Engineering Performance for Multiphysics Applications

William Gropp
www.cs.illinois.edu/~wgropp

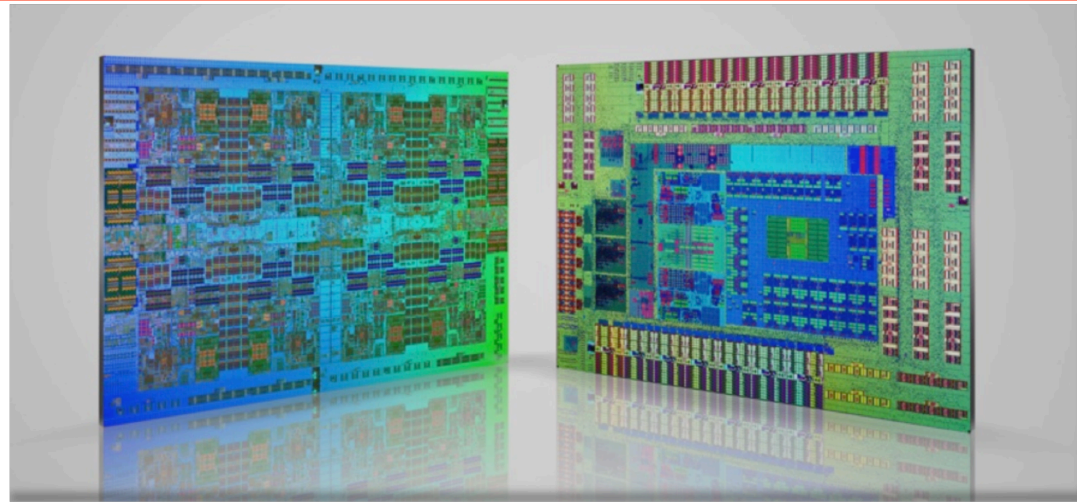
Performance, then Productivity

- Note the "then" – not "instead of"
  - For "easier" problems, it is correct to invert these
- For the very hardest problems, we must focus on getting the best performance possible
  - Rely on other approaches to manage the complexity of the codes
  - Performance can be understood and engineered (note I did not say predicted)
- We need to start now, to get practice
  - "Vector" instructions, GPUs, extreme scale networks
  - Because Exascale platforms will be even more complex and harder to use effectively

Exascale Directions

- Exascale systems are likely to have
  - ♦ Extreme power constraints, leading to
    - Clock Rates similar to today's systems
    - A wide-diversity of simple computing elements (simple for hardware but complex for software)
    - Memory per core and per FLOP will be much smaller
    - Moving data anywhere will be expensive (time and power)
  - ♦ Faults that will need to be detected and managed
    - Some detection may be the job of the programmer, as hardware detection takes power
  - ♦ Extreme scalability and performance irregularity
    - Performance will require enormous concurrency
    - Performance is likely to be variable
      - Simple, static decompositions will not scale
  - ♦ A need for latency tolerant algorithms and programming
    - Memory, processors will be 100s to 10000s of cycles away. Waiting for operations to complete will cripple performance

IBM PERCS: Two New Chips



Power7 Chip

Up to 256 GF peak performance

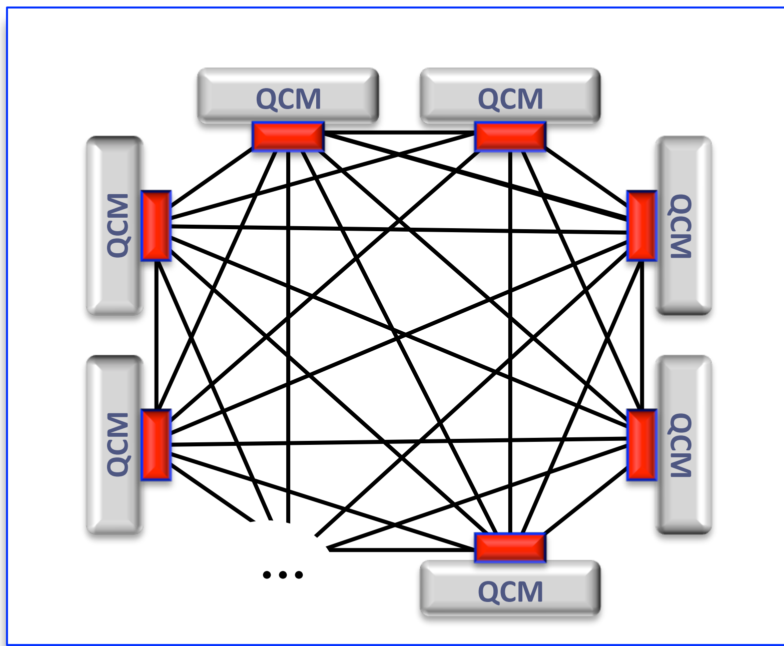
3.5–4.0 GHz
Up to 8 cores, 32 SMT threads
Caches
  L1 (2x64 KB), L2 (256 KB),
  L3 (32 MB, complex policy)
Memory Subsystem
  Two memory controllers
  128 GB/s memory bandwidth

