OP2: an open-source library for unstructured grid applications

Mike Giles, Gihan Mudalige

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute

Oxford e-Research Centre

2nd UK GPU Computing Conference

Cambridge University, December 13, 2010

Outline

- opportunity, challenges, context
- user perspective (i.e. application developer)
  - API
  - build process
- implementation issues
  - hierarchical parallelism on GPUs
  - data dependency
  - code generation
- current status
- lessons learned so far

New heterogeneous hardware

For 10 years, 1995-2005, HPC was relatively simple:

- large clusters, with each node having 2 scalar CPUs
- MPI programming with FORTRAN / C / C++

Now things have become much more complicated:

- multi-core CPUs – up to 12 cores / 24 threads per CPU
- each core also has a vector unit – doubling in size to AVX very soon (and doubling again in near future)
- GPUs have up to 512 cores
- Best programming approach unclear:
  - MPI + OpenMP, or MPI + Ct for CPUs
  - CUDA for GPUs (and CPUs?)

Software Challenges

- HPC application developers want the benefits of the latest hardware but are very worried about the software development costs, and the level of expertise required

- status quo is not an option – running 24 MPI processes on a single CPU would give very poor performance, plus we need to exploit the vector units

- For GPUs, I'm happy with CUDA, but like MPI it's too low-level for many people

- For CPUs, MPI + OpenMP may be a good starting point, and PGI/CRAY are proposing OpenMP extensions which would support GPUs and vector units

- However, hardware is likely to change rapidly in next few years, and developers can not afford to keep changing their software implementation

Software Abstraction

To address these challenges, need to move to a suitable level of abstraction:

- separate the user's specification of the application from the details of the parallel implementation

- aim to achieve application level longevity with the top-level specification not changing for perhaps 10 years

- aim to achieve near-optimal performance through re-targetting the back-end implementation to different hardware and low-level software platforms
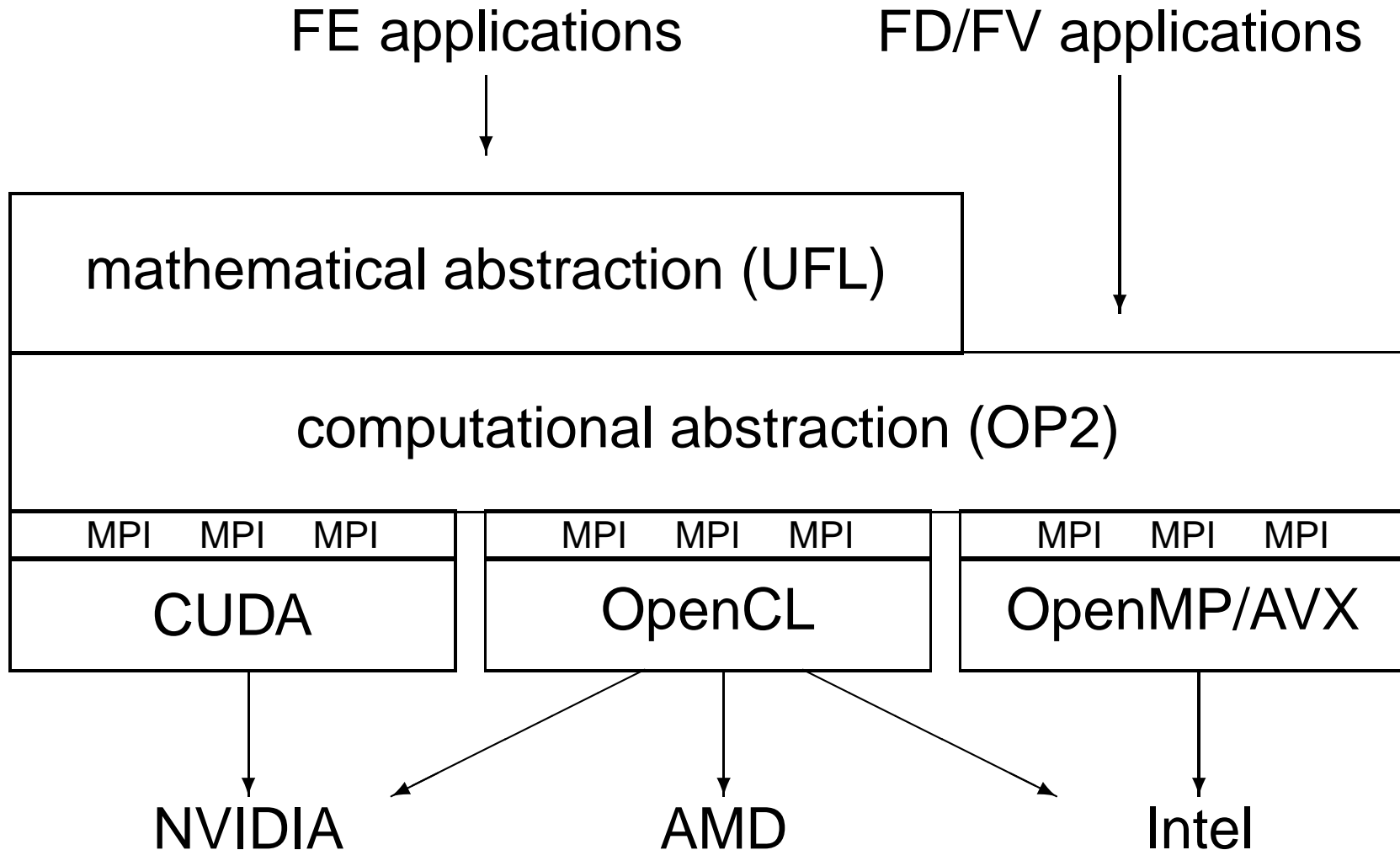
Context

