Scaling Soft Matter Physics to a Thousand GPUs and Beyond

NVIDIA GTC Webinar, 3rd October 2012

Alan Gray, Kevin Stratford (EPCC, The University of Edinburgh)
Alistair Hart (Cray Exascale Research Initiative Europe)

CRAY THE SUPERCOMPUTER COMPANY | epcc

Acknowledgements

- This work was partly funded by the CRESTA Project
  - www.cresta-project.eu

CRAY THE SUPERCOMPUTER COMPANY | epcc |

Introduction

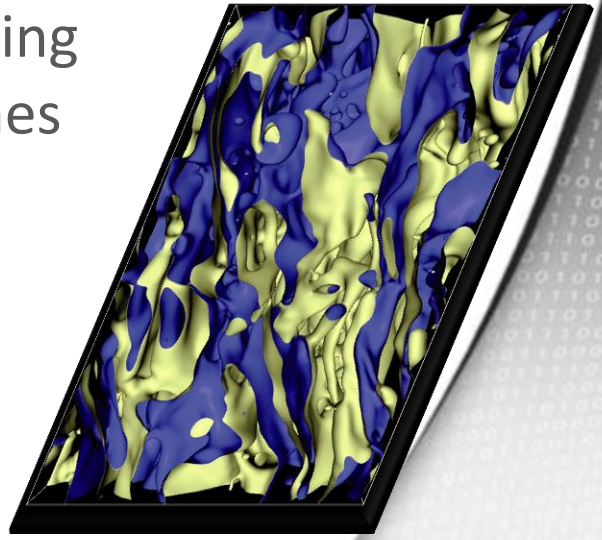
- Graphics Processing Unit (GPU) accelerated architectures are proliferating
  - power-performance advantages over traditional CPU based systems
  - in-depth development required to fully exploit for real applications
    - especially when scaling up to many nodes.
- This talk describes work performed to enable the *Ludwig* lattice Boltzmann fluid dynamics application to efficiently utilize massively parallel GPU accelerated systems

Outline

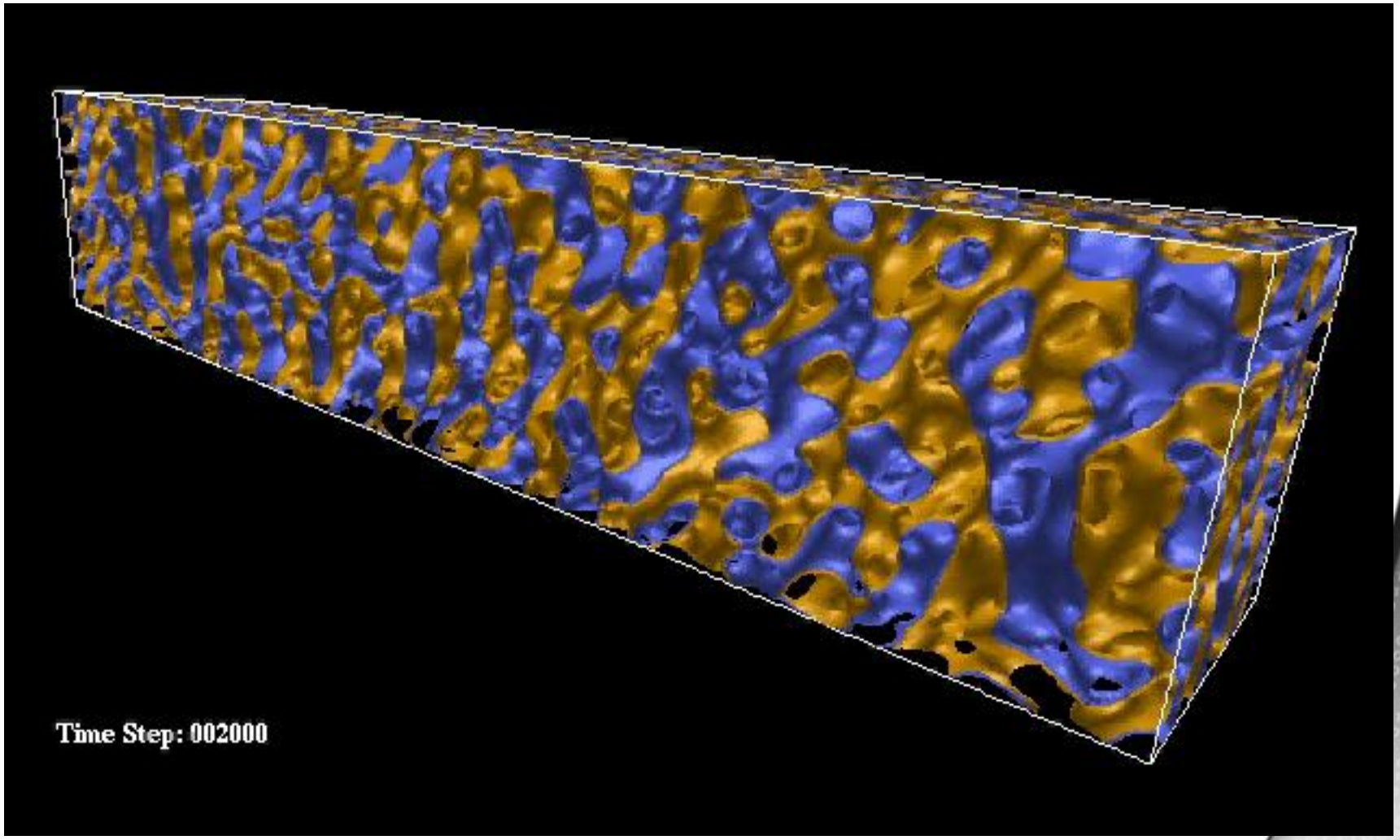
- Introduction to Ludwig fluid dynamics application
- Single-GPU implementation and optimization
- Multi-GPU implementation and optimization
- Performance results
- Current work: introduction of particles into the simulation

Ludwig

- Lattice Boltzmann method: popular approach for solving Navier-Stokes equations which govern fluid flow
  - Suitable for parallel implementations
- Ludwig: versatile LB package capable of simulating hydrodynamics of *complex* fluids
  - e.g. mixtures, surficants, liquid crystals, particle suspensions
    - cutting-edge research into condensed matter physics, including the search for new materials
- Original C/MPI Ludwig capable of exploiting large-scale traditional CPU based machines
  - good parallel scaling up to many thousands of cores.
  - can simulate large, complex systems



