OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures

Mike Giles, Gihan Mudalige, István Reguly
Carlo Bertolli, Paul Kelly

Oxford e-Research Centre / Imperial College

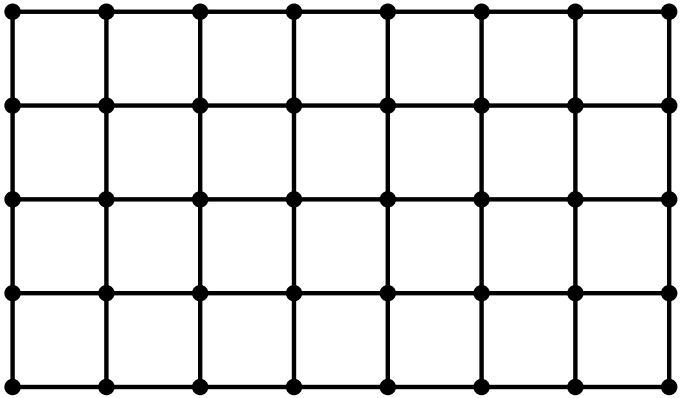
InPar 2012: Innovative Parallel Computing

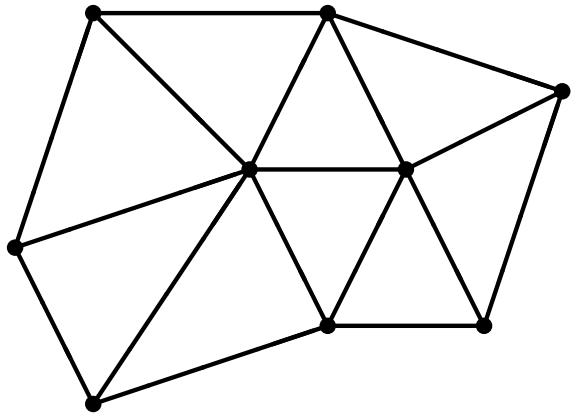
May 13, 2012

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 and unstructured grids





Structured grids:

- logical (i, j) indexing in 2D, (i, j, k) in 3D, with implicit connectivity
- easy to parallelize, including on GPUs with L1/L2 caches

Unstructured grids:

- a collection of nodes, edges, etc., with explicit connectivity – e.g. mapping tables define connections from edges to nodes
- much harder to parallelize (not in concept so much as in practice) but a lot of existing literature on the subject

Software Challenge

- Application developers want the benefits of the latest hardware but are very worried about the development effort required
- 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't afford to keep changing their codes

Solution?

- high-level abstraction to separate the user's specification of the application from the details of the parallel implementation
- aim to achieve application level longevity together with near-optimal performance through re-targetting the back-end implementation

OP2

- open source project
- based on OPlus (Oxford Parallel Library for Unstructured Solvers) developed over 10 years ago for industrial CFD code on distributed-memory clusters
- supports application codes written in C++ or FORTRAN
- looks like a conventional library, but uses code transformation to generate CUDA for NVIDIA GPUs and OpenMP/AVX for CPUs/MIC
- keeps OPlus abstraction, but slightly modifies API

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)

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, 0,col,1,"float",OP_READ),
  op_arg_dat(du,0,row,1,"float",OP_INC));
```

This is equivalent to the C code:

