An Evaluation of UPC in the Ludwig Application

Alan Gray

EPCC, The University of Edinburgh

CUG 2009, Atlanta

Introduction

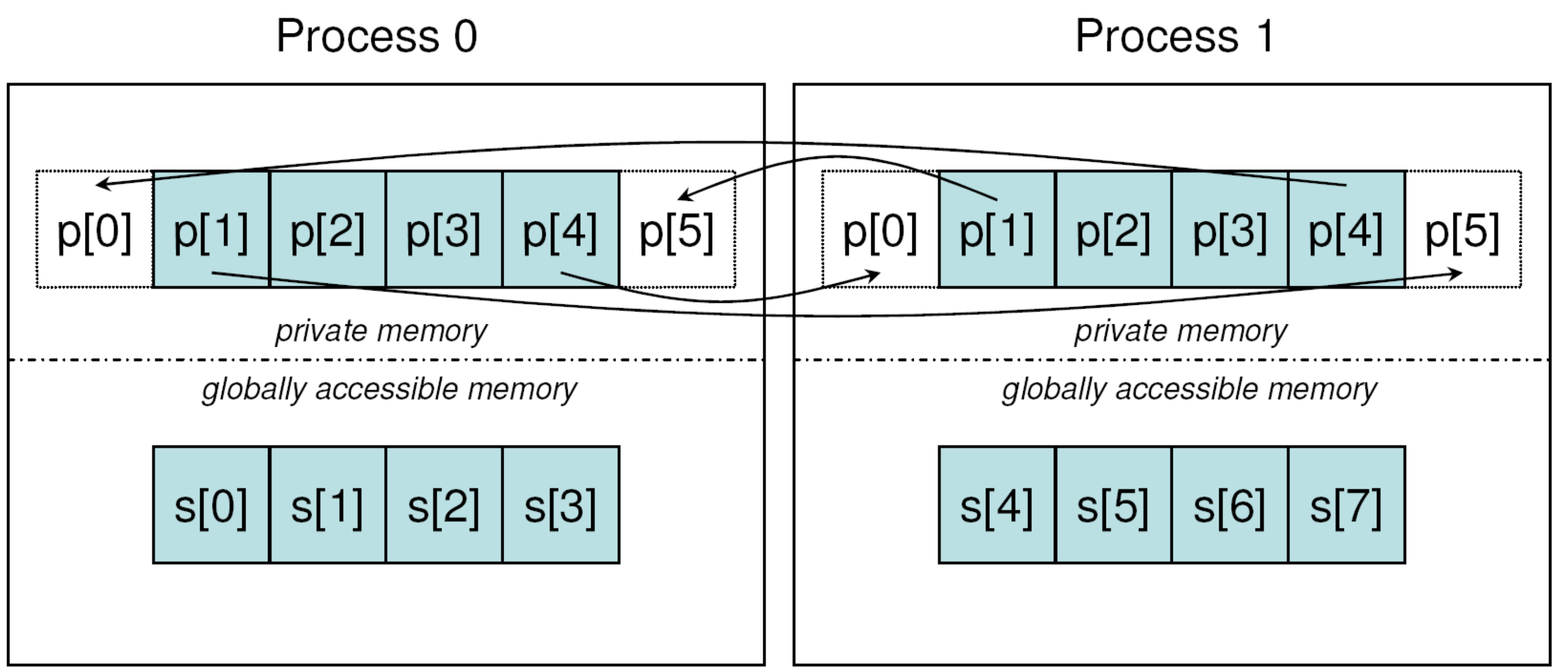
- Modern HPC architectures comprise multiple nodes
  - connected via interconnect
- Applications must utilise these multiple nodes to solve single problem
  - Mechanism needed for each process to acquire remote data
- Message passing (MPI) has become de-facto standard
  - need for complex coding to manage the message passing
  - performance overheads due to underlying 2-way communication
- Novel PGAS languages offer intuitive access of remote data
  - Potentially increase productivity and performance in HPC
- UPC (arguably) most mature and portable PGAS language today

Introduction (cont.)