Unstructured grid methods are one of Phil Colella's seven dwarfs (*Parallel Computing: A View from Berkeley*)

- dense linear algebra
- sparse linear algebra
- spectral methods
- N-body methods
- structured grids
- unstructured grids
- Monte Carlo

Extensive GPU work for the other dwarfs, except perhaps for direct sparse linear algebra.

Context

Part of a larger project led by Paul Kelly at Imperial College

FE applications   FD/FV applications

mathematical abstraction (UFL)

computational abstraction (OP2)

MPI	MPI	MPI		MPI	MPI	MPI		MPI	MPI	MPI

CUDA   OpenCL   OpenMP/AVX

NVIDIA   AMD   Intel

History

OPlus (Oxford Parallel Library for Unstructured Solvers)

- developed for Rolls-Royce 10 years ago

- MPI-based library for HYDRA CFD code on clusters
  with up to 200 nodes

OP2:

- open source project

- keeps OPlus abstraction, but slightly modifies API

- an "active library" approach with code transformation to
  generate CUDA, OpenCL and OpenMP/AVX code
  for GPUs and CPUs

OP2 Abstraction

- sets (e.g. nodes, edges, faces)

- datasets (e.g. flow variables)

- pointers (e.g. from edges to nodes)

- parallel loops
  - operate over all members of one set
  - datasets have at most one level of indirection
  - user specifies how data is used
    (e.g. read-only, write-only, increment)

OP2 Restrictions

- set elements can be processed in any order, doesn't affect result to machine precision
  - explicit time-marching, or multigrid with an explicit smoother is OK
  - Gauss-Seidel or ILU preconditioning in not
- static sets and pointers (no dynamic grid adaptation)

OP2 API

```
op_init(int argc, char **argv)

op_decl_set(int size, op_set *set, char *name)

op_decl_ptr(op_set from, op_set to, int dim,
            int *iptr, op_ptr *ptr, char *name)

op_decl_const(int dim, char *type,
            T *dat, char *name)

op_decl_dat(op_set set, int dim, char *type,
            T *dat, op_dat *data, char *name)

op_exit()
```

OP2 API

Parallel loop for user kernel with 3 arguments:

```
op_par_loop_3(void (*kernel)(T0*, T1*, T2*),
    char *name, op_set set,
    op_dat arg0, int idx0, op_ptr ptr0,
    int dim0, char *typ0, op_access acc0,
    op_dat arg1, int idx1, op_ptr ptr1,
    int dim1, char *typ1, op_access acc1,
    op_dat arg2, int idx2, op_ptr ptr2,
    int dim2, char *typ2, op_access acc2)
```

