At Exascale, Will Bandwidth Be Free?

Samuel Williams
Lawrence Berkeley National Laboratory
SWWilliams@lbl.gov

Background…

- Exascale machines will provide orders of magnitude greater ***potential*** performance and memory capacity.
- We will use those machines for new science / analysis capabilities and **not simply to run larger problems with finer grids**.
- Thus we **cannot rely on having larger local working sets** to amortize other artifacts of system performance.
- We must be focused on the impact of exascale on **important applications** and **not certain benchmarks**

Overarching questions…

- Given there are 100M-1B lines of legacy code, we need to know how well today's apps (**as written today**) will run at exascale? (need a pre-silicon answer)

- When codes are predicted to perform poorly (less than the exaflop expectorations), **what constrained their performance** (software, architecture, algorithm)

- How can these performance deficiencies be fixed?
  - new compilation techniques
  - programming models
  - algorithms
  - architectural research

- If it is addressed, what is the **ultimate performance potential.**

Today, Flop's are Free

- Over the last few decades, improvements in peak flop/s have quickly outstripped improvements in peak bandwidth.
- Today, it is **>50x more expensive** to move a word from DRAM than to perform a floating-point operation on it.

- The kernels in most real applications exhibit such limited data locality that there is O(1) words of data movement per floating-point operation.
- With data movement now the primary bottleneck to performance, some quip that **"flops are free"**
- This paradigm is not invalidated just because one can find or construct a benchmark with sufficient locality to be compute-bound (linpack).

What about at Exascale?

- The lack of locality will persist (flops are still free)
- However, a new set of challenges may emerge…
  - synchronization across thousands of cores (this overhead is required for OpenMP and CUDA threading constructs)
  - insufficient parallelism (expressed or inherent) in kernels
  - complex data movement interactions on both inter-core and inter-node networks

- If these displace data movement as the preeminent bottleneck, then in general, **data movement (bandwidth) may seem free !**
  - Today, flop/s are over provisioned given the lack of locality in real apps
  - **At Exascale, Bandwidth may be over provisioned due to a lack of parallelism in real applications.**
  - This rule of thumb for exascale wouldn't be invalidated just because one can find or construct (e.g. STREAM) a benchmark for which bandwidth or flop/s is the bottleneck.

Overhead of Synchronization

(Coarse vs. Fine-grained Parallelism)

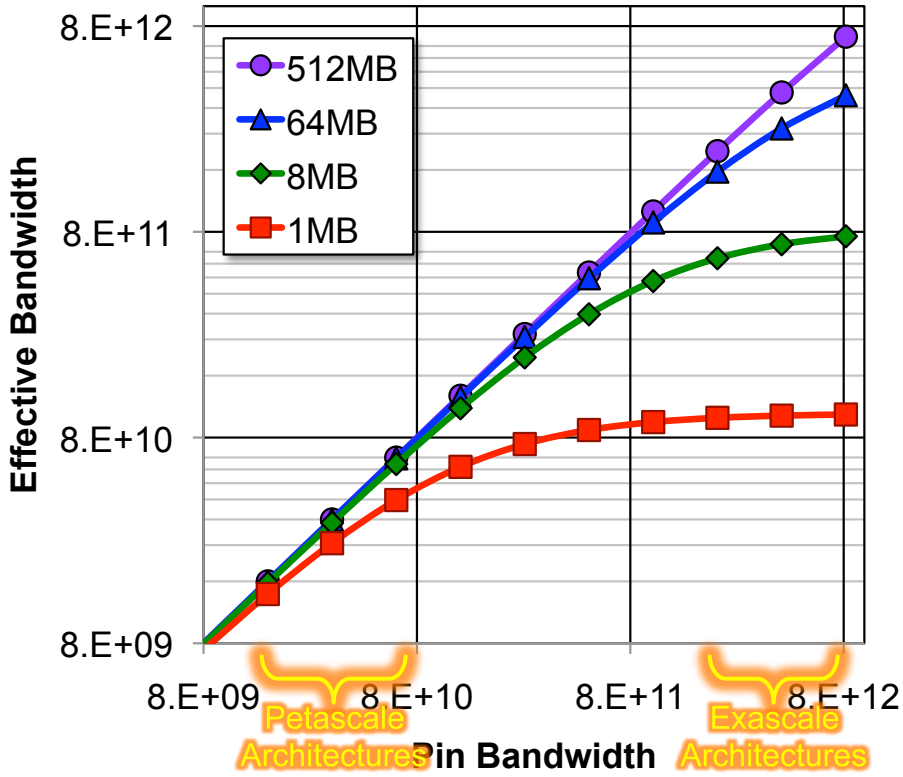
Overheads for Threading

- OpenMP-threaded applications are in effect bulk synchronous…
  - …sequential code…
  - …OpenMP region (with implied barrier at end)…
  - …sequential code…
- MPI communication can similarly act as a synchronization point.
- This overhead for synchronization sets limits on the application of fine-grained parallelism.

