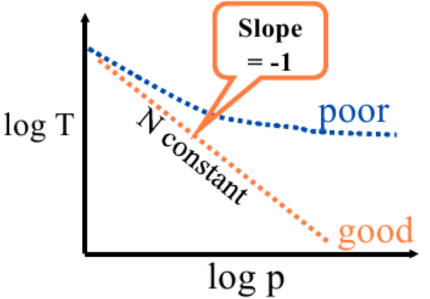
Scalability definitions

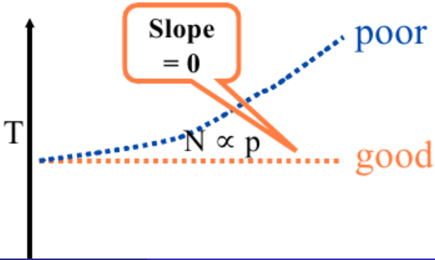
Strong scalability

- Fixed problem size
- execution time T inversely proportional to number of processors p



Weak scalability

- Fixed problem size per processor
- execution time constant as problem size increases



Scalability Warning

The easiest way to make software scalable is to make it sequentially inefficient.
(Gropp 1999)

- We really want <u>efficient</u> software
- Need a performance model
  - memory bandwidth and latency
  - algorithmically critical operations (e.g. dot products, scatters)
  - floating point unit
- Scalability shows marginal benefit of adding more cores, nothing more
- Constants hidden in the choice of algorithm
- Constants hidden in implementation

Limits of “scalability”?

- **Transient simulation does not weak scale.**
  - Fixed turn-around needed: policy, manufacturing/supply-chain, active control, real-time guidance (field work, surgery, etc.)
  - d-dimensional problem, increase resolution by $2\times$.
  - Data increases by 2^d, but we need $2\times$ more time steps (hyperbolic).
  - With perfect scaling, we use 2^{d+1} more cores.
  - Local data changes by $2^d/2^{d+1} = \frac{1}{2}$
- More applications feeling this
  - Asymptotics are relentless
  - New analysis requires more solves in sequence
    - From forward simulation to optimization with uncertainty ...
  - New physics and higher fidelity observation requires more calibration/validation
- Other applications are safe for now
  - Steady-state solves with scalable methods
  - Transient with a small number of time steps
  - Maximize resolution/problem size – memory-constrained

Evaluating methods

- Performance of methods will depend on grid resolution and model parameters (regime and heterogeneity).
- A method is:
  - scalable (also “optimal”) if its performance is independent of resolution and parallelism
  - robust if its performance is (nearly) independent of model parameters
  - efficient if it solves the problem in a small multiple of the cost to evaluate the residual[1]
- Linear problems typically arise from linearizing a nonlinear problem. This step is not necessary, but it is convenient for reusing software and for debugging.

[1] We’ll settle for “as fast as the best known method”.

Evaluating methods

- Performance of methods will depend on grid resolution and model parameters (regime and heterogeneity).
- A method is:
  - scalable (also “optimal”) if its performance is independent of resolution and parallelism
  - robust if its performance is (nearly) independent of model parameters
  - efficient if it solves the problem in a small multiple of the cost to evaluate the residual[1]
- Linear problems typically arise from linearizing a nonlinear problem. This step is not necessary, but it is convenient for reusing software and for debugging.

[1] We’ll settle for “as fast as the best known method”.

Importance of Computational Modeling

**Without a model,
performance measurements are meaningless!**

Before a code is written, we should have a model of

- computation
- memory usage
- communication
- bandwidth
- achievable concurrency

This allows us to

- verify the implementation
- predict scaling behavior

Complexity Analysis

The key performance indicator, which we will call the *balance factor* β, is the ratio of flops executed to bytes transfered.

- We will designate the unit $\frac{\text{flop}}{\text{byte}}$ as the *Keyes*
- Using the peak flop rate r_{peak}, we can get the required bandwidth B_{req} for an algorithm

$$B_{\text{req}} = \frac{r_{\text{peak}}}{\beta} \qquad (1)$$

- Using the peak bandwidth B_{peak}, we can get the maximum flop rate r_{max} for an algorithm

$$r_{\text{max}} = \beta B_{\text{peak}} \qquad (2)$$

STREAM Benchmark

Simple benchmark program measuring sustainable memory bandwidth

- Protoypical operation is Triad (WAXPY): $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$
- Measures the memory bandwidth bottleneck (much below peak)
- Datasets outstrip cache

Machine	Peak (MF/s)	Triad (MB/s)	MF/MW	Eq. MF/s
Matt's Laptop	1700	1122.4	12.1	93.5 (5.5%)
Intel Core2 Quad	38400	5312.0	57.8	442.7 (1.2%)
Tesla 1060C	984000	102000.0*	77.2	8500.0 (0.8%)

