Implementing MPI RMA in the ONETEP DFT code

Chris Brady, Heather Ratcliffe
University of Warwick

ONETEP

Warwick RSE

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk)

Background

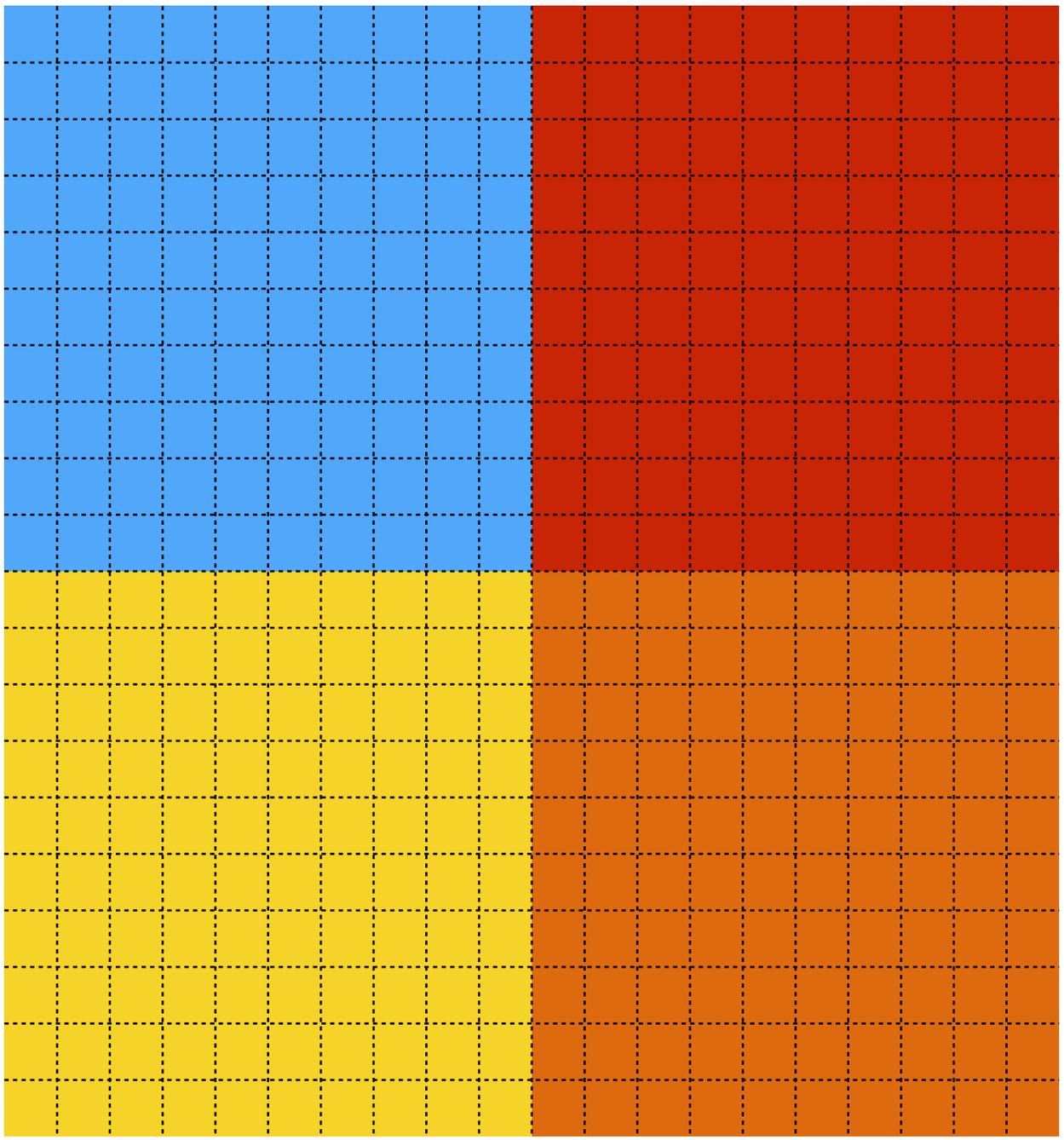
- ONETEP is a linear scaling density functional theory code
  - Calculate electronic structure for sets of atoms
- Conventional DFT codes scale as $O(N^3)$
- ONETEP reduces this to O(N) by localising orbitals and truncating the density matrix
  - This means that it has electron density and local effective potential defined in boxes that have to be either deposited to or extracted from an underlying grid