- We can model the execution of a loop nest as some function of…
  - computation (flop's)
  - data movement
  - synchronization overhead (e.g. OpenMP overheads)
- When the cost data movement is high, it seems like "flops are free", ***but when the cost of synchronization is high, it will seem like data movement is free…***

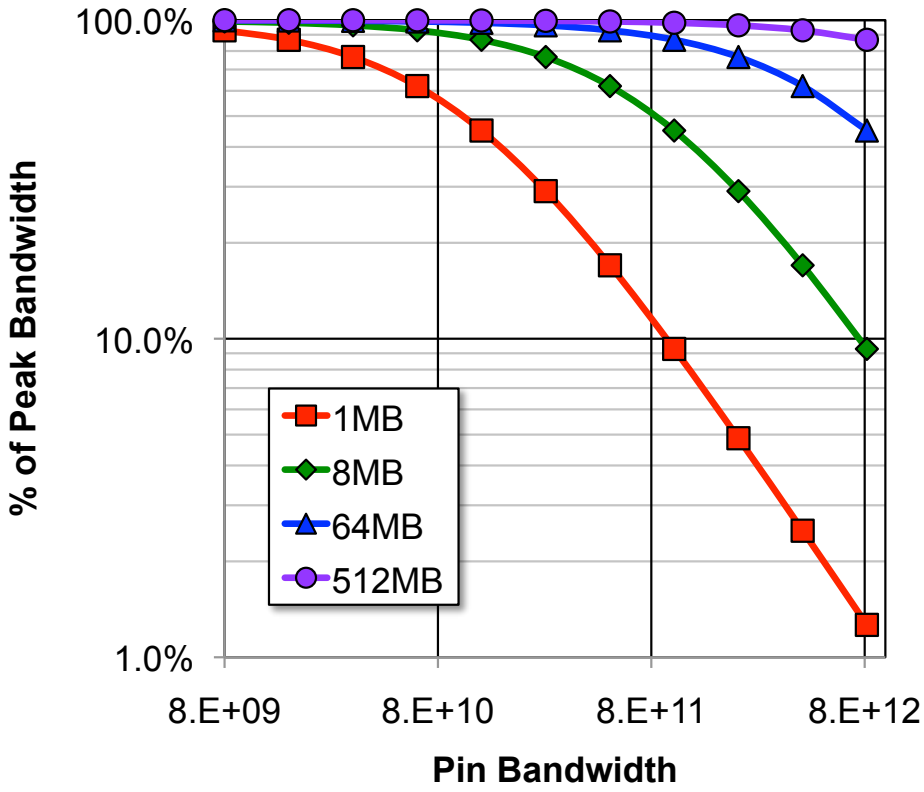
Overheads for Threading

- Consider a nondescript application where threading is applied in bulk to units of work (e.g. matrices, vectors, boxes…)
- Each unit requires a threading construct to start, run, synchronize, and terminate. (e.g. omp parallel region or CUDA kernel)
- Assuming data movement would nominally dominate, we can easily model the effective bandwidth as a function of…
  - the processor's pin bandwidth
  - the size of the unit of work
  - fixed 10us overhead (no worse when we have 100x more cores)



