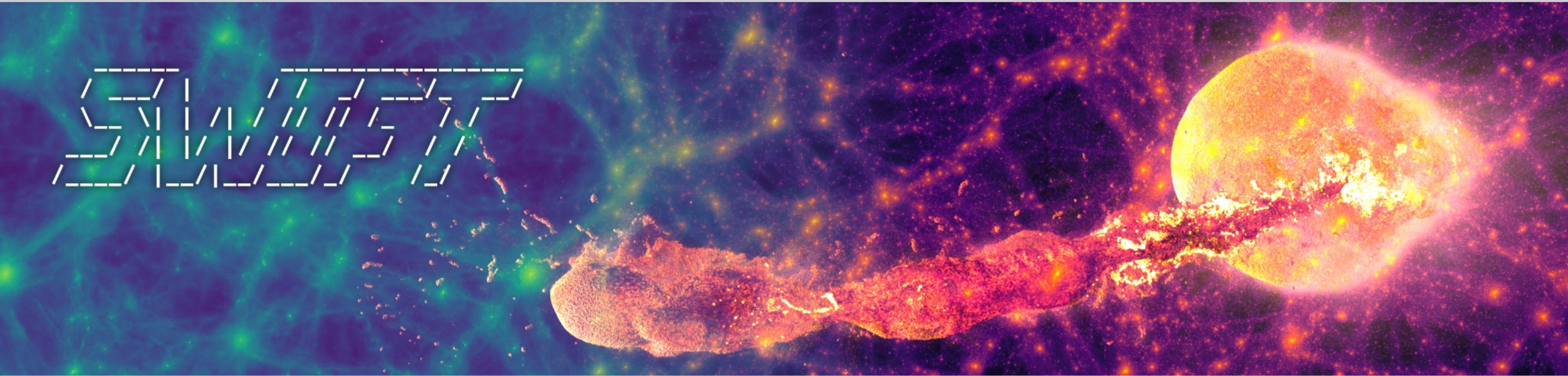
Task based parallelism and asynchronous MPI in SWIFT



Peter W. Draper, Alastair Basden and the SWIFT developers
https://www.swiftsim.com/

PAX meeting May 2023

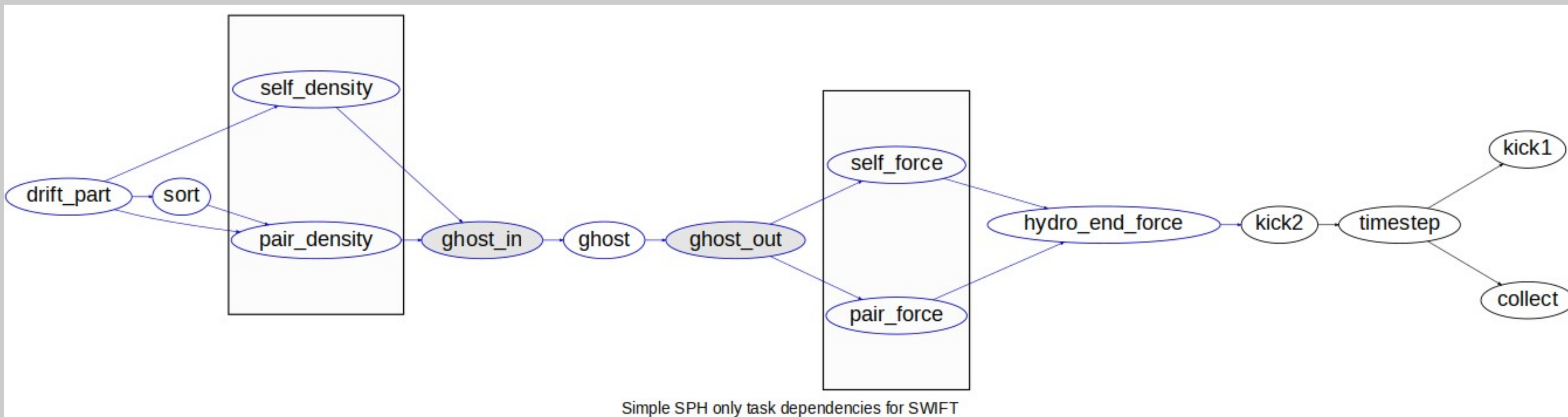
SWIFT

SPH With Inter-dependent Fine-grained Tasking

SWIFT is an open source hydrodynamics and gravity code for astrophysics and cosmology.
https://www.swiftsim.com & https://gitlab.cosma.dur.ac.uk/swift/swiftsim / https://github.com/SWIFTSIM
It is currently used to run simulations of astrophysics objects, such as planets, galaxies and very large cosmologies.

Task based parallelism

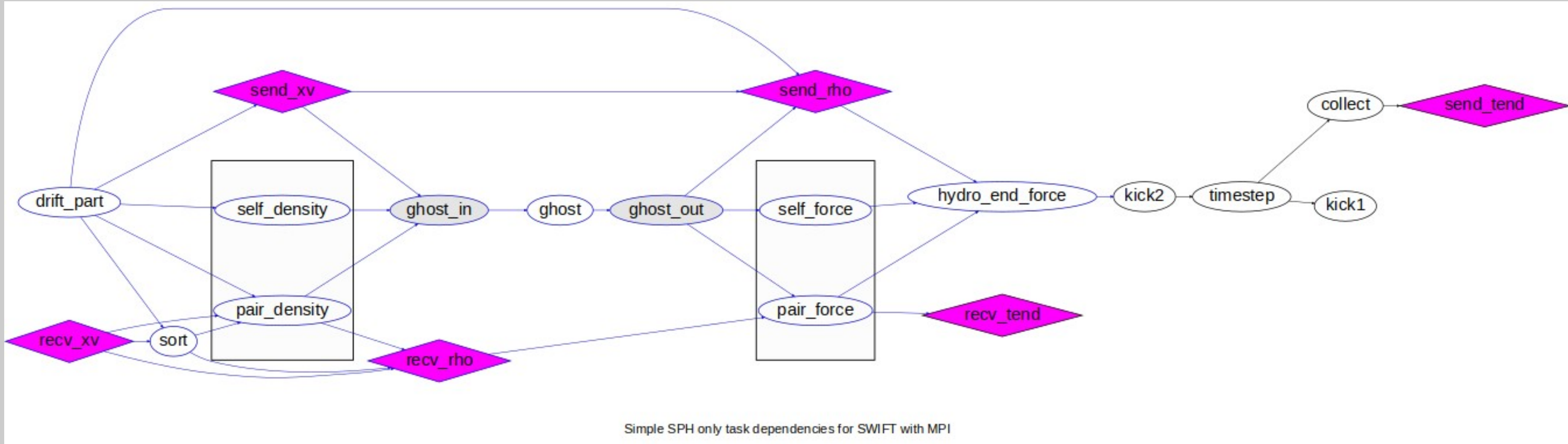
- SWIFT uses an internal task scheduling system based on pthreads
  - allows lower level control than OpenMP
- Computation is split into discrete tasks
  - operating on a hierarchy of grid based cells populated with particle data, either singly or in pairs.
  - Tasks are scheduled when data can be locked and have no unmet prerequisites.



