OP2 – an open-source library for unstructured grid applications

(2023 comment: slides originally from 2013)

Mike Giles, Gihan Mudalige, Istvan Reguly

mike.giles@maths.ox.ac.uk

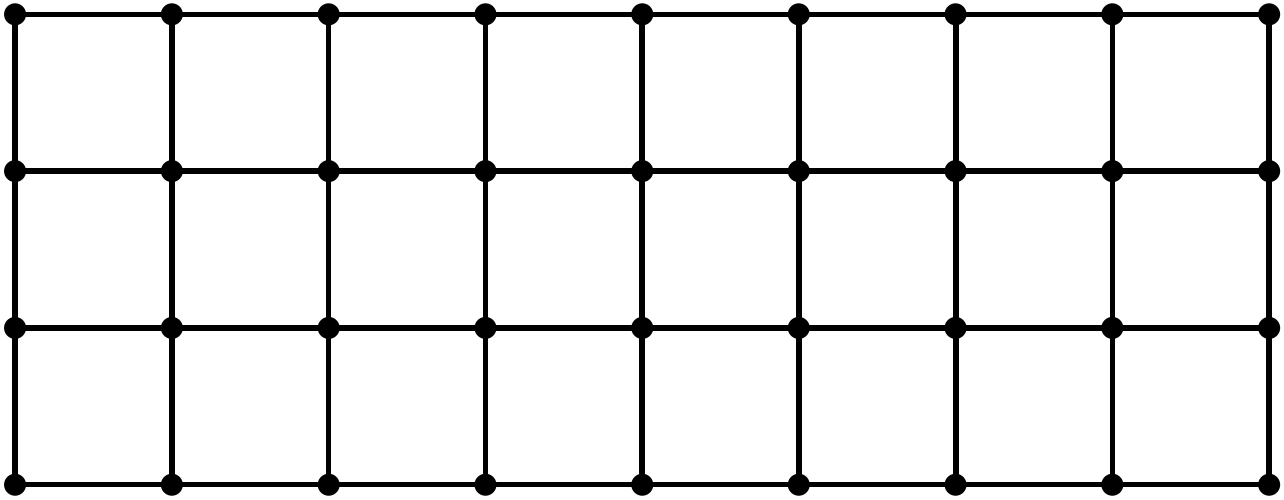
Oxford University Mathematical Institute

Oxford e-Research Centre

Outline

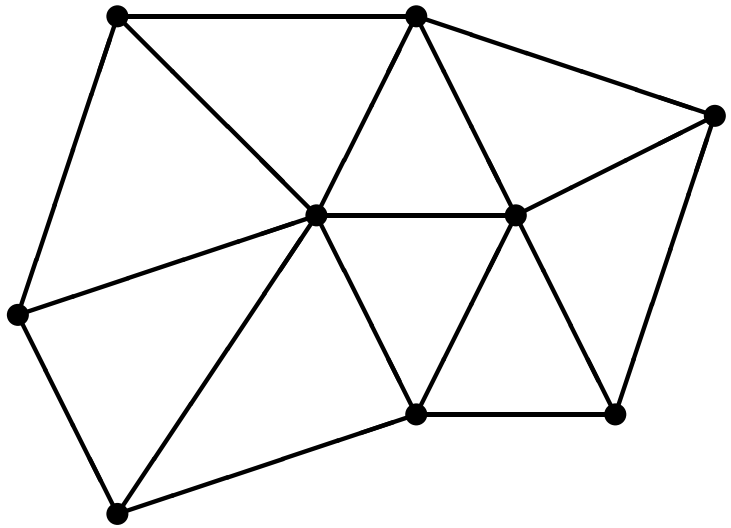
- structured and unstructured grids
- software challenge
- user perspective (i.e. application developer)
  - API
  - build process
- implementation issues
  - hierarchical parallelism on GPUs
  - data dependency
  - code generation
  - auto-tuning

Structured grids



- logical (i, j) indexing in 2d; (i, j, k) in 3D
- implicit connectivity – neighbours of node (i, j, k) are $(i \pm 1, j \pm 1, k \pm 1)$
- fairly easy to parallelise – see `laplace3d` example

Unstructured grids



- a collection of nodes, edges, faces, cells, etc., each addressed by a 1D index
- explicit connectivity – mapping tables define connections from edges to nodes, or faces to cells, etc.
- much harder to parallelise (not in concept so much as in practice) but a lot of existing literature on the subject
- used a lot because of geometric flexibility