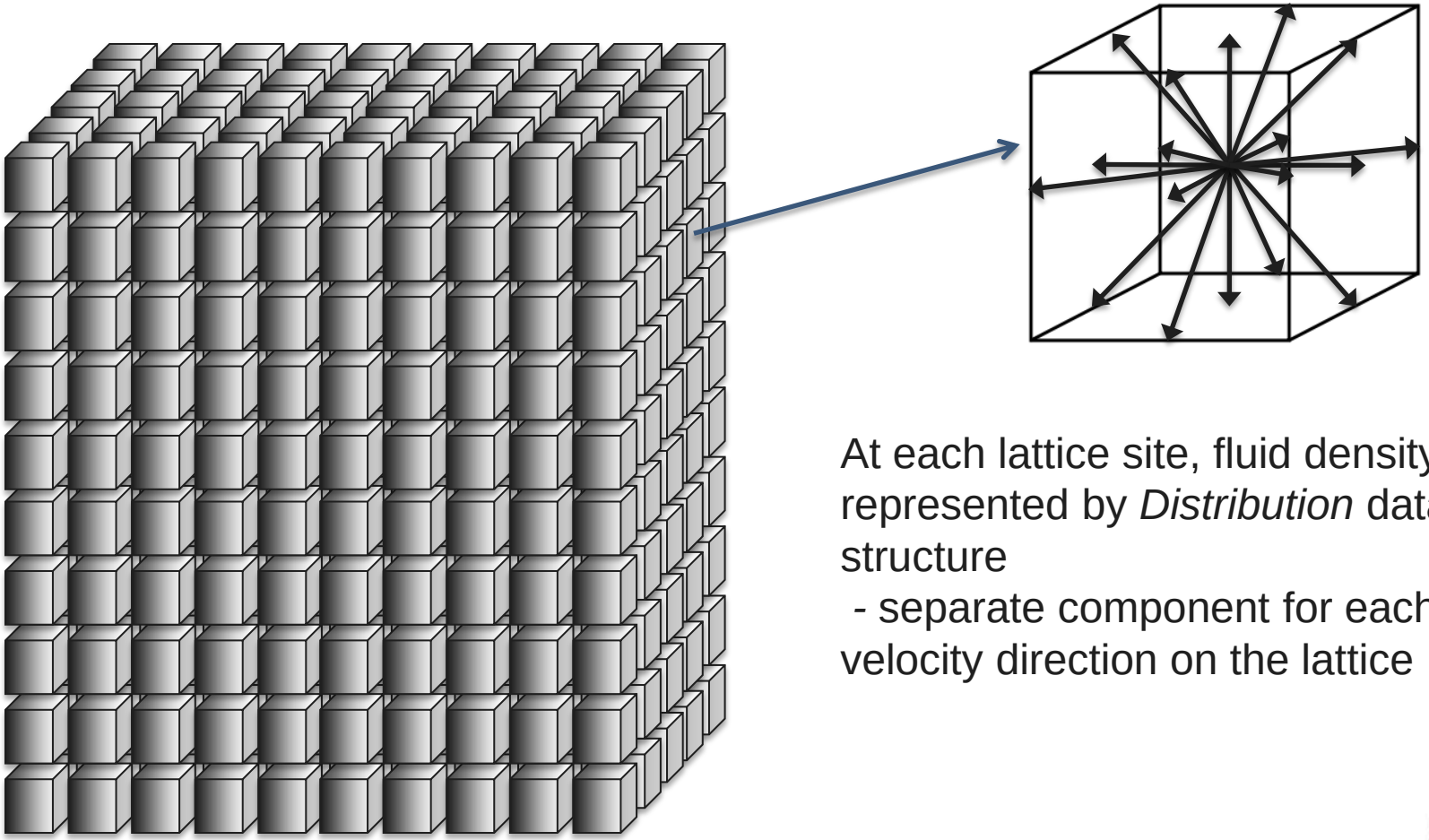
CRAY THE SUPERCOMPUTER COMPANY | epcc



CRAY THE SUPERCOMPUTER COMPANY | epcc |

Ludwig Lattice Boltzmann Model

- Fluid represented as “particles” of fluid density, moving and colliding on a lattice



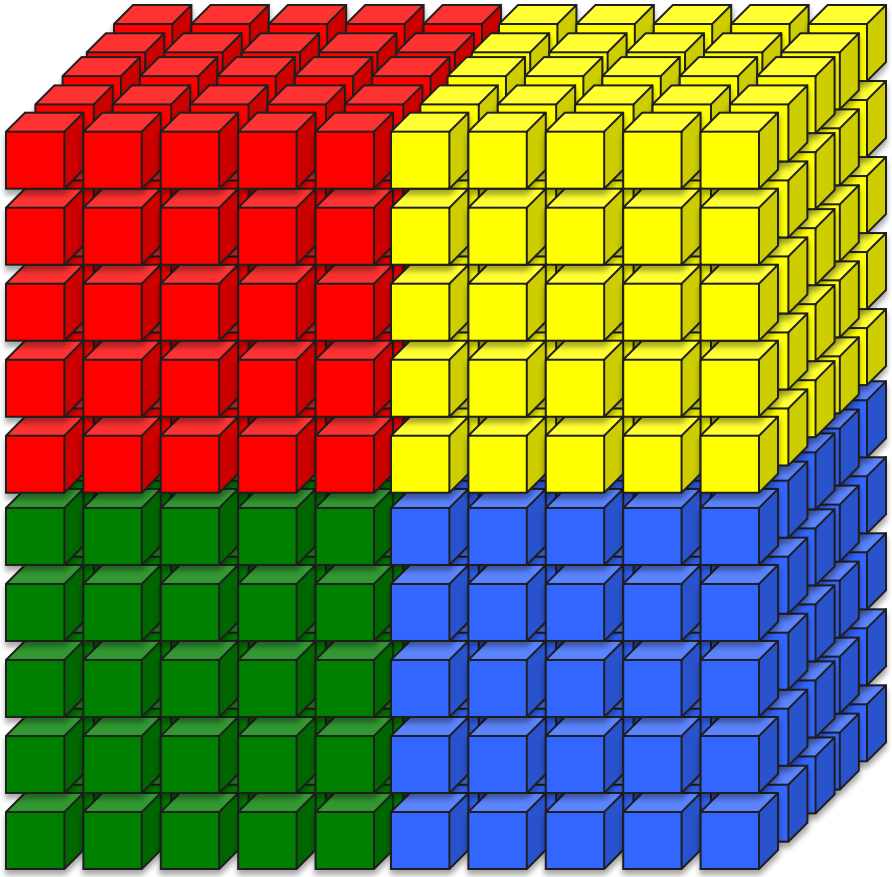
At each lattice site, fluid density represented by *Distribution* data structure

