Type oriented parallel programming and scalable linear solvers

CCS-EPCC Workshop

Nick Brown
EPCC
n.brown@epcc.ed.ac.uk

Overview

- Type oriented programming
- Mesham
- Case study: Asynchronous Jacobi
- Further extension to scalable linear solvers
- Conclusions

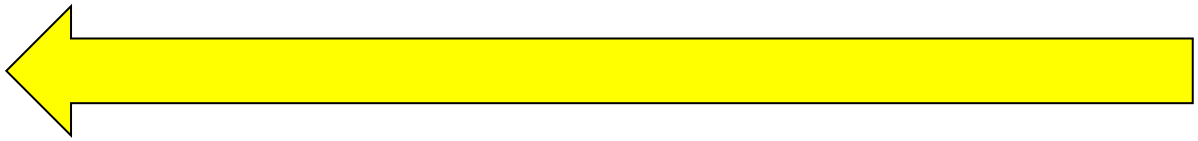
Background

- Writing parallel codes is far more complex than sequential code
  - Issues such as data locality, task placement
  - Modifying an early decision such as distribution method can require an entire rewrite
  - Commonly use lower level sequential languages for a variety of reasons
  - Languages with many options can quickly become bloated
- This problem will only get worse as we move towards exascale and the challenges become greater
- Trade off between programmability and performance
  - Control vs abstraction

Type oriented programming

register const volatile int a;

var a : Int :: volatile[A::B::C::D] :: const :: register[“ax”] ;



precedence

a : a :: writable;

a:=99;

Changing the variable type from that point on

(a :: writable):=99;

Changing the type just for this single expression

The type matters whenever the variable is “used”

What does this give us?

1. Opportunities for optimisation

By the programmer specifying their code in this high level manner the compiler can obtain a much more complete view of the system and apply optimisation.

2. A choice between explicit and implicit programming

In the absence of type information the compiler can apply sensible defaults

3. The right person for the right job

An expert in the field might create the type using their in depth knowledge and experience. The user of the type need not understand all the underlying complexities.

4. Changing fundamental aspects is trivial

For example changing the method or partitioning or data distribution only requires a change in type

Mesham

- Core language centred around a simple imperative model with extensions to support type oriented programming (and some other stuff such as parallel extensions and exception handling.)

- An external type library with 48 types
  - 23 basic types
  - 25 more complex types

- A vehicle for experimenting with types and our application of them

- Multiple example codes implemented in Mesham

http://www.mesham.com

An example

var a : Int :: allocated[single[on[1]]];

"a" is an integer allocated to process 1

var b : Int :: allocated[single[on[3]]];

"b" is an integer but only allocated to processes 3

a:=b;

"a" on process 1 gets the value of "b" which is held on process 3.

This simple shared memory style guarantees safety and consistency but it might not be particularly performant

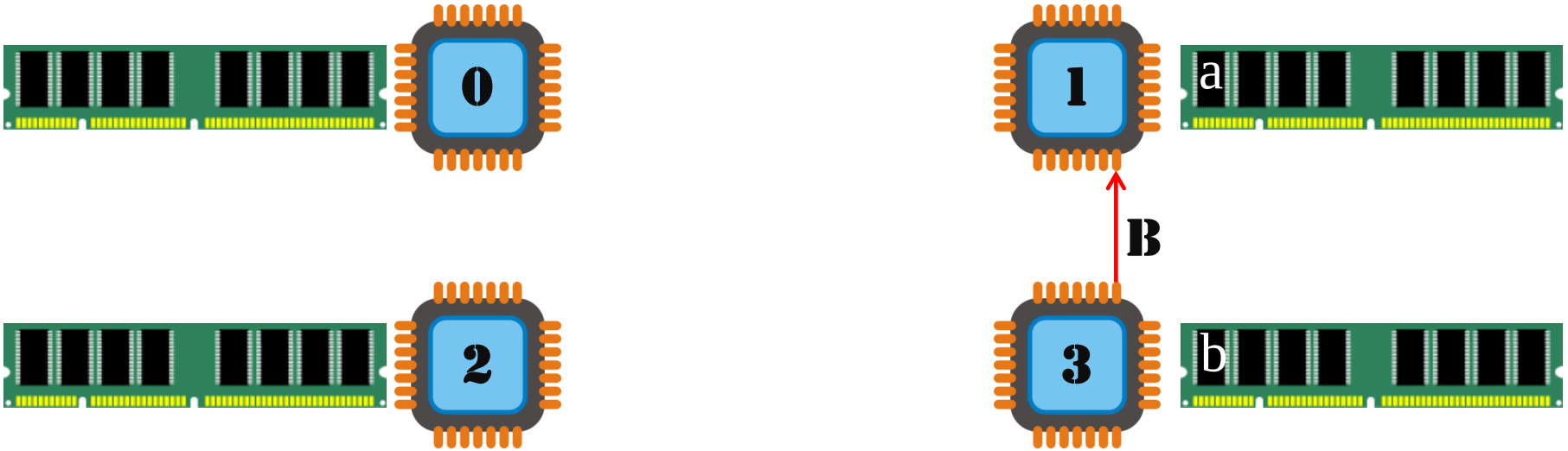
An example

```
var a : Int :: allocated[single[on[1]]];
var b : Int :: allocated[single[on[3]]];
(a :: channel[3,1]):=b;
```