Overheads for Threading

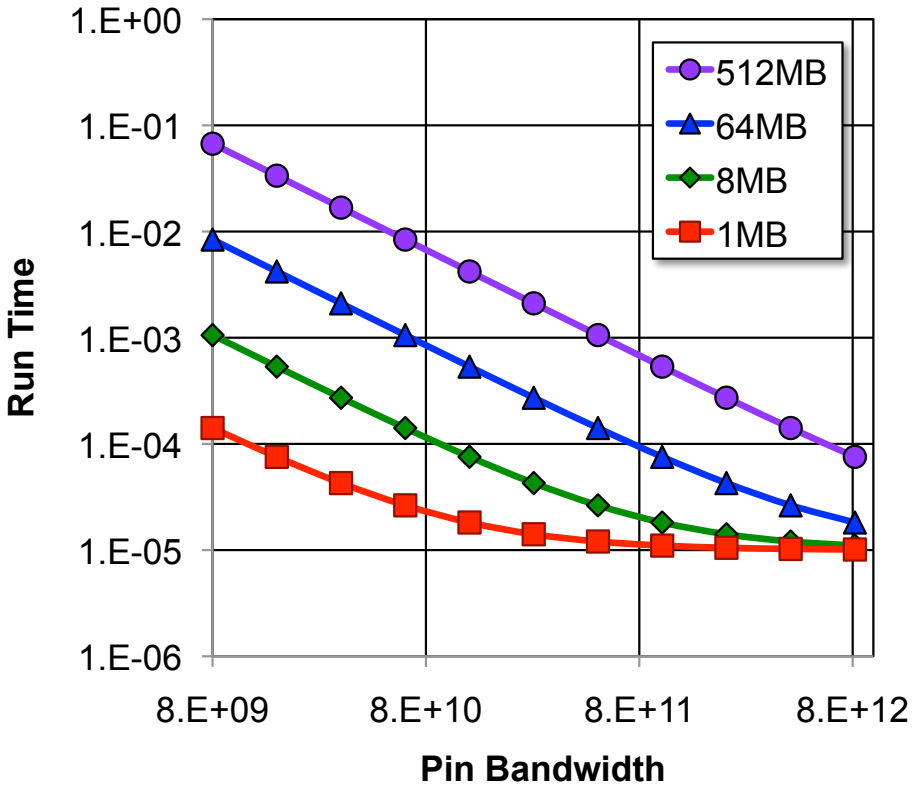
- …viewed as a fraction of the pin (peak) bandwidth…
- For today's architectures (left) it is **easy to get near peak bandwidth**.
- At exascale, the usual approach to threading computation on objects less than 8MB (e.g. an 64^3 grid), **will never achieve better than 10% of peak *bandwidth* !!**
- In fact, threading coarse 512MB objects only attains 87% of pin bandwidth.



- Similar conditions arise on compute-limited kernels…
- To attain 90% of peak (10TF), each processor needs to perform approximately 100 million floating-point operations between synchronization points.

Overheads for Threading

- We can also visualize the time required to perform the loop nest.
- For applications that apply coarse-grained parallelism (purple) any additional bandwidth is beneficial
- Applications that attempt fine-grained parallelization **can not exploit more than 1TB/s** of bandwidth if overheads remain around 10us.



Overheads for Threading
(summary of issues)

- If we were to simply compile and run an existing application on a exascale processor, we may see minimal increases in performance and **may only attain 10% of peak bandwidth** (and a smaller fraction of peak flop/s).

- Clearly, it is imperative we have tools today that …
  - precisely **identify where application of TLP is inefficient** so that the codes can be fixed before the machine is on the floor (pre-silicon).
  - identify what the **synchronization minimizing implementation** is.
- Vendors may use this information to drive their design point. i.e.
  - can they get by with 10us overheads at 10K cores?
  - or do they need sub microsecond overheads for global synchronization?
- From a CS research perspective, knowing how critical this issue is, allows us to quantify the **importance of alternate execution and threading models**.

Insufficient Parallelism ?