- separate component for each velocity direction on the lattice

CRAY THE SUPERCOMPUTER COMPANY | epcc

Ludwig Lattice Boltzmann Model

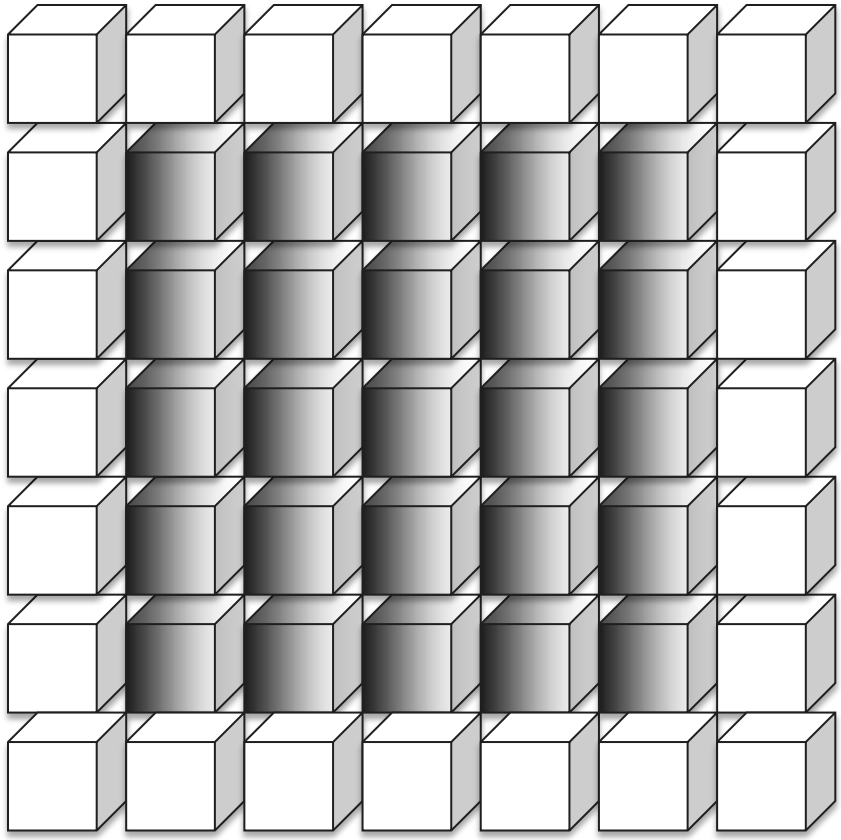
- When running in parallel, lattice is subdivided between MPI tasks



CRAY THE SUPERCOMPUTER COMPANY | epcc |

Ludwig Lattice Boltzmann Model

- For each subdomain, “Halo” sites added to hold copies of data from remote neighbours



Ludwig Algorithm

- Iterative updates to distribution function *f:*

$$f_i(\mathbf{r} + \mathbf{c}_i \Delta t; t) - f_i(\mathbf{r}; t) = -\mathcal{L}_{ij} f_j(\mathbf{r}; t).$$

- This corresponds to 2 stages:
  - RHS: collision stage: particles interact (collide)
    - Update distribution local to each lattice site
    - Computationally dominated by matrix-vector operations
    - Dominates simulation. Compute and memory bandwidth intensive
  - LHS: propagation stage: particles move according to their velocity
    - update distribution based on values at neighbouring lattice sites
    - Memory bandwidth bound
  - For parallel implementation, a communication stage is also required before propagation
    - halo swap of *f* (using MPI)

Single GPU Implementation

- All new GPU functionality implemented in additive manner
  - GPU acceleration optionally invoked at compile time
- GPU kernels and data management facilities implemented using CUDA
  - Each CUDA thread assigned to single lattice site
- Wrapper routines developed to specify decomposition, invoke kernels and manage data
  - With interfaces similar to original routines
- Important to offload all computational components in timestep, not just dominant collision stage
  - In order to be able to keep data resident on GPU
- Work was needed to allow use of encapsulated data in kernels

Single GPU Optimisation

- Matrix Vector operations:

Original

```
for (i = 0; i < 19; i++) {
  a[i] = 0.0;
  for (j = 0; j < 19; j++) {
    a[i] += f[is*NSITE+j]*L[i][j];
  }
}
```

Optimised

```
for (i = 0; i < 19; i++) {
  tmp = 0.0;
  for (j = 0; j < 19; j++) {
    tmp += f[j*19+is]*L[i][j];
  }
  a[i] = tmp;
}
```

