Grid status update

- **Peter Boyle,**
- **Guido Cossu,**
- **Antonin Portelli,**
- **Azusa Yamaguchi**
- **+ contributors (Richtman, O'Haigan, Kettle, Erben, Marshall, Jung, Lehner)**
- **+ Gianluca Filaci - starting to help on gpu port**

- **Introduction to Grid**
- **GPU-port status**
- **Physics functionality**
- **Outlook**

www.github.com/paboyle/Grid

Grid Documentation

Guido Cossu, Antonin Portelli, Azusa Yamaguchi

Sep 25, 2018

CONTENTS:

1 Preliminaries **1**
- 1.1 Who will use this library . . . 1
- 1.2 Data parallel interface . . . 2
- 1.3 Internal development . . . 2

2 Reporting Bugs **3**

3 Download, installation and build **4**
- 3.1 Required libraries . . . 4
- 3.2 Compilers . . . 4
- 3.3 Quick start . . . 4
- 3.4 Build setup for Intel Knights Landing platform . . . 7
- 3.5 Build setup for Intel Haswell Xeon platform . . . 8
- 3.6 Build setup for Intel Skylake Xeon platform . . . 8
- 3.7 Build setup for AMD EPYC / RYZEN . . . 9

4 Execution model **11**
- 4.1 Accelerator memory model . . . 11

5 Data parallel API **12**
- 5.1 Tensor classes . . . 12
- 5.2 Vectorisation . . . 19
- 5.3 Coordinates . . . 19
- 5.4 Grids . . . 20
- 5.5 Lattice containers . . . 22
- 5.6 Data parallel expression template engine . . . 25
- 5.7 Site local fused operations . . . 27
- 5.8 Inter-grid transfer operations . . . 29

6 Random number generators **32**

7 Input output facilities **34**
- 7.1 Serialisation . . . 34
- 7.2 Data parallel field IO . . . 35

8 Linear operators **40**
- 8.1 Linear Operators . . . 41
- 8.2 Red Black . . . 42

9 Operator Functions **44**

10 Algorithms **45**
- 10.1 Approximation . . . 45
- 10.2 Iterative solvers and algorithms . . . 46

11 Lattice Gauge theory utilities **50**
- 11.1 Spin . . . 50
- 11.2 SU(N) . . . 52
- 11.3 Space time grids . . . 54

12 Lattice actions **55**
- 12.1 Wilson loops . . . 55
- 12.2 Gauge Actions . . . 57
- 12.3 Fermion . . . 57
- 12.4 Pseudofermion . . . 61

13 HMC **63**

14 Development of the internals **66**
- 14.1 Simd classes . . . 66
- 14.2 Communications facilities . . . 66
- 14.3 Cartesian Grid facilities and field layout . . . 66
- 14.4 Stencil construction . . . 66
- 14.5 Optimised fermion operators . . . 66
- 14.6 Optimised communications . . . 66

15 Interfacing with external software **67**
- 15.1 MPI initialization . . . 67
- 15.2 Grid Initialization . . . 67
- 15.3 MPI coordination . . . 68
- 15.4 Mapping fields between Grid and user layouts . . . 68

- Grid is a C++11 high level data parallel interface for cartesian Grid problems
  - The whole machine is programmed lockstep, similar to QDP++.

```
std::vector<int> dims({64,64,64,128});
```
← Lattice dims

```
std::vector<int> simd
    = GridDefaultSimd(Nd,vComplex::Nsimd());
```
← Architecture dependent use of SIMD lanes

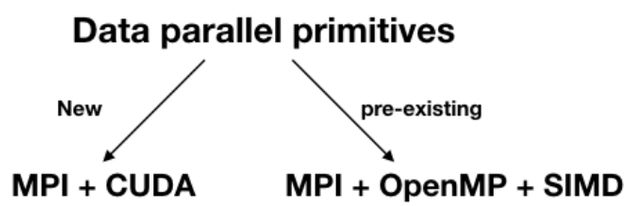
```
std::vector<int> mpi ({4,4,4,8});
```
← MPI decomposition

```
GridCartesian grid(dims, simd, mpi);
```
← Multiple grids supported

```
LatticeLorentzColourMatrix A(grid);
```
← Distributed array

```
A = 1.0; // Unit gauge
```
← 1.0 scalar replicated in each MPI process
A distributed across MPI processes
Threads, MPI, vectorisation handled under the hood

- It aims to give performance portability between many Exascale architectures
  - The API provides high level primitives which are broken down according to architecture.
  - Multicore performance and portability has been good for some time, but GPU support has been absent.

