring relationships are based on mesh decomposition
  - → `parallelClass` class has `nProcs` lists as a member variable `lists_` (Each list is to be transferred to the process's neighbour)
- Your task is to perform an exchange of the lists so that, at the end of all transfers; each process holds the lists from its neighbors (instead of its own ones).
  - → This should be done in `parallelClass::swapLists()` method
  - → Again in `execises/parallelClass.C` file

Activity 04: Non-Blocking P2P comms for swapping ops

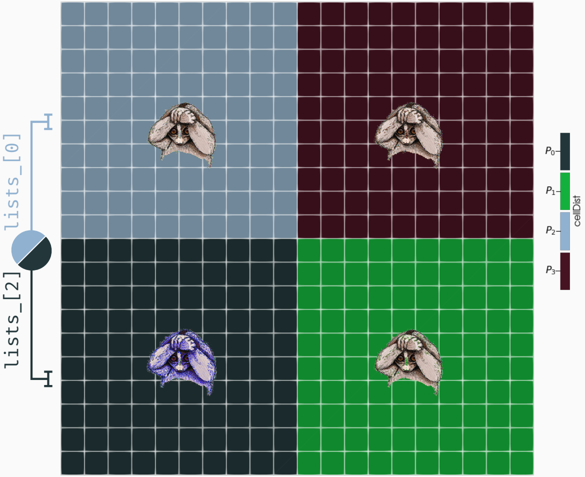

Figure 5: Default (hierarchical) decomposition of the block-mesh in the cavity case

Activity 04: Non-Blocking P2P comms for swapping ops

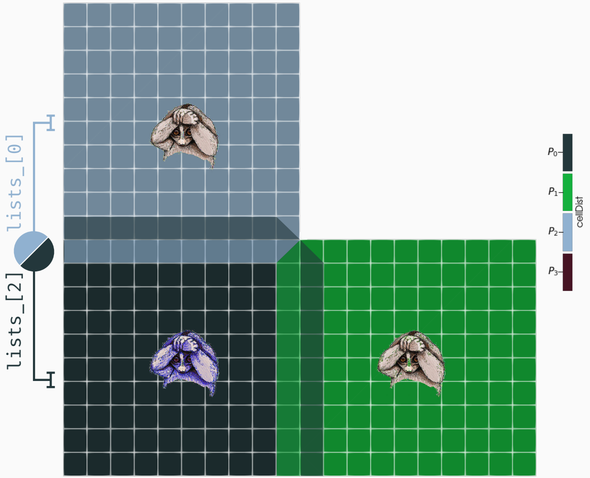

Figure 5: Default (hierarchical) decomposition of the block-mesh in the cavity case

Activity 04: Non-Blocking P2P comms for swapping ops

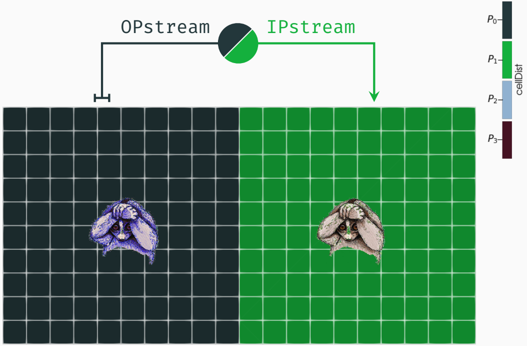

Figure 5: Default (hierarchical) decomposition of the block-mesh in the cavity case

Activity 05: Collective comms - Reference cell

- It's popular to assign a **reference cell** if all BCs are gradient-based for a field:

Common pressure reference point in `system/fvSolution`

```
PISO
{
    pRefPoint     (0.0975 0.0025 0);          // At far right-bottom cell of the cavity mesh
    pRefValue     0;
}
```

→ Note that `pRefCell` is enough for most cases. Information propagated automatically through processor patches!

- `mesh.findCell(position);` returns:
  → `-1` if the position is outside the local mesh (on calling process).
  → `cellID` of the cell containing the position otherwise.

