Logistics

- QwikLABS credits now distributed
- Read “Landscape” paper
Berkeley Parallel Computing Landscape

  - On CIS 631 website

- University of Berkeley EECS technical report “The Landscape of Parallel Computing Research: A View From Berkeley”
  https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

- Introduced the 13 Dwarves of parallel computing
Models and Abstractions

Programmer’s view

- **Programming Model**
- **Computational Model**
- **Execution Model**
- **Architecture Model**
- **Machine Model**

Abstractions

- Languages Semantics
- Software operation on platform
- Hardware components

Logical operation

System operation / interaction

Translation
Two General Issues for Programming Models

- Concurrency
  - How is concurrency created?
  - What are the concurrent execution entities?
  - What are the abstractions for concurrent interactions?

- Memory
  - How is the memory address space defined?
  - What is the address space for a concurrent entity?
  - How is memory shared / distributed?
  - How is memory managed and mapped?
Concurrency + Memory = Parallel Model

- Claims:
  - Concurrency relates to how parallelism is created and how entities synchronize parallel operations.
  - Memory relates to how “state” is shared between parallel entities (both logically and physically).

- From a low-level perspective ...
  - Concurrency: use OS to create processes and threads, plus libraries (e.g., POSIX, sockets, ...)
  - Memory: explicitly defined by how memory addressing is implemented for processes and threads.

- ... everything else is higher-level abstraction.
Concurrency is Affected by Memory

- How memory is implemented affects how concurrency is realized
  - Primarily with respect to interactions
- If 2 concurrent entities shared a memory address space (or some part), interactions could take place using that shared space
  - Necessarily requires synchronization to do so
- Question is to what extent shared memory abstractions are apparent in programming model
Memories and Structures (in programs)

- Software memory
  - Distinct logical storage area
  - Possibly multiple memories

- Structure
  - Collection of data created by program execution (arrays, trees, graphs, …)

- Partition
  - Division of structure into parts

- Mapping
  - Assignment of parts to memories
Affinity and Nonlocal Access

- Let the term *thread* mean a concurrent entity.
- Let the term *affinity* be the association of a thread to a memory.
  - If a thread has affinity with a memory, it can access its structures (memory address space)
    - this is a *local* memory
    - otherwise, it is a *remote* memory
- Non-local (remote) access
  - Thread 0 wants part B
  - Part B in Memory 1
  - Thread 0 does not have memory 1 affinity
Parallel Programming Methods

- Two general methods for programming parallel computers based on whether the parallel systems support shared memory
  - Shared memory multi-threading
  - Distributed memory message passing
- Of course, modern systems support shared memory
- So, what is the problem?
Why so many programming models?

- Bottom-up approach to programming models
- Given a system and its core capabilities...
  ...provide features that can access the available performance
  - Issues of portability, generality, programmability?
  - Not necessarily important

- Benefits:
  - Lots of control, decent generality, easy to implement

- Downsides
  - Lots of user-managed detail, brittle to changes
Better Programming Models

- Higher-level programming models can help insulate algorithms from parallel implementation details
  - Yet, without necessarily abdicating control
- Build in productive programming abstractions ...
  - Concurrency constructs
  - Memory sharing constructs
  - Synchronization mechanism
- … while providing power execution models
  - Task parallel
  - Data parallel
- Target evolving parallel computing systems
Parallel Programming

- Not just about finding parallelism
- Have to have correct execution
  - Synchronization to address dependencies
- Have to avoid synchronization errors
  - Consequence of concurrent programming
- Issues of performance and productivity
- Rich interaction of hardware-software tradeoffs
Tension Between Models and Implementation

- Proliferation of parallel programming models
  - New languages, language extensions, libraries
  - Strive to increase programmer productivity
  - Also want to achieve portable performance while handling complex architectures
Communication Sequential Processes (CSP)

- Formal language for describing patterns of interaction in concurrent systems
- Process algebras (process calculi) based on message passing via channels
- Influential in the design of Occam
- First described in 1978 by Tony Hoare
- Still actively pursued

https://en.wikipedia.org/wiki/Communicating_sequential_processes
Occam

- Builds on CSP process algebra
- Concurrent programming language
- Named after William of Ockham of Occam’s razor
- Imperative procedural language
- Developed by INMOS first in 1983
- Used to program the T9000 transputer

https://en.wikipedia.org/wiki/Occam_(programming_language)
https://en.wikipedia.org/wiki/Transputer
Erlang