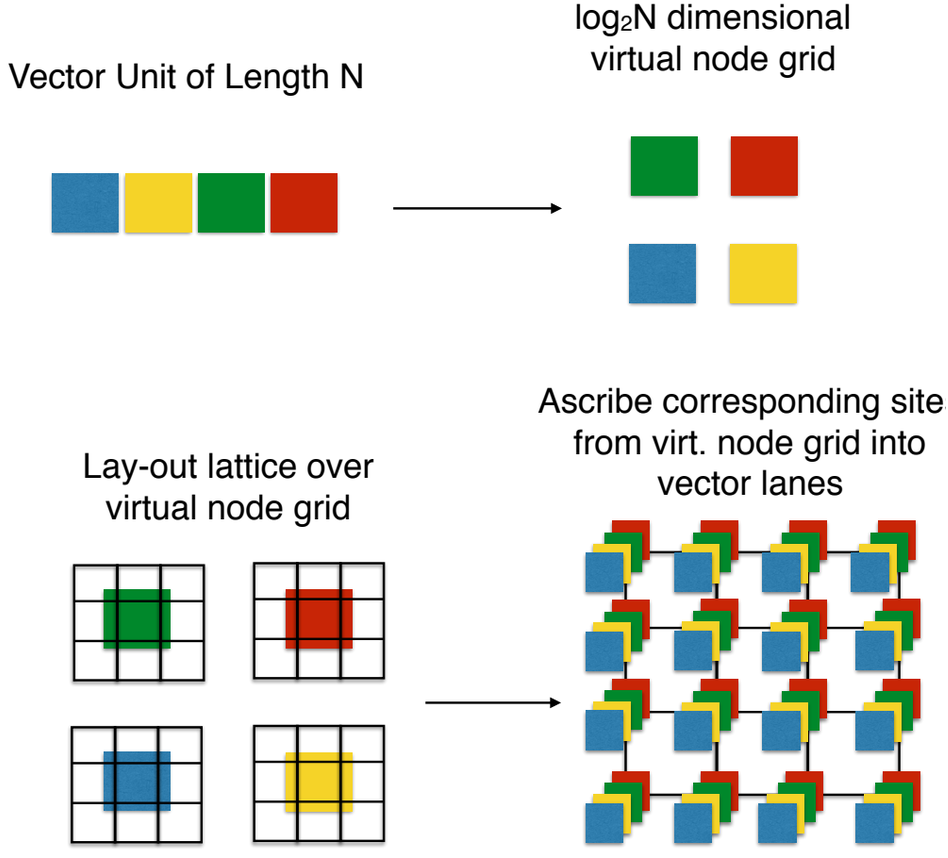


Suppresses SIMD lane crossing by surface to volume

Permute particularly trivial on GPU: (WilsonKernelsGpu.cc)

```
if (SE._is_local) {
  int mask = Nsimd >> (ptype + 1);
  int plane= SE._permute ? (lane ^ mask) : lane;
  auto in_l = extractLane(plane,in[SE._offset+s]);
  spProj(chi,in_l);

} else {
  chi  = extractLane(lane,buf[SE._offset+s]);
}
```



Tensor type system:

- Build scalar and struct-of-array (SoA) fields with Lorentz, Colour, Spin indices from template objects.

class iScalar<class Datum> { Datum internal; };
class iVector<class Datum, int N> { Datum internal[N]; };
class iMatrix<class Datum, int N > { Datum internal[N][N]; }

- Rules of matrix multiplication, tracing, inner products are recursively inferred for each index in a nested object.

	Scalar	Vector	Matrix
Scalar	Scalar	Vector	Matrix
Vector	Scalar	-	-
Matrix	Matrix	Vector	Matrix

```
template<class l,class r,int N> accelerator_inline
auto operator * (const iMatrix<l,N>& lhs,
                 const iMatrix<r,N>& rhs)
      -> iMatrix<decltype(lhs._internal[0][0]*rhs._internal[0][0]),N>
```

Macro offloaded function attributes

- Specific indices may be traced:

```
template<int Index,class vobj>
accelerator_inline auto traceSpin(const Lattice<vobj> &arg)
       -> Lattice<decltype(traceIndex<SpinIndex>(lhs._odata[0]))>
{
      return traceIndex<SpinIndex>(arg);
}
```

C++11 features decrease code size !
AND add flexibility

Return type inference saved 100k LOC c.f. QDP++, C++98 approach
And allows arbitrary tensor structures LatticeSpinColourSpinColourMatrix 4q operator

Implementation:

- inline an internal arithmetic interface to vectorisation:

  class Simd<scalar, vector> ; // vectors of Complex or Real;

  - The intrinsic vector operations are defined in a single location, and propagate globally through the code via inlining.
  - Partial SOA transformation used in opaque container Lattice classes to hide the data layout.

```
template<typename vtype> using iColourMatrix = iScalar<iScalar<iMatrix<vtype, Nc> > >;
typedef iColourMatrix<ComplexF  >         ColourMatrixF; // Scalar
typedef iColourMatrix<vComplexF>         vColourMatrixF; // SOA
typedef Lattice<vColourMatrixF>          LatticeColourMatrixF;
```

- Implementations are provided for each instruction set using inline intrinsics.

Develop branch:

- CPU targets: SSE, AVX, AVX2, AVX512, QPX, and ARM/Neon

The intrinsic vector operations are defined in a single location, and propagate globally through the code via inlining.

Grid single node performance

Architecture	Cores	GF/s (Ls x Dw)	peak
Intel Knight's Landing 7250	68	770	6100
Intel Knight's Corner	60	270	2400
Intel Skylakex2	48	1200	9200
Intel Broadwellx2	36	800	2700
Intel Haswellx2	32	640	2400
Intel Ivybridgex2	24	270	920
AMD EPYCx2	64	590	3276
AMD Interlagosx4	32 (16)	80	628

- Dropped to inline assembly for key kernel in KNL and BlueGene/Q
- EPYC is MCM; ran 4 MPI ranks per socket, one rank per die
- Also: ARM Neon and ARM SVE port

Common source accelerator port is functional but under tuning.

