Parallel Programming Models

Lawrence Rauchwerger
rwerger@illinois.edu

Course Objective

- Introduction to Parallelism
- Introduction to Programming Models
- Some Performance Considerations
- CnC, Cuda, STAPL, StreamIt, Halide, MLIR, etc.

Most of the material in this course has been adapted from various (cited) authoritative sources.

Lecture Plan

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- PGAS Languages
- CnC, Cuda, STAPL, StreamIt, Halide, MLIR

Most of the material in this course has been adapted from various (cited) authoritative sources.

Tentative Syllabus

- Introduction to Parallel Computer Systems
  - Parallel Architectures:

    Brief history and taxonomy
  - Today: Multicores, Clusters (of multicores, multi socket multicores), accelerators, e.g., GPGPU, TPU), i.e., systems on a chip.

Performance

- Performance issues in parallel programming
  - Locality and communication
  - Load balance
  - Parallelism or lack of it (synchronizations)
  - The issues are coupled ... How do we deal with them ? What is most important ?

Programming Models

- Taxonomy of programming models (languages, directives, libraries, etc.)
  - Examples from each area: MPI, OpenMP, UPC, TBB, STAPL
- Generic Libraries I
  - TBB, STAPL, etc
- Programming Models -- Languages
- Languages: UPC, X10, MapReduce, Julia etc.

Applications with Domain Specific Libraries and Languages

- Building applications using
  - Doman Specific Libraries
  - Domain Specific Languages (DSL)
  - e.g., STAPL, TBB, TENSORFLOW
- High level building blocks, fine grain program specification with performance control (Halide)
- MLIR – Use of compiler IR to control performance .. Complicated ….

Expectations

- Pick a topic – make a presentation

Mid Sept. : Choose a presentation topic

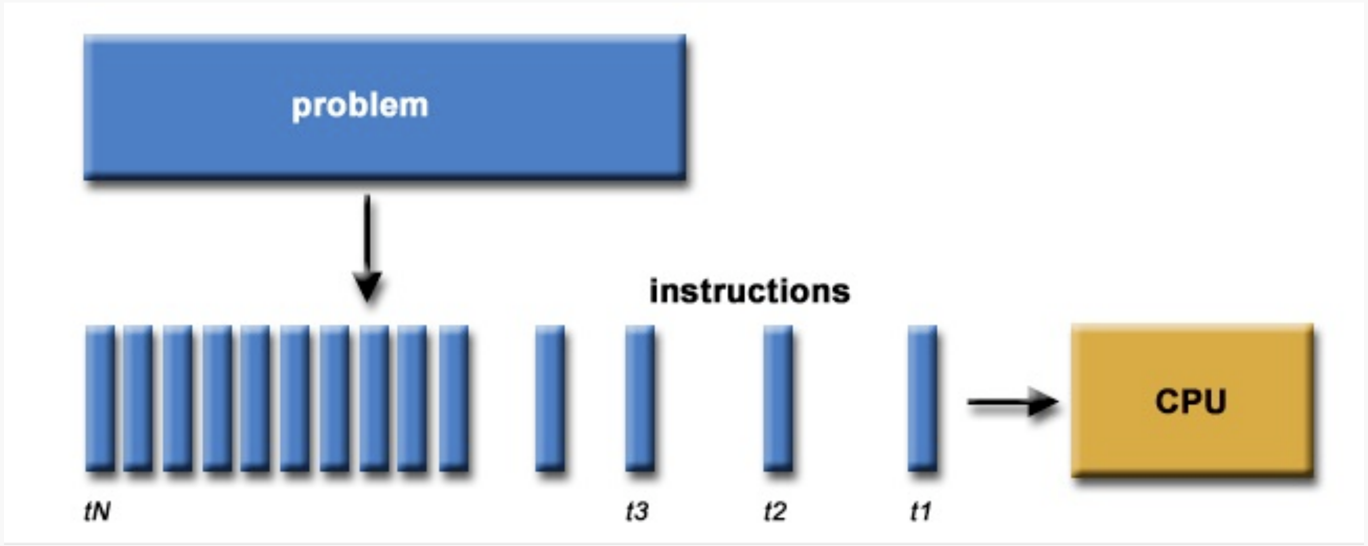
- Pick a topic – do a project

End of September – present project topic

Table of Contents

Introduction to Parallelism

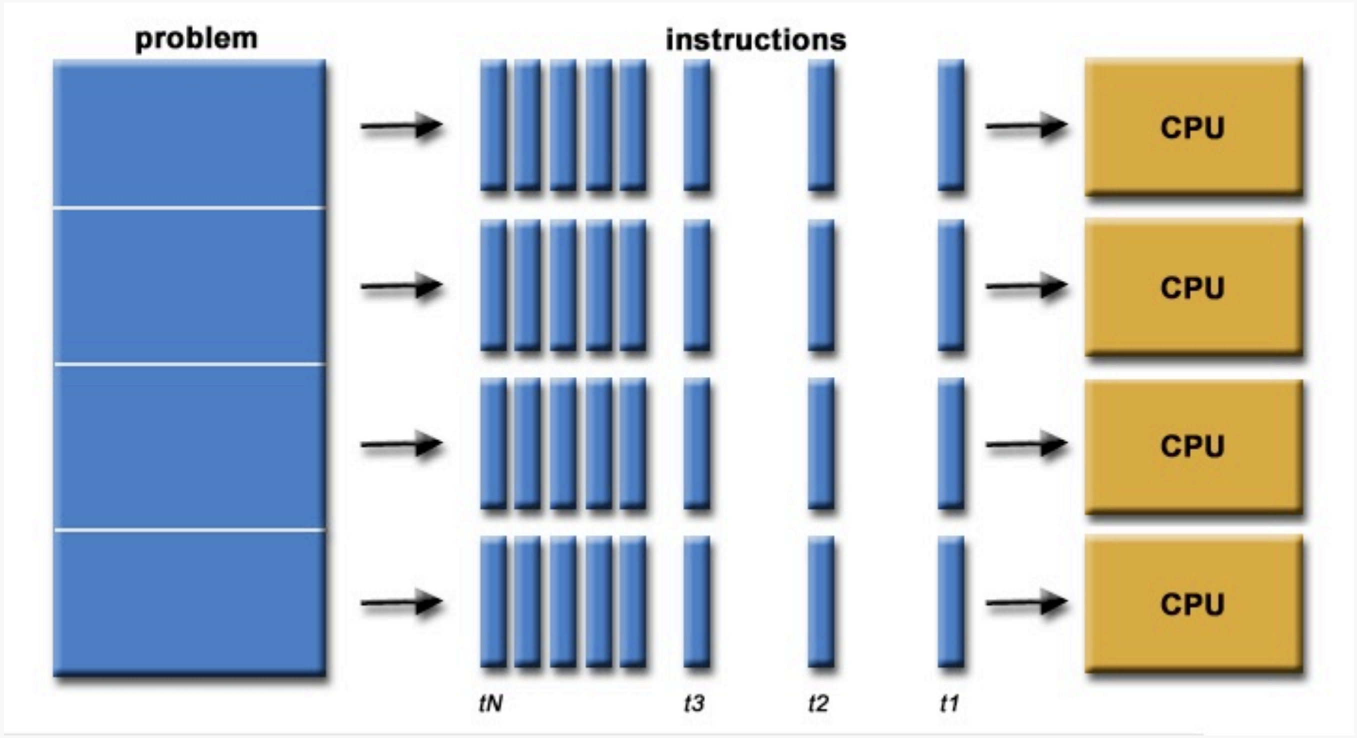
- Sequential Computing
  - Single CPU executes stream of instructions.



Adapted from: http://www.llnl.gov/computing/tutorials/parallel_comp

Introduction to Parallelism

- Parallel computing
  - Partition problem into multiple, concurrent streams of instructions.



Classification

Flynn's Taxonomy (1966-now)		Nowadays
SISD *Single Instruction* *Single Data*	**SIMD** *Single Instruction* *Multiple Data*	**SPMD** *Single Program* *Multiple Data*
MISD *Multiple Instructions* *Single Data*	**MIMD** *Multiple Instructions* *Multiple Data*	**MPMD** *Multiple Program* *Multiple Data*

- Execution models impact the above programming model
- Traditional computer is SISD
- SIMD is *data parallelism* while MISD is pure *task parallelism*
- MIMD is a mixed model (harder to program)
- SPMD and MPMD are less synchronized than SIMD and MIMD
- SPMD is most used model, but MPMD is becoming popular

Introduction to Parallelism

- Goal of parallel computing
  - Save time - reduce wall clock time.
    - Speedup - $\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$
  - Solve larger problems - problems that take more memory than available to 1 CPU.

Reduce wall clock time

- Methods
  - Parallelizing serial algorithms (parallel loops)
    - Total number of operations performed changes only slightly
    - Scalability may be poor (Amdahl's law)
  - Develop parallel algorithms
    - Total number of operations may increase, but the running time decreases
- Work Complexity of
  - Serialized Parallel algorithm = Optimal Seq. Algorithm
  - Serialized Parallel algorithm > Optimal Seq. Algorithm.

    ➔ sub-optimal sequential complexity (not good)

Performance Models

- Abstract Machine Models (PRAM, BSP, and many, many others)
  - Allow asymptotical analysis and runtime estimations
  - Often inaccurate for selecting the right implementation/algorithm on a given architecture
- Programming Primitives Behavior
  - Allow the selection of the right implementation
  - Increases programming effort

Abstract Machine

- PRAM (Parallel RAM, shared memory)
  - Processors access a shared flat memory
  - Performing an operation or accessing a memory location has cost = 1
- BSP (Bulk Synchronous Parallel, distributed memory) (Leslie Valiant)
  - Computation proceeds through supersteps
  - Cost of a superstep is *w+hg+l*
  - *w* is the time for computing on local data
  - *h* is the size of the largest message sent
  - *g* and *l* are architectural parameters describing network bandwidth and latency, respectively
- Nested BSP (new version form Valiant)

Table of Contents

Parallel Programming Models

Parasol

Many languages and libraries exist for creating parallel applications.

Each presents a programming model to its users.

During this course, we'll discuss criteria for evaluating a parallel model and use them to explore various approaches.

OpenMP	Charm++	Linda
Pthreads	UPC	MapReduce
Cilk	STAPL	Matlab DCE
TBB	X10	CnC
HPF	Fortress	Cuda
MPI	Chapel	Streamit,
		Brooks

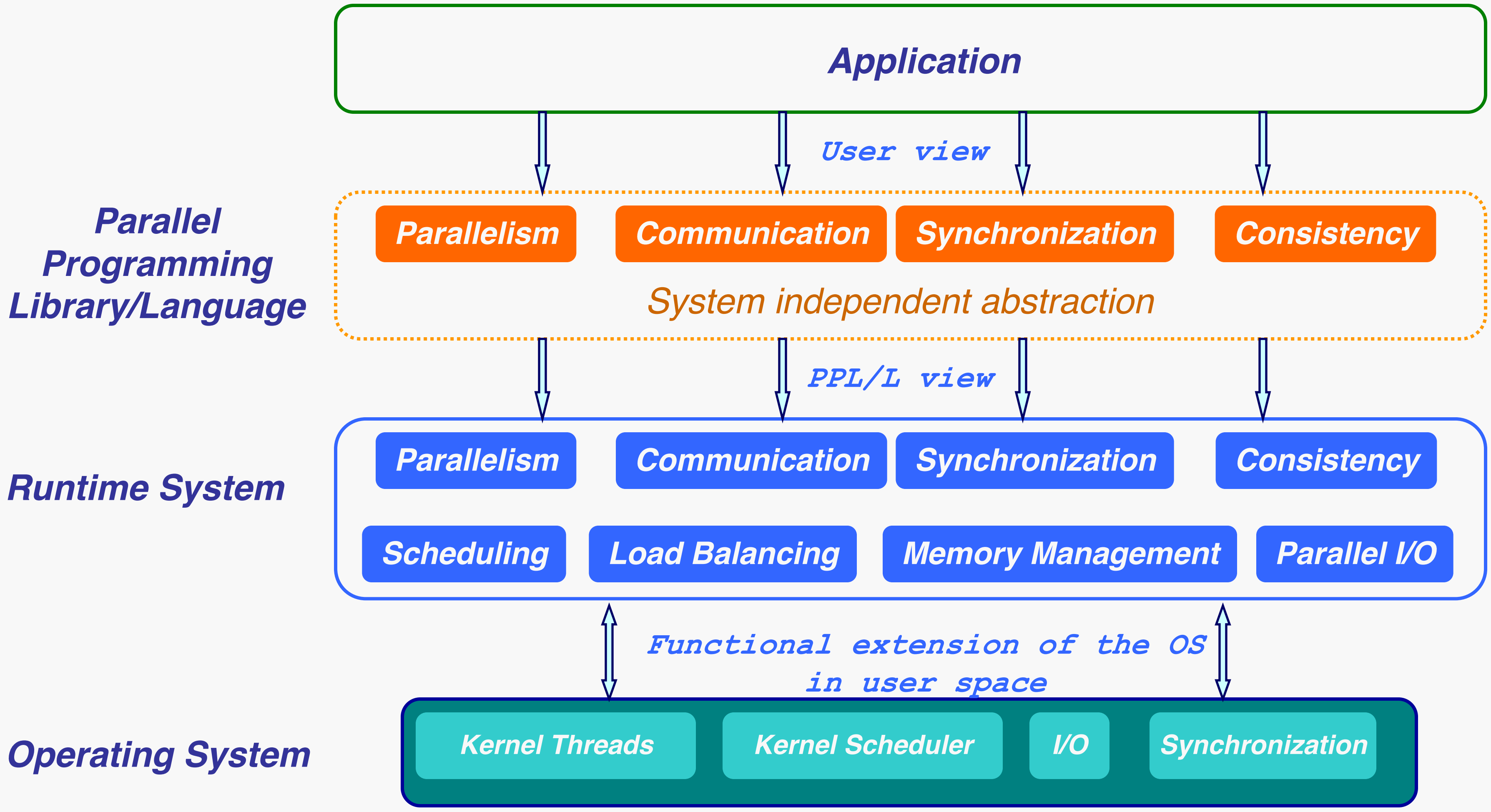
Programming Models Evaluation

What should we consider when evaluating a parallel programming model?

- Parallel Execution Model
- Productivity
- Performance
- Portability

Table of Contents

Parallel Execution Model



Parallel Execution Model

- Parallel Programming Model (user view)
  - Parallelism
  - Communication
  - Synchronization
  - Memory consistency
- Runtime System (RTS)
  - Introduction, definition and objectives
  - Usual services provided by the RTS
  - Portability / Abstraction

Parallel Programming Model (user view)

- Parallelism
- Communication
- Synchronization
- Memory consistency

PPM – Implicit Parallelism

Implicit parallelism (single-threaded view)

- User not required to be aware of the parallelism
  - User writes programs unaware of concurrency
    - Possible re-use previously implemented sequential algorithms
    - Often minor modifications to parallelize
  - User not required to handle synchronization or communication
    - Dramatic reduction in potential bugs
    - Straightforward debugging (with appropriate tools)
- Productivity closer to sequential programming
- Performance may suffer depending on application
- E.g. Matlab DCE, HPF, OpenMP*, Charm++*

* at various levels of implicitness

PPM – Explicit Parallelism

Explicit parallelism (multi-threaded view)

- User required to be aware of parallelism
  - User required to write parallel algorithms
    - Complexity designing parallel algorithms
    - Usually impossible to re-use sequential algorithms (except for embarrassingly parallel ones)
  - User responsible for synchronization and/or communication
    - Major source of bugs and faulty behaviors (e.g. deadlocks)
    - Hard to debug
    - Hard to even reproduce bugs
- Considered low-level
  - Productivity usually secondary
  - Best performance when properly used, but huge development cost
  - E.g. MPI, Pthreads

PPM – Mixed Parallelism

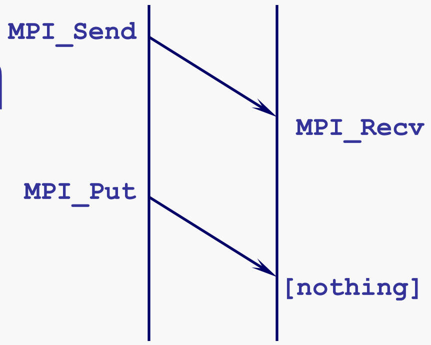
Mixed view

- Basic usage does not require parallelism awareness
- Optimization possible for advanced users

- Benefits from the two perspectives
  - High productivity for the general case
  - High performance possible by fine-tuning specific areas of the code
- E.g., STAPL, Chapel, Fortress

Table of Contents

PPM – Explicit Communication



Explicit Communication

- Message Passing (two-sided communication, P2P)
  - User explicitly sends/receives messages (e.g., MPI)
  - User required to match every Send operation with a Receive
  - Implicitly synchronizes the two threads
    - Often excessive synchronization (reduces concurrency)
    - Non-blocking operations to alleviate the problem (e.g., MPI_Isend/Recv)
- One-sided communication
  - User uses get/put operations to access memory (e.g., MPI-2, GASNet, Cray T3D)
  - No implicit synchronization (i.e., asynchronous communication)

PPM – Explicit Communication

Explicit Communication – Active Message, RPC, RMI

- Based on Message Passing
- Messages activate a handler function or method on the remote side
- Asynchronous
  - No return value (no `get` functions)
  - Split-phase programming model (e.g. Charm++, GASNet)
    - Caller provides a callback handler to asynchronously process “return” value
- Synchronous
  - Blocking semantic (caller stalls until acknowledgement/return is received)
  - Possibility to use `get` functions
- Mixed (can use both)
  - E.g., ARMI (STAPL)

PPM – Implicit Communication

Parasol

Implicit Communication

- Communication through shared variables
- Synchronization is primary concern
  - Condition variables, blocking semaphores or monitors
  - Full/Empty bit
- Producer/consumer between threads are expressed with synchronizations
- Increases productivity
  - User does not manage communication
  - High risk of introducing bugs

Table of Contents

PPM – Explicit Synchronization

Parasol

Explicit Synchronization

- Critical section / locks
  - One thread allowed to execute the guarded code at a time
- Condition variables / blocking semaphores
  - Producer/consumer synchronization
  - Introduces order in the execution
- Monitors / counting semaphores
  - Shared resources management
- Barrier / Fence (global synchronization)
  - Threads of execution wait until all reach the same point
- E.g., Pthreads, TBB, OpenMP

PPM – Implicit Synchronization

Implicit Synchronization

- Hidden in communication operations (e.g., two-sided communication)
- Data Dependence Graph (DDG)
  - PPL synchronizes where necessary to enforce the dependences
  - E.g., STAPL, CnC
- Distributed Termination Detection
  - When implemented as background algorithm (e.g., in Charm++, STAPL)
- Improved productivity
  - Less bugs from race conditions, deadlocks …
- E.g., STAPL, Charm++, MPI-1 and GASNet (to a certain extent)

Table of Contents

Runtime System (RTS)

- Introduction
  - Definition
  - Objectives
- Usual services provided by the RTS
- Portability / Abstraction

RTS – Introduction

- Software layer
  - Linked with the application
  - Executes in user space
- Provides applications with functionalities
  - Missing in the Operating System and drivers
  - More advanced/specialized than the OS counterpart
- Virtualizes OS and Architecture

RTS – Definition*

Functional extension of the Operating System in user space

- No precise definition available
- Fuzzy functional boundary between RTS and OS
  - Services are often a refined or extended version of the OS
  - Functional redundancy with OS services
    - Avoid entering Kernel space
    - Provide reentrancy
    - E.g., threading, synchronization, scheduling …
- Widely variable set of provided services
  - No minimum requirements
  - No limit on the amount of functionality

***Non-formal definition**

RTS – Objectives

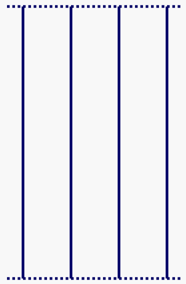
Objectives of RTS for Parallel Programming Languages/Libraries:

- Enable portability
  - Decouple the PPL from the system
  - Exploit system-specific optimized features (e.g., RDMA, Coprocessor)
- Abstract complexity of large scale heterogeneous systems to enable portable scalability
  - Provide uniform communication model
  - Manage threading, scheduling and load-balancing
  - Provide parallel I/O and system-wide event monitoring
- Improve integration between application and system
  - Use application runtime information
    - Improve RTS services (e.g., scheduling, synchronization)
    - Adaptive selection of specialized code

RTS – Provided Services

- **Common RTS provide a subset of the following** (not limited to)
  - Parallelism
    - Type of parallelism (API)
    - Threading Model (underlying implementation)
  - Communication
  - Synchronization
  - Consistency
  - Scheduling
  - Dynamic Load Balancing
  - Memory Management
  - Parallel I/O
- **Some functionalities are only provided as a thin abstraction layer on top of the OS service**

RTS – Flat Parallelism

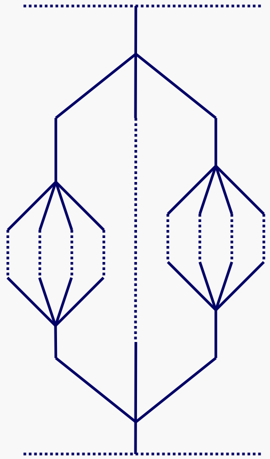


Parallelism types – Flat Parallelism

- All threads of execution have the same status
  - No parent/child relationship
- Threads are active during the whole execution
- Usually constant number of threads of execution

- Well adapted for problems with large granularity
- Difficult to achieve load-balance for non-embarrassingly parallel applications
- E.g. MPI

RTS – Nested Parallelism

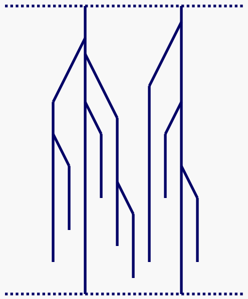


Parallelism types – Nested Parallelism

- Parallelism is hierarchal
  - Threads of execution can spawn new threads to execute their task
  - Exploits multiple levels of parallelism (e.g. nested parallel loops)
- Good affinity with heterogeneous architectures (e.g. clusters of SMPs)*
  - Allows the exploitation of different levels of granularity
- Natural fit for composed parallel data structures*
  - E.g. `p_vector< p_list< Type > >`
- E.g. OpenMP (not quite there) , Cilk, TBB

* Also for dynamic parallelism.

