Optimization and Scalability

Drew Dolgert
CAC
29 May 2009

Intro to Parallel Computing

Great Little Program

- What happens when I run it on the cluster?
- How can I make it faster?
- Can I run it on 40 nodes, 4000 nodes?

Lots of Things Contribute To Finishing Your Work

- Well-posed model for the system.
- Choosing among algorithms that express that model.
- Implementation of that algorithm in code.
- Compilation of the code.
- Runtime environment.

Realistic Concerns

- Do you have time to make it parallel?
- Do you have the time to rewrite in a faster language?
- Do you have compute hours to burn, or do they cost a lot?
- Do you have to understand the code and use it again?

Use Libraries

- Optimized for specific architectures
- Much faster than hand-coding your own, even from NR
- Offered by different vendors (ESSL/PESSL on IBM systems, Intel MKL for IA32, EM64T and IA64, Cray libsci for Cray systems, SCSL for SGI)

Libraries on Ranger

Performance	Math Libs	Method Libs	Applications	I/O
gprof	fftw	petsc	Amber	netcdf
tau	GotoBLAS	scalapack	NAMD	hdf5
papi	Metis/parmetis		charm++	
	MKL 10.0		Gamess	
	Gnu Scientific Library			

Intel MKL 10.0

- Basic Linear Algebra Subroutines, such as ax+y
- LAPACK
- FFT
- All highly optimized
- Call from C, Fortran, other languages
- Module load mkl
- mpicc –l$TACC_MKL_INC –l$tacc_mkl_lib –LMKL_em64t

GotoBLAS

- Hand-optimized BLAS
- Test to see what kind of advantage your code gets.
- Minimizes TLB misses.

Fastest Fourier Transform in the West

- Cooley-Tukey algorithm
- Prime Factor algorithm {most efficient with small prime factors (2,3,5, and 7)}
- Rader's algorithm for prime sizes
- split-radix algorithm (with a variation due to Dan Bernstein)
- automatic performance adaptation

PETSc

- PETSc, the **P**ortable, **E**xtensible **T**oolkit for **S**cientific **c**omputation, provides sets of tools for the parallel (as well as serial), numerical solution of PDEs that require solving large-scale, sparse nonlinear systems of equations. PETSc includes nonlinear and linear equation solvers that employ a variety of Newton techniques and Krylov subspace methods.

PETSc

- Parallel vectors
  - scatters (handles communicating ghost point information)
  - gathers
- Parallel matrices
  - several sparse storage formats
  - easy, efficient assembly.
- Scalable parallel preconditioners
- Krylov subspace methods
- Parallel Newton-based nonlinear solvers
- Parallel timestepping (ODE) solvers

- http://acts.nersc.gov/petsc/
- http://www-unix.mcs.anl.gov/petsc/petsc-as

Misc Mathematical Libraries

- dense and band matrix software (**ScaLAPACK**)
- http://www.netlib.org/scalapack/
- large sparse eigenvalue software (**PARPACK** and **ARPACK**)
  http://www.caam.rice.edu/software/ARPACK/

Gnu Scientific Library

- Complex Numbers, Roots of Polynomials
- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- BLAS Support
- Linear Algebra
- Eigensystems
- Fast Fourier Transforms
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions

GNU Scientific Library cont.

- Statistics
- Histograms
- N-Tuples
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation

GNU Scientific Library cont.

- Series Acceleration
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- Physical Constants
- IEEE Floating-Point
- Discrete Wavelet Transforms

- http://www.gnu.org/software/gsl/

Compilation Optimization Levels

- -O0 no optimization: Fast compilation, disables optimization
- -O2 low to moderate optimization: partial debugging support, disables inlining
- -O3 aggressive optimization: compile time/space intensive and/or marginal effectiveness; may change code semantics and *results* (sometimes even breaks codes!)

A cycle of what?

Measuring Division

Compiler Option	#cycles per iteration
None	30.0
-O2	15.7
-O3 –qhot	12.7

What the Compiler Does for You

- Operations performed at moderate optimization levels
  - instruction rescheduling
  - copy propagation
  - software pipelining
  - common subexpression elimination
  - prefetching, loop transformations
- Operations performed at aggressive optimization levels
  - enables –O3
  - more aggressive prefetching, loop transformations

PGI pgcc, pgcpp, pgf95

PGI Compiler Option	Description
-O3	Performs some compile time and memory intensive optimizations in addition to those executed with -O2, but may not improve performance for all programs.
-Mipa=fast, inline	Creates inter-procedural optimizations. **There is a loader problem with this option.**
-tp barcelona-64	Includes specialized code for the barcelona chip.
-fast	Includes: -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align –Mflushz
-g, -gopt	Produces debugging information.
-mp	Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.
-Minfo=mp,ipa	Provides information about OpenMP, and inter-procedural optimization.

Intel icc ifort

Intel Compiler Option	Description
-O3	More than O2, but maybe not faster
-ipo	Creates inter-procedural optimizations.
-vec_report[0|...|5]	Controls the amount of vectorizer diagnostic information.
-xW	Includes specialized code for SSE and SSE2 instructions (recommended).
-xO	Includes specialized code for SSE, SSE2 and SSE3 instructions.
-fast	Includes: -ipo, -O2, -static **DO NOT USE -- static load not allowed.**
-g -fp	debugging information produced
-openmp	Enable OpenMP directives
-openmp_report[0|1|2]	OpenMP parallelizer diagnostic level.

Usually, Start Here

- PGI: -O3 -fast –tp barcelona-64 –Mipa=fast
- Intel: -O3 –xW –ipo
- But don’t exceed –O2 without checking that your output is correct.

Compilation Exercise

- Code is from Numerical Recipes to do LU decomposition.
- Compare timings with different optimizations.
- Compare with implementation in GSL.

- Compile with different flags, including “-g”, “-O2”, “-O3”.
- Submit a job to see how fast it is.
- Recompile with new flags and try again.

