Programming Models for HPC

Marc Snir
Goals of Lecture

• Establish a taxonomy of HPC programming models and systems

• Introduce the main programming models and systems in current use

• Understand there is no magic bullet, but trade-offs
INTRODUCTION
Some Taxonomy

• **Programming model**: Set of operations and constructs used to express parallelism
  – Message-passing, Shared memory, Bulk-synchronous...

• **Programming system**: Library or language that embodies one or more programming models
  – MPI, OpenMP, Cuda...
  – C++ and Java are systems that implement the same programming model
  – MPI supports multiple programming models: message-passing, one-sided communication, bulk synchronous, asynchronous...

• Model: defines semantics of execution and *defines performance of execution*
  – Usually, semantics are carefully defined, but not so performance model
Implementation Stack

- Language & Libraries
- Compiler, autotuner
- Executable
- Run-time
- Hardware

High-Level Execution model (defined by run-time)

Low-Level Execution model (defined by HW)

Possibly, more layers
Generic Hardware -- Node

- Vector unit
  - Physical Thread
  - Cache
  - Cache
  - Cache

- Shared Memory
Communication & Synchronization

• Load/store (from memory to cache or cache to cache)

• Hardware and software cache prefetch

• Read-modify-write operations
  – Mutual exclusion (locks, atomic updates) – non-ordering, symmetric synchronization
  – Monitor/wait – ordering, asymmetric synchronization

• CPUs provide extensive support to the former; HPC software has strong need for the later.
Node Architecture -- Evolution

• Increasing number of physical threads
• Increasing number of cache levels
  – Lower level caches may be replaced with scratchpads (local memory)
• Shared memory may not be coherent and will be NUMA
  – May be partitioned
• Physical threads may not be all identical
  – E.g., CPU-like vs. GPU-like physical threads
  – May have same ISA and different performance, or different ISAs
Communication & Synchronization Primitives:

- **Point-to-point**: send/receive…
- **One-sided** (rDMA): put, get, accumulate…
- **Collective**: barrier, reduce, broadcast…

- Flat network vs. partially or completely exposed topology
PROGRAMMING MODELS
Low-Level Programming Model

• Fixed number of threads (1/2/4 per core)
  – Threads communicate via shared memory
  – Fixed amount of memory per node
• Fixed number of nodes
  – Nodes communicate via message-passing (point-to-point, one-sided, collective)
• Performance model:
  – All threads compute at same speed (equal work = equal time)
  – Memory is flat
  – Communication time predicted by simple model (e.g. \( a+bm \), where \( m \) is message size)
• Model can be implemented via MPI + pthreads, or MPI + restricted use of OpenMP
Low-Level Programming Model Pros & Cons

✔ Very hardware specific – can achieve best performance on current systems
✗ Very low-level: Hard to get correct programs
✗ Very hardware-specific: number of threads, amount of memory and number of nodes are code parameters
✗ Does not accommodate heterogeneous cores
✗ Performance model may not work in the future, because of power management and error handling: Equal work ≠ equal compute time
Higher-level Programming Models

1. Virtualize resource and have the run-time handle the mapping of virtual resources to physical resources
   - “Virtual model execution” has well-defined semantics and performance model

2. Encourages/mandates use of programming patterns that reduce the likelihood of errors and facilitate virtualization
   - Ensures that high-level performance model gives correct predictions
     ✓ Simplifies programming
     ✓ Provides more portability: program need not exactly match the physical resources – e.g., number of threads or nodes
     ✓ Can hide imperfections in the low-level model (e.g., assumption of flat memory or fixed compute speed)
     ❌ Can be inefficient

• Need to understand when high-level performance model is valid
Example: Dijskra, Go-to Statement Considered Harmful -- Restrict to Facilitate Programming

• If program uses go-to’s arbitrarily then it is hard to understand relation between static program and dynamic execution state

