

Changing How Programmers Think about Parallel Programming

William Gropp

www.cs.illinois.edu/~wgropp



Association for
Computing Machinery

Advancing Computing as a Science & Profession



ACM Learning Center

<http://learning.acm.org>

- 1,350+ trusted technical books and videos by leading publishers including O'Reilly, Morgan Kaufmann, others
- Online courses with assessments and certification-track mentoring, member discounts on tuition at partner institutions
- Learning Webinars on big topics (Cloud Computing/Mobile Development, Cybersecurity, Big Data, Recommender Systems, SaaS, Agile, Natural Language Processing)
- ACM Tech Packs on big current computing topics: Annotated Bibliographies compiled by subject experts
- Popular video tutorials/keynotes from ACM Digital Library, A.M. Turing Centenary talks/panels
- Podcasts with industry leaders/award winners



Association for
Computing Machinery

Advancing Computing as a Science & Profession

Talk Back

- Use the [Facebook](#) widget in the bottom panel to share this presentation with friends and colleagues
- Use [Twitter](#) widget to Tweet your favorite quotes from today's presentation with hashtag [#ACMWebinarGropp](#)
- Submit questions and comments via Twitter to [@acmeducation](#) – we're reading them!

Outline

- Why Parallel Programming?
- What are some ways to think about parallel programming?
- Thinking about parallelism: Bulk Synchronous Programming
- Why is this bad?
- How should we think about parallel programming
- Separate the Programming Model from the Execution Model
- Rethinking Parallel Computing
- How does this change the way you should look at parallel programming?
- Example





Why Parallel Programming?

- Because you need more computing resources that you can get with one computer
 - ◆ The focus is on *performance*
 - ◆ Traditionally compute, but may be memory, bandwidth, resilience/reliability, etc.
- High Performance Computing
 - ◆ Is just that – ways to get exceptional performance from computers – includes both parallel and sequential computing



What are some ways to think about parallel programming?

- At least two easy ways:
 - ◆ Coarse grained - Divide the problem into big tasks, run many at the same time, coordinate when necessary. Sometimes called “Task Parallelism”
 - ◆ Fine grained - For each “operation”, divide across functional units such as floating point units. Sometimes called “Data Parallelism”



Example – Coarse Grained

- Set students on different problems in a related research area
 - ◆ Or mail lots of letters – give several people the lists, have them do everything
 - ◆ Common tools include threads, fork, TBB



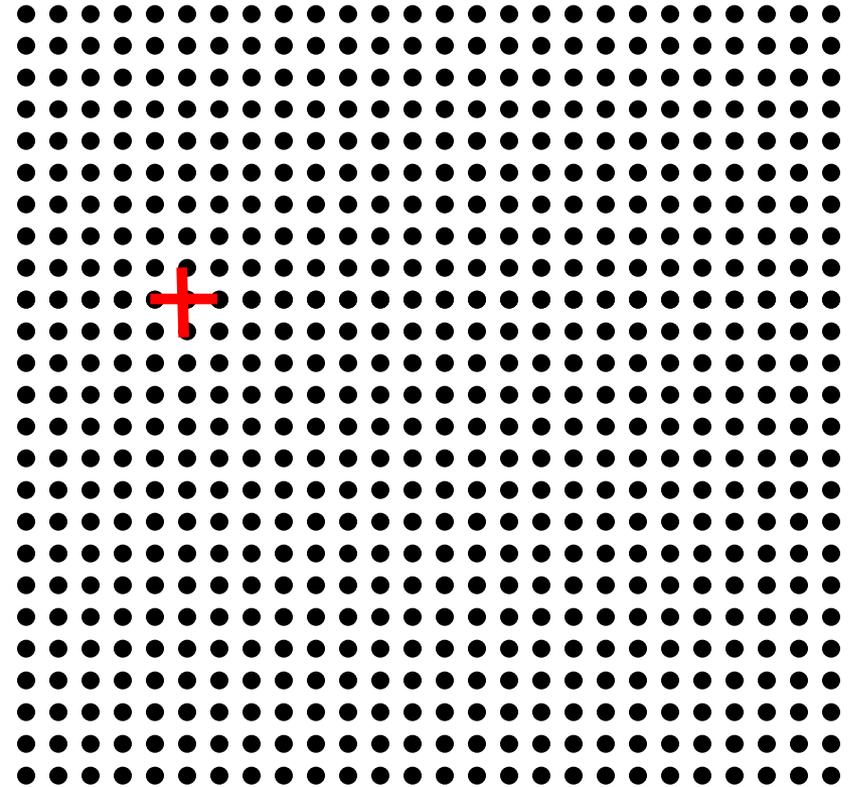
Example – Fine Grained

- Send out lists of letters
 - ◆ break into steps, make everyone write letter text, then stuff envelope, then write address, then apply stamp. Then collect and mail.
 - ◆ Common tools include OpenMP, autoparallelization or vectorization
- Both coarse and fine grained approaches are relatively easy to think about



