Performance experiments on BOUT++

Praveen Narayanan and Alice Koniges

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

Why Analyze Performance?

- Improving performance on HPC systems has compelling economic and scientific rationales.
  - Dave Bailey: Value of improving performance of a single application, 5% of machine's cycles by 20% over 10 years: $1,500,000
  - Scientific benefit probably much higher
- Goal: solve problems faster; solve larger problems
- Accurately state computational need
- Only that which can be measured can be improved
- The challenge is mapping the application to an increasingly more complex system architecture
  - or set of architectures

Performance Analysis Issues

- Difficult process for real codes
- Many ways of measuring, reporting
- Very broad space: Not just time on one size
  - for fixed size problem (same memory per processor): Strong Scaling
  - scaled up problem (fixed execution time): Weak Scaling
- A variety of pitfalls abound
  - Must compare parallel performance to best *uniprocessor* algorithm, not just parallel program on 1 processor (unless it's best)
  - Be careful relying on any single number
- Amdahl's Law

Performance Questions

- How can we tell if a program is performing well?
- Or isn't?
- If performance is not "good", how can we pinpoint why?
- How can we identify the causes?
- What can we do about it?

The multicore era

- Moore's law still extant
- Traditional sources of performance improvement ending
  - Old trend: double clock frequency every 18 months
  - New trend double #cores every 18 months
  - Implications: flops cheap, communication, network bandwidth expensive in future