• If program uses only well-nested loops and function calls, the relation is simple: Execution state is defined by
  – Program counter (what statement is currently executed)
  – Call stack (nesting of function calls)
  – Values of loop indices (which iteration is currently executed)

• Much easier to get understand semantics and performance

• Not restrictive: General programs can be transformed into well-structured programs without significant performance loss.
SHARED-MEMORY PROGRAMMING
Locality in Shared Memory: Virtualization and Performance Model

- **Caching = memory virtualization**: Name (address) of variable does not determine its location; caching hardware does so
- **Performance model**: Memory capacity is DRAM size; memory latency is L1 speed
  - Reasonable approximation if cache hit rate is high
  - Good cache hit rate is essential to performance, because of high memory latency
  - >99% of power consumed in future chips is spent moving data; reduced bandwidth consumption is essential
- For high cache hit rate, need good locality:
  - Temporal locality, spatial locality, thread locality
- **Performance model is valid only for programs with good locality**
Dynamic Task Scheduling: Virtualization and Performance Model

- It is convenient to express parallel computations as a set of tasks and dependencies between tasks (task graph).
- E.g., reduction:

  ![Task Graph Image]

- Example has static task graph; in general it can be dynamic (new tasks spawned during execution).
Mapping Tasks to Physical Threads

• **Work**: Total number of operations (sequential compute time)
• **Depth**: Critical path length (compute time with unbounded number of threads)
• **Desired performance model**: Time = max (Work/P, Depth)
  – Can be achieved, with suitable restrictions on task graph

• **Restrictions on task granularity** (size) – need to be large enough so as to amortize cost of scheduling task
  – Cost: overhead of scheduling + overhead of moving data to place where task is scheduled (1000’s instructions in shared memory)

• **Restrictions on task graph**:
  – Static graph
  – “Well-structured” dynamic task graph
Well-Structured Program

- Task can spawn children tasks and wait for their completion
  - Spawn
  - Parallel loop
- No other dependencies exist, except parent to child
  - Series-parallel (and/or) execution graph
- Parallel equivalent of go-to less program
- Can be implemented in suitably restricted OpenMP (no synchronization between concurrent tasks, etc.)
  - Scheduling can be done using work-stealing
Work Stealing

• A thread appends a newly created task to local queue
• An idle thread picks a task from its local queue
• if local queue is empty, it steals a task from another queue

✔ Theoretically optimal and works well in practice
✔ Handles well variance in thread speed
✗ Needs many more tasks than physical threads (*over-decomposition*)
Can a Shared-Memory Programming Model Work on Distributed Memory?

- Naïve mapping: Distributed virtual shared memory (DVSM): Software emulation of HW coherence
  - Cache line = page
  - Cache miss = page miss
- Abysmal performance – lines too large and coherence overhead too large

- Next step: Give up caching & coherence; give up dynamic task scheduling – Partitioned Global Address Space (PGAS)
  - Languages: UPC, Co-Array Fortran
  - Libraries: Global Arrays
PGAS model

- Fixed number of (single-threaded) locales
- Private memory at each locale
- Shared memory and shared arrays are partitioned across locales
- Library implementation: remote shared variables accessed via put/get operation
- Language implementation: references to shared memory variables are distinct from references to local variables
PGAS Performance Model

• Not same as shared memory: Remote accesses are much more expensive than local accesses
• Temporal/spatial/thread locality do not cure the problem: No hardware caching & no efficient SW caching or aggregation of multiple remote accesses
• *Performance model is same as distributed memory*
  ⚡️ Programming style for performance must be distributed-memory-like: In order to obtain good performance need to explicitly get/put remote data in large chunks and copy to local memory

• Open problem: Can we define a “shared-memory like” programming model for distributed memory that provides shared memory performance model (with reasonable restrictions on programs)?
How About the Vector Units?