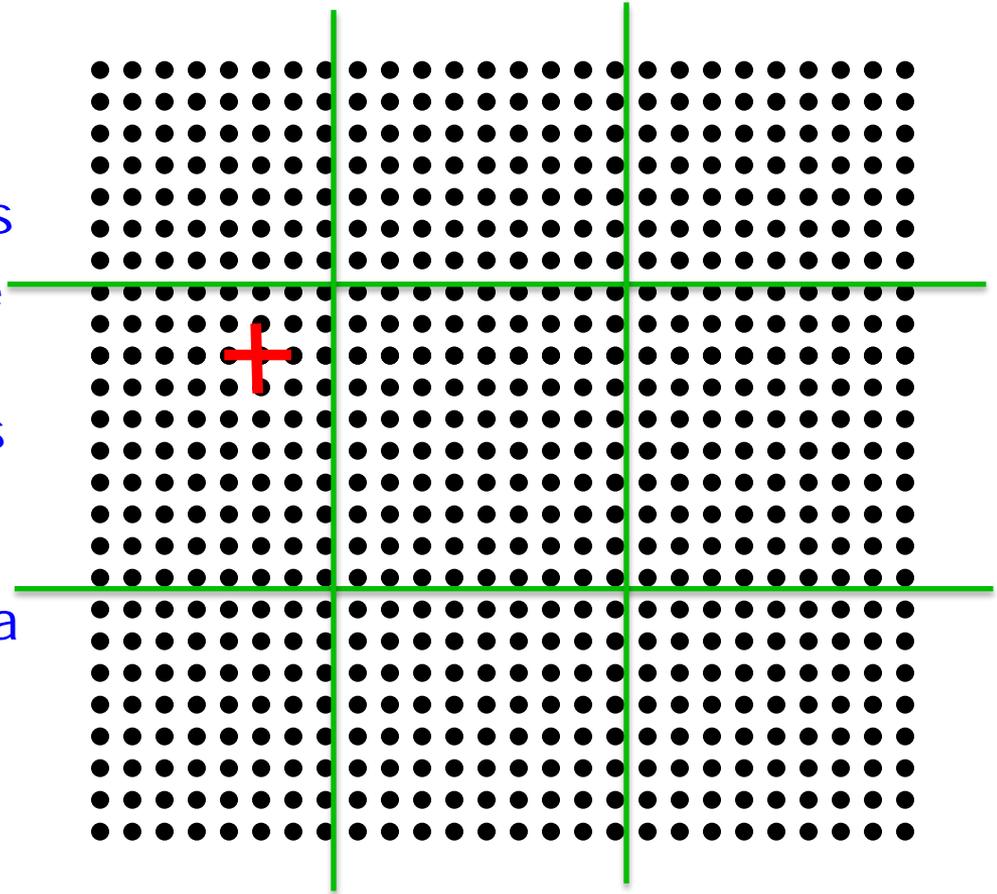
Example: Computation on a Mesh

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation $Au=f$; solving this requires computing Bv , where B is a matrix derived from A . These evaluations involve computations with the neighbors on the mesh.