Simple SPH only task dependencies for SWIFT

MPI and task based parallelism

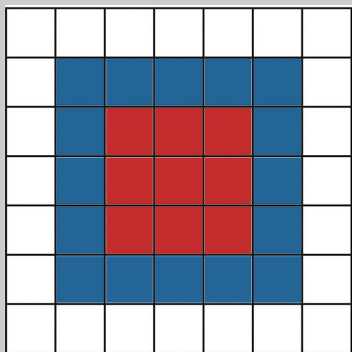
Communications between compute nodes are scheduled, superficially like normal tasks, using asynchronous calls to MPI to maximise the overlap between communication and computation.



Simple SPH only task dependencies for SWIFT with MPI

MPI and task based parallelism (cont)

MPI tasks are bound to a single grid cell, so they send or receive different flavours of particle data or other metadata of that cell only.

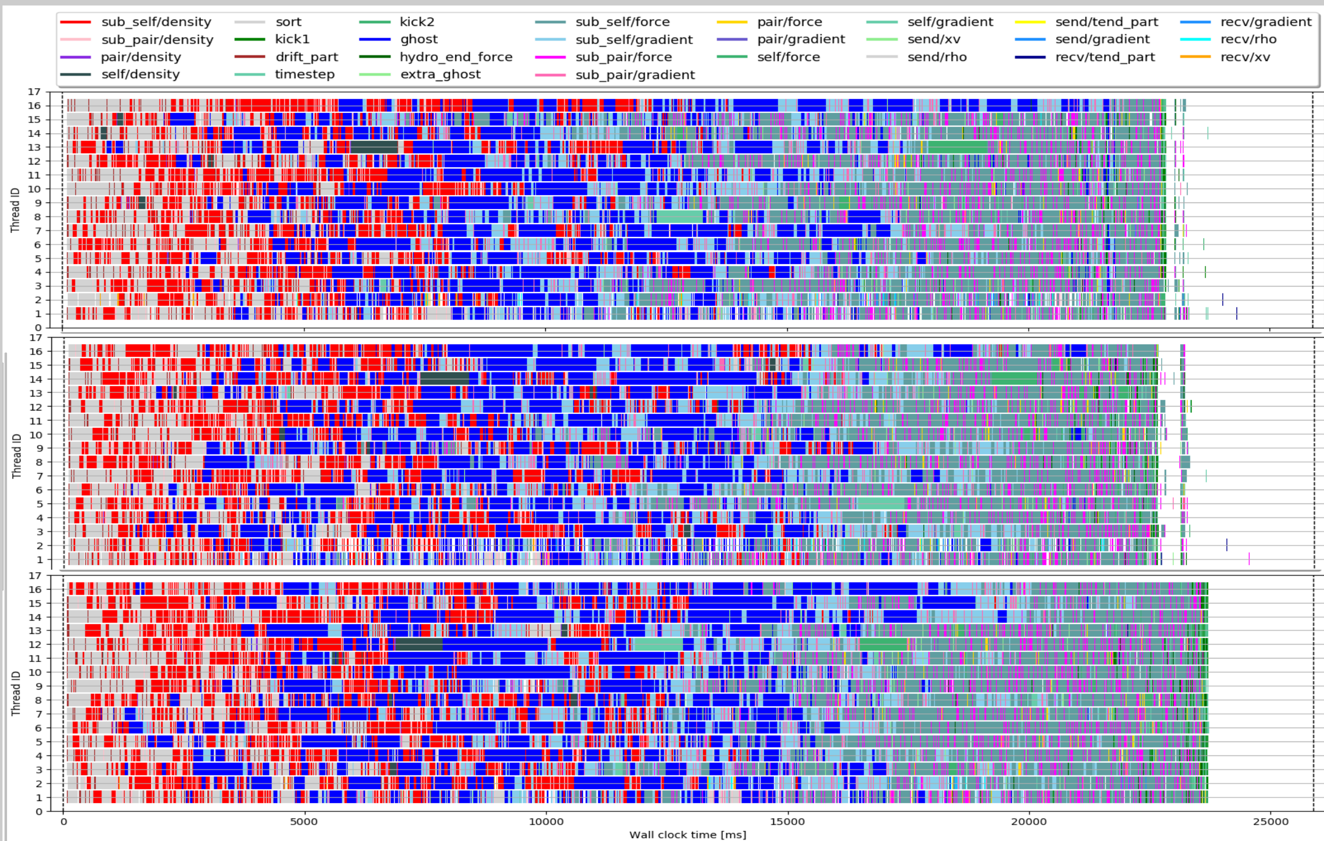


- Only data which is needed for intra-cell interactions (pair tasks) needs to be communicated.

- Foreign cells on MPI ranks provide a halo of cells around the local ones.
  - Effectively read-only copies.
  - It is the job of the MPI tasks to maintain the content of these foreign cells.

SWIFT

Task plots for 3 (of 8) ranks



- Task plots from a single step.
  - Each block is an MPI rank
  - Each with 16 threads
- The different colour segments show the time taken by tasks.
- They do not complete together
  - Imbalance in the loads, but clearly the ranks are kept busy.

How is this implemented

The MPI tasks are implemented using three MPI calls:

- MPI_Isend()
- MPI_Irecv()
- MPI_Test()

MPI_Isend() and MPI_Irecv() are used when a task is enqueued, that is ready to run, so all its prerequisites are satisfied, that is tasks that need to run before it have done so. MPI_Test() checks when either of these operations has completed, at which point the real task runs and unlocks its dependencies and the transferred data can be used by waiting tasks.

MPI calls in a nutshell

```
void enqueue(struct task *t) {
    if (t->type == task_type_send) {
      MPI_Isend(t->data, &t->req);
    }
}

void lock(struct task *t) {
    if (t->type == task_type_send) {
       int flag = 0;
       MPI_Test(t->req, &flag);
       return flag;
    }
}

thread main() {
   while (true) {
      if (lock(t) == true) {
         /* Run task to handle data. */
      }
   }
}
```

The use of MPI calls in the task system of SWIFT In a nutshell.

When ready to be queued an MPI_Isend() is initiated and a data lock to run the associated task is refused until the data is reported as offloaded by MPI_Test().

Note offloaded isn't the same as sent, it just means the local data is no longer needed by MPI and can be modified once more.

A similar arrangement is used for MPI_Irecv(). This should ideally be initiated ahead of the MPI_Isend() and when MPI_Test() reports that done, the message has really been received and the data can be used by waiting tasks.