Software Challenge

- Application developers want the benefits of the latest hardware but are very worried about the software development effort, and the expertise required
- Status quo is not really an option – running lots of single-thread MPI processes on multiple CPUs won't give great performance
- Want to exploit GPUs using CUDA, and CPUs using OpenMP/AVX
- 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 this challenge, 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 user 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

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 for GPUs and OpenMP/AVX for CPUs

OP2 Abstraction

- sets (e.g. nodes, edges, faces)
- datasets (e.g. flow variables)
- mappings (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 is not
- static sets and mappings (no dynamic grid adaptation)

OP2 API

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

op_set op_decl_set(int size, char *name)

op_map op_decl_map(op_set from, op_set to,
                   int dim, int *imap, char *name)

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

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

void op_exit()
```

OP2 API

Example of parallel loop syntax for a sparse matrix-vector product:

```
op_par_loop(res,"res", edges,
  op_arg_dat(A,-1,OP_ID,1,"float",OP_READ),
  op_arg_dat(u, 1,pedge,1,"float",OP_READ),
  op_arg_dat(du,0,pedge,1,"float",OP_INC));
```

This is equivalent to the C code:

```
for (e=0; e<nedges; e++)
   du[pedge[2*e]] += A[e] * u[pedge[1+2*e]];
```

where each “edge” corresponds to a non-zero element in the matrix `A`, and `pedge` gives the corresponding row and column indices.

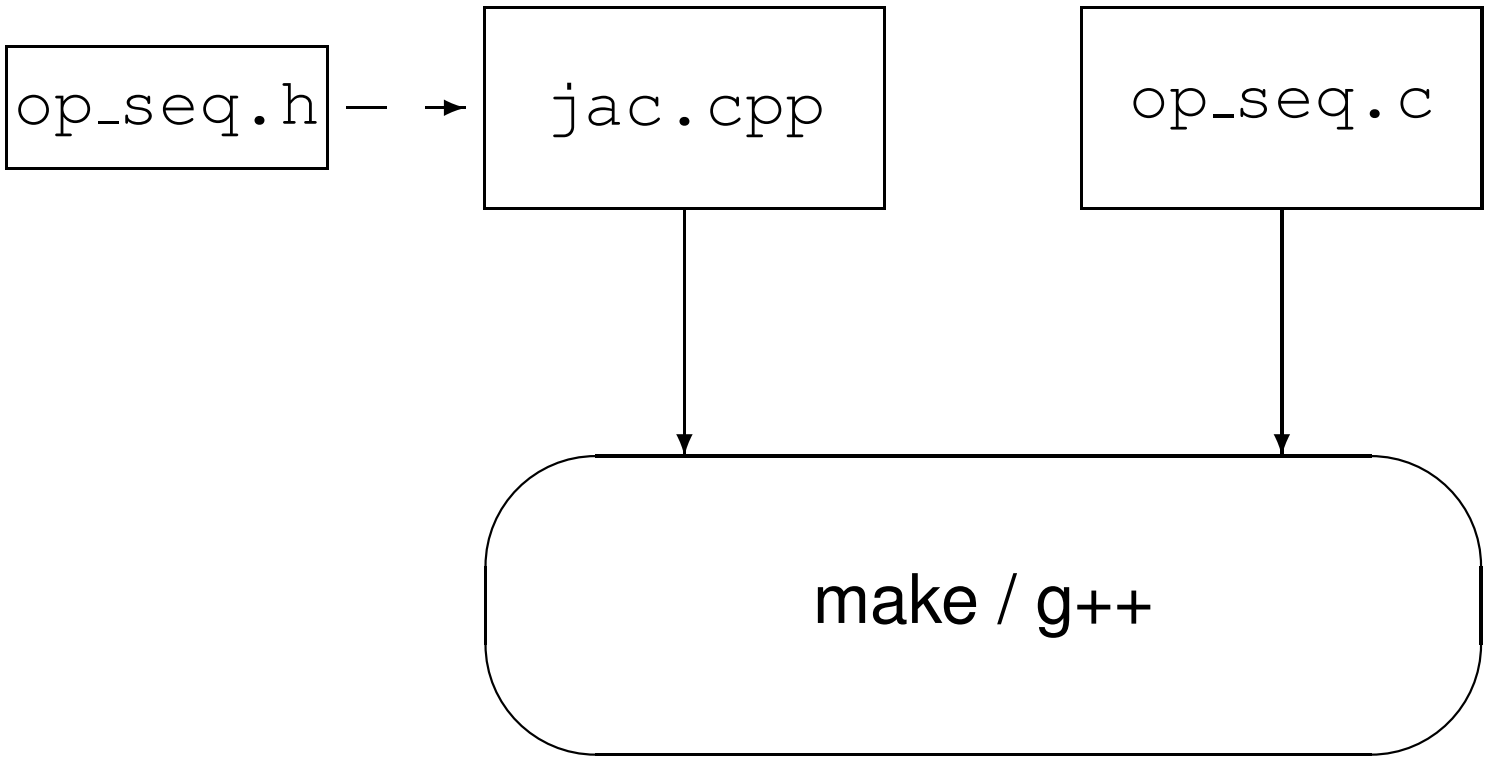
User build processes

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 for single GPU
- OpenMP/AVX for multicore CPU systems
- MPI plus any of the above for clusters

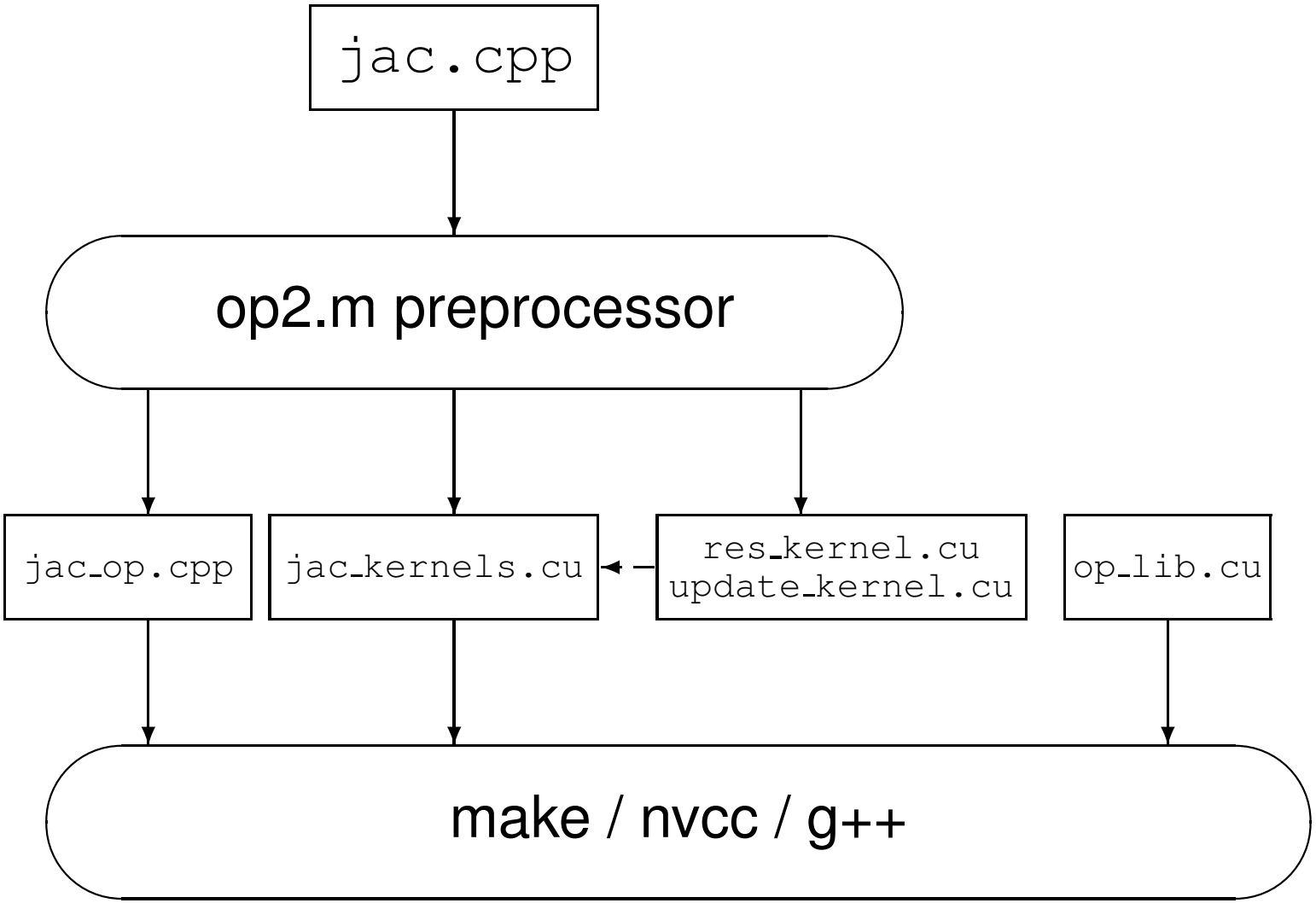
Sequential build process

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



CUDA build process

Preprocessor parses user code and generates new code:



Implementation Approach

The question now is how to deliver good performance on multiple GPUs

Initial assessment:

- lots of natural parallelism on grids with up to 10^9 nodes/edges