"a" is an integer allocated to process 1

"b" is an integer but only allocated to processes 3

Point to point default (blocking) communication



The programmer has explicitly controlled some aspects of parallelism which might be more performant but we make no guarantee to the safety or consistency of this

Case study – Asynchronous Jacobi

- Jacobi's algorithm is the simplest iterative solution method

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}\right), \qquad i = 1:n.$$

  - Stop iterating when the residual $\|Ax - b\|_2 £ tol$

- Requires communication between iterations
  - For halo swapping data between processes
  - For calculating the global residual
  - Traditionally this has been synchronous

- Can break the lock step algorithmic nature by using asynchronous communication
  - To achieve scalability & fault tolerance

Case study – Asynchronous Jacobi

- Three versions:
  - Synchronous
  - Safe asynchronous
  - Racy asynchronous

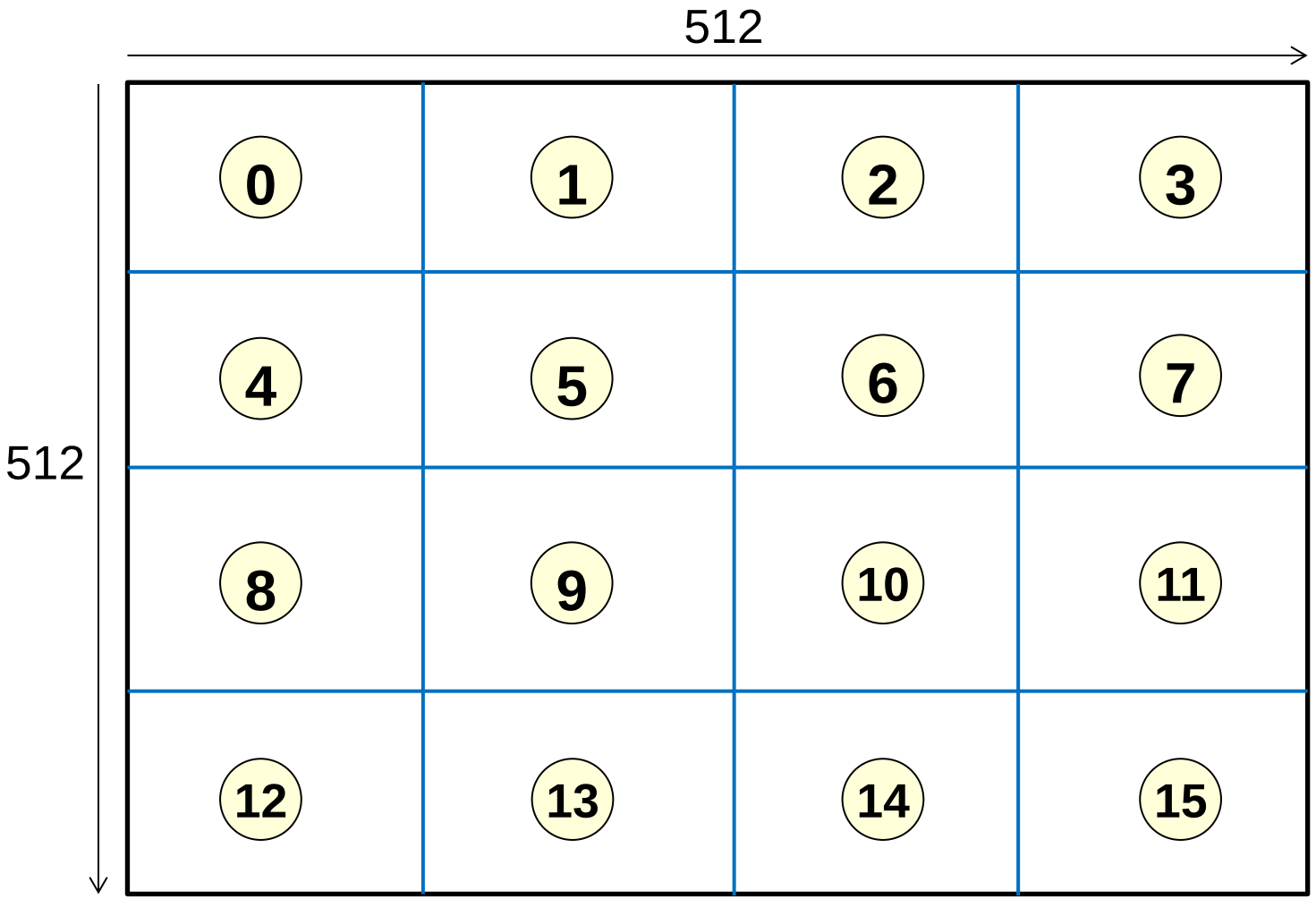
- Pollution problem
  - Based upon Laplace with fixed boundary conditions
  - Pollution problem (3d diffusion)