Background

- ONETEP is a OpenMP threaded MPI code using MPI_THREAD_SERIALIZED level multithread support
  - Performance is **much** worse with MPI_THREAD_MULTIPLE - not worth considering
- MPI decomposition is **NOT** simple domain decomposition
  - A given rank is responsible for a given section of the underlying grid but not necessarily the boxes that overlap that section of the grid

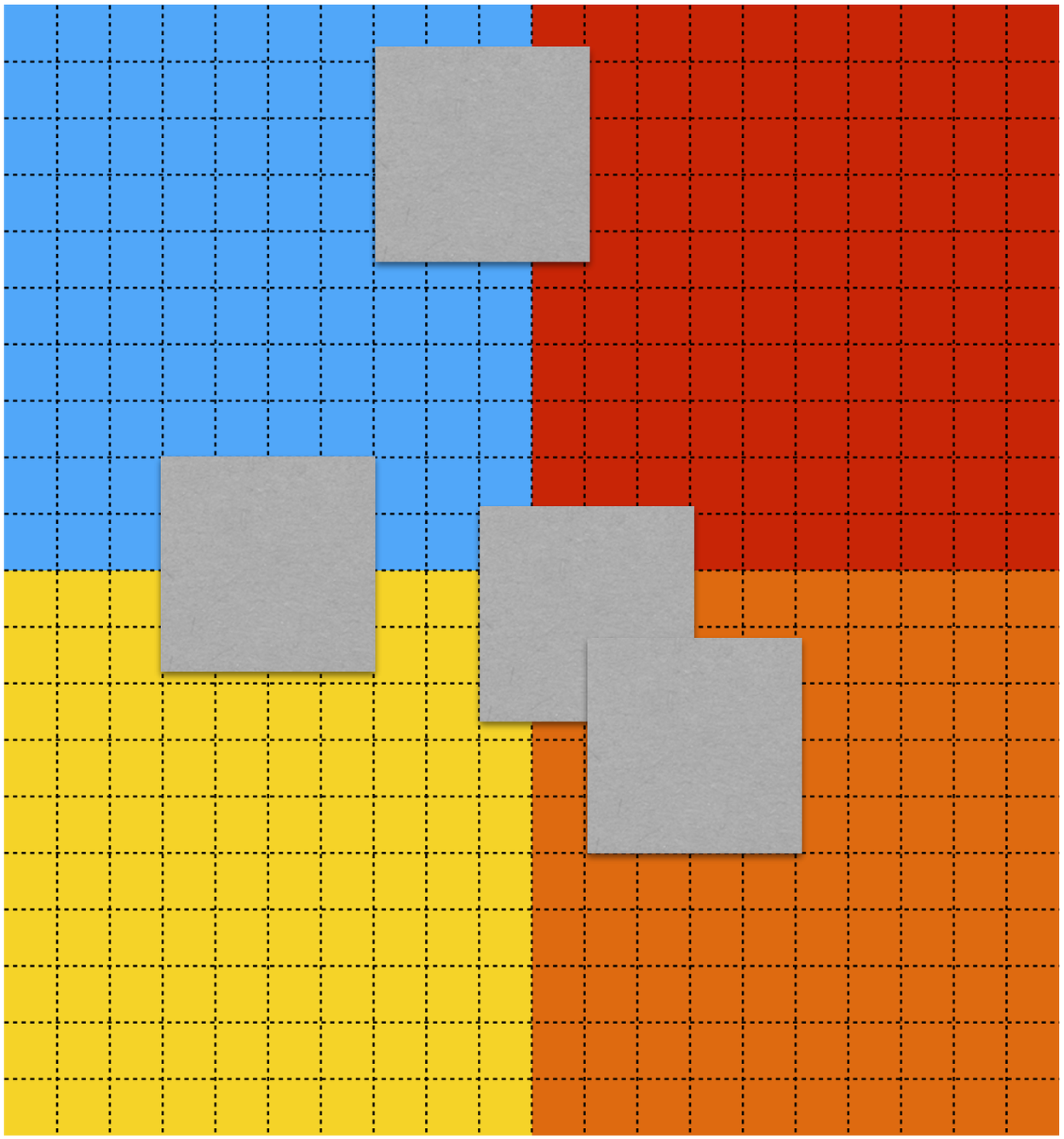
Background

Background

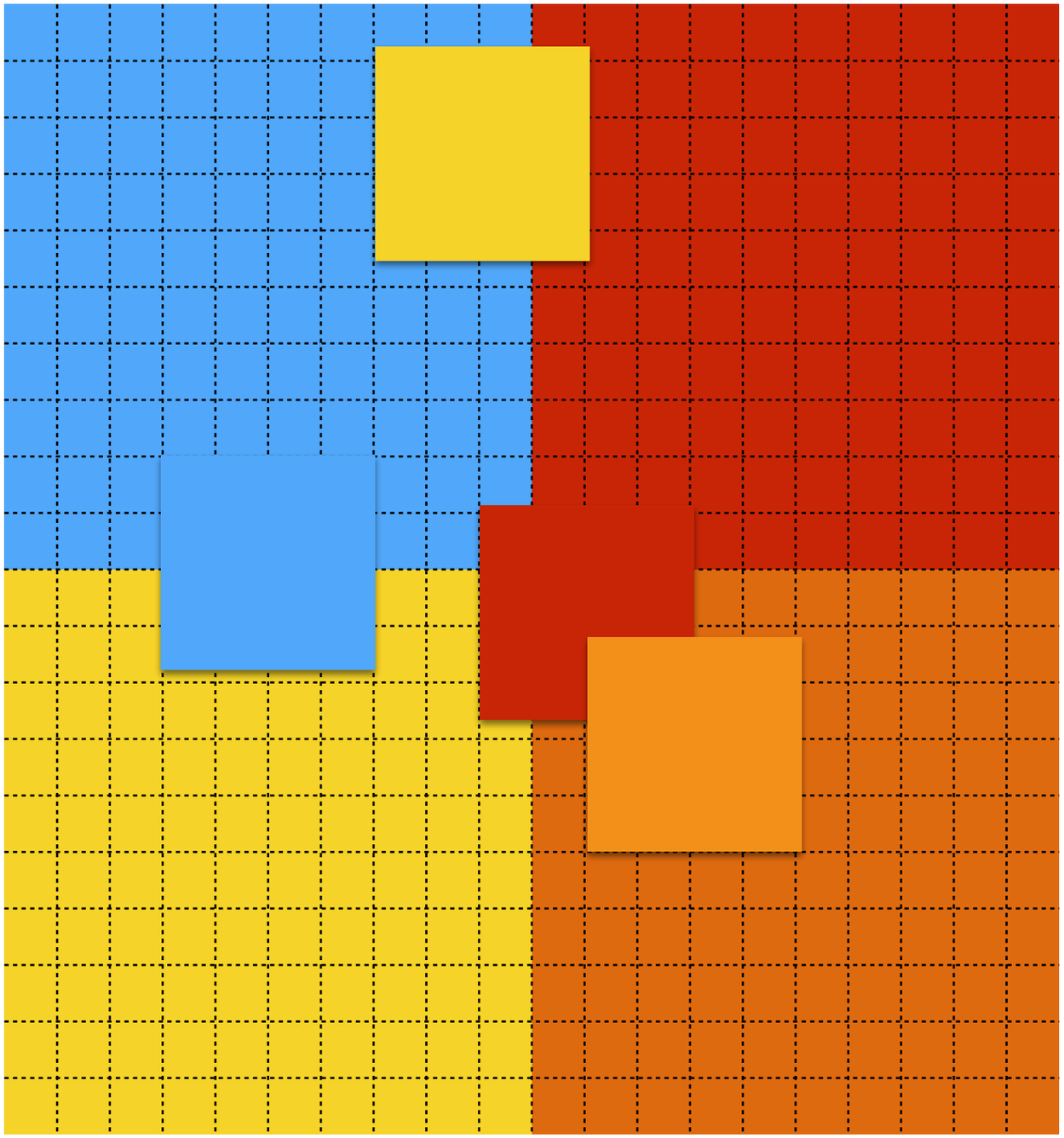


ONETEP
Is actually 3D!