PERCS Hub Chip

1.128 TB/s total bandwidth

Connections:
  192 GB/s QCM connection
  896 GB/s to other QCMs
  40 GB/s general purpose I/O

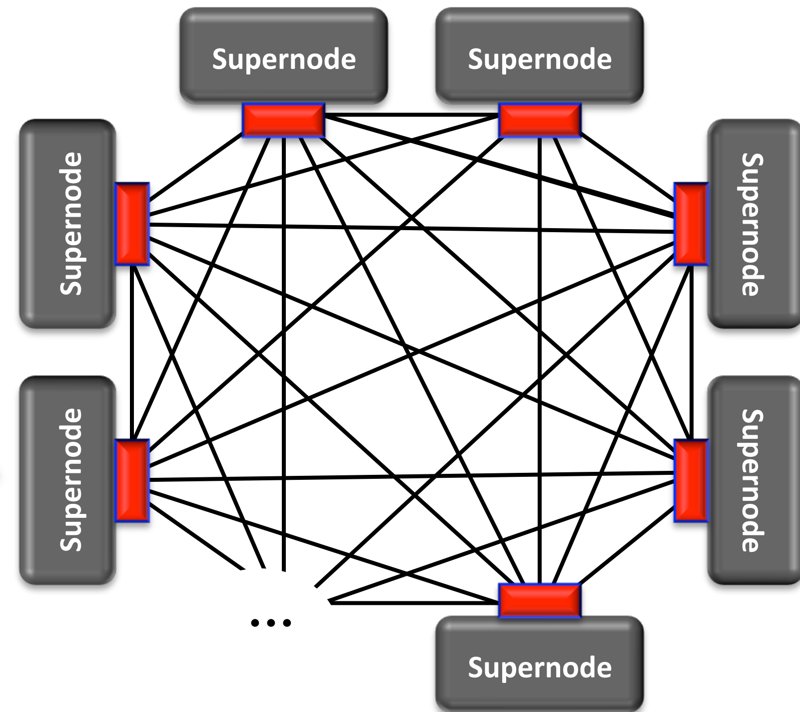
Two-level (L, D) Direct-connect Network



Each Supernode = 32 QCMs
(4 Drawers x 8 SMPs/Drawer)

Fully Interconnected with L_{local} and L_{remote} Links

★ **Result: Very low hardware latency Very high bandwidth**



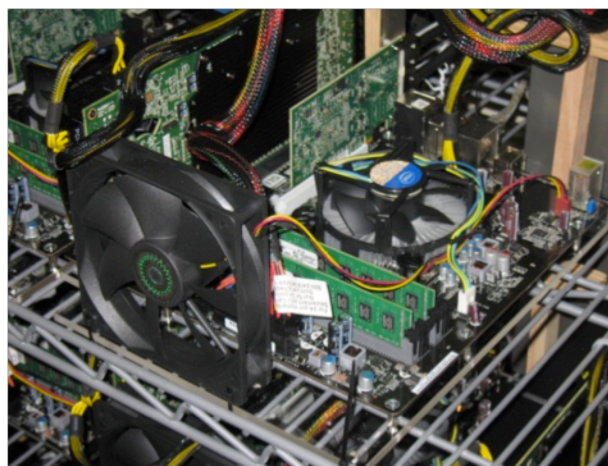
Blue Waters = 320 Supernodes
(40 BBs x 8 SNs/BB)

Fully Interconnected with D Links

But complex, nonuniform network

Another Example System

- 128 node GPU Cluster
- #3 on Green500
- Each node has
  - One Core i3 530 2.93 GHz dual-core CPU
  - One Tesla C2050 GPU per node
- 33.62 TFLOPS on HPL
- 934 MFLOPS/Watt
- How can we *engineer* codes for performance on these complex systems?
- And an exercise for the viewer: what do performance models tell you about the CPU/GPU comparisons you see?