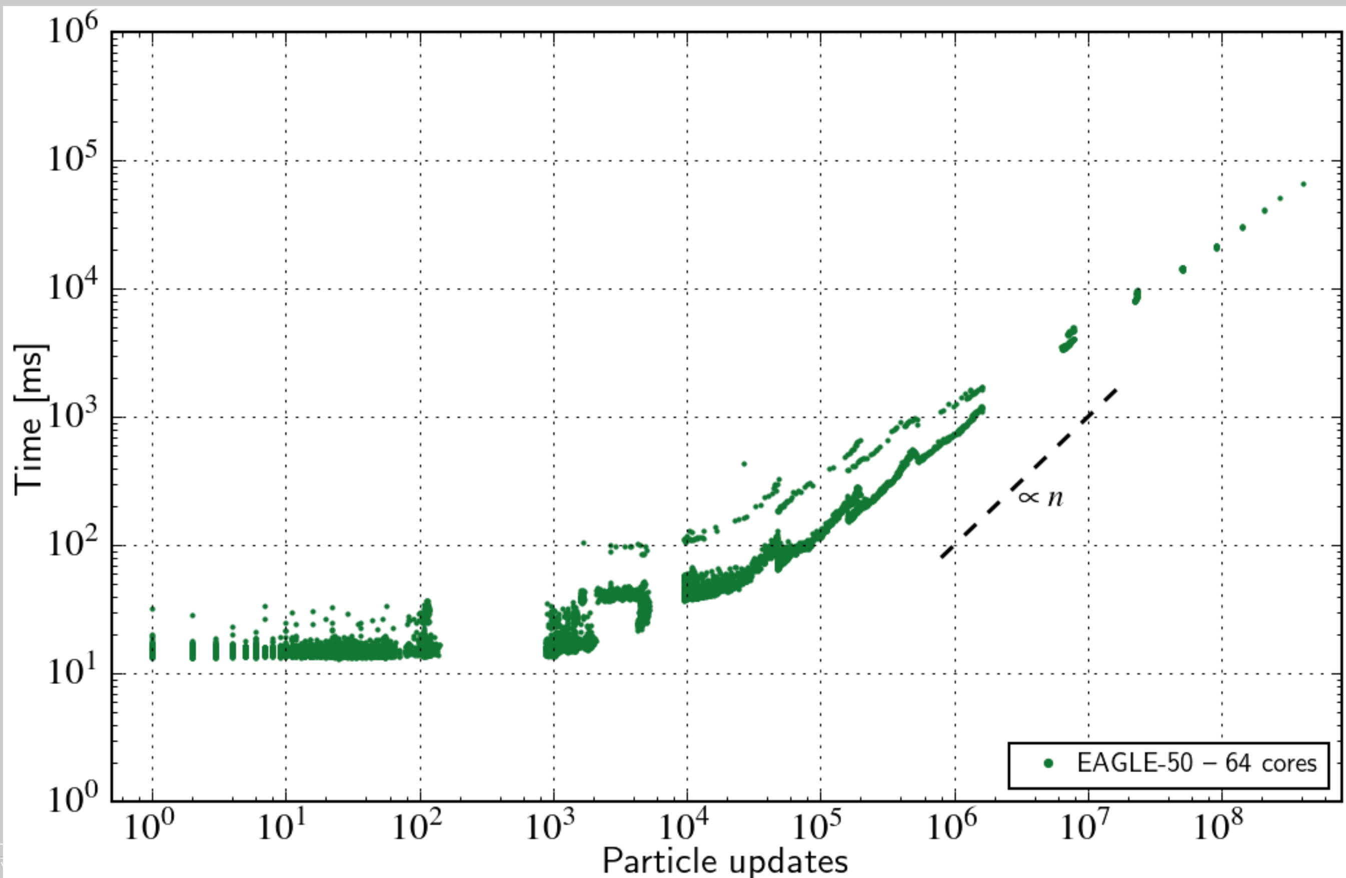
Disclaimer

Although only three MPI calls are used in the task system, in fact a number of other MPI calls (synchronous, asynchronous and collectives) are used when reading in and writing out data, as well as when constructing and communicating the grid cells and their initial contents, and also when data needs to have the MPI rank holding it changed. That occurs when particles drift out of their cell and when a domain decomposition is needed to maintain a good load balance. These are not addressed here.

Multiple time stepping

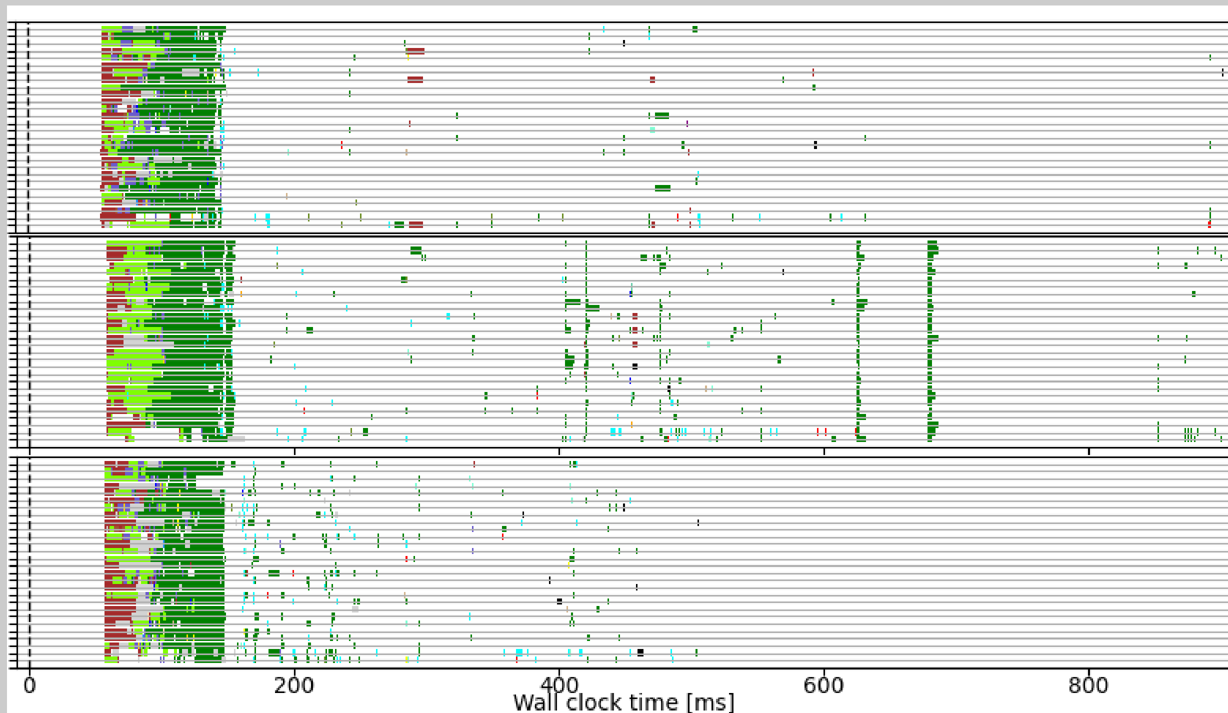
- SWIFT avoids work by not interacting every particle at every step
  - Only those whose dynamics require it (e.g. fast moving ones, etc)
- Not all time steps are the same
  - Huge dynamic difference in the computation required
  - Less computation for the communication to overlap with
    - Communication can dominate for steps when there is little work to do

SWIFT workload



- X axis: number of particles updated in a step. (a proxy for work)
- Y axis: time taken.
- Huge dynamic range (log-log), some steps with a few, some with billions.
- The strong linear regime scales well
- Overheads dominate for smaller steps
  - eg MPI latency