RTS – Dynamic Parallelism



Parallelism types – Dynamic Parallelism

- Threads of execution are dynamically created whenever new parallelism is available
  - Exploits any granularity of parallelism available
  - Necessary to achieve scalability for dynamic applications
- Improves load-balancing for dynamic applications
  - Work stealing
  - Thread migration
- Parallelism can be dynamically refined (e.g. mesh refinement*)
- E.g. STAPL, Charm++, AMPI, Chapel

* Can also be achieved by redistributing the data.

RTS – Threading Models (1:1)

1:1 threading model: (1 user-level thread mapped onto 1 kernel thread)

- Default kernel scheduling
  - Possibility to give hints to scheduler (e.g., thread priority levels)
  - Reduced optimization opportunities
- Heavy kernel threads
  - Creation, destruction and swapping are expensive
  - Scheduling requires to cross into kernel space
- E.g., Pthreads

RTS – Threading Models (M:1)

M:1 threading model: (M user-level threads mapped onto 1 kernel thread)

- Customizable scheduling
  - Enables scheduler-based optimizations (e.g., priority scheduling, good affinity with latency hiding schemes)
- Light user-level threads
  - Lesser threading cost
    - User-level thread scheduling requires no kernel trap
- Problem: no effective parallelism
  - User-level threads’ execution serialized on 1 kernel thread
  - Often poor integration with the OS (little or no communication)
  - E.g., GNU Portable Threads

RTS – Threading Models (M:N)

M:N threading model: (M user-level threads mapped onto N kernel threads)

- Customizable scheduling
  - Enables scheduler-based optimizations (e.g. priority scheduling, better support for relaxing the consistency model …)
- Light user-level threads
  - Lesser threading cost
    - Can match N with the number of available hardware threads : no kernel-thread swapping, no preemption, no kernel over-scheduling …
    - User-level thread scheduling requires no kernel trap
  - Perfect and free load balancing within the node
    - User-level threads are cooperatively scheduled on the available kernel threads (they migrate freely).
- E.g., PM2/Marcel (Univ. of Bordeaux)

RTS – Communication

- Systems usually provide low-level communication primitives
  - Not practical for implementing high-level libraries
  - Complexity of development leads to mistakes
- Often based on other RTS libraries
  - Layered design conceptually based on the historic ISO/OSI stack
  - OSI layer-4 (end-to-end connections and reliability) or layer-5 (inter-host communication)
  - Communication data is not structured
  - E.g., MPI, Active Message, SHMEM
- **Objective:** Provide structured communication
  - OSI layer-6 (data representation) – data is structured (type)
  - E.g., RMI, RPC

RTS – Synchronization

- Systems usually provide low-level synchronization primitives (e.g., semaphores)
  - Impractical for implementing high-level libraries
  - Complexity of development leads to mistakes
- Often based on other RTS libraries
  - E.g., POSIX Threads, MPI …
- Objective: Provide appropriate synchronization primitives
  - Shared Memory synchronization
    - E.g., Critical sections, locks, monitors, barriers …
  - Distributed Memory synchronization
    - E.g., Global locks, fences, barriers …

RTS – Consistency

- In shared memory systems
  - Use system's consistency model
  - Difficult to improve performance in this way

- In distributed systems: relaxed consistency models
  - Processor Consistency
    - Accesses from a processor on another's memory are sequential
    - Limited increase in level of parallelism
  - Object Consistency
    - Accesses to different objects can happen out of order (inconsistent)
    - Uncovers fine-grained parallelism
      - Accesses to different objects are concurrent
      - Potential gain in scalability

RTS – Scheduling

- Available for RTS providing some user-level threading (M:1 or M:N)
- Performance improvement
  - Threads can be cooperatively scheduled (no preemption)
  - Swapping does not require to cross into kernel space
- Automatically handled by RTS
- Provide API for user-designed scheduling

RTS – Dynamic Load Balancing

- Available for RTS providing some user-level threading (M:1 or M:N)
- User-level threads can be migrated
  - Push: the node decides to offload part of its work on another
  - Pull: when the node idles, it takes work from others (work stealing)
- For the M:N threading model
  - Perfect load balance within the node (e.g., dynamic queue scheduling of user-level threads on kernel threads)
  - Free within the node (I.e., no additional cost to simple scheduling)

RTS – Memory Management

- RTS often provide some form of memory management
  - Reentrant memory allocation/deallocation primitives
  - Memory reuse
  - Garbage collection
  - Reference counting

- In distributed memory
  - Can provide Global Address Space
    - Map every thread's virtual memory in a unique location
  - Provide for transparent usage of RDMA engines

RTS – Parallel I/O

- I/O is often the bottleneck for scientific applications processing vast amounts of data
- Parallel applications require parallel I/O support
  - Provide abstract view to file systems
  - Allow for efficient I/O operations
  - Avoid contention, especially in collective I/O
- E.g., ROMIO implementation for MPI-IO
- Archive of current Parallel I/O research:

  http://www.cs.dartmouth.edu/pario/
- List of current projects:

  http://www.cs.dartmouth.edu/pario/projects.html

RTS – Portability / Abstraction

- Fundamental role of runtime systems
  - Provide unique API to parallel programming libraries/languages
  - Hide discrepancies between features supported on different systems
- Additional layer of abstraction
  - Reduces complexity
  - Encapsulates usage of low-level primitives for communication and synchronization
- Improved performance
  - Executes in user space
  - Access to application information allows for optimizations

Some related References

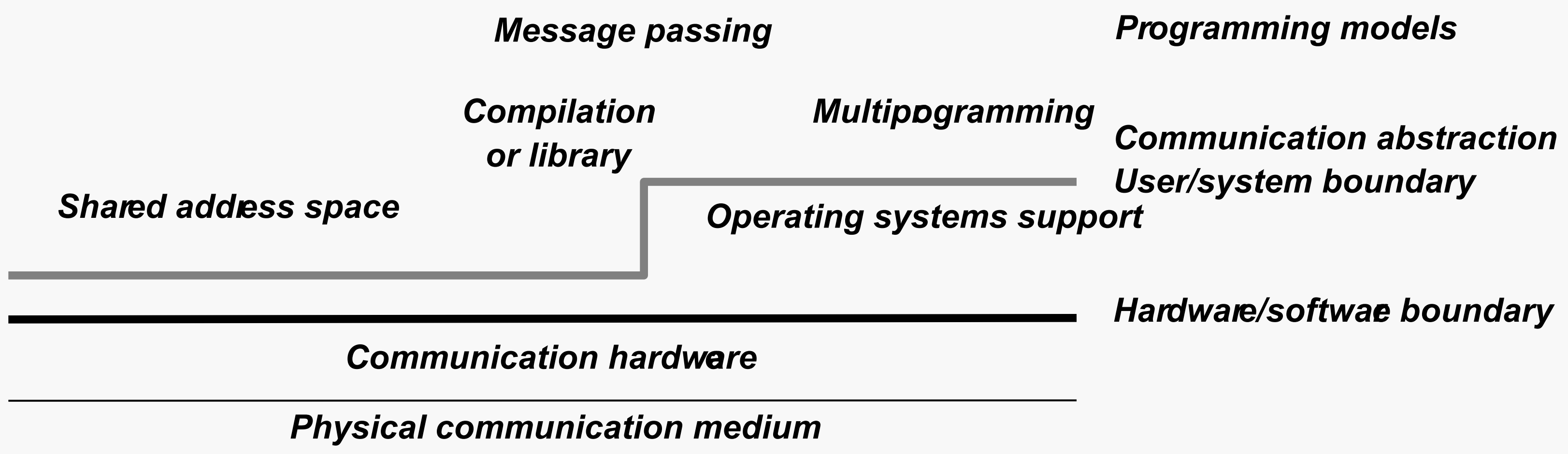
- Hill, M. D. 1998. Multiprocessors Should Support Simple Memory-Consistency Models. Computer 31, 8 (Aug. 1998), 28-34. DOI=http://dx.doi.org/10.1109/2.707614
- Adve, S. V. and Gharachorloo, K. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (Dec. 1996), 66-76. DOI=http://dx.doi.org/10.1109/2.546611
- Dictionary on Parallel Input/Output, by Heinz Stockinger, February 1998.

Table of Contents

Shared Memory Multiprocessors

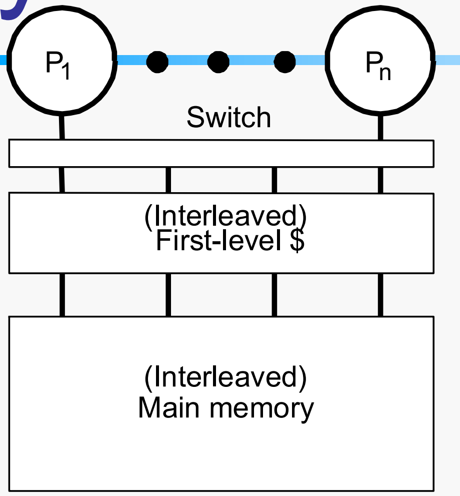
- Symmetric Multiprocessors (SMPs)
  - Symmetric access to all of main memory from any processor (a bit of a PRAM)
- Normal uniprocessor mechanisms to access data (reads and writes)
  - Key is extension of memory hierarchy to support multiple processors

Supporting Programming Models

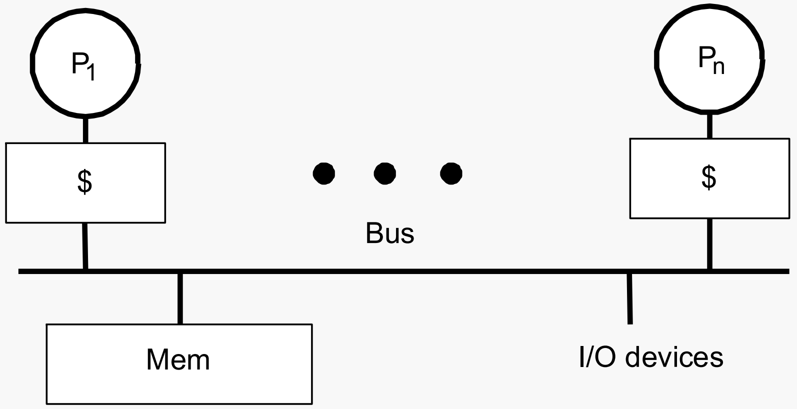


- Address translation and protection in hardware
- Message passing using shared memory buffers
  - can be very high performance since no OS involvement necessary
- Focus here on supporting coherent shared address space

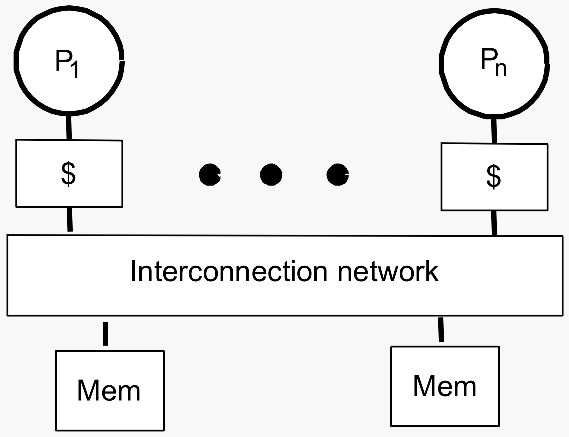
Natural Extensions of Memory System



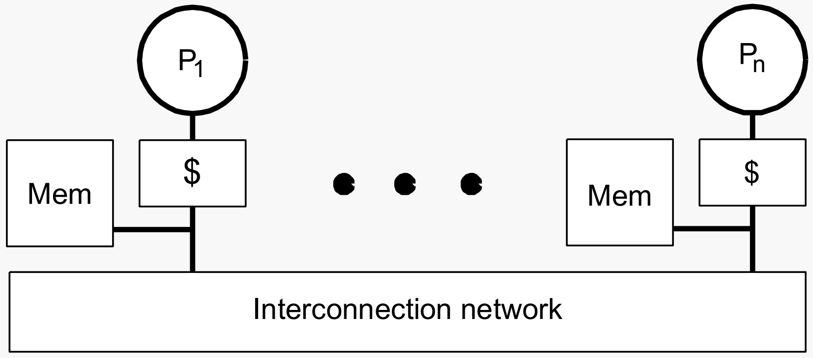
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Caches and Cache Coherence

- Caches play key role in all cases
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect

Cache Coherence

- But private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - They'll keep accessing stale value in their caches
  - *Cache coherence* problem
  - Need to take actions to ensure visibility

Focus: Bus-based, Centralized Memory

- Shared cache
  - Low-latency sharing and prefetching across processors
  - Sharing of working sets
  - No coherence problem (and hence no false sharing either)
  - But high bandwidth needs and negative interference (e.g. conflicts)
  - Hit and miss latency increased due to intervening switch and cache size

Focus: Bus-based, Centralized Memory

Today: for multiprocessor on a chip (for small-scale systems or nodes)

- Distributed memory
  - Most popular way to build scalable systems, discussed later

Outline

- Coherence and Consistency
- Snooping Cache Coherence Protocols
- Quantitative Evaluation of Cache Coherence Protocols
- Synchronization
- Implications for Parallel Software

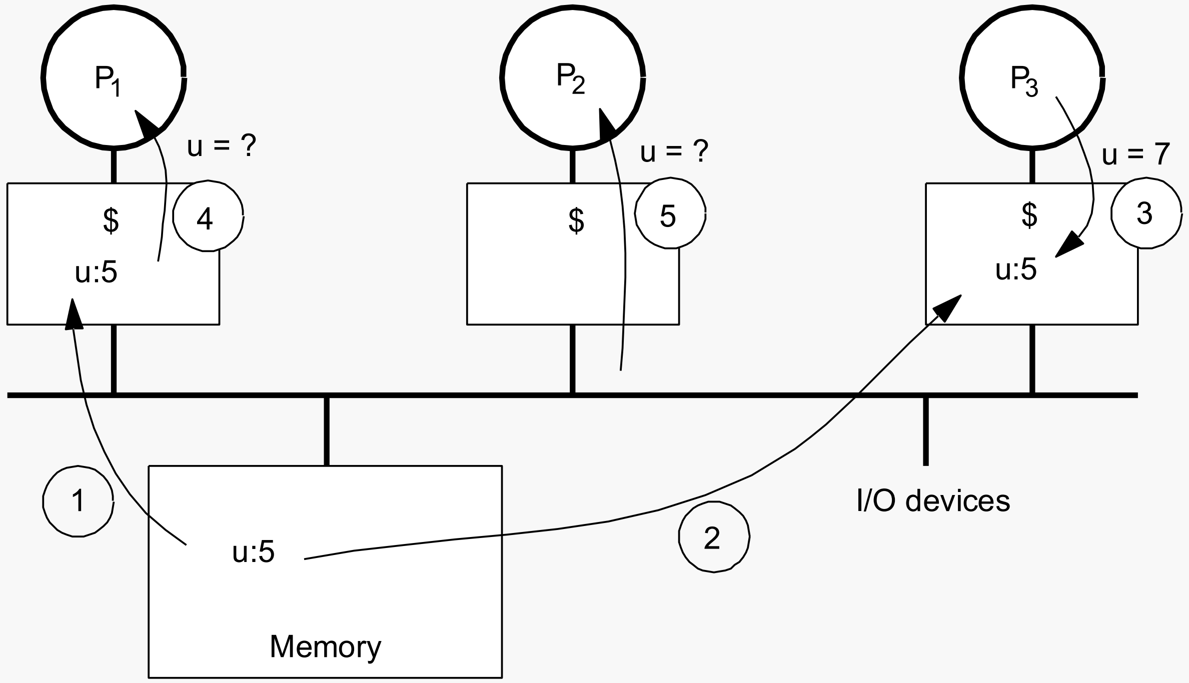
A Coherent Memory System

- Reading a location should return latest value written (by any process)
- Easy in uniprocessors
  - Except for I/O: coherence between I/O devices and processors
  - But infrequent so software solutions work
    - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches
- Would like same to hold when processes run on different processors
  - E.g. as if the processes were interleaved on a uniprocessor

A Coherent Memory System

- But coherence problem much more critical in multiprocessors
  - Pervasive
  - Performance-critical
  - Must be treated as a basic hardware design issue

Cache Coherence Problem



Cache Coherence Problem

- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

Problems with the Intuition

- Recall: Value returned by read should be last value written
- But “last” is not well-defined
- Even in seq. case, last defined in terms of program order, not time
  - Order of operations in the machine language presented to processor
  - “Subsequent” defined in analogous way, and well defined

Problems with the Intuition

- In parallel case, program order defined within a process, but need to make sense of orders across processes
- Must define a meaningful semantics

Some Basic Definitions

- Extend from definitions in uniprocessors to those in multiprocessors
- *Memory operation*: a single read (load), write (store) or read-modify-write access to a memory location
  - Assumed to execute atomically w.r.t each other
- *Issue*: a memory operation issues when it leaves processor's internal environment and is presented to memory system (cache, buffer …)

Some Basic Definitions

- *Perform*: operation appears to have taken place, as far as processor can tell from other memory operations it issues
  - A write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
  - A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read
- In multiprocessors, stay same but replace “the” by “a” processor
  - Also, *complete*: perform with respect to all processors
  - Still need to make sense of order in operations from different processes

Sharpening the Intuition

- Imagine a single shared memory and no caches
  - Every read and write to a location accesses the same physical location
  - Operation completes when it does so
- Memory imposes a *serial* or *total order* on operations to the location
  - Operations to the location from a given processor are in program order
  - The order of operations to the location from different processors is some interleaving that preserves the individual program orders

Sharpening the Intuition

- "Last" now means most recent in a hypothetical serial order that maintains these properties
- For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look, i.e. to read)
- Note that the total order is never really constructed in real systems
  - Don't even want memory, or any hardware, to see all operations
- But program should behave as if some serial order is enforced
  - Order in which things appear to happen, not actually happen

Formal Definition of Coherence

- *Results of a program*: values returned by its read operations
- A memory system is *coherent* if the results of any execution of a program are such that at each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:
- 1. operations issued by any particular process occur in the order issued by that process, and
- 2. the value returned by a read is the value written by the last write to that location in the serial order
- Two necessary features:
  - *Write propagation*: value written must become visible to others
  - *Write serialization*: writes to location seen in same order by all
    - if I see w1 after w2, you should not see w2 before w1
    - no need for analogous read serialization since reads not visible to others

Cache Coherence Using a Bus

- Built on top of two fundamentals of uniprocessor systems
  - Bus transactions
  - State transition diagram in cache
- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible

Cache Coherence Using a Bus

- Uniprocessor cache states:
  - Effectively, every block is a finite state machine
  - Write-through, write no-allocate has two states: valid, invalid
  - Writeback caches have one more state: modified (“dirty”)
- Multiprocessors extend both these somewhat to implement coherence

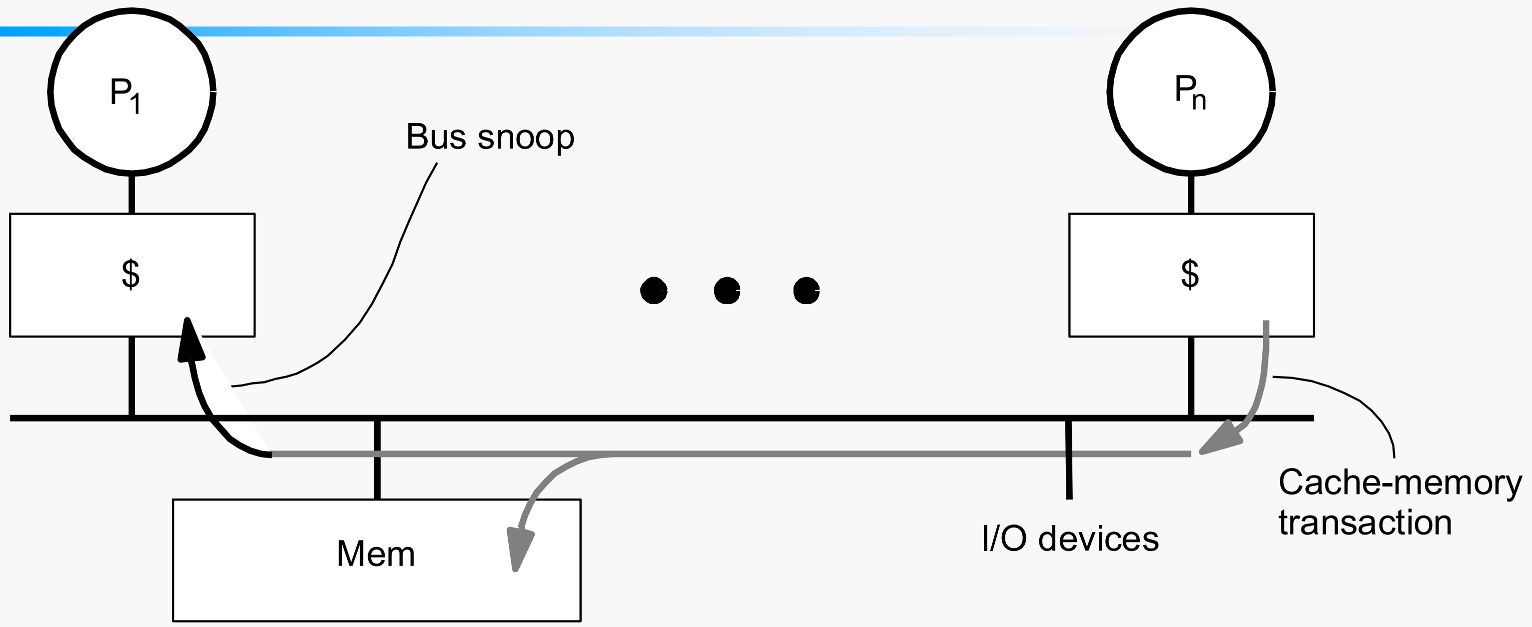
Snooping-based Coherence

- *Basic Idea*
- Transactions on bus are visible to all processors
- Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)