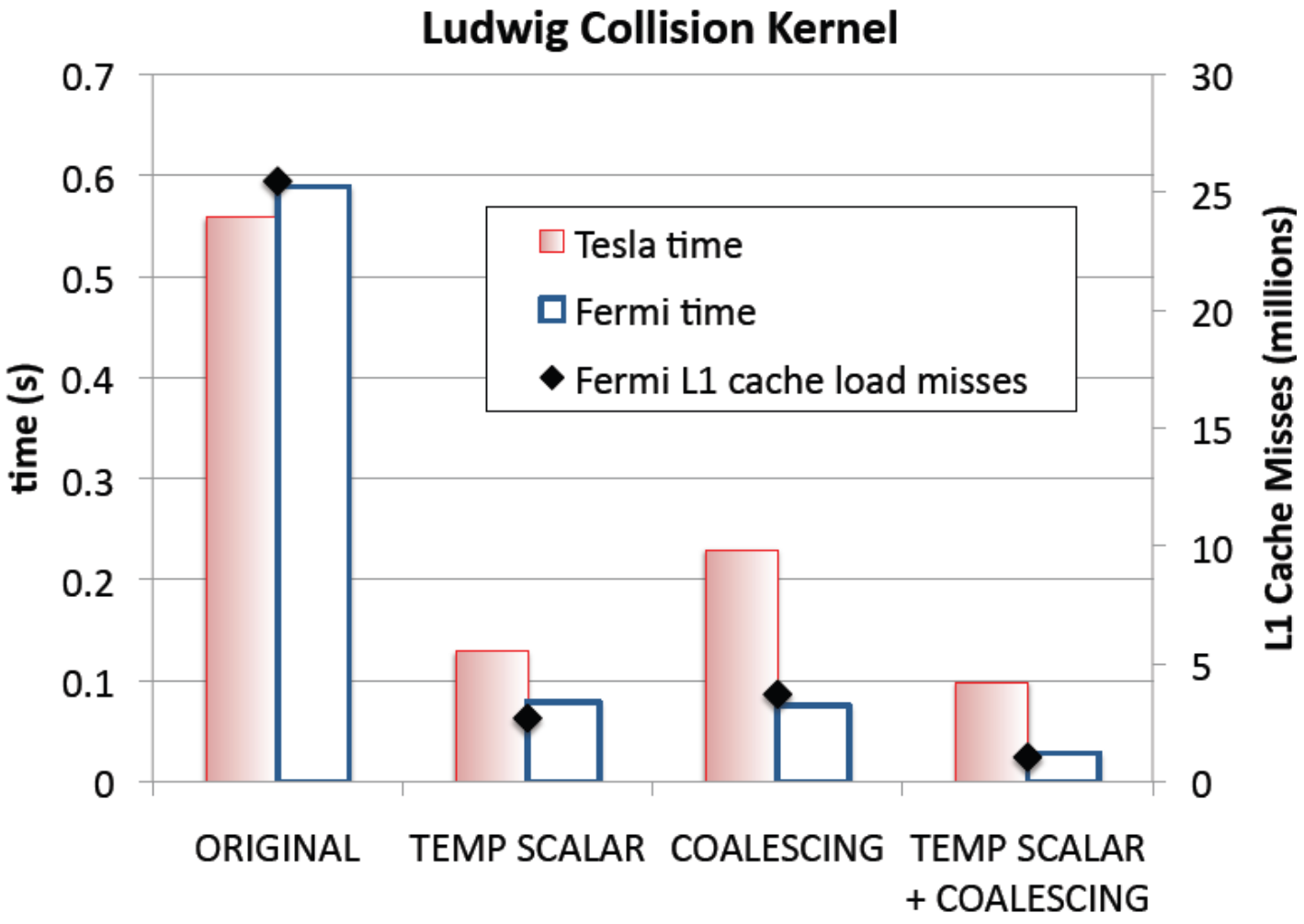
Temporary scalar

Reordered for coalescing

- Reordering of *f* allows coalesced memory accesses
  - Consecutive threads reading consecutive memory addresses
- Use of temporary scalar allows on-chip caching of intermediate summation values