- Sits in lude.tar.gz

The Makefile

- Edit top of makefile to change compiler and flags
  - COMPILER=pgcc
  - FFLAGS=-O2 -tp barcelona-64
  - VERSION=0
- “VERSION” is tacked onto the end of the executable names
  - nr0 and gsl0 or nr1 and gsl1.
- “make” generates executables.
- “make list” looks through your directory to find all executables.
- ./nr0 –f –o output_file –n 10000
  - -f tells it to tell you how you compiled the executable.
  - -o is the name of an optional output file to verify results.
  - -n is the size of the nxn matrix.

More Specifically

- Edit makefile to use “FFLAGS=-g” and VERSION=0. Then “make”.
- Edit makefile to use “FFLAGS=-O2” and VERSION=1. Then “make”.
- Edit makefile to use “FFLAGS=-O3” and VERSION=2. Then “make”.
- “make list” to see that they are all there.
  - ./nr0 pgcc -O2 -tp barcelona-64
  - ./gsl0 pgcc -O2 -tp barcelona-64
  - ./nr1 pgcc -O3 -tp barcelona-64
  - ./gsl1 pgcc -O3 -tp barcelona-64
  - ./nr2 pgcc -g -tp barcelona-64
  - ./gsl2 pgcc -g -tp barcelona-64
- “qsub –A 20090528HPC job.sge” or “make submit”
- Find the runtimes in the output to see the speeds.

If You Have Time

- Try other optimization flags.
  - Get more flags from http://services.tacc.utexas.edu/index.php/ranger-user-guide
  - Or look at “man pgcc” or “man icc”
- Try the Intel compiler by using the modules command.
- “make list” – lists all executables in your directory with their flags
- “make count” – counts the number of lines of code for nr vs. gsl
- How can the executable tell you the compiler and flags used to compile it?

From the Lab

- Why didn’t timings change much for GSL, even for debug version?
- How much faster is GSL than Numerical Recipes?
- What’s the difference in code size? (“make count”)

Single-Strided Array Access in C and Fortran

- The order of indices indicates how an array is stored in memory.
- The wrong order is *very* slow.

Fortran Example:

```
real*8 :: a(m,n), b(m,n), c(m,n)
...
do i=1,n
   do j=1,m
      a(j,i)=b(j,i)+c(j,i)
   end do
end do
```

C Example:

```
double a[m][n], b[m][n], c[m][n];
...
for (i=0;i < m;i++){
   for (j=0;j < n;j++){
      a[i][j]=b[i][j]+c[i][j];
   }
}
```

Streaming SIMD Extensions

- Feature of the CPU. SSE, SSE2, SSE3, SSE4.
- Perform simple instructions in parallel on single- or double-precision floating point.
- Very helpful for scientific code, because it tends to loop over arrays of floating point.
- Need to tell compiler the CPU type in order for it to compile for SSE.
- Generally, loops with independent iterations help use SSE.

Interprocedural Optimizations

- -ipo flags
- They examine function calls and loop structure in a single file or across files.
- Can inline functions, moving the function's code where it would have been called.
- One version lets you run the code on test data, profiles that code, then you recompile, and the compiler uses what it learned from the test data.

Optimization Conclusions

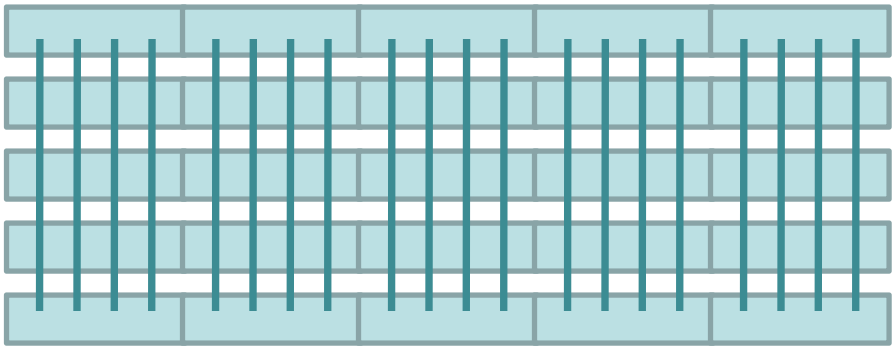
- Experiment with options.
- Test to ensure the program output is still correct.
- Write as little as possible yourself.

Efficiency of Parallel Algorithms

- Parallel programs are slower.