- Simple data parallel code saturates memory bandwidth on Volta
- Assume Unified Virtual Memory (not restriction to Nvidia as Intel and AMD GPU's support under OpenCL/Linux)
- CUDA; considering OpenMP 5.0 and SyCL for broader accelerator portability

feature/gpu-port branch

In GPU targets the Lattice containers use a Unified Virtual Memory (UVM) allocator.

```
template<class T> using Vector     = std::vector<T,alignedAllocator<T> >;

Vector<float> ArrayThatUsesUVM(size);
```

Dimensions of all field types are template parameters known at compile time

- compile time unrolling on planned accelerator architectures.

All offloadable functions are marked with an *accelerator* function attribute

- (macro, translates to __host__ __device__ under CUDA compilation).
    - Presently an assumption that A21 environment can match this

- GPU targets: use vectors of float2/float4, double2 generating 128 bit load instructions on each thread.
- Vectors of these *wide* loads are arranged to coalesce over consecutive threads in Wilson kernel.
    - Do *not* have to store as RRRR,IIII but RI,RI,RI,RI complex accesses are fine
    - Write in C++, read the generated assembly useful performance optimisation model

- work with double2, float2/4 implementation of complex class.
- Playing with two detailed implementations (—enable-simd=GPU, —enable-simd=VGPU)
    - Likely VGPU will “win”

```
template<class pair> class GpuComplex { pair z; };
typedef GpuComplex<float2 > GpuComplexF;

constexpr int NSIMD_ComplexF = COALESCE_GRANULARITY / sizeof(float2);
template<int _N, class _datum> struct GpuVector { _datum v[_N]; };
typedef GpuVector<NSIMD_ComplexF, GpuComplexF > GpuVectorCF;
```

Expression template (ET) engine:

High level data parallel expressions transformed to sitewise operations using a compact (400 LoC) C++11 ET engine.

- Port to GPU significant element of ECP Grid activity (proceedings, Boyle, Clark, DeTar, Lin, Rana, Vaquero Avilés-Casco).

Flat objects representing the abstract syntax tree AST are created containing all information required to assemble operations and operands.
This was the essential modification for successfully passing an expression object by value to device lambda's.
The only pointers are to UVM data arrays.

```
template <typename Op, typename T1,typename T2>
inline Lattice<vobj> &
// This is the closure of an expression, assignment operator with root of AST
operator=(const LatticeBinaryExpression<Op,T1,T2> &expr)
{
    auto me  = View(); // View object can be copied by value to Device
    accelerator_loop(ss,me,{
      me[ss]=eval(ss,expr);   // Loop Body captured as device lambda function
    });
   return *this;
}
```

Implementation details

The accelerator_loop macro creates a device lambda and a CUDA kernel call on GPU targets.
It creates an OpenMP loop on multicore targets.

An expression object **automatically** maps type of leaves of tree from a host Lattice object to a device view Lattice object.
The expression object can be safely copied by value to the device:

```
template <typename Op, typename _T1, typename _T2>
class LatticeBinaryExpression : public LatticeExpressionBase
{
  typedef typename ViewMap<_T1>::Type T1;
  typedef typename ViewMap<_T2>::Type T2;
  Op op;    // Op, arg1, arg2 are concatenated in this object
  T1 arg1;  // Lattice objects are mapped to LatticeView objects
  T2 arg2;  // Other objects contained by value in this struct.

  LatticeBinaryExpression(Op _op, _T1 &_arg1,const _T2 &_arg2) :
     op(_op), arg1(_arg1), arg2(_arg2) {};
};
```

Expression Template goodness:

```
LatticeColourMatrix x(&Grid); random(pRNG,x);
LatticeColourMatrix y(&Grid); random(pRNG,y);

double start=usecond();
for(int64_t i=0;i<Nloop;i++){
    x=x*y;
}
double stop=usecond();
```

```
==================================================
= Benchmarking SU3xSU3 x= x*y
==================================================
 L            bytes                GB/s       GFlop/s
--------------------------------------------------
 8            5.9e+05              31.2       28.6
16            9.44e+06             305        279
24            4.78e+07             563        516
32            1.51e+08             704        645
40            3.69e+08             764        700
48            7.64e+08             793        727
==================================================
```

Current badness:

```
LatticeColourMatrix x(&Grid); random(pRNG,x);
LatticeColourMatrix y(&Grid); random(pRNG,y);

double start=usecond();
for(int64_t i=0;i<Nloop;i++){
    z=x*Cshift(y,mu,1);
}
double stop=usecond();
```

~ few GB/s

Why? Cshift is not yet offloaded, and arrays ping pong between host/device

Entire code base compiles through NVCC (100k+ LOC) in CUDA 9.2 with g++ == clang++

- Presently experimental, not pretty and success is CUDA version and compiler version dependent.

```
export MPICXX=mpiicpc
export CXXFLAGS="-arch=sm_70"
export CXX=nvcc
../configure\
 --enable-precision=single \
 --enable-simd=VGPU \
 --enable-comms=mpi3-auto \
 --enable-gen-simd-width=32
```

Configure script picks up some wacky compiler flags:

```
nvcc -x cu -Xcompiler -fno-strict-aliasing -Xcompiler -Wno-unusable-partial-specialization --expt-extended-lambda --expt-relaxed-constexpr
```

Runs correctly on Summit/V100 (IBM AC922) and Tesseract/V100 (HPE 8600)

High-performance kernels have been written to offload Dslash and read coalesce optimisations made.

DWF Dslash kernel ran with 90% of QUDA performance on single Volta.

CUDA got faster recently, and grid got a little slower recently so more work to do !