Background



Background



Deposition

- First consider depositing density to the grid
- The density is not simply stored for each box but is calculated just before it is deposited
  - Each box is calculated in a separate thread
- In an OpenMP critical section the actual deposition occurs

Deposition

- All ranks call MPI_Alltoall to describe the part of their current box that they are going to deposit on each other rank
- All ranks then simply loop over all ranks and send and receive the parts of the boxes that are needed by or sent from each other rank
- Received box parts are then added to the underlying grid using local stores
    - Box parts that are on local ranks are accumulated by a local store
- Introduces artificial synchronisation - different ranks only need to synchronise because of the two sided communications.
    - Because of OpenMP critical section threads are blocked as well

RMA Addition

- Performance at high thread counts per rank was limited at high rank count
- The idea of using RMA was to remove the necessity to synchronise across ranks that is enforced by the call to MPI_Alltoall and hence reduce the time in an OpenMP critical section
- The code could also be simplified by using MPI_Accumulate, removing the need for separate transmit and store semantics
- I am thinking here only as an application programmer, I'm not thinking at all about what would be a major annoyance to implement for an MPI library
    - If you spot that I am describing any misunderstandings of the MPI standard, please let me know!

Initial Approach

- The grid to be accumulated to is not a fixed piece of memory so a new window is created every time the density is to be accumulated (and freed at the end)

- The destination array is not updated except by calls to MPI_Accumulate until the window is freed so first thought is that I can use MPI_Win_fence to start the epochs immediately after window is created (and end it before MPI_Win_free is called)

  - Have to reuse the source buffer to MPI_Accumulate so can't do that

  - Would be nice to have source lifetime control in MPI RMA without having to use MPI_Win_lock and MPI_Win_flush_local

Initial Approach

- Would have been nice to be able to have multiple calls to MPI_Win_start/MPI_Win_complete for a single call to MPI_Win_post/MPI_Win_wait

- Instead do MPI_Win_lock_all immediately after MPI_Win_create and MPI_Win_unlock_all immediately before MPI_Win_free

  - Every processor has to (potentially) communicate with every other processor, hence lock_all

Initial Approach

- Actual transfer of the data involves working out the overlap between the source box and the grid domain of every rank
  - If there is an overlap between the source box and a given rank then types for source and destination are created and MPI_Accumulate is called with MPI_SUM operation
- MPI_Win_flush_all_local is called after the loop over ranks to ensure that the source array can be reused
- All individual MPI commands are placed in OpenMP critical sections

Problem

- This worked and gave the same answer as the conventional comms version

- There was an unexpected problem as the problem scaled up
  - Code started to be killed for excessive memory use