5 Serial Programs

5 Parallel Programs

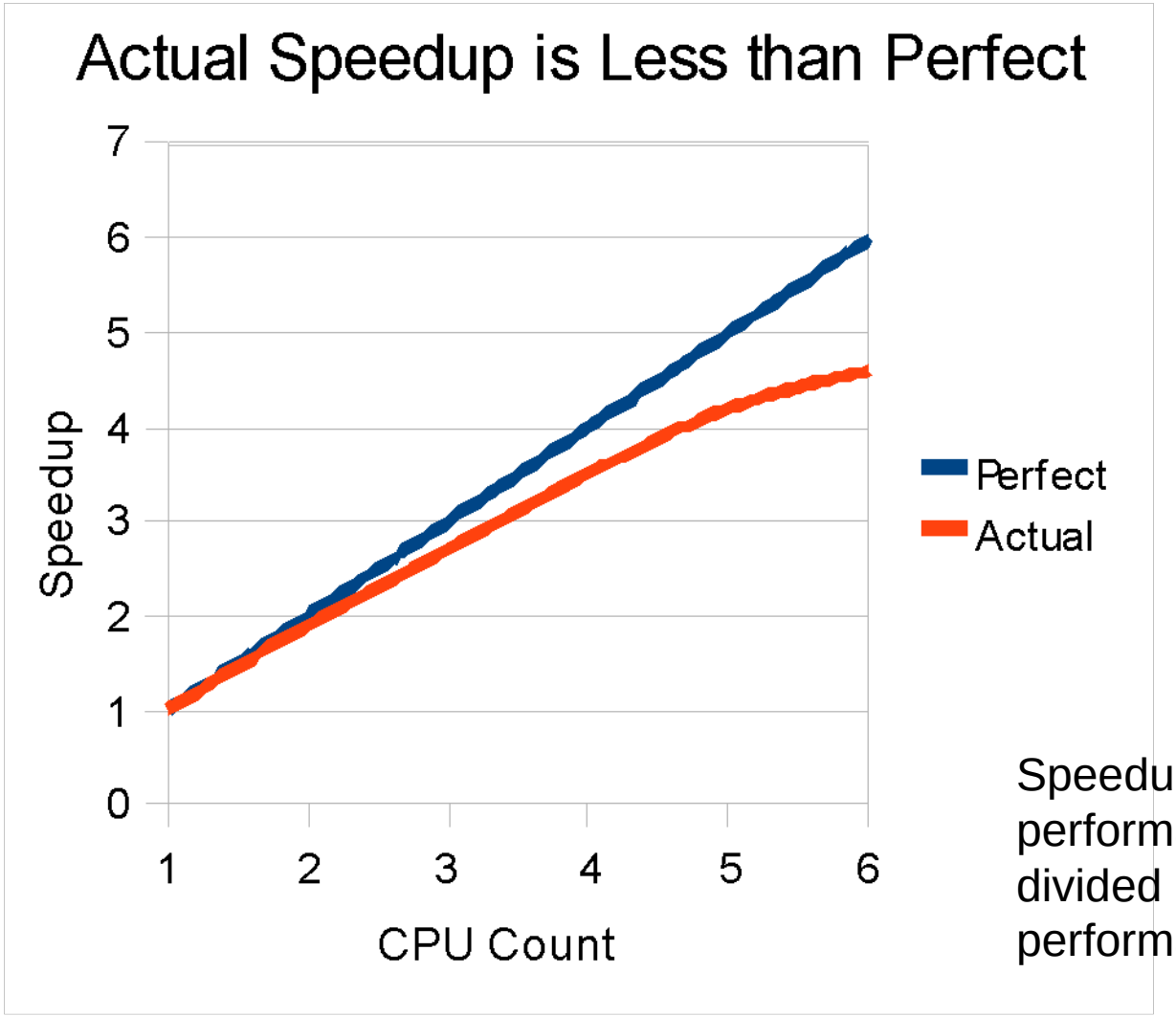


But We Do It Anyway Because

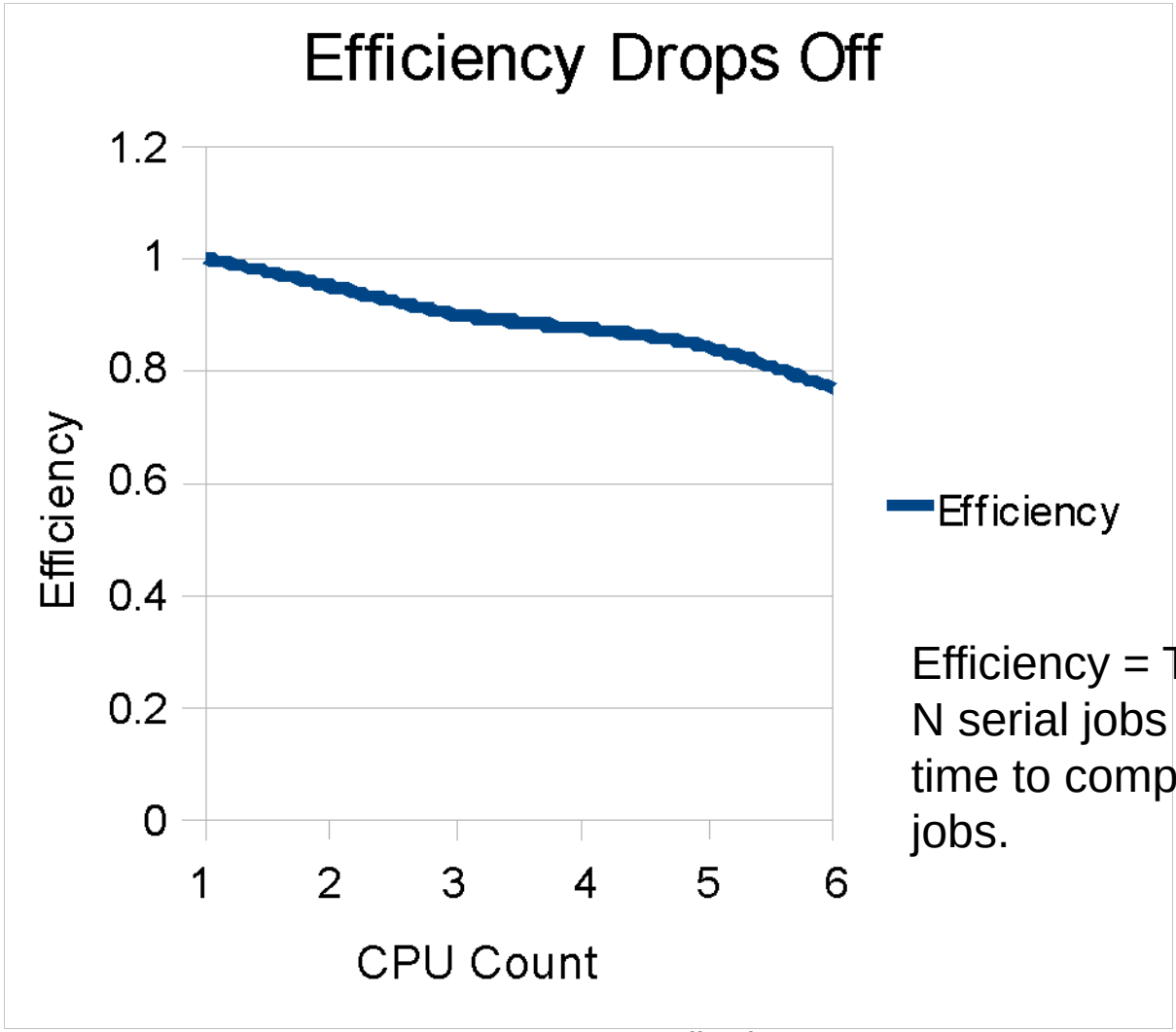
- It wouldn’t fit into memory on a smaller machine.
- The calculation would take too long otherwise.
  - There is one big calculation.
  - It’s not about efficient computers but about helping me make the next decision. I don’t know yet what I want to run next.