Volume	Grid	QUDA
12^4	1.4TF/s	1.7TF/s
24^4	1.9TF/s	2.3TF/s

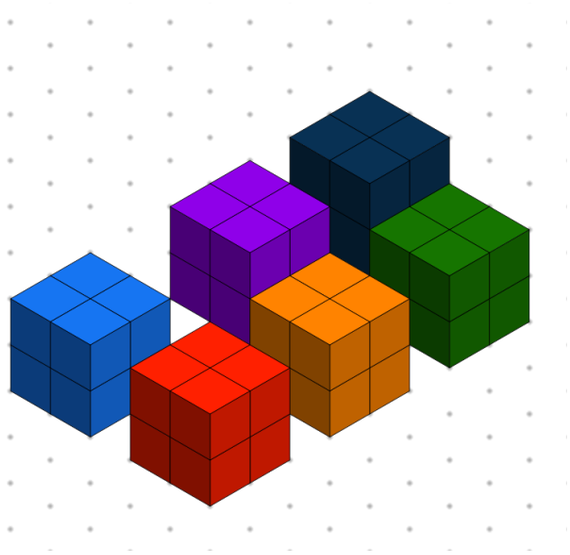
Exploiting locality: multiple chips/GPUs per node

Multi-socket servers: NUMA aware code

- Hybrid OpenMP + MPI
- Use 1:1 mapping between MPI ranks and sockets
- Unix shared memory between sockets (over UPI)
- Reserve MPI transfers for inter-node

Multi-GPU servers: NVlink aware code

- Use 1:1 mapping between MPI ranks and GPU's
- Use Cuda to open up direct GPU-GPU device memory access over NVlink
- Reserve MPI transfers for inter-node, direct to GPU if possible



Multi-GPU

```
#ifdef GRID_NVCC
void GlobalSharedMemory::SharedMemoryAllocate(uint64_t bytes, int flags)
{
  void * ShmCommBuf ;

  ////////////////////////////////////////////////////////////////////////////////////
  // allocate the pointer array for shared windows for our group
  ////////////////////////////////////////////////////////////////////////////////////
  MPI_Barrier(WorldShmComm);
  WorldShmCommBufs.resize(WorldShmSize);

  ////////////////////////////////////////////////////////////////////////////////////
  // TODO/FIXME : NOT ALL NVLINK BOARDS have full Peer to peer connectivity.
  // The annoyance is that they have partial peer 2 peer.
  // This occurs on the 8 GPU blades e.g. DGX1, supermicro board,
  ////////////////////////////////////////////////////////////////////////////////////
  //  cudaDeviceGetP2PAttribute(&perfRank, cudaDevP2PAttrPerformanceRank, device1, device2);
  cudaSetDevice(WorldShmRank);

  ////////////////////////////////////////////////////////////////////////////////////
  // Each MPI rank should allocate device buffer
  ////////////////////////////////////////////////////////////////////////////////////
  auto err =  cudaMalloc(&ShmCommBuf, bytes);
  SharedMemoryZero(ShmCommBuf,bytes);

  ////////////////////////////////////////////////////////////////////////////////////
  // Loop over ranks/gpu's on our node
  ////////////////////////////////////////////////////////////////////////////////////
  for(int r=0;r<WorldShmSize;r++){

    ////////////////////////////////////////////////
    // If it is me, pass around the IPC access key
    ////////////////////////////////////////////////
    cudaIpcMemHandle_t handle;


    ////////////////////////////////////////////////
    // Share this IPC handle across the Shm Comm
    ////////////////////////////////////////////////
     int ierr=MPI_Bcast(&handle,
                        sizeof(handle),
                        MPI_BYTE,
                        r,
                        WorldShmComm);

    ////////////////////////////////////////////////////////////
    // If I am not the source, overwrite thisBuf with remote buffer
    ////////////////////////////////////////////////////////////
    void * thisBuf = ShmCommBuf;
    if ( r!=WorldShmRank ) {
      err = cudaIpcOpenMemHandle(&thisBuf,handle,cudaIpcMemLazyEnablePeerAccess);
   }
    ////////////////////////////////////////////////////////////
    // Save a copy of the device buffers
    ////////////////////////////////////////////////////////////
    WorldShmCommBufs[r] = thisBuf;
  }
}
#else
…
#endif
```

- Grid has used Unix shared memory between ranks in a node for some time
  - Enables multi-threaded intra-node comms
  - MPI is *single threaded*

- NVLINK peer to peer required no new routines
  - Just different ifdef implementation of same routine!

- Assume 1:1 mapping of MPI ranks to GPUs

- Under PBS need to arrange MPI rank - GPU mapping. Summit “different”.

Multi-GPU status

- Intra-GPU face gather and spin project is not yet read coalesced
  - Known SIGSEGV if it is a “SIMD” dir etc…
  - Scale to 3-4 TF/s on 4 GPU’s on Tesseract
  - NVLINK already only 20% overhead prior to optimisation

- Inter-node communication will be interesting

  - STFC Tesseract: ICE XA HPE 8600
    - 1372 nodes OPA + 2x Skylake Xeon
    - 8 nodes: 4x V100, 2x Skylake Xeons
    - 2.5PB Lustre + 10PB DMF tape store