```
for (e=0; e<nedges; e++)
    du[row[e]] += A[e] * u[col[e]];
```

where each "edge" corresponds to a non-zero element in the matrix `A`, and `row`, `col` give the corresponding row and column indices.

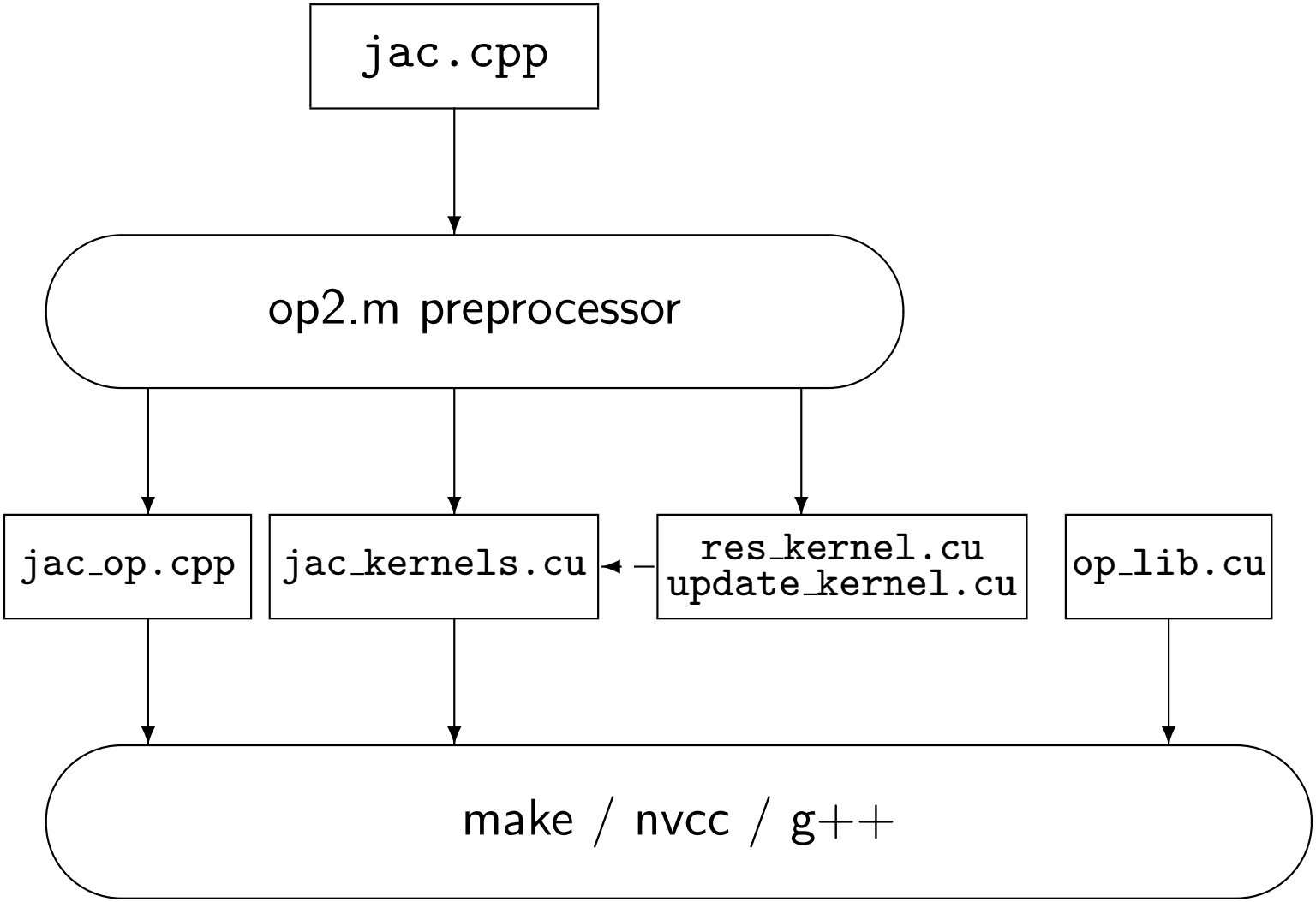
User build processes

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

- sequential single-thread CPU execution
  - no code generation – just uses a header file
  - purely for program development and debugging
- CUDA (and OpenCL in the future) for single GPU
- OpenMP (and AVX in the future) for multicore CPU systems
- MPI plus any of the above for clusters

CUDA build process

Preprocessor parses user code and generates new code:



Implementation Approach

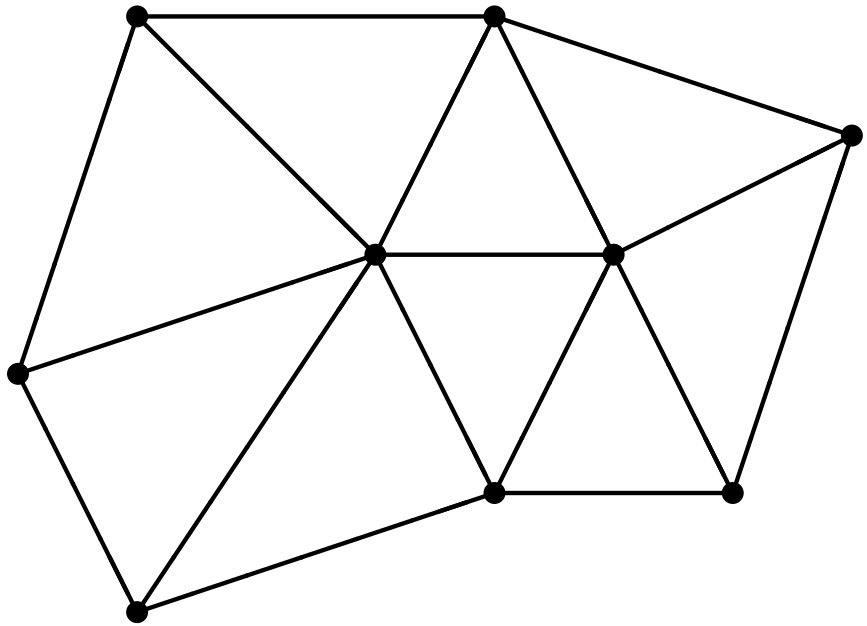
The question now is how to deliver good performance on multiple GPUs

- MPI distributed-memory parallelism (1-100)
  - one MPI process for each GPU, with standard partitioning so that each partition fits within global memory of GPU
  - only halos need to be transferred from one GPU to another
- block parallelism (100-2000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different Streaming Multiprocessors 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
- thread parallelism (64-256)
  - each mini-partition is worked on by a block of threads in parallel