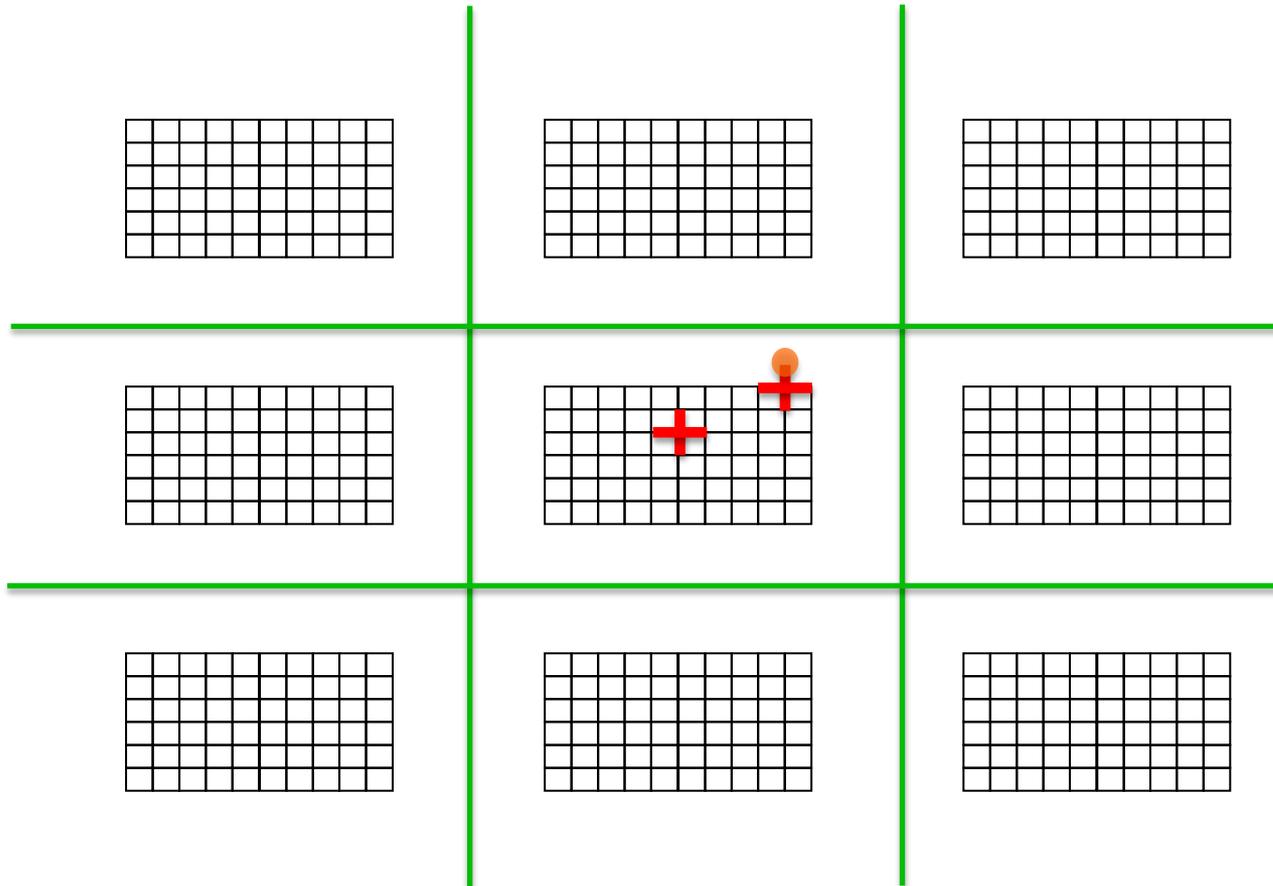


Example: Computation on a Mesh

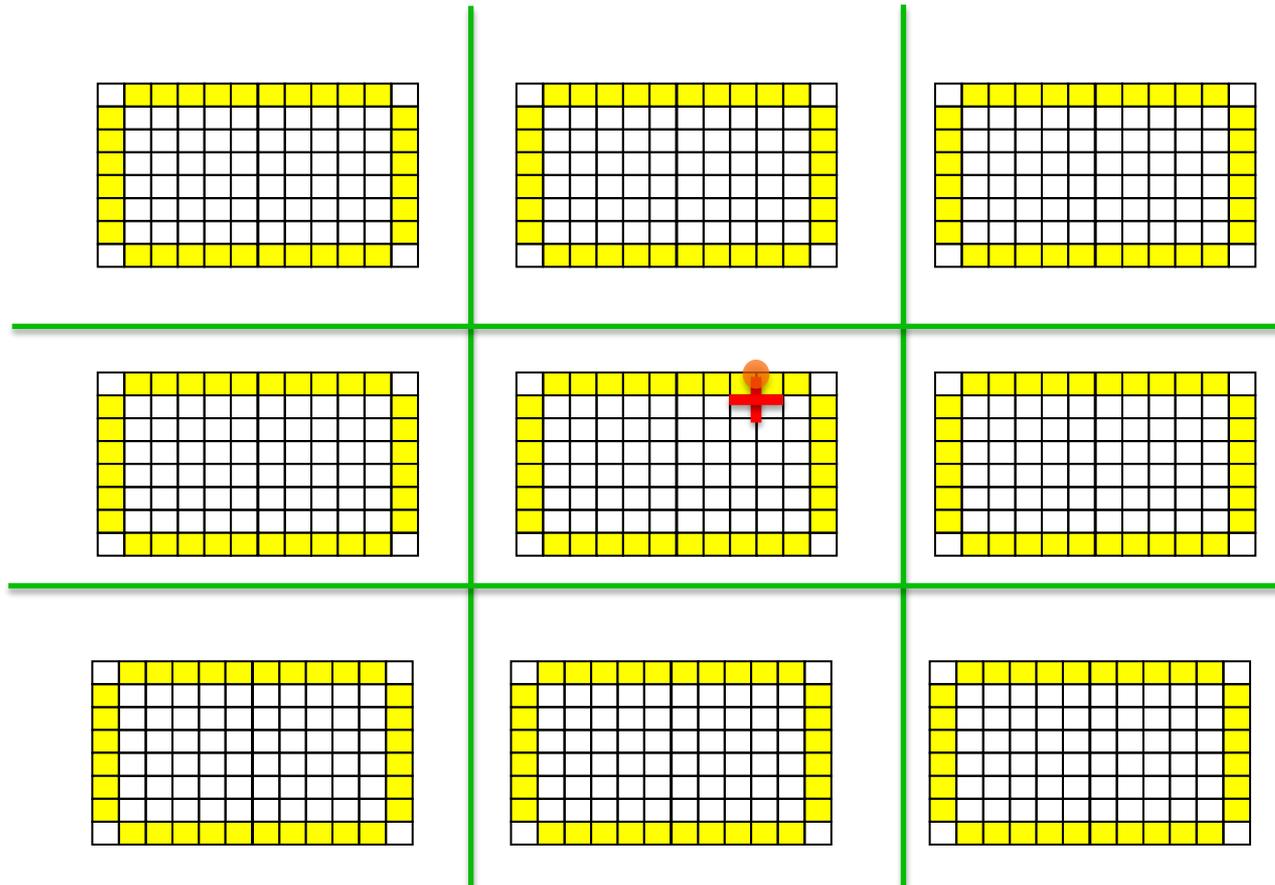
- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation $Au=f$; solving this requires computing Bv , where B is a matrix derived from A . These evaluations involve computations with the neighbors on the mesh.
- Decompose mesh into equal sized (work) pieces