- However……
  - Asynchronous algorithms project have demonstrated potential benefit however **the asynchronous (F90 + MPI) code is far more complex and required a rewrite of the synchronous code** to contain tricky and uninteresting bookkeeping

Can type oriented programming help here?

Partitioning and distribution

var data:array[Double,512,512]::allocated[grid[halo[1],4,4]::single[evendist]];



Synchronous

var data:array[Double,nx,ny,nz]::allocated[grid[halo[1], x,y,z]::single[evendist]];

```
var new_data:array[Double,nx,ny,nz]::allocated[grid[x,y,z]::single[evendist]];
zeroGrid(data);
var norm_b:=fillBoundaryConditions(data);
for i from 0 to maxIters {
  norm_r:=computeResidue(data);
  norm_r:=norm_r / norm_b;
  if (norm_r < threshold) break;
  for i from data[pid()].low to data[pid()].high {
    for j from data[pid()][i].low to data[pid()][i].high {
      for k from data[pid()][i][j].low to data[pid()][i][j].high {
        new_data[i][j][k]:=(data[i+1][j][k]+data[i-1][j][k]+data[i][j+1][k]+data[i][j-1][k]+data[i][j][k+1]+data[i][j][k-1]) * 1/6;
      };
    };
  };
  data:=new_data;
  sync data;
}
```

Asynchronous

var data:array[Double,nx,ny,nz]::allocated[grid[halo[1]::async[2],x,y,z]::single[evendist]];

```
var new_data:array[Double,nx,ny,nz]::allocated[grid[x,y,z]::single[evendist]];
zeroGrid(data);
var norm_b:=fillBoundaryConditions(data);
for i from 0 to maxIters {
   norm_r:=computeResidue(data);
  norm_r:=norm_r / norm_b;
  if (norm_r < threshold) break;
  for i from data[pid()].low to data[pid()].high {
    for j from data[pid()][i].low to data[pid()][i].high {
      for k from data[pid()][i][j].low to data[pid()][i][j].high {
        new_data[i][j][k]:=(data[i+1][j][k]+data[i-1][j][k]+data[i][j+1][k]+data[i][j-1][k]+data[i][j][k+1]+data[i][j][k-1]) * 1/6;
      };
    };
  };
   data:=new_data;
  sync data;
}
```

Racy asynchronous

var data:array[Double,nx,ny,nz]::allocated[grid[halo[1]::async::racy, x,y,z]::single[evendist]];

```
var new_data:array[Double,nx,ny,nz]::allocated[grid[x,y,z]::single[evendist]];
zeroGrid(data);
var norm_b:=fillBoundaryConditions(data);
for i from 0 to maxIters {
   norm_r:=computeResidue(data);
   norm_r:=norm_r / norm_b;
   if (norm_r < threshold) break;
   for i from data[pid()].low to data[pid()].high {
      for j from data[pid()][i].low to data[pid()][i].high {
         for k from data[pid()][i][j].low to data[pid()][i][j].high {
            new_data[i][j][k]:=(data[i+1][j][k]+data[i-1][j][k]+data[i][j+1][k]+data[i][j-1][k]+data[i][j][k+1]+data[i][j][k-1]) * 1/6;
         };
      };
   };
   data:=new_data;
   sync data;
}
```

