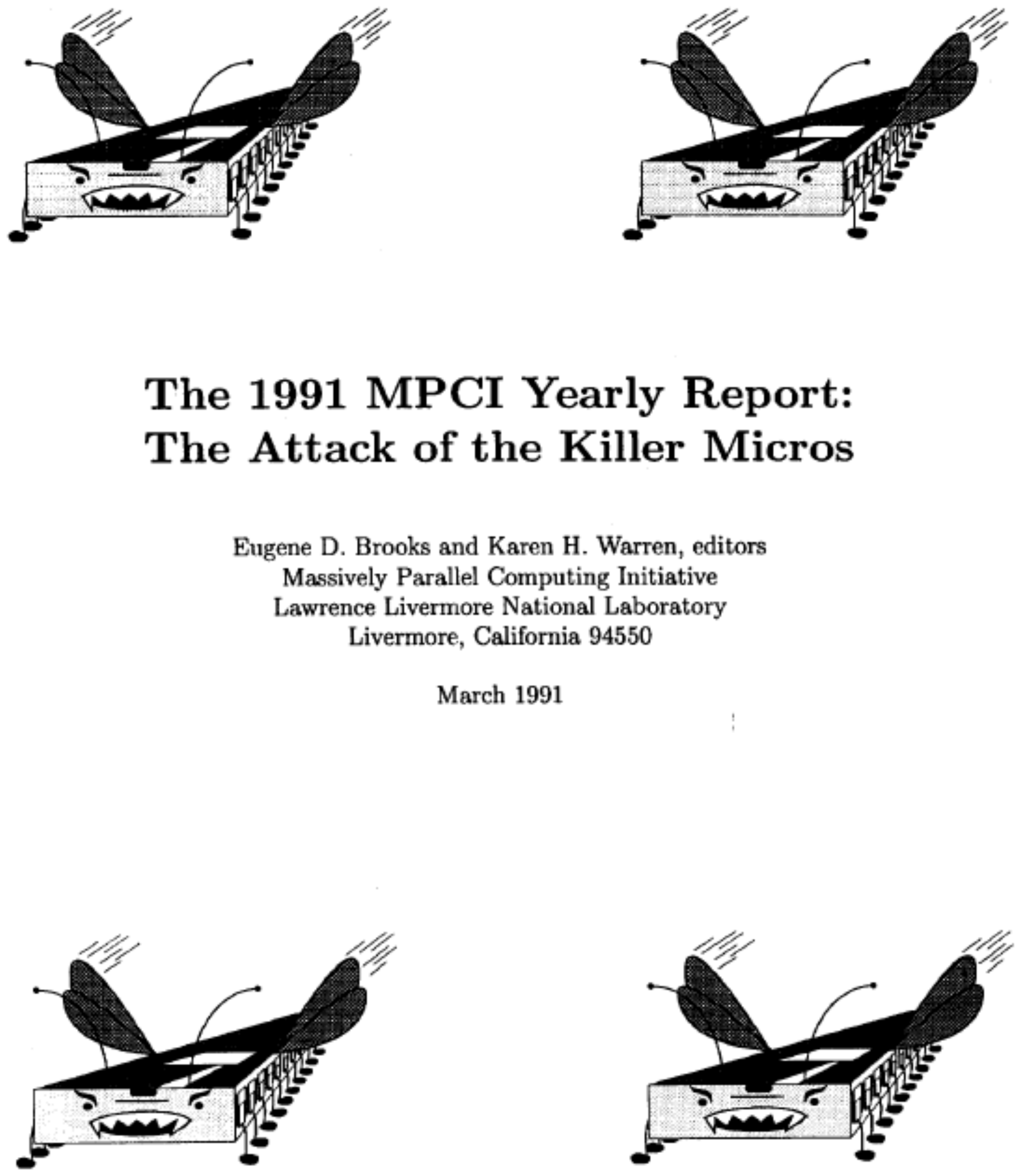
Programming Systems for High-Performance Computing

Marc Snir

Preamble

In The Beginning

- Distributed Memory Parallel Systems are replacing vector machines.
- CM-5 (using SPARC) announced 1991
  - 1024 node CM-5 is at top of first TOP500 list in 1993
- *Problem: Need to rewrite all (vector) code in new programming model*