Table: Bandwidth limited machine performance

http://www.cs.virginia.edu/stream/

Sparse Mat-Vec performance model

Compressed Sparse Row format (AIJ)

For $m \times n$ matrix with N nonzeros

- **ai** row starts, length $m + 1$
- **aj** column indices, length N, range $[0, n-1)$
- **aa** nonzero entries, length N, scalar values

$$y \leftarrow y + Ax$$

```
for (i=0; i<m; i++)
  for (j=ai[i]; j<ai[i+1]; j++)
    y[i] += aa[j] * x[aj[j]];
```

- One add and one multiply per inner loop
- Scalar `aa[j]` and integer `aj[j]` only used once
- Must load `aj[j]` to read from `x`, may not reuse cache well

Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

- m Number of matrix rows
- nz Number of nonzero matrix elements
- V Number of vectors to multiply

We can look at bandwidth needed for peak performance

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V}\ \mathrm{byte/flop} \tag{3}$$

or achieveable performance given a bandwith BW

$$\frac{Vnz}{(8V+2)m + 6nz} BW\ \mathrm{Mflop/s} \tag{4}$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.

Performance Caveats

- The peak flop rate r_{peak} on modern CPUs is attained through the usage of a SIMD multiply-accumulate instruction on special 128-bit registers.
- SIMD MAC operates in the form of 4 simultaneous operations (2 adds and 2 multiplies):

$$c_1 = c_1 + a_1 * b_1 \tag{5}$$
$$c_2 = c_2 + a_2 * b_2 \tag{6}$$

  You will miss peak by the corresponding number of operations you are missing. In the worst case, you are reduced to 25% efficiency if your algorithm performs naive summation or products.
- Memory alignment is also crucial when using SSE, the instructions used to load and store from the 128-bit registers throw very costly alignment exceptions when the data is not stored in memory on 16 byte (128 bit) boundaries.

Profiling basics

- Get the math right
  - Choose an algorithm that gives robust iteration counts and really converges
- Look at where the time is spent
  - Run with `-log_summary` and look at events
  - `VecNorm`, `VecDot` measures latency
  - `MatMult` measures neighbor exchange and memory bandwidth
  - `PCSetUp` factorization, aggregation, matrix-matrix products, ...
  - `PCApply` V-cycles, triangular solves, ...
  - `KSPSolve` linear solve
  - `SNESFunctionEval` residual evaluation (user code)
  - `SNESJacobianEval` matrix assembly (user code)

Communication Costs

- Reductions: usually part of Krylov method, latency limited
  - `VecDot`
  - `VecMDot`
  - `VecNorm`
  - `MatAssemblyBegin`
  - Change algorithm (e.g. IBCGS)
- Point-to-point (nearest neighbor), latency or bandwidth
  - `VecScatter`
  - `MatMult`
  - `PCApply`
  - `MatAssembly`
  - `SNESFunctionEval`
  - `SNESJacobianEval`
  - Compute subdomain boundary fluxes redundantly
  - Ghost exchange for all fields at once
  - Better partition

Performance Debugging

- PETSc has integrated profiling
  - Option `-log_summary` prints a report on `PetscFinalize()`
- PETSc allows user-defined events
  - Events report time, calls, flops, communication, etc.
  - Memory usage is tracked by object
- Profiling is separated into stages
  - Event statistics are aggregated by stage

Profiling

- Use `-log_summary` for a performance profile
  - Event timing
  - Event flops
  - Memory usage
  - MPI messages
- Call `PetscLogStagePush()` and `PetscLogStagePop()`
  - User can add new stages
- Call `PetscLogEventBegin()` and `PetscLogEventEnd()`
  - User can add new events
- Call `PetscLogFlops()` to include your flops

Reading -log_summary

-
```
                          Max       Max/Min        Avg      Total
Time (sec):           1.548e+02   1.00122   1.547e+02
Objects:              1.028e+03   1.00000   1.028e+03
Flops:                1.519e+10   1.01953   1.505e+10  1.204e+11
Flops/sec:            9.814e+07   1.01829   9.727e+07  7.782e+08
MPI Messages:         8.854e+03   1.00556   8.819e+03  7.055e+04
MPI Message Lengths:  1.936e+08   1.00950   2.185e+04  1.541e+09
MPI Reductions:       2.799e+03   1.00000
```
- Also a summary per stage
- Memory usage per stage (based on when it was allocated)
- Time, messages, reductions, balance, flops per event per stage
- Always send `-log_summary` when asking performance questions on mailing list