CRAY THE SUPERCOMPUTER COMPANY | epcc

Single GPU Optimisation



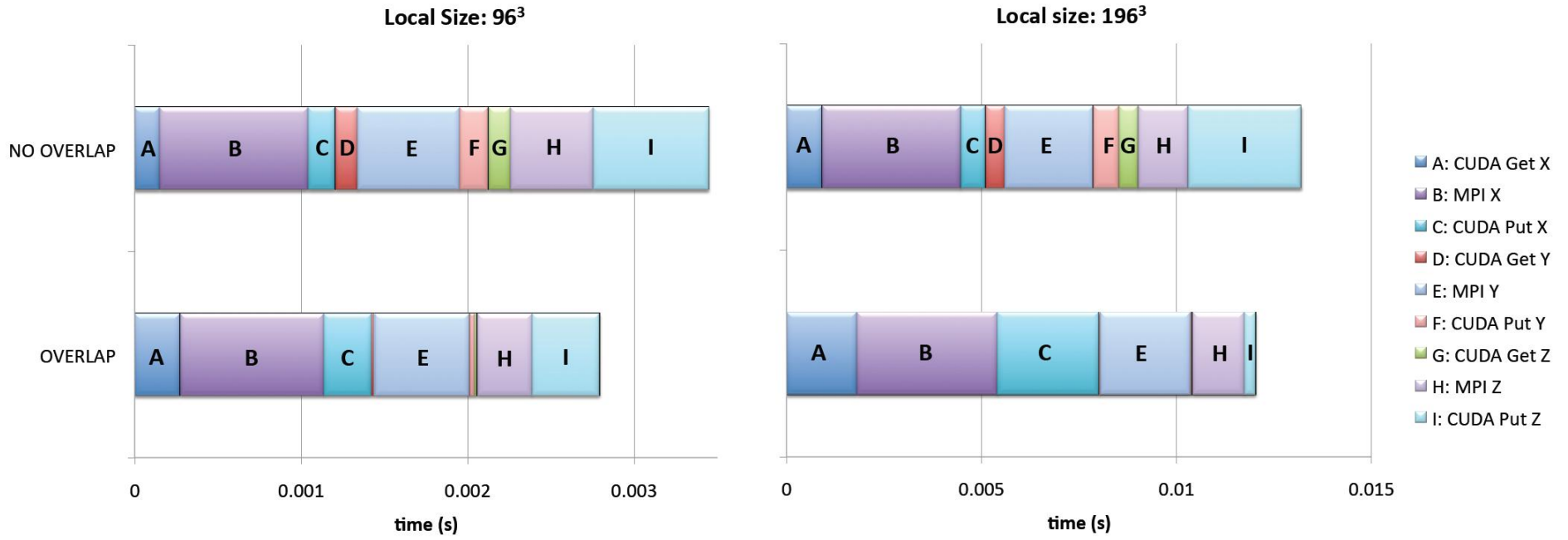
Multi-GPU Implementation

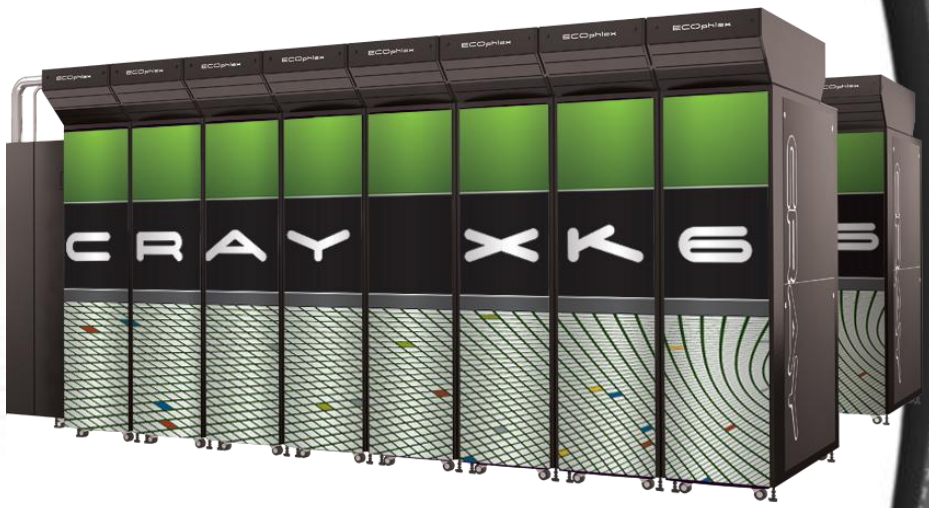
- Major redevelopment of communication phase was required to achieve good inter-GPU communication performance
  - and allow scaling to many GPUs.
- During each timestep, distribution halo cells (six 2D planes) must be transferred between MPI processes.
- Original code performs halo swaps “in-place”
  - Uses MPI datatypes to specify the planes of the sub-lattice to be sent/received in each direction
- GPU version needs halos transferred between GPUs
  - must be staged through host CPUs
  - Explicit buffering must be performed in application
    - No MPI datatype functionality available on GPU

Multi-GPU Implementation

- For each halo plane:
  - buffer explicitly packed on GPU (CUDA kernels)
  - buffer copied from GPU to host CPU (CUDA memory copies)
  - buffers exchanged between hosts (MPI)
  - buffer copied from host CPU to GPU (CUDA memory copies)
  - buffer unpacked on GPU (CUDA kernels)
- Comms reduced by factor of 4 by only sending velocity components propagating outward from a local domain
  - On GPU this filtering coded explicitly, since MPI Datatype functionality unavailable
    - Special care needed for corner sites which propagate in more than one direction

- Asynchronous CUDA functionality (Streams) used to overlap different communication operations where possible



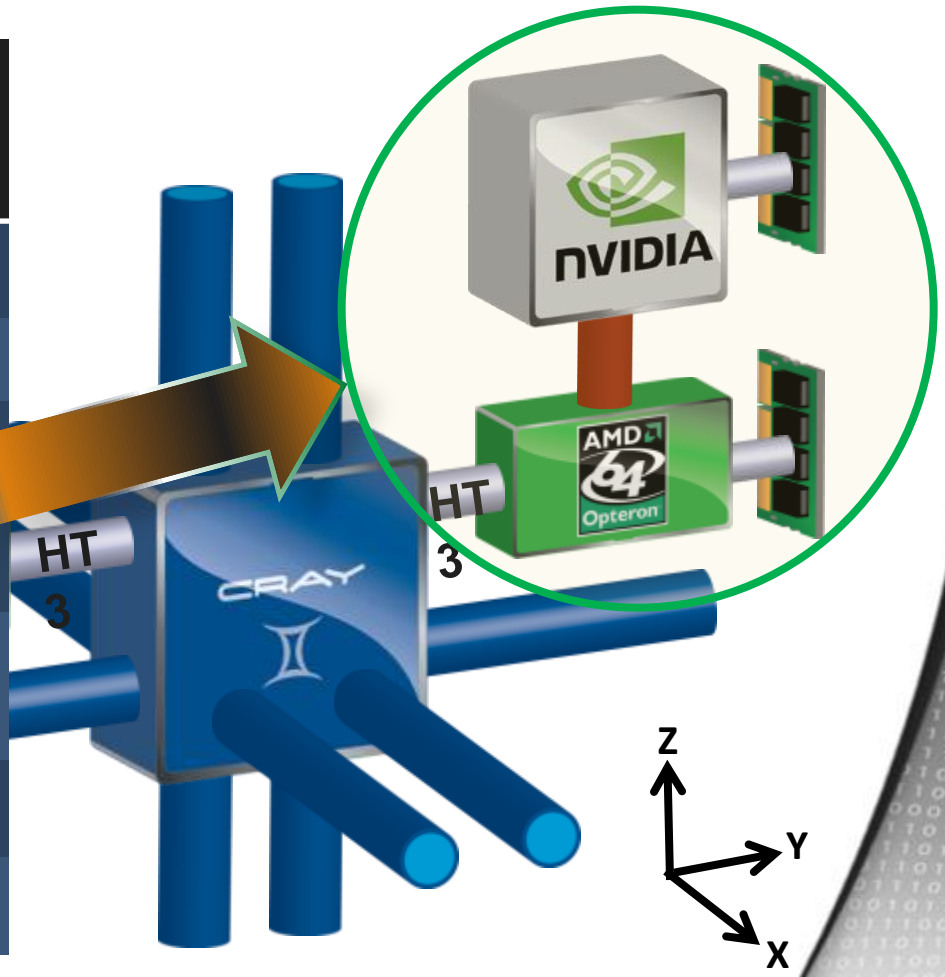


- CRAY XE6: “traditional” supercomputer
  - Each compute node contains 2 AMD Interlagos CPUs
- CRAY XK6: GPU Accelerated version
  - Each node contains 1 CPU + 1 NVIDIA X2090 GPU

CRAY THE SUPERCOMPUTER COMPANY | epcc

Cray XK6 Compute Node

XK6 Compute Node Characteristics
AMD Series 6200 (Interlagos)
NVIDIA Tesla X2090
Host Memory 16 or 32GB 1600 MHz DDR3
NVIDIA Tesla X2090 Memory 6GB GDDR5 capacity
Gemini High Speed Interconnect
Upgradeable to future GPUs