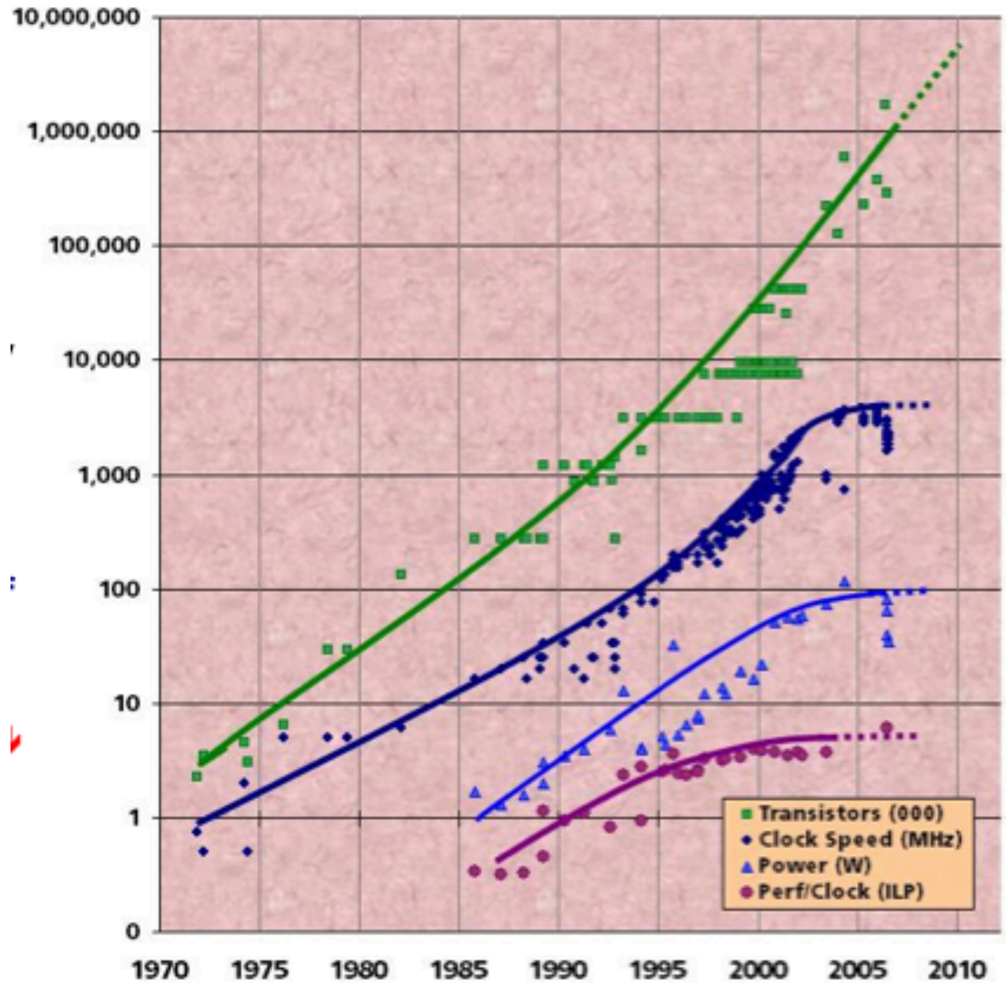
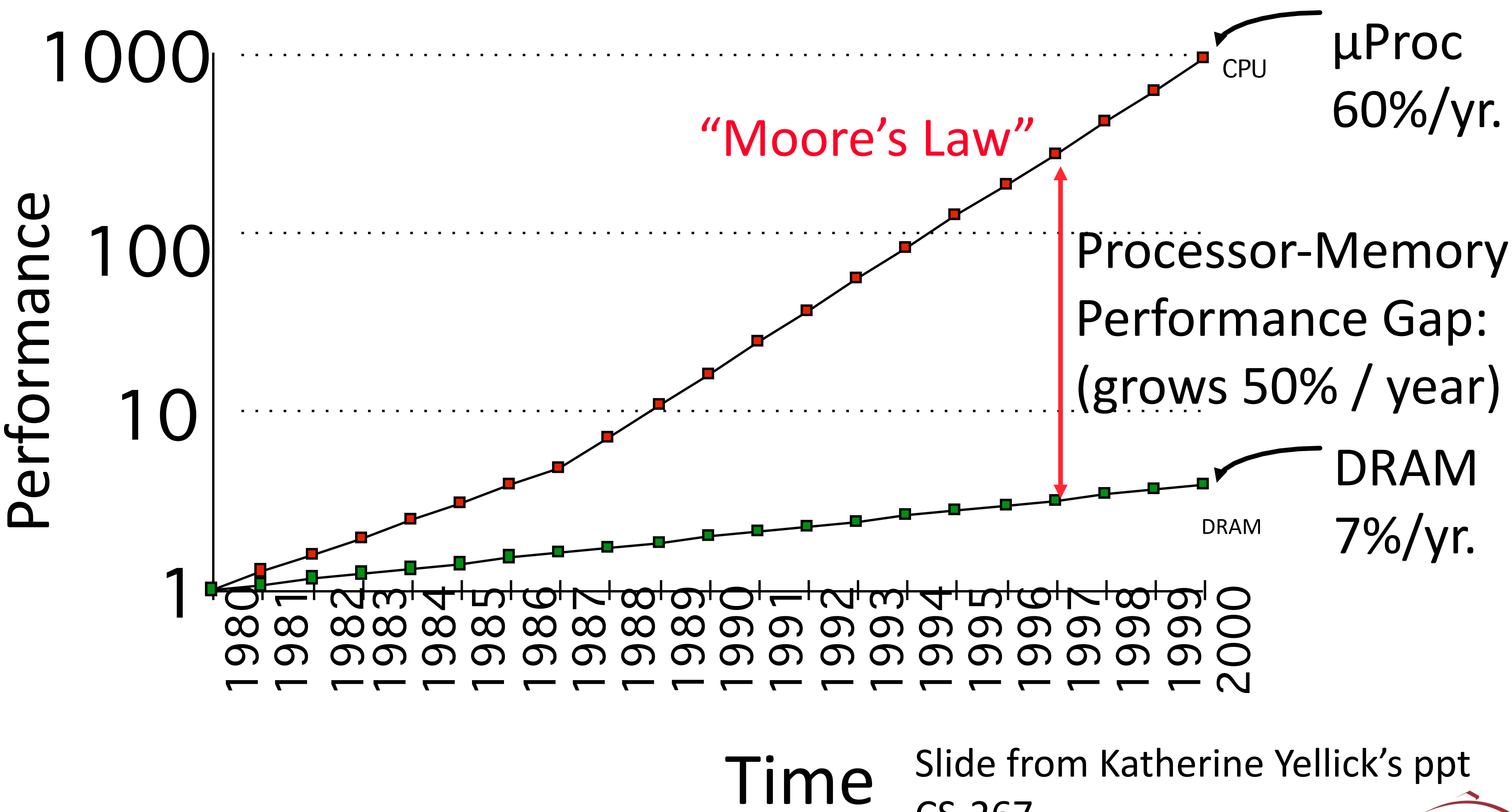