Necessary Data Transfers

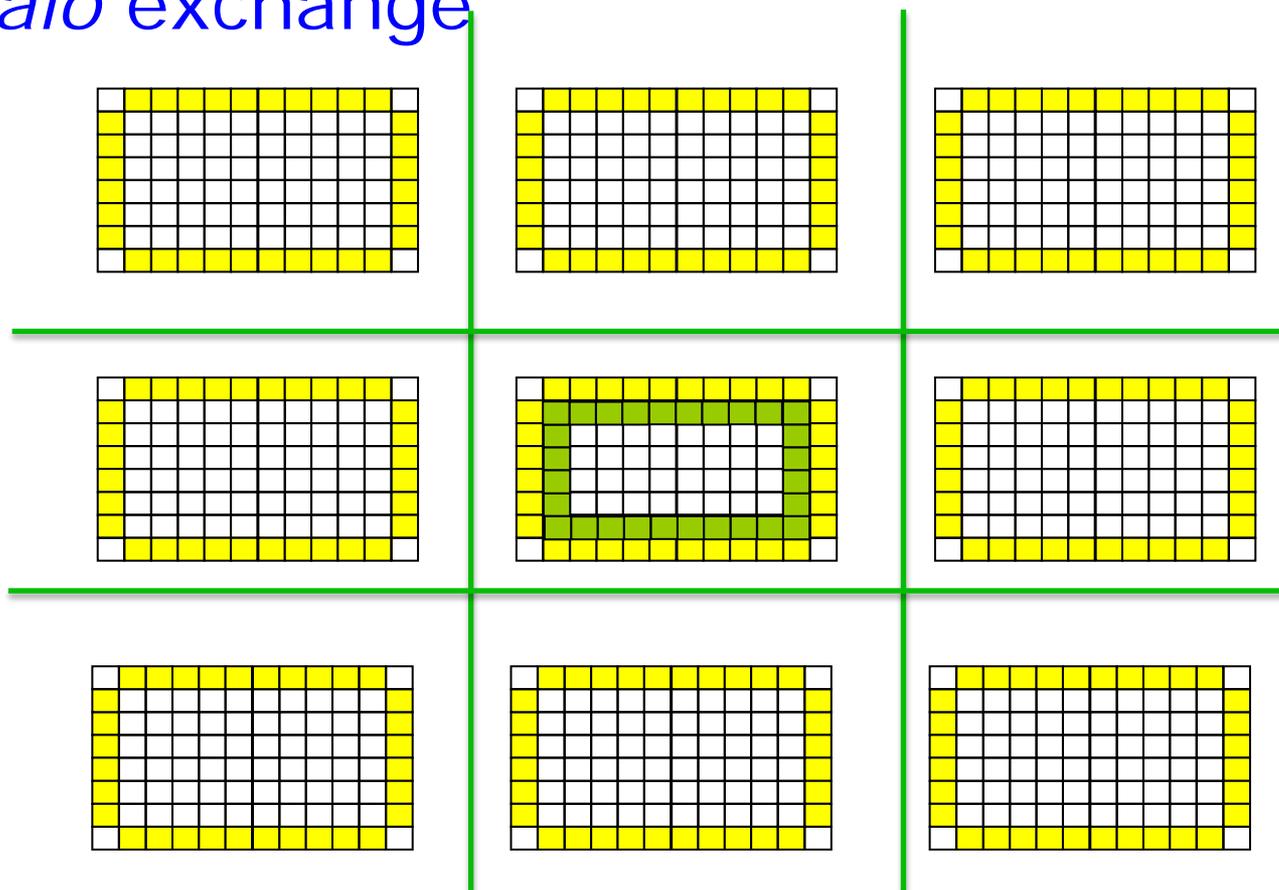


Necessary Data Transfers



Necessary Data Transfers

- Provide access to remote data through a *halo* exchange



PseudoCode

- Iterate until done:
 - ◆ Exchange “Halo” data
 - MPI_Isend/MPI_Irecv/MPI_Waitall or MPI_Alltoallv or MPI_Neighbor_alltoall or MPI_Put/MPI_Win_fence or ...
 - ◆ Perform stencil computation on local memory
 - Can use SMP/thread/vector parallelism for stencil computation – E.g., OpenMP loop parallelism



Thinking about Parallelism

- Parallelism is *hard*
 - ◆ Must achieve both correctness and performance
 - ◆ Note for parallelism, performance *is* part of correctness.
- Correctness requires understanding how the different parts of a parallel program interact
 - ◆ People are bad at this
 - ◆ This is why we have multiple layers of management in organizations



Thinking about Parallelism: Bulk Synchronous Programming

- In HPC, refers to a *style* of programming where the computation alternates between communication and computation phases
- Example from the PDE simulation

- ◆ Iterate until done:

- Exchange data with neighbors (see mesh) Communication
- Apply computational stencil Local computation
- Check for convergence/compute vector product Synchronizing communication



Thinking about Parallelism: Bulk Synchronous Programming

- Widely used in computational science and technical computing
 - ◆ Communication phases in PDE simulation (halo exchanges)
 - ◆ I/O, often after a computational step, such as a time step in a simulation
 - ◆ Checkpoints used for resilience to failures in the parallel computer





Bulk Synchronous Parallelism

- What is BSP and why is BSP important?
 - ◆ Provides a way to think about performance and correctness of the parallel program
 - Performance modeled by computation step and communication steps separately
 - Correctness also by considering computation and communication separately
 - ◆ Classic approach to solving hard problems – break down into smaller, easier ones.
- BSP formally described in “A Bridging Model for Parallel Computation,” CACM 33#8, Aug 1990, by Leslie Valiant
 - ◆ Use in HPC is both more and less than Valiant’s BSP





Why is this bad?

- Not really bad, but has limitations
 - ◆ Implicit assumption: work can be evenly partitioned, or at least evenly enough
 - But how easy is it to accurately predict performance of some code or even the difference in performance in code running on different data?
 - Try it yourself – What is the performance of *your* implementation of matrix-matrix multiply for a dense matrix (or your favorite example)?
 - Don't forget to apply this to every part of the computer – even if multicore, heterogeneous, such as mixed CPU/GPU systems
 - There are many other sources of performance irregularity – its hard to precisely predict performance





Why is this bad?