Implementation

- Cache controller now receives inputs from both sides:
  - Requests from processor, bus requests/responses from snooper
- In either case, takes zero or more actions
  - Updates state, responds with data, generates new bus transactions
- Protocol is distributed algorithm: cooperating state machines
  - Set of states, state transition diagram, actions
- Granularity of coherence is typically cache block
  - Like that of allocation in cache and transfer to/from cache

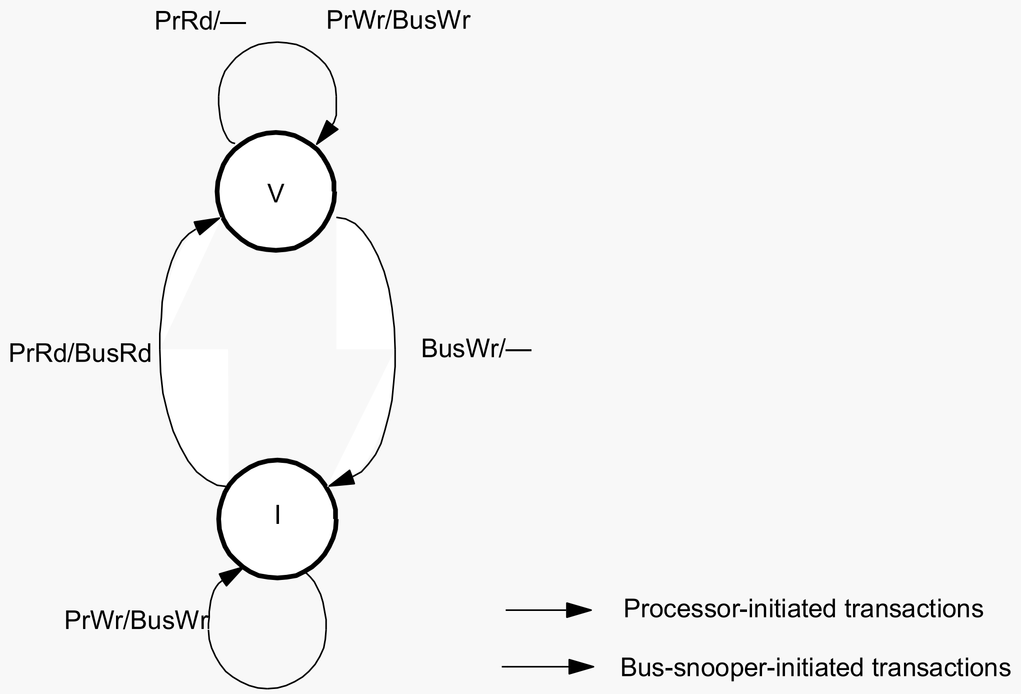
Coherence with Write-through Caches



– Key extensions to uniprocessor: snooping, invalidating/updating caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols

– Write propagation: even in inval case, later reads will see new value: inval causes miss on later access, and memory up-to-date via write-through

Write-through State Transition Diagram


PrRd/—
PrWr/BusWr
V
PrRd/BusRd
BusWr/—
I
PrWr/BusWr
Processor-initiated transactions
Bus-snooper-initiated transactions

Write-through State Transition Diagram

- Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as *p*-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block, but write invalidates them

Is it Coherent?

- Construct total order that satisfies program order, write serialization?
- Assume atomic bus transactions and memory operations for now
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus xaction
  - (we'll relax these assumptions in more complex systems later)

Is it Coherent?

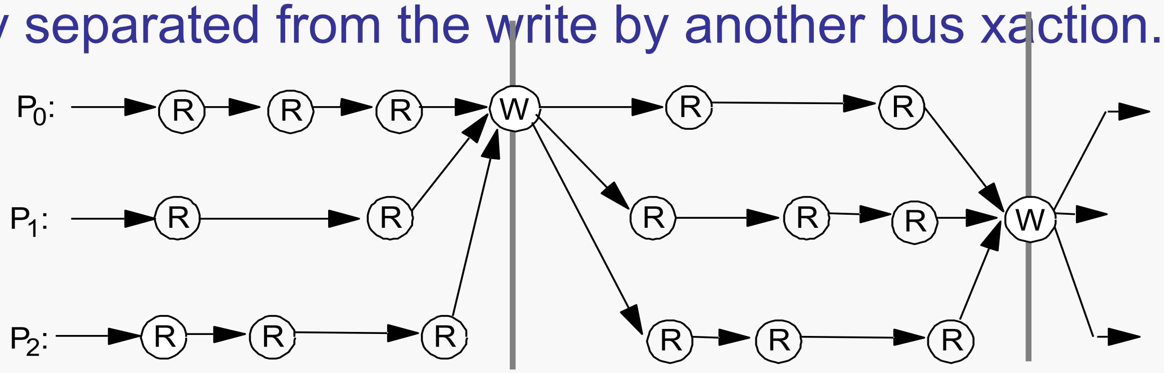
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (*bus order*)
  - Per above assumptions, invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering Reads

- Read misses: appear on bus, and will see last write in bus order
- Read hits: do not appear on bus
  - But value read was placed in cache by either
    - most recent write by this processor, or
    - most recent read miss by this processor
  - Both these transactions appear on the bus
  - So reads hits also see values as being produced in consistent bus order

Determining Orders More Generally

- A memory operation M2 is subsequent to a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.
- Read is subsequent to write W if read generates bus xaction that follows that for W.
- Write is subsequent to read or write M if M generates bus xaction and the xaction for the write follows that for M.
- Write is subsequent to read if read does not generate a bus xaction and is not already separated from the write by another bus xaction.



- ***Writes establish a partial order***
- ***Doesn't constrain ordering of reads, though bus will order read misses to :***

any order among reads between writes is OK, as long as in program order

Problem with Write-Through

- High bandwidth requirements
  - Every write from every processor goes to shared bus and memory
  - Consider 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
  - Each processor generates 30M stores or 240MB data per second
  - 1GB/s bus can support only about 4 processors without saturating
  - Write-through especially unpopular for SMPs

Problem with Write-Through

- Write-back caches absorb most writes as cache hits
  - Write hits don't go on bus
  - But now how do we ensure write propagation and serialization?
  - Need more sophisticated protocols: large design space
- But first, let's understand other ordering issues

What is Memory Consistency ?

Writes to a location become visible to all in the same order

But when does a write become visible

- How to establish orders between a write and a read by different procs?
- Typically use event synchronization, by using more than one location

P_1	P_2
/*Assume initial value of A and ag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

Example– Memory Consistency

Effect of write buffers

Example :

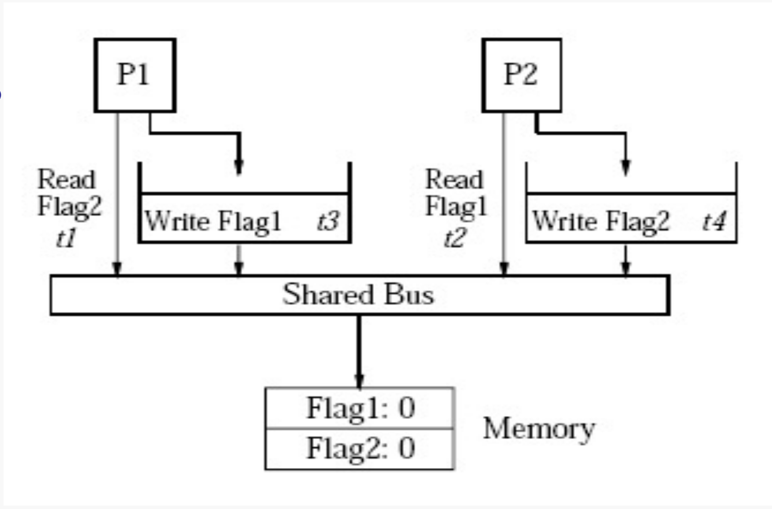
```
// Dekker's algorithm for critical sections
// Initially Flag1 = Flag2 = 0
```

```
            P1
Flag1 = 1;          W(Flag1)
If (Flag2 == 0)  R(Flag2)
// critical section
...
```

```
            P2
Flag2 = 1;          W(Flag2)
if (Flag1 == 0)  R(Flag1)
// critical section
...
```

Correct execution if a processor's Read operation returns 0 iff its Write operation occurred before both operations on the other processor.

- Relaxed consistency : buffer write operations
  - Breaks Sequential Consistency
  - Invalidates Dekker's algorithm
  - Write operations delayed in buffer



Memory Consistency

```
      P1                          P2
---------------------------------------------------------
        /*Assume initial value of A and  ag is 0*/
  A = 1;                 while (flag == 0); /*spin idly*/
  flag = 1;              print A;
```

- Sometimes expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence doesn’t help: pertains only to single location
- Intuition not guaranteed by coherence

Caches Complicate Things More

- Multiple copies of the same location

```
        P1                  P2                     P3

     A = 1;          wait (A == 1);

                         B = 1;
                                            wait (B == 1);

                                               .. = A;
```

- P3 had A=B=0 in its cache, invalidations for B have arrived before the invalidations for A. P3 reads 0
- Cache has a hiding effect

Another Example of Orders

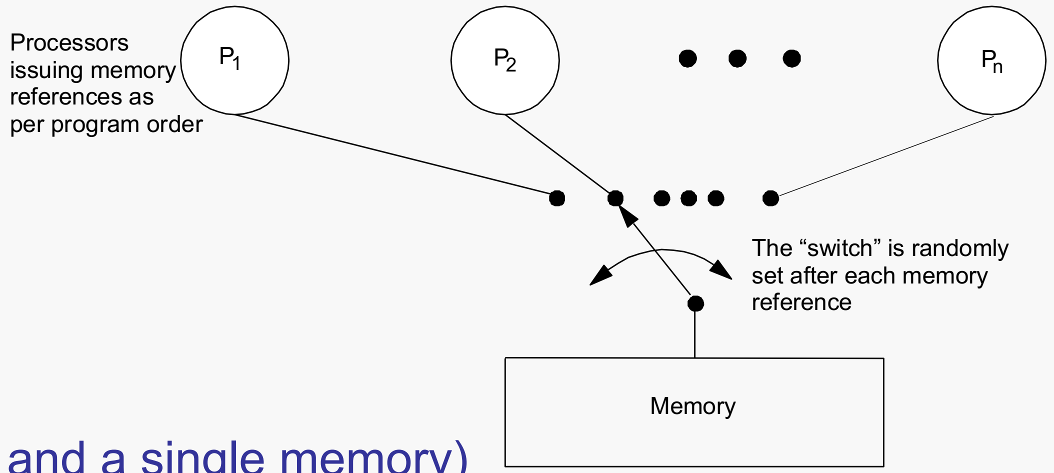
P_1	P_2
/*Assume initial values of A and B are 0*/	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- What's the intuition?
- Whatever it is, we need an ordering model for clear semantics
  - across different locations as well
  - so programmers can reason about what results are possible
- This is the memory consistency model

Memory Consistency Model

- Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Without it, can’t tell much about an Shared Address Space (SAS) program’s execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency


Processors issuing memory references as per program order
P_1
P_2
P_n
The "switch" is randomly set after each memory reference
Memory

- (as if there were no caches, and a single memory)
- Total order achieved by *interleaving* accesses from different processes
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
- Programmer's intuition is maintained

- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

What is Program Order?

- Intuitively, order in which operations appear in source code
  - Straightforward translation of source code to assembly
  - At most one memory operation per instruction
- But not the same as order presented to hardware by compiler
- So which is program order?
- Depends on which layer, and who's doing the reasoning
- *We assume order as seen by programmer*

What matters is order in which *appears to execute*, not *executes*

P_1	P_2
/*Assume initial values of A and B are 0*/	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- possible outcomes for (A,B): (0,0), (1,0), (1,2); impossible under SC: (0,2)
- we know 1a->1b and 2a->2b by program order
- A = 0 implies 2b->1a, which implies 2a->1b
- B = 2 implies 1b->2a, which leads to a contradiction

- BUT, actual execution 1b->1a->2b->2a is SC, despite not program order
  - appears just like 1a->1b->2a->2b as visible from results
- actual execution 1b->2a->2b-> is not SC

Another Example

- Initially: all vars are 0

P1	P2
A =1	
Flag = 1	x = Flag
	y = A

- Possible (x,y) = (0,0),(0,1),(1,1)
- Impossible (x,y) = (1,0)

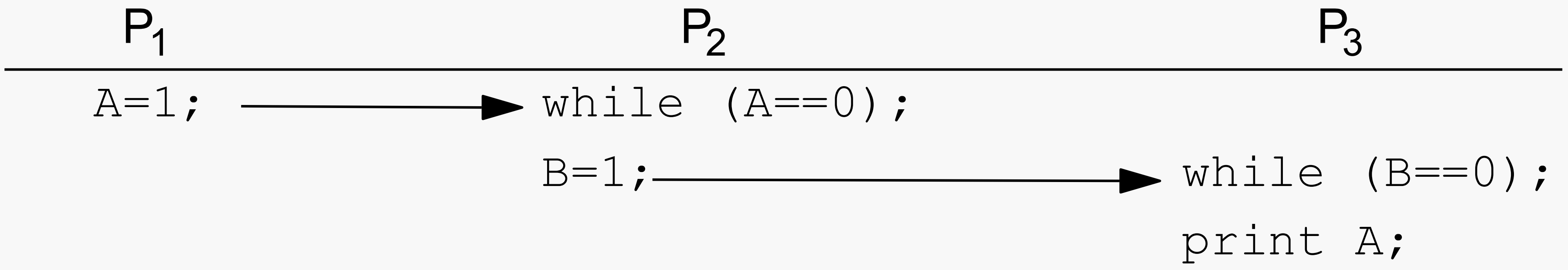
Implementing SC

- Two kinds of requirements
  - Program order
    - memory operations issued by a process must appear to become visible (to others and itself) in program order
  - Atomicity
    - in the overall total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - needed to guarantee that total order is consistent across processes
    - tricky part is making writes atomic

Write Atomicity

- *Write Atomicity*: Position in total order at which a write appears to perform should be the same for all processes
  - Nothing a process does after it has seen the new value produced by a write W should be visible to other processes until they too have seen W
  - In effect, extends write serialization to writes from multiple processes

Write Atomicity



```
A=1;          while (A==0);
              B=1;              while (B==0);
                                print A;
```

- Transitivity implies A should print as 1 under SC
- Problem if P_2 leaves loop, writes B, and P_3 sees new B but old A (from its cache, say)

More Formally

- Each process's program order imposes partial order on set of all operations
- Interleaving of these partial orders defines a total order on all operations
- Many total orders may be SC (SC does not define particular interleaving)

- *SC Execution*: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving)
- *SC System*: A system is SC if any possible execution on that system is an SC execution

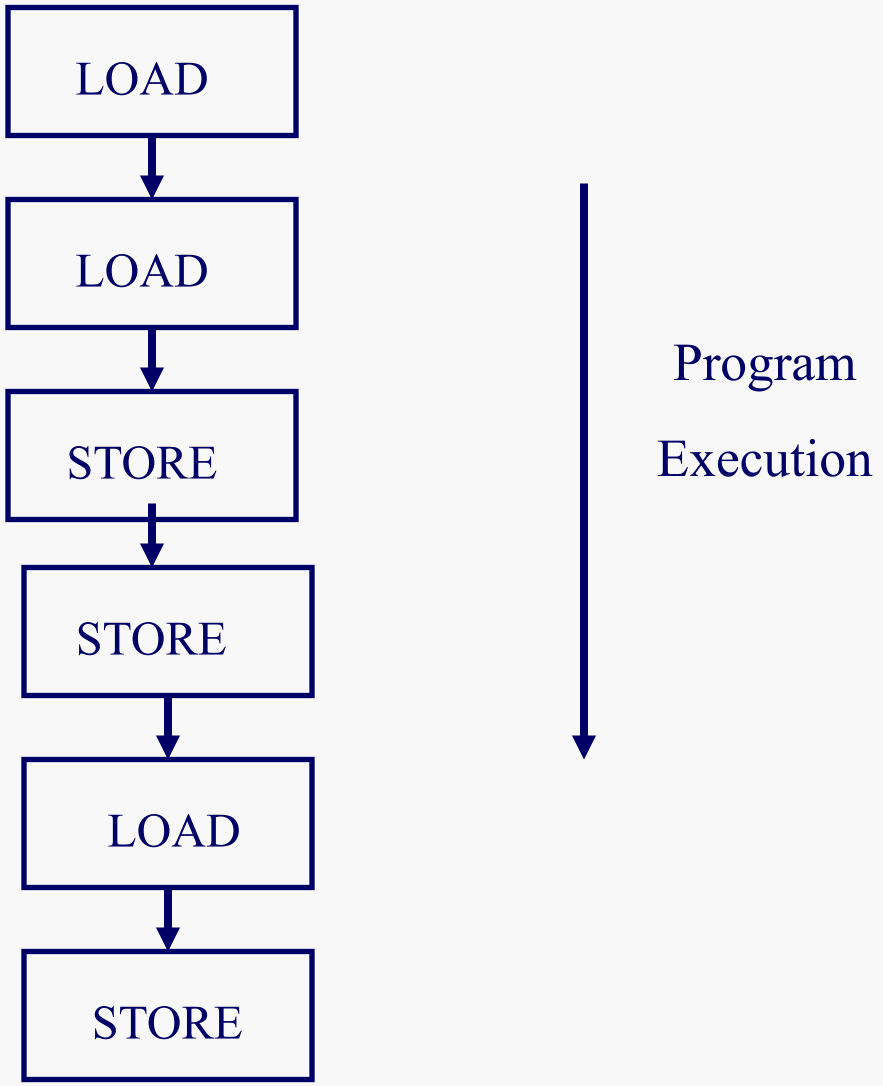
Sequential Consistency (SC)

- Before a LOAD is allowed to perform wrt any processor, all previous LOAD/STORE accesses must be performed wrt everyone

- Before a STORE …. (same)

/* Note GLOBALLY performed */

Sequential Consistency (SC)



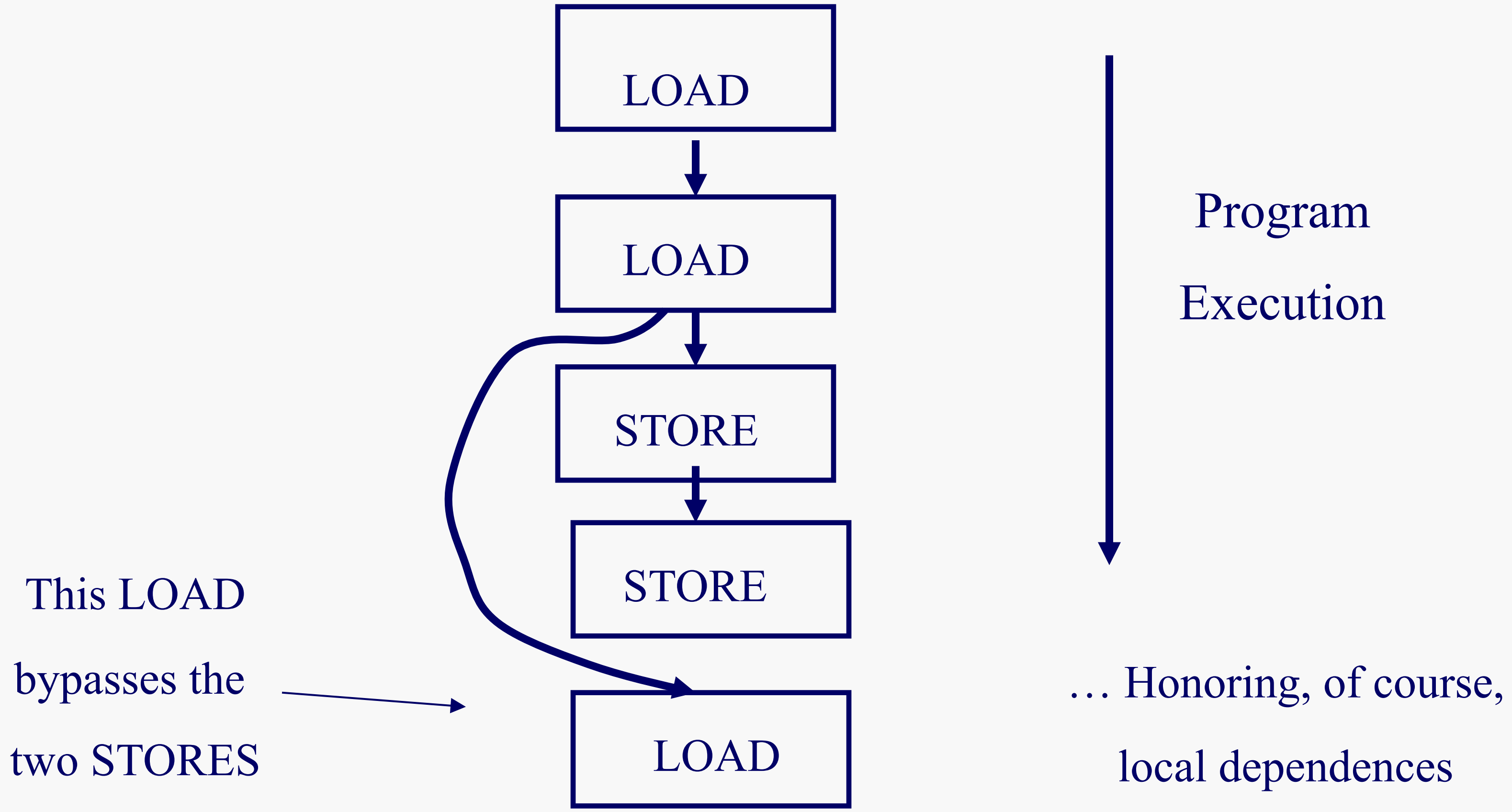
Sufficient Conditions for SC

- Every process issues memory operations in program order
- After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
- After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity)

- Sufficient, not necessary, conditions
- Clearly, compilers should not reorder for SC, but they do!
  - Loop transformations, register allocation (eliminates!)
- Even if issued in order, hardware may violate for better performance
  - Write buffers, out of order execution
- Reason: uniprocessors care only about dependences to same location
  - Makes the sufficient conditions very restrictive for performance

Processor Consistency

- Main idea: LOADs are allowed to bypass STORES


LOAD
LOAD
STORE
STORE
LOAD
Program Execution
This LOAD bypasses the two STORES
… Honoring, of course, local dependences

Processor Consistency

- Before a LOAD is allowed to perform wrt any processor, all previous LOAD/~~STORE~~ accesses must be performed wrt everyone

- Before a STORE ….