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Processor-DRAM Gap (latency)

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory



Slide from Katherine Yellick’s ppt CS-267

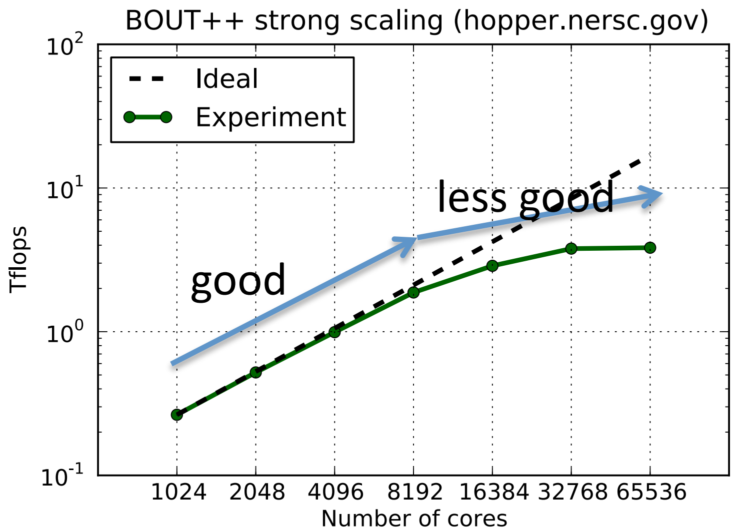
Performance analysis tools

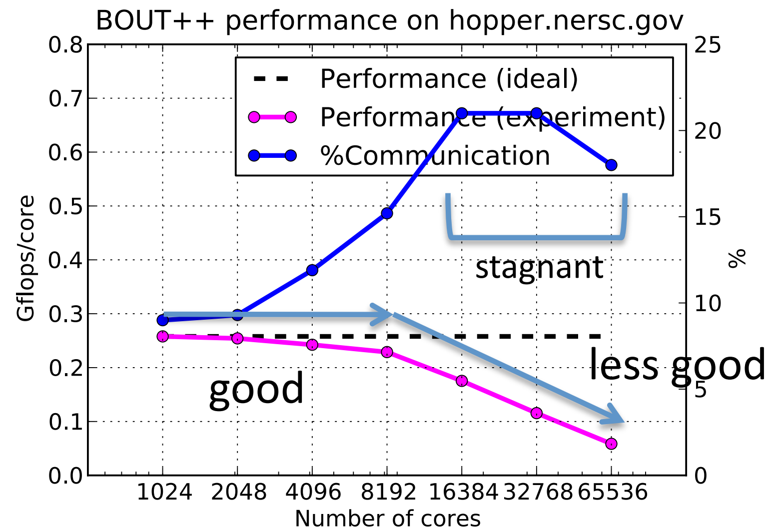
- Use of profilers to measure performance
- Approach
    - Build and instrument code with binary to monitor function groups of interest
        - MPI, OpenMP, PGAS, HWC, IO
    - Run instrumented binary
    - Identify performance bottlenecks
    - Suggest (and possibly implement) improvements to optimize code
- Tools used
    - IPM: Low overhead, communication, flops, code regions
    - CrayPAT: Communication, flops, code regions, PGAS, variety of tracing options
- Overhead depends on number of trace groups monitored
- Level of detail in study depends on specifics: time available, difficulties presented by code