- General purpose concurrent functional programming language
- Erlang runtime system
  - Distributed
  - Fault-tolerant and highly available
- Developed within Ericsson in 1986
- Erlang worldview
  - Everything is a process which are strongly isolated
  - Process creation and destruction is a lightweight operation
  - Message passing is the only way for processes to interact
  - Processes have unique names
  - If you know the name of a process you can send it a message
  - Processes share no resources

https://www.erlang.org
Parallel Virtual Machine (PVM)

- Software package that permits a heterogeneous collection computers connected by a network to be used as a single large parallel computer

- PVM consists of a run-time environment and library for message-passing, task and resource management, and fault notification

- Installed in a given "virtual machine” environment
  - User programs access PVM through library

- Develop by Oak Ridge National Laboratory in 1989

- In some respects, a predecessor to MPI

https://www.csm.ornl.gov/pvm/
PVM (Physical Vs Logical View)

Languages and extensions of C++ for parallel computation

- pC++ : uses class and templates to achieve parallelization
- CC++ : expands the syntax of C++ with parallel primitives

pC++ is a language extension to C++ (Gannon, 1990)

- Data-parallel style operations using "collections of objects"
- Member functions applied to the entire collection in parallel
- Compose distributed data structures with parallel semantics
- Aligned and distributed over memory of parallel machine

pC++ preprocessor (Sage++) translates pC++ into C++

- compiled on the target architecture with pC++ runtime system

pC++ ran on CM5, SGI Challenge, KSR1, Intel Paragon, Meiko CS2, IBM SP1, BBN TC2000, Sequent

http://www.extreme.indiana.edu/sage/
CC++

- Compositional C++ (Chandy and Kesselman, 1992)
  - CC++ is a strict superset of C++
  - CC++ compiler translates CC++ code into C++ code with calls to the CC++ runtime library (Nexus)

- Basic abstractions
  - Processor object: controlling locality (address space)
  - Global pointer: linking processor objects (communicate)
  - Thread: specifying concurrent execution (spawn)
  - Sync variable: synchronize thread executions
  - Atomic function: controlling the interleaving of threads
  - Transfer function: allow arbitrary data structures to be transferred between processor objects as arguments to remote processor objects
High Performance Fortran (HPF)

- HPF is an extension of Fortran 90 (Kennedy, 1992)
- Builds on F90 array syntax for a data parallelism
  - Spread array computation over multiple processors
  - Both SIMD and MIMD style
- New Fortran statements such as FORALL
- Compiler directives for distributions of array data
  - Compiler constructs and intrinsics allow computations and manipulations on data with different distributions
- Fortran 95 incorporated several HPF capabilities
- Most vendors and users have moved to OpenMP

https://en.wikipedia.org/wiki/High_Performance_Fortran
Programming Model and Shared Memory

- Programming model
  - Contract between programmer and hardware

- Shared memory abstraction has two concepts:
  - Coherence
    - Writes are propagated to other nodes
    - Writes to a particular memory location are seen in order
  - Consistency model
    - multiple writes or multiple are seen in a well-defined order
Shared Memory versus Distributed Memory

- Shared memory implies a physically shared address space
  - Physical addresses go to the same physical memory
  - Processes with shared segments or multiple threads

- Distributed memory is multiple non-shared address spaces
  - Processes (threads) only can access local physical memory with a physical address

- A *global address space (GAS)* is logically a globally accessible memory space that any processor with a GAS address can access
  - What is this GAS address?
  - Is a physical address or something else?
  - Shared memory (shared address space) abstraction in a physically distributed memory system
Distributed Shared Memory (DSM)

- General abstraction is *distributed shared memory*
  - GAS a type of DSM that gives global addressability

- Tradeoff in supporting the abstraction:
  - What is the programming interface?
  - What is the implementation mechanism

- Use operating system support and virtual memory

- Shared virtual memory (SVM)
  - IVY (Kai Li, Princeton)
  - Seminal system that sparked the entire field of DSM

https://en.wikipedia.org/wiki/Distributed_shared_memory
Shared Virtual Memory

- Pool of global “shared pages” (not local)
- Page table entry access bits determine if a page is local
- Virtual memory system is leveraged
- Read faults on invalid pages handled by OS
  - Trigger global page lookup
- OS maintains consistency at VM page level
- Memory managers run on different nodes to implement distributed global page consistency
- Programmers are effectively unaware
  - Can use same programming interface
  - Need to somehow identify globally shared pages
Distributed Shared Memory (DSM)