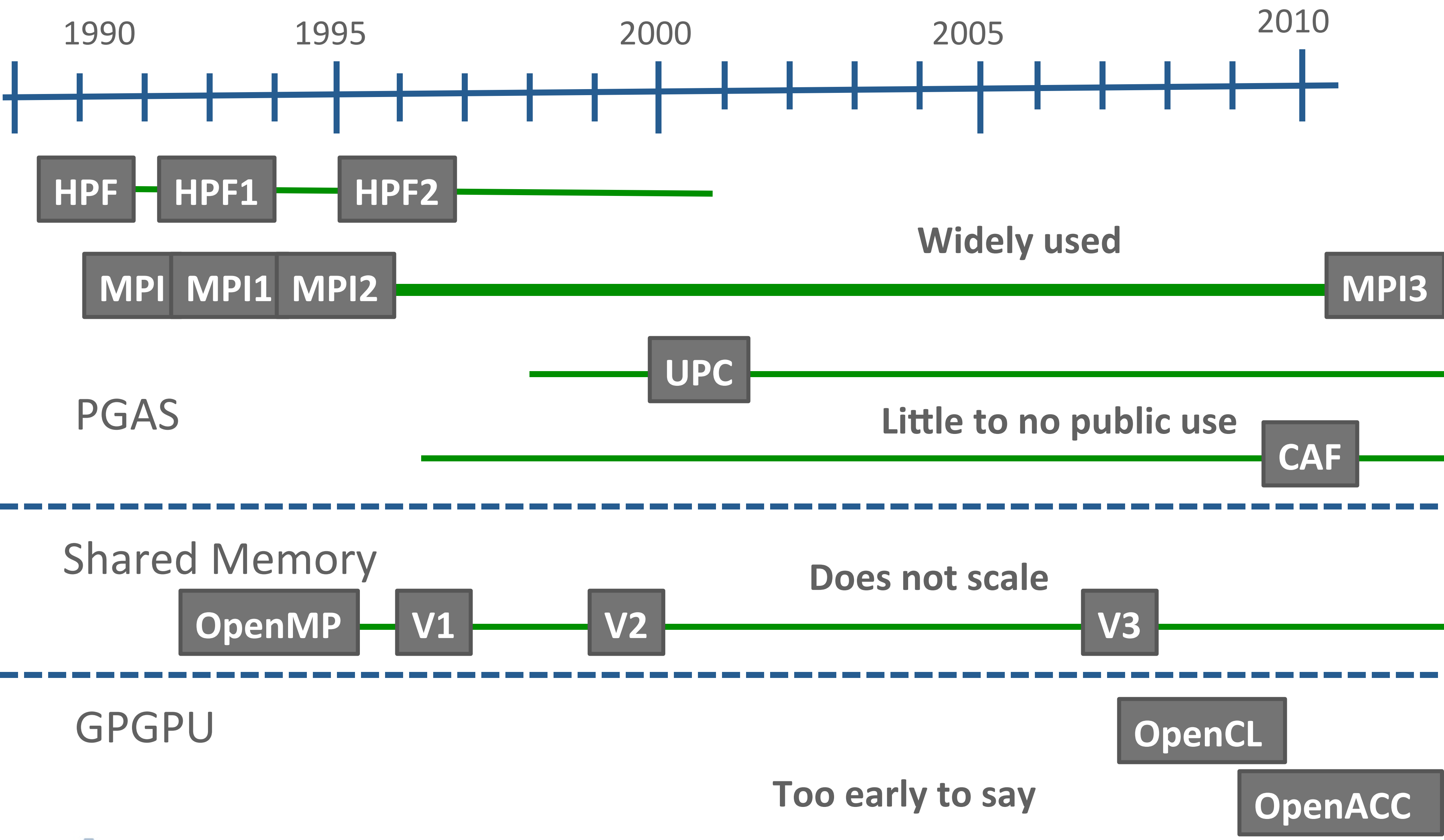


The 1991 MPCI Yearly Report:
The Attack of the Killer Micros

Eugene D. Brooks and Karen H. Warren, editors
Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, California 94550

March 1991

Some of the Attempts



Questions

1. *Message-passing is claimed to be "a difficult way to program". If so, why is it the only widely used way to program HPC systems?*

2. *What's needed for a programming system to succeed?*

3. *What's needed in a good HPC parallel programming system?*

Definitions

- **Parallel programming model**: a parallel programming pattern
  - Bulk synchronous, SIMD, message-passing…
- **Parallel programming system**: a language or library that implements one or multiple models
  - MPI+C, OpenMP…
- MPI+C implements multiple models (message-passing, bulk synchronous…)
- MPI+OpenMP is a **hybrid system**

Why Did MPI Succeed?