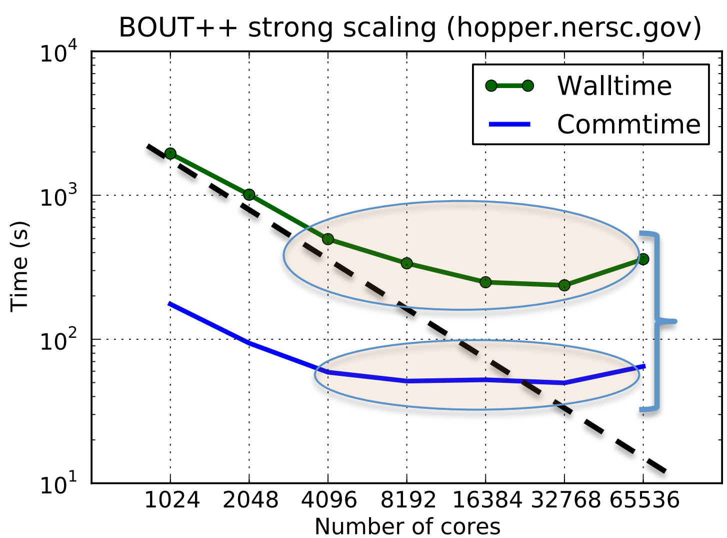
Performance checklist

- Scaling
  - Application time, speed (flop/s)
  - Double concurrency, does speed double?
- Communication (vs computation)
- Load imbalance
  - Check cabinet for mammoths and mosquitoes
- Size and composition of communication
  - Bandwidth bound?
  - Latency bound?
  - Collectives (large concurrency)
- Memory bandwidth sensitivity

BOUT++ scaling results (Elm-pb)







Does not scale
But …
Communication
Does not increase

Runtime increases because of other reasons
Performance decreases because of other reasons

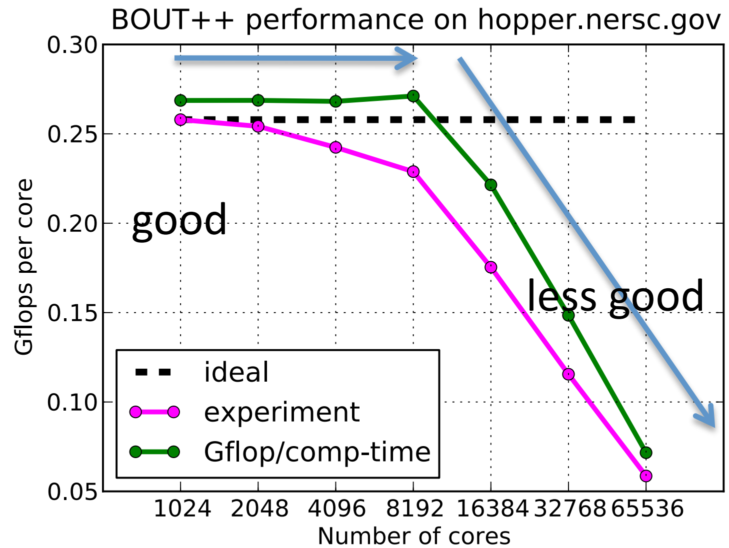
Grid used:
nx=2052 ny=512

Set: nxpe=256

Communication not reason for performance degradation

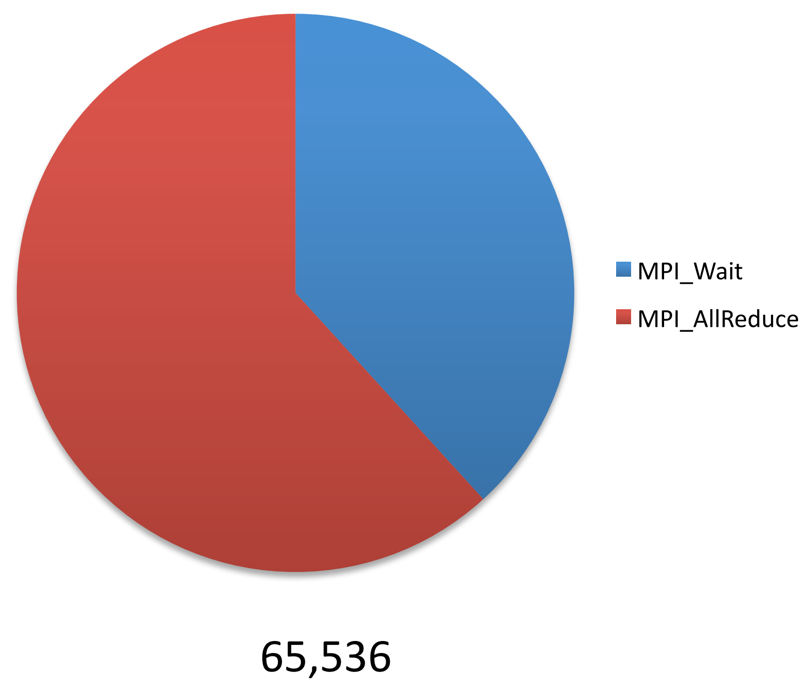
- Separate out communication portion from walltime and compute

  speed=Flop/(computation time-core)
- Should be constant for ideal scaling, if comm were reason for performance degradation