- There are other ways to implement DSM
  - Hardware support (different from CC-NUMA)
    - Wisconsin Wind Tunnel
    - DEC memory channel
  - Compiler support
- Also there are other variants of GAS
  - Partitioned GAS (PGAS)
- Distributed shared data versus just memory
Partitioned Global Address Space (PGAS)

- Managing locality is key in distributed-memory programming.
- PGAS combines a global address space with locality awareness:
  - Single global address space across all nodes
  - It is partitioned such that partitions reflect locality:
    - Each partition must fit within a shared-memory node
    - Each partition contains a collection of tasks and objects
- PGAS libraries and languages:
  - Global Arrays
  - Titanium, UPC, CAF, X10, Chapel, …
Global Arrays (GA)

- Project at Pacific Northwest National Lab
- Library that provides an abstraction to global data structures (arrays) distributed across nodes
- Distributed dense arrays can be accessed using a shared memory-like style
  - Single, shared data structure/ global indexing
- GA is built on a PGAS runtime system
  - Primitives for one-sided communication (get, put, accumulate)
  - Primitives for atomic operations
  - Non-blocking and blocking
  - Aggregated remote memory copy interface (ARMCI)

http://hpc.pnl.gov/globalarrays/
https://en.wikipedia.org/wiki/Global_Arrays
http://hpc.pnl.gov/armci/
Global Arrays (GA)

Physically distributed data

Global Address Space

Put

Get

Process 1

Process i

Process p
Global Arrays vs. Other Models

- Inter-operates with MPI
  - Convenient global-shared view for multi-dimensional arrays, but can use MPI model wherever needed
- Data-locality and granularity control is explicit with GA’s get-compute-put model, unlike the non-transparent communication overheads with other models (except MPI)
- Library-based approach does not rely upon smart compiler optimizations to achieve performance
- Data consistency must be explicitly managed
Creating Global Arrays

\[
g_a = \text{NGA\_Create}(\text{type}, \text{ndim}, \text{dims}, \text{name}, \text{chunk})
\]

- integer array handle
- character string
- minimum block size on each processor
- float, double, int, etc.
- array of dimensions
- dimension
Remote Data Access in GA vs MPI

Message Passing:

identify size and location of data blocks

loop over processors:
  if (me = P_N) then
    pack data in local message buffer
    send block of data to message buffer on P0
  else if (me = P0) then
    receive block of data from P_N in message buffer
    unpack data from message buffer to local buffer
  endif
end loop

copy local data on P0 to local buffer

Global Arrays:

\[ \text{NGA}\_\text{Get}(\text{g}_\text{a}, \text{lo}, \text{hi}, \text{buffer}, \text{ld}); \]

- Global Array handle
- Global upper and lower indices of data patch
- Local buffer and array of strides
**UPC**

- **UPC - Unified Parallel C**
  - An explicitly-parallel extension of ANSI C
  - A distributed shared memory parallel programming language

- Similar to the C language philosophy
  - Programmers are clever and careful
  - May need to get close to hardware to get performance

- Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C

[https://upc-lang.org](https://upc-lang.org)
[http://upc.lbl.gov](http://upc.lbl.gov)
**UPC**

- Programmer is presented with a single shared, partitioned address space
  - Variables may be directly read and written by any processor
  - Each variable is physically associated with a single processor
- UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time
  - Typically with a single thread of execution per processor.
- UPC extends ISO C 99 with the following constructs:
  - An explicitly parallel execution model
  - A shared address space
  - Synchronization primitives
  - A memory consistency model
  - Parallel utility libraries
General UPC View

- A collection of threads operating in a *partitioned global address space* that is logically distributed among threads
- Each thread has affinity with a portion of the globally shared address space
- Each thread has also a private space
- Elements in partitioned global space belonging to a thread are said to have affinity to that thread
## UPC Memory Model

- A pointer-to-shared can reference all locations in the shared space.
- A pointer-to-local ("plain old C pointer") may only reference addresses in its private space or addresses in its portion of the shared space.
- Static and dynamic memory allocations are supported for both shared and private memory.