Beyond the expectations of its designers

Some Pragmatic Reasons for the Success of MPI

- **Clear need**: Each of the vendors had its own message-passing library; there were no major differences between them, and no significant competitive advantage. Users wanted portability
  - Same was true of OpenMP
- **Quick implementation**: MPI libraries were available as soon as the standard was ready
  - As distinct from HPF [*rise & fall of HPF*]
- **Cheap**: A few people-years (at most) to port MPI to a new platform

Some Technical Reasons for the Success of MPI

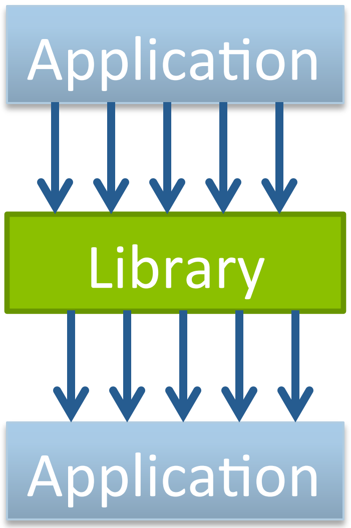
- **Good performance:** HPC programming is about performance. An HPC programming system must provide to the user the raw HW performance of the HW. MPI mostly did that
- **Performance portability**: Same optimizations (reduce communication volume, aggregate communication, use collectives, overlap computation and communication) are good for any platform
  - This was a major weakness of HPF [*rise & fall of HPF*]
- **Performance transparency**: Few surprises; a simple performance model (e.g., postal model – *a+bm*) predicts well communication performance most of the time
  - Another weakness of HPF

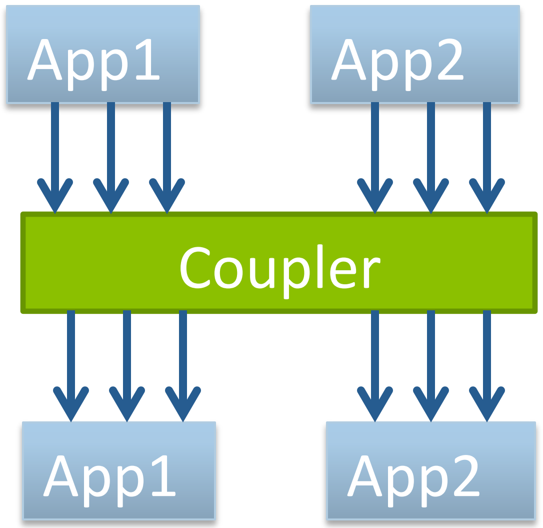
How About Productivity?

- Coding productivity is much overrated: A relative small fraction of programming effort is coding: Debugging, tuning, testing & validating are where most of the time is spent. MPI is not worse than existing alternatives in these aspects of code development.
- MPI code is very small fraction of a large scientific framework
- Programmer productivity depends on entire tool chain (compiler, debugger, performance tools, etc.). It is easier to build a tool chain for a communication library than for a new language

Some Things MPI Got Right

- Communicators
  - Support for time and space multiplexing
- Collective operations
- Orthogonal, object-oriented design





MPI Forever?

- MPI will continue to be used in the exascale time frame
- MPI may increasingly become a performance bottleneck at exascale
- It’s not MPI; it is MPI+X, where X is used for node programming. The main problem is a lack of adequate X.

It is Hard to Kill MPI

- By now, there are 10s of millions LOCs that use MPI
- MPI continues to improve (MPI3)
- MPI performance can still be improved in various ways (e.g., JIT specialization of MPI calls)
- MPI scalability can still be improved in a variety of ways (e.g., distributed data structures)
- The number of nodes in future exascale systems will not be significantly higher than the current numbers

What’s Needed for a New System to Succeed?

Factors for Success

- Compelling need
  - MPI runs out of steam (?)
  - OpenMP runs out of steam (!)
  - MPI+OpenMP+Cuda+… becomes too baroque
  - Fault tolerance and power management require new programming models
- Solution to backward compatibility problem
- Limited implementation cost & fast deployment
- Community support (?)