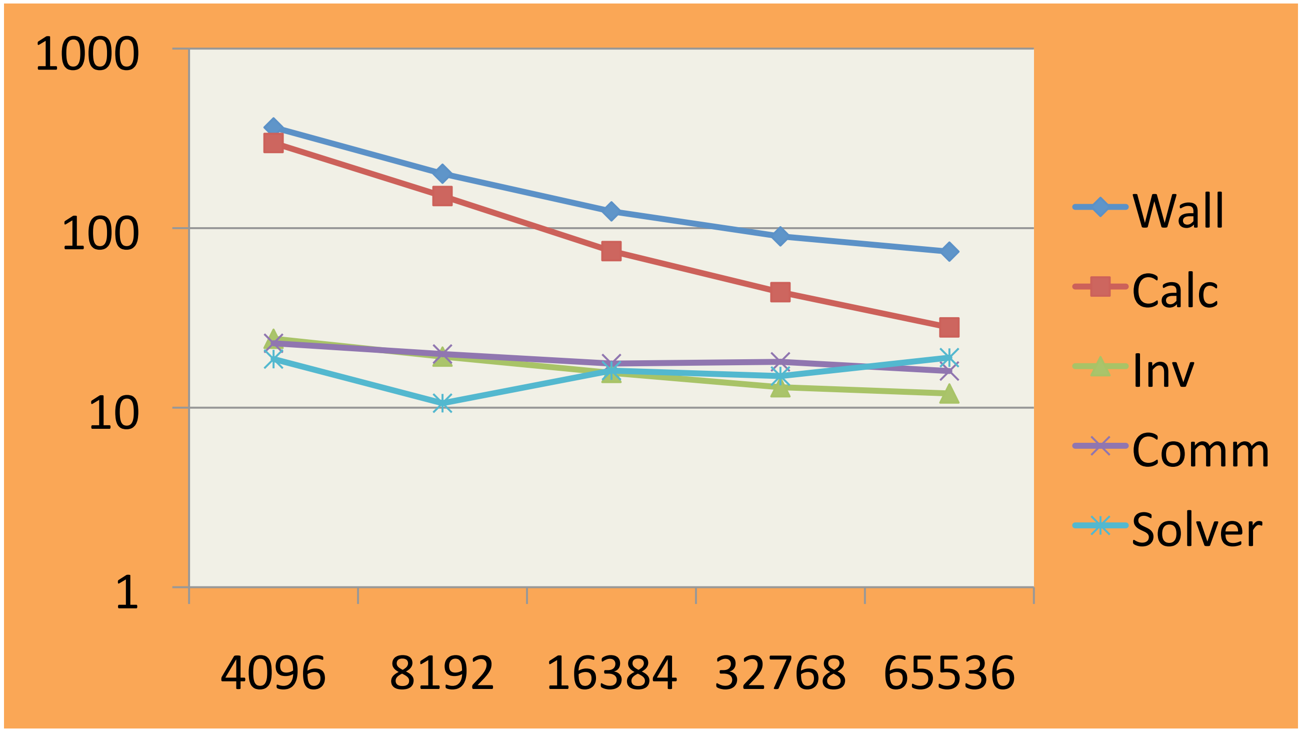
MPI pie shows significance of Allreduce calls

- MPI_Allreduce calls form bulk of pie
- MPI average message size 5 kB, 74,000 calls
- Not quite entirely latency bound on hopper (5 kB should be large enough)
- Might become bottleneck after other issues are sorted out
- (communication not yet a bottleneck)



Bout++: break up times in each kernel to check how they scale

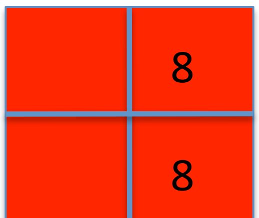
- Breakup by time spent: Calc scales somewhat, but inv, solver do not scale