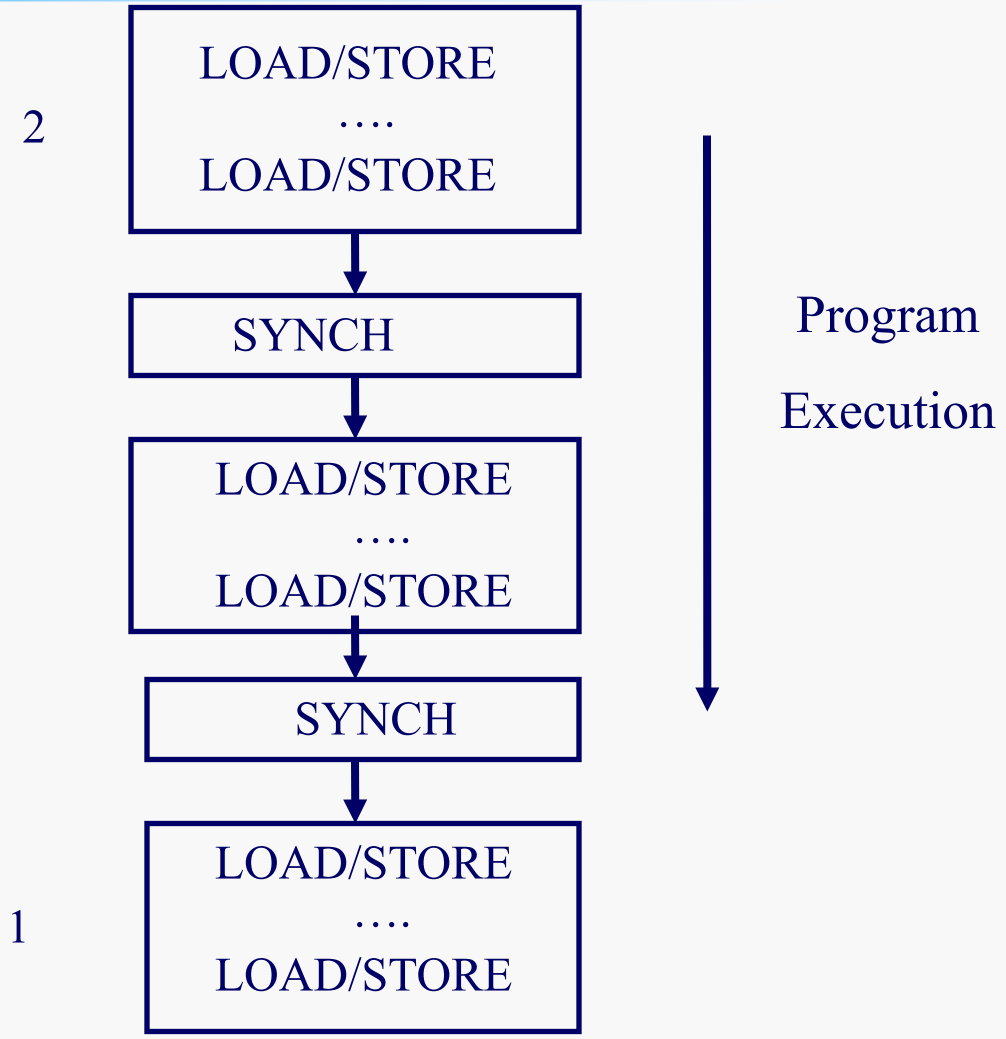
  …. LOAD/STORE ...

Order among the LD and the ST respectively

Weak Consistency

- Suppose we are in a critical section
- Then, we can have several accesses pipelined b/c programmer has made sure that:
  - no other process can rely on that data structure being consistent until the critical section is exited
- Pros: Higher performance (more overlap)
- Cons: Need to distinguish between ordinary LOAD/STORES and SYNCH

Weak Consistency



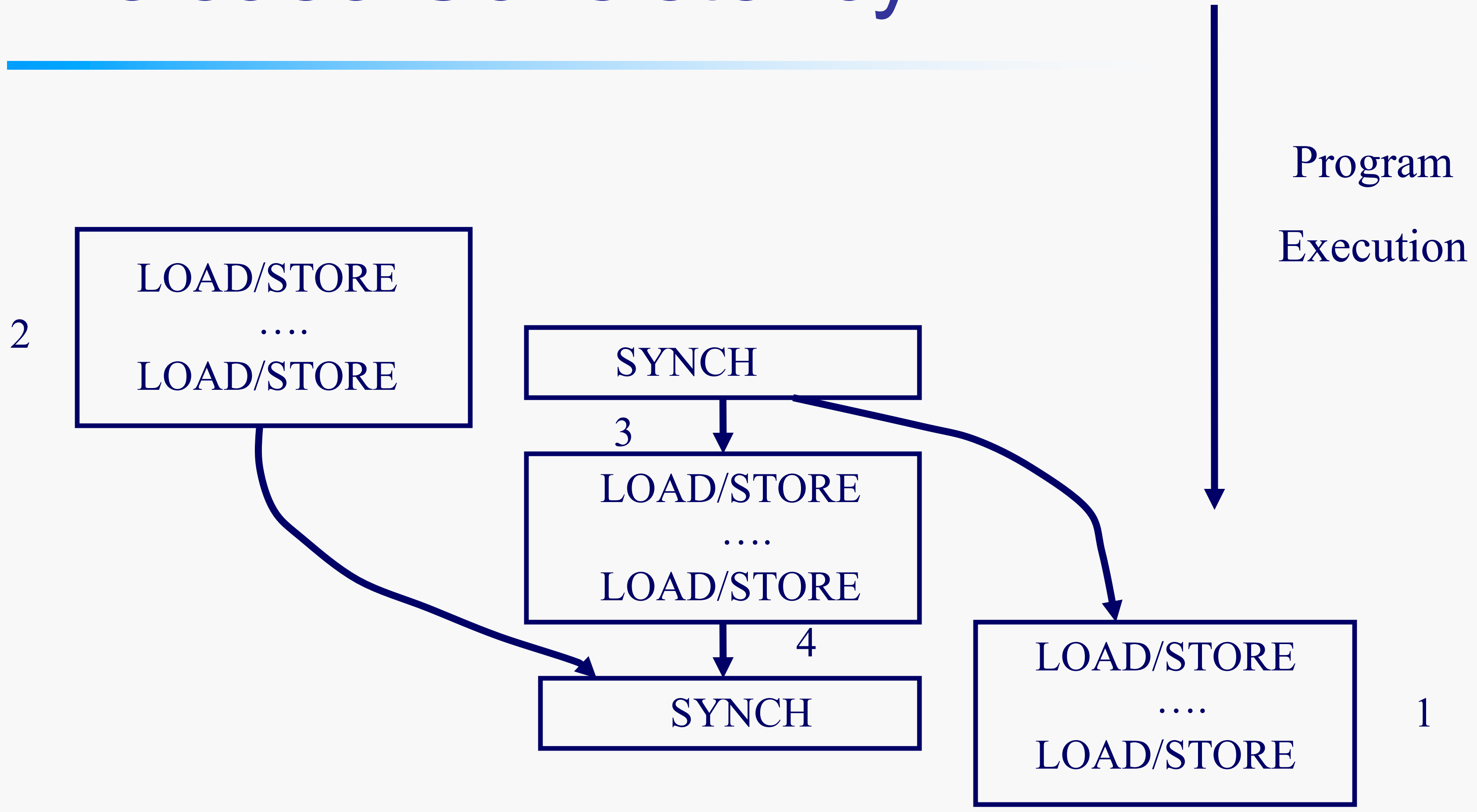
Weak Consistency

- 1. Before an ordinary LOAD/STORE is allowed to perform wrt any processor, all previous SYNCH accesses must be performed wrt everyone

- 2. Before SYNCH access is allowed to perform wrt any processor, all previous ordinary LOAD/STORE accesses must be performed wrt everyone

- SYNCH accesses are sequentially consistent wrt one another

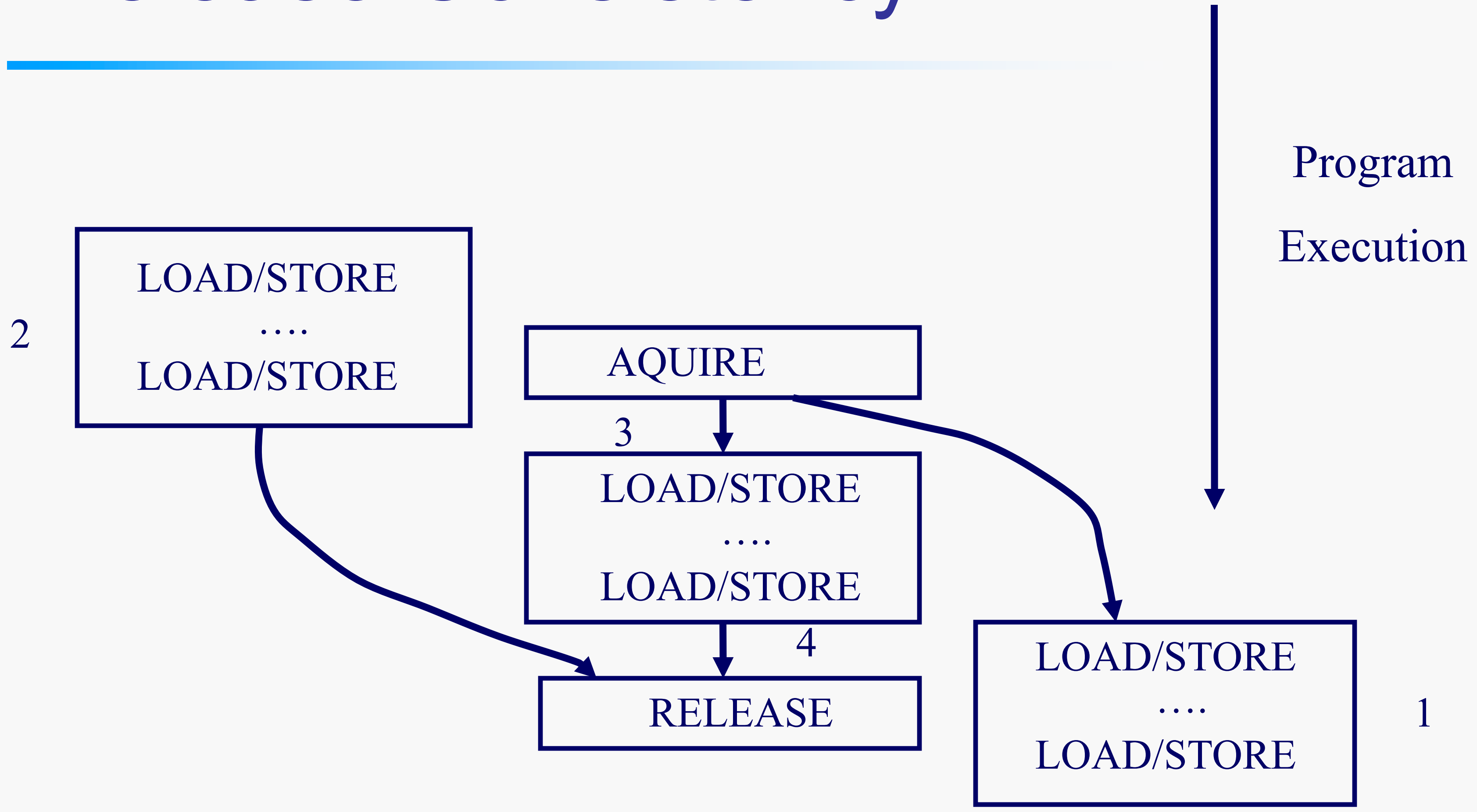
Release Consistency



Release Consistency

- Distinguish between:
  - SYNCH acquires: e.g. LOCK
  - SYNCH releases: e.g. UNLOCK
- LOAD/STORE following a RELEASE do not have to be delayed for the RELEASE to complete
- An ACQUIRE needs not to be delayed for previous LOAD/STORES to complete
- Accesses in the critical section do not wait or delay LOAD/STORES outside the critical section

Release Consistency



Release Consistency

- 3. Before an ordinary LOAD/STORE is allowed to perform wrt any processor, all previous ~~SYNCH~~ ACQUIRE accesses must be performed wrt everyone

- 4. Before S~~YNCH~~ RELEASE access is allowed to perform wrt any processor, all previous ordinary LOAD/STORE accesses must be performed wrt everyone

- ACQ/REL accesses are processor consistent wrt one another

Release Consistency

- Advantages: Higher performance – more parallelism
- Disadvantages: Need to additionally distinguish between ACQUIRE/RELEASE

How to enforce these stalls?

- With Fence instructions
- Different types of fences present in current processors
- Check manuals of processors to see which types of fences are supported

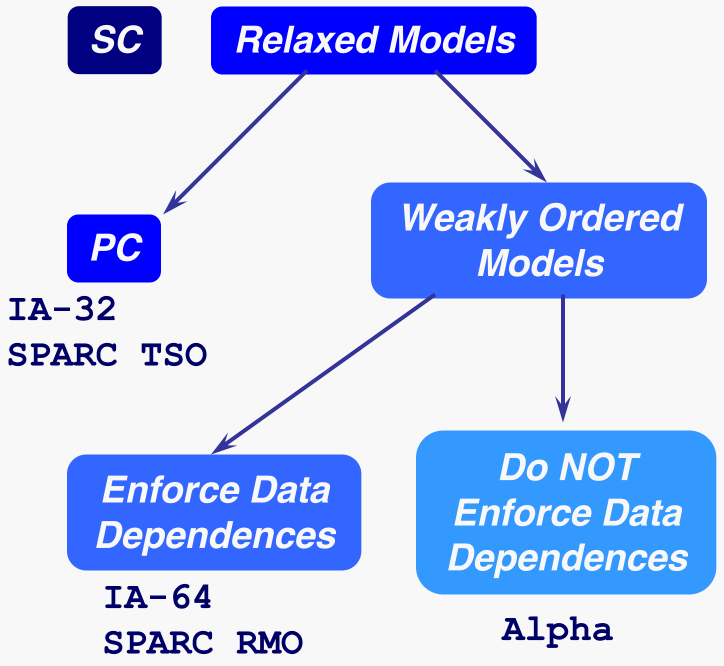
Performance Gains in SW

- Common compiler optimizations require:
  - Change the order of memory operations
  - Eliminate memory operations
- More advanced optimizations such as loop transformation and blocking
- Relaxed models allow compilers to do more re-arrangements

- Examples:
  - Register allocating a flag that is used to synchronize
    ```
    While (flag==0);
    ```
  - Code motion or register allocation across synchronization
    ```
    Lock L
    Read A
    Write B
    Unlock L
    Lock L
    Read A
    Read B
    Unlock L
    ```
- Sequential consistency disallows reordering of shared accesses

Sequential Consistency: Don't assume it !

- Sequential Consistency (SC)
  - **MIPS/SGI**
  - **HP PA-RISC**
- Processor Consistency (PC)
  - *Relax write→read dependencies*
  - **Intel x86 (IA-32)**
  - **Sun TSO (Total Store Order)**
- Relaxed Consistency (RC)
  - *Relax all dependencies, but add fences*
  - **DEC Alpha**
  - **IBM PowerPC**
  - **Intel IPF (IA-64)**
  - **Sun RMO (Relaxed Memory Order)**



Material from: Hill, M. D. 2003. Revisiting "Multiprocessors Should Support Simple Memory Consistency Models" , http://www.cs.wisc.edu/multifacet/papers/dagstuhl03_memory_consistency.ppt

Further Readings & Acknowledgent

- Shared Memory Consistency Models: A Tutorial, S.V. Adve and K. Gharachorloo, *IEEE Computer*, December 1996, 66-76.

- An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors, Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* October 1996, 12-23.

- Culler and Singh course textbook
- Processors have their own memory consisteny models: e.g. SUN's PSO, TSO

- Some slides adapted from Prof. **Josep Torrellas**' (UIUC) course.

Table of Contents

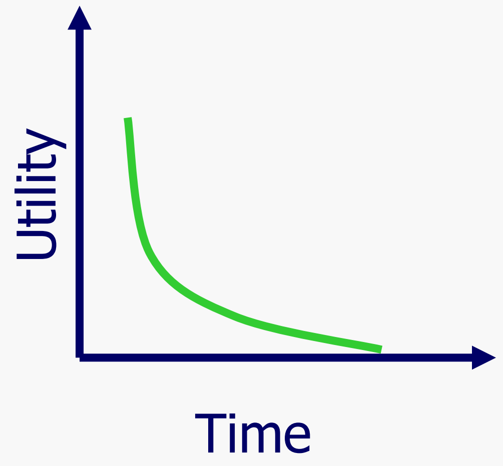
Productivity

- Reduce time to solution
  - Programming time + execution time
- Reduce cost of solution
- Function of:
  - problem solved *P*
  - system used *S*
  - Utility function *U*

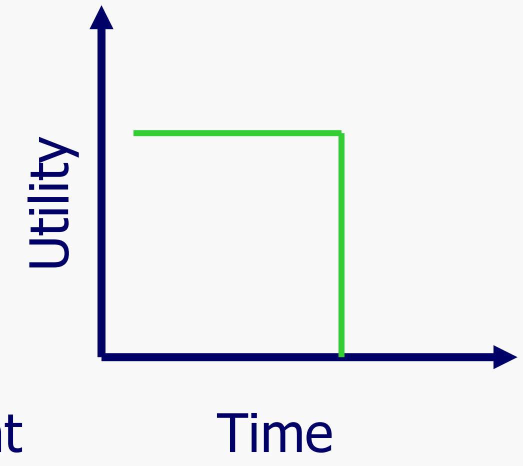
$$\Psi = \Psi(P, S, U)$$

Utility Functions

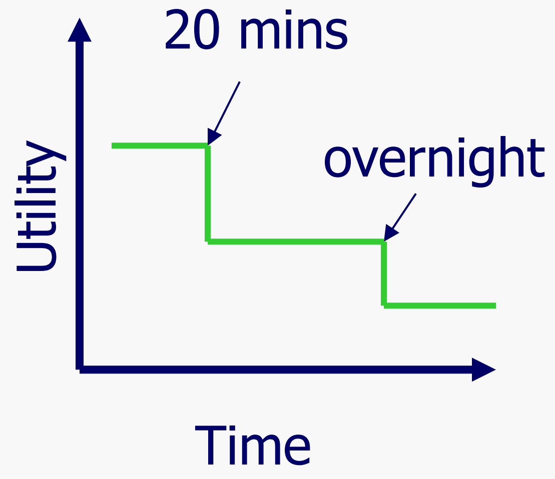
- Decreasing in time.