Example for sparse matrix-vector product:

```
op_par_loop_3(res,"res", edges,
    p_A, -1,OP_ID, 1,"float",OP_READ,
    p_u,  0,pedge2,1,"float",OP_READ,
    p_du, 0,pedge1,1,"float",OP_INC);
```
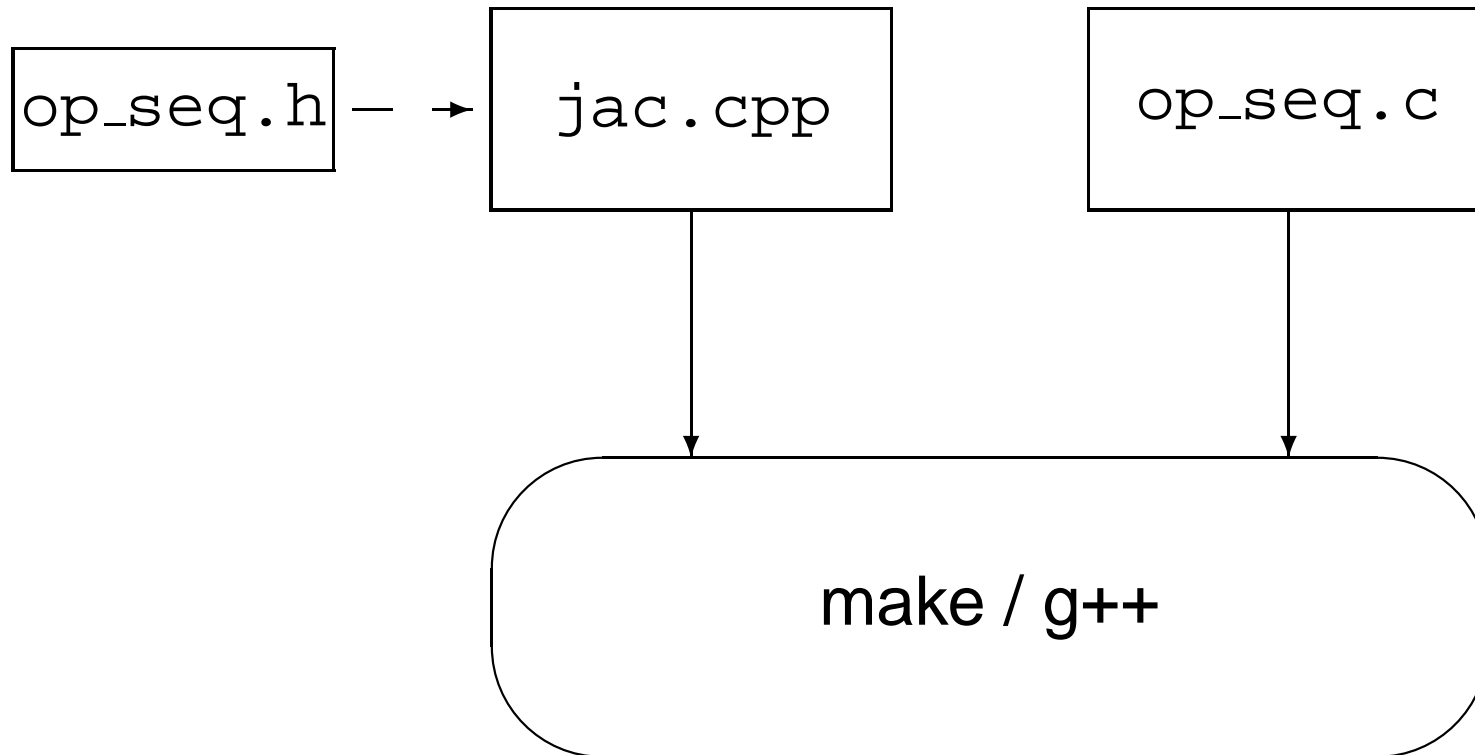
User build processes

Using the same source code, the user can build different executables for different target platforms:

- sequential single-thread CPU execution
  - purely for program development and debugging
  - very poor performance
- CUDA / OpenCL for single GPU
- OpenMP/AVX for multicore CPU systems
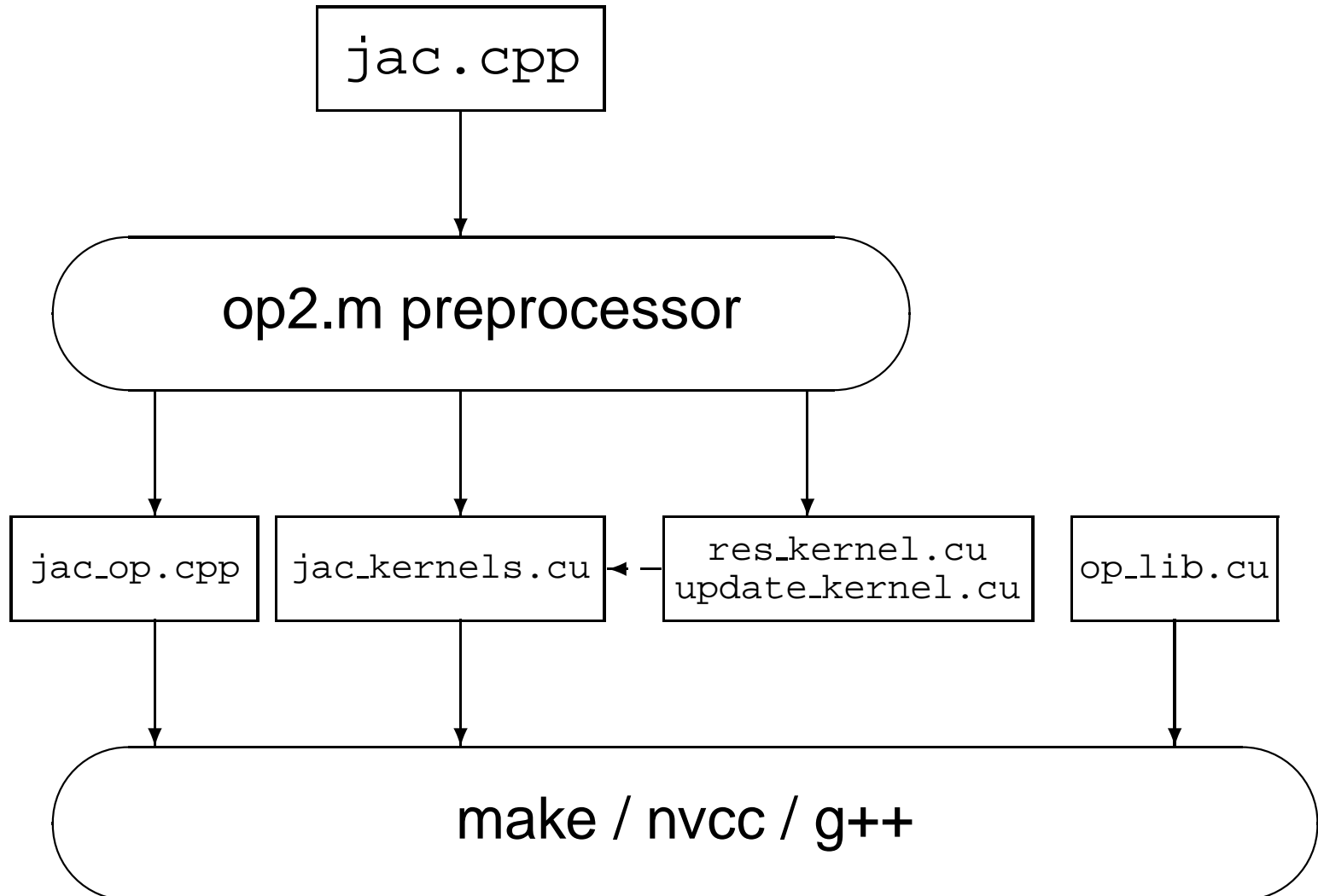- MPI plus any of the above for clusters

Sequential build process

Traditional build process, linking to a conventional library
in which many of the routines do little but error-checking:

```
op_seq.h ─ ─► jac.cpp          op_seq.c
                 │                  │
                 │                  │
                 ▼                  ▼
         ┌──────────────────────────────┐
         │         make / g++           │
         └──────────────────────────────┘
```

CUDA build process

Preprocessor parses user code and generates new code:

GPU Parallelisation

Could have up to 10^6 threads in 3 levels of parallelism:

- MPI distributed-memory parallelism (1-100)
  - one MPI process for each GPU
  - all sets partitioned across MPI processes, so each MPI process only holds its data (and halo)

- block parallelism (50-1000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different functional units in the GPU

- thread parallelism (32-128)
  - each mini-partition is worked on by a block of threads in parallel

GPU Parallelisation

The 14 functional units in an NVIDIA Fermi GPU each have

- 32 cores

- 48kB of shared memory and 16kB of L1 cache
  (or vice versa)