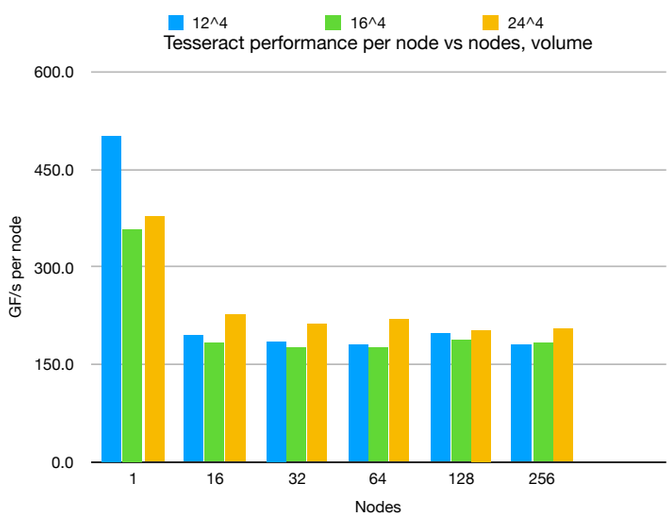
  - 4x 100Gbit/s OPA (on 2 PCIe Gen3 cards)
  - 100GB/s bidirectional OPA
    - 64GB/s bidirectional PCIe.

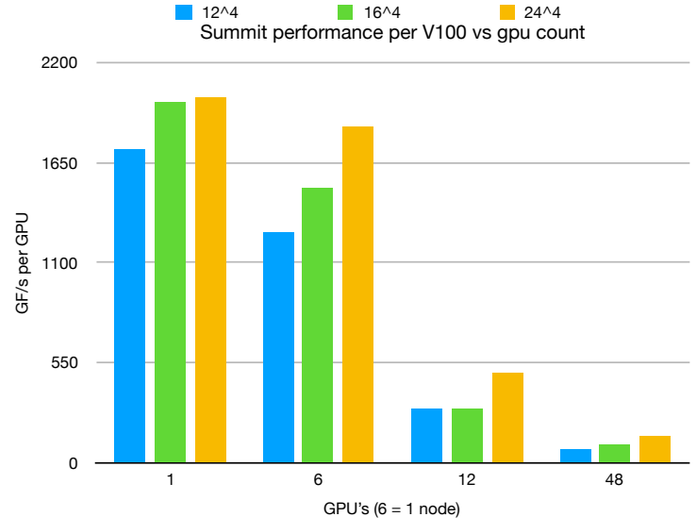
- Summit inter-node communication is crippling

  - Summit: 6 V100, 2x IBM Power 9

  - 2x 100Gbit/s EDR (on 2 PCIe cards)
  - 50GB/s bidirectional OPA

Caveat : results were Summer 2018, Summit GDR may improve

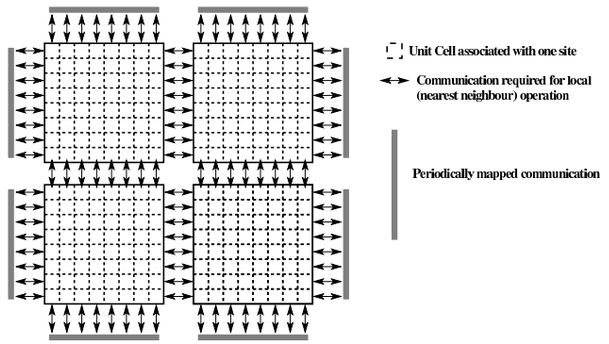




What are Nvidia going to do with sMellanox?
What is the Cray Perlmutter system going to scale?

QCD sparse matrix PDE solver communications

- L^4 local volume (space + time)
- finite difference operator 8 point stencil



- $\sim \frac{1}{L}$ of data references come from off node

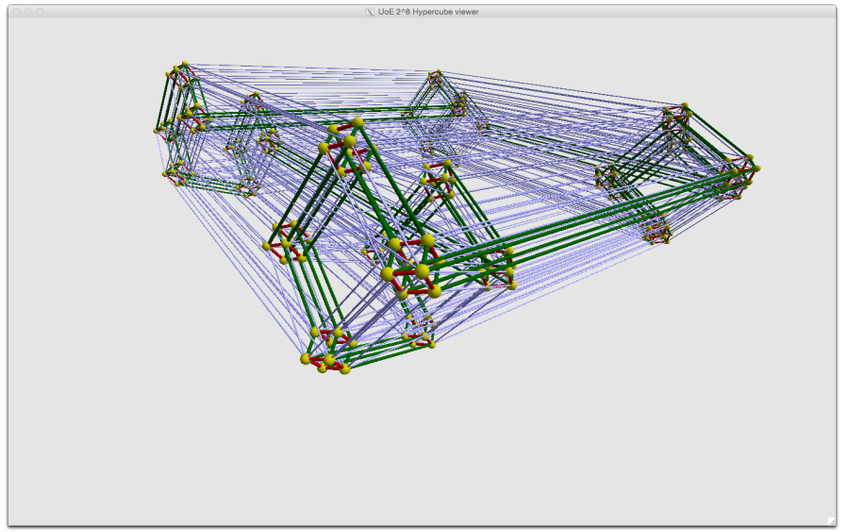
Scaling QCD sparse matrix requires interconnect bandwidth for halo exchange