1. Your first task is to check if the provided position is inside the **global mesh**
   → Using collective comms; Think: `Foam::reduce`
   → Change `parallelClass::checkPosition(const vector&)` so it returns `true` if given position is inside the mesh.

Activity 05: Collective comms - Reference cell

2. It's useful for everyone to know **which process has the corresponding reference cell**
   - → Change `parallelClass::whoHasReferenceCell(const vector&)` so it returns `-1` if position is outside the mesh; and returns the rank of the process which is responsible for the reference cell otherwise.
   - → Again, using collective comms.

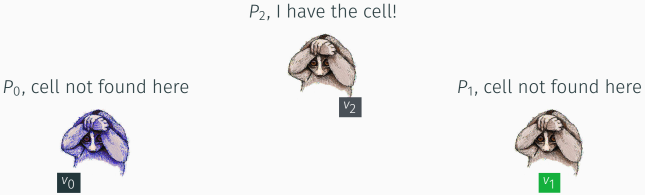

Figure 6: Suggested setup for reference cell communication

2. It's useful for everyone to know **which process has the corresponding reference cell**
   - → Change `parallelClass::whoHasReferenceCell(const vector&)` so it returns `-1` if position is outside the mesh; and returns the rank of the process which is responsible for the reference cell otherwise.
   - → Again, using collective comms.

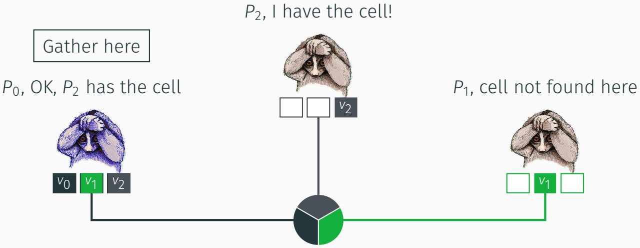

Figure 6: Suggested setup for reference cell communication

Activity 05: Collective comms - Reference cell

2. It's useful for everyone to know **which process has the corresponding reference cell**
   - → Change `parallelClass::whoHasReferenceCell(const vector&)` so it returns `-1` if position is outside the mesh; and returns the rank of the process which is responsible for the reference cell otherwise.
   - → Again, using collective comms.

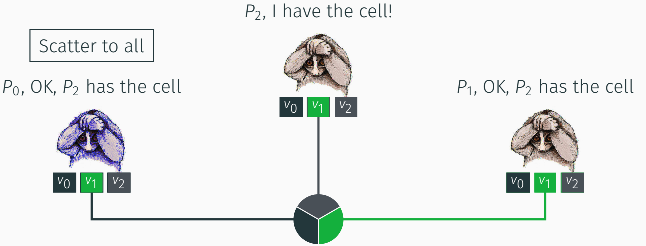

Figure 6: Suggested setup for reference cell communication

Activity 06: Parallel comms for custom data types

- So, I wrote a class; now I want its objects to travel through processors.
  - → `parallelClass.H` file declares an `Edge` class, to represent ”neighbouring relationship” between two processors.
  - → The goal is to build a graph of such edges.

1. Your first task is to make this `Edge` class compatible with random-access lists. I.e an Edge object can be put in a list.
   - → Try `./Alltest` and take a look at compilation log.
   - → Modify `Edge` class declaration/definition so that errors related to its compatibility with `List` template disappear.

Activity 06: Parallel comms for custom data types

2. Implement `operator<<` and `operator>>` so that an `Edge` object can be passed-to/read-from Output/Input streams.
   - → Things should at least compile.
   - → Compatibility of `Edge` with `List` is tested by **the compiler**.
   - → There is also a check for correct graph communications when the graph is gathered then scattered.