Mini-partitions are sized so that all of the indirect data
can be held in shared memory and re-used as needed

- reduces data transfer from/to main graphics memory

- very similar to maximising cache hits on a CPU to
  minimise data transfer from/to main system memory

- implementation requires re-numbering from global
 indices to local indices – tedious but not difficult

GPU Parallelisation

One important difference from MPI parallelisation

- when using one GPU, all data is held in graphics memory in between each parallel loop

- each loop can use a different set of mini-partitions

- current implementation constructs an "execution plan" the first time the loop is encountered

- auto-tuning will be used in the future to optimise the plan, either statically based on profiling data, or dynamically based on run-time timing

Data dependencies

Key technical issue is data dependency when incrementing indirectly-referenced arrays.
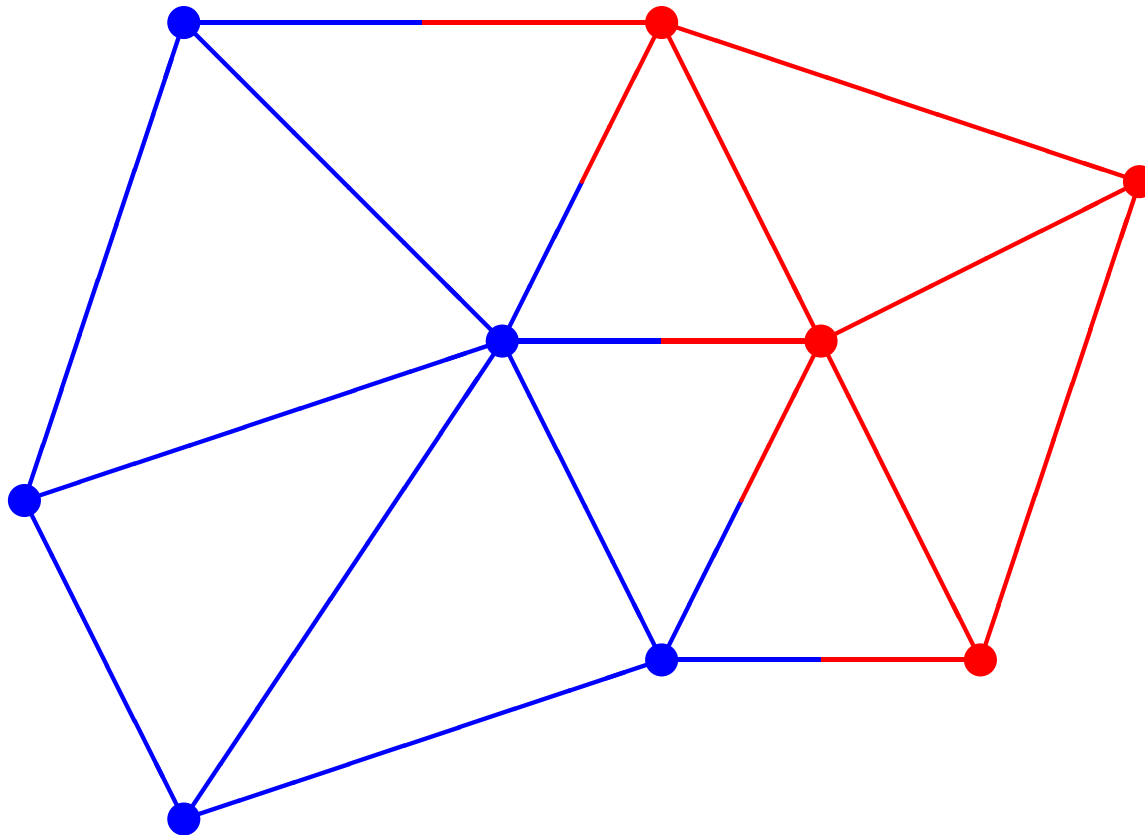
e.g. potential problem when two edges update same node
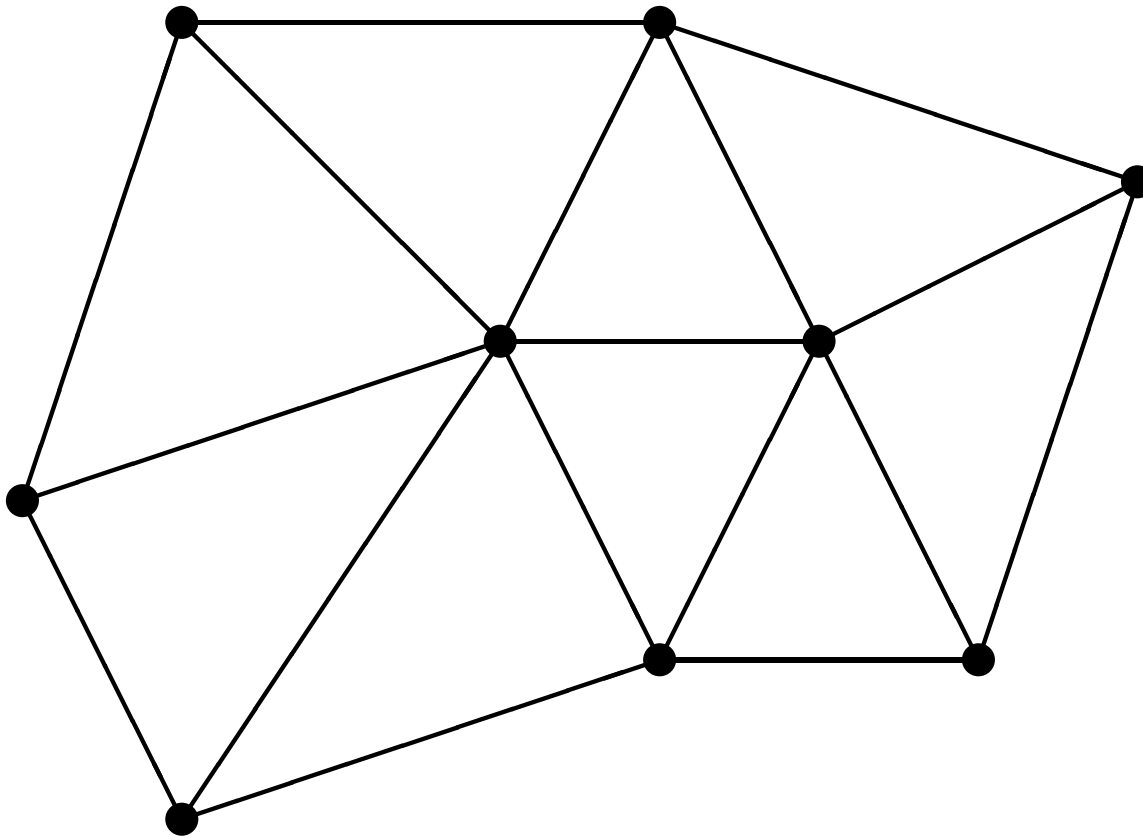
Data dependencies