Potential Problems with MPI

- Software overhead for communication
  - Strong scaling requires finer granularity
- MPI processes with hundreds of threads – serial bottlenecks
- Interface to node programming models: The basic design of MPI is for a single threaded process; tasks are not MPI entities
- Interface to heterogeneous nodes (NUMA, deep memory hierarchy, possibly non-coherent memory)
- Fault tolerance
  - No good exception model
- Good rDMA leverage (low-overhead one-sided communication)

Potential Problems with OpenMP

- No support for locality – flat memory model
- Parallelism is most naturally expressed using fine grain parallelism (parallel loops) and serializing constructs (locks, serial sections)
- Support for heterogeneity is lacking:
  - OpenACC has narrow view of heterogeneous architecture (two, noncoherent memory spaces)
  - No support for NUMA (memory heterogeneity)

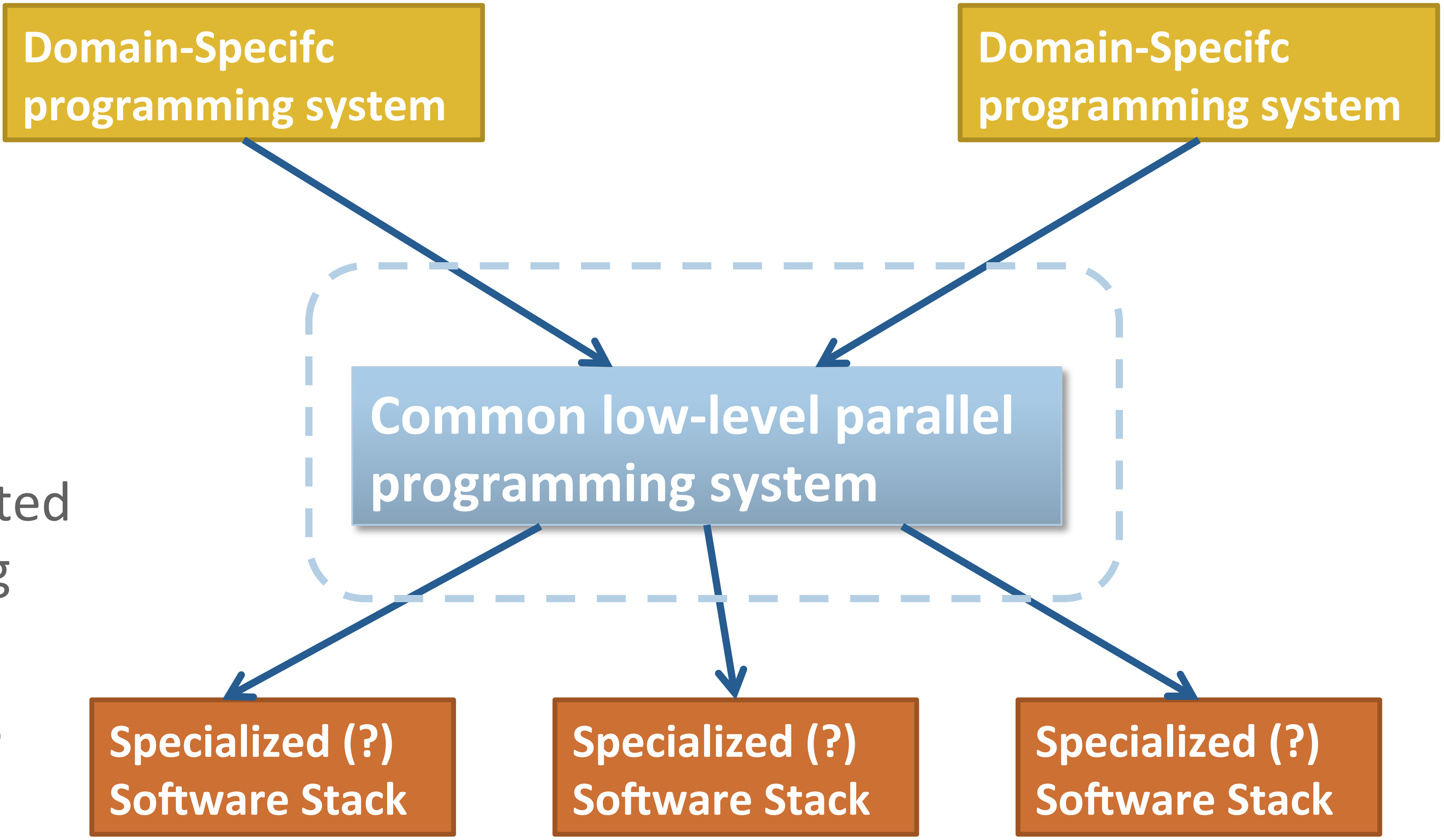
(There is significant uncertainty on the future node architecture)

New Needs

- Current mental picture of parallel computation: Equal computation effort means equal computation time
  - Significant effort invested to make this true: jitter avoidance
- Mental picture is likely to be inaccurate in the future
  - Power management is local
  - Fault correction is local
- Global synchronization is evil!

