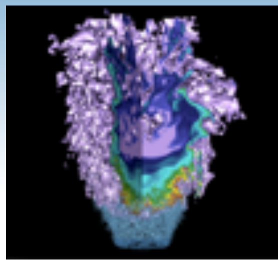
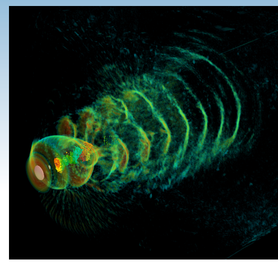
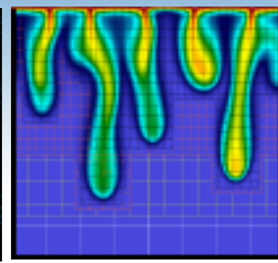
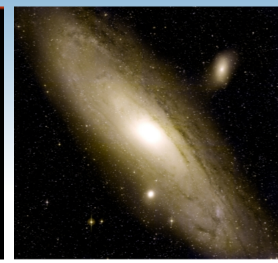
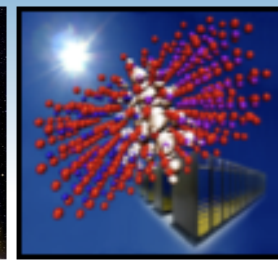
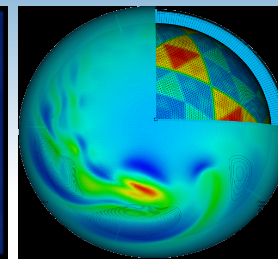







To Virtualize or Not to Virtualize?

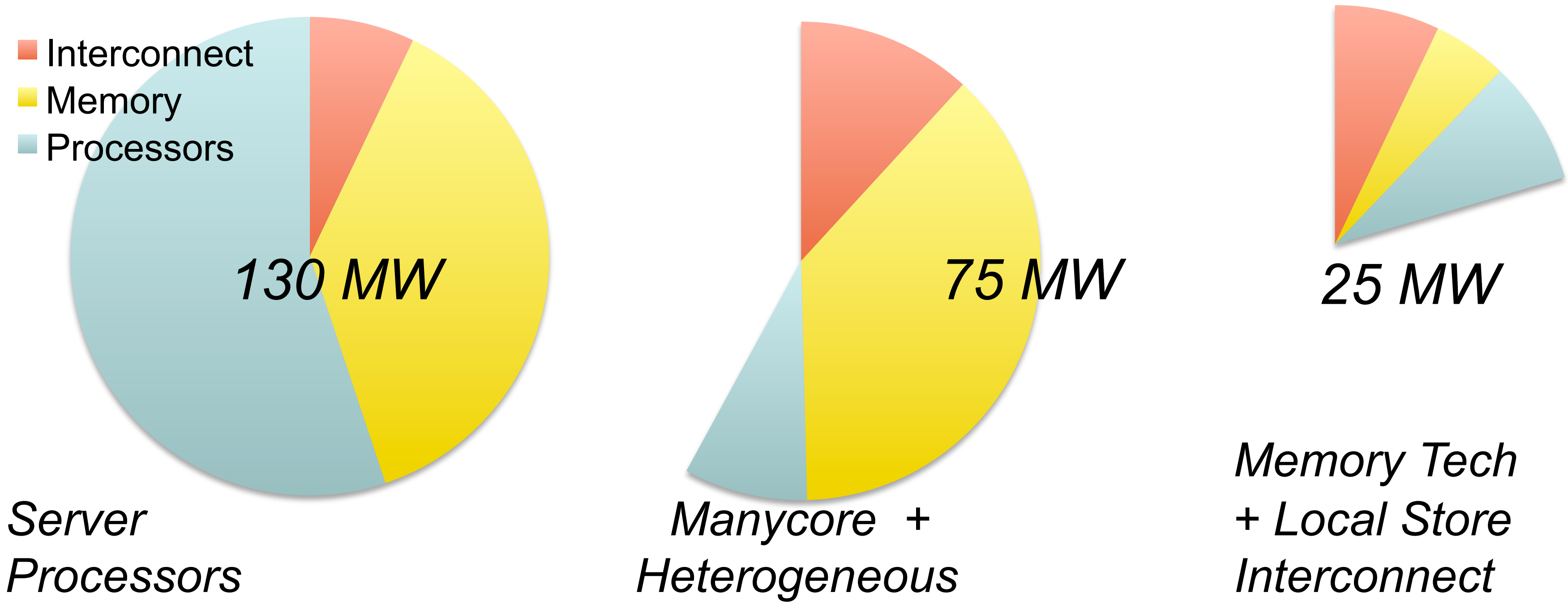
Kathy Yelick

Associate Laboratory Director and NERSC Director

Lawrence Berkeley National Laboratory

EECS Professor, UC Berkeley

New Processors Means New Software



- **Exascale will have chips with thousands of tiny processor cores, and a few large ones**
- **Architecture is an open question:**
  - Sea of embedded cores with heavyweight “service” nodes
  - Lightweight cores are accelerators to CPUs
- **Software managed memory and interconnect topology?**