- not a huge amount of compute per node/edge so important to
  - avoid PCIe transfers as much as possible
  - achieve good data reuse to minimise GPU / global memory transfers
- have to be careful with data dependencies

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)
  - each partition sized to fit within global memory of GPU (up to 6GB)
  - only halos need to be transferred from one GPU to another, via the CPUs
  - hopefully, this will give a balanced implementation – slight possibility that MPI networking will end up being the primary bottleneck, so will work hard to overlap computation and MPI communication

GPU Parallelisation

- block parallelism (50-1000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different SMs within the GPU
  - each mini-partition is sized so that all of the indirect data can be held in shared memory and re-used as needed
  - implementation requires re-numbering from global indices to local indices – tedious but not difficult
  - can use different mini-partitions for different parallel loops – “execution plan” generated during startup
- thread parallelism (32-128)
  - each mini-partition is worked on by a block of threads in parallel

Shared memory or L1 cache?

Caches:

- easy to use, but hard to predict/understand performance
- good performance for structured grids where often all of the cache line is used
- not so good for unstructured grids with indirect addressing

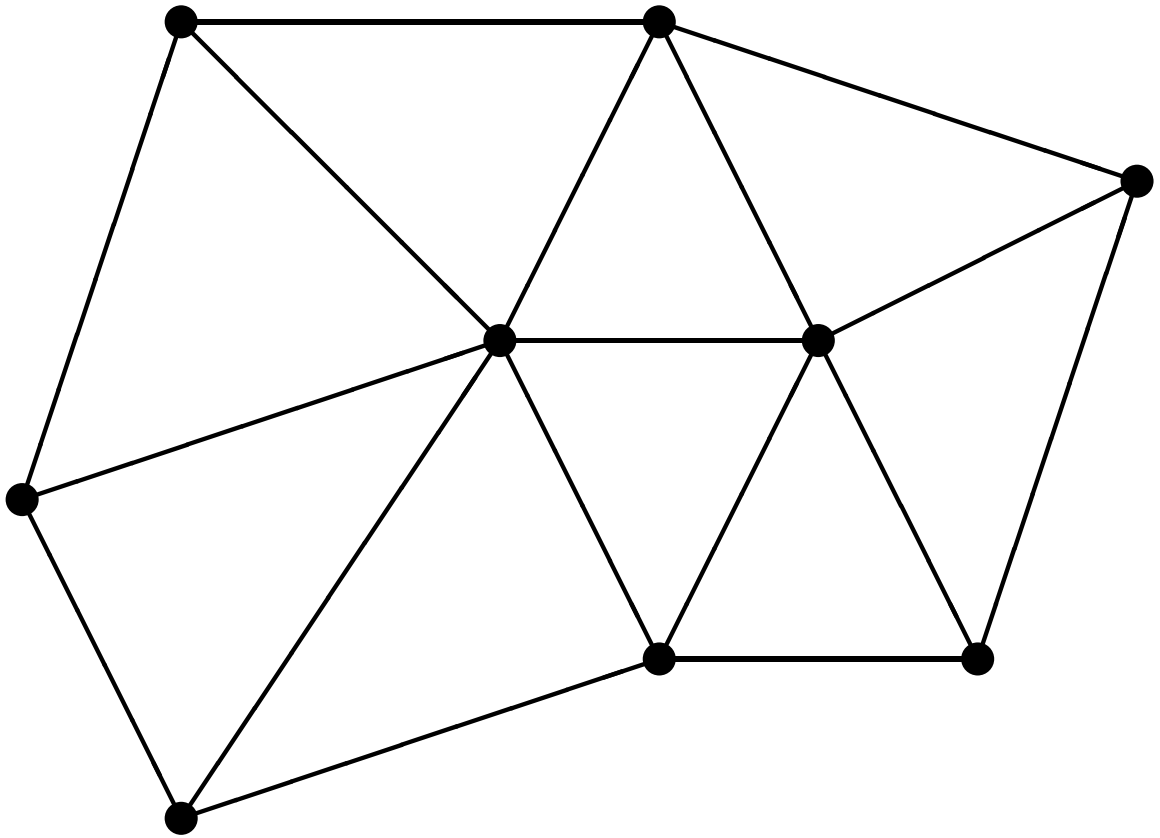
Shared memory:

- full control means you understand performance
- only store the data which is actually needed
- tedious to implement, but that's the point of a library, to do the tedious things so users don't have to

Data dependencies

Key technical issue is data dependency when incrementing indirectly-referenced arrays.

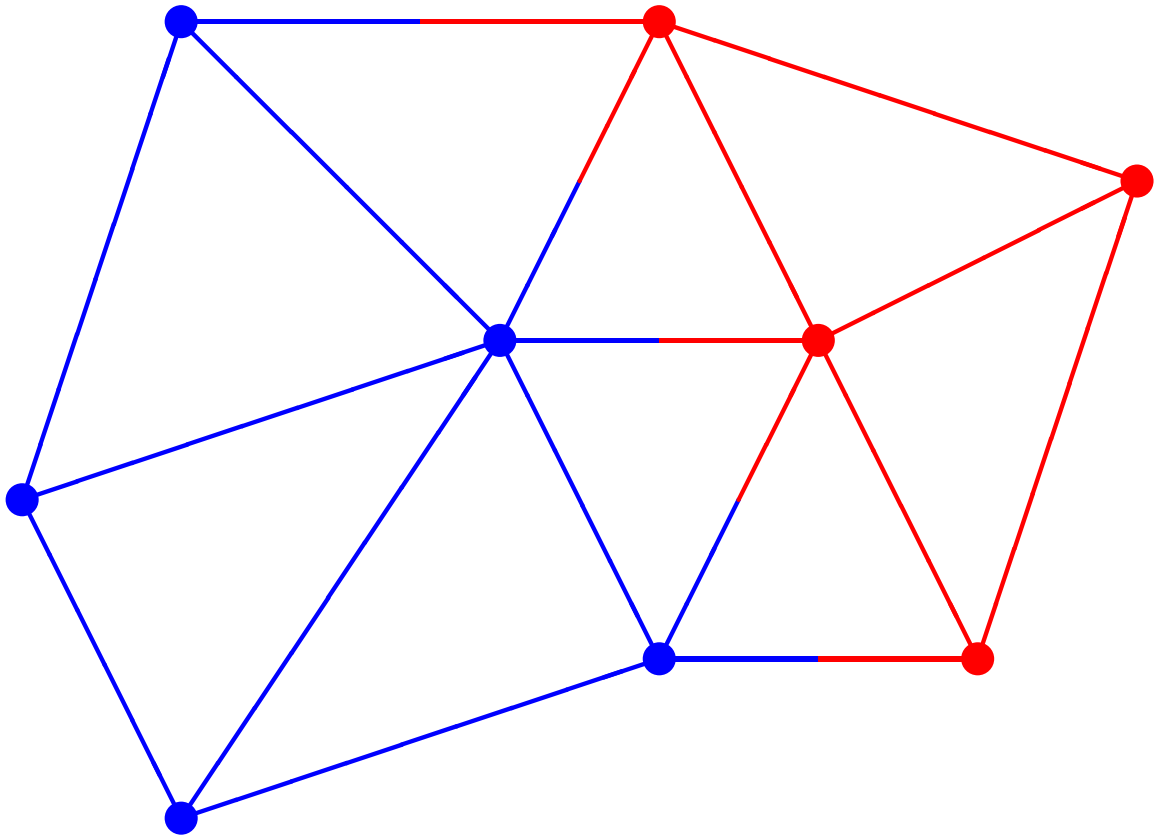
e.g. potential problem when two edges update same node



Data dependencies

Method 1: “owner” of nodal data does edge computation

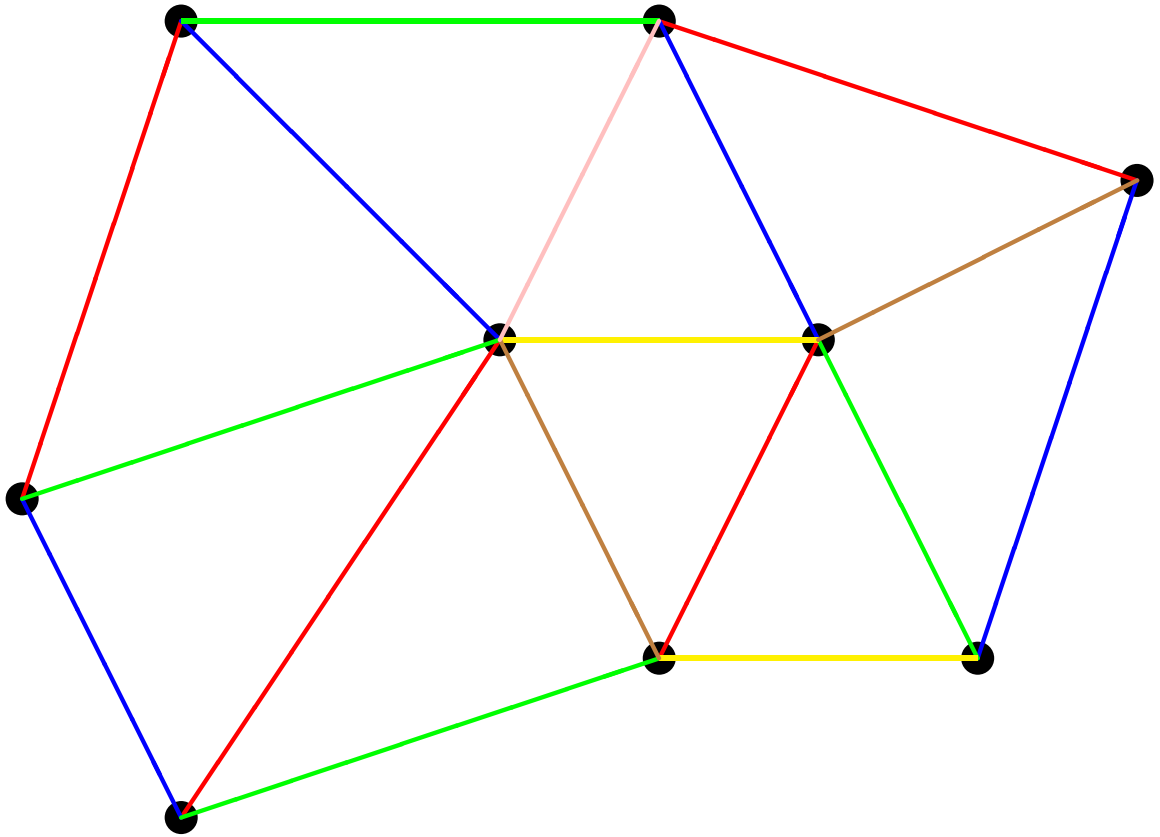
- drawback is redundant computation when the two nodes have different “owners”



Data dependencies

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

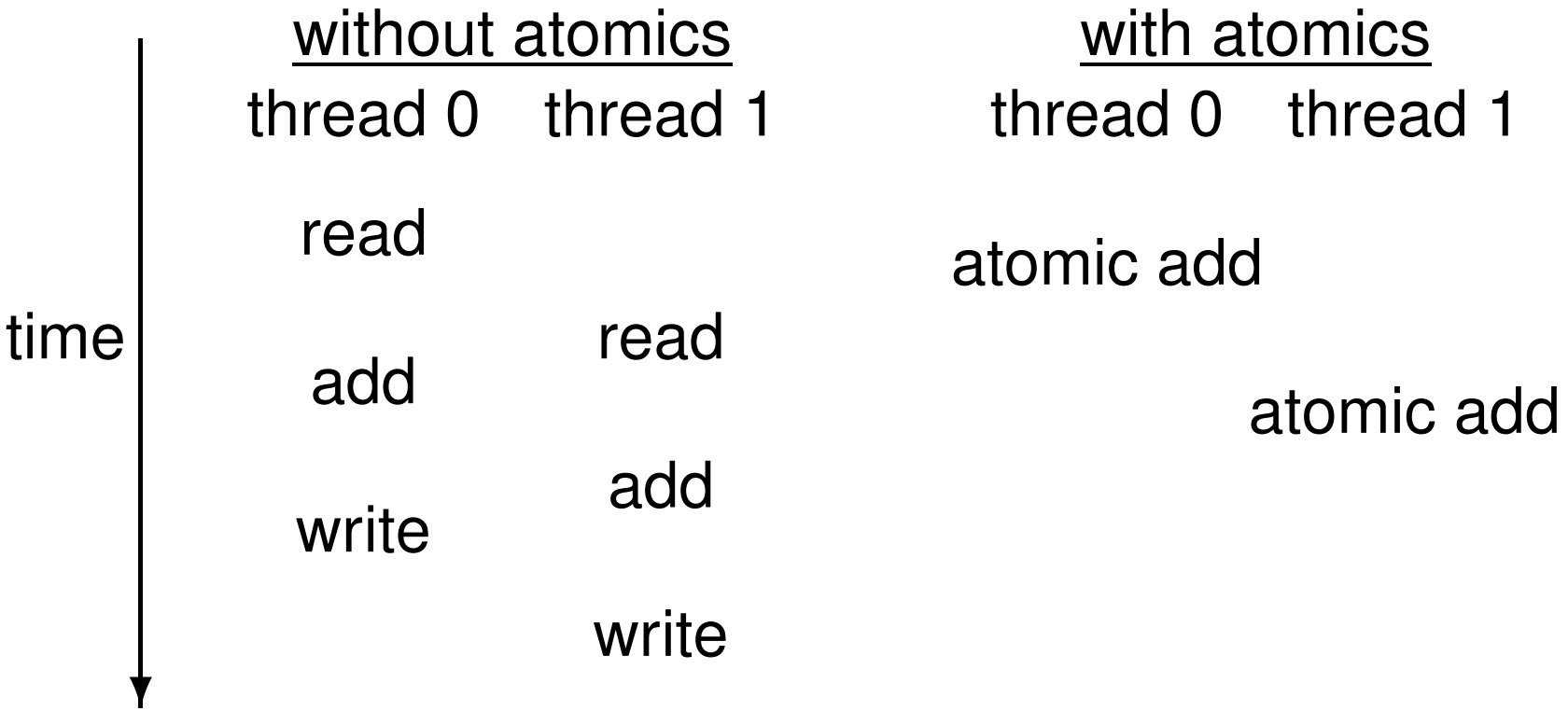
- parallel execution for each color, then synchronize
- possible loss of data reuse and some parallelism