How Efficient Is My Program In Parallel?

- Each task does some unique computation.
- Each task does some repeated computation.
- Time to move data
  - From computational buffers to/from send buffers
  - Into the correct structure to start computation
- Time to send data
  - across the network
  - to the next core



Speedup = performance of serial divided by performance of parallel

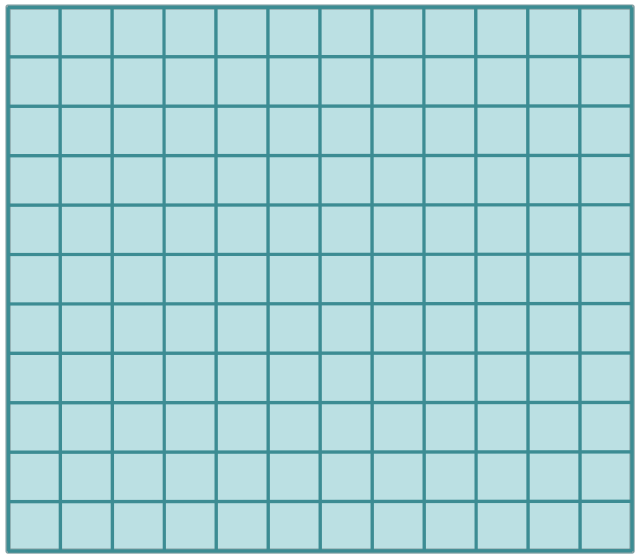


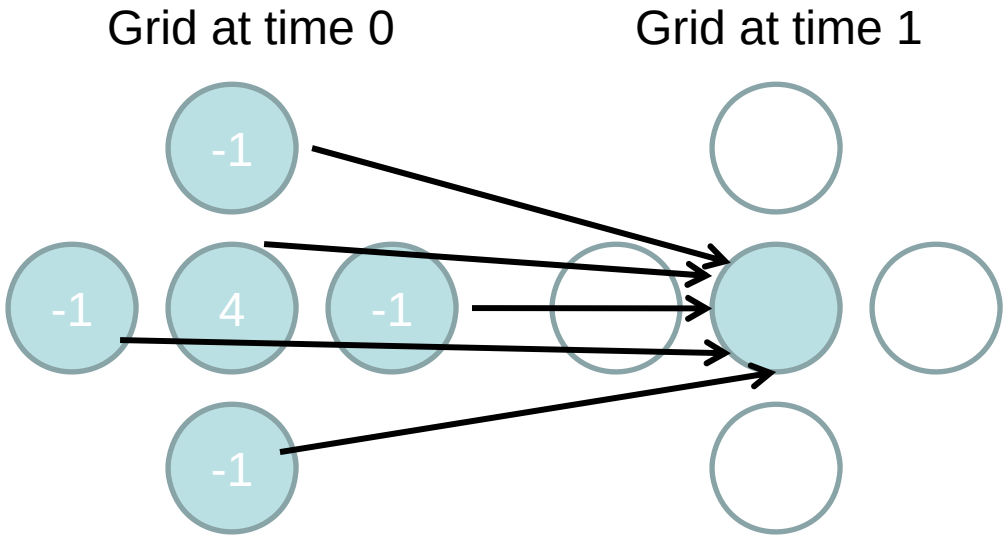
Efficiency = Time to complete N serial jobs divided by time to complete N parallel jobs.

Program as a Black Box

- How do you figure out how it will scale?

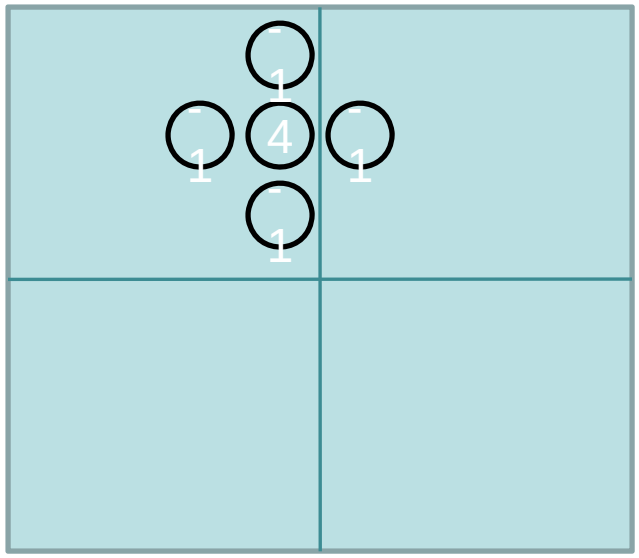
Example: Compute Evolution on 2D Grid





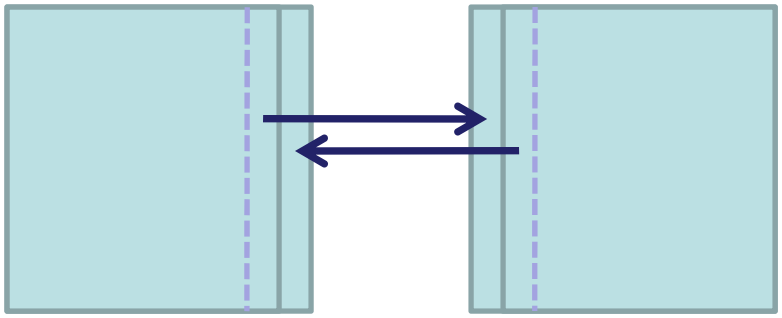
- At each time step, compute a new value from the old value at neighboring points. (No, you wouldn't do it this way. You would use an implicit method with Strang splitting.)