Challenges to ~~Exascale~~

Performance Growth

1) **System power** is the primary constraint
2) **Concurrency** (1000x today)
3) **Memory** bandwidth and capacity are not keeping pace
4) **Processor** architecture is open, but likely heterogeneous
5) **Programming model** heroic compilers will not hide this
6) **Algorithms** need to minimize data movement, not flops
7) **I/O bandwidth** unlikely to keep pace with machine speed
8) **Reliability and resiliency** will be critical at this scale
9) **Bisection bandwidth** limited by cost and energy

Unlike the last 20 years most of these (1-7) are equally important across scales, e.g., 1000 1-PF machines

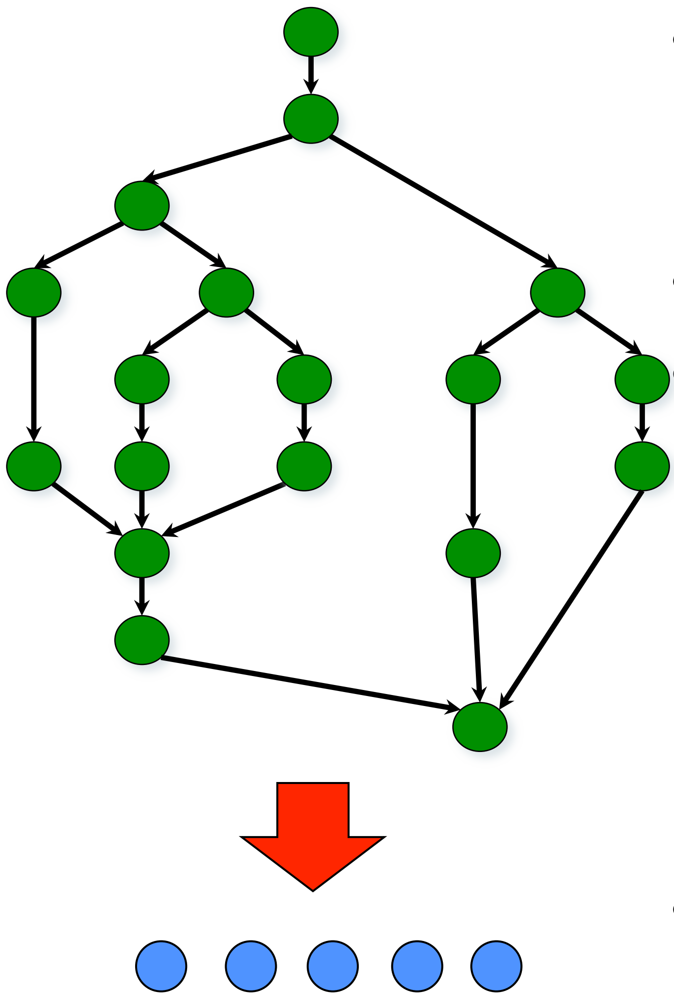
To Virtualize or Not

- The fundamental question facing in the design of parallel programming models is:

  What should be virtualized?

- Hardware has finite resources with complex structures:
  - Processor count, register, link topology, is finite
  - On chip memory is finite: caches hide this, local stores do not
- Does the programming model expose this or hide it? E.g., one thread per core, or many?
  - Many threads may have advantages for load balancing, fault tolerance and latency-hiding
  - But one thread is better for deep memory hierarchies, i.e., a many to few load balancer tends to work better on shared memory than distributed
- Which level is responsible for virtualizing?