### Global address space

<table>
<thead>
<tr>
<th>Global address space</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Private 0</td>
<td>Private 1</td>
<td>Private n-1</td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

CIS 631: Advanced Parallel Computing, University of Oregon

Lecture 4 – Parallel Programming
UPC Execution Model

- Threads working independently in SPMD fashion
  - Similar to MPI
  - MYTHREAD specifies thread index (0..THREADS-1)
  - Number of threads specified at compile-time or run-time

- Synchronization only when needed
  - Barriers
  - Locks
  - Memory consistency control
**UPC++**

- UPC++ exposes a PGAS memory model
  - Includes one-sided communication (RMA and RPC)
- Departs from the approaches taken by some predecessors
  - Such as UPC
  - Changes reflect a design philosophy that encourages the UPC++ programmer to directly express what can be implemented efficiently
    - Without a need for parallel compiler analysis
- Most operations are non-blocking, and the powerful synchronization mechanisms encourage applications to design for aggressive asynchrony
- All communication is explicit - there is no implicit data motion

https://bitbucket.org/berkeleylab/upcxx/wiki/Home
New Features in UPC++ v1.0

- Futures, promises and continuations
  - Futures are central to handling asynchronous operations: RMA and RPC.
- Progress guarantees
  - Library makes progress only when a core enters an active UPC++ call
- Remote atomics
  - Leverages available hardware support for significant performance benefits
  - Remote atomics use the C++11 memory model and an abstraction that enables efficient offload support
- Distributed objects
  - Scalable distributed object from any C++ object type, with one instance on each rank of a team
  - RPC can be used to access remote instances
- View-based Serialization
  - Mechanism for efficiently passing large and/or complicated data arguments to RPCs
- Non-contiguous RMA
  - Functions for non-contiguous data transfers directly on shared memory
  - Expands and generalizes the support for non-contiguous RMA
- Teams
  - Mechanism for grouping ranks, and are similar to MPI_Group
  - Play a role in collective communication
- Memory kinds
  - Global operations on memory with different kinds of access methods or performance properties
  - Uniform interface for transfers between such memories
**PGAS with Asynchrony (APGAS)**

Fine-grain concurrency
- `async S`
- `finish S`

Place-shifting operations
- `at(p) S`
- `at(p) e`

Atomicity
- `when(c) S`
- `atomic S`

Distributed heap
- `GlobalRef[T]`
- `PlaceLocalHandle[T]`
Co-Array Fortran (CAF)

- A CAF program is runs as a SPMD program
- Each replication of the program (image) has:
  - Own data objects
  - Shared data objects which are spread across images
- Array syntax of Fortran is extended with []
  - Concise representation of references to shared data objects
- CAF extension has been available in some Fortran compiler
  - Now included in Fortran 2008
  - GNU Fortran provides wide coverage
  - OpenCoarrays library (http://www.opencoarrays.org)
- CAF is often implemented on top of MPI
- CAF also uses other support for PGAS, such as GASNet

http://www.opencoarrays.org
Message Passing Parallel Programming

- Definition
  - Set of processes using only local memory
  - Processes communicate by sending and receiving messages
  - Data transfer requires cooperative operations to be performed by each process

- Communicating Sequential Processes (CSP)

- Message Passing Interface (MPI)
Onesided versus Message Passing

- **Message-passing**
  - Communication patterns are regular and predictable
  - Algorithms have a high degree of synchronization
  - Data consistency is straightforward

- **One-sided**
  - Communication is irregular
    - load balancing
  - Algorithms are asynchronous
    - but also can be used for synchronous algorithms
  - Data consistency must be explicitly managed
One-sided Communication

Message Passing:
Message passing requires cooperation on both sides. The processor sending the message (P1) and the processor receiving the message (P0) must both participate.

One-sided Communication:
Once message is initiated on sending processor (P1) the sending processor can continue computation. Receiving processor (P0) is not involved. Data is copied directly from switch into memory on P0.
Data Parallel Programming Paradigm

- The data parallel model is defined as:
  - Each process works on a different part of the same data structure
  - Commonly a Single Program Multiple Data (SPMD) approach
  - Data is distributed across processors
  - All message passing is done invisibly to the programmer
  - Commonly built "on top of" one of the common message passing libraries

- Programming with data parallel model is accomplished by writing a program with data parallel constructs and compiling it with a data parallel compiler.

- The compiler converts the program into standard code and calls to a message passing library to distribute the data to all the processes.
Data Parallel

- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- Certainly may have interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
  - Makes data parallel algorithms more efficient
- Single Program Multiple Data (SPMD)
  - Convenient way to implement data parallel computation
Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things

- Parallel computation models
  - Data parallel
  - Producer-Consumer
  - Task graph
  - Work pool
  - Master-Worker