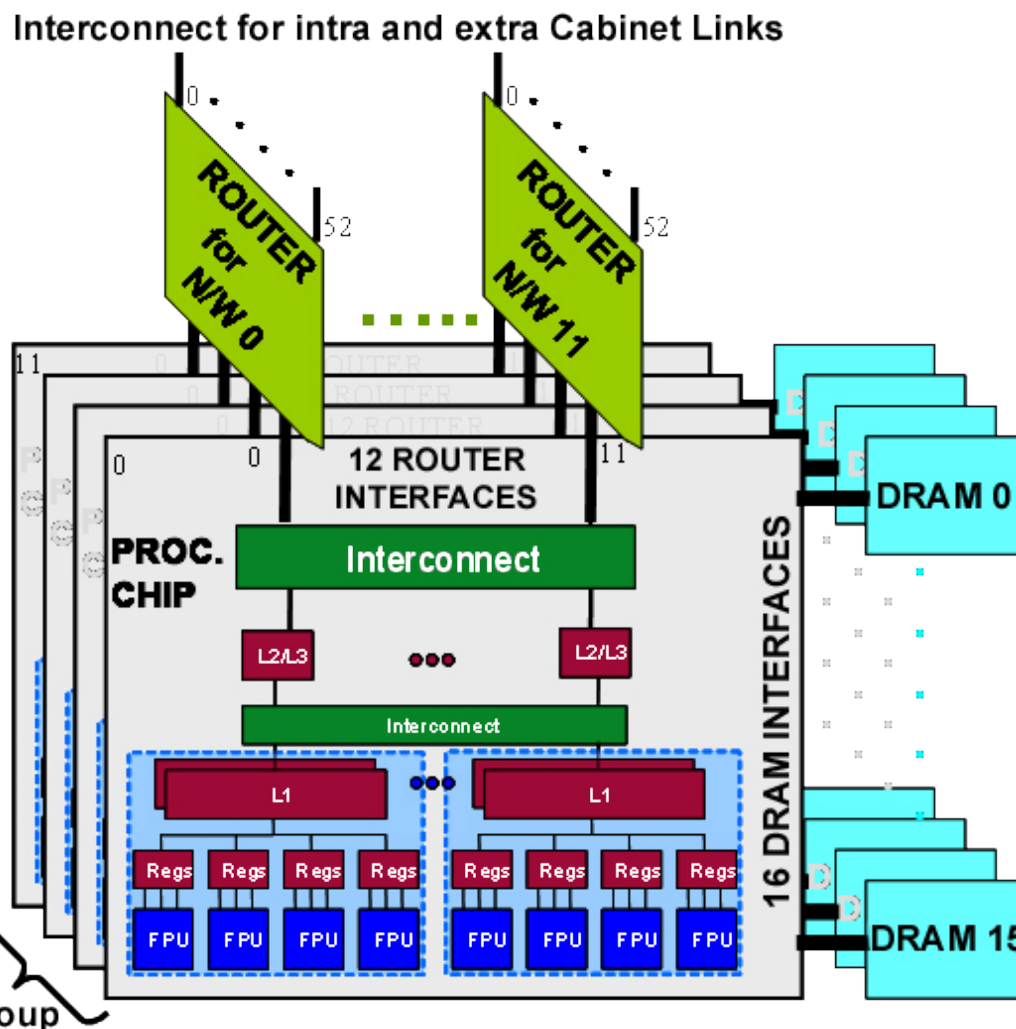


1 EFlop/s "Clean Sheet of Paper" Strawman

Sizing done by "balancing" power budgets with achievable capabilities

- **4 FPUs+RegFiles/Core (=6 GF @1.5GHz)**
- **1 Chip = *742* Cores (=4.5TF/s)**
  - **213MB of L1I&D; 93MB of L2**
- **1 Node = 1 Proc Chip + 16 DRAMs (16GB)**
- **1 Group = 12 Nodes + 12 Routers (=54TF/s)**
- **1 Rack = 32 Groups (=1.7 PF/s)**
  - **384 nodes / rack**
- **3.6EB of Disk Storage included**
- **1 System = 583 Racks (=1 EF/s)**
  - **166 MILLION cores**
  - **680 MILLION FPUs**
  - **3.6PB = 0.0036 bytes/flops**
  - **68 MW w'aggressive assumptions**