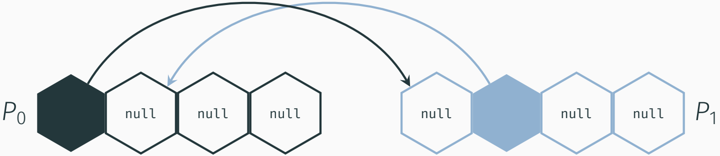

Figure 7: Typical way of sending custom objects through `*Pstreams` using Collective Comms

Activity 07: Special parallel comms for custom data types

- All good, but my class can't have a null constructor!
  - → Best solution is to use **Linked Lists**.
  - → If constructor arguments are needed on the other end; you need a factory sub-class.
- `Edge` class was modified to fit into linked lists, and can be constructed from a mesh instance and an input stream.
  - → Your task is to make it `*Pstreams`-ready

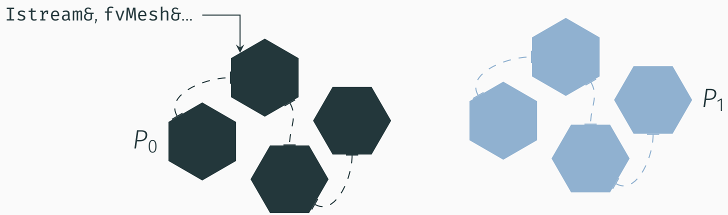

Figure 8: Typical way of sending custom objects through `*Pstreams` using P2P Comms

Activity 07: Special parallel comms for custom data types

- All good, but my class can't have a null constructor!
  - → Best solution is to use **Linked Lists**.
  - → If constructor arguments are needed on the other end; you need a factory sub-class.
- `Edge` class was modified to fit into linked lists, and can be constructed from a mesh instance and an input stream.
  - → Your task is to make it `*Pstreams`-ready

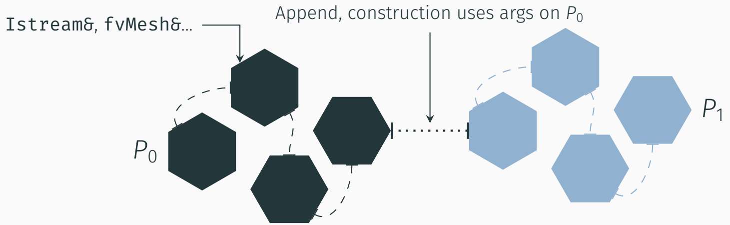

Figure 8: Typical way of sending custom objects through `*Pstreams` using P2P Comms

Final projects

Project 01: Parallelizing a coded source term

→ This project does not work with Foam-Extend as it doesn't support `fvOptions`.

1. You inherit a repository with a cavity case which works "as expected" in serial
2. Run `./Allrun` from inside the case's directory to compare results from serial and parallel runs.
    - This requires ParaView to be installed.
    - Use the server version for headless machines if you're on a container.
    - The script writes a `log.pvpython` file containing max/min absolute error in velocity values at t = 0.5s between serial and parallel runs of a cavity case.
3. The provided cavity case has a coded `fvOptions` which adds a source term to the velocity equation.

- Run `./Allrun` with the source active
- Disable the `fvOptions` source and `./Allclean && ./Allrun` again.
    - By either setting `codedSource.active` to false, in `system/fvOptions`.
    - Or by moving fvOptions file elsewhere.

Project 01: Parallelizing a coded source term

How does the custom source work?

- We define a box inside our cavity-case mesh to add an explicit vector source to the `UEqn` there.
- The x-component of the source at each cell $S_i = S_v * \frac{1}{nNeighbours+1}(\sum_{j=1}^{nNeighbours} k_j + k_i)$ depends on:
    1. Some "total" source value S_v provided by the user.
    2. An average value of a coefficient field k over the cell and its immediate neighbours.
- the target volume is defined through spacial dimensions and the corresponding cells are found through a Cell Set: `box (0.02 0.02 -1) (0.06 0.06 1)`