What Do We Need in a Good HPC Programming System?

Our Focus


Domain-Specifc programming system
Domain-Specifc programming system
Common low-level parallel programming system
Specialized (?) Software Stack
Specialized (?) Software Stack
Specialized (?) Software Stack

Automated mapping

Multiple Systems

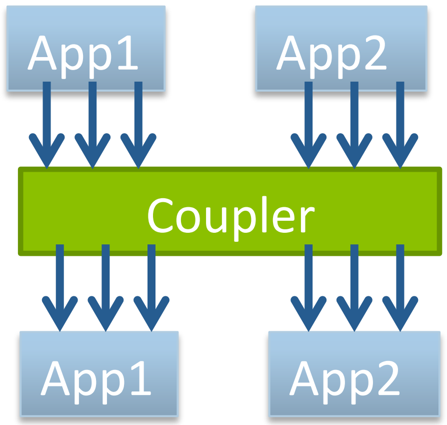
Shared Memory Programming Model

Basics:

- Program expresses parallelism and dependencies (ordering constraint)
- Run-time maps execution threads so as to achieve load balance
- Global name space
- Communication is implied by referencing
- Data is migrated close to executing thread by caching hardware
- User reduces communication by ordering accesses so as to achieve good temporal, spatial and thread locality

Incidentals:

- Shared memory programs suffer from races
- Shared memory programs are nondeterministic
- Lack of collective operations
- Lack of parallel teams



Distributed Memory Programming Model (for HPC)

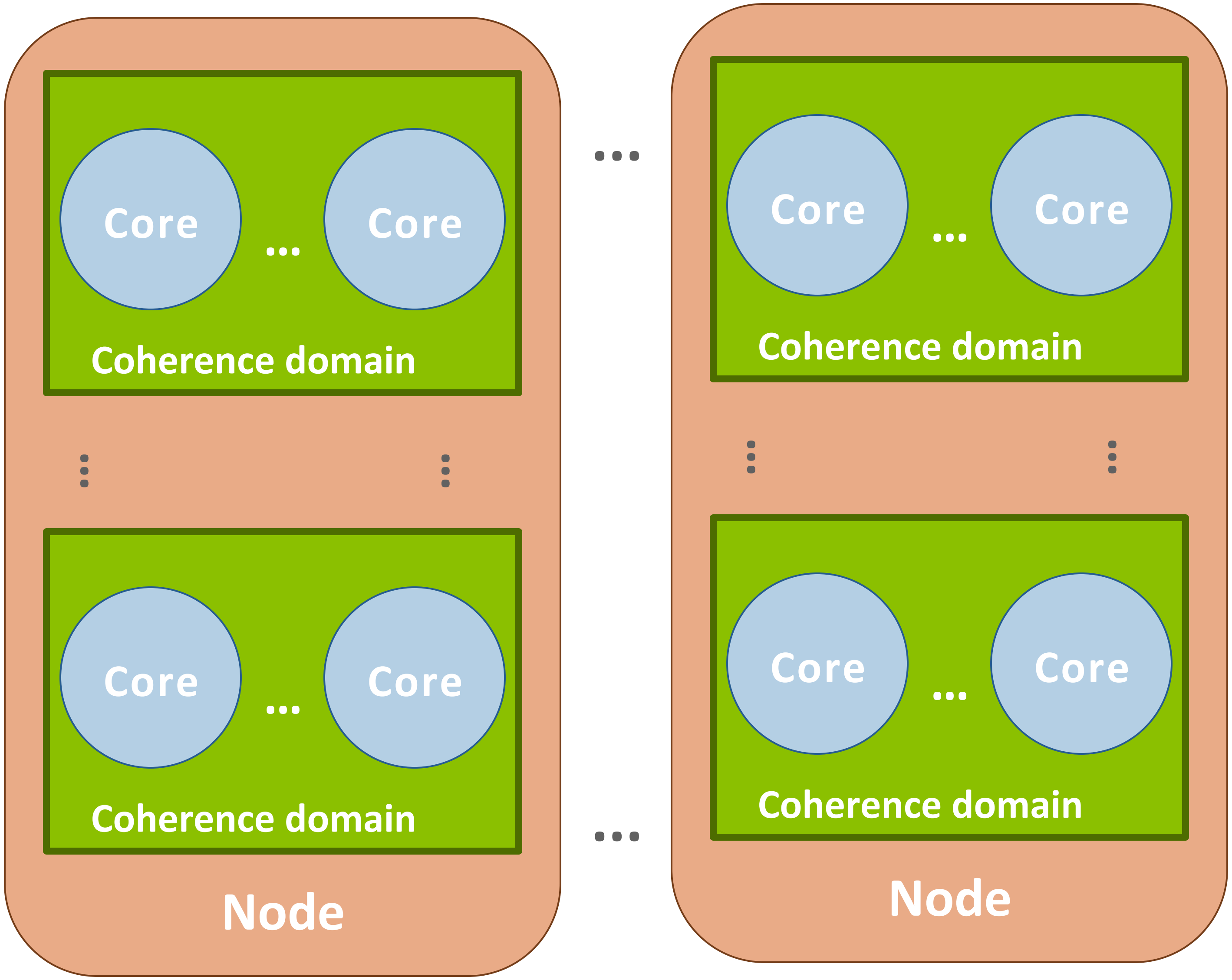
Basics:

- Program distributes data, maps execution to data, and expresses dependencies
- Communication is usually explicit, but can be implicit (PGAS)
  - Explicit communication needed for performance
  - User explicitly optimizes communication (no caching)

Incidentals:

- Lack of global name space
- Use of send-receive

Hypothetical Future Architecture: (At Least) 3 Levels



- Assume core heterogeneity is transparent to user (alternative is too depressing…)

Possible Direction

- Need, for future nodes, a model intermediate between shared memory and distributed memory
  - Shared memory: global name space, implicit communication and caching
  - Distributed memory: control of data location and communication aggregation
  - With very good support for latency hiding and fine grain parallelism
- Such a model could be used uniformly across the entire memory hierarchy!
  - With a different mix of hardware and software support

Extending the Shared Memory Model Beyond Coherence Domains - Caching in Memory

- Assume (to start with) a fixed partition of data to "homes" (locales)
- Global name space: easy
- Implicit communication (read/write): performance problem
  - Need accessing data in large *user-defined* chunks ->
    - User-defined "logical cache lines" (data structure, tile; data set that can be moved together
    - User (or compiler) inserted "prefetch/flush" instructions (acquire/release access)
  - Need latency hiding ->
    - Prefetch or "concurrent multitasking"

Living Without Coherence

- Scientific codes are often organized in successive phases where a variable (a family of variables) is either exclusively updated or shared during the entire phase. Examples include
  - Particle codes (Barnes-Hut, molecular dynamics…)
  - Iterative solvers (finite element, red-black schemes..)

☛ Need not track status of each "cache line"; software can effect global changes at phase end

How This Compares to PGAS Languages?

- Added ability to cache and prefetch
  - As distinct from copying
  - *Data changes location without changing name'*
- Added message driven task scheduling
  - Necessary for hiding latency of reads in irregular, dynamic codes

Is Such Model Useful?

- Facilitates writing dynamic, irregular code (e.g., Barnes-Hut)
- Can be used, without performance loss, by adding annotations that enable early binding

Example: caching

- General case: “Logical cache line” defined as a user-provided class
- Special case: Code for simple, frequently used tile types is optimized & inlined (e.g., rectangular submatrices) [Chapel]

- General case: need dynamic space management and (possibly) an additional indirection (often avoided via pointer swizzling)
- Special case: same remote data (e.g., halo) is repeatedly accessed. Can preallocate local buffer and compile remote references into local references (at compile time, for regular halos, after data is read, for irregular halos)

There is a L o n g List of Additional Issues

- How general are home partitions?
  - Use of “owner compute” approach requires general partition
  - Complex partitions lead to expensive address translation
- Is the number of homes (locales) fixed?
- Can data homes migrate?
  - No migration is a problem for load balancing and failure recovery
  - Usual approach to migration (virtualization & overdecomposition) unnecessarily reduces granularity
- Can we update (parameterized) decomposition, rather than virtualize locations?

There is a L o n g List of Additional Issues (2)

- Local view of control or global view?

Forall i on A[i] A[i] = foo(i)

  - Transformation described in terms of global data structure

Foreach A.tile foo(tile)

  - Transformation described in terms of local data structure (with local references)
  - There are strong, diverging opinions on which is better

- Hierarchical data structures & hierarchical control
- Exception model
- …

The End