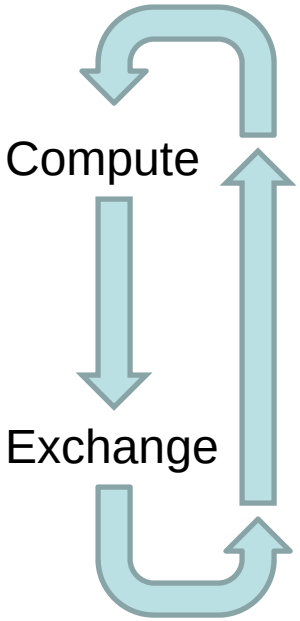
Domain Decomposition



- Calculating values near the edge needs information from neighboring domains.
- That data must be sent at every time step.

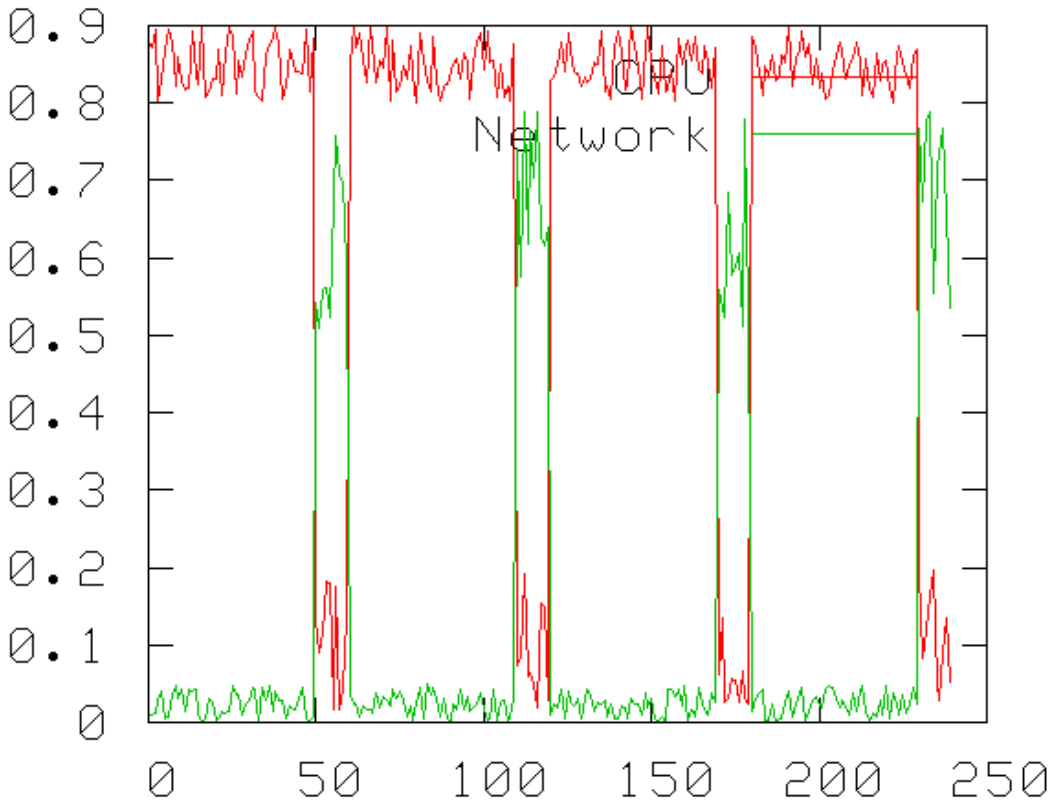
Compute and Exchange





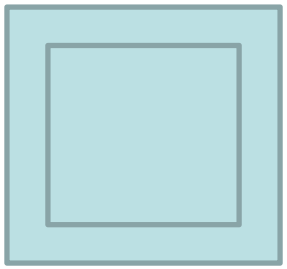
- Time per iteration = computation time + exchange time
- This is an example of a very *local* communication pattern.

Communication Pattern



- Now we understand the communication pattern in time and among tasks.
- It is local, synchronous, and regular.