- Cost of “Synchronous”
 - ◆ Background: Systems are getting very large
 - Top systems have tens of thousands of nodes and order 1 million cores:
 - Tianhe-2 (China) 16,000 nodes
 - Blue Waters (Illinois) 25,000 nodes
 - Sequoia (LLNL) 98,304 nodes, >1M cores
 - ◆ Just getting all of these nodes to agree takes time
 - $O(10\mu\text{secs})$ or about 20,000 cycles of time)



Barriers and Synchronizing Communications

- Barrier:
 - ◆ Every thread (process) must enter before any can exit
- Many implementations, both in hardware and software
 - ◆ Where communication is pairwise, Barrier can be implemented in $O(\log p)$ time. Note $\text{Log}_2(10^6) \approx 20$
 - But each step is communication, which takes 1us or more
- Barriers rarely required in applications (see "functionally irrelevant barriers")



Barriers and Synchronizing Communications

- A communication operation that has the property that all must enter before any exits is called a “synchronizing” communication
 - ◆ Barrier is the simplest synchronizing communication
 - ◆ Summing up a value contributed from all processes and providing the result to all is another example
 - Occurs in vector or dot products important in many HPC computations





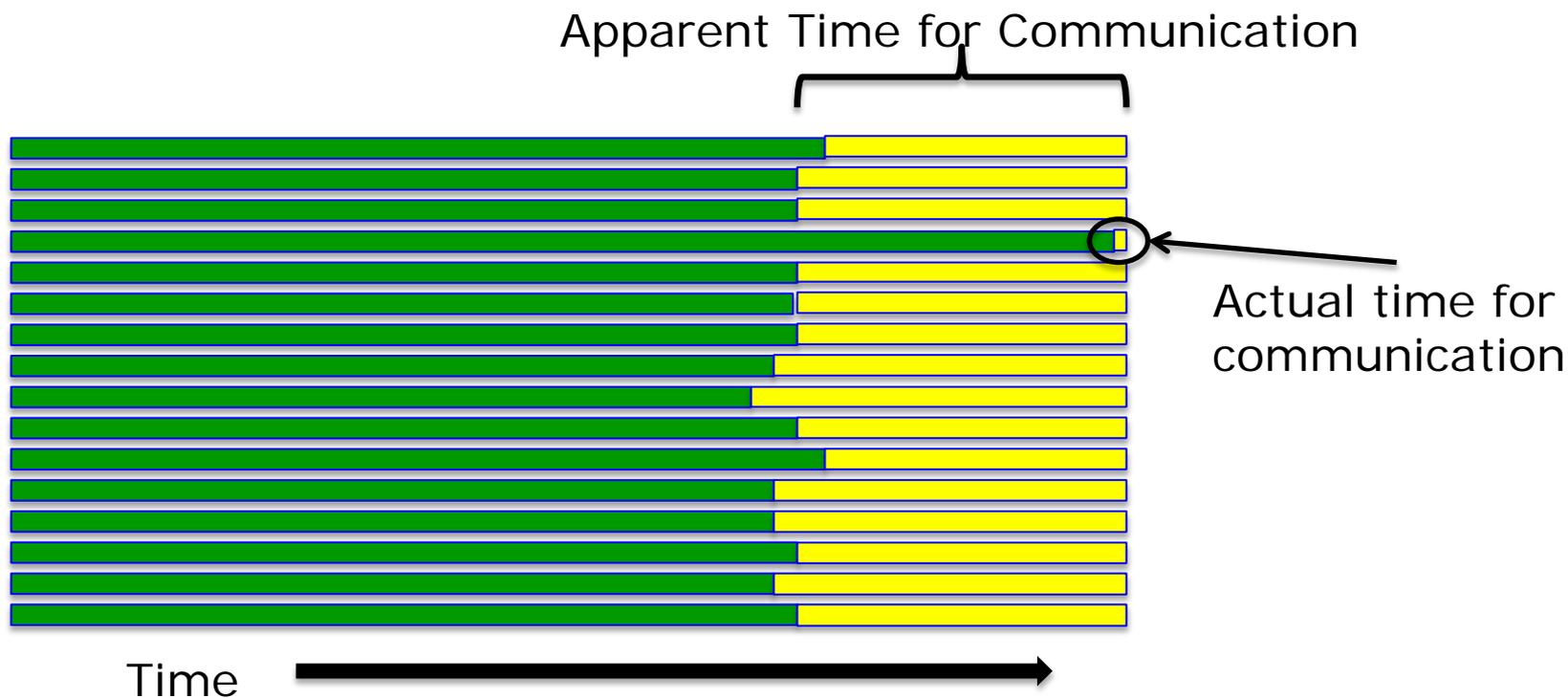
Synchronizing Communication

- Other communication patterns are more weakly synchronizing
 - ◆ Recall the halo exchange example
 - ◆ While not synchronizing across all processes, still creates dependencies
 - Processes can't proceed until their neighbors communicate
 - Some *programming implementations* will synchronize more strongly than required by the data dependencies in the algorithm



So What Does Go Wrong?