• Low-level: vector instructions
• High-level: compiler handles
• High-level performance model: floating-point performance is a (large) fraction of vector unit performance
  
  – Requires suitable programming style to write vectorizable loops
  
  – Direct addressing, aligned dense operands...
DISTRIBUTED-MEMORY PROGRAMMING
Data Partitioning

• **Shared memory**: program does not control explicitly data location; focus is on control partitioning, and data follows control.
  – Performs well if code satisfies restrictions listed for efficient dynamic tasks scheduling and locality

• **Distributed memory**: data location is (mostly) static and controlled by program; focus is on data partitioning and control follows data
  – Performs well if (mostly) static control partition works and communication across data partitions is limited
Data Partitioning

• Locale = physical location (usually, node)
  – Global address = <locale id, local address>

• **Global view of data**: Aggregates (e.g. arrays) are partitioned across locales
  – Partition is static (or slow changing)
  – Predefined partition types and (possibly) user-defined partitions
  – Data is accessed with “global name” (e.g., array indices); compiler translates to global address.

• **Local view of data**: Data is accessed using global address (if remote) or local address (if local)
  – Syntactic sugar: Co-arrays, with a locale index
    A(7,3)[8] – entry with indices (7,3) on locale 8.
Address Computation

- Array index: $i$
- $Block\_size = (N+1)/P$
- $Locale\_id = i/Block\_size$
- $Local\_address = i\%Block\_size + base$

- Computation is expensive (and even more so for fancier partitions)
Control Partitioning

• **Local view of control**: each local runs its own (possibly multithreaded) code; executions in distinct locales synchronize explicitly (e.g., with barriers). MPI, UPC, etc.
  – **Variant**: support remote task invocation (code on one local can spawn computation on another locale). Charm++, Chapel

• **Global view of control**: one global parallel loop; compiler & run-time maps each iterate to a locale chosen to minimize communication (e.g., owner compute rule)
  – User can associate control with data using “on” statements
Pros & Cons

• Global view of data & control
  ✔ Program need not change if number of locales or partition is changed
  ✗ Hard to identify and reason about communication in the program (implicitly determined by partition of data)
• Local view of data & control
  – Vice-versa
Can a Distributed-Memory Programming Model Work on Shared Memory?

• Sure: Partition memory into distinct locales, and associate each locale with a fixed set of physical threads (one or more)
  - ✗ Looses some flexibility in dynamic scheduling
  - ✗ Uses shared memory inefficiently to emulate send-receive or put-get
  - ✔ Achieves good locality

1. Reducing communication is essential in order to reduce power
2. We may not have coherence
   🛡 Will need shared-memory programming model with user control of communication
Non-Coherent, Shared-Memory-Like Programming Model

• Cache =
  1. Local memory
  2. Address translation (virtualization)
  3. Transfer is implicit
  4. coherence

(1) is essential
(2) could be done much more efficiently, with a good HW/SW mix (e.g., pointer swizzling)
(3) can be done, if maintain many contexts – can be augmented/ replaced with prefetch
(4) Is too expensive and probably superfluous in scientific computing
• Could be common shared memory / distributed memory model?
Is HPC Converging to One Architecture?

• Punctuated equilibrium: Relatively long periods with one dominant architecture, interspersed with periods of fast change
  – Bipolar vector -> killer micros -> multicore -> accelerators
  – We are in a period of fast change (hybrid memory, PIM, accelerators…)
  – *It is not clear there will be a convergence by the exascale era*
  – Need for faster architecture evolution as Moore’s Law slows down
  – Different markets pulling in different directions
  – Possible divergence between HPC and commodity

• Can we “hide” differences across different architectures?
Can we redefine compilation so that:

- It supports well a human in the loop (manual high-level decisions vs. automated low-level transformations)
- It integrates auto-tuning and profile-guided compilation
- It preserves high-level code semantics
- It preserves high-level code “performance semantics”
Grand Dream: Not One Programming Model

- More levels and more paths possible
- Automated translation available at each level (performance feedback driven)
- User coding supported at each level
- Semantics and performance preserved by translations
BASTA

QUESTIONS?