Performance Results

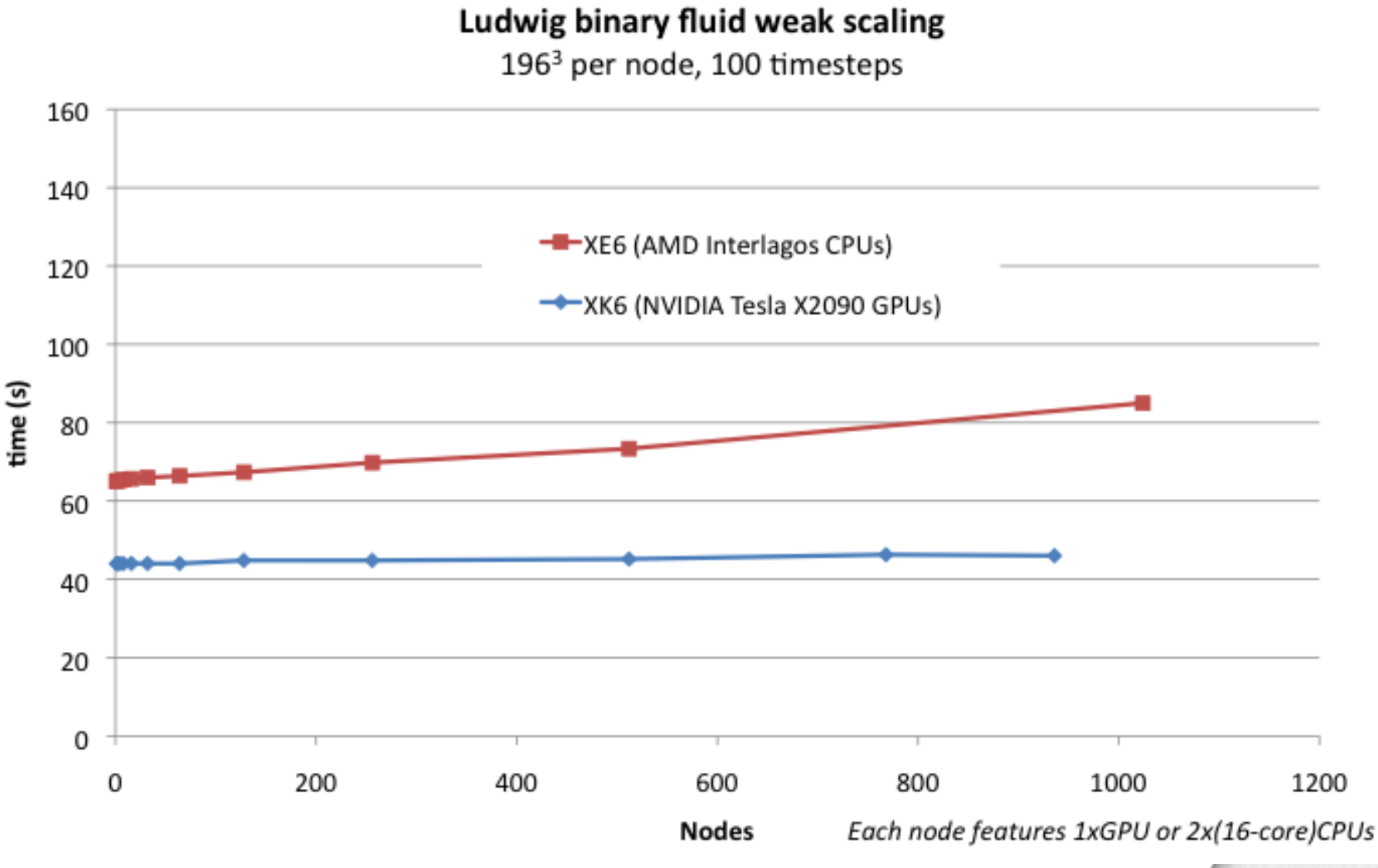
- The performance of the new GPU adaptation has been measured on
  - **Titan Prototype** (Cray/Oak Ridge National Laboratory)
    - Cray XK6
    - ~1000 compute nodes == ~1000 NVIDIA Tesla (Fermi) X2090 GPUs
      - (number of nodes regularly changed due to hardware testing)
    - nodes connected via Cray Gemini interconnect
    - Used 1 MPI task per node (1 per GPU)
- and compared to the original CPU version run on
  - **HECToR** (University of Edinburgh)
    - CRAY XE6
    - 2816 compute nodes == 5632 AMD Interlagos 16-core CPUs == 90,112 cores
    - nodes connected via Cray Gemini interconnect.
    - Used 32 MPI tasks per node (1 per CPU core)
    - CPU version highly optimised, including full utilization of SIMD vector units

- Theoretical Peak Capabilities

	AMD 16-core CPU (6276)	NVIDIA X2090 GPU
Double Precision Performance	147 Gflop/s	665 Gflop/s
Memory Bandwidth	36.5 GB/s	177 GB/s