Performance results

Cores	Version	Runtime F90+MPI (s)	Runtime Mesham (s)
512	Sync	132.1	150.5
	Async (100)	146.9	145.5
	Async racy	126.4	125.7
2048	Sync	159.4	208.9
	Async (100)	184.9	188.4
	Async racy	163.8	163.5
8192	Sync	247.1	298.7
	Async (100)	272.5	271.8
	Async racy	264.9	265.0

Local problem size: 50x50x50. Weak scaling. Running on Cray XE6

Jacobi with blocks

- Point wise Jacobi's algorithm is very slow to converge
- If the linear system is rewritten in terms of blocks X_i each consisting of several individual elements x_i with the matrix split as shown:

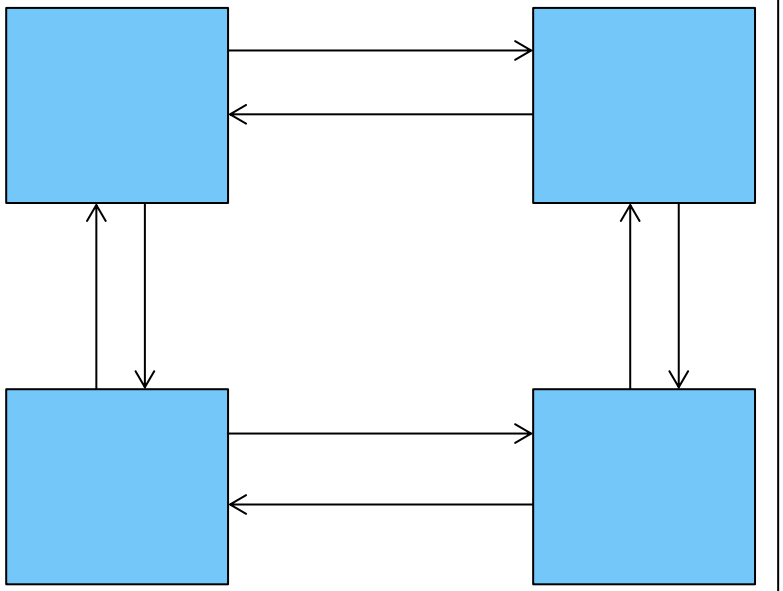
$$\begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ B_n \end{pmatrix}$$

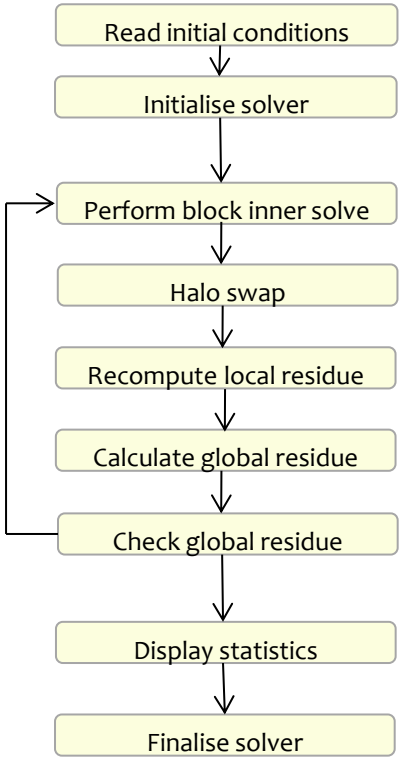
- Then we can also rewrite Jacobi's iteration in terms of blocks:

$$X_i^{(k)} = A_{ii}^{-1}\left(B_i - \sum_{j \neq i} A_{ij} X_j^{(k-1)} \right)$$

Jacobi with blocks

- Assuming domain decomposition and local operators, this requires only halo swaps between neighbouring domains
- The *inner solve* of the block equation is then fully local