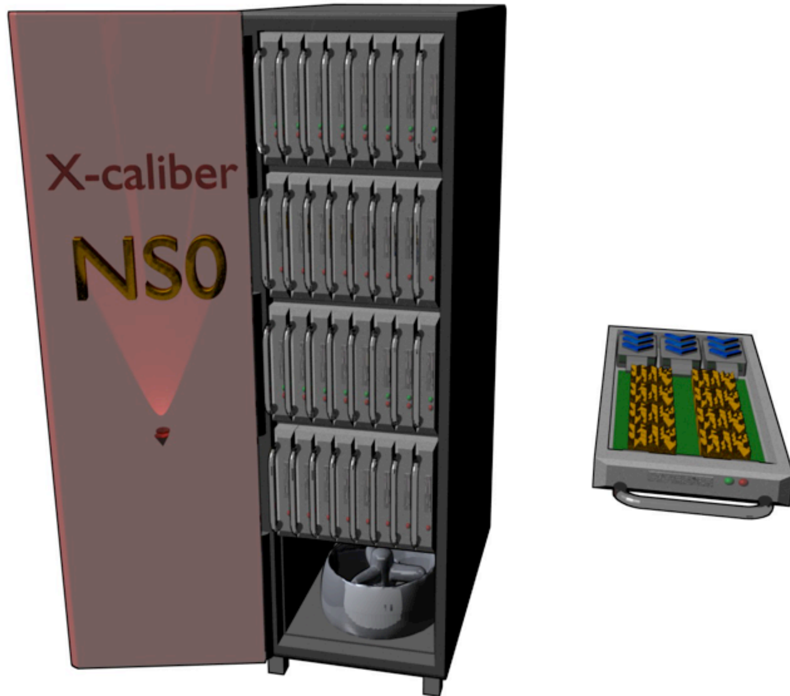
Largely due to Bill Dally, Stanford



UNIVERSITY OF NOTRE DAME

Thanks to Peter Kogge for this slide, based on the DARPA report

An Even More Radical System



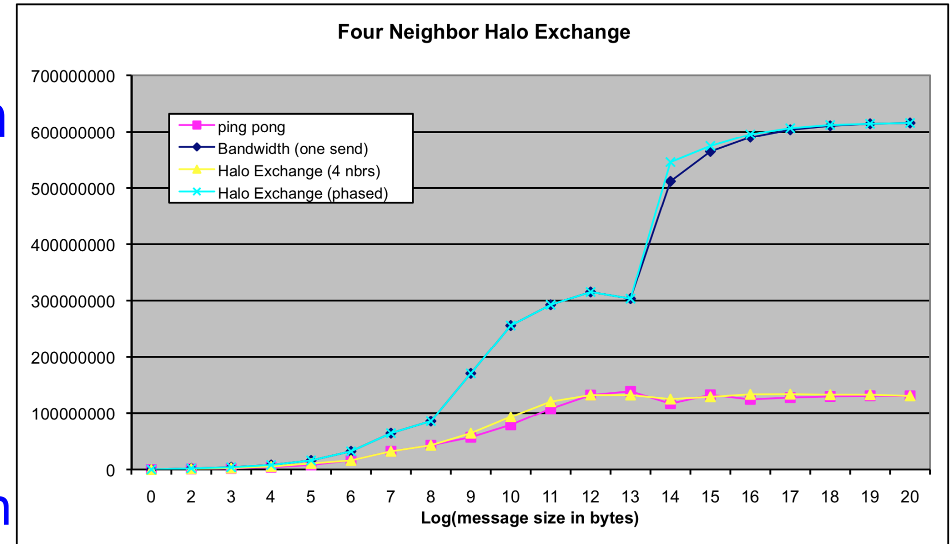
- Rack Scale
  - Processing:128 Nodes, 1 (+) PF/s
  - Memory:
    - 128 TB DRAM
    - 0.4 PB/s Aggregate Bandwidth
  - NV Memory
    - 1 PB Phase Change Memory (addressable)
    - Additional 128 for Redundancy/RAID
  - Network
    - 0.13 PB/sec Injection, 0.06 PB/s Bisection

Deployment	Nodes	Topology	Compute	Mem BW	Injection BW	Bisection BW
Module	1	N/A	8 TF/s	3 TB/s	1 TB/s	N/A
Deployable Cage	22	All-to-All	176 TF/s	67.5 TB/s	22.5 TB/s	31 TB/s
Rack	128	Flat. Butterfly	1 PF/s	.4 PB/s	0.13 PB/s	0.066 PB/s
Group Cluster	512	Flat. Butterfly	4.1 PF/s	1.6 PB/s	0.52 PB/s	0.26 PB/s
National Resource	128k	Hier. All-to-All	1 EF/s	0.4 EB/s	0.13 EB/s	16.8 PB/s
Max Configuration	2048k	Hier. All-to-All	16 EF/s	6.4 EB/s	2.1 EB/s	0.26 EB/s

Need for Adaptivity

- Uniform meshes rarely optimal
  - More work than necessary
  - Note that minimizing floating-point operations will not minimize running time – perfect irregular mesh is also not optimal
- Once adaptive meshing/model approximations used, need to address load balance, avoid the use of synchronizing operations
  - No barriers
  - Nothing that looks like a barrier (MPI_Allreduce)
    - See MPI_Iallreduce, likely to appear in MPI 3
  - Care with operations that are weakly synchronizing– e.g., neighbor communication (it synchronizes, just not as tightly)
    - Using MPI_Send synchronizes

Consequences of Unnecessary Synchronization



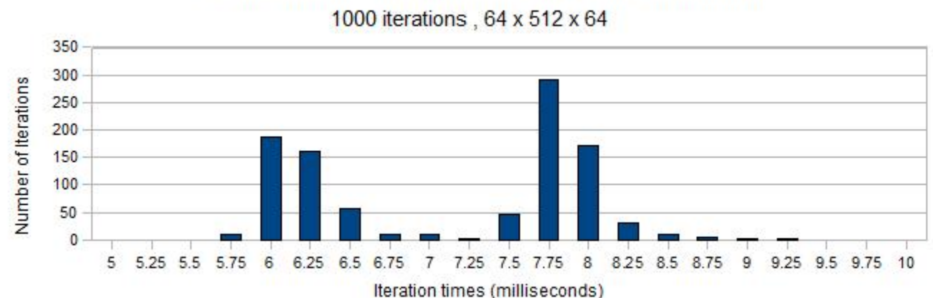
- How relevant is ping-pong bandwidth and real system
- What are the correct parameters?
  - Model the real system, but abstractly
  - For Blue Gene, must model independent communication
  - Impacts choice of communication algorithm (many benchmarks do not provide a relevant measurement)
- Using one MPI_Send at a time prevents use of concurrent communication
  - Similar effects even if there is one communication path out of node, but contention in the network. Performance can suffer 2x or more slow down
  - Unnecessary in many cases
  - Benchmarks that sue MPI_Send are not "fair"