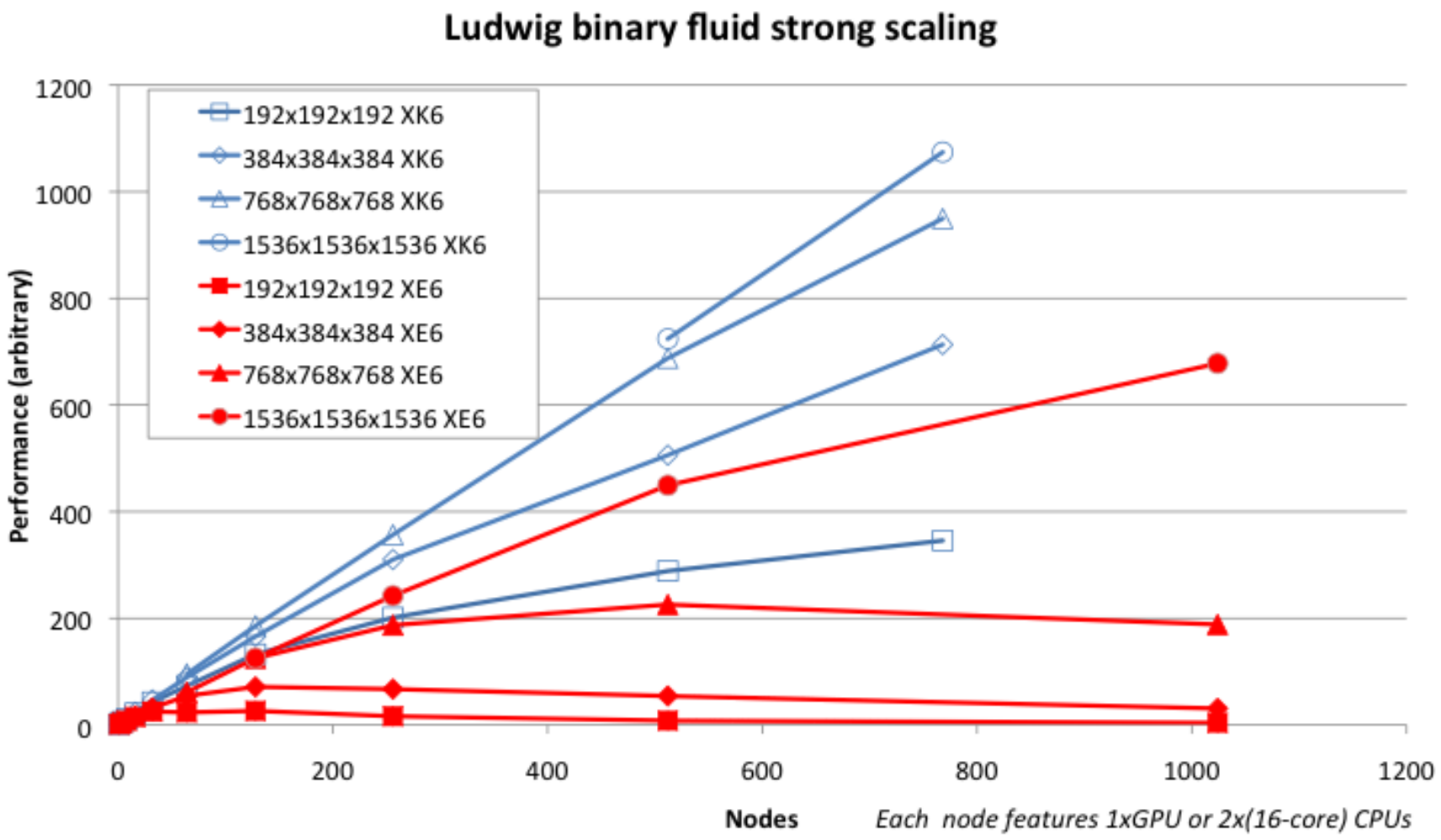
- We are comparing 2xCPUs (XE6 node) with 1xGPU (XK6 node)
  - Noting that the GPU version does not do any significant computation on the host CPU.