Virtualization of Processors



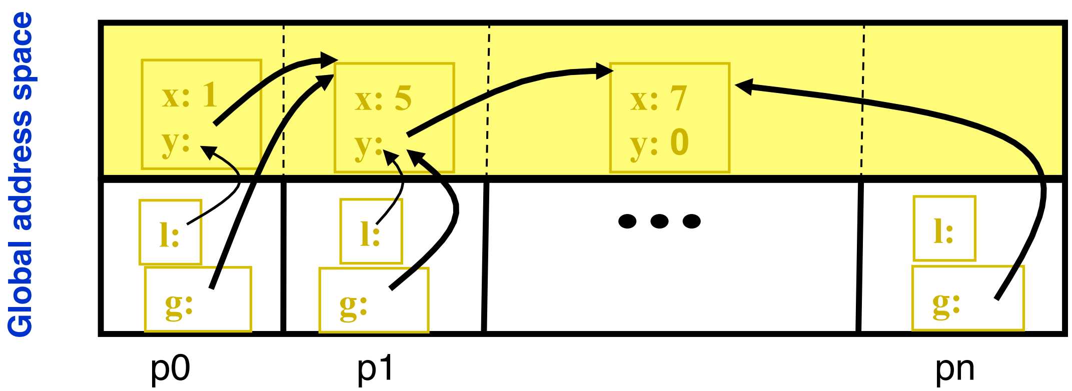
- Many possible tasks graphs, depending on how much parallelism is exposed
- Abstraction can constrain this
- Where does the mapping of the graph to a particular number of processors happen?
  - The compiler: NESL, ZPL
  - The runtime system : Cilk, Charm++, OpenMP, X10, Chapel
  - The programmer: MPI, UPC
- Data decomposition then computation scheduling?
- Fairness and resource management are subtle

Irregular vs. Regular Parallelism

- Computations with regular task graphs can be automatically virtualized / scheduled
  - By a compiler or runtime system
- Fork/Join graphs (no out-of-band dependencies) can be scheduled
  - By a runtime system (e.g., Cilk)
  - A greedy scheduler (stealing or pushing) is optimal time
  - Stealing is optimal in space (but slower to load balance)
- General DAGs are more complicated
  - Either preemption or user awareness is needed
- Conclusion: If your computation is not regular, the runtime system should be dynamic, i.e., virtualize the processors

Virtualizing Memory Structure

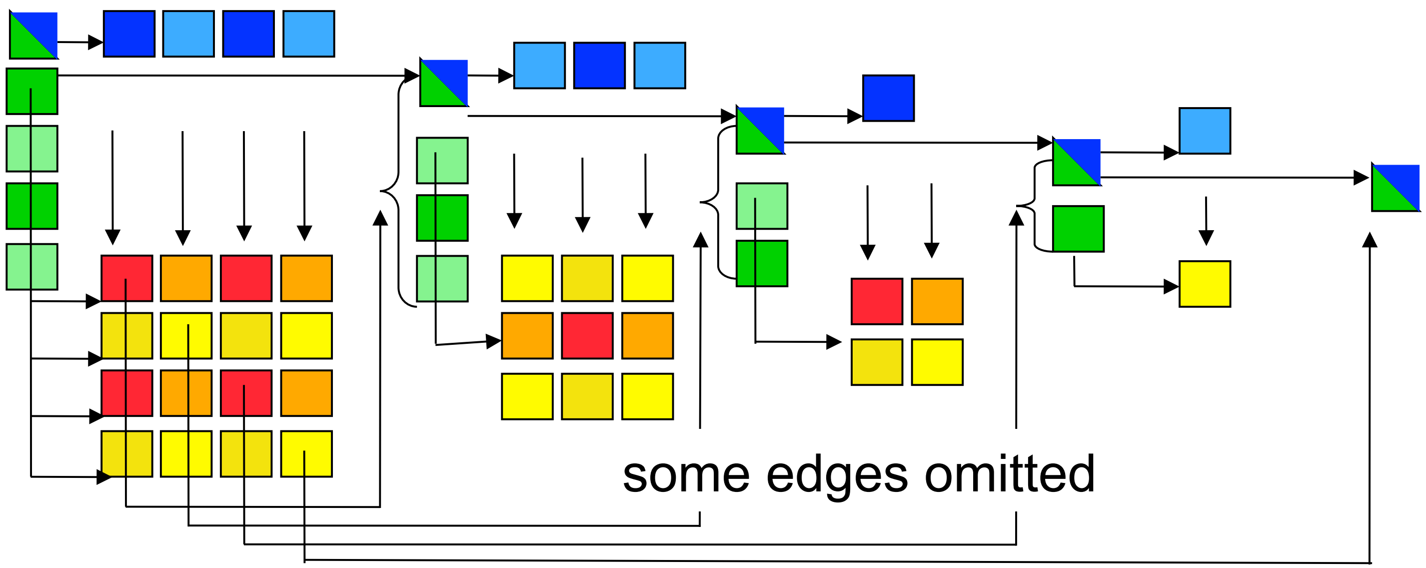
- **Should we hide memory locality or make them visible to the programmer?**
  - **Can programmers optimize locality? Not in OpenMP**
  - **Must the programmer optimize for locality? MPI**
  - **Can it be optional? PGAS**
- **Can Cache-oblivious or over-partitioned approaches work at scale (locality costs at scale)?**
- **Can we have portable mechanisms for locality optimization that are good enough?**