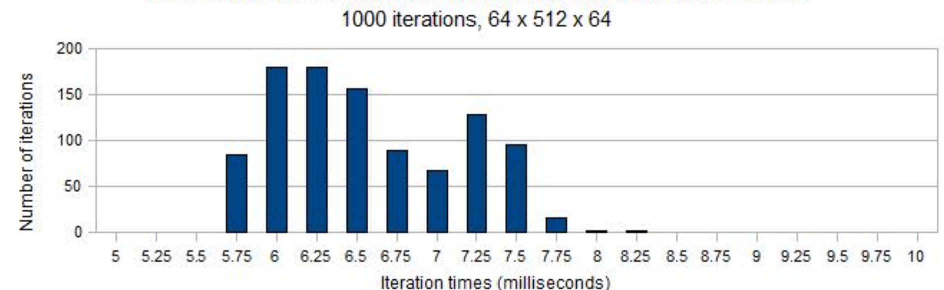
Processes and SMP nodes

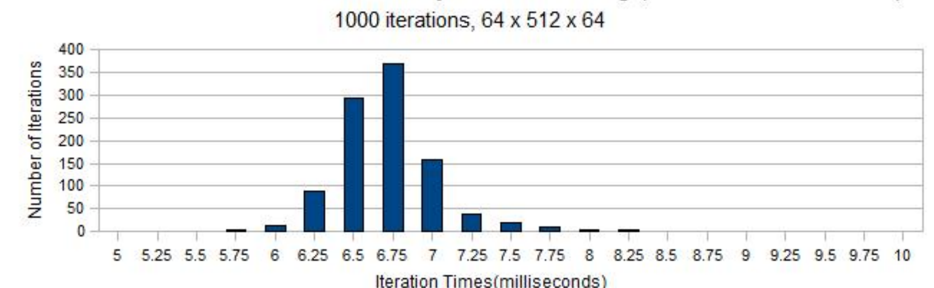
- HPC users typically believe that their code "owns" all of the cores all of the time
  - ♦ The reality is that was never true, but they did have all of the cores the same fraction of time when there was one core /node
  - ♦ Given this belief, load balancing is unnecessary for regular grid codes
  - ♦ Is this true?
- We can use a simple performance model to check the assertion and then use measurements to identify the problem and suggest fixes.
- Consider a simple Jacobi sweep on a regular mesh, with every core having the same amount of work. How are run times distributed?

Sharing an SMP

- Having many cores available makes everyone think that they can use them to solve other problems ("no one would use all of them all of the time")
- However, compute-bound scientific calculations are often *written* as if all compute resources are owned by the application
- Such *static* scheduling leads to performance loss
- Pure dynamic scheduling adds overhead, but is better
- Careful mixed strategies are even better
- Recent results give 10-16% performance improvements on large, scalable systems
- Thanks to Vivek Kale