- AIM: evaluate UPC as a replacement of MPI within real application (LUDWIG)
  - measure performance
- Full conversion beyond scope of work
  - But UPC and MPI can co-exist: can target area of interest
- UPC fully supported at hardware level on Cray X2
  - This study uses X2 component of HECToR (112 processors)
  - UPC will be fully supported on XT after upgrade to GEMINI interconnect

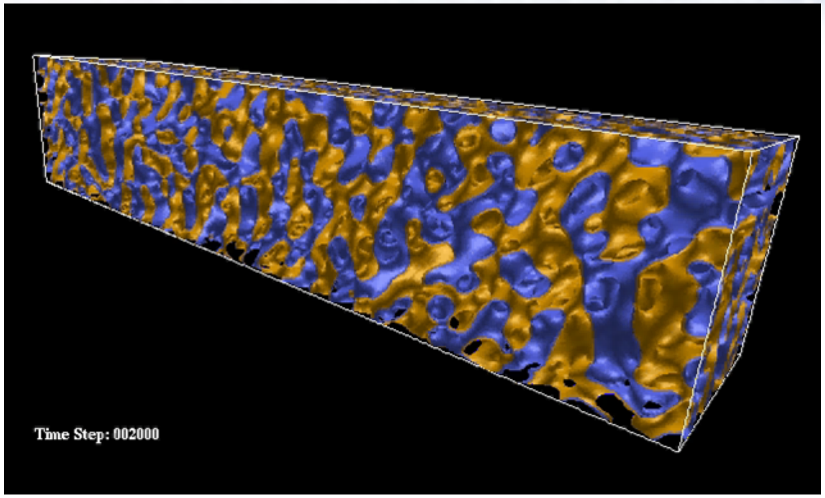
UPC

- Consider simplistic case: 8 elements distributed between 2 processes
  – Where updates require neighbouring values



- Regular C array (local): `int p[6];`
- UPC shared array (global): `shared [8/THREADS] int s[8];`

LUDWIG

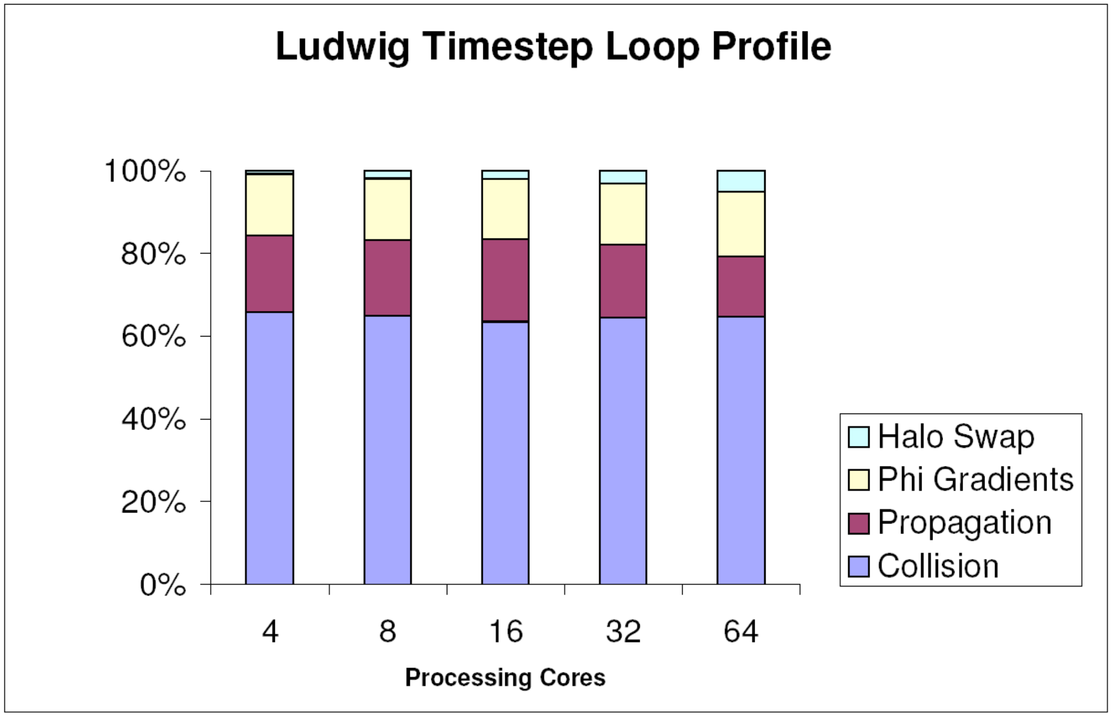


- LUDWIG uses Lattice-Boltzmann models to enable simulation of hydrodynamics of complex fluids (mixtures of fluids, solids/fluids) in 3D
  - Jean Christophe Desplat, Dublin Institute for Advanced Studies
  - Kevin Stratford, Mike Cates, The University of Edinburgh
  - Applications include personal care products, e.g. shampoo

LUDWIG

- Original Code:

```
---------------------
initialisation

loop over timesteps
  Phi Gradients
  Collision
  Halo Swap
  Propagation
end loop

finalisation
---------------------
```