Load Balancing with Locality

- UPC uses a static threads (SPMD) programming model
  - No dynamic load balancing built-in
- Berkeley compiler has some extensions
  - Allows programmers to execute active messages (AMs)
  - AMs have limited functionality (no messages except acks) to avoid deadlock in the network
- A more dynamic runtime would have many other uses
  - Application load imbalance, OS noise, fault tolerance
- Two extremes are well-studied
  - Dynamic parallelism without locality
  - Static parallelism (with threads = processors) with locality
- What issues do we run into if we want dynamic threads with locality?

Memory Constrained Scheduling



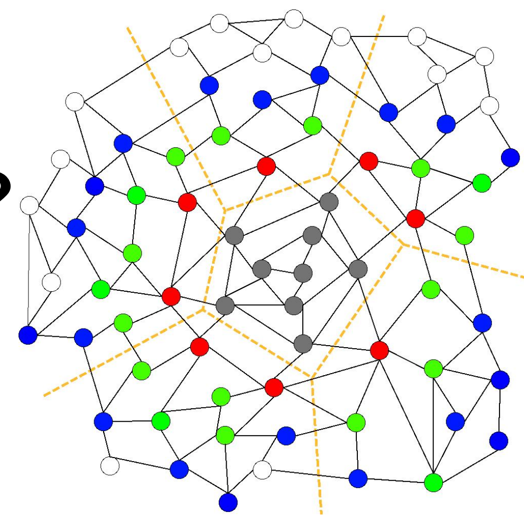
- **Theoretical and practical problem: Memory deadlock**
  - **Not enough memory for all tasks at once.**
    - **(Each update needs two temporary blocks, a green and blue, to run.)**
  - **If updates are scheduled too soon, you will run out of memory**
  - **Allocate memory in increasing order of factorization:**
    - **Don't skip any!**
  - **Thread blocks until enough memory available**

What to Virtualize?

- Register count: hide
  - Compilers prove this, but need autotuners
- Separate address spaces: hide
  - PGAS proves this; Needs to show for GPUs
- Performance-partitioned memory: expose
  - For distributed memory needs to be exposed
- Number of cores: depends
  - On shared memory, we can virtualize
  - Distributed memory mostly not
  - Open question for non-SPMD PGAS

Aside: Communication-Avoiding Algorithms

- Sparse Iterative (Krylov Subpace) Methods
  - Nearest neighbor communication on a mesh
  - Dominated by time to read matrix (edges) from DRAM
  - And (small) communication and global synchronization events at each step



- Can we lower data movement costs?
  - Take *k* steps with one matrix read from DRAM and one communication phase
    - Serial: O(1) moves of data moves vs. O(k)
    - Parallel: O(log p) messages vs. O(k log p)
- Can we make communication provably optimal?
  - Communication both to DRAM and between cores
  - Minimize independent accesses ('latency')
  - Minimize data volume ('bandwidth')