- Extreme example: deadline driven

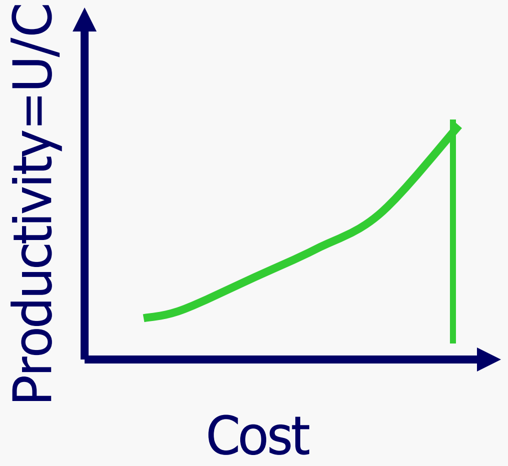


- Practical approximation: staircase



Simple Example

- Assume deadline-driven Utility and decreasing Cost
- Max productivity achieved by solving problem just fast enough to match deadline
- Need to account for uncertainty



Programming Model Impact

- Features try to reduce development time
  - Expressiveness
  - Level of abstraction
  - Component Reuse
  - Expandability
  - Base language
  - Debugging capability
  - Tuning capability
  - Machine model
  - Interoperability with other languages
- Impact on performance examined separately

Expressive

Parasol

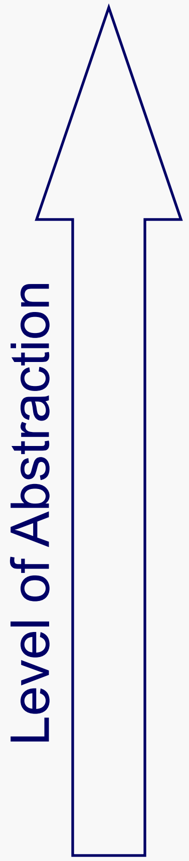
Programming model's ability to express solution in:

- The closest way to the original problem formulation
- A clear, natural, intuitive, and concise way
- In terms of other solved (sub)problems

Definition from http://lml.ls.fi.upm.es/~jjmoreno/expre.html

Level of Abstraction

- Amount of complexity exposed to developer



MATLAB

```
% a and b are matrices
c = a * b;
```

STAPL

```
// a and b are matrices
Matrix<double> c = a * b;
```

C

```
/* a and b are matrices */
double c[10][10];
int i, j, k;
for(int i=0; i<10; ++i) {
  for(int k=0; k<10; ++k) {
    for(int j=0; j<10; ++j) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Component Reuse

- Goal: Increase reuse to reduce development time
- Programming model provides component libraries

STAPL pContainers and pAlgorithms

```
p_vector<double> x(100);
p_vector<double> y(100);

p_generate(x, rand);
p_generate(y, rand);

double result = p_inner_product(x,y);
```

Expandable

- Programming model provides a subset of components needed for a parallel application.

- Expansion enabled by:
  - Transparent components
  - Compositional construction

Component Transparency

Parasol

- Opaque objects hide implementation details
  - raises level of abstraction
  - makes expansion difficult
- Transparent components
  - allow internal component reuse
  - example of working in programming model

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  // …
}
```

```
template<class T>
class p_array : public p_container_indexed<T> {
  typedef p_container_indexed<T> base_type;
  size_t m_size;
  //…
};
```

Component Composition

Parasol

Build a new component using building blocks.

```
template<typename View>
bool p_next_permutation(View& vw) {
  …
  reverse_view<View> rvw(vw);
  iter1 = p_adjacent_find(rvw);
  …
  iter2 = p_find_if(rvw, std::bind1st(pred, *iter1));
  …
  p_reverse(rvw);
  return true;
}
```

Issue: performance of composed blocks….

Programming Language

- Programming model language options:
  - provide a new language
  - extend an existing language
  - provide directives for an existing language
  - use an existing language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n"
  end
end
```

Cilk

```
cilk void hello() {
  printf("Hello, world!\n");
}

int main() {
  spawn hello();
  sync;
}
```

Providing a new language

Parasol

- Advantage
  - Complete control of level of abstraction
  - Parallel constructs embedded in language
- Disadvantage
  - Compiler required for every target platform
  - Developers must learn language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n"
  end
end
```

Extending a language

- Advantage
  - Developers have less to learn
  - Complete control of level of abstraction
  - Parallel constructs embedded in syntax
- Disadvantage
  - Compiler required for every target system
  - Limited by constraints of base language

```
cilk void hello() {
  printf("Hello, world!\n");
}
int main() {
  spawn hello();
  sync;
}
```

Directives for a language

- Advantage
  - Developers have less to learn
  - Parallel constructs easily expressed in directives
  - Use available compilers if needed (no parallelization)
  - Specialized not necessarily needed on system
- Disadvantage
  - Compiler required for every target system
  - Higher levels of abstraction can’t be achieved
  - Limited by constraints of base language
  - No composition

```
#pragma omp parallel for
for(int i=0; i<N; ++i) {
  C[i] = A[i]*B[i];
}
```

Library for a language

- Advantage
  - Developers learn only new API
  - Compilers available on more systems
- Disadvantage
  - Limited by constraints of base language

```
void* hello(void*) {
  printf("Hello, world!\n");
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  pthread_create(&thread, &attr,
   hello, NULL);
}
```

Debuggable

Programming environments provide many options for debugging parallel applications.

Built-in	provides proprietary tools that utilize extra runtime information	Charm++
Tracing	provides hooks for tools to log state during execution	MPI, Charm++
Interoperability with standard tools	Leverage standard tools available on platform (e.g., gdb, totalview)	STAPL, TBB, Pthreads, MPI, OpenMP

Defect Management

- Reduce Defect Potential
  - Programming style reduces likelihood of errors
  - Use of container methods reduces out-of-bounds accesses

```
class tbb_work_function {
  void operator()(const blocked_range<size_t>& r) {
    for(size_t i = r.begin(); i != r.end(); ++i)
      C[i] = A[i]*B[i];
  }
};
```

- Provide Defect Detection
  - Components support options to detect errors at runtime
  - E.g., PTHREAD_MUTEX_ERRORCHECK enables detection of double-locking and unnecessary unlocking

Tunability

Programming environments support application optimization on a platform using:

- Performance Monitoring
  - Support measuring application metrics
- Implementation Refinement
  - Support for adaptive/automatic modification of application
  - Manual mechanisms provided to allow developer to implement refinement

Performance Monitoring

- Built-in support
  - Environment’s components instrumented
  - Output of monitors enabled/disabled by developer
  - Components written by developer can use same instrumentation interfaces
- Interoperable with performance monitoring tools
  - Performance tools on a platform instrument binaries

Implementation Refinement

- Adjust implementation to improve performance
  - distribution of data in a container
  - scheduling of iterations to processors
- Adaptive/Automatic
  - Monitors performance and improves performance without developer intervention
  - Example: Algorithm selection in STAPL
- Manual mechanisms
  - Model provides methods to allow developer adjustment to improve performance
  - Example: Grain size specification to TBB algorithms

Machine Model

- Programming models differ in the amount and type of machine information available to user
  - TBB, Cilk, OpenMP: user unaware of number of threads
  - MPI: user required to write code as a function of the machine in order to manage data mapping
- Programming as a function of the machine
  - Lowers level of abstraction
  - Increases programming complexity

Interoperability with other models

- Projects would like to use multiple models
  - Use best fit for each application module
  - Modules need data from one another
- Models need flexible data placement requirements
  - Avoid copying data between modules
  - Copying is correct, but expensive
- Models need generic interfaces
  - Components can interact if interfaces meet requirements
  - Avoids inheriting complex hierarchy when designing new components

Table of Contents

Performance

- Latency Management
- Load Balancing
- Creating a High Degree of Parallelism

Performance - Memory Wall

Complex memory hierarchies greatly affect parallel execution.

Processing elements may share some components (e.g., L1/L2 caches, RAM), but usually not all.

Parallelism exacerbates the effects of memory latency.

- **Contention** from centralized components.
- **Non uniform latency** caused by distributed components.

Desktop Core2Duo
Private L1 Cache
Shared L2 Cache
Shared Centralized UMA

SGI Origin
Private L1 Cache
Private L2 Cache
Shared, Distributed NUMA

Linux Cluster
Private L1 Cache
Private L2 Cache
Private, Distributed NUMA

Performance - Memory Contention

The extent to which processes access the same location at the same time.

- Types of contention and mitigation approaches.
  - False sharing of cache lines.
    - Memory padding to cache block size.
  - ‘Hot’ memory banks.
    - Better interleaving of data structures on banks.
  - True Sharing.
    - Replication of data structure.
    - Locked refinement (i.e., distribution) for aggregate types.
- Most models do not directly address contention.

Hiding Latency - Runtime System

Runtime system uses extra parallelism made available to transparently hide latency.

e.g., Multithreading **(STAPL / ARMI)**

Communication library overlaps computation with communication (of work)

Performance - Managing Latency

There are two approaches to managing latency.

- Hiding - tolerate latency by overlapping a memory accesses with other computation.
  - User Level
  - Runtime System
- Reducing - minimize latency by having data near the computation that uses it.

Hiding Latency - User Level

Model has programming constructs that allow user to make asynchronous remote requests.

- Split-Phase Execution **(Charm++)**

  Remote requests contain address of return handler.

```
class A {                              class B {
  foo() {                                xyz(Return ret) {
    B b;                                   …
    b.xyz(&A::bar());                      ret(3);
  }                                      }
  bar(int x) { … }                     };
};
```

- Futures

  Remote requests create a handle that is later queried.

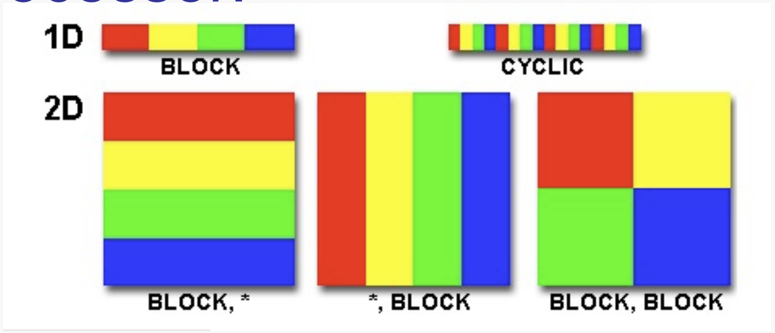
```
future<double> v(foo());    //thread spawned to execute foo()
…                           //do other unrelated work
double result = v.wait();   //get result of foo()
```

Performance - Latency Reduction

Data placement (HPF, STAPL, Chapel)

Use knowledge of algorithm access pattern to place all data for a computation near executing processor.

```
INTEGER, DIMENSION(1:16):: A,B
!HPF$ DISTRIBUTE(BLOCK) :: A
!HPF$ ALIGN WITH A :: B
```



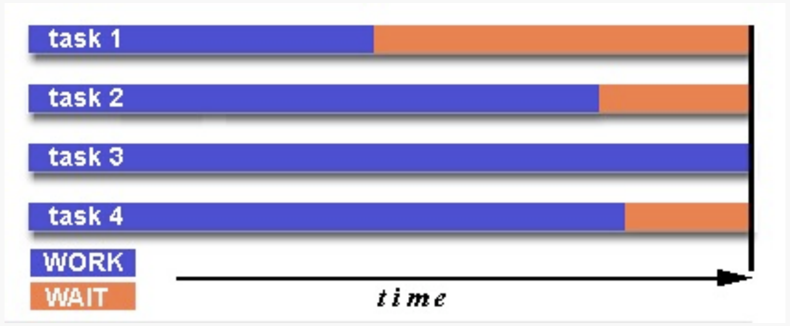
Work placement (STAPL, Charm++)

Migrate computation to processor near data and return final result. Natural in RMI based communication models.

Load Balancing

Keep all CPUs doing equal work.

Relies on good *work scheduling*.



- Static **(MPI)**
  Decide before execution how to distribute work.

- Dynamic **(Cilk, TBB)**
  Adjust work distribution during execution.
  - Requires finer work granularity (> 1 task per CPU)
    Some models change granularity as needed (minimize overhead).
  - Work Stealing
    Allow idle processors to 'steal' queued work from busier processors.

Enabling a High Degree of Parallelism

Parasol

Parallel models must strive for a high degree of parallelism for maximum performance.

Makes transparent latency hiding easy.

Enables finer granularity needed for load balancing.

Table of Contents

Portability

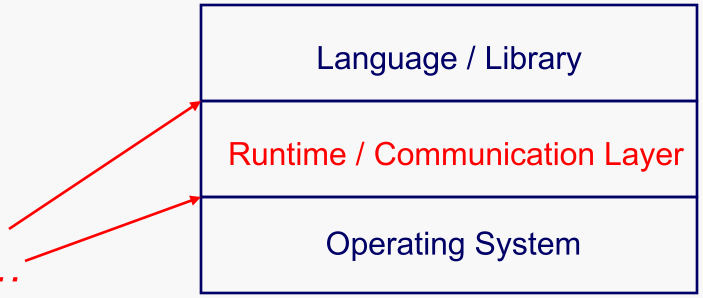
- Language versus Library
- Runtime System
  - Interchangeable
  - Virtualization
  - Load balancing
  - Reliance on specific machine features
- Effects of exposed machine model on portability
- I/O Support

Language versus Library

- Models with specialized language require a compiler to be ported and sometimes additional runtime support.
  - Cray's **Chapel**, **Titanium**, Sun's **Fortress**.
- Library approaches leverage standard toolchains, and often rely on widely available standardized components.
  - **STAPL** requires C++, Boost, and a communication subsystem (MPI, OpenMP Pthreads).
  - **MPI** requires communication layer interfaceand command wrappers (mpirun) to use portable versions (MPICH or LamMPI). Incremental customization can improve performance.

Runtime System

- **Interchangeable**
  Runtime system (e.g., threading and communication management) specific to model or is it modular?
- **Processor Virtualization**
  How are logical processes mapped to processors?
  Is it a 1:1 mapping or multiple processes per processor?

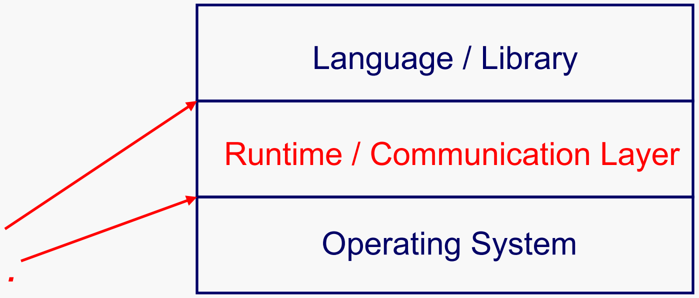


These Lines Often Get Blurred…

Runtime System

- **Load Balancing**
  Support for managing processor work imbalance?
  How is it implemented?

- **Reliance on Machine Features**
  Runtime system require specific hardware support?
  Can it optionally leverage hardware features?



These Lines Often Get Blurred…

Effects of Parallel Model

What effect does the model's level of abstraction have in mapping/porting to a new machine?

– Does it hide the hardware's model (e.g., memory consistency) or inherit some characteristics? Portability implications?

– Is there interface of machine characteristics for programmers? Optional use (i.e., performance tuning) or fundamental to code development?

– *Ideally – Optional user level architectural model*

Support for I/O

Some parallel models specifically address I/O, providing mechanisms that provide an abstract view to various disk subsystems.

ROMIO - *portable I/O extension included with MPI (Message Passing Interface).*

Table of Contents

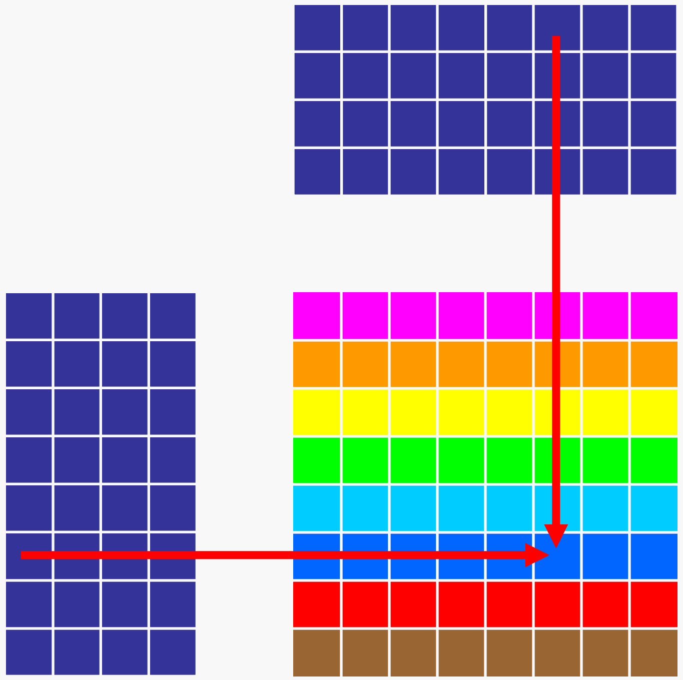
Shared Memory Programming

- Smaller scale parallelism (100’s of CPUs)
- Single system image
- Thread-based
- Threads have access to entire shared memory
  - Threads may also have private memory

Shared Memory Programming

- No explicit communication
  - Threads write/read shared data
  - Mutual exclusion used to ensure data consistency
- Explicit Synchronization
  - Ensure correct access order
  - E.g., don’t read data until it has been written

Example - Matrix Multiply



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

One way to parallelize is to compute each row Independently.

Table of Contents

OpenMP

- Allows explicit parallelization of loops
  - Directives for Fortran and C/C++
  - Limited support for task parallelism

```
#pragma omp parallel for
for(int i=0; i<N; ++i) {
  C[i] = A[i] + B[i];
}
```

- Vendor standard
  - ANSI X3H5 standard in 1994 not adopted
  - OpenMP standard effort started in 1997
  - KAI first to implement new standard

Materials from http://www.llnl.gov/computing/tutorials/openMP/

The OpenMP Model

Execution Model

- Explicitly parallel
- Single-threaded view
- SPMD
- Implicit data distribution
- Nested parallelism support
- Relaxed consistency within parallel sections

The OpenMP Model

Productivity

- Provides directives for existing languages
- Low level of abstraction
- User level tunability
- Composability supported with nesting of critical sections and parallel loops

Performance

- Load balancing
  - optional selection of runtime scheduling policy
- Scalable parallelism
  - Parallelism proportional to data size

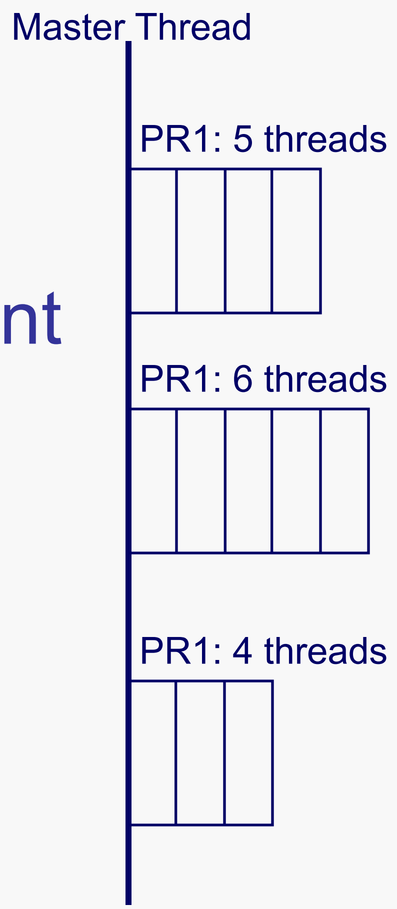
The OpenMP Model

Portability

- Directives allow use of available compilers
  - Application compiles and runs, but no parallelization
- Supports processor virtualization
  - N:1 mapping of logical processes to processors
- Load balancing
  - optional selection of runtime scheduling policy
- No reliance on system features
  - can utilize specialized hardware to implement Atomic update

OpenMP Thread Management

- Fork-Join execution model
- User or developer can specify thread count
  - Developer's specification has priority
  - Variable for each parallel region
  - Runtime system has default value
- Runtime system manages threads
  - User/developer specify thread count only
  - Threads "go away" at end of parallel region



OpenMP Thread Management

- Determining number of threads
  - omp_set_num_threads(int) function
  - OMP_NUM_THREADS environment variable
  - Runtime library default

- Threads created only for parallel sections

Creating Parallel Sections

- Parallel for

```
#pragma omp parallel for
  shared(a,b,c,chunk)
  private(i)
  schedule(static,chunk)
for (i=0; i < n; i++)
  c[i] = a[i] + b[i];
```

- Options
  - Scheduling Policy
  - Data Scope Attributes

- Parallel region

```
#pragma omp parallel
{
  // Code to execute
}
```

- Options
  - Data Scope Attributes

Data Scope Attributes

Private	variables are private to each thread
First Private	variables are private and initialized with value of original object before parallel region
Last Private	variables are private and value from last loop iteration or section is copied to original object
Shared	variables shared by all threads in team
Default	specifies default scope for all variables in parallel region
Reduction	reduction performed on variable at end of parallel region
Copy in	assigns same value to variables declared as thread private

OpenMP Synchronization

- Mutual exclusion by critical sections

```
#pragma omp parallel
{
  // …
  #pragma omp critical
  sum += local_sum
}
```

- Named critical sections
  - unnamed sections treated as one
- Critical section is scoped

- Atomic update

```
#pragma omp parallel
{
  // …
  #pragma omp atomic
  sum += local_sum
}
```

- Specialized critical section
- May enable fast HW implementation
- Applies to following statement

OpenMP Synchronization

- Barrier directive
  - Thread waits until all others reach this point
  - Implicit barrier at end of each parallel region

```
#pragma omp parallel
{
   // …
   #pragma omp barrier
   // …
}
```

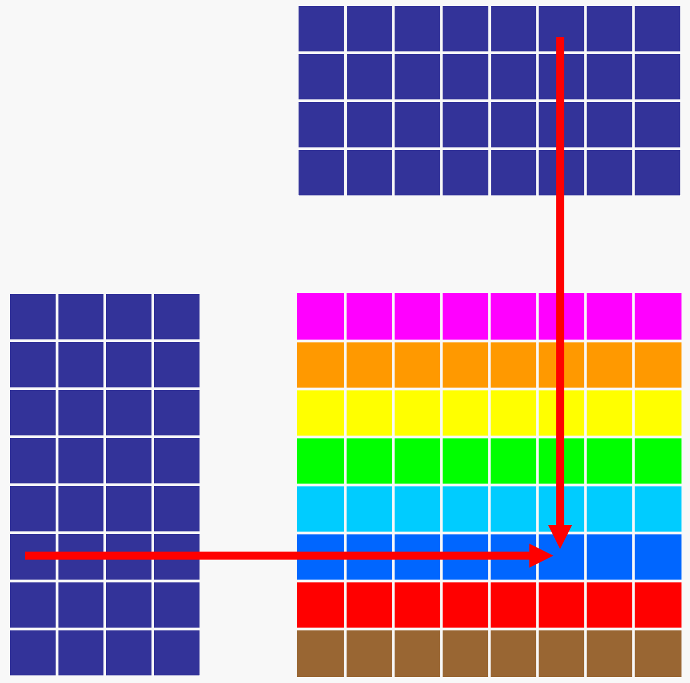
OpenMP Scheduling

- Load balancing handled by runtime scheduler
- Scheduling policy can be set for each parallel loop

Scheduling Policies

Static	Create blocks of size *chunk* and assign to threads before loop begins execution. Default chunk creates equally-sized blocks.
Dynamic	Create blocks of size *chunk* and assign to threads during loop execution. Threads request a new block when finished processing a block. Default chunk is 1.
Guided	Block size is proportional to number of unassigned iterations divided by number of threads. Minimum block size can be set.
Runtime	No block size specified. Runtime system determines iteration assignment during loop execution.