Adding More Nodes Makes Domains Smaller But Neighbors Still Need The Same Piece



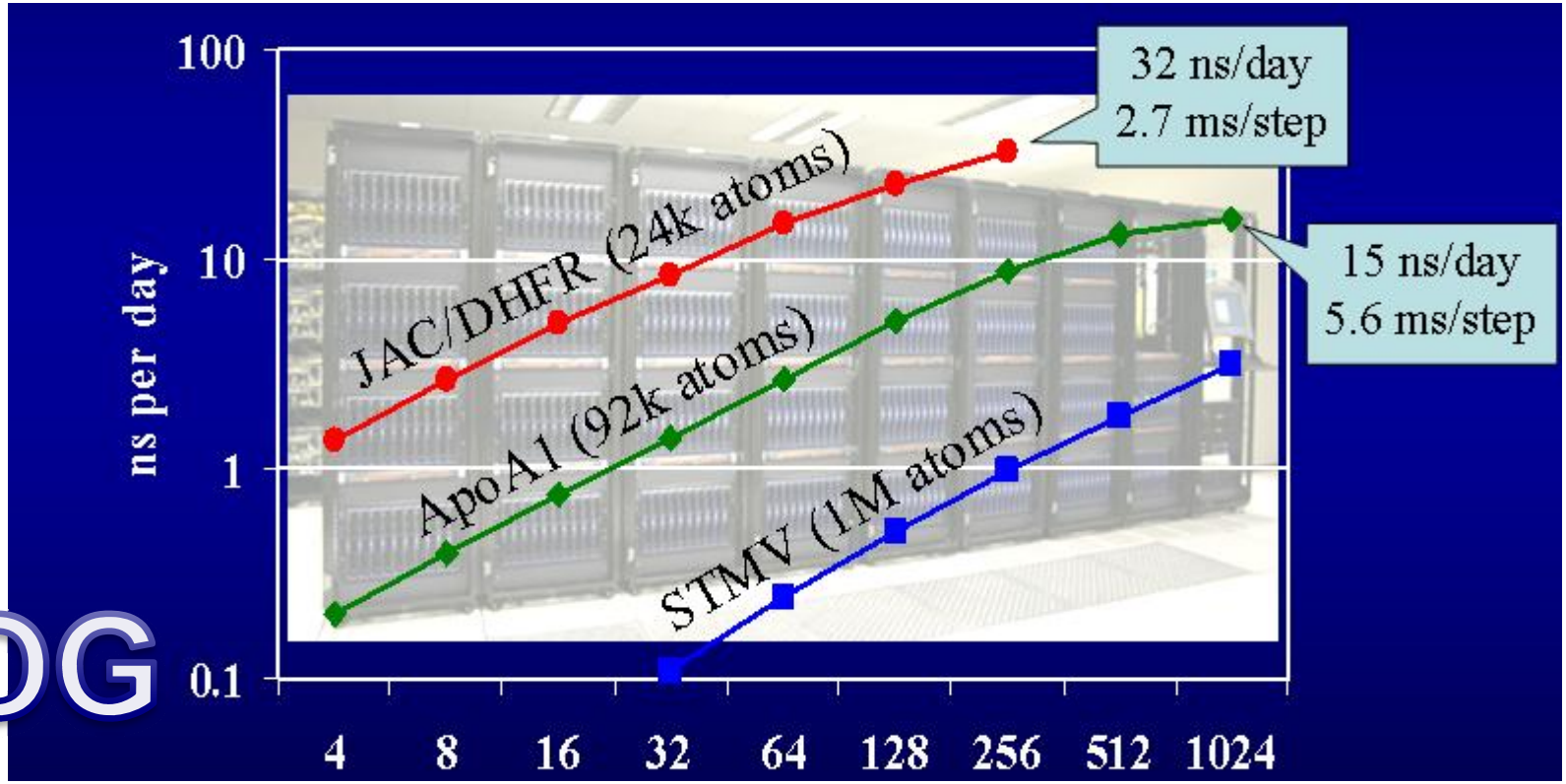
- Percentage of time communicating increases.
- Called *Strong Scaling.*
- Efficiency drops steadily.
- Eventually, no faster to add nodes.

Same Strong Scaling as an Equation

$$time = O\left(\frac{A}{N}\right) + O\left(\frac{L}{\sqrt{N}}\right)$$

- You need the time to decrease as 1/N in order to go faster. Boundary sending doesn't.
- What if you increased the size of the domain as you increased N?

Example of Strong Scaling for NAMD



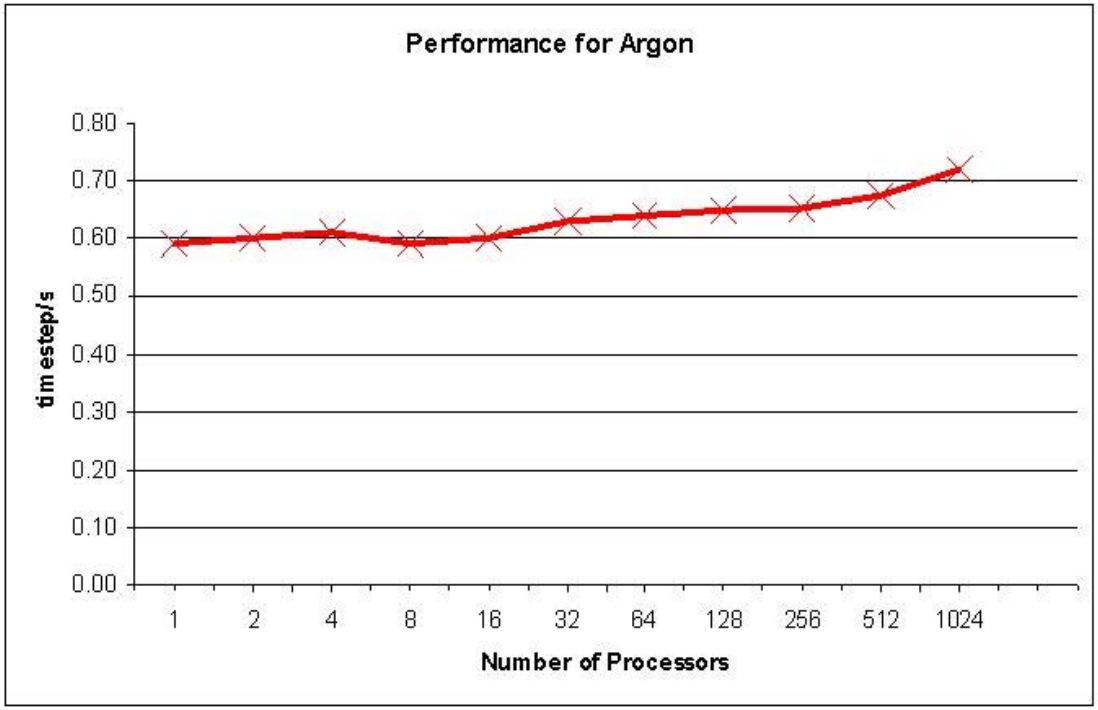
LOG

LOG

Doing the Same Problem Just Larger

- Increasing the size of the problem as the size of the computing resource increases is called *Weak Scaling.*
- Given our previous model of the 2D domain, we could double the size as we double the compute nodes and still be just as efficient.
- But the number of network messages typically increases faster than the number of nodes.
- But every time you ask all nodes to wait for each other, they take time to synchronize.
- But the network can only handle so many messages total.
- So strong scaling is good, but it doesn't fix everything.