  - Halo cells only accessed in Propagation

LUDWIG Conversion

- Main data structure is array `site[]`, where
  - each element corresponds to a lattice site
    - consists of a struct containing physical variables
- Original Code Propagation section: updates require values from neighbouring sites

```
Loop over index
  …
  site[index].f[0]=site[index-1].f[0]+…;
  …
```

- Halo cells + message passing halo swap routines required

LUDWIG Conversion

- Strategy: mirror site with UPC Shared structure **`s_site`**.
  - New functionality:

`sindex[index]` Mapping of local (**`site`**) - global (**`s_site`**) index

`put_site_in_shared()` Copy data local -> shared

`get_site_from_shared()` Copy data shared -> local

- Allows for specific area of application to be targeted
  - Propagation section adapted to work with shared arrays

```
Loop over index
  …
  s_site[sindex[index]].f[0]
                 =s_site[sindex[index-1]].f[0]+…;
  …
```

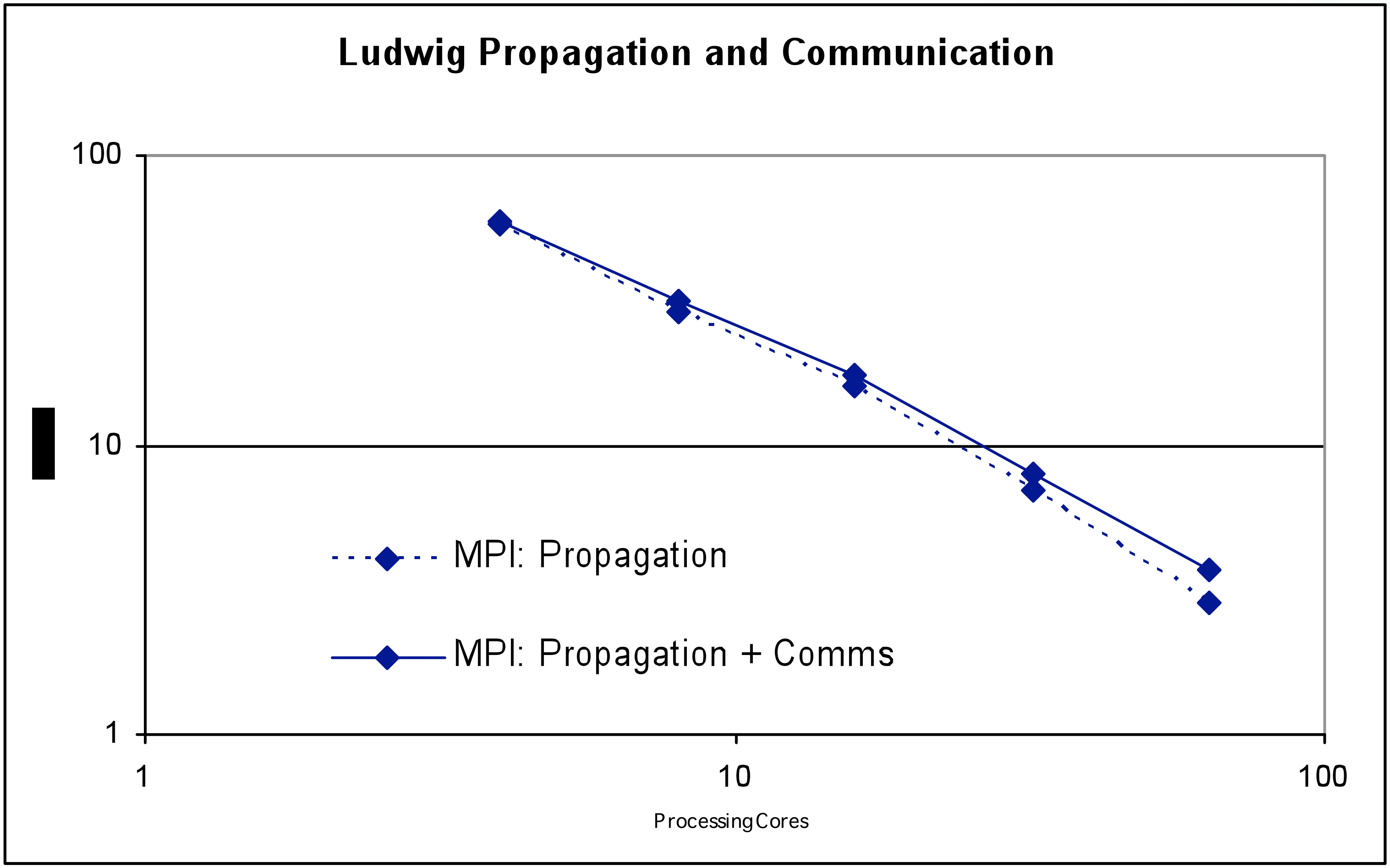
- No halo cells/swaps needed, remote accesses done directly