Joint work with Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin

Optimizing for Communication ≠ Ignore Running Time

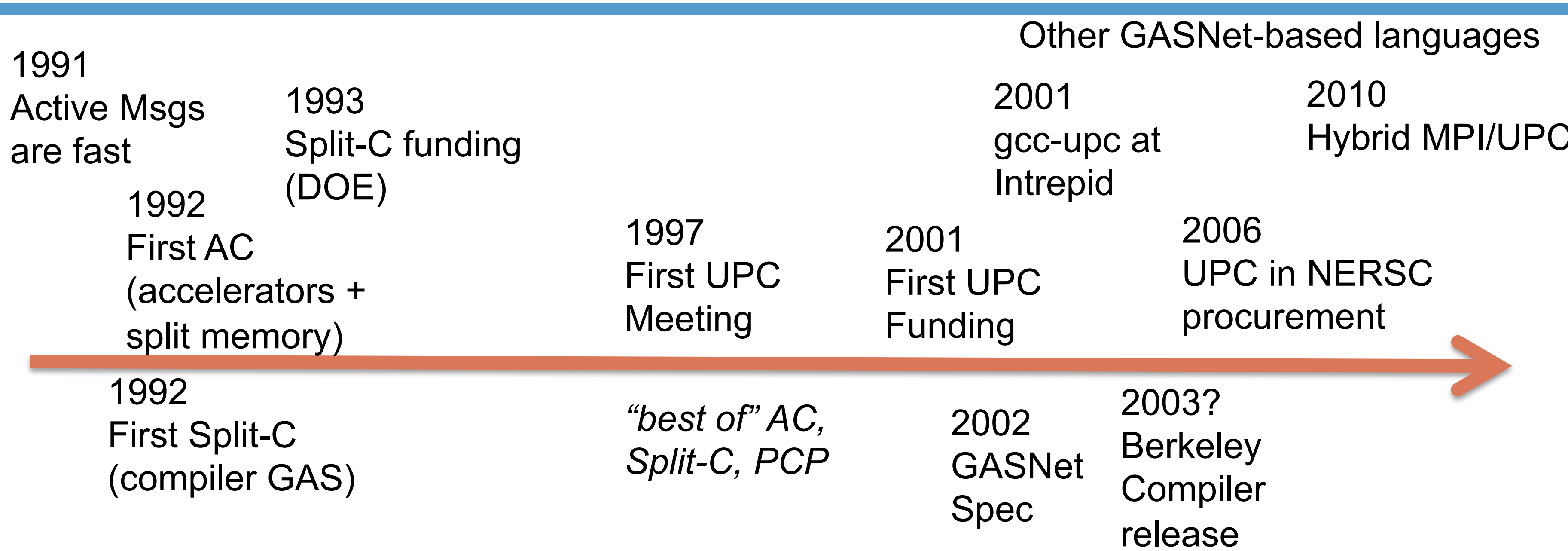
Complexity of 2D Poisson Equation with N unknowns

Algorithm	Serial	PRAM	Memory	#Procs
Dense LU	N^3	N	N^2	N^2
Band LU	N^2	N	$N^{3/2}$	N
Jacobi	N^2	N	N	N
Explicit Inv.	N^2	log N	N^2	N^2
Conj.Grad.	$N^{3/2}$	$N^{1/2}$ *log N	N	N
RB SOR	$N^{3/2}$	$N^{1/2}$	N	N
Sparse LU	$N^{3/2}$	$N^{1/2}$	N*log N	N
FFT	N*log N	log N	N	N
Multigrid	N	$\log^2 N$	N	N
Lower bound	N	log N	N	

Good ideas taken to the extreme become bad:

- Don’t use dense LU where something smaller/faster will work
- Don’t use a dense matrix rather than sparse (but do fill in some zeros if that makes it faster)