Multiple ranks with limited computation



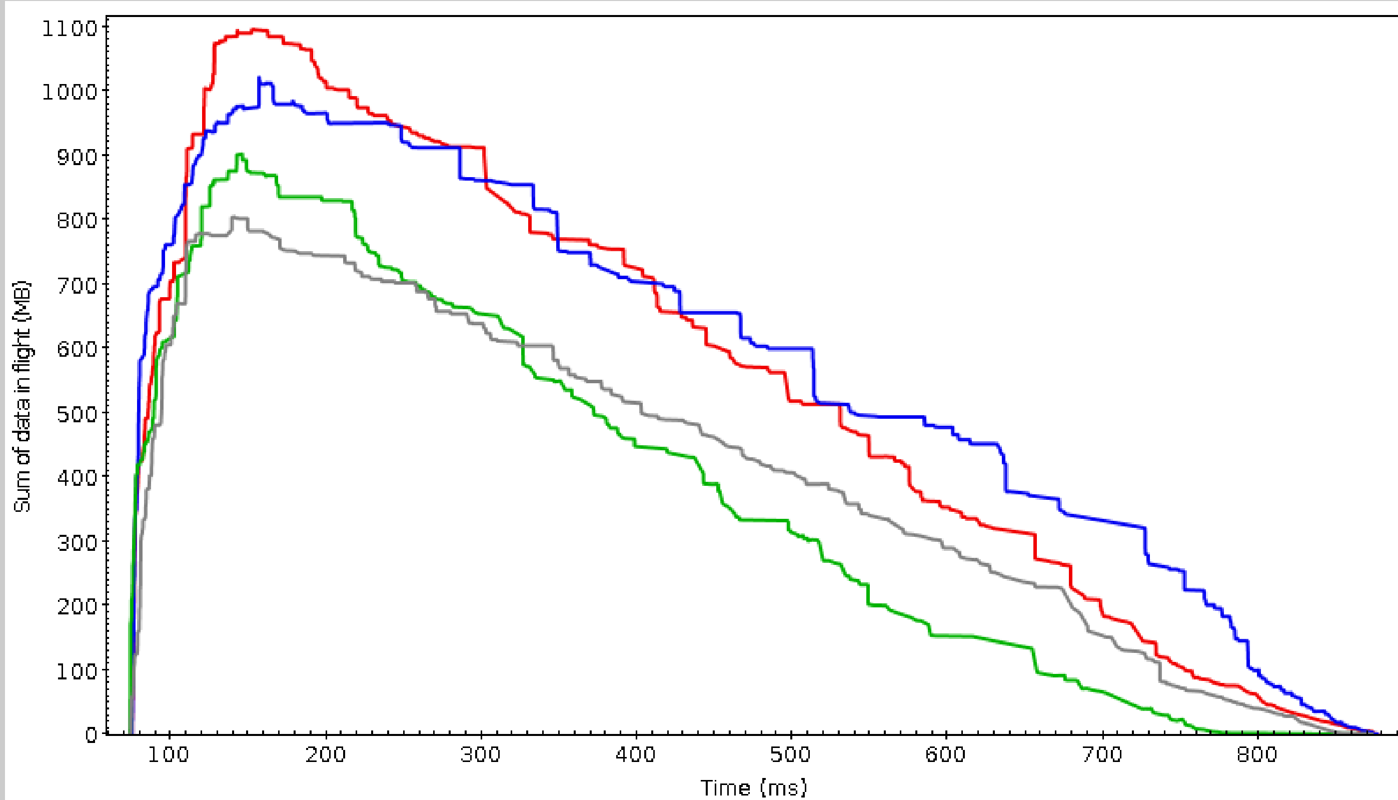
With limited computation not all threads are busy for the complete step. We seem to have a lot of dead time.

Suspects are balance, lots of short tasks (too small to render) or time taken for MPI or is MPI progression playing a part?

Mystery of what is happening.

- Difficult to work out the impact of MPI on the previous task graphs
  - No direct timing of data transfers (just with the MPI calls are made)
  - SWIFT task ordering is non-reproducible
  - Profiling doesn’t help much
- Internal logger implemented
  - Accurate times of data sent and received (and data size)
  - Obvious that MPI is busy during the “dead” times

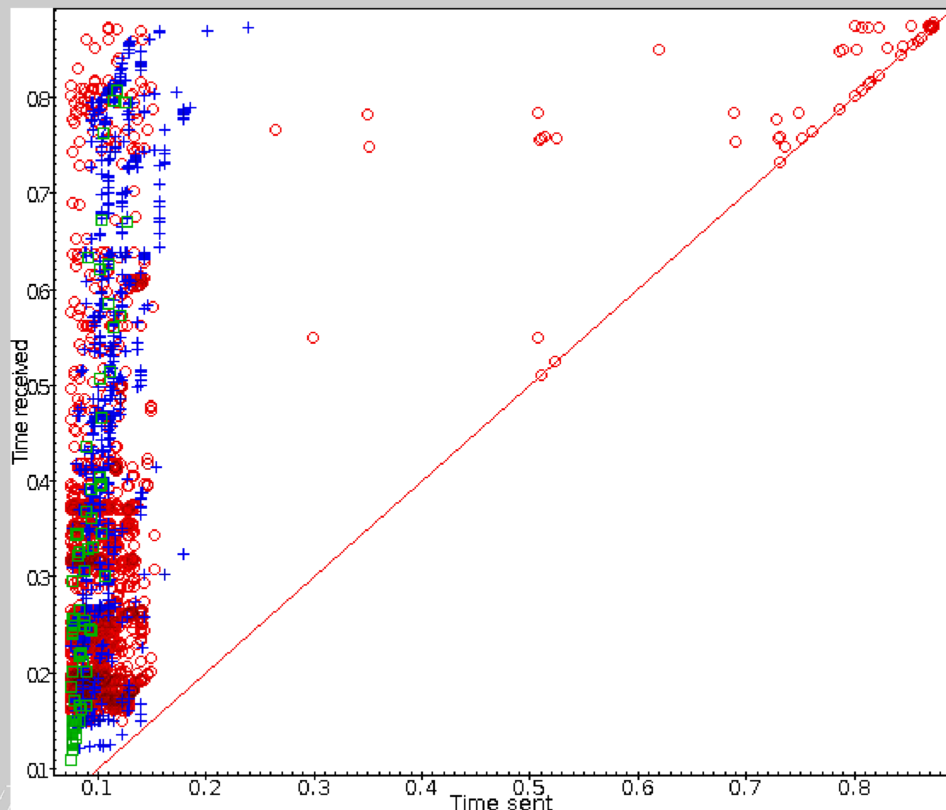
MPI logger: data in flight



Data in flight during a step for a number of MPI ranks. This is the sum of data that have not yet been “received”

Received is not reported as arrived/complete by MPI_Test(), so could be delayed by computation from any active tasks, but there are none of those in later times.