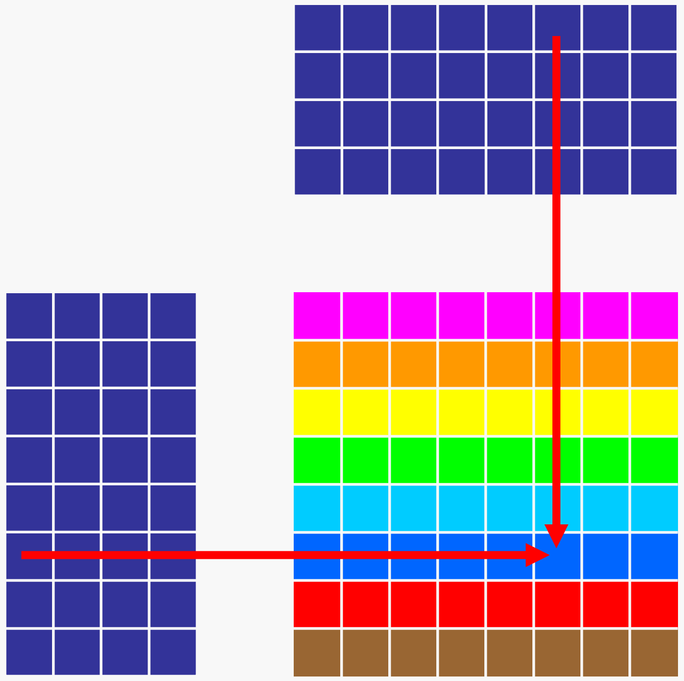
OpenMP Matrix Multiply



```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
   for(int j=0; j<N; ++j) {
      for(int k=0; k<L; ++k) {
         C[i][j] +=
               A[i][k]*B[k][j];
      }
   }
}
```

OpenMP Matrix Multiply

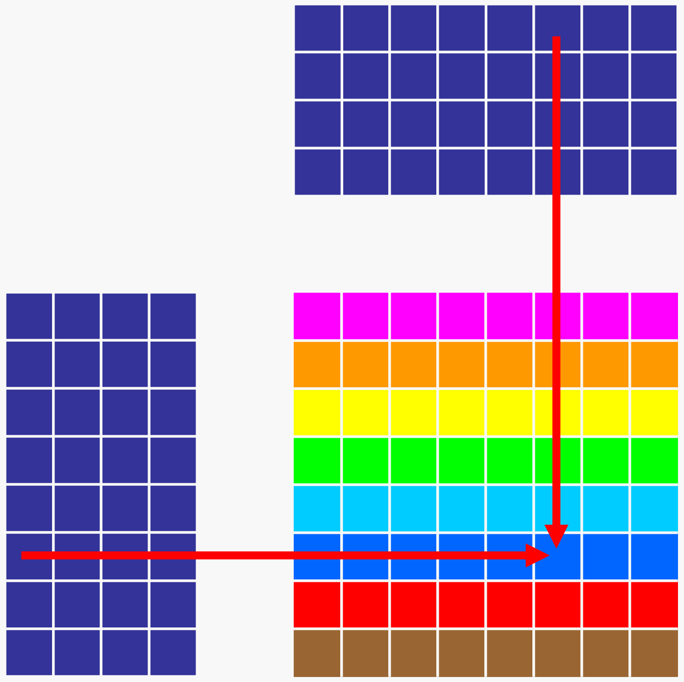
- Parallelizing two loops
  - Uses nested parallelism support
  - Each element of result matrix computed independently



```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
  #pragma omp parallel for
  for(int j=0; j<N; ++j) {
      for(int k=0; k<L; ++k) {
       C[i][j] +=
           A[i][k]*B[k][j];
     }
   }
}
```

OpenMP Matrix Multiply

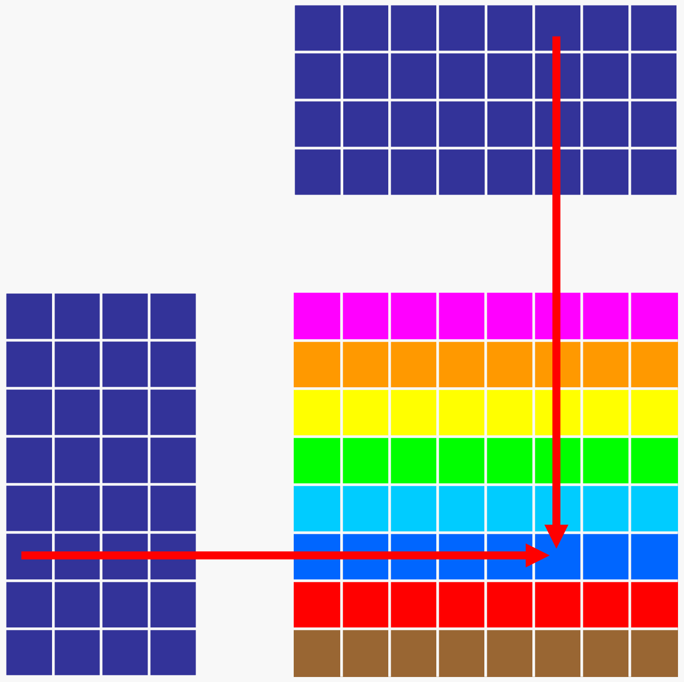
- Parallelizing inner loop
  - Inner loop parallelized instead of outer loop
    - Minimizes work in each parallel loop – for illustration purposes only
  - Multiple threads contribute to each element in result matrix
  - Critical section ensures only one thread updates at a time



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    #pragma omp parallel for
    for(int k=0; k<L; ++k) {
      #pragma omp critical
      C[i][j] +=
            A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

- Use dynamic scheduling of iterations



```
#pragma omp parallel for \
schedule(dynamic)
for(int i=0; i<M; ++i) {
   for(int j=0; j<N; ++j) {
      for(int k=0; k<L; ++k) {
         C[i][j] +=
               A[i][k]*B[k][j];
      }
   }
}
```

Table of Contents

Pthreads

- Specification part of larger IEEE POSIX standard
  - POSIX is the **P**ortable **O**perating **S**ystem **I**nterface
  - Standard C API for threading libraries
    - IBM provides Fortran API
  - Introduced in 1995

- Explicit threading of application
  - User calls functions to create/destroy threads

Materials from http://www.llnl.gov/computing/tutorials/pthreads/

The Pthreads Model

- Execution Model
  - Explicit parallelism
  - Explicit synchronization

- Productivity
  - Not a primary objective
  - Library for existing language
  - Low level of abstraction
  - Uses opaque objects – prevents expansion

The Pthreads Model

- Performance
  - No attempts to manage latency
  - Load balancing left to OS
  - Developer responsible for creating high degree of parallelism by spawning threads
- Portability
  - Library widely available

Pthreads Thread Management

- User creates/terminates threads
- Thread creation
  - `pthread_create`
  - Accepts a single argument (void *)
- Thread termination
  - `pthread_exit`
  - Called from within terminating thread

Pthreads Synchronization

Mutual Exclusion Variables (mutexes)

- Mutexes must be initialized before use
- Attribute object can be initialized to enable error checking

```
pthread_mutex_t mutexsum;
void *dot_product(void *arg) {
  …
  pthread_mutex_lock (&mutexsum);
  sum += mysum;
  pthread_mutex_unlock (&mutexsum);
  …
}
int main() {
  pthread_mutex_init(&mutexsum, NULL);
  …
  pthread_mutex_destroy(&mutexsum);
}
```

Pthreads Synchronization

Condition Variables

- Allows threads to synchronize based on value of data
- Threads avoid continuous polling to check condition
- Always used in conjunction with a mutex
  - Waiting thread(s) obtain mutex then wait
    - pthread_cond_wait() function unlocks mutex
    - mutex locked for thread when it is awakened by signal
  - Signaling thread obtains lock then issues signal
    - pthread_cond_signal() releases mutex

Condition Variable Example

Two threads update a counter

Third thread waits until counter reaches a threshold

```
pthread_mutex_t mtx;
pthread_cond_t cv;

int main() {
…
pthread_mutex_init(&mtx, NULL);
pthread_cond_init (&cv, NULL);
…
pthread_create(&threads[0], &attr,
               inc_count, (void *)&thread_ids[0]);
pthread_create(&threads[1], &attr,
               inc_count, (void *)&thread_ids[1]);
pthread_create(&threads[2], &attr,
               watch_count, (void *)&thread_ids[2]);
…
}
```

Condition Variable Example

Incrementing Threads

```
void *inc_count(void *idp) {
…
for (i=0; i<TCOUNT; ++i) {
  pthread_mutex_lock(&mtx);
  ++count;
  if (count == LIMIT)
    pthread_cond_signal(&cv);
  pthread_mutex_unlock(&mtx);
  …
}
…
}
```

Waiting Thread

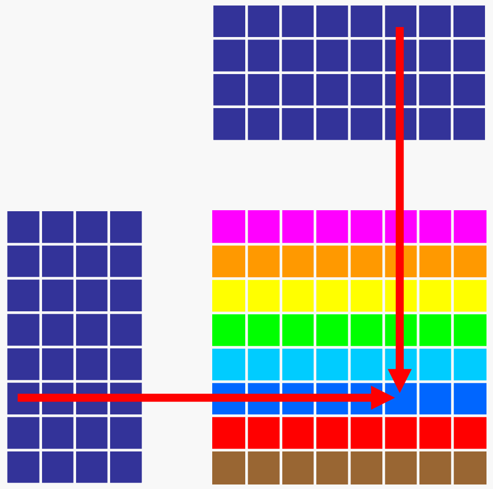
```
void *watch_count(void *idp) {
…
pthread_mutex_lock(&mtx);
while (count < COUNT_LIMIT) {
  pthread_cond_wait(&cv, &mtx);
}
pthread_mutex_unlock(&mtx);
…
}
```

pthread_cond_broadcast() used if multiple threads waiting on signal

Pthreads Matrix Multiply

```
int tids[M];
pthread_t threads[M];
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(
  &attr,
  PTHREAD_CREATE_JOINABLE);

for (i=0; i<M; ++i) {
  tids[i] = i;
  pthread_create(&threads[i],
   &attr, work, (void *) &tids[i]);
}

for (i=0; i<M; ++i) {
  pthread_join(threads[i], NULL);
}
```

```
void* work(void* tid) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[tid][j] +=
        A[tid][k]*B[k][j];
    }
  }
  pthread_exit(NULL);
}
```



References

OpenMP

- http://www.openmp.org
- http://www.llnl.gov/computing/tutorials/openMP

Pthreads

- http://www.llnl.gov/computing/tutorials/pthreads
- "Pthreads Programming". B. Nichols et al. O'Reilly and Associates.
- "Programming With POSIX Threads". D. Butenhof. Addison Wesley

Table of Contents

Message Passing Model

- Large scale parallelism (up to 100k+ CPUs)
- Multiple (possibly heterogeneous) system images
- Distributed memory
  - Nodes can only access local data
  - Application (User) responsible for:
    - Distributing data
    - Redistributing data (when necessary)
    - Maintaining memory coherent

Message Passing Model

- Explicit communication
  - Two-sided P2P:
    - Communication initiated on one side requires matching action on the remote side
    - E.g. MPI_Send – MPI_Recv
  - One-sided P2P:
    - Communication is initiated on one side and no action is required on the other
    - E.g. MPI_Get/Put, gasnet_get/put ...

- Implicit synchronization with two-sided communication
  - The matching of communication operations from both sides ensures synchronization

Message Passing Model

- Objectives of the model
  - Enabling parallelization on highly scalable hardware
  - Support for heterogeneous systems
  - Often coarse-grained parallelism

- Main issues
  - Communication
  - Synchronization
  - Load balancing

Projects of Interest

- Message Passing Interface (MPI)
  - De facto standard for this model
  - Deemed low level and difficult to program
  - Two-sided and one-sided communication

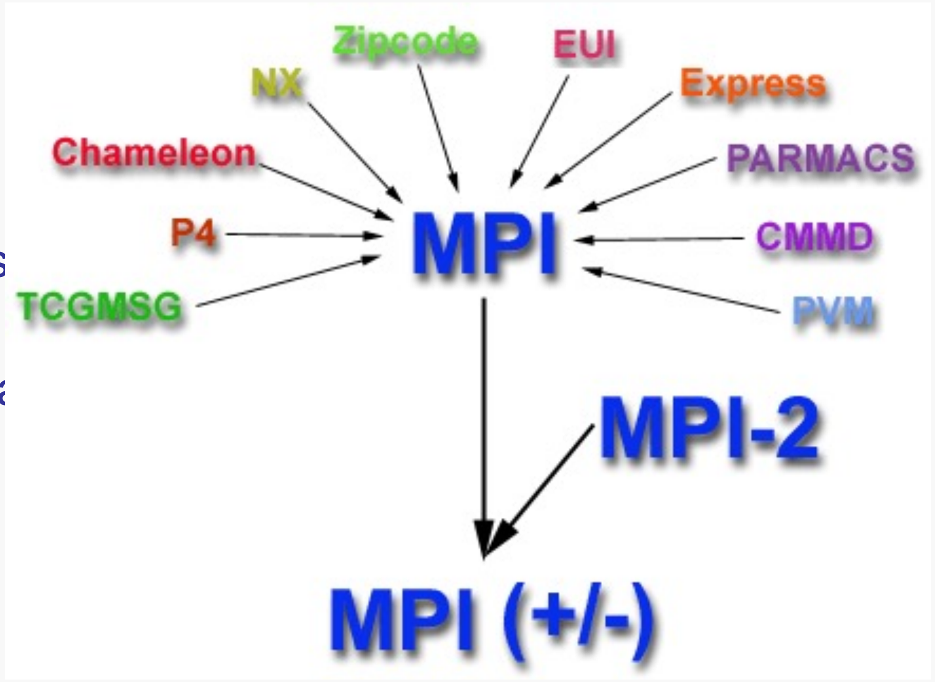
- Charm++
  - Asynchronous Remote Method Invocation (RMI) communication
  - Split-phase programming model
    - No synchronous communication
    - Caller provides a callback handler to asynchronously process “return” value

Table of Contents

Message Passing Interface (MPI)

- 1980s – early 1990s
  - Distributed memory, parallel computing develops
  - Many incompatible software tools
  - Usually tradeoffs between portability, performance, functiona and price

- Recognition of the need for a standard arose.



Material from: http://www.llnl.gov/computing/tutorials/mpi/

Message Passing Interface (MPI)

- Standard based on the consensus of the MPI Forum
  - Not sanctioned by any major standards body
  - Wide practical acceptance
  - No effective alternative to date
- First draft of the MPI-1 standard presented at Supercomputing 1993
- Current standard MPI-2 developed between 1995 and 1997
- Standardization committee open to all members of the HPC community

Further reading and standard documents: http://www.mpi-forum.org/

Message Passing Interface (MPI)

- Objectives
  - High performance and scalability
  - Portability
  - Productivity is not an objective (actually it was)

- Used as communication layer for higher-level libraries
  - Often for more productivity-oriented libraries
  - ISO/OSI layer-5 interface
    - Communication is reliable and sessions are managed internally
    - Data is not structured

MPI: Specification, not Implementation

- Language Independent Specification (LIS)
- Library implementations of MPI vary in:
  - Performance
    - Target or rely on specific hardware (RDMA, PIM, Coprocessors …)
    - Provide load-balancing and processor virtualization (e.g., AMPI)
  - Functionality
    - Support for parallel I/O
    - Support for multithreading within MPI processes

- Standard provides language bindings for Fortran, C and C++
- Implementations usually provide APIs for C, C++ and Fortran
- Project implementations for Python, OCaml, and Java

MPI – Programming Model

Execution Model

- Explicitly parallel
  - Programmer responsible for correctly identifying parallelism and for implementing parallel algorithms using MPI constructs
  - Multi-threaded view
- SPMD
- Explicit data distribution
- Flat parallelism
  - Number of tasks dedicated to run a parallel program is static
- Processor Consistency (one-sided communication)

MPI – Programming Model

Productivity

- Not a principal objective
  - Low level of abstraction
  - Communication is not structured (marshalling done by the user)

Performance

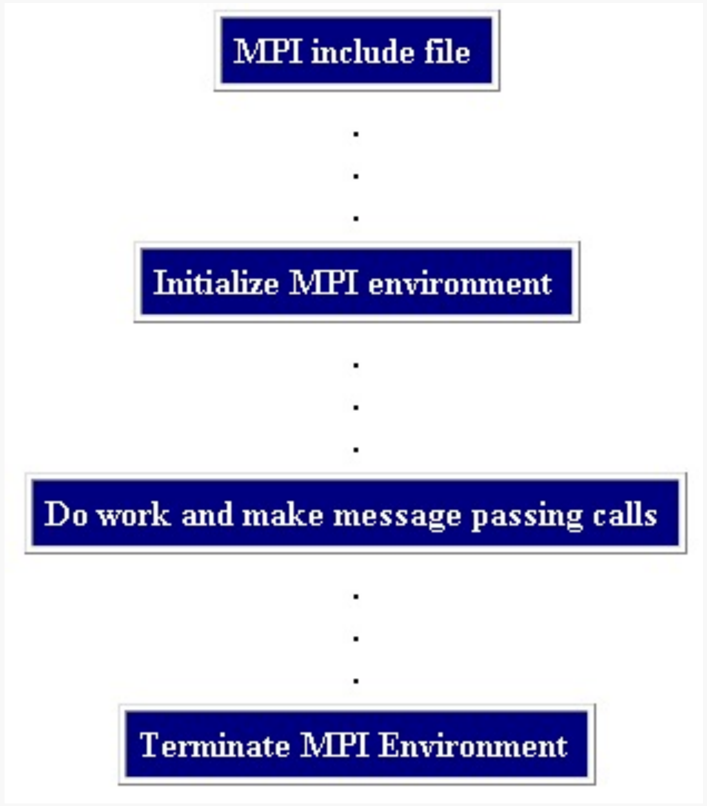
- Vendor implementations exploit native hardware features to optimize performance

Portability

- Most vendors provide an implementation
  - E.g., Specialized open source versions of MPICH, LAM or OpenMPI
- Standard ensures compatibility

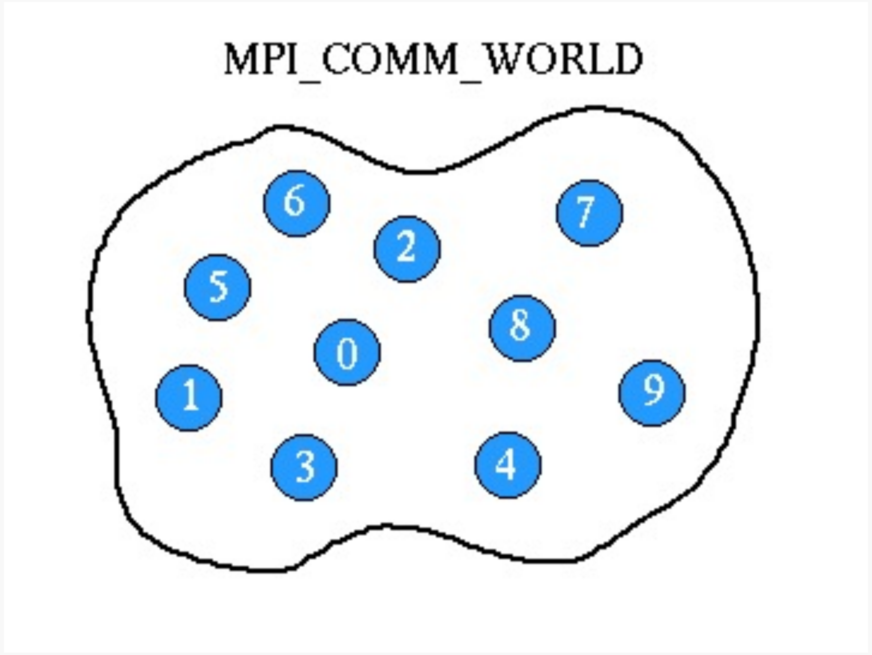
MPI – Program Structure

General program structure


MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Communicators and groups

- Collection of processes that may communicate
- Unique rank (processor ID) within communicator
- Default communicator: MPI_COMM_WORLD



Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Types of Point-to-Point Operations:

- Message passing between two, and only two, different MPI tasks
  - One task performs a send operation
  - The other task matches with a receive operation
- Different types of send/receive routines used for different purposes
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send
- Any type of send can be paired with any type of receive
- Test and Probe routines to check the status of pending operations