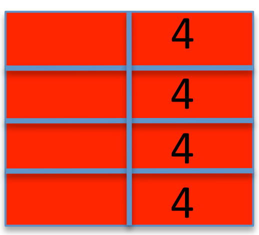
Bout++ (elm) scaling summary

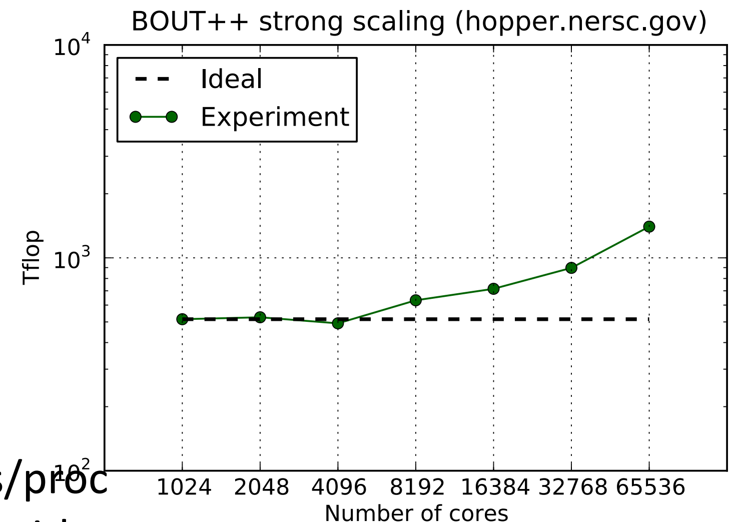
- Up to concurrency 8,192, code scales nearly perfectly.
- Two issues beyond 8,192
  - Performance decreases (flop/time decreases)
  - Wall time increases
  - MPI not the reason for performance degradation
  - Computational performance decreases

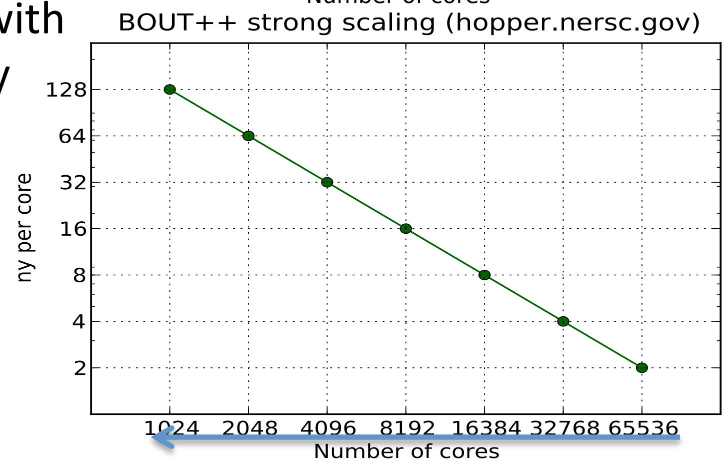
Issues with increasing flop count



Grid points/proc
Decreases with
concurrency







Increaseasing concurrency

- Steady increase in flop count
  (number of operations)

Conjecture: Extra computations in ghost cells (and more cycles spent in doing these)

Valid region (excluding ghost region) does same amount of work

BOUT++: Expt-LAPD-drift

- Experiment: turbulence in an annulus (LAPD)
- Investigate source of extra computations in code [more work done – leads to greater flop's (not 'flop/s') count]
- Code annotated to give flop count with CrayPAT
- A given portion is annotated and flop count in that code section is compared across concurrency
- Conjecture: increase in flops in this section because of ghost cell computations (arrays consist of valid+ghost regions)

Annotated code region

```
PAT_region_begin(24,
  "phys_run-14");
  nu        = nu_hat * Nit /
   (Tet^1.5);
  mu_i      = mui_hat *
   (Tit^2.5)/Nit;
  kapa_Te = 3.2*(1./fmei)*
   (wci/nueix)*(Tet^2.5);
  kapa_Ti = 3.9*(wci/
   nuiix)*(Tit^2.5);

  // Calculate pressures
  pei = (Tet+Tit)*Nit;
  pe  = Tet*Nit;
  PAT_region_end(24);
```