MPI logger: send and receive times

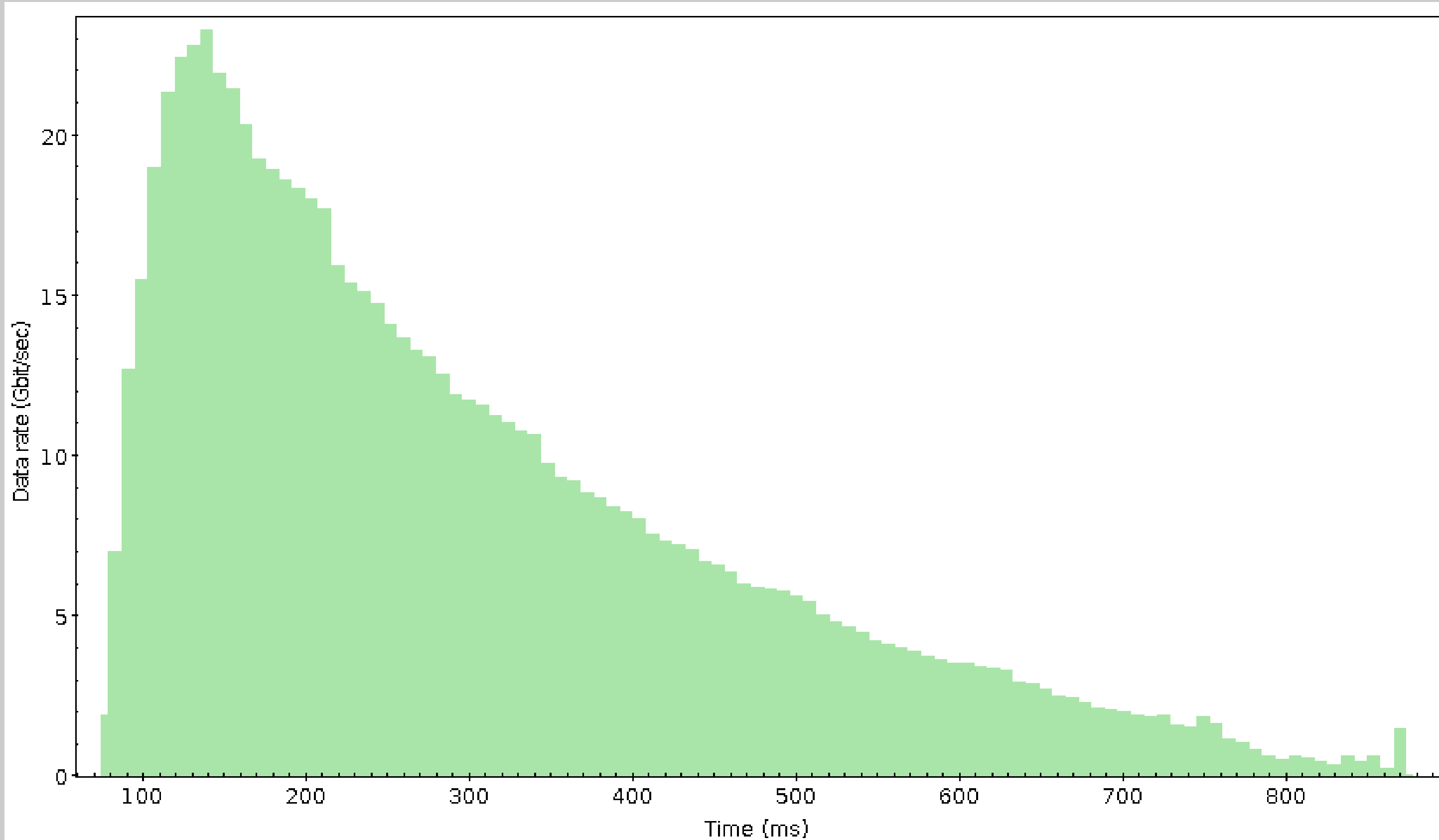


Plot of the time a message was sent against the time it was received.

Clearly it takes a long time to just process all the messages and most are dispatched quite early (this makes sense from the perspective of the task graphs).

The puzzle here is that the network is not bandwidth limited as we can see next.

MPI logger: data rate per rank



The estimated data rate per rank.

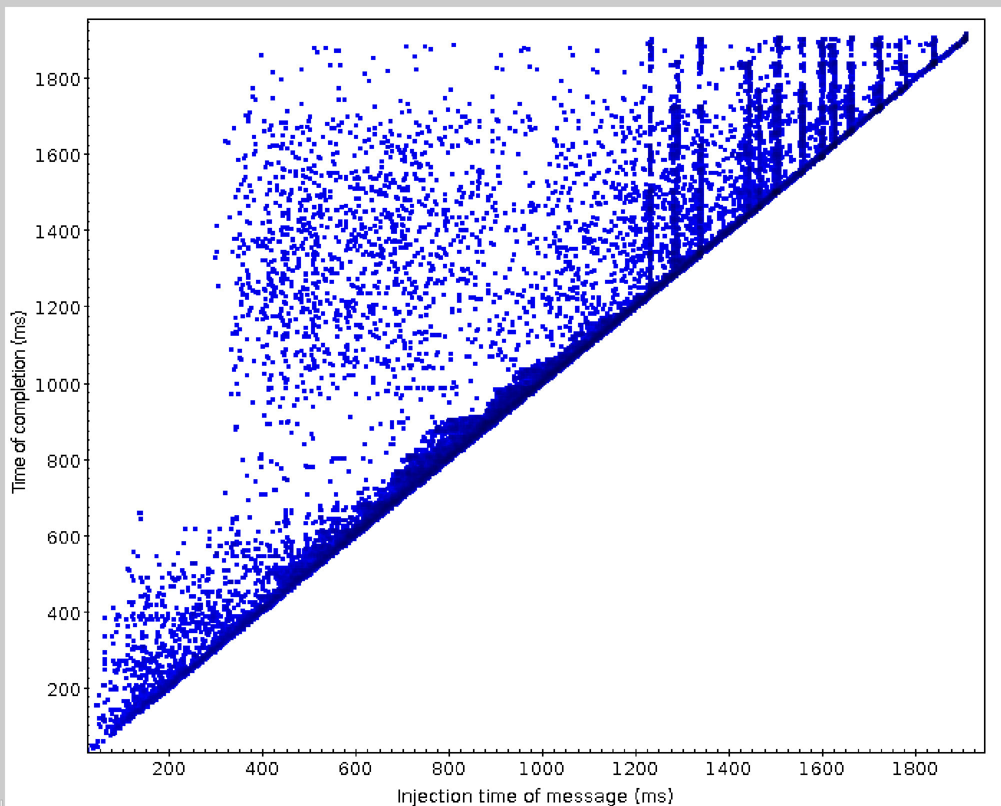
The expected peak here is around 25Gbit/s (one lane of a 4 lane 100Gbit/s card), which we achieve at first.

SWIFT step simulator

- Communication times driving step dead times
- Hard to investigate in SWIFT
- So a play-back tool developed
  - Plays back MPI message exchange using the output from the MPI logger
  - Allows investigation of added delays, number of threads, MPI tuning etc.
  - But didn’t help!

SWIFT

SWIFT step simulator: send and receive times.