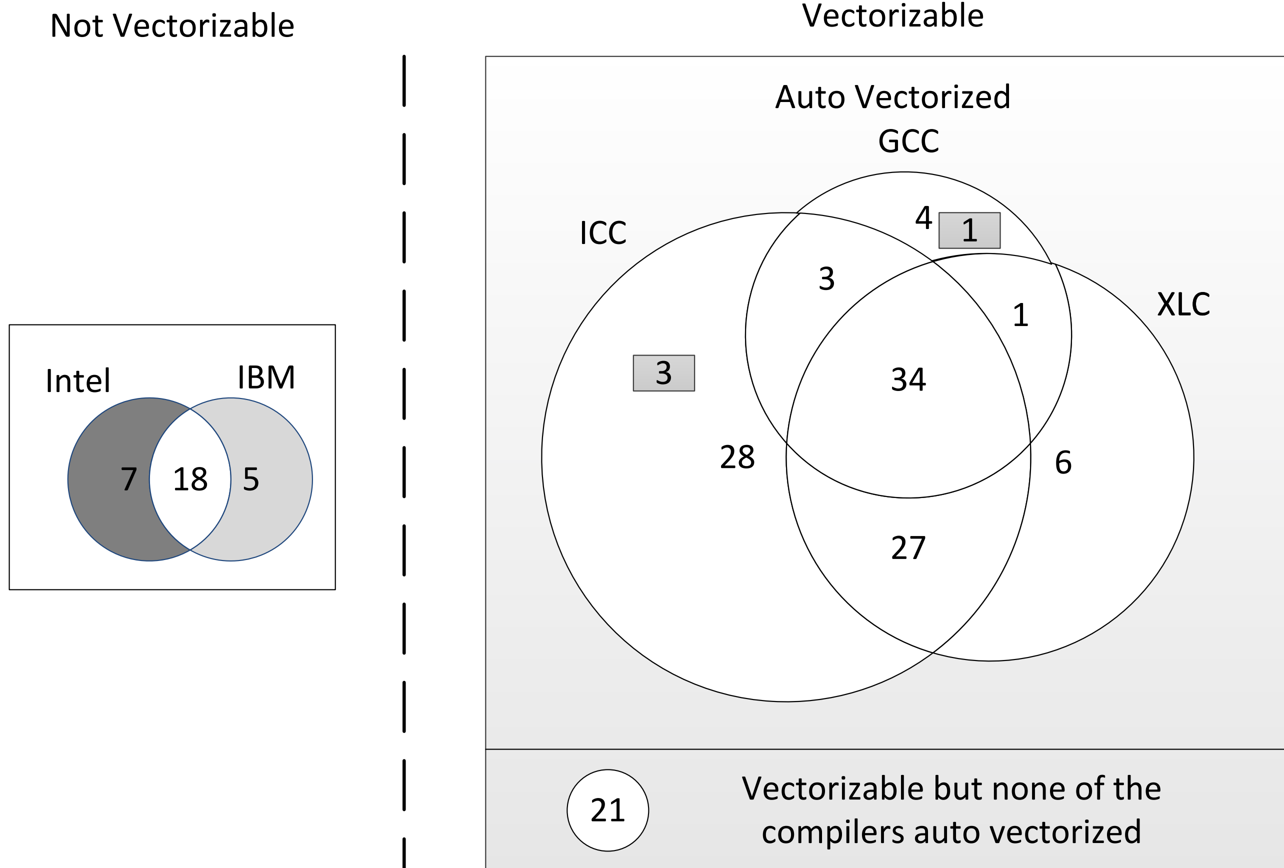
Need for Aggregation

- Functional units are cheap
  - ♦ Small amount of area, relatively small amount of power
  - ♦ Memory motion is expensive
  - ♦ Easy to arrange many floating point units, in different patterns
    - Classic vectors (Cray, NEC SX)
    - Commodity vectors (2 or 4 elements)
    - Streams
    - GPU
  - ♦ All have different requirements on both the algorithms (e.g., work with full vectors) and programming (e.g., satisfy alignment rules)
  - ♦ Compilers will be able to help but will not solve the problem
    - The following compares three compilers success at producing good commodity vector code from loops in applications

Utilizing the Processor

- Note rapidly growing numbers of functional units – Power7 has 2 multiply-add units per core; x86 increasingly long; accessed through "vector" instructions
- How do we know how well we are doing?
- How do we know how well the compiler is doing?
- We can model the expected performance, including vectorization!
- Using the model, we can also identify where manually applying well-known transformations will help
- Also identifies where extra constraints, such as alignment restrictions, may inhibit use of vectorization

How Good are Compilers at Vectorizing Codes?



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. An Evaluation of Vectorizing Compilers. In preparation. 2011.

Media Bench II Applications

Appl	XLC	ICC	GCC	XLC	ICC	GCC
	Automatic			Manual		
JPEG Enc	-	1.33	-	1.39	2.13	1.57
JEPG Dec	-	-	-	-	1.14	1.13
H263 Enc	-	-	-	1.25	2.28	2.06
H263 Dec	-	-	-	1.31	1.45	-
MPEG2 Enc	-	-	-	1.06	1.96	2.43
MPEG2 Dec	-	-	1.15	1.37	1.45	1.55
MPEG4 Enc	-	-	-	1.44	1.81	1.74
MPEG4 Dec	-	-	-	1.12	-	1.18

Table shows **whole program speedups** measured against unvectorized application

S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. An Evaluation of Vectorizing Compilers. In preparation. 2011.

Need for Appropriate Data Structures

- Choice of data structure strongly affects ability of the system to provide good performance (duh!)
  - Key is to work *with* the hardware provided for improving memory system performance, rather than using it as a crutch
  - This choice often requires a large scale view of the problem and is not susceptible to typical autotuning approaches

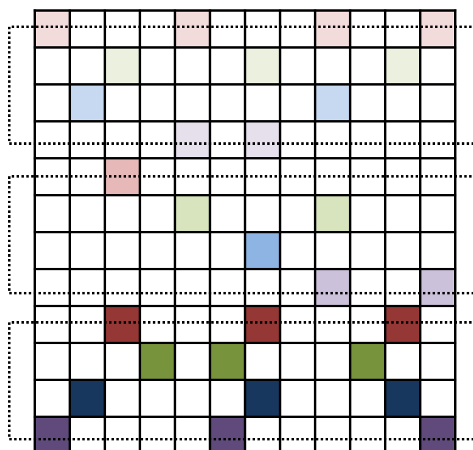
Processes and Memory

- For many computations, sustained memory performance is the limiting resource
  - As in sparse matrix-vector multiply
- What is the appropriate sustained rate?
  - Memory bus bandwidth is nearly irrelevant – it is the sustained rate that is usually important
  - What about other ways to increase effective sustained performance, such as prefetch?
- Prefetch hardware can detect regular accesses and prefetch data, making use of otherwise idle memory bus time.
  - However, the hardware must be presented with enough independent data streams

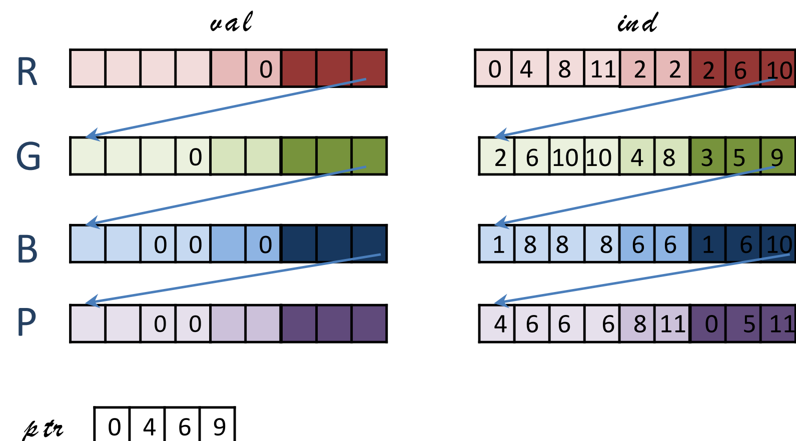
Streamed Compressed Sparse Row (S-CSR) format

- S-CSR format partitions the sparse matrix into blocks along rows with size of bs. Zeros are added in to keep the number of elements the same in each row of a blockThe first rows of all blocks are stored first, then second, third … and bs-th rows.
- For the sample matrix in the following Figure, NNZ = 29. Using a block size of bs = 4, it generates four equal length streams R, G, B and P. This new design only adds 7 zeros every 4 rows.

