* Introduction
  + What I set out to do
  + What I achieved
  + What I will talk about in report
* How I implemented my MPI
  + SPMD: Single Program, Multiple Data • Independent processors run the same program. • Not running in lock step like SIMD. • MPI programming falls into this model.
  + MPI = message passing interface – specification for library interface for passing messages between processes
  + Intel compiler on bcp4 has MPI integrated
  + Each process has its own memory space, with each process running the same code
  + Code will run redundantly unless we share work between processes manually
  + How avoided deadlock
* Halo exchanges performance
  + Affects/limits 1024 most
  + Explain increase in exchange time as number of cores used increases
* MPI scalability (look at lecture 1)
  + How it scales up to 56 cores
  + 1 node vs 2 nodes
    - 2 nodes faster as have more cache available, therefore more memory bandwidth
  + speedup = old time / new time
  + Strong scaling
    - **=** serial time / parallel time on n processors
    - refers to speedup of a fixed problem size as processor count increases.
    - fixed grid size, throw more and more processors at it and see how much faster it gets each time
    - **Perfect strong scaling –** goes n times faster on n processors (linear speedup)
    - Scaling behaviour changes at high processor count as
      * workload per processor get smaller and **suddenly fits into cache**, and so rapid speed up occurs (super linear speedup)
      * communication overhead may cause speedup reduce/drop off (sublinear speedups)
  + Parallel efficiency
    - How well the code is scaling
    - Speedup / processor count (as percentage)
    - Perfect scaling will have 100% PE
    - Useful for plotting scaling of multiple input problems on same graph
    - Strong Scaling efficiency = Time for 1 proc / (time for n procs \* n)
  + Used MPI\_Barrier to synchronise ranks after completing stencil and get more accurate timings (Don’t need non-blocking barriers as collection is dependant on all ranks being finished)
* Possible improvements and conclusion
  + Pack data into char (smaller data type) for speedier halo exchanges
  + Use immediate return send/receive to speed up computation
  + When grid doesn’t divide my workers exactly, divide work up between multiple processes instead of handing all to master rank – master could, in some instances, have to do nearly double the work, slowing down the entire computation
  + Shared memory access for tasks being performed on same CPU – could be more efficient and quicker way to populate halo regions with correct data