MPI – Point to Point Communication

Blocking vs. Non-blocking

- Most routines can be used in either blocking or non-blocking mode
- Blocking communication routines
  - Blocking send routines only return when it is safe to reuse send buffer
    - Modifications to send buffer will not affect data received on the remote side
      - Data already sent
      - Data buffered in a system buffer
  - Blocking send calls can be synchronous
    - Handshaking with the receiver
  - Blocking send calls can be asynchronous
    - System buffer used to hold the data for eventual delivery to the receiver
  - Blocking receive calls only return after the data has arrived and is ready for use by the program

MPI – Point to Point Communication

Blocking communication example

```
#include "mpi.h"
#include <stdio.h>

int main(int argc,char *argv[]) {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
  else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
```

Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Blocking vs. Non-blocking

- Non-blocking communication routines
  - Send and receive routines behave similarly
    - Return almost immediately
    - Do not wait for any communication events to complete
      - Message copying from user memory to system buffer space
      - Actual arrival of message
  - Operations "request" the MPI library to perform an operation
    - Operation is performed when its requirements are met (e.g., message arrives)
    - User cannot predict when that will happen
  - Unsafe to modify the application buffer until completion of operation
    - Wait and Test routines used to determine completion
- Non-blocking communications primarily used to overlap computation with communication and exploit possible performance gains

MPI – Point to Point Communication

Non-blocking communication example

```
MPI_Request reqs[4];
MPI_Status stats[4];

prev = rank-1;
next = rank+1;
if (rank == 0) prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

  {
    // do some work
    // work will overlap with previous communication
  }

MPI_Waitall(4, reqs, stats);
```

MPI – Point to Point Communication

Parasol

Order and Fairness

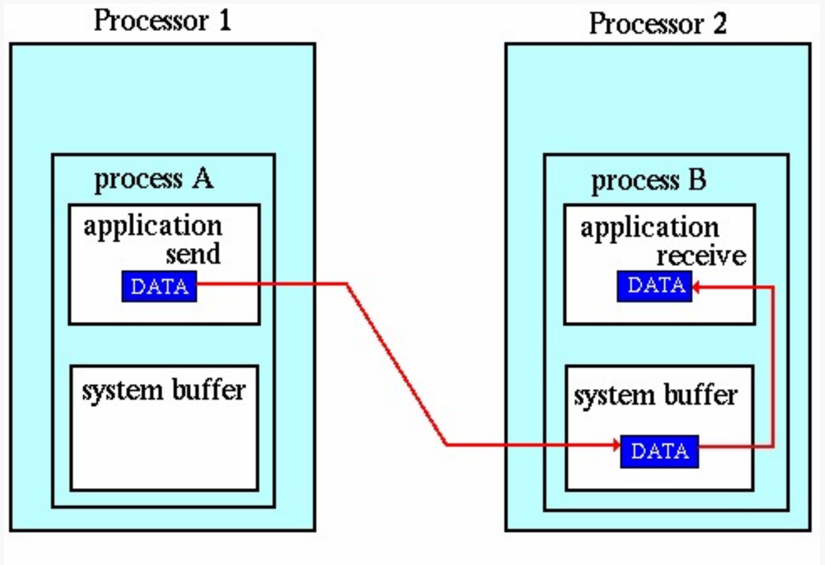
- Message Ordering
  - Messages do not overtake each other
    - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
    - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both match the same message, Receive 1 will receive the message before Receive 2.
  - Ordering is not thread-safe
    - If multiple threads participate in the communication, no order is guaranteed
- Fairness of Message Delivery
  - No fairness guarantee
    - Programmer responsible for preventing operation starvation
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Buffering when tasks are out of sync

- If a receive operation is not ready, sent data is buffered
  - On receiving side, sending side or both
- User can manage buffering memory on sending side



Path of a message buffered at the receiving process

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Collective Communication

- All or None
  - Must involve **all** processes in the scope of the used communicator
  - User responsible to ensure all processes within a communicator participate in any collective operation
- Types of Collective Operations
  - Synchronization (barrier)
    - Processes wait until all members of the group reach the synchronization point
  - Data Movement
    - Broadcast, scatter/gather, all to all
  - Collective Computation (reductions)
    - One member of the group collects data from the other members and performs an operation (e.g., min, max, add, multiply, etc.) on that data

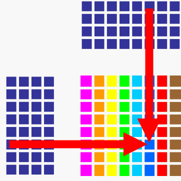
Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Collective Communication

Programming Considerations and Restrictions

- Collective operations are blocking
- Collective communication routines do not take message tag arguments
- Collective operations within subsets of processes
  - Partition the subsets into new groups
  - Attach the new groups to new communicators
- Can only be used with MPI predefined data types
  - Not with MPI Derived Data Types

MPI – Matrix Multiply (master task)



Common to both master and worker processes

- Initialization

```
#define NRA 15     // Number of rows in matrix A
#define NCA 25     // Number of columns in A
#define NCB 10     // Number of columns in B
#define TAG 0      // MPI communication tag
// Data structures
double A[NRA][NCA];// matrix A to be multiplied
double B[NCA][NCB];// matrix B to be multiplied
double C[NRA][NCB];// result matrix C
```

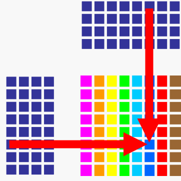
- Distribute data to workers

```
avgNumRows    = NRA/numWorkers;
remainingRows = NRA%numWorkers;
offset        = 0;
for (dest = 1; dest <= numWorkers; ++dest) {
  rows = (dest <= remainingRows) ? avgNumRows + 1 : avgNumRows;
  MPI_Send(&offset, 1, MPI_INT, dest, TAG, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, dest, TAG, MPI_COMM_WORLD);
  count = rows * NCA;
  // Send horizontal slice of A
  MPI_Send(&A[offset][0], count, MPI_DOUBLE, dest, TAG, MPI_COMM_WORLD);
  // Send matrix B
  count = NCA * NCB;
  MPI_Send(&B, count, MPI_DOUBLE, dest, TAG, MPI_COMM_WORLD);
  offset += rows;
}
```

- Wait for results from workers

```
for (i = 1; i <= numworkers; ++i)       {
  source = i;
  MPI_Recv(&offset, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
  count = rows * NCB;
  MPI_Recv(&C[offset][0], count, MPI_DOUBLE, source, TAG,MPI_COMM_WORLD,&status);
}
```

MPI – Matrix Multiply (worker task)



- **Receive data from master**

```
source = 0;
MPI_Recv(&offset, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
MPI_Recv(&rows,   1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
// Receive horizontal slice of A
count = rows * NCA;
MPI_Recv(&A, count, MPI_DOUBLE, source, TAG, MPI_COMM_WORLD, &status);
// Receive matrix B
count = NCA * NCB;
MPI_Recv(&B, count, MPI_DOUBLE, source, TAG, MPI_COMM_WORLD, &status);
```

- **Process data**

```
// Compute the usual matrix multiplication on the slice of matrix A and matrix B
for (k = 0; k < NCB; ++k) {
  for (i = 0; i < rows; ++i) {
    C[i][k] = 0.0;
    for (j = 0; j < NCA; ++j) {
      C[i][k] += A[i][j] * B[j][k];
    }
  }
}
```

- **Send results back to master**

```
destination = 0;
MPI_Send(&offset, 1, MPI_INT, destination, TAG, MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, destination, TAG, MPI_COMM_WORLD);
count = rows * NCB;
// Send horizontal slice of result matrix C computed on this node
MPI_Send(&C, count, MPI_DOUBLE, destination, TAG, MPI_COMM_WORLD);
```

Table of Contents

Charm++

- C++ library for dynamic multithreaded applications
- Developed since 1993
  - Prequel Chare Kernel developed since 1988
- Parallel Programming Laboratory at University of Illinois at Urbana-Champaign
- Prof. Laxmikant V. Kale

Charm++ – Programming Model

Execution Model

- Implicit parallelism
  - Parallelism expressed at the task level (Chare)
  - User unaware of concurrency
- Explicit communication
  - Exclusively through asynchronous RMI (on Chare entry methods)
  - User responsible for implementing packing/unpacking methods
- MPMD
- Message-driven execution
- Dynamic parallelism
  - Every task is a thread
  - Load-balancing with task migration
- Object Consistency model

Charm++ – Programming Model

Productivity

- Charmdebug graphical parallel debugger
- Graphical load balance monitor
- Relatively high level of abstraction

Performance

- Split-phase communication tolerates latency
- Static and dynamic load-balancing
- Processor virtualization

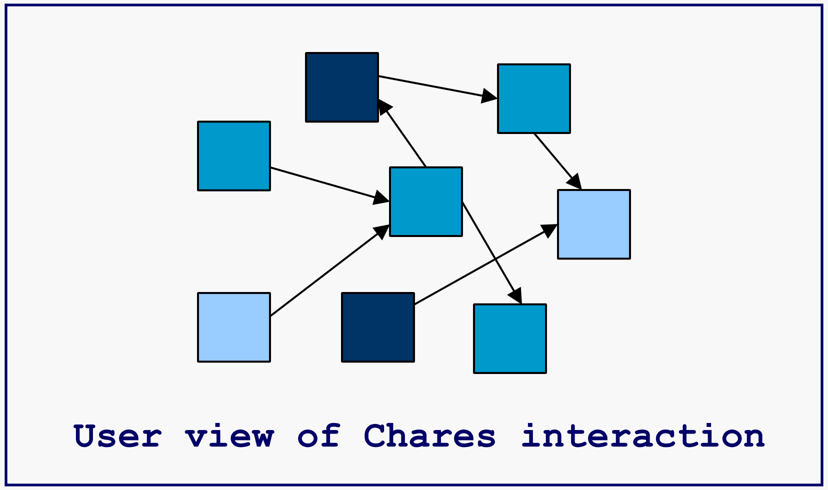
Portability

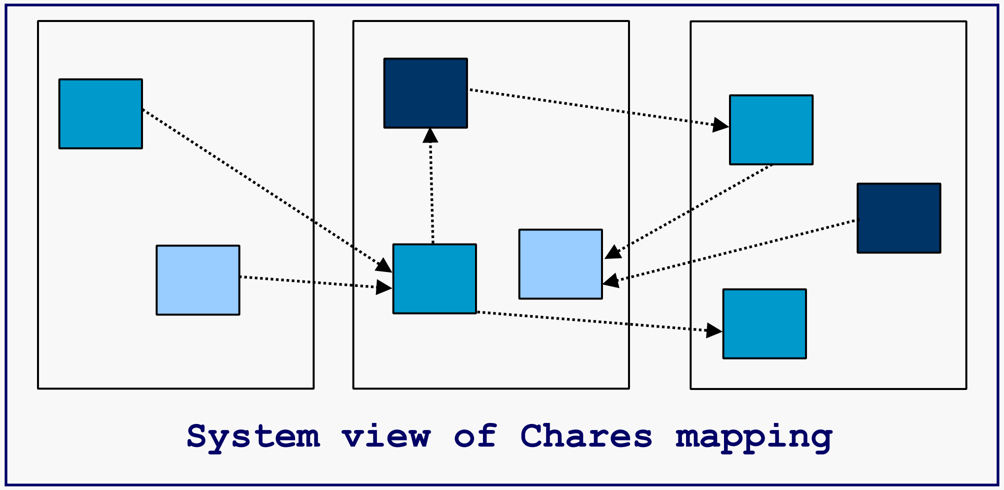
- Library implemented on top of MPI

Charm++ – Virtualization

Object-based decomposition

- Divide the computation into a large number of pieces
  - Independent of the number of processors
  - Preferably significantly larger than the number of processors
- Let the system map objects to processors


User view of Chares interaction


System view of Chares mapping

Charm++ – Chares

- Dynamically created on any available processor
- Can be accessed from other processors
  - Chare_ID instead of Thread_ID (virtualization)
- Send messages to each other asynchronously
- Contain entry methods that can be invoked from other Chares

Charm++ – Chares

- User only required to think of the interaction between chares
- Message-driven execution
  - New Chares are only created as “Seed messages”
  - Construction happens when a first message reaches the new Chare

Charm++ – "Hello World"

HelloWorld.ci

```
mainmodule hello {
  mainchare mymain {
    entry mymain (CkArgMsg *m);
  };
};
```

Charmc

Generates:

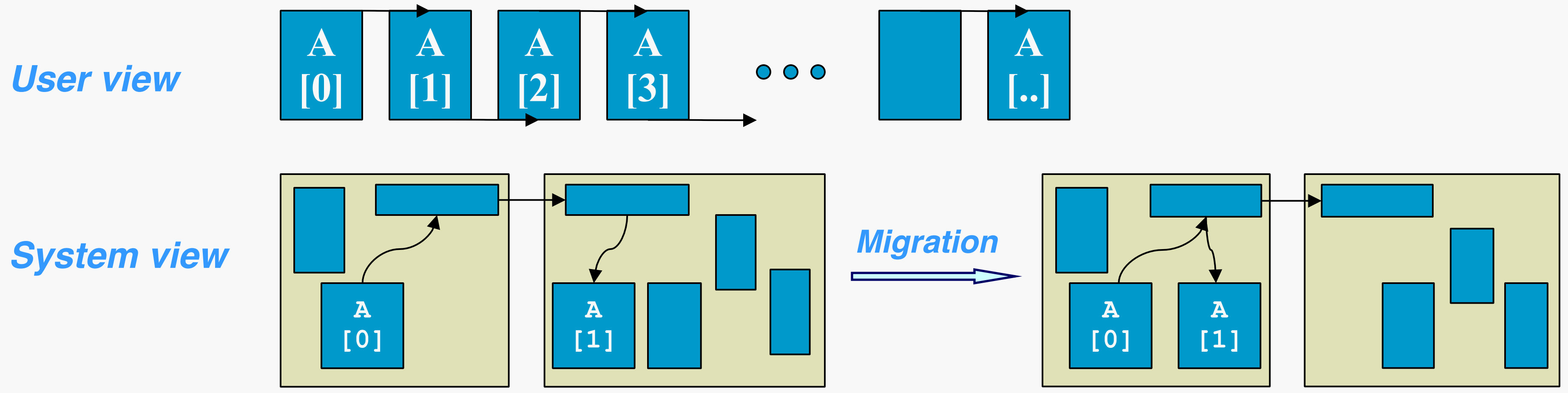
- ***HelloWorld.decl.h***
- ***HelloWorld.def.h***

HelloWorld.C

```
#include "HelloWorld.decl.h"

class mymain : public Chare {
public:

   mymain(CkArgMsg *m) {
     ckout << "Hello world !" << endl;
     CkExit();
   }

};
```

Charm++ – Chare Arrays

- Array of Chare objects
  - Each Chare communicates with the next one
  - More structured view than individual chares
- Single global name for the collection
- Members addressed by index
- Mapping to processors handled by the system



Charm++ – Dynamic Load-Balancing

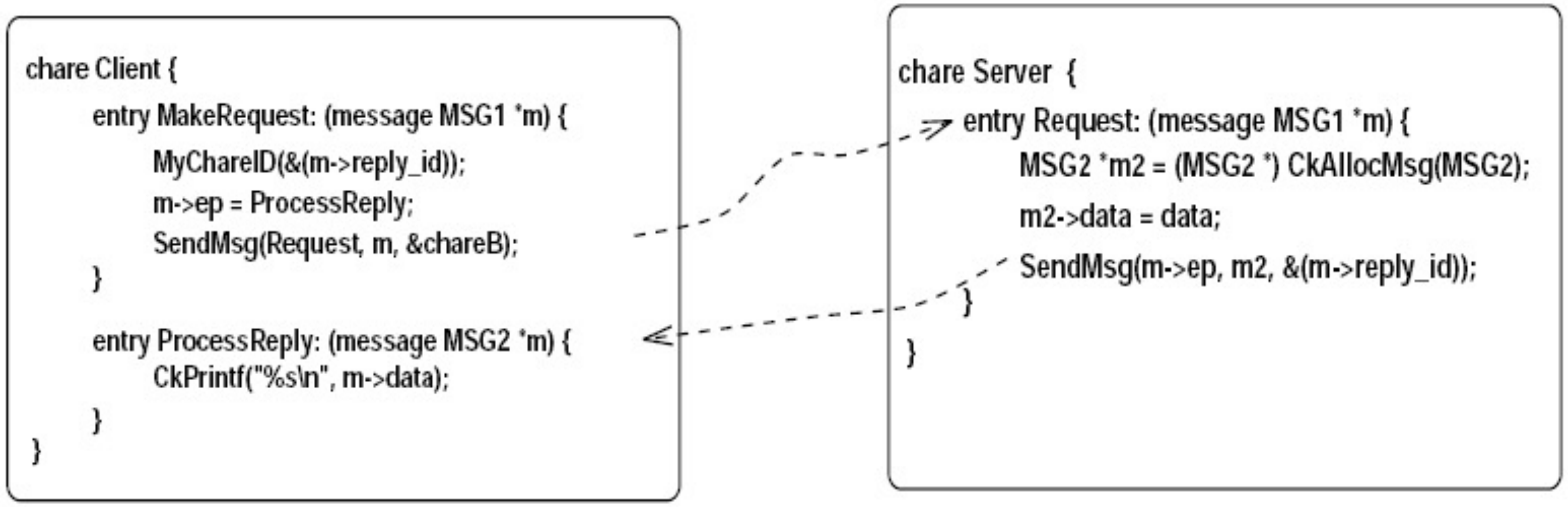
- Object (Chare) migration
  - Array Chares can migrate from one processor to another
  - Migration creates a new object on the destination processor and destroys the original
  - Objects must define pack/unpack (PUP) methods
- Initial load-balancing
  - New Chares created on least loaded processors

Charm++ – Dynamic Load-Balancing

- Centralized load-balancing
  - High-quality balancing with global information
  - High communication cost and latency
- Distributed load-balancing
  - Same principle in small neighborhoods
  - Lower communication cost
  - Global load-imbalance may not be addressed

Charm++ – Split-phase Communication

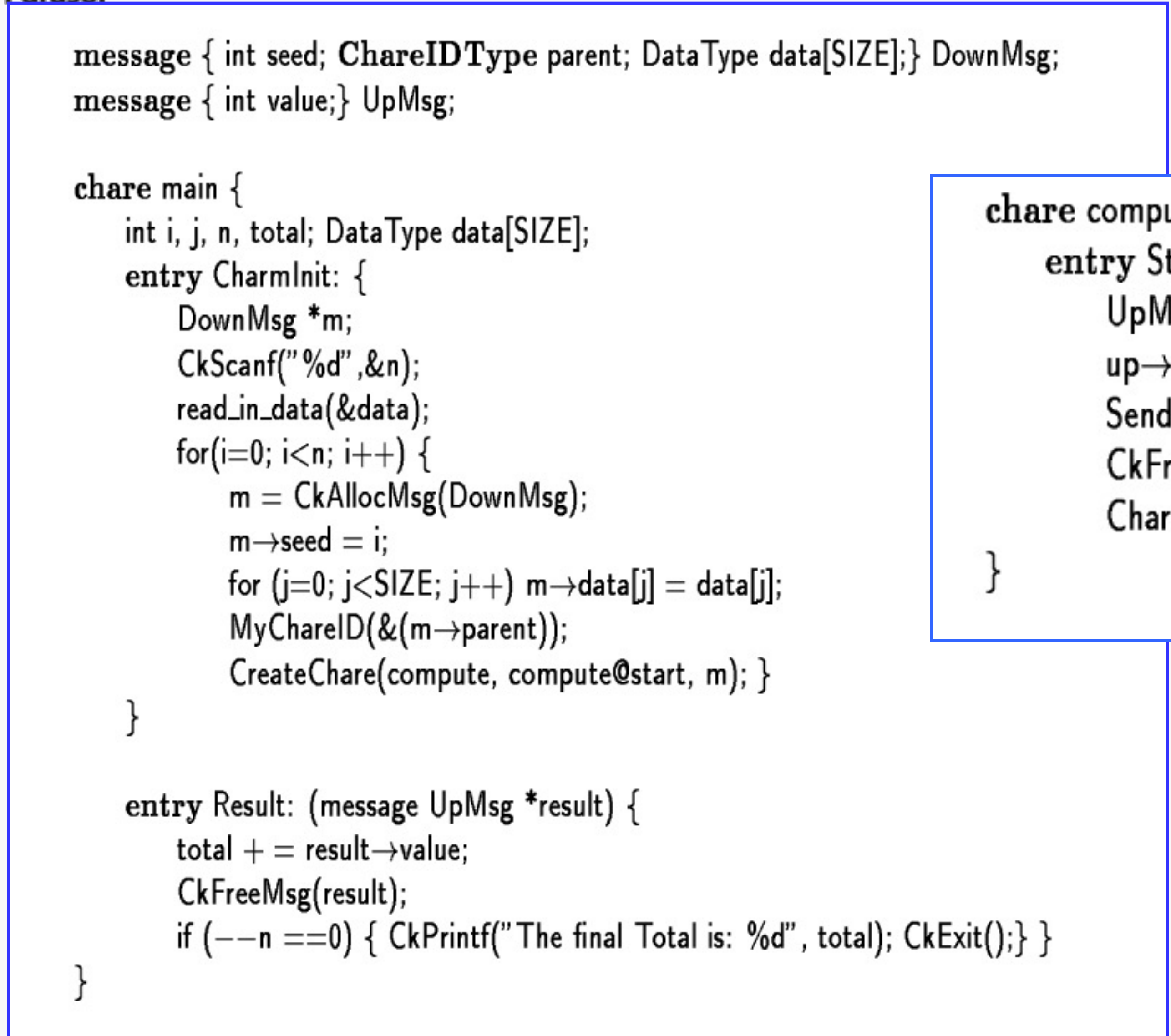
- Asynchronous communication
  - Sender does not block or wait for a return
  - Sender provides callback handler that will process any return value
- Efficient for tolerating latency
  - No explicit waiting for data
  - No stalls with sufficient parallelism