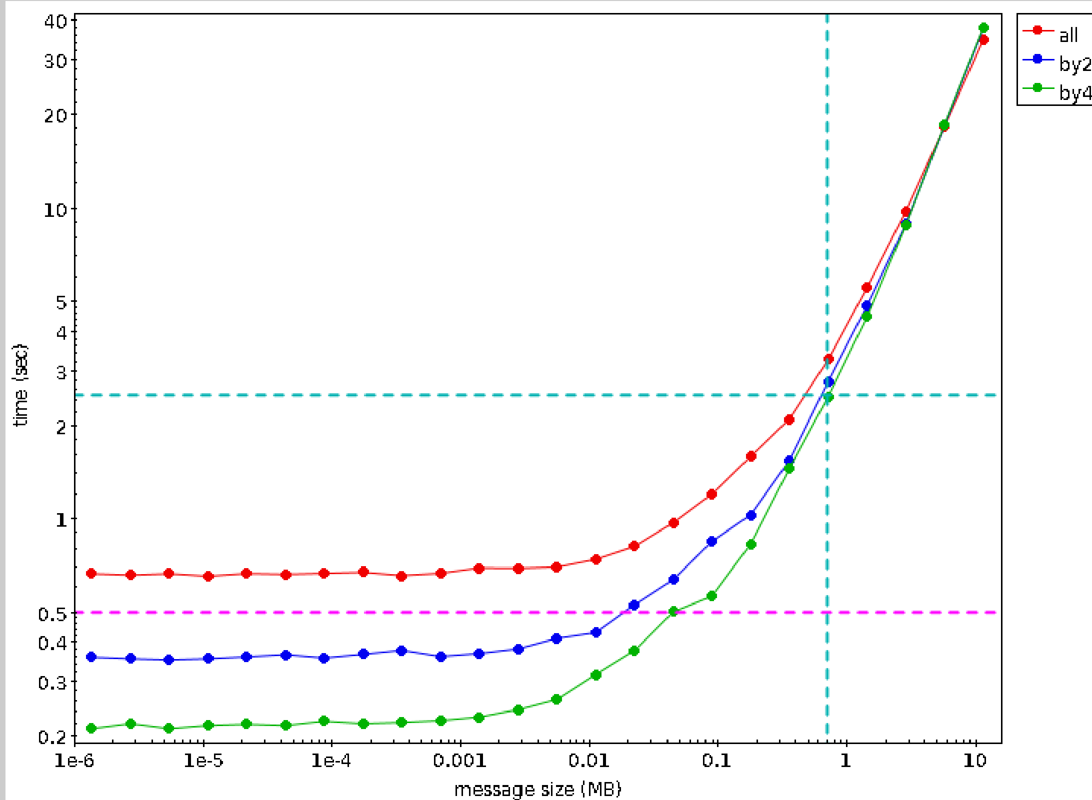


Similar to before when we time the MPI_Isend() and MPI_Irecv() calls.

Once again we see the delay in the initiation of the send and its receipt.

This time we have no other tasks to worry about and can feed in the sends at any rates we want.

SWIFT step simulator: scaling plot



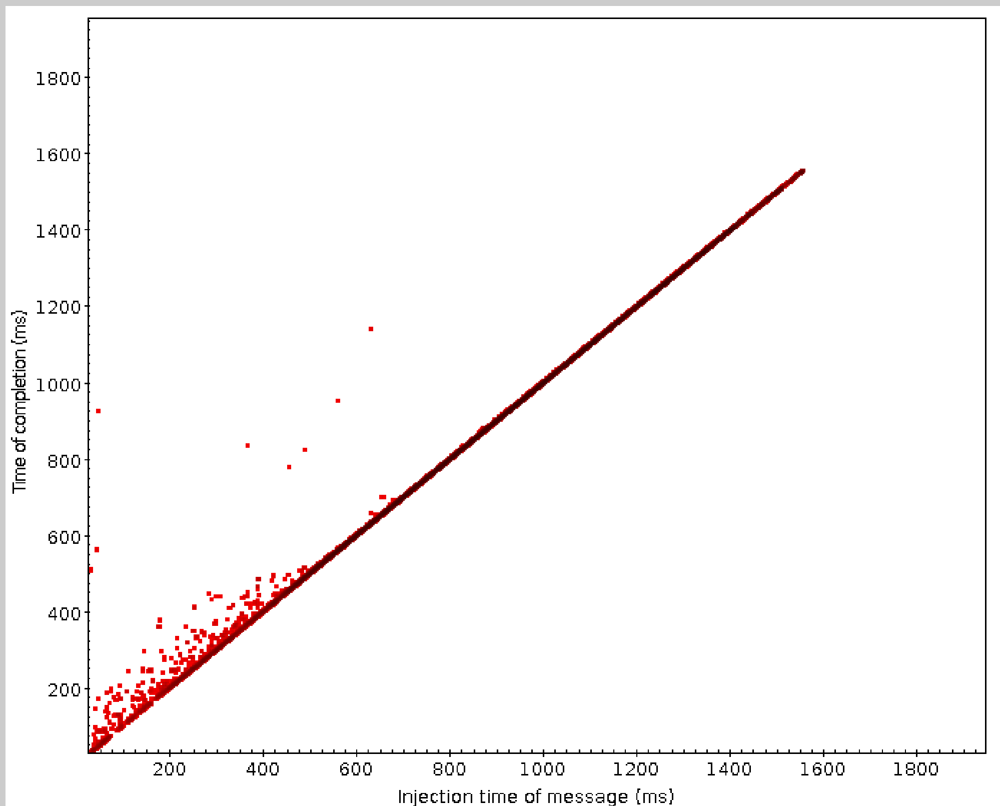
- You can also use the simulator to explore interesting questions like what happens when you keep the number of messages fixed, but scale their sizes.
- In this case “all” is the SWIFT captured log information and “by2” halves the data size and “by4” quarters it.
- Exchanging large packets clearly scales well, but small messages reveal an underlying latency.
- (Note plot is log-log, this tool is also useful to test MPI setups and the capabilities of new fabrics)

RDMA step simulator

Another way to explore the what is happening to the expected performance is to remove MPI completely. To do this an RDMA version of the SWIFT step simulator was developed (not recommended).

This reveals a different send and receive behaviour and is somewhat faster to exchange the same data.

RDMA step simulator: send and receive



Now we have consistent send and receive times with just a small amount of scatter.

In principle this should be similar to MPI, they both use asynchronous exchanges (as you can see from the small amount of scatter).

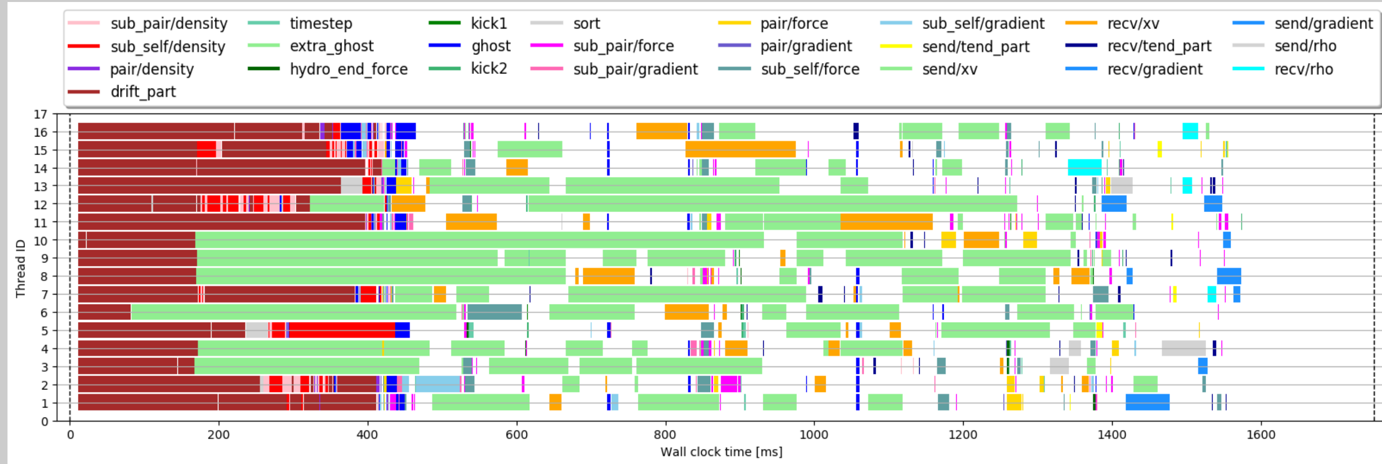
Leaves you wondering what MPI is getting up to.