- Quantities Tet, Tit, Nit, etc declared as follows

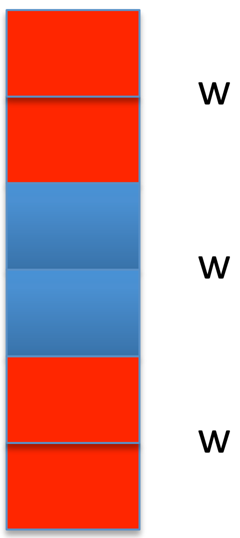
```
// 3D evolving fields
Field3D rho, ni, ajpar, te;

// Derived 3D variables
Field3D phi, Apar, Ve, jpar;

// Non-linear coefficients
Field3D nu, mu_i, kapa_Te, kapa_Ti;

// 3D total values
Field3D Nit, Tit, Tet, Vit, phit, VEt, ajp0;

// pressures
Field3D pei, pe;
Field2D pei0, pe0;
```

Variables defined to comprise valid region +ghost cells

Extra Computation in box can be measured

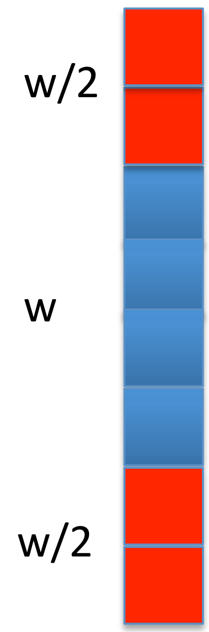
Computation in guard cells

- Grid: 204x128, ghost cells: MXG, MYG=2



6400 (extreme-2 inner grid pts)

Extreme case 3 times the work as expected!



3200 (extreme but one)

Twice as much work as expected

Observations: validation of ghost cell conjecture

Concurrency	FPO in given region from CrayPat	Factor predicted	Actual
6400	51512160000	1(reference case)	--
3200	34341360000	2/3	2/3
1600	25755960000	½	½
800	21463260000	1.25/3	1.25/3

Predicted values match exactly with computed flops, in terms of ratios
Hypothesis seems to be correct

Kernels: Calc scales, affected by ghost cells

- Breakup by time spent

Concurrency	wall	Calc	Inv	Comm	Solver
4096	364	298	24	23	19
8192	201	151	19	20	11
16384	124	74	16	18	16
16384	197	136	17	30	13

Does not scale

Scales

2	8	2

16384 (1^{st} expt)

8	8	8

16384 (2^{nd} expt)