Project 01: Parallelizing a coded source term

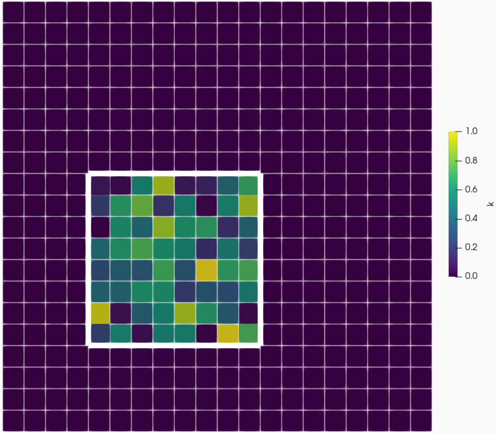

Figure 9: A sample run of the randomized coefficient field k. (The white box is where the source is applied)

Project 01: Parallelizing a coded source term

Suggested Steps

1. Identify the problem lines in fvOptions' code.

   Hint: Look for lines which use local mesh information!

2. Implement a fix for the identified issues.

   Hint: Can we get information about neighbouring cells that are on the other processor?

3. How optimized/sophisticated do you think your solution is? Share it with your peers and take a look at theirs!

Project 01: Parallelizing a coded source term

To help you identify the issue, take a look at this parameter variation study (varying source box dimensions, S_v parameter and number of MPI processes involved):

Table 1: Sample trial data for the parameter variation study on the cavity case

Trial	S_v	x_{min}	x_{max}	y_{min}	y_{max}	nProcs	**MaxError**
0	9.569e-4	0.01	0.07	0.04	0.06	5	0.75085
1	8.102e-4	0.01	0.06	0.03	0.06	8	0.36281
2	3.061e-5	0.03	0.06	0.04	0.06	7	0.01018
3	8.598e-4	0.03	0.06	0.01	0.05	4	0.65839

Project 01: Parallelizing a coded source term

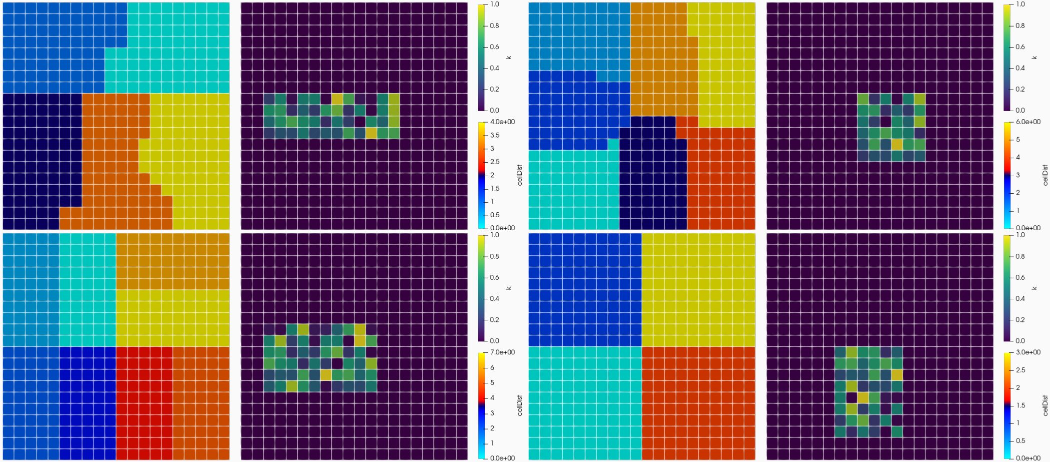

Figure 10: k coefficient field for a sample from trial data

Project 01: Parallelizing a coded source term

Want a quick way to test your fix?

Conduct your own parameter variation studies while you fix the case!

```
# Clone the helper repository
git clone https://github.com/FoamScience/OpenFOAM-Multi-Objective-Optimization multiOptFoam
cd multiOptFoam
# Install dependencies
pip3 install -r requirements.txt
# Copy your case
cp -r /path/to/cavity/case template
# Grab the config file (provided with the case)
cp /path/to/config.yaml .
# Run parameter variation
./paramVariation.py
# Now change template/system/fvOptions, clean and rerun the variation study
rm -rf Example* && ./paramVariation
# The goal is to get the maxError column for all trials as close to zero as possible (~ 1e-6)
```