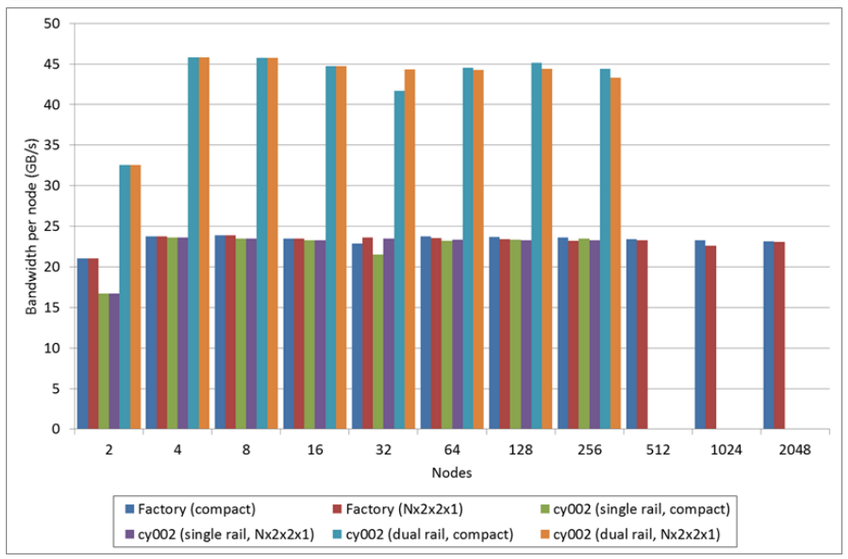
$$B_{network} \sim \frac{B_{memory}}{L} \times R$$

where R is the *reuse* factor obtained for the stencil in caches

- Aim: Distribute 100^4 datapoints over 10^4 nodes

Exploiting locality: hypercube network



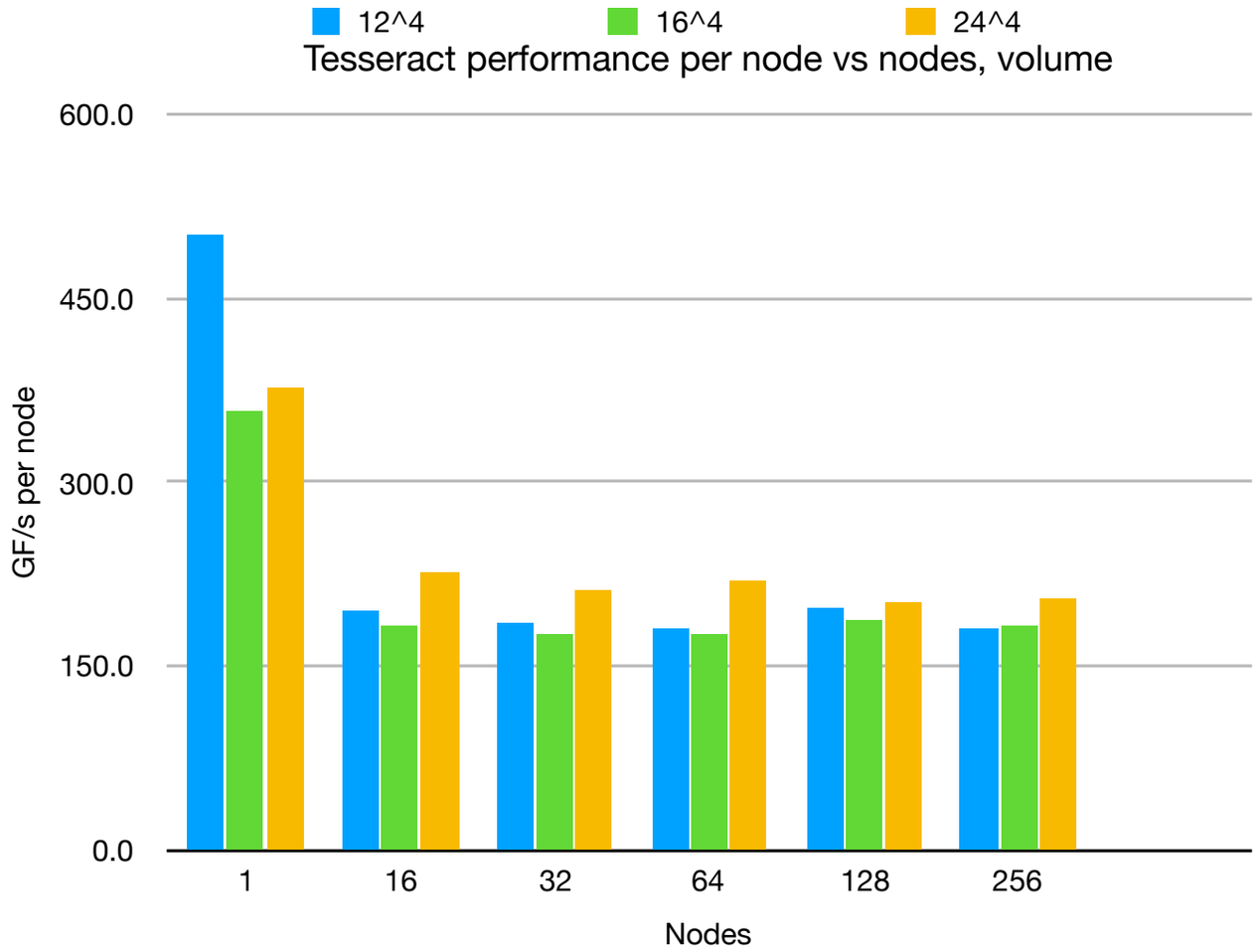


Improvement over Default Process Placement		
Nodes	**Decomp**	**Bandwidth**
2	2x1x1x1	0.98
4	2x2x1x1	1.00
8	2x2x2x1	1.00
16	4x2x2x1	1.27
32	4x4x2x1	1.70
64	4x4x4x1	2.00
128	8x4x4x1	2.09
256	8x8x4x1	2.60
512	8x8x8x1	3.30
1024	16x8x8x1	3.51
2048	16x16x8x1	3.84