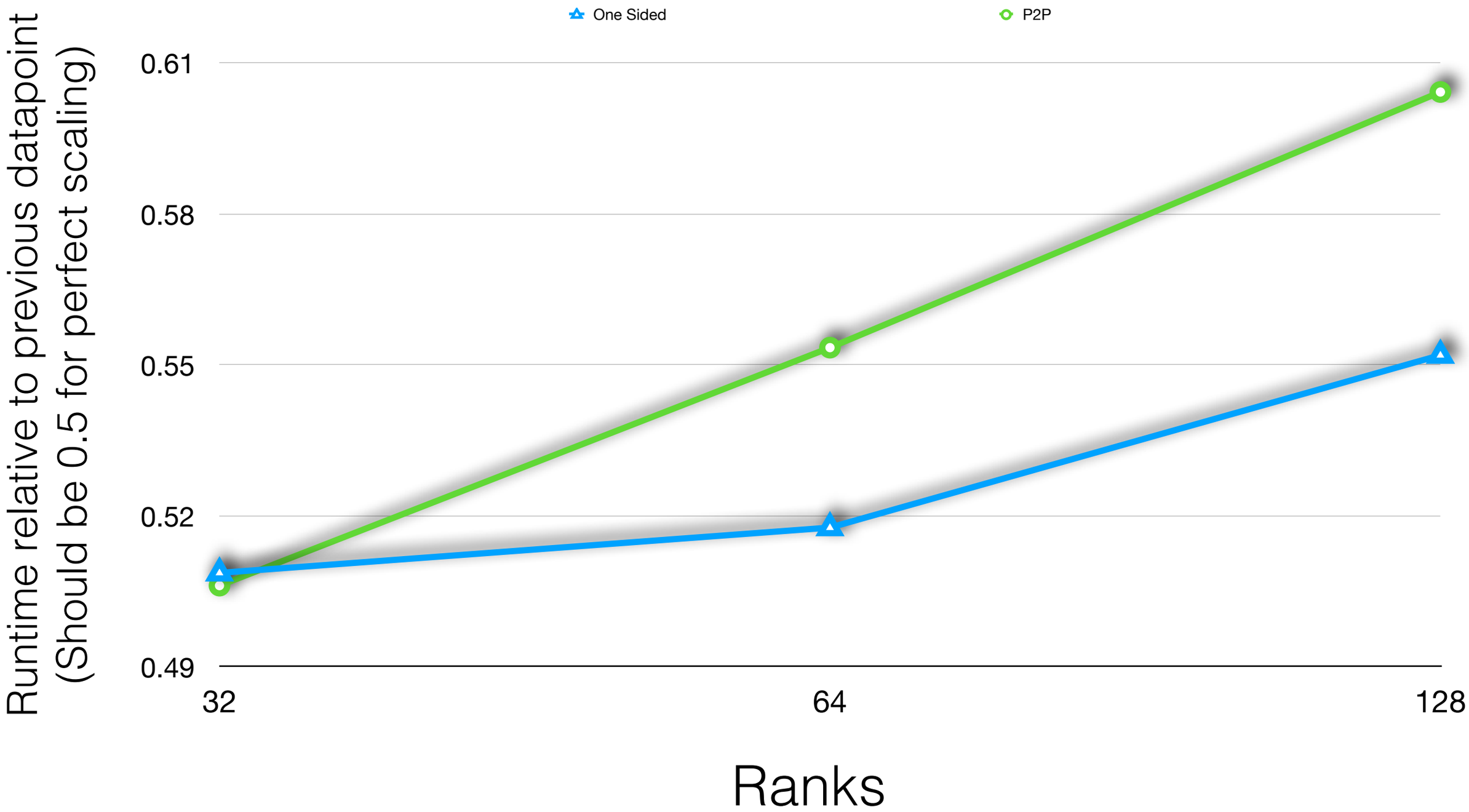
- Turns out that both the OpenMPI and Cray MPI implementations were dealing with my calls to MPI_Win_flush_all_local by copying the input buffer

  - Would be nice to have an option to set a buffer to this and have it block until the buffer has space rather than crashing

Solution

- Tried to replace with call to MPI_Win_flush_all
- Doesn’t actually seem to change behaviour
- So had to switch to another approach
- Switched across to calling MPI_Win_lock as soon as start trying to deposit a box and MPI_Win_unlock once you have finished
- Now have to return to putting the whole deposition of a given box into a critical section
    - Limits performance improvement a bit

Scaling over three doublings



128 Threads per Rank

Strange result

- This is an implementation issue but…
  - Under both Cray and OpenMPI there is a very strange result
  - Scaling of the code is better than with the two sided implementation
  - Actual speed is slower when the code is running with > 1 thread per rank
  - Slowdown is actually at the end of the OpenMP end parallel section (actually inside pthread_join)
    - Not sure why?

Conclusions

- MPI RMA works and does improve scaling
- Current MPI RMA interface isn't natural idiom for this type of problem
  - Would be nice to have more control over local completion in active target synchronisation modes
  - Would be **really** be good to have some way of avoiding local completion in passive target synchronisation modes from causing memory usage growth and that to be specified by standard rather than implementation dependent