RDMA SWIFT

- All the previous doesn't help directly with SWIFT, just gives an indication that communication is probably at the heart of the problem.
- Some of which could be down to the asynchronous MPI implementation (the effect of all that scatter).
- Although to be fair the speed up from RDMA isn't huge. So back to SWIFT.

- Using the code ideas from the RDMA step simulator an RDMA layer to replace the MPI calls in SWIFT was created, this is based on different communication model, which is essentially synchronous on the send side and asynchronous on the receive side (a one-sided technique), the advantage to that is we can time the exchanges and associate the time with a task… *A long time passes…*

Task plot from RDMA SWIFT



So we now see that the dead time is taken with the data exchanges. The light green is time taken to send. *But after all this work, RDMA SWIFT is no faster than the MPI version, in fact they are very similar.*

SWIFT RDMA: further explorations.

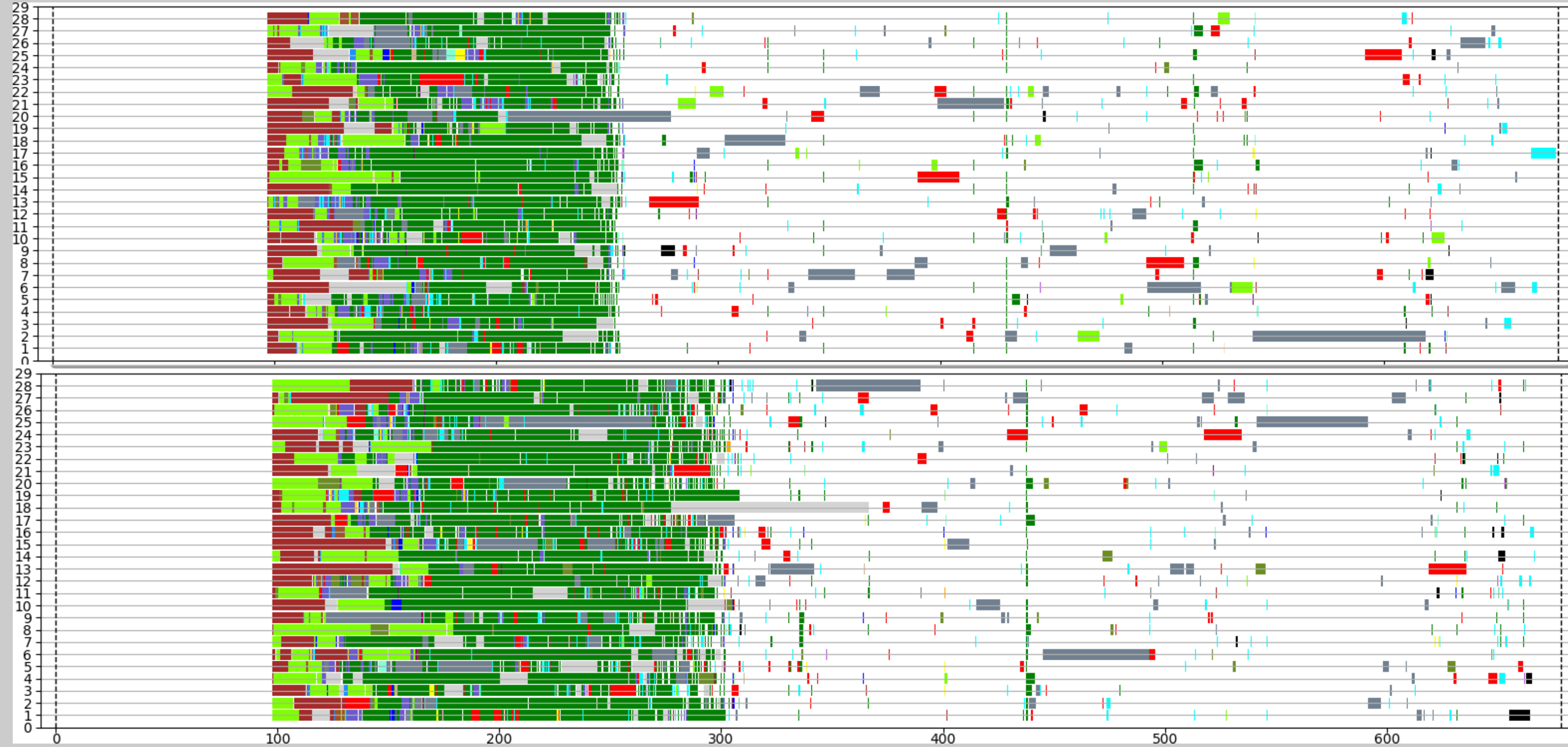
- Oddly receives also take some time, not just the sends. But they have very little to do, in this version just move the data into the local particle buffers.
- Asynchronous MPI will also move the data into local registered memory before RDMA can move it to the remote side.
  - This is one of the reasons on-node memory use increases when using asynchronous MPI (that increase can be a significant overhead and not necessarily controllable).
  - It is also one of the jobs that polling MPI_Test() does and causes the required MPI progression, which obsesses the compute bound world.

SWIFT

SWIFT RDMA: further explorations.

Anyway, the receive task timings suggest we could have an issue with moving memory into and out of the buffers used by RDMA: Could it be that simple after all. Time to add some new tasks that split memory movement into work for multiple threads.

SWIFT RDMA: multiple memory copies



RDMA task plots from a normal run.

SWIFT RDMA: multiple memory copies

Now with tasks that copy using many threads ahead of time.