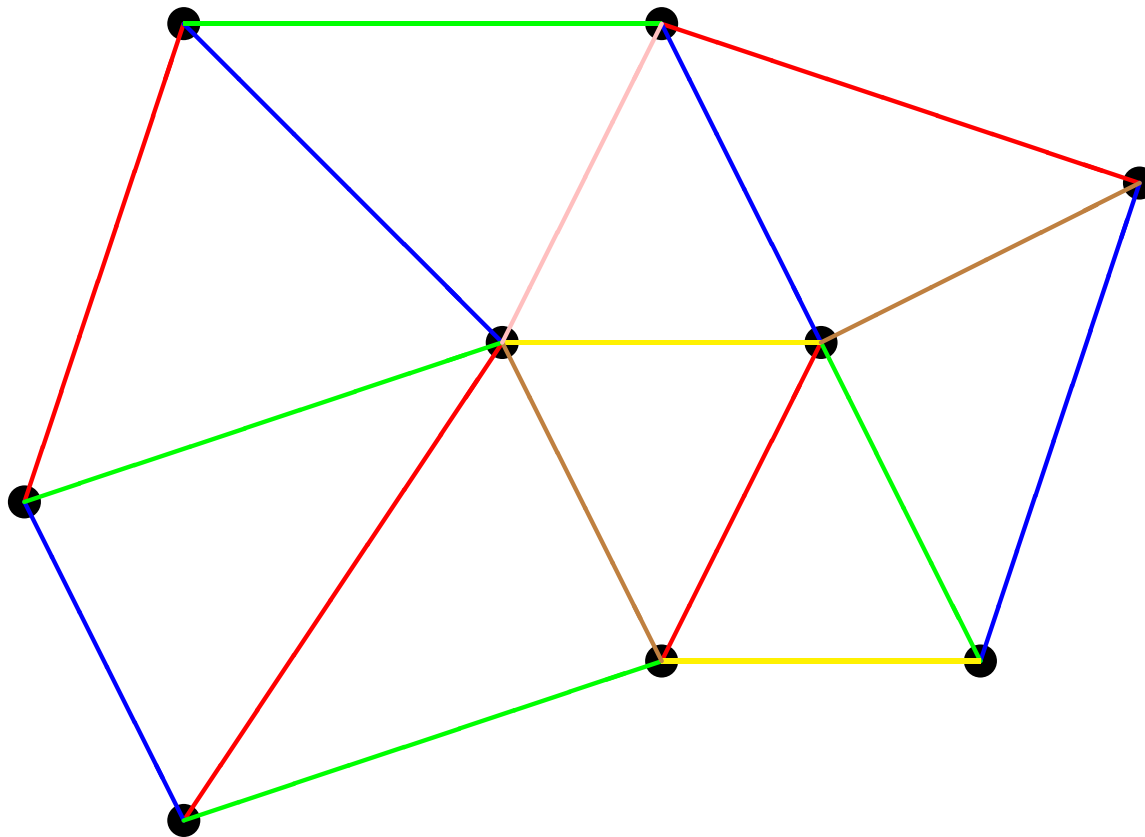Method 1: "owner" of nodal data does edge computation