- What if one core (out of a million) is delayed?



- Everyone has to wait at the next synchronizing communication

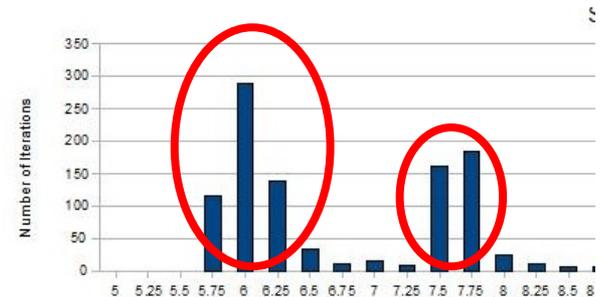
And It Can Get Worse

- What if while waiting, another core is delayed?
 - ♦ “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation,” Torsten Hoefler, Timo Schneider, Andrew Lumsdaine
 - Best Paper, SC10
 - ♦ Becomes more likely as scale increases – the probability that no core is delayed is $(1-f)^p$, where f is the probability that a core is delayed, and p is the number of cores
 - $\approx 1 - pf + \dots$
- The delays can cascade



Many Sources of Delays

- Dynamic frequency scaling (power/temperature)
- Adaptive routing (network contention/resilience)
- Deep memory hierarchies (performance, power, cost)
- Dynamic assignment of work to different cores, processing elements, chips (CPU, GPU, ...)
- Runtime services (respond to events both external (network) and internal (gradual underflow))
- OS services (including I/O, heartbeat, support of runtime)
- etc.



Summary so Far

- BSP (in its general form) provides an effective way to reason about parallel programs in HPC
 - ◆ Addresses both performance and correctness
 - Formal models of performance in wide use, from Hockney's original $T_c = a + rn$ to LogP and beyond
 - Increasing number of tools for evaluating correctness of communication patterns
- But increasingly poor fit to real systems, especially (but not only) at extreme scale



How should we think about parallel programming?

- Need a more formal way to think about programming
 - ◆ Must be based on the realities of real systems
 - ◆ Not the system that we wish we could build (see PRAM)
- Not talking about a *programming model*
 - ◆ Rather, first need to think about what an extreme scale parallel system can *do*
 - ◆ System – the hardware and the software together





Separate the Programming Model from the Execution Model

- What is an execution model?
 - ◆ It's how you think about how you can use a parallel computer to solve a problem
- Why talk about this?
 - ◆ The execution model can influence what solutions you consider (see the *Whorfian hypothesis* in linguistics)
 - ◆ After decades where many computer scientists only worked with one execution model, we are now seeing new models and their impact on programming and algorithms





Examples of Execution Models

- Von Neumann machine:
 - ◆ Program counter
 - ◆ Arithmetic Logic Unit
 - ◆ Addressable Memory
- Classic Vector machine:
 - ◆ Add “vectors” – apply the same operation to a group of data with a single instruction
 - Arbitrary length (CDC Star 100), 64 words (Cray), 2 words (SSE)
- GPUs with collections of threads (Warps)



Programming Models and Systems

- In past, often a tight connection between the execution model and the programming approach
 - ◆ Fortran: FORMula TRANslation to von Neumann machine
 - ◆ C: e.g., “register”, ++ operator match PDP-11 capabilities, needs
- Over time, execution models and reality changed but programming models rarely reflected those changes
 - ◆ Rely on compiler to “hide” those changes from the user – e.g., auto-vectorization for SSE(n)
- Consequence: Mismatch between users’ expectation and system abilities.
 - ◆ Can’t fully exploit system because user’s mental model of execution does not match real hardware
 - ◆ Decades of compiler research have shown this problem is extremely hard – can’t expect system to do everything for you.



Programming Models and Systems

- Programming Model: an abstraction of a way to write a program
 - ◆ Many levels
 - Procedural or imperative?
 - Single address space with threads?
 - Vectors as basic units of programming?
 - ◆ Programming model often expressed with pseudo code
- Programming System: (My terminology)
 - ◆ An API that implements parts or all of one or more programming models, enabling the precise specification of a program



Why the Distinction?