Weak Scaling Example

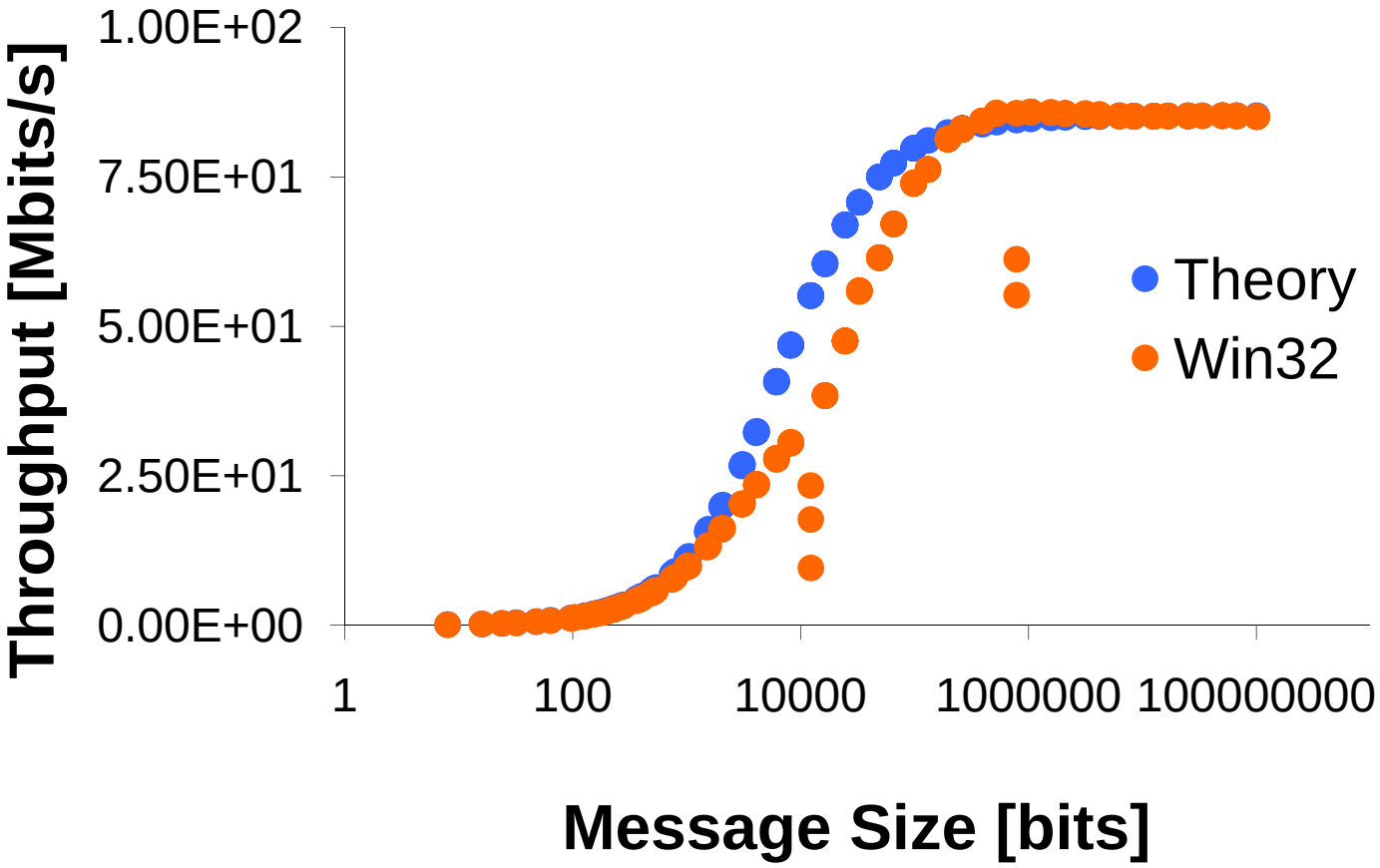


DL_POLY 3 (32,000 atoms per PE)