LUDWIG Conversion

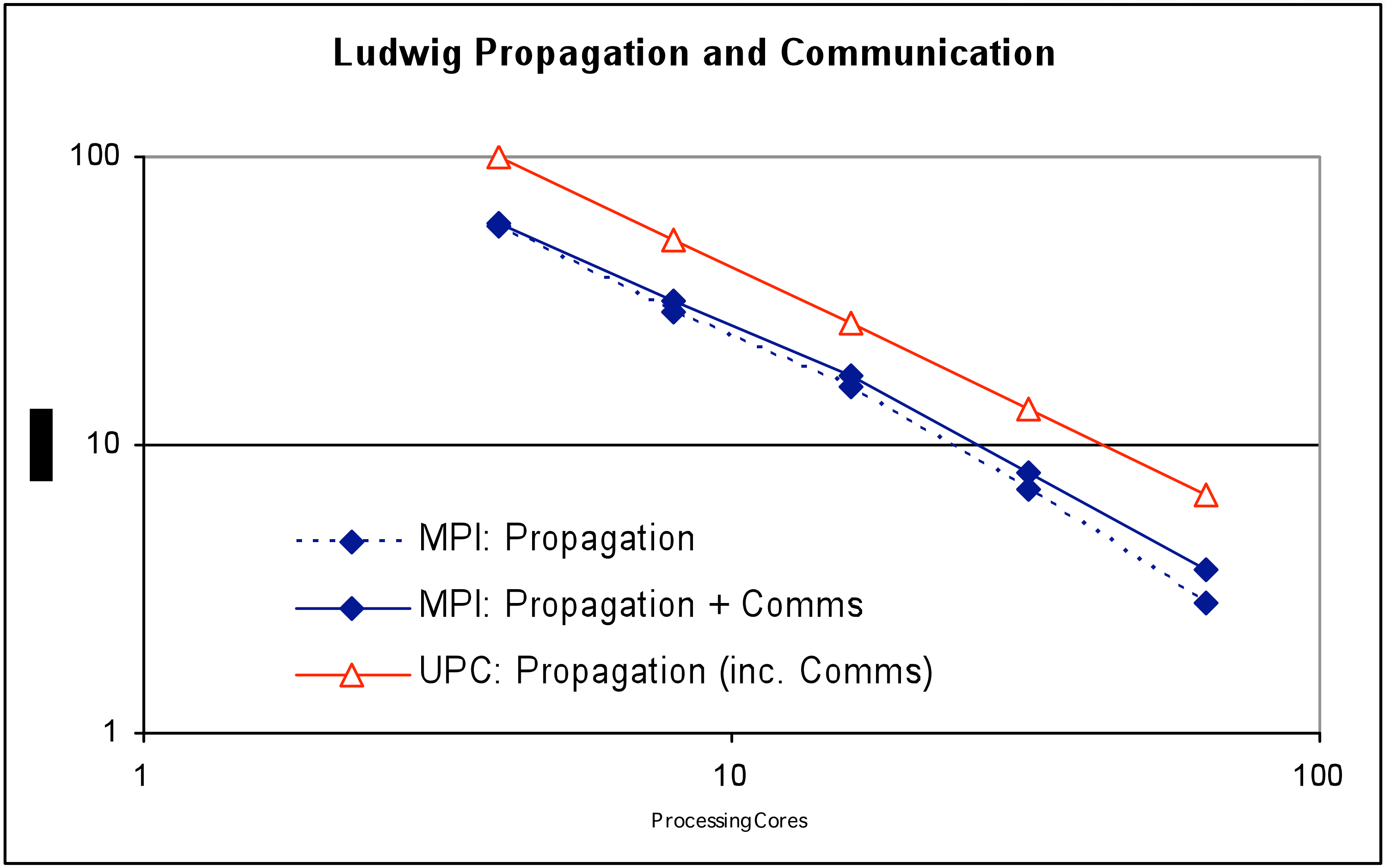
- Modified LUDWIG code:

```
--------------------
initialisation (including creation of local->shared mapping table)

loop over timesteps

  Phi Gradients
  Collision
  //Halo Swap no longer required
  copy local array to shared array
  Propagation (using shared array)
  copy shared array to local array

end loop

finalisation
--------------------
```