The UPC Experience

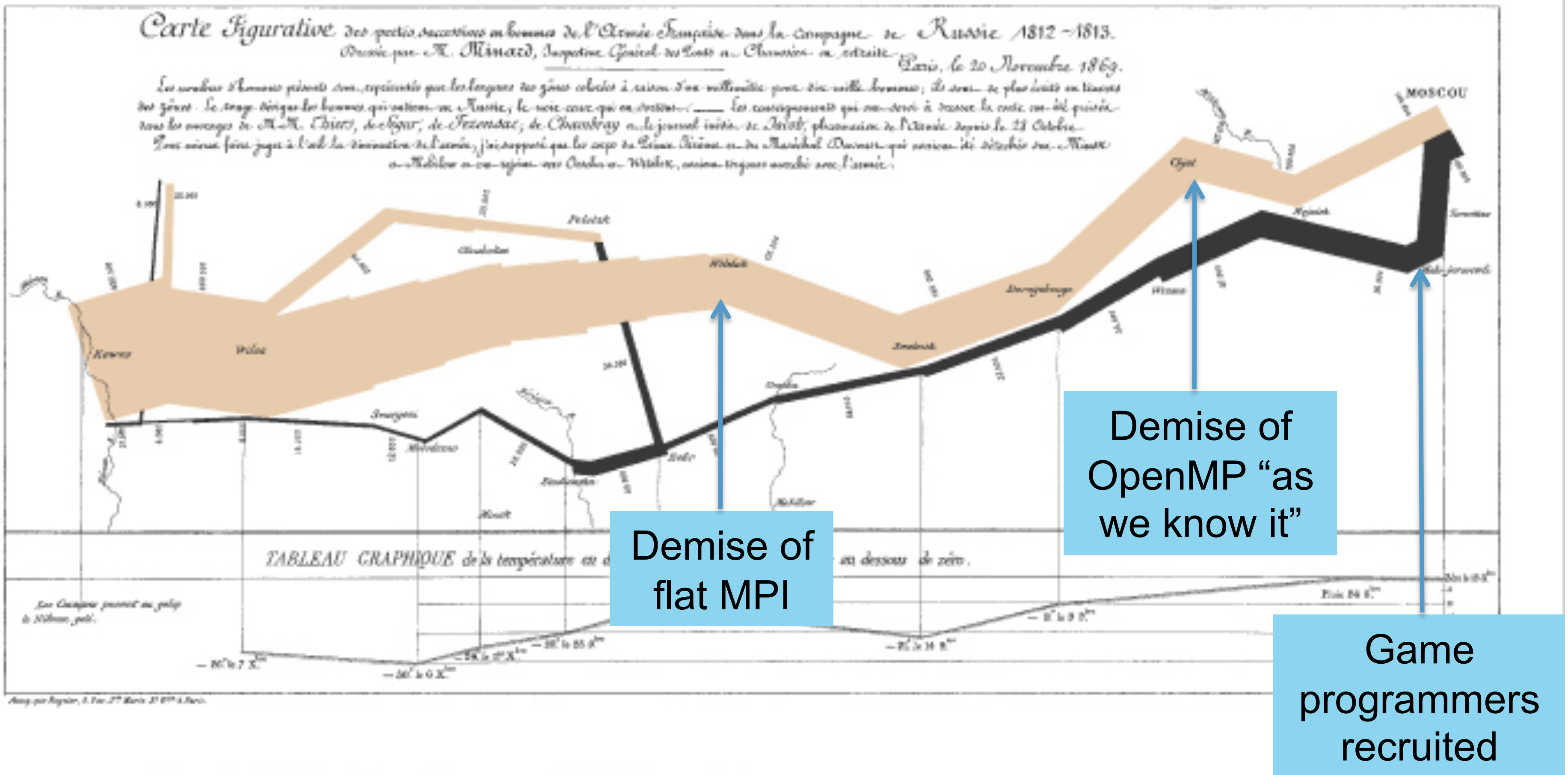


- **Ecosystem:**
  - Users with a need (fine-grained random access)
  - Machines with RDMA (not hardware GAS)
  - Common runtime
  - Commercial and free software
  - Center procurements
  - Sustained many-year funding

Conclusions

- Solve the problems that must be solved
  - Locality (how many levels are necessary?)
  - Heterogeneity
  - Vertical communication management
    - Horizontal is solved by MPI (or PGAS)
  - Fault resilience, maybe
    - Look at the 800-cabinet K machine
  - Dynamic resource management
    - Definitely for irregular problems
    - Maybe for regular ones on “irregular” machines
  - Resource management for dynamic distributed runtimes

Keeping the Users with Us



Napoleon's March to Moscow   The War of 1812

Charles Joseph Minard

This classic of Charles Joseph Minard (1781-1870), the French engineer, shows the terrible fate of Napoleon's army in Russia. Described by E. J. Marey as seeming to defy the pen of the historian by its brutal eloquence, this combination of data map and time-series, drawn in 1869, portrays the devastating losses suffered in Napoleon's Russian campaign of 1812. Beginning at the left on the Polish-Russian border near the Niemen River, the thick band shows the size of the army (422,000 men) as it invaded Russia in June 1812. The width of the band indicates the size of the army at each place on the map. In September, the army reached Moscow, which was by then sacked and deserted, with 100,000 men. The path of Napoleon's retreat from Moscow is depicted by the darker, lower band, which is linked to a temperature scale and dates at the bottom of the chart. It was a bitterly cold winter, and many froze on the march out of Russia. As the graphic shows, the crossing of the Berezina River was a disaster, and the army finally struggled back into Poland with only 10,000 men remaining. Also shown are the movements of auxiliary troops, as they sought to protect the rear and the flank of the advancing army. Minard's graphic tells a rich, coherent story with its multivariate data, far more enlightening than just a single number bouncing along over time. Six variables are plotted: the size of the army, its location on a two-dimensional surface, direction of the army's movement, and temperature on various dates during the retreat from Moscow. It may well be the best statistical graphic ever drawn.

Edward R. Tufte, *The Visual Display of Quantitative Information*   Graphics Press Box 430 Cheshire, Connecticut 06410

We are both too early and too late for an exascale programming model

→ Focus in critical general challenges