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



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 (2023 update: method 3)
  - indirect data in local shared memory, so get reuse
  - individual threads are colored to avoid conflict when incrementing shared memory

Code Generation

Initial prototype, with code parser/generator written in MATLAB (2023 update: later switched to python), can generate:

- CUDA code for a single GPU
- OpenMP code for multiple CPUs

The parallel loop API requires redundant information:

- simplifies MATLAB program generation – just need to parse loop arguments, not entire code
- numeric values for dataset dimensions enable compiler optimisation of CUDA code
- "programming is easy; it's debugging which is difficult" – not time-consuming to specify redundant information provided consistency is checked automatically

Auto-tuning

In the CUDA implementation there are various parameters and settings which apply to the whole code:

- compiler flags, such as whether to use L1 caching
- (whether to use AoS or SoA storage for each dataset)

and others which can be different for each CUDA kernel:

- number of threads in a thread block
- size of each mini-partition
- (whether to use a 16/48 or 48/16 split for the L1 cache / shared memory)

Auto-tuning

In each case, the optimum choice / value is not obvious, but it is possible to

- give a small set of possible values for each (usually two or three)
- state which can be optimised independently (e.g. the parameters for one kernel don't affect the execution of another kernel)

What is then needed is a flexible auto-tuning system to select the optimum combination by exhaustive "brute force" search.

The parameter independence is essential to making this viable.

Auto-tuning

A flexible auto-tuning package has been developed:

- written in Python
- input specification includes
  - parameters and possible values
  - a mechanism to compile the code, perhaps using some of the parameter values
  - a mechanism to run the code, again perhaps using some of the parameter values
  - by default, the run-time is used as the "figure-of-merit" to be optimised
- at present only brute-force optimisation is supported, but in the future other strategies may be included

Auto-tuning

Example configuration file:

```
#
# parameters and values
#
PARAMS = { flag, {block0, part0}, {block1, part1} }

flag   = {"-Xptxas -dlcm=ca", "-Xptxas -dlcm=cg" }  # compiler flag
block0 = {64, 96, 128}   # thread block size for loop 0
part0  = {128, 192, 256} # partition size for loop 0
block1 = {64, 96, 128}   # thread block size for loop 1
part1  = {128, 192, 256} # partition size for loop 1

#
# compilation and evaluation mechanisms
#
COMPILER = make -B flag=%flag% block0=%block0% part0=%part0%
                              block1=%block1% part1=%part1%
EVALUATION = ./executable
```

Airfoil test code

- 2D Euler equations, cell-centred finite volume method with scalar dissipation (miminal compute per memory reference – should consider switching to more compute-intensive “characteristic” smoothing more representative of real applications)
- 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

Library is instrumented to give lots of diagnostic info:

```
new execution plan #1 for kernel res_calc
number of blocks       = 11240
number of block colors = 4
maximum block size     = 128
average thread colors  = 4.00
shared memory required = 3.72 KB
average data reuse     = 3.20
data transfer (used)   = 87.13 MB
data transfer (total)  = 143.06 MB
```

- factor 2-4 data reuse in indirect access, but up to 40% of cache lines not used on average

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using initial parameter values:

- mini-partition size (PS): 256 elements
- blocksize (BS): 256 threads

count	time	GB/s	GB/s	kernel name
1000	0.23	107.8		save_soln
2000	1.26	61.0	63.1	adt_calc
2000	5.10	32.5	53.4	res_calc
2000	0.11	4.8	18.4	bres_calc
2000	1.07	110.6		update
TOTAL	7.78			

Second B/W column includes whole cache line

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.22	101.8		save_soln		512
2000	1.09	74.1	75.4	adt_calc	256	128
2000	4.95	36.9	60.6	res_calc	128	128
2000	0.10	5.3	20.0	bres_calc	64	128
2000	1.03	94.7		update		64
TOTAL	7.40					

This is a 5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Double precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.44	104.9		save_soln		512
2000	2.62	52.9	53.8	adt_calc	256	128
2000	10.35	30.5	50.8	res_calc	128	128
2000	0.08	11.2	27.9	bres_calc	64	128
2000	1.87	104.5		update		64
TOTAL	15.36					

This is a 7.5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would again increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Single precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 16

count	time	GB/s	GB/s	kernel name	PS
1000	1.68	13.7		save_soln	
2000	11.15	7.3	7.5	adt_calc	128
2000	16.57	10.3	11.2	res_calc	1024
2000	0.16	3.2	11.9	bres_calc	64
2000	4.67	20.9		update	
TOTAL	34.25				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Airfoil test code

Double precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 12

count	time	GB/s	GB/s	kernel name	PS
1000	2.51	18.3		save_soln	
2000	11.68	11.8	11.9	adt_calc	1024
2000	20.99	12.8	13.5	res_calc	1024
2000	0.17	5.0	12.4	bres_calc	512
2000	9.29	21.1		update	
TOTAL	44.64				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Conclusions

- have created a high-level framework for parallel execution of unstructured grid algorithms on GPUs and other many-core architectures
- looks encouraging for providing ease-of-use, high performance and longevity through new back-ends
- auto-tuning is useful for code optimisation, and a new flexible auto-tuning system has been developed
- C2070 GPU speedup versus two 6-core Westmere CPUs is roughly $5\times$ in single precision, $3\times$ in double precision
- currently working on MPI layer in OP2 for computing on GPU clusters
- key challenge then is to build user community

Acknowledgements

- Carlo Bertolli, David Ham, Paul Kelly, Graham Markall and others (Imperial College)
- Nick Hills (Surrey) and Paul Crumpton (original OPlus development)
- Yoon Ho, Leigh Lapworth, David Radford (Rolls-Royce) Jamil Appa, Pierre Moinier (BAE Systems)
- Tom Bradley, Jon Cohen and others (NVIDIA)
- EPSRC, TSB, NVIDIA and Rolls-Royce for financial support
- Oxford Supercomputing Centre

2023 update

- OP2 continues, funded by Rolls-Royce and EPSRC
- subsequent OPS development supported calculations on unstructured collections of structured blocks
- project is now being led by Gihan Mudalige (Warwick) and Istvan Reguly (PPCU in Budapest)
- primary codes using OP2 are
  - HYDRA – Rolls-Royce's primary CFD code
  - VOLNA – tsunami simulation code
- a huge HYDRA calculation and visualisation won an award at Supercomputing 2022
- main project webpage is
  `https://op-dsl.github.io/`