- drawback is redundant computation when the two nodes have different "owners"

Data dependencies

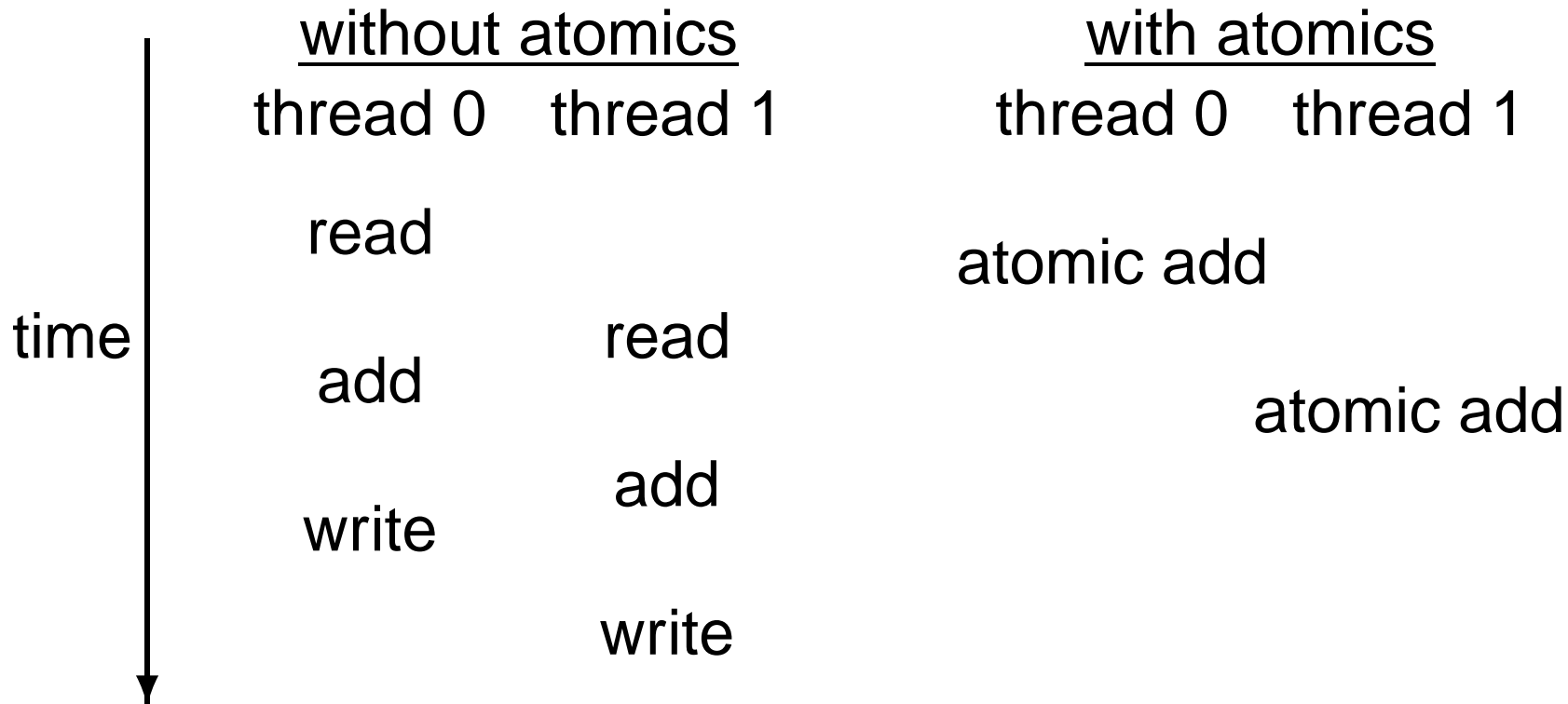Method 2: "color" edges so no two edges of the same color update the same node