Data dependencies

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

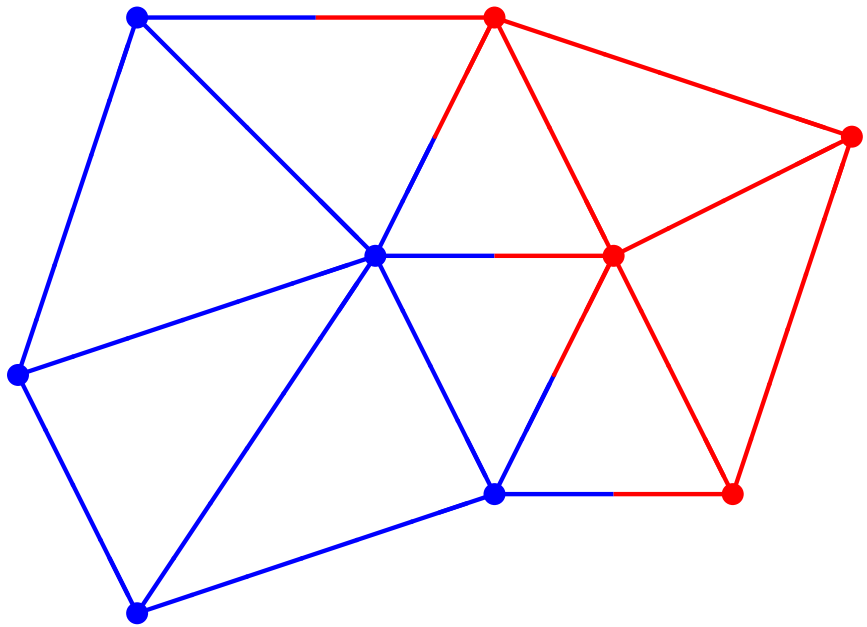
e.g. potential problem when two edges update same node



Data dependencies

MPI level: "owner" of nodal data does edge computation

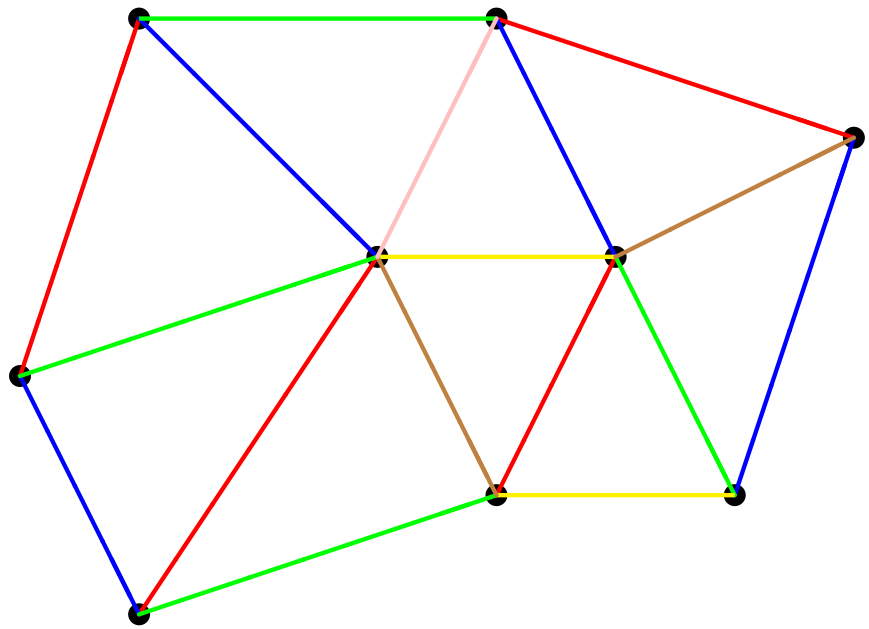
- drawback is redundant computation when the two nodes have different "owners"



Data dependencies

Thread level: "color" edges so no two edges of the same color update the same node

- compute increments in parallel, then apply them color by color with synchronisation between
- similar strategy also used at thread block level to avoid race condition



Other implementation issues

- array-of-structs storage preferred to struct-of-arrays
  - better cache hits for indirect addressing
  - transfers between graphics memory and GPU still largely "coalesced"
- auto-tuning very useful to optimize size of partitions and number of threads

Airfoil test code

- 2D Euler equations, cell-centred finite volume method with scalar dissipation
- two test cases:
  - 1.5M edges, 0.75M cells
  - 15M edges, 7.5M 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× in single precision, 3× in double precision
- currently working on MPI layer in OP2 for computing on GPU clusters
- key challenge then is to build user community