Performance results



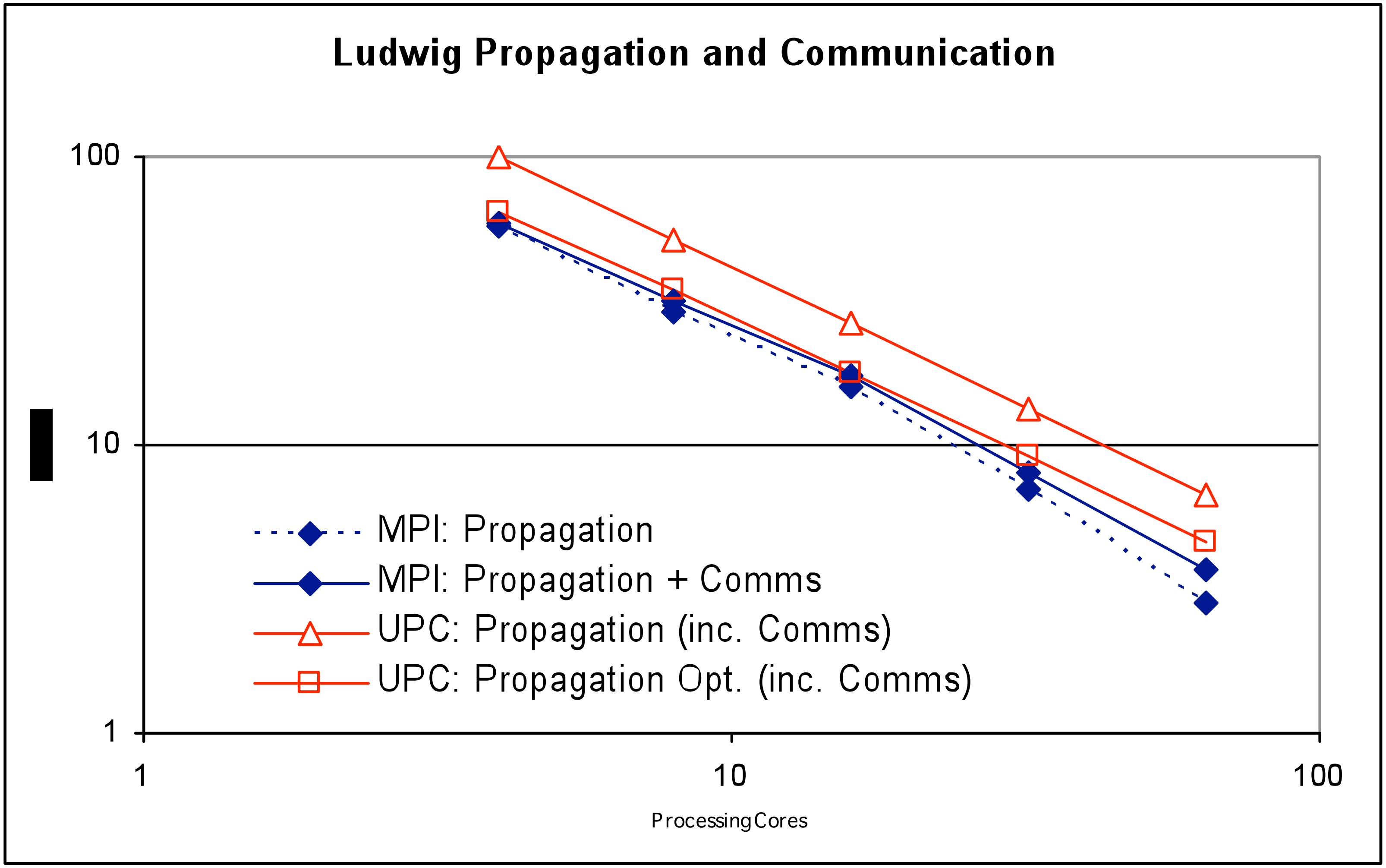
Performance results



Performance results

- Naïve adaptation has substantial negative impact
- Underlying communication is not cause of this
- Shared pointer dereferencing more costly than for regular pointers
- Optimised version: access memory through regular C pointers where possible
  - Obtained by casting from shared pointers
  - Boundary updates must still use shared array accesses to get remote data.

Performance results



Conclusions

- UPC allows for intuitive access to remote data
  - Potentially increasing performance and productivity in HPC
- LUDWIG adapted to utilise UPC functionality
  - Focusing on key section
  - Shared structures remove need for complicated halo swaps
- Significant performance degradation with naïve adaptation
  - Due to sensitivity to costly shared pointer operations
- Optimised version uses regular C pointers to access data where possible
  - Performs similarly to (but slightly worse than) MPI version
    - remaining degradation likely due to remaining shared pointer operations
- Would be interesting to test on larger system (inc. future Cray XT)