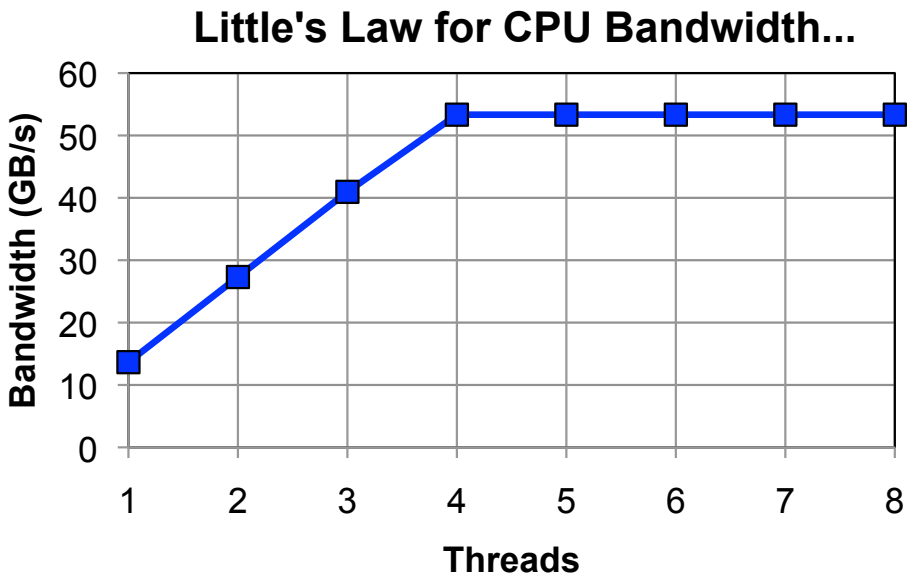
Insufficient Parallelism in code

- Thus far, we have assumed our effective bandwidth is simply peak bandwidth amortized by overhead/synchronization time
- Unfortunately, kernels as written today often attempt to apply a simple **`#pragma omp parallel for`** construct to various loops or iteration spaces.
  - In the multicore era with 4-8 cores, even short (64-iteration) outer loops can be efficiently parallelized in this manner.
  - As the number of threads increases, load balancing will emerge (64 iterations on 14 SM's or 57 cores).
  - Ultimately, some cores may not receive any loop iterations (128 iterations on 10K cores).

- ***What's the effect of under parallelizing a kernel?***
  - The effect of insufficient parallelism on peak flop/s is obvious.
  - The effect on peak bandwidth can be subtle --- **Specifically, how many threads are needed to fully saturate the memory subsystem ?**

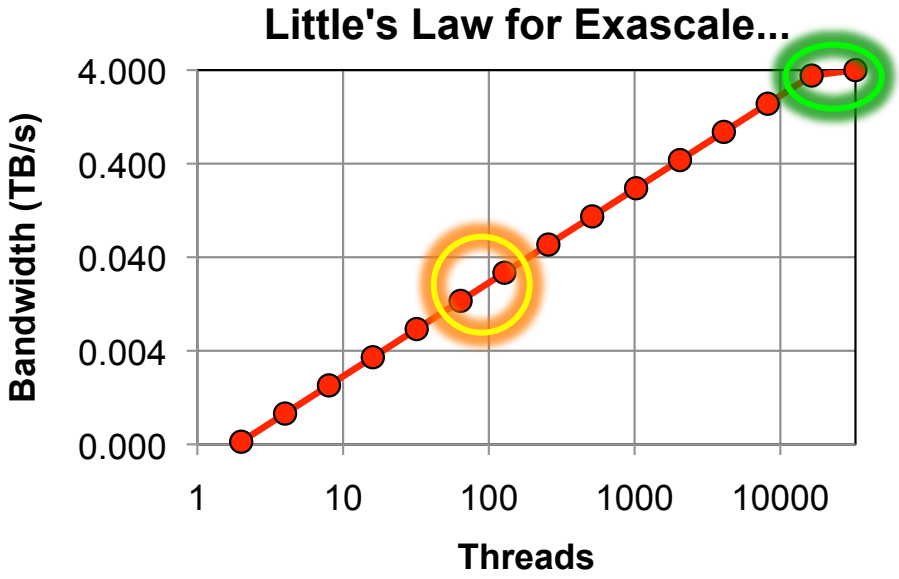
Insufficient Parallelism in code

- We can model Little's Law = omnipresent concurrency needed to fully utilize the memory bandwidth.
- On today's CPU architectures, OOO+HW prefetchers can inject sufficient concurrency per thread to hit peak BW with a subset of the cores.



Insufficient Parallelism in code

- We can model Little's Law = omnipresent concurrency needed to fully utilize the memory bandwidth.
- On today's CPU architectures, OOO+HW prefetchers can inject sufficient concurrency per thread to hit peak BW with a subset of the cores.



- However on exascale architectures, where power-hungry OOO and prefetchers have been eliminated in favor of multithreading, we need **10s of thousands of threads to get peak bandwidth**.
- Today, many applications may only express 64- to 128-way TLP. If run at exascale, **they may only attain 1% of peak bandwidth !!**
- Needing all the threads always running for max BW **may cast doubt on powering down cores to save energy** when BW limited.