Ludwig Performance Results

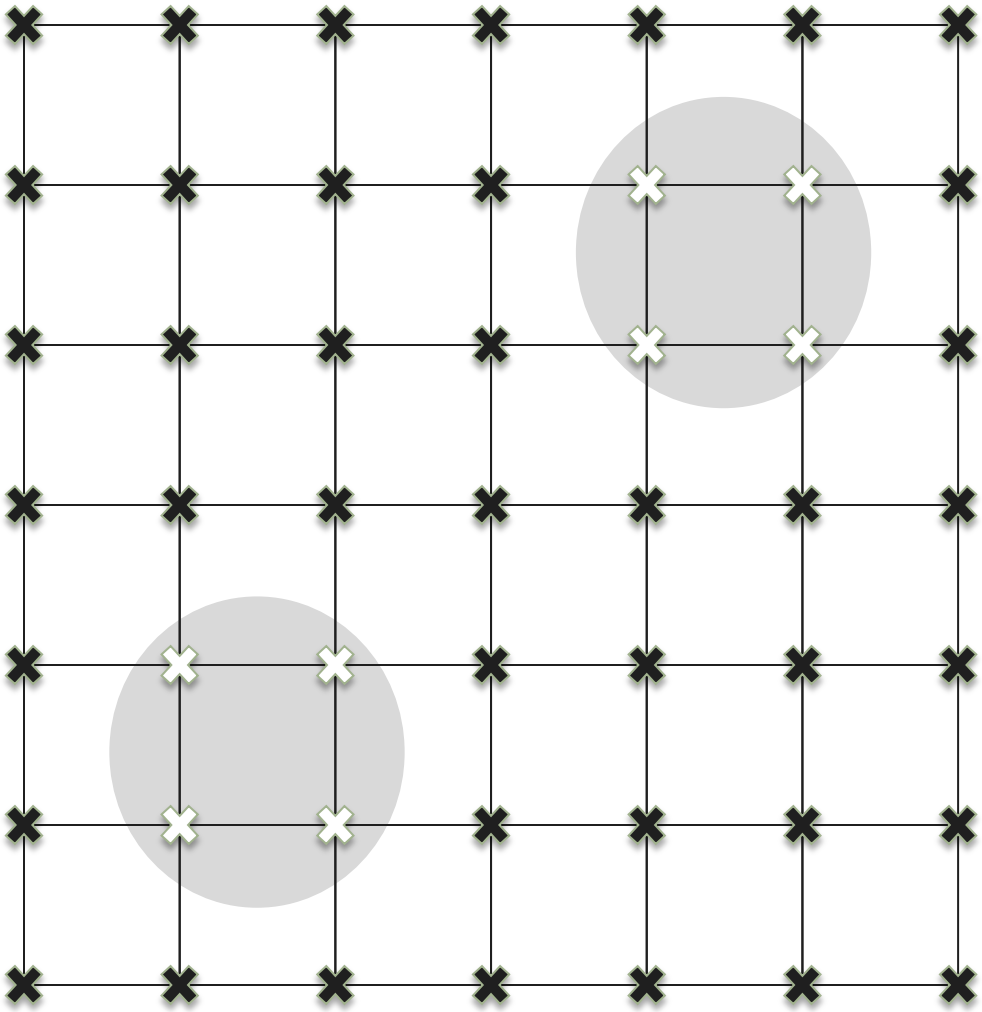


CRAY THE SUPERCOMPUTER COMPANY | epcc

Ludwig Performance Results



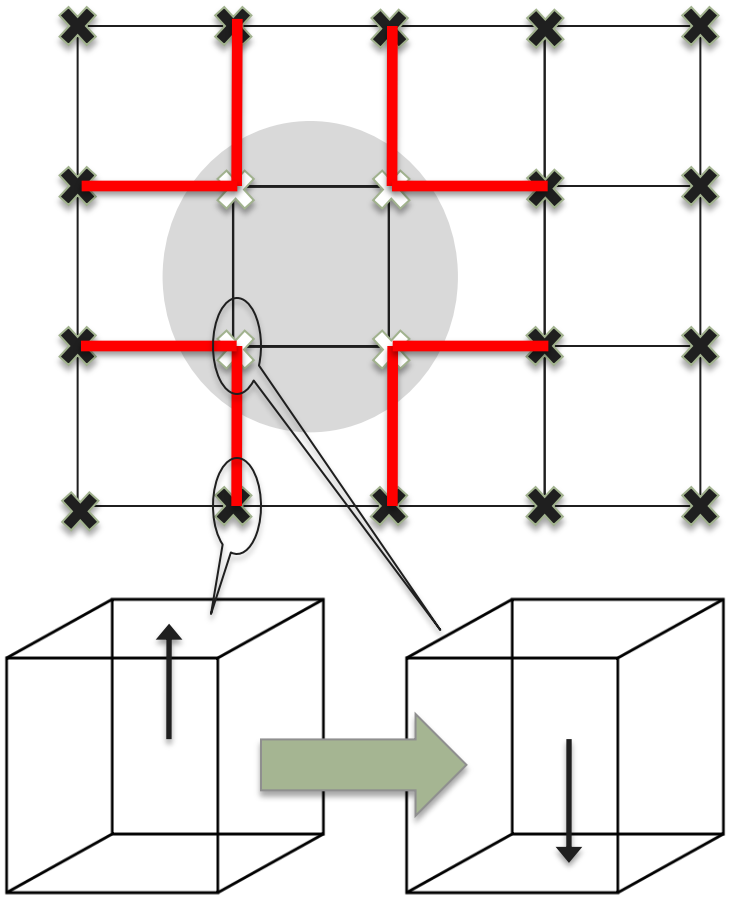
Current work: Introducing Particles



- Dynamic particles interact with fluid
- Each particle spans multiple lattice sites (which are marked x in diagram)
- Transfer of momentum
  - Fluid bounces off particle surface
  - Giving a kick to the particle

CRAY THE SUPERCOMPUTER COMPANY | epcc

Particles



- Each particle representation contains a list of all the links that it intercepts
- Fluid “bounces back” from particle:
  - Appropriate velocity component of fluid on site OUTSIDE particle is moved to site INSIDE particle at other end of link
    - And reversed in direction
  - Then, during propagation stage it will leave the particle again
    - Bounce back

Particle-Fluid interaction

- For each particle
  - For each link
    - Calculate coefficient
    - Bounce back fluid using coefficient
    - Update particle momentum
- This process is relatively inexpensive
- To integrate with GPU version, could:
  - Perform fully on CPU
    - Would require fluid data transfer every timestep – too expensive
  - Perform fully on GPU
    - Would require particle data to be kept resident on GPU
      - Major coding effort and divergence of CPU and GPU versions of code
      - Complications with particles moving between MPI subdomains
  - Solution: keep particles resident on CPU, but offload interaction only to GPU

Particle-Fluid interaction

- For each particle
  - For each link
    - Calculate coefficient
    - Bounce back fluid using coefficient
    - Update particle momentum
- This can be restructured as (introducing arrays of length Nparticles*Nlinks)

1. For each particle, for each link
   - Store site and velocity indices associated with link
   - Calculate and store coefficient
2. For each particle, for each link
   - Update velocity data using stored values
   - Calculate and store particle momenta updates
3. For each particle, for each link
   - Update particle momentum using stored values

- Stage 2 is only one that that accesses fluid data: can be moved to the GPU. Stages 1 and 3 are inexpensive and can be kept on CPU

Particle-Fluid interaction

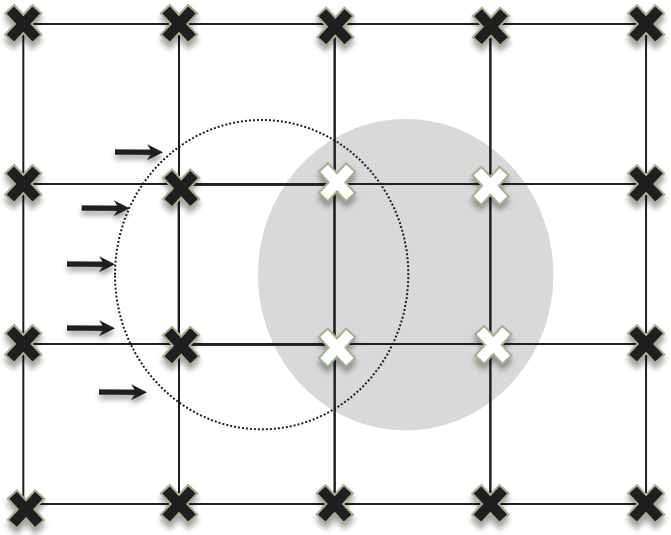
Stage 2: For each particle, for each link

- Update velocity data using stored values
- Calculate and store particle momenta updates

This becomes:

- Transfer coefficient and link index arrays to GPU
- GPU kernel to update fluid data, calculate and store particle momenta updates
- Transfer particle momenta updates array back to CPU

Particle movement



- Particles move slowly through the lattice
- When a particle moves off a lattice site, the fluid on that site must be reconstructed
- Similarly, the fluid on the newly obstructed site must be removed
- We keep the particle dynamics on the CPU, and just transfer the (small) subset of sites affected at each timestep.
- Work in progress

CRAY THE SUPERCOMPUTER COMPANY | epcc

Summary

- The Ludwig LB code has been successfully adapted to use a large number of GPUs in parallel.
- Both single-GPU and multi-GPU optimizations important in harnessing available performance capability.
- Work has resulted in a software package able to scale excellently on traditional or GPU accelerated systems.
  - GPU version has performance advantage and scales excellently to 936 GPUs
- Current and future work will include GPU enablement of advanced functionality.
  - Particle suspensions and liquid crystals of particular interest for new fast-switching LCD displays

CRAY THE SUPERCOMPUTER COMPANY | epcc |