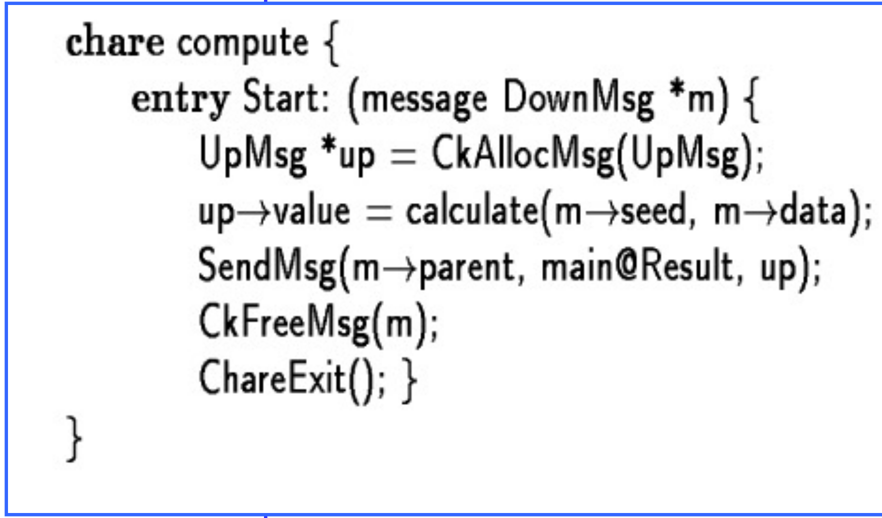
```
chare Client {
        entry MakeRequest: (message MSG1 *m) {
                MyChareID(&(m->reply_id));
                m->ep = ProcessReply;
                SendMsg(Request, m, &chareB);
        }

        entry ProcessReply: (message MSG2 *m) {
                CkPrintf("%s\n", m->data);
        }
}
```

```
chare Server {
        entry Request: (message MSG1 *m) {
                MSG2 *m2 = (MSG2 *) CkAllocMsg(MSG2);
                m2->data = data;
                SendMsg(m->ep, m2, &(m->reply_id));
        }
}
```

Charm++



```
message { int seed; ChareIDType parent; DataType data[SIZE];} DownMsg;
message { int value;} UpMsg;

chare main {
    int i, j, n, total; DataType data[SIZE];
    entry CharmInit: {
        DownMsg *m;
        CkScanf("%d",&n);
        read_in_data(&data);
        for(i=0; i<n; i++) {
            m = CkAllocMsg(DownMsg);
            m→seed = i;
            for (j=0; j<SIZE; j++) m→data[j] = data[j];
            MyChareID(&(m→parent));
            CreateChare(compute, compute@start, m); }
    }

    entry Result: (message UpMsg *result) {
        total + = result→value;
        CkFreeMsg(result);
        if (−−n ==0) { CkPrintf("The final Total is: %d", total); CkExit();} }
}
```



```
chare compute {
    entry Start: (message DownMsg *m) {
        UpMsg *up = CkAllocMsg(UpMsg);
        up→value = calculate(m→seed, m→data);
        SendMsg(m→parent, main@Result, up);
        CkFreeMsg(m);
        ChareExit(); }
}
```

References

- MPI
  - http://www.llnl.gov/computing/tutorials/mpi/
  - http://www.mpi-forum.org/
- Charm++
  - http://charm.cs.uiuc.edu/research/charm/
  - http://charm.cs.uiuc.edu/papers/CharmSys1TPDS94.shtml
  - http://charm.cs.uiuc.edu/papers/CharmSys2TPDS94.shtml
  - http://charm.cs.uiuc.edu/manuals/html/charm++/
  - https://agora.cs.uiuc.edu/download/attachments/13044/03_14charmTutorial.ppt
  - http://charm.cs.uiuc.edu/workshops/charmWorkshop2005/slides2005/charm2005_tutorial_charmBasic.ppt

Table of Contents

Cilk

- Language for dynamic multithreaded applications
- Superset of C
- Developed since 1994
- Supercomputing Technologies Group at MIT Laboratory for Computer Science
- Prof. Charles E. Leiserson

Materials from Charles Leiserson, “Multithreaded Programming in Cilk”, http://supertech.csail.mit.edu/cilk/ . Used with permission.

Cilk extends C

- C elision
  - Removal of Cilk keywords produces valid sequential C program
  - A valid implementation of the semantics of a Cilk program

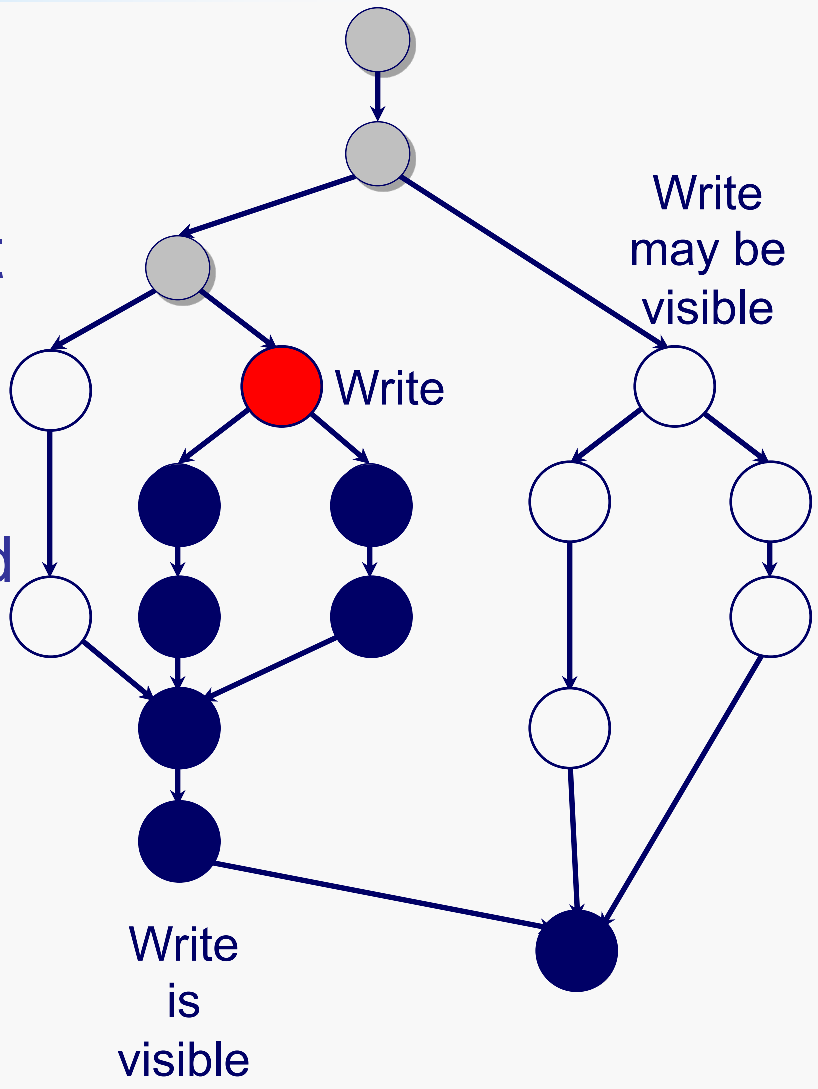
```
cilk int fib (int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

The Cilk Model

- Execution Model
  - DAG consistency model
  - Explicit Parallelism
  - Explicit Synchronization
- Productivity
  - Simple extension of an existing language
  - No details of machine available to application
  - Low level of abstraction
  - No component reuse or language expansion possible
  - Debug and tune using standard tools

DAG consistency

- Vertices are tasks
- Edges are data dependencies
- Read operation can see result of write operation if:
  - there is a serial execution order of the tasks consistent with the DAG where the read is executed after the write
- Successors of a task guaranteed to see write
- Other tasks may or may not see the write



The Cilk Model

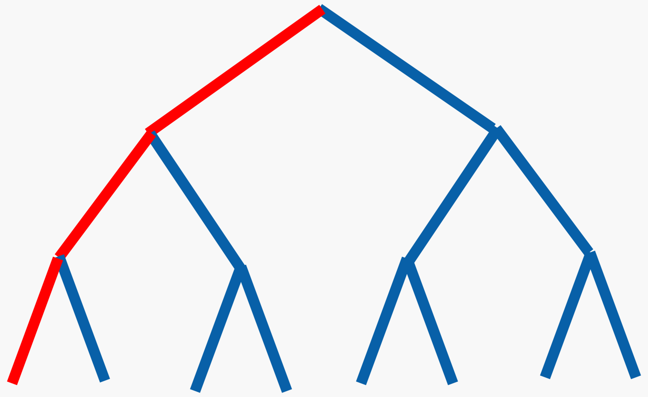
- Performance
  - Developer easily generates high degree of parallelism
  - Work stealing runtime scheduler provides load balance
- Portability
  - Source-to-source compiler provided
  - Runtime system must be ported to new platforms
  - Applications completely unaware of underlying system

Cilk Thread Management

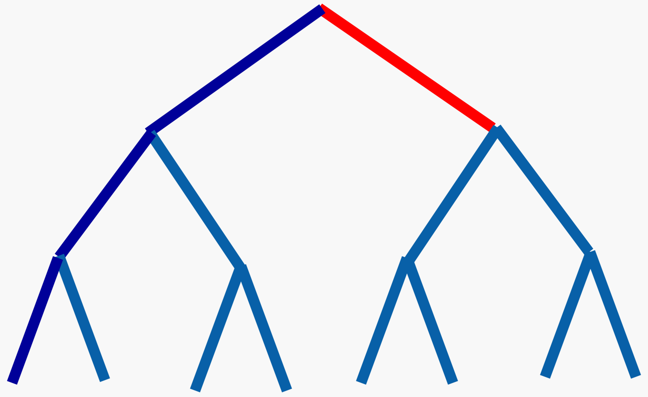
- Application completely unaware of threads
  - Work split into Cilk threads
    - Cilk thread is a task assigned to a processor
    - Tasks scheduled to run on processors by runtime system
    - “Spawn” of Cilk thread is 3-4 times more expensive than C function call
  - Runtime system employs work stealing scheduler

Work Stealing Task Scheduler

- Each processor maintains a deque of tasks
  - Used as a stack
  - Small space usage
  - Excellent cache reuse



- Processor steals when nothing remains in deque
  - Chooses random victim
  - Treats victim deque as queue
  - Task stolen is usually large

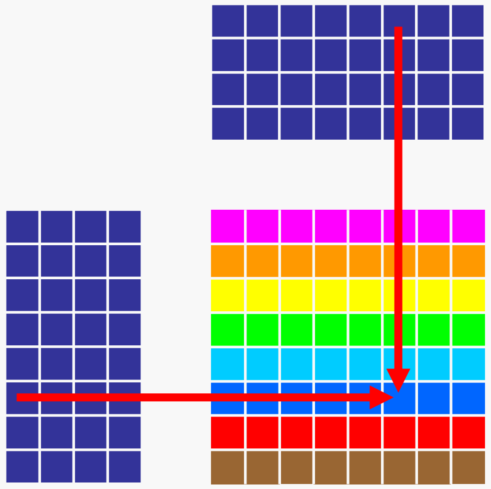


Cilk Synchronization

- Cilk_fence()
  - All memory operations of a processor are committed before next instruction is executed.

- Cilk_lockvar variables provide mutual exclusion
  - Cilk_lock attempts to lock and blocks if unsuccessful
  - Cilk_unlock releases lock
  - Locks must be initialized by calling Cilk_lock_init()

Cilk Matrix Multiply



```
cilk void work(*A, *B, *C, i, L, N) {
   for(int j=0; j<N; ++j) {
      for(int k=0; k<L; ++k) {
         C[i][j] +=
               A[i][k]*B[k][j];
      }
   }
}

void matmul(*A, *B, *C, M, L, N) {
   for(int i=0; i<M; ++i) {
      spawn work(A, B, C, i, L, N);
   }
   sync;
}
```

Cilk Recursive Matrix Multiply

Divide and conquer —

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $(n/2) \times (n/2)$ matrices.
1 addition of $n \times n$ matrices.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
   float *T = Cilk_alloca(n*n*sizeof(float));
   h base case & partition matrice  i
   spawn Mult(C11,A11,B11 n/2);
   spawn Mult(C12,A11,B12   /2);
   spawn Mult(C22,A21,B12,   2);
   spawn Mult(C21,A21,B11,    );
   spawn Mult(T11,A12,B21,n     ;
   spawn Mult(T12,A12,B22,n,
   spawn Mult(T22,A22,B22,n/
   spawn Mult(T21,A22,B21,n/
   sync;
   spawn Add(C,T,n);
   sync;
   return;
}
```

$C = A \square B$

Absence of type declarations.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  ⟨ base ca  & partition matrices ⟩
  spawn Mu  t(C11,A11,B11,n/2);
  spawn Mult C12,A11,B12,n/2);
  spawn Mult(C 2,A21,B12,n/2);
  spawn Mult(C2  A21,B11,n/2);
  spawn Mult(T11, 2,B21,n/2);
  spawn Mult(T12,A  B22,n/2);
  spawn Mult(T22,A2   2,n/2);
  spawn Mult(T21,A22,  n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

Coarsen base cases for efficiency.

$C = A \cdot B$

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base case & partition matrices i
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B  n/2);
  spawn Mult(C21,A21,B1   /2);
  spawn Mult(T11,A12,B21,
  spawn Mult(T12,A12,B22,n
  spawn Mult(T22,A22,B22,n/
  spawn Mult(T21,A22,B21,n/2
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A\cdot B$

Also need a row-size argument for array indexing.

Submatrices are produced by pointer calculation, not copying of elements.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base case & partition matrices i
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B12,n/2);
  spawn Mult(C21,A21,B11,n/2);
  spawn Mult(T11,A12,B21,n/2);
  spawn Mult(T12,A12,B22,n/2);
  spawn Mult(T22,A22,B22,n/2);
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A \times B$

$C = C + T$

```
cilk void Add(*C, *T, n) {
  h base case & partition matrices i
  spawn Add(C11,T11,n/2);
  spawn Add(C12,T12,n/2);
  spawn Add(C21,T21,n/2);
  spawn Add(C22,T22,n/2);
  sync;
  return;
}
```

Table of Contents

Threading Building Blocks

- C++ library for parallel programming
- STL-like interface for library components
  - **Algorithms** accept **Ranges** that provide access to **Containers**
- Initial release by Intel in August 2006
- Strongly influenced by Cilk, STAPL, and others

Intel® Threading Building Blocks

Generic Parallel Algorithms
parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task Scheduler

Low-Level Synchronization Primitives
atomic
spin_mutex
queuing_mutex
reader_writer_mutex
mutex

Memory Allocation
cache_aligned_allocator

Timing
tick_count

The TBB Model

- Execution Model
  - Implicit parallelism
  - Mixed synchronization
    - Locks provided for mutual exclusion
    - Containers provide safe concurrent access
- Productivity
  - Library for an existing language
    - Provides components for reuse
  - Few details of machine available to developer
  - Higher level of abstraction
  - Timing class provided in library for manual tuning
  - Designed to be interoperable with OpenMP and Pthreads

The TBB Model

- Performance
  - Algorithms attempt to generate high degree of parallelism
  - Same work stealing algorithm as Cilk for load balance
- Portability
  - Library implementation must be ported to new platforms
  - Currently requires x86 architecture

TBB Thread Management

- Developer mostly unaware of threads
  - Can specify the desired thread count at TBB initialization
  - Runtime system defaults to single thread per processor
- Developer creates tasks instead of threads
  - Tasks mapped to threads by runtime scheduler as in Cilk
  - TBB algorithms attempt to generate many tasks
- TBB runtime system handles management of threads used to process tasks

TBB Synchronization

Task synchronization

- Tasks are logical units of computation
- Tasks dynamically create new tasks
  - Split-join model applied to child tasks
  - Parent task may specify a task to be executed when all child tasks complete (explicit continuation)
  - Parent task may block and wait on children to complete before it finishes (implicit continuation)
    - Cilk threads use this model
- TBB algorithms generate and manage tasks
  - Use continuations to implement execution pattern

TBB Synchronization

Concurrent Containers

- Allow threads to access data concurrently
- Whole-container methods
  - Modify entire container
  - Must be executed by a single task
- Element access methods
  - Multiple tasks may perform element access/modification
  - Containers use mutexes as needed to guarantee consistency

TBB Synchronization

Low-level Synchronization Primitives

- Atomic template class provides atomic operations
  - Type must be integral or pointer
  - read, write, fetch-and-add, fetch-and-store, compare-and-swap operations provided by class
- Mutexes use scoped locking pattern
  - lock released when variable leaves scope
  - initialization of variable is lock acquisition

```
{
// myLock constructor acquires lock on myMutex
M::scoped_lock myLock( myMutex );
... actions to be performed while holding the lock ...
// myLock destructor releases lock on myMutex
}
```

TBB Synchronization

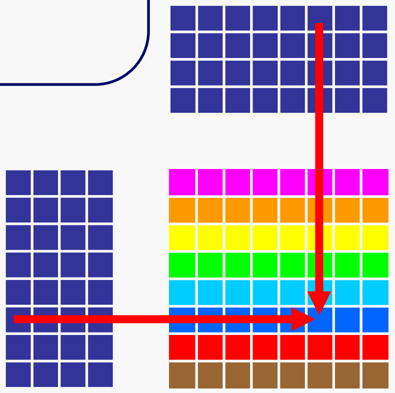
Low-level Synchronization Primitives

Mutex	Implements mutex concept using underlying OS locks (e.g., pthread mutexes)
Spin Mutex	Thread busy waits until able to acquire lock
Queuing Mutex	Threads acquire lock on mutex in the order they request it.
Reader-Writer Mutex	Multiple threads can hold lock if reading. Writing thread must have exclusive lock on mutex

TBB Matrix Multiply

```
class work {
  //data members A,B,C,L,N
public:
  void operator()(const blocked_range<size_t>& r) const {
    for(int i = r.begin(); i != r.end(); ++i) {
      for(int j=0; j<N; ++j) {
        for(int k=0; k<L; ++k) {
          C[i][j] += A[i][k]*B[k][j];
        }
      }
    }
  }
};

task_scheduler_init init;
parallel_for(
  blocked_range<size_t>(0,M,1),
   work(A,B,C,L,M)
);
```

Grainsize parameter determines how many iterations will be executed by a thread at once.



TBB Parallel Sum

```
class sum {
  float* a;
public:
  float sum;

  void operator()(const blocked_range<size_t>& r ) {
    for(size_t i=r.begin(); i!=r.end(); ++i)
      sum += a[i];
  }

  void join(sum& other) { sum += other.sum; }
};

float ParallelSumFoo(float a[], size_t n) {
  sum sum_func(a);
  parallel_reduce(blocked_range<size_t>(0,n,1), sum_func);
  return sum_func.sum;
}
```

Table of Contents

HPF - High Performance Fortran

- History
  - High Performance Fortran Forum (HPFF) coalition founded in January 1992 to define set of extensions to Fortran 77
  - V 1.1 Language specification November, 1994
  - V 2.0 Language specification January, 1997
- HPF
  - Data Parallel (SPMD) model
  - Specification is Fortran 90 superset that adds FORALL statement and data decomposition / distribution directives

**** Adapted from presentation by Janet Salowe - http://www.nbcs.rutgers.edu/hpc/hpf{1,2}/***

The HPF Model

- Execution Model
  - Single-threaded programming model
  - Implicit communication
  - Implicit synchronization
  - Consistency model hidden from user
- Productivity
  - Extension of Fortran (via directives)
  - Block imperative, function reuse
  - Relatively high level of abstraction
  - Tunable performance via explicit data distribution
  - Vendor specific debugger

The HPF Model

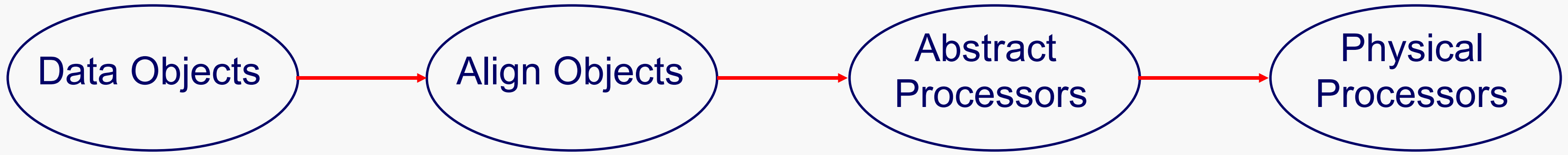
- Performance
  - Latency reduction by explicit data placement
  - No standardized load balancing, vendor could implement
- Portability
  - Language based solution, requires compiler to recognize
  - Runtime system and feature vendor specific, not modular
  - No machine characteristic interface
  - Parallel model not affected by underlying machine
  - I/O not addressed in standard, proposed extensions exist

HPF - Concepts

- DISTRIBUTE - replicate or decompose data
- ALIGN - coordinate locality on processors
- INDEPENDENT - specify parallel loops
- Private - declare scalars and arrays local to a processor

Data Mapping Model

- HPF directives - specify data object allocation
- Goal - minimize communication while maximizing parallelism
- ALIGN - data objects to keep on same processor
- DISTRIBUTE - map aligned object onto processors
- Compiler - implements directives and performs data mapping to physical processors
  - Hides communications, memory details, system specifics



HPF

Ensuring Efficient Execution

- User layout of data
- Good specification to compiler (ALIGN)
- Quality compiler implementation

Simple Example (Integer Print)

```
INTEGER, PARAMETER :: N=16
    INTEGER, DIMENSION(1:N):: A,B
    !HPF$ DISTRIBUTE(BLOCK) :: A
    !HPF$ ALIGN