- In parallel computing,
 - ◆ Message passing is a programming model
 - Abstraction: A program consists of processes that communicate by sending messages. See “Communicating Sequential Processes”, CACM 21#8, 1978, by C.A.R. Hoare.
 - ◆ The Message Passing Interface (MPI) is a programming system
 - Implements message passing and other parallel programming models, including:
 - Bulk Synchronous Programming
 - One-sided communication
 - Shared-memory (between processes)
 - ◆ CUDA/OpenACC/OpenCL are systems implementing a “GPU Programming Model”
 - Execution model involves teams, threads, synchronization primitives, different types of memory and operations



The Devil Is in the Details

- There is no unique execution model
 - ◆ What level of detail do you need to design and implement your program?
 - Don't forget – you decided to use parallelism because you could not get the performance you need without it
- Getting what you need already?
 - ◆ Great! It ain't broke
- But if you need more performance of any type (scalability, total time to solution, user productivity)
 - ◆ Rethink your model of computation and the programming models and systems that you use



Rethinking Parallel Computing

- Changing the execution model
 - ◆ No assumption of performance regularity – but not unpredictable, just imprecise
 - Predictable within limits and most of the time
 - ◆ Any synchronization cost amplifies irregularity – don't include synchronizing communication as a desirable operation
 - ◆ Memory operations are always costly, so moving operation to data may be more efficient
 - Some hardware designs provide direct support for this, not just software emulation
 - ◆ Important to represent key hardware operations, which go beyond simple single ALU
 - Remote update (RDMA)
 - Remote operation (compare and swap)
 - Execute short code sequence (Active Messages, parcels)



How does this change the way you should look at parallel programming?

- More dynamic. *Plan* for performance irregularity
 - ◆ But still exploit as much regularity as possible to minimize the overhead)
- Recognize communication takes time, which is not precisely predictable
 - ◆ Communication between cache and memory or between two nodes in a parallel system
- Think about the *execution model*
 - ◆ *Your* abstraction of how a parallel machine works
 - ◆ Include the hardware-supported features that you need for performance
- Finally, use a programming system that lets you express the elements you need from the execution model.



Example: The Mesh Computation

- Rethinking: Performance not perfectly predictable, so must not assume that a perfect data distribution gives a perfect work distribution
 - ◆ One solution: “over decompose” mesh into more pieces that there are processes or threads; use a combination of a priori and dynamic scheduling to adapt
- Rethinking: Communication dependencies introduce delays
 - ◆ Many solutions: Use one-sided communication; use non-blocking communication; use multi-step algorithms; use over decomposition to give greater flexibility to comm schedule, ...



Take Away

- Be aware of the capabilities of a parallel system
 - ◆ Not just what a particular programming *model* provides
- Think about the realities of execution on your parallel computer
 - ◆ Use as simple an abstraction as possible *but no simpler*
- Find a programming *system* with which you can efficiently express your algorithm
 - ◆ Don't be confused by statements that a particular programming system only implements a single programming model or only works with a single execution model



Further Investigation

- Programming systems and tools that support a more dynamic form of computing:
 - ◆ Charm++ and Adaptive MPI
 - ◆ DAQUE, used in the MAGMA and PLASMA numerical libraries
 - ◆ Many thread-based tools, such as TBB; “guided” scheduling in OpenMP
 - ◆ Don’t forget to explore the full capabilities of MPI-3



Further Investigation

- Research systems
 - ◆ EARTH and EARTH Threaded-C
<http://www.capsl.udel.edu/earth.shtml>
 - ◆ XPRESS, HPX and ParalleX
<https://www.xstackwiki.com/index.php/XPRESS> (and see other X-Stack projects)



Further Investigation

- Algorithms
 - ◆ Nonblocking reductions in Conjugate Gradient
 - ◆ Multistep methods (reduce, not eliminate synchronizing collectives)
 - ◆ Data-centric graph algorithms (move computation to data, rather than remote access of data)





Further Investigation

- Many more; the preceding is just a small sampling
- Search for “execution model parallel computing”
- Meet with others using parallel programming
 - ◆ We recommend SC13, November 17-22, in Denver!



ACM: The Learning Continues...

- Questions about this webcast? learning@acm.org
- ACM Learning Webinars:
<http://learning.acm.org/multimedia.cfm>
- ACM Learning Center: <http://learning.acm.org>
- ACM SIGHPC: <http://www.sighpc.org/>



Association for
Computing Machinery

Advancing Computing as a Science & Profession