- Small project with SGI/HPE on Mellanox EDR networks (James Southern)
- Embed 2^n QCD torus inside hypercube so that nearest neigbour comms travels single hop
  4x speed up over default MPI Cartesian communicators on large systems
  $\Rightarrow$ Customise HPE 8600 (SGI ICE-XA) to use $16 = 2^4$ nodes per leaf switch

DiRAC HPE ICE-XA hypercube network

- Edinburgh HPE 8600 system (Installed March 2018, 2 days ahead of schedule)
  - Low end Skylake Silver 4116, 12 core parts
  - Single rail Omnipath interconnect
  - Relatively cheap node: high node count and scalability
    - Improve price per core, bandwidth per core, reduce power



- 16 nodes (single switch) delivers bidirectional 25GB/s to every node (wirespeed)
- 512 nodes topology aware bidirectional 19GB/s
  - 76% wirespeed using every link in system concurrently

GPU status summary

- Data parallel site local expressions offload nicely
  - Cshift needs work
- 4D dslash kernels probably acceptable but could be optimised
  - Communications is the real battle ground
- Intra-node communications “under control”, only optimisation/tweaking
- Reductions still need further work
- Gianluca Filaci working on 5D Mobius matrices (Mee, Mee^-1)
  - 800GB/s on simple code for Mee
  - Fused Mee^-1 Meo kernels probably needed
- Plan to have full CG and then full Mobius EOFA HMC asap

This branch is 516 commits ahead, 94 commits behind develop.

= Danger

Back on the main branch !

- **Wilson Multigrid (Daniel Richtman, Regensburg, PB Edinburgh)**
  - Claim: ~2x faster than DDalphaAMG
    - Haven't verified because DDalphaAMG challenging to run (!)
    - Uses HDCR (Boyle/Yamaguchi) CoarsenedMatrix
      - Regensburg finished making this recursive
      - added many GMRES family solvers
      - Fixed the blockProject to be fast.

- **(R)HMC**
  - **Exact one flavour algorithm (David Murphy, Columbia)**
  - **Lanczos & block compressed Lanczosd**
    - **(PB, Chulwoo Jung, Christoph Lehner BNL)**
  - **All-to-all (Fionn O'Haigan, Portelli, PB, Edinburgh)**
  - **Distillation (Felix Erben, Michael Marshall, Edinburgh)**
  - **NPR (Kettle, PB, Edinburgh)**
    - **Bilinears and Delta_F=2**
    - **Twisted BC, non-exception SMOM**

Actions

HMC enabled

- Plaquette, Rectangle
- Gparity BC's.
- Wilson, Clover, RHQ, anisotropic
- Mobius, DWF, PartialFraction, ContinuedFraction, Cayley
  - Tanh, Zolotarev and other sign function approximations
- QED, QCD, arbitrary SU(n), U(1)
- Different fermion reps :
  - Fundamental, Adjoint, TwoIndexSymmetric, TwoIndexAntisymmetric

Valence sector

- Improved staggered

Eliminating communication (valence)

- **Split Grids : subdivide machine for many solves**

```
GridCartesian(const std::vector<int> &dimensions,
              const std::vector<int> &simd_layout,
              const std::vector<int> &processor_grid,
              const GridCartesian &parent,int &split_rank)
        : GridBase(processor_grid,parent,split_rank)
{
     Init(dimensions,simd_layout,processor_grid);
}
```

- **NB: Not the same as trivial parallelism:**
    - **Eigenvectors held globally**
    - **Deflate many RHS**
    - **Solves are local**
        - **Combine with BlockCG**

- **3x gain on Cori K-pipi jobs (for example)**

EOFA Mobius HMC moving to production

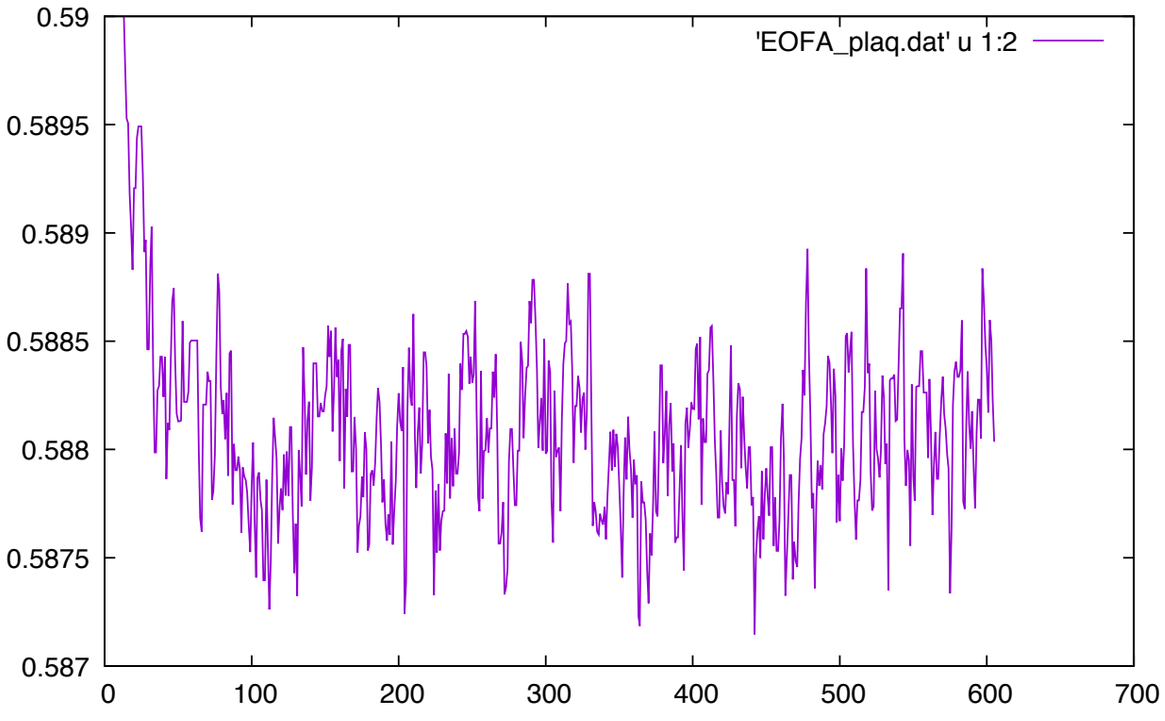
Verification: 16^3x32 Iwasaki+DWF 2+1f mu0.01, ms0.04