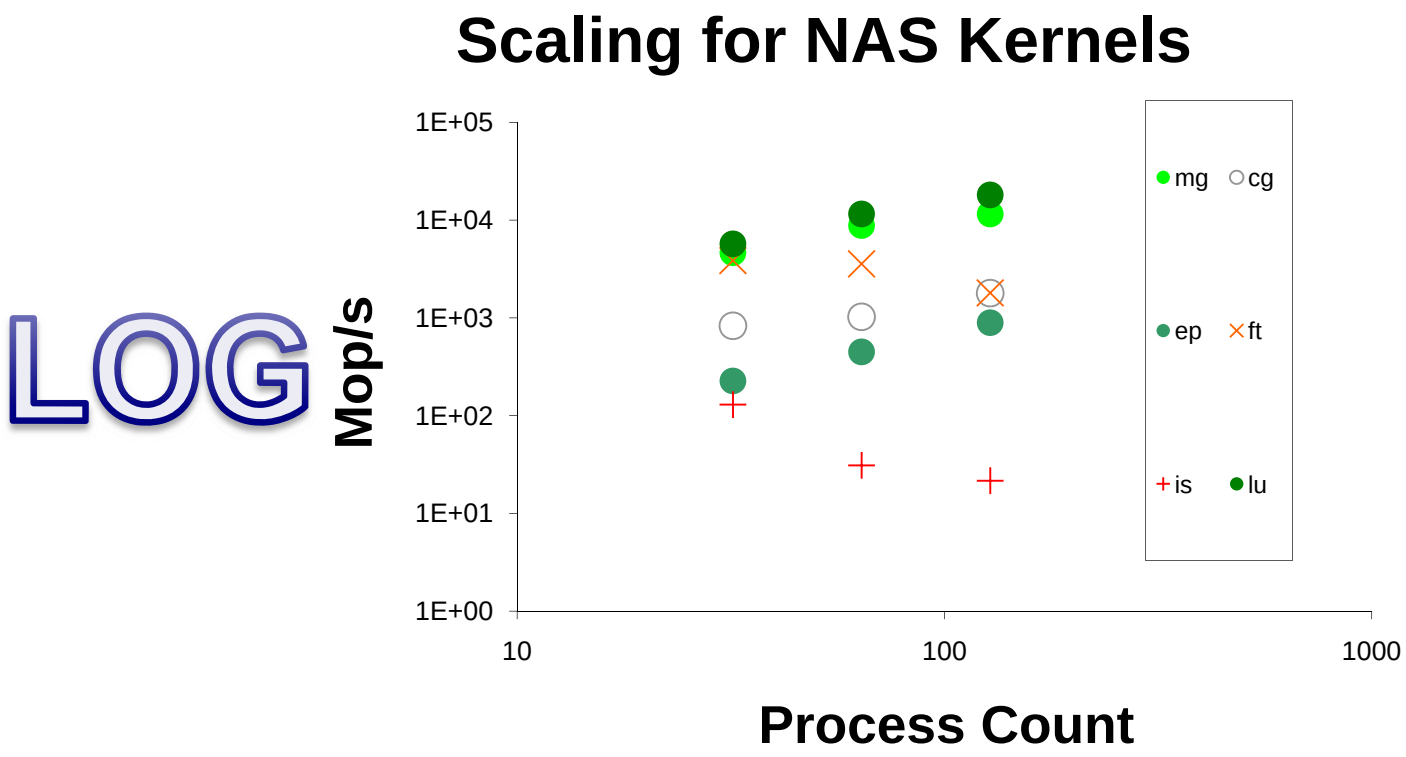
Timings on a Real Code

- Fluent is a spectral code for fluid dynamics.
- It's behavior is complex as the number of nodes increases.
- Look at ~train200/NetworkEstimate.xls.

TCP Throughput



Different Algorithms Scale Differently



Scalability Lab

- 3D Real FFTW
- Uses FFTW2 with its MPI support
- You can run it to your heart's content:
  - -pe 16way, -pe 1way, -pe 14way
  - Node counts that fit in the queue you use
- It may fail if the 1024 isn't divisible by the task count.
- Then we plot.

Scalability Lab - Start

- tar zxf ~train200/fftw_mpi.tar.gz
- cd fftw_mpi
- make
- Why won't it build? If you feel like checking the next slide for the answer, then don't.

Scalability Lab - Modules

- You need the right libraries loaded to build it.
- module del mvapich
- module swap intel pgi
- module load mvapich
- module load fftw2

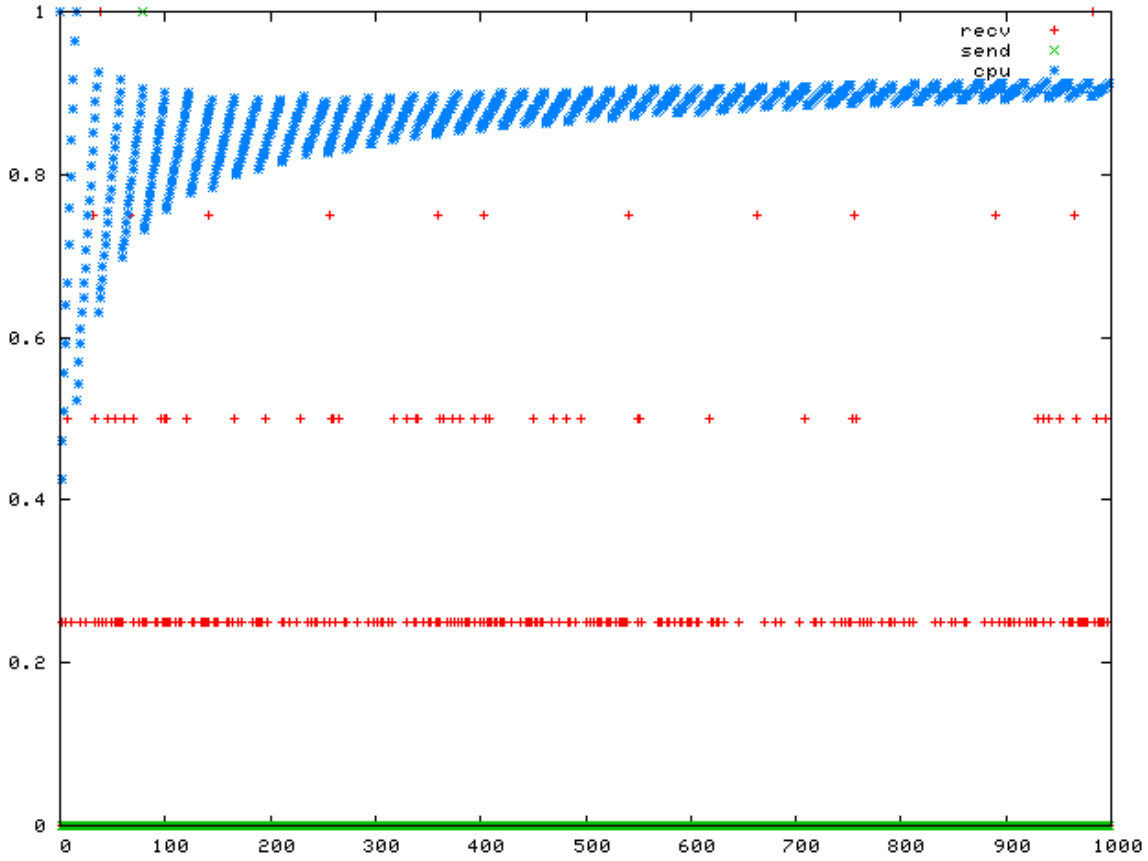
Scalability – Edit the Job Script

- It starts with “-pe 16way 16”. Submit it this way.
- Submit a few jobs with other wayness and core count.
- When you have a few output files that have good results, type “make results”. This creates “~/fftwtimes” with the following:
  - Number of cores used for the run
  - Wayness
  - Seconds taken
  - log10(number of cores)
  - log10(time)

Scalability Lab – For More

- To make a plot
  - cp ~/fftwtimes alltimes
  - gnuplot alltimes.gp
- Then copy the resulting png files to your local computer to view.
- How are results different running 8 processes on 1 node versus 8 split among two nodes or 8 on 8 nodes?
- Does “tacc_affinity” from http://services.tacc.utexas.edu/index.php/ranger-user-guide affect the speed?

FFTW CPU Usage Every 0.01 Seconds



Conclusions

- Communication pattern controls scalability.
- It’s all about powers, so use log-log plots.