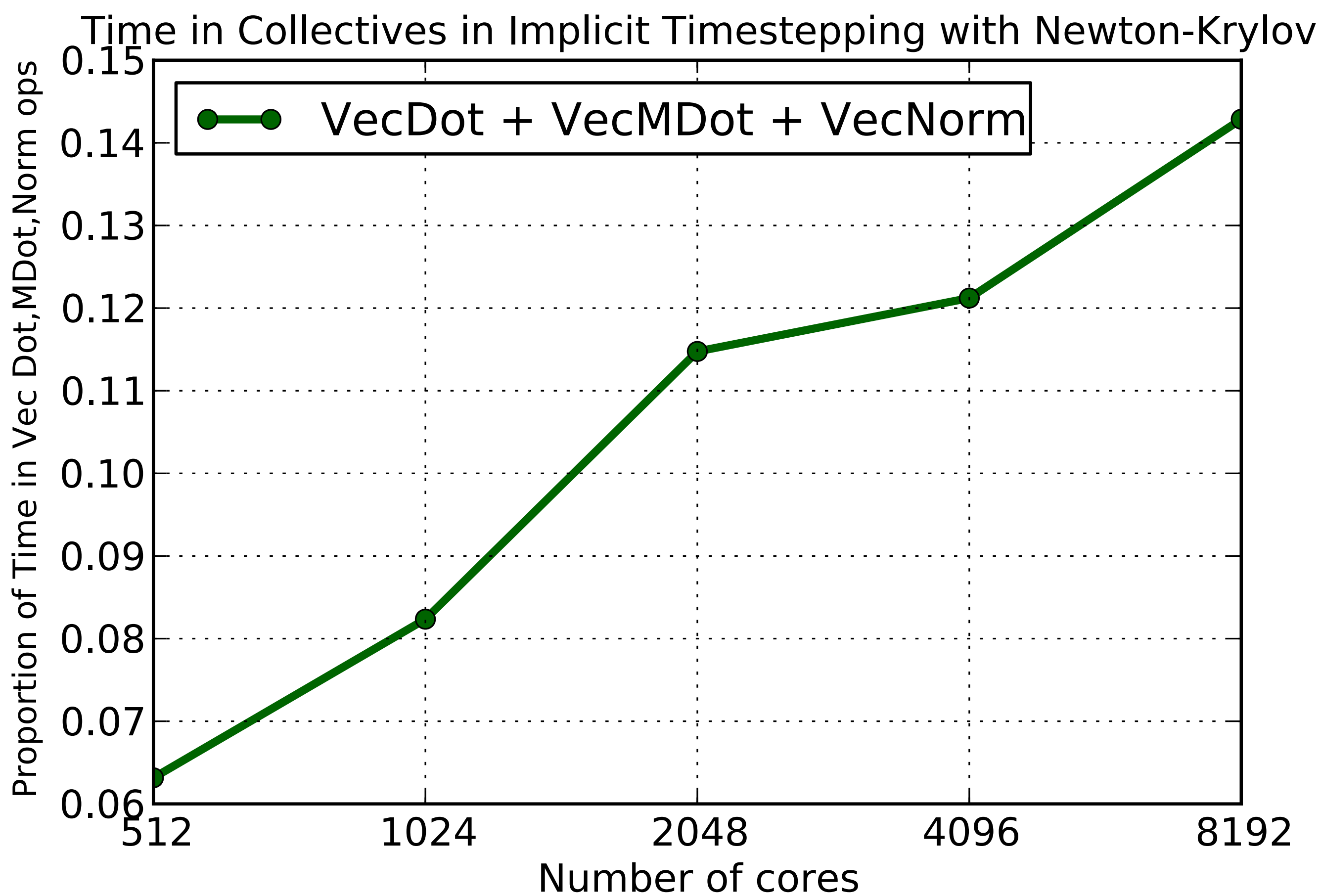
BOUT++ results: improve INV, PVODE

- Summary
  - Scaling degrades beyond 8192 procs
  - Scaling efficiency is very poor at 32768 procs
  - Issues
    - Extra computations in ghost cells
      - » Put in place to reduce communication
      - » Need to check if extra computations performed are worth it
      - » Performance degrades because
        - Laplacian inversion does not scale
        - Pvode solver does not scale
        - MPI collectives
    - Surface to volume ratio tested may not be best but issues remain
  - MPI: Collectives grow with concurrency, but Laplacian inversion and PVODE solver seem to be culpable in equal measure
- Recommendations: need to check if replacing ghost cell computation with communication improves runtime (or not)
- **Need to improve Laplacian inversion and PVODE kernels**

Investigation of collectives in time-stepping algorithms (PETSc)

- Time-stepping algorithms suspected to have collectives
  - First step: check growth of collectives in time-steppers
  - Hook with PETSc and turn on profiling layer
    - -log_summary
  - Examine %collectives vis a vis runtime

PETSc



Time in Collectives in Implicit Timestepping with Newton-Krylov
VecDot + VecMDot + VecNorm
Proportion of Time in Vec Dot,MDot,Norm ops
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06
512
1024
2048
4096
8192
Number of cores

Overall conclusions and future directions(?)

- BOUT++ scales remarkably well for a strong scaling test
  - Performance degradation ‘not just’ because of increase in surface to volume ratio
- Communication increases at higher concurrency, could constitute the ultimate scaling bottleneck
- Extra ghost cell computations
  - Put in there for a reason, to lessen communication, but manifests as extra time spent in computation
  - Might be good overhead when flops become cheaper
- Bandwidth sensitivity not an issue in BOUT++
- Two dimensional domain decomposition
  - Possibly add OpenMP in third direction?
- Collectives might play dominant role in time-steppers
  - Find ways of minimizing this
  - What is the effect of putting in preconditioners?