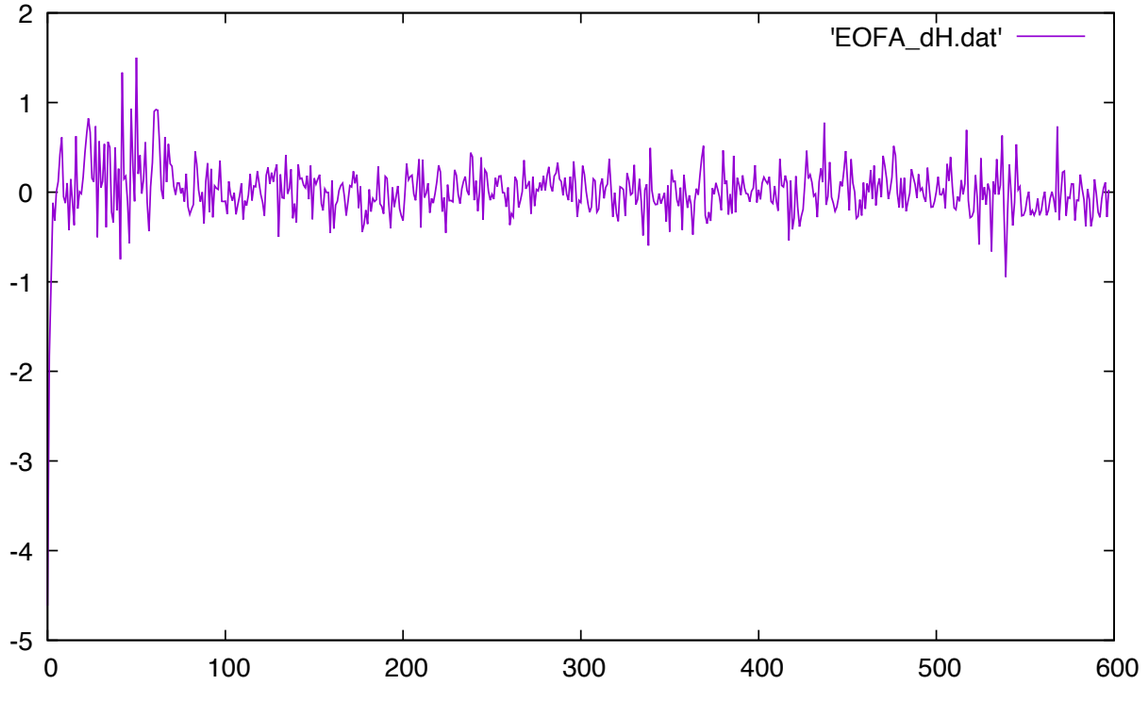
- Both RHMC and EOFA have same plaquette as published 0.01/0.04 16^3 runs. ~0.5880

- Initial untuned RHMC + Grid : 4.5h for 10 trajectories (~100 trajectories)
- Current tuned EOFA + Grid : 53m for 10 trajectories (~600 trajectories)

- Force gradient, multi-hasenbusch, EOFA, mixed precision solvers

	Heatbath	Action	Force
EOFA	double	mixed	mixed
2F	double	mixed	mixed





Started a 48^3 2.8 GeV evolution

- Will re-profile and tune as it evolves
- Once condition number justifies will resume looking at HDCR deflation in HMC

Merge develop with feature/gpu-port and get this running on Edinburgh GPU's.

Aurora

How future proof is all this?

- I've abstracted Nvidia specific hacks like "__host__ __device__"
  - accelerator function attributes
  - accelerator_loop(iteration, { loop body }) macros and lambda capture
    - Common with Kokkos and RAJA

- All files are .cc files, and I tell nvcc that C++ is CUDA (-x cu)
  - There are NO cuda sources.

- Work with Intel & pathforward forward to *encourage*
  - Unified memory space (if not, software managed device cache required)
  - Attribute based function labelling
  - Lambda offload or similar
  - OpenMP 5 should inline to target loops ok.

- OpenMP target is not sufficient
  - Does not address layout of arrays, which is fixed by language. Read coalescing left to programmer
- I hope performance portability is possible, even on emerging architectures
  - I think Grid has gone in the right direction but it is a lot of work !
  -

Next steps

- **Complete CUDA work (Summit, NESAP)**
- **SyCL and non-Nvidia GPUs (AMD, Intel)**

Questions?