SWIFT RDMA: multiple copies

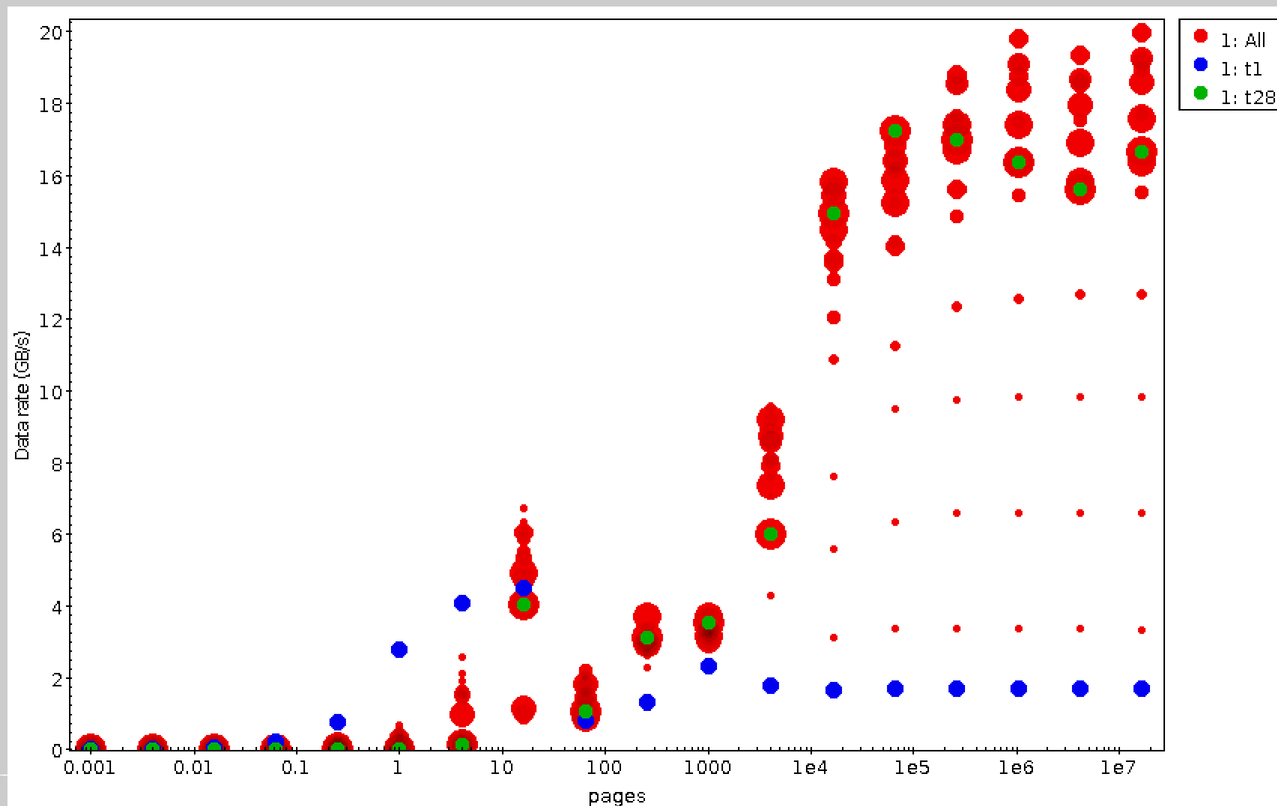
Inspecting the task plots it is clear that copying memory into and from the RDMA buffers using special tasks is faster, at least for the types of step we have been looking into. Sadly for steps that already overlap well with compute, the story isn't as good and the additional work makes those go a little slower. That makes sense we are already using all the memory bandwidth.

So the main issue is not making use of the full memory bandwidth of the nodes. So not MPI as such, although it is playing its part in hiding this detail. This could suggest that our effective memory model isn't good for these sorts of messaging, and it would be better to directly share the memory with RDMA and avoid this overhead. That is a job for the next generation of code.

Progress for MPI SWIFT?

Using some of these improvements in MPI SWIFT isn’t straightforward because MPI hides the memory movements internally. Probably the best we can do it look at splitting the sends into smaller parts, i.e. more tasks, but tests don’t show much improvement, probably because of how the overheads work out...

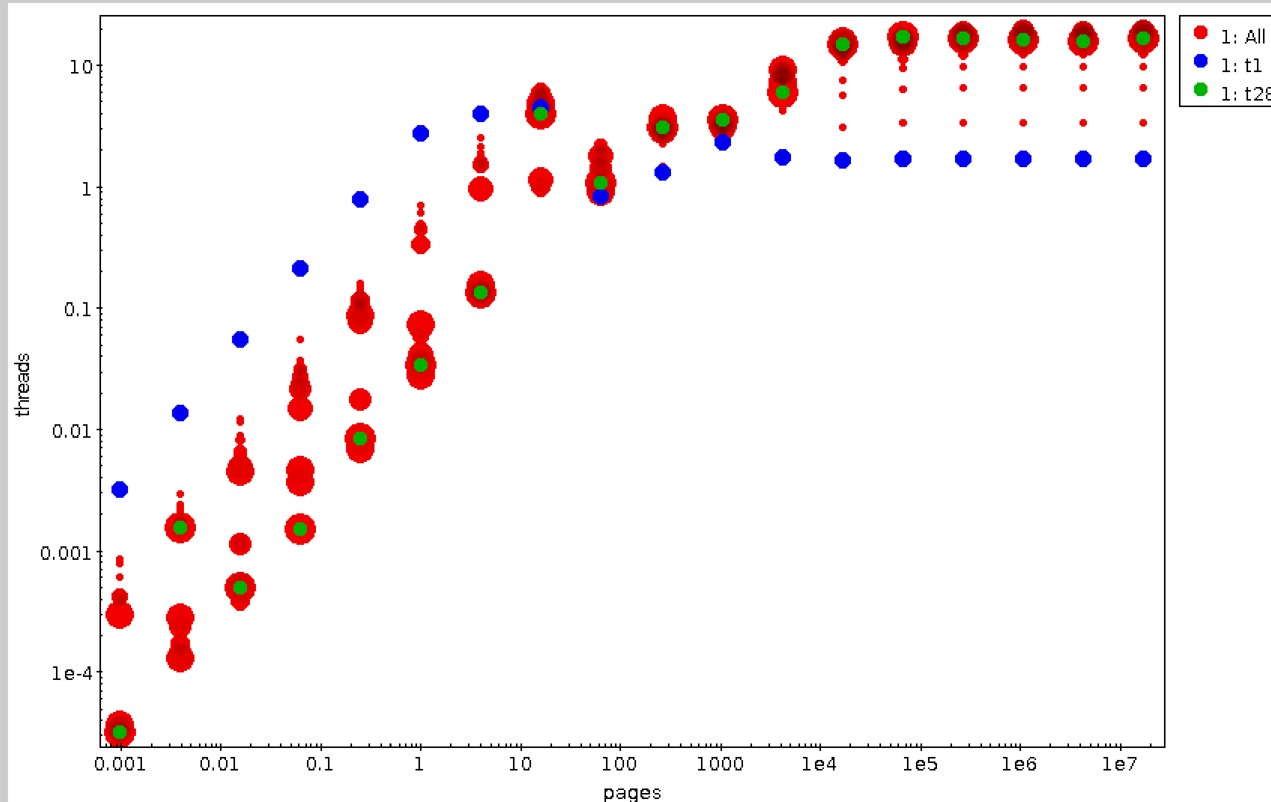
Node memory bandwidth.



Using all the cores to copy isn't always a good idea either…

Strong NUMA and page effects.

Node memory bandwidth.



Future progress

- So perhaps the best we can easily do is wait for better hardware.
- We have faster fabrics
  - Although no improvements in latency
- Cards with increasing capabilities
  - Although we still need to get the data to them
- CXL - sharing memory directly using composability, if you wanted to break down the MPI memory model.
- The upside of MPI is that it hides details of the communication, so we can use tcp, shared memory etc., so that makes it a hard route to choose.

Future possibilities with a shared-memory fabric

- Using CXL to share a bank of RAM between different nodes
- Hopefully MPI would realise the SHM communicator would be best
- Reduces latency by ~3x over a NIC

Full task graph