A sparse matrix (N = 12, NNZ= 29)



Streamed Compressed Sparse Row format (S-CSR)



NoIS

Performance Ratio Compared to CSR Format

- S-CSR format is better than CSR format for all (on Power 5 and 6) or Most (on Power 4) matrices
- S-BCSR format is better than BCSR format for all (on Power 6) or Most (on Power 4 and 5) matrices
- Blocked format performance from ½ to 3x CSR.

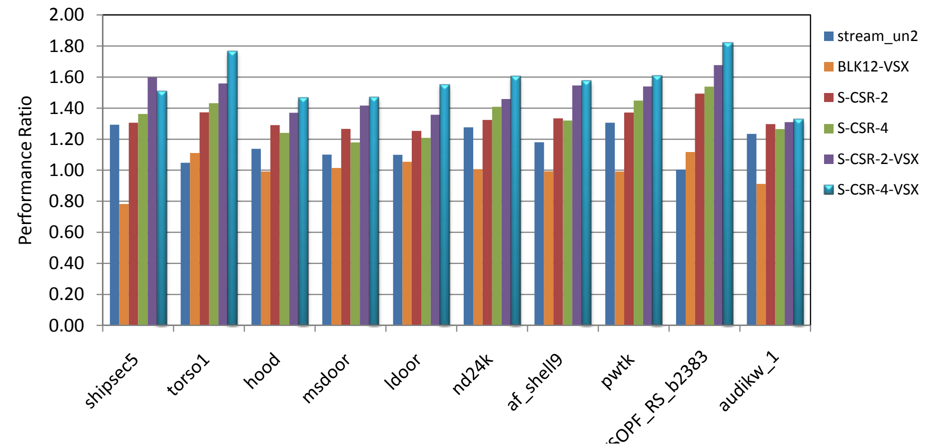
BSCR-4
S-BCSR-4
S-CSR-4

Performance Ratio
Power 4
Power5
Power 6

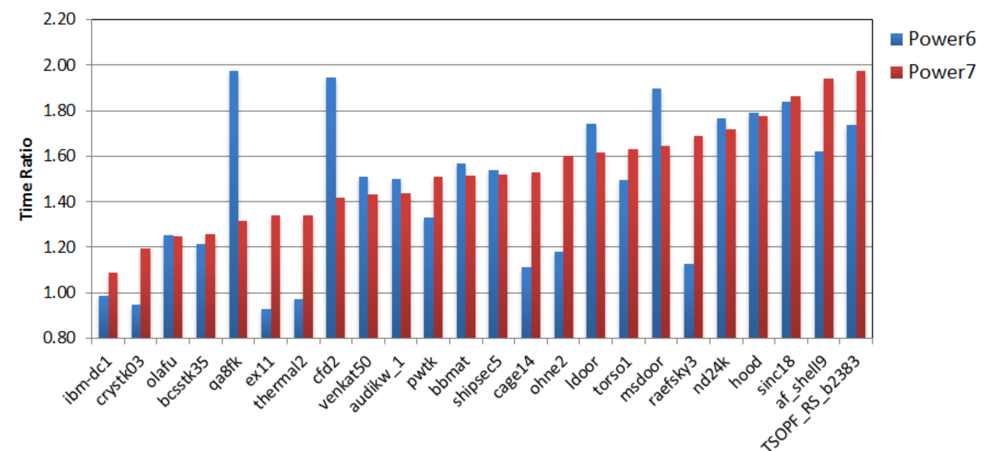
ohne2, sinc18, airfoil_2d, thermal2, cfd2, ibm-dc1, audikw_1, shipsec5, bcircuit, msdoor, finan512, hood, ldoor, finance256, nd24k, af_shell9, qa8fk, goodwin, gupta3, gemat11, pwt, TSOPF_RS_b2383, 494_bus, shyy161, raefsky3_5, bbmat, bcsstk25, dw4096, crystk03, olafu, bcsstk35, s3rmt3m1, bcsstm27, venkat50

Combining With Other Optimizations

- We can further modify the S-CSR and S-BCSR to match the requirements for vectorization
- We can use OSKI to optimize "within the loops"