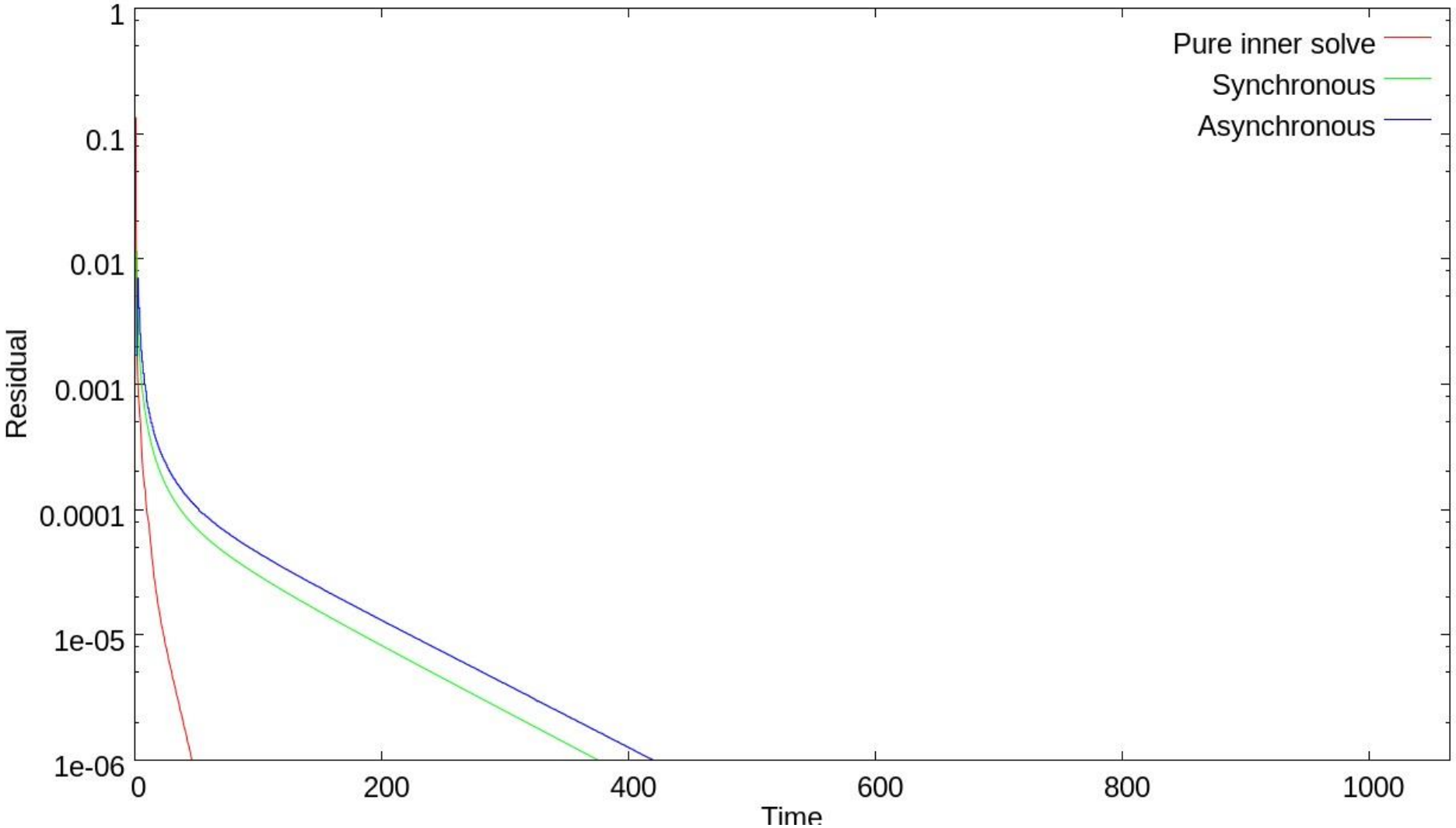




Asynchronous Communication

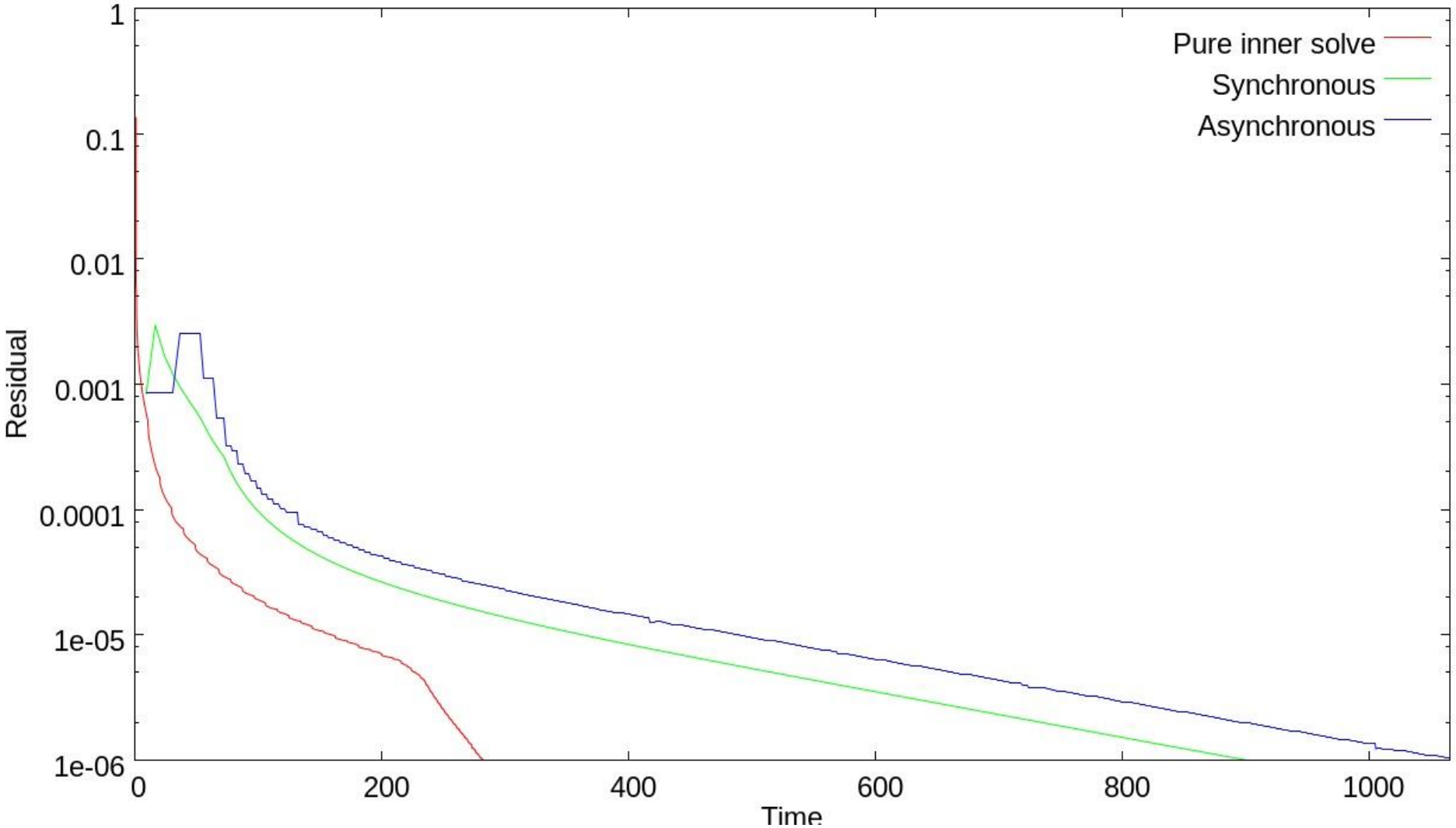
- Existing synchronous parallel iterative solvers exist and scale reasonably well (e.g. PETSc)
- Leverage this for the inner solver – each block is solved by a group of multiple processes in parallel (synchronously)
- Introduce asynchrony in halo swap between blocks:
  - Happens at the processor level between blocks
  - If new halo data has arrived from neighbouring block copy it into local work array
  - If no new data, start a new inner solve with existing halo data (from a previous iteration)
- Residual checking
  - Asynchronous reduction to determine residual
  - Termination criteria checked against the latest value

1024 cores



Problem size:50x50x50 per core, weak scaling. Running on HECToR (Cray XE6)

16384 cores



Problem size:50x50x50 per core, weak scaling. Running on HECToR (Cray XE6)