Reading -log_summary

```
Event                Count      Time (sec)     Flops                             --- Global ---  --
                   Max Ratio  Max        Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %
------------------------------------------------------------------------------------------------------
--- Event Stage 1: Full solve
VecDot                43 1.0 4.8879e-02 8.3 1.77e+06 1.0 0.0e+00 0.0e+00 4.3e+01  0  0  0  0  0
VecMDot             1747 1.0 1.3021e+00 4.6 8.16e+07 1.0 0.0e+00 0.0e+00 1.7e+03  0  1  0  0 14
VecNorm             3972 1.0 1.5460e+00 2.5 8.48e+07 1.0 0.0e+00 0.0e+00 4.0e+03  0  1  0  0 31
VecScale            3261 1.0 1.6703e-01 1.0 3.38e+07 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
VecScatterBegin     4503 1.0 4.0440e-01 1.0 0.00e+00 0.0 6.1e+07 2.0e+03 0.0e+00  0  0 50 26  0
VecScatterEnd       4503 1.0 2.8207e+00 6.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
MatMult             3001 1.0 3.2634e+01 1.1 3.68e+09 1.1 4.9e+07 2.3e+03 0.0e+00 11 22 40 24  0  2
MatMultAdd           604 1.0 6.0195e-01 1.0 5.66e+07 1.0 3.7e+06 1.3e+02 0.0e+00  0  0  3  0  0
MatMultTranspose     676 1.0 1.3220e+00 1.6 6.50e+07 1.0 4.2e+06 1.4e+02 0.0e+00  0  0  3  0  0
MatSolve            3020 1.0 2.5957e+01 1.0 3.25e+09 1.0 0.0e+00 0.0e+00 0.0e+00  9 21  0  0  0  1
MatCholFctrSym         3 1.0 2.8324e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
MatCholFctrNum        69 1.0 5.7241e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 0.0e+00  2  4  0  0  0
MatAssemblyBegin     119 1.0 2.8250e+00 1.5 0.00e+00 0.0 2.1e+06 5.4e+04 3.1e+02  1  0  2 24  2
MatAssemblyEnd       119 1.0 1.9689e+00 1.4 0.00e+00 0.0 2.8e+05 1.3e+03 6.8e+01  1  0  0  0  1
SNESSolve              4 1.0 1.4302e+02 1.0 8.11e+09 1.0 6.3e+07 3.8e+03 6.3e+03 51 50 52 50 50  9
SNESLineSearch        43 1.0 1.5116e+01 1.0 1.05e+08 1.1 2.4e+06 3.6e+03 1.8e+02  5  1  2  2  1  1
SNESFunctionEval      55 1.0 1.4930e+01 1.0 0.00e+00 0.0 1.8e+06 3.3e+03 8.0e+00  5  0  1  1  0  1
SNESJacobianEval      43 1.0 3.7077e+01 1.0 7.77e+06 1.0 4.3e+06 2.6e+04 3.0e+02 13  0  4 24  2  2
KSPGMRESOrthog      1747 1.0 1.5737e+00 2.9 1.63e+08 1.0 0.0e+00 0.0e+00 1.7e+03  1  1  0  0 14
KSPSetup             224 1.0 2.1040e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 3.0e+01  0  0  0  0  0
KSPSolve              43 1.0 8.9988e+01 1.0 7.99e+09 1.0 5.6e+07 2.0e+03 5.8e+03 32 49 46 24 46  6
PCSetUp              112 1.0 1.7354e+01 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  6  4  0  0  1  1
PCSetUpOnBlocks     1208 1.0 5.8182e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  2  4  0  0  1
PCApply              276 1.0 7.1497e+01 1.0 7.14e+09 1.0 5.2e+07 1.8e+03 5.1e+03 25 44 42 20 41  4
```

Adding A Logging Class

```
static int CLASS_ID;

PetscLogClassRegister(&CLASS_ID, "name");
```

- Class ID identifies a class uniquely
- Must initialize before creating any objects of this type

Adding A Logging Event
C

```
static int USER_EVENT;

PetscLogEventRegister(&USER_EVENT, "name", CLS_ID);
PetscLogEventBegin(USER_EVENT,0,0,0,0);

/* Code to Monitor */

PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

Adding A Logging Event

Python

```python
with PETSc.logEvent('Reconstruction') as recEvent:
  # All operations are timed in recEvent
  reconstruct(sol)
  # Flops are logged to recEvent
  PETSc.Log.logFlops(user_event_flops)
```

Adding A Logging Stage
C

```
int stageNum;

PetscLogStageRegister(&stageNum, "name");
PetscLogStagePush(stageNum);

/* Code to Monitor */

PetscLogStagePop();
```