Time comparison between updated OSKI and original OSKI

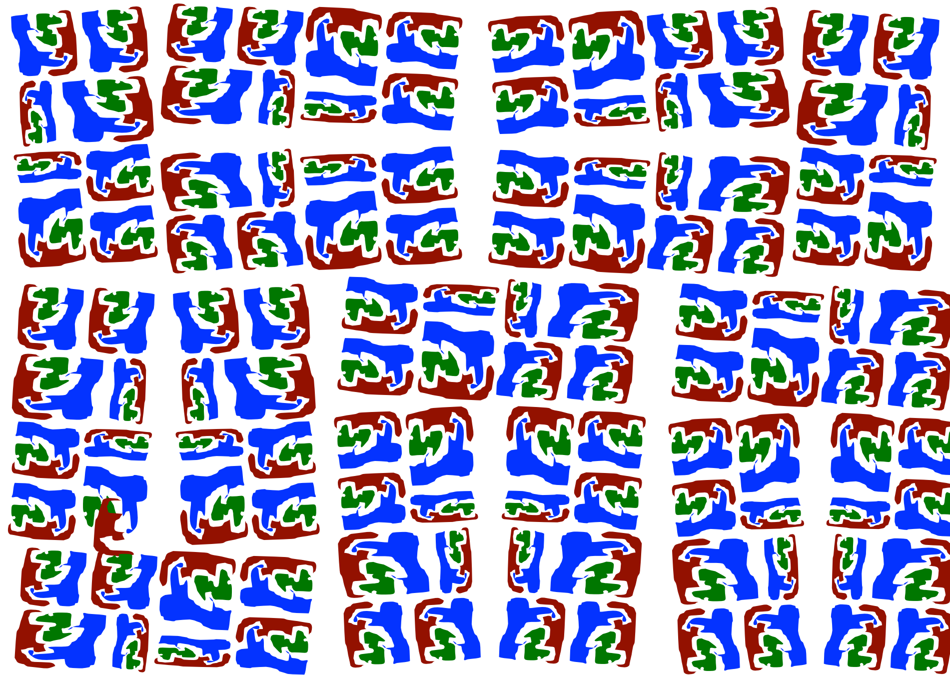


PARALLEL@ILLINOIS

Implications

- Vertically integrated (all modules within same "locality domain")
  - ♦ Not horizontally in processor blocks
  - ♦ Adapt for load balance
    - Challenges
      - – Minimize memory motion
      - – Work within limited memory
  - ♦ Likely approach: interleave components in regions (nodes, if nodes have 1000's of cores)

Locality Domains



- In hardware, the memory is in a hierarchy – core, memory stick, chip, node, module, rack, ..
- Algorithm/ implementation needs to respect this hierarchy

1867

PARALLEL@ILLINOIS

Implications 2

- Restrict the use of separate computational and communication "phases"
  - Need more overlap of communication and computation to achieve latency tolerance (and energy reduction)
  - Adds pressure to be memory efficient

Implications 3

- Use aggregates that match the hardware
- Limit scalars to limited, essential control
  - Data must be in a hierarchy of small to large
- Fully automatic fixes unlikely
  - No vendor compiles the simple code for DGEMM and uses that for benchmarks
  - No vendor compiles simple code for a shared memory barrier and uses that (e.g., in OpenMP)
  - Until they do, the best case is a human-machine interaction, with the compiler helping

Possible Solution Directions

- Use mathematics as the organizing principle
  - Continuous representations, possibly adaptive, memory-optimizing representation, lossy (within accuracy limits) but preserves essential properties (e.g., conservation)
- Manage code by using data-structure-specific languages to handle operations and vertical integration across components
  - So-called "domain specific languages" are really data-structure specific languages – they support more applications but fewer algorithms.
  - Difference is important because a "domain" almost certainly require flexibility with data structures and algorithms

Possible Solution Directions

- Adaptive program models with a multi-level approach
  - Lightweight, locality-optimized for fine grain
  - Within node/locality domain for medium grain
  - Regional/global for coarse grain
  - May be different programming models (hierarchies are ok!) but they must work well together
- Performance annotations to support a complex compilation environment
- Asynchronous algorithms
- Integrated Development Environment (IDE) to ease vertical code development, maintenance, and refactoring

Conclusions

- Planning for extreme scale systems requires rethinking both algorithms and programming approaches (duh!)
- Key requirements include
  - ♦ Minimizing memory motion at all levels
  - ♦ Avoiding unnecessary synchronization at all levels
- Decisions must be informed by performance modeling / understanding
  - ♦ Not necessarily performance estimates – the goal is to guide the decisions

Conclusions

- Practical issues require separating algorithm, data structure, and implementation
  - ♦ Libraries will need to be supplemented by generated code
  - ♦ They may be data-structure-specific languages or annotations
    - Most proposals are not for domain specific, as they make assumptions about data structure and algorithm
    - Matlab is, after all, not domain specific – it is primarily data structure specific

Thanks

- Torsten Hoefler
  - Performance modeling lead, Blue Waters; MPI datatype
- David Padua, Maria Garzaran, Saeed Maleki
  - Compiler vectorization
- Dahai Guo
  - Streamed format exploiting prefetch
- Vivek Kale
  - SMP work partitioning
- Hormozd Gahvari
  - AMG application modeling
- Marc Snir and William Kramer
  - Performance model advocates
- Abhinav Bhatele
  - Process/node mapping
- Elena Caraba
  - Nonblocking Allreduce in CG
- Van Bui
  - Performance model-based evaluation of programming models
- Ankeeth Ved
  - Model-based updates to NAS benchmarks
- Funding provided by:
  - Blue Waters project (State of Illinois and the University of Illinois)
  - Department of Energy, Office of Science
  - National Science Foundation