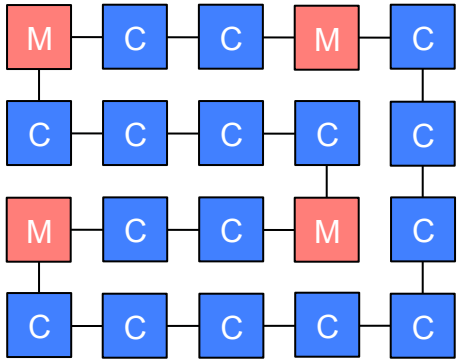
Insufficient Parallelism in code (summary of issues)

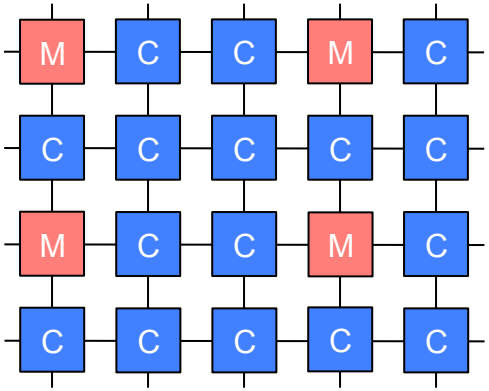
- Need tools (or info from vendors) that…
  - **quantify the relationship between TLP and effective bandwidth**. (today, we simply run benchmarks like stream)
  - highlight how a processor responds to **increasing the number of streams per thread** (today, on BGQ, having too many threads can cause contention in the prefetchers can impede net bandwidth)
  - **precisely identify regions of applications that are under parallelized and quantify the penalty**.
  - provide a **high-water mark for parallelism**. When algorithmic high-water mark is less than the demands of the processor, significant algorithmic rework is mandated.

Do we have to worry about NoC Topologies as well ?

On-Chip Network (NoC)…

- When we had simple on-chip networks (e.g. star or crossbars), performance was readily understood and could be easily modeled.
- unfortunately, programmers (let alone software or runtimes) lack intuition for even simple topologies likes rings and meshes.
- how differently do applications perform on these two NoC topologies?
- When scaled to 1000's of cores, modeling and simulation tools will be essential in predicting and understanding performance.





On-Chip Network (NoC)…

- Unfortunately, with scalable yet complex NoC topologies…
  - memory bandwidth and latencies may **vary from core to core.**
  - requisite concurrency will vary from core to core (overprovision?)
- Moreover, highly-optimized applications will perform **complex, on-chip inter-thread data exchanges** avoid DRAM bottlenecks.
  - This creates a **complex interaction between NoC topology and the communication pattern**
  - programmers are ill-equipped to quantify these effects (**intuition fails**)

- We cannot assume core injection bandwidth is the bottleneck anymore than we can assume DRAM bandwidth is the bottleneck.
- The NoC may be the bottleneck for real applications.

On-Chip Network (NoC)
… summary of issues

- We must be **focused on real application drivers** and not contrived benchmarks that highlight a particular NoC topology…

- We tools that …
  - determine **if streaming DRAM bandwidth is sensitive to NoC topology** in the presence of 10's of thousands of threads/requests.
  - can identify **which real codes / kernels are sensitive** to on-node (inter-thread) synchronization and data exchanges.
  - can **quantify the performance** of complex data exchanges for various NoC topologies and parameters expected in exascale machines.
  - how **placement of threads** (or variability in the placement of threads) effects performance or latency.
  - Does variability demand **new approaches to load balancing?**
  - Would novel NoC topologies/functionality **enable new algorithms?**
- Static analysis may address some of these, but increasingly component simulation or processor emulation are required

Algorithms and Distributed Memory

Algorithmic Effects and MPI
(Global Interconnect)

- Many algorithms have >O(N) computational complexity
- At exascale…
  - although we may have 100x more total (NVRAM) memory per node, exascale is not about finer meshes. It is about enabling new science.
  - We may run lots of **similarly sized "solves" per node** as today on 1-10x more nodes than today (perhaps 10x bigger overall).
  - Larger problems may require more iterations, but message size may not increase significantly.
- We need to run **scaling experiments to understand the algorithmic effects** that increase time-to-solution.
- **We need network models and simulators** that allow us to understand the performance challenges of **real applications**
  - P2P communication on **smallish** messages
  - **limited injection rates** (MPI_THREAD_PARALLEL is rare)
  - collectives at full machine scale

Summary

Summary…

- Just as a lack of locality makes flop/s seem free today, we've enumerated a number of hazards that might make **bandwidth seem free at exascale** for the key applications we have today.

- Rather than dictating solutions, we have enumerated a list of **key questions** that need to be answered via modeling and simulation.
- Using the results from this effort will allow one to identify where the major roadblocks are on the road to exascale.

Acknowledgements

- All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231.

Questions?