- parallel execution for each color, then synchronize
- possible loss of data reuse and some parallelism

Data dependencies

Method 3: use "atomic" add which combines read/add/write into a single operation

- avoids the problem but needs hardware support

- drawback is slow hardware implementation

without atomics

	thread 0	thread 1
time	read	
	add	read
	write	add
		write

with atomics

	thread 0	thread 1
	atomic add	
		atomic add

Data dependencies

Which is best for each level?

- MPI level: method 1
    - each MPI process does calculation needed to update its data
    - partitions are large, so relatively little redundant computation
- GPU level: method 2
    - plenty of blocks of each color so still good parallelism
    - data reuse within each block, not between blocks
- block level: method 2 or 3
    - indirect data in local shared memory, so get reuse
    - which costs more, local synchronization or atomic updates?

Current status

- initial prototype (with code parser/generator written in MATLAB!) can generate:
    - CUDA code for a single GPU
    - OpenMP code for multiple CPUs
- Imperial College have re-implemented CUDA generator in Rose
- airfoil test case shows:
    - $30\times$ speedup on a single GPU
    - $7\times$ speedup for 2 quad-core CPUs

    relative to a single CPU thread
- work on MPI implementation will start in 2011, using Parmetis or Chaco for domain decomposition

Airfoil test code

- 2D Euler equations, cell-centred finite volume with scalar dissipation (miminal compute per memory reference – should consider switching to characteristic smoothing)

- roughly 1.5M edges, 0.75M cells

- 5 parallel loops:
  - `save_soln` (direct over cells)
  - `adt_calc` (indirect over cells)
  - `res_calc` (indirect over edges)
  - `bres_calc` (indirect over boundary edges)
  - `update` (direct over cells with RMS reduction)

Airfoil test code

- factor 2-4 data reuse in indirect access, but cache efficiency not known (need extra coding for this, or hardware monitoring)

- some routines seem close to bandwidth-limited; all have at least 30% bandwidth utilisation

- only factor 7 speedup on 16 CPU threads (2×4 cores, but hyperthreaded) – we think due to memory bandwidth limits

- single precision CUDA limited to 32 registers; double precision needs up to 63 (but no spillage into "local" memory)

Lessons learned so far

1) Code generation works, and it's not too difficult!

- in the past I've been scared of code generation since I have no computer science background

- key is the routine arguments have all of the information required, so no need to parse the entire user code

- now helping a maths student develop a code generator for stochastic simulations in computational biology

  - a generic solver is inefficient – a "hand-coded" specialised implementation for one specific model is much faster

  - code generator takes in model specification and tries to produce "hand-coded" custom implementation

- I think this is an important trend for the future

Lessons learned so far

2) Auto-tuning is going to be important

- there are various places in the CUDA code where I have a choice of parameter values (e.g. number of threads, number of blocks, size of mini-partitions, use of L1 cache, 16kB/48kB split between L1 cache and shared memory)

- there are also places where I have a choice of implementation strategy (e.g. thread coloring or atomic updates?)

- what I would like is a generic auto-tuning framework which will optimise these choices for me, given a reasonably small set of possible values

- as a first step, a undergraduate CS student is working with me on a 3rd year project on this

Conclusions

- have defined a high-level framework for parallel execution of algorithms on unstructured grids

- looks encouraging for providing ease-of-use, high performance, and longevity through new back-ends