Results summary

Cores	Group Sync	Group Async
1024	8.33	9.33
4096	4.25	5.10
16384	3.19	3.78

Ratio of execution time to pure GMRES for specific core count

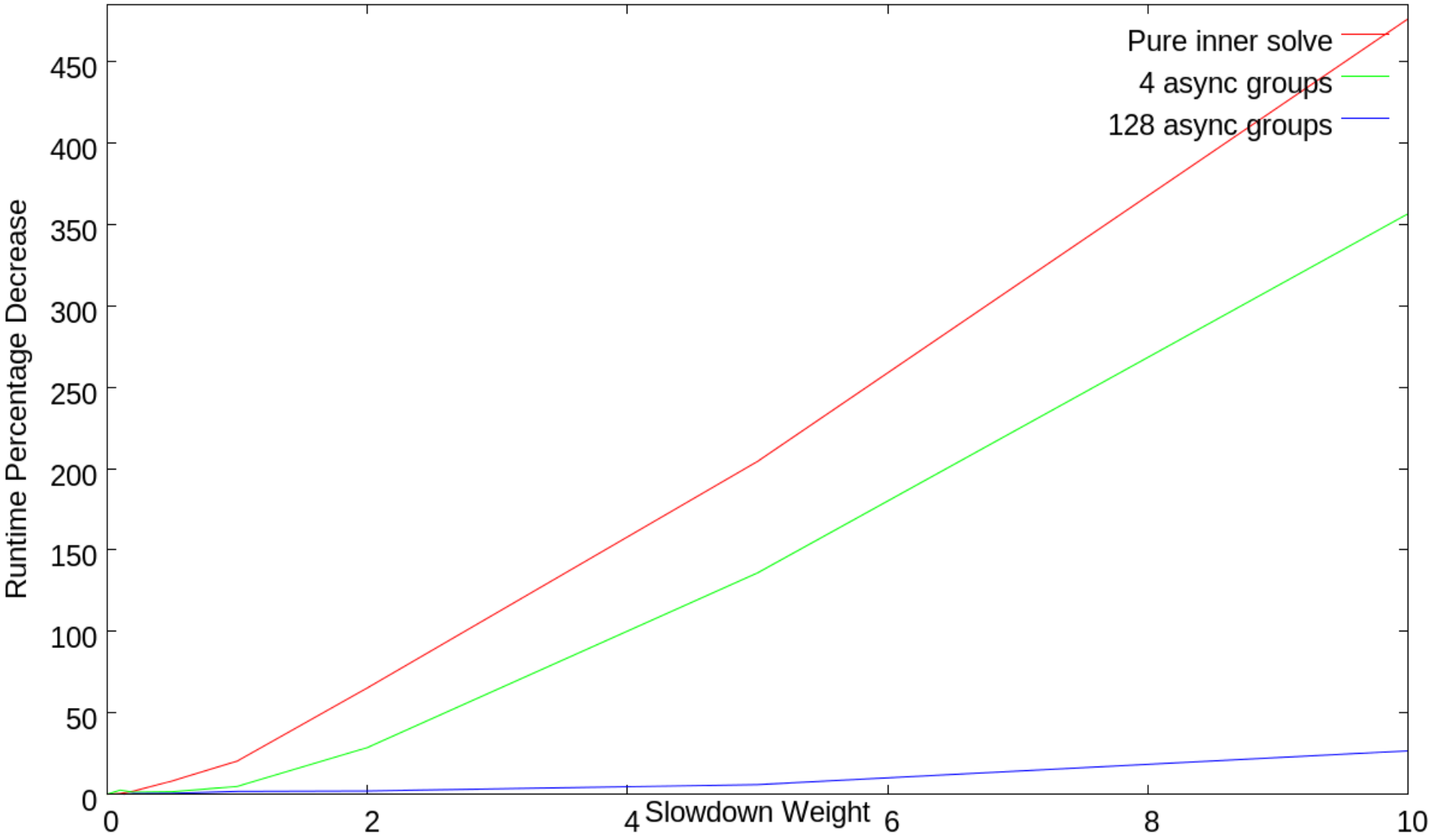
Cores	Internal	Group Sync	Group Async
1024	1	1	1
4096	2.88	1.47	1.57
16384	6.24	2.39	2.55

Ratio of execution time to 1024 cores for each version

Cores	Internal	Group Sync
1024	19.20	26.11
4096	16.57	20.72
16384	10.05	19.41

Iteration rate (iterations per second)

Slowdown resilience



Problem size:50x50x50 per core, 1024 cores. Running on HECToR (Cray XE6)

Conclusions

- Demonstrated how types can be used to provide for varying degrees of control
- At no performance penalty* we can gain improvements in programmability by using types
- Can retrofit type approach to existing languages

- We have extended existing synchronous solvers using an asynchronous Block Jacobi Approach
- Performance and scalability of the block method starts to looks favourable with large core counts and problem size
- In agreement with previous Jacobi iteration work, synchronous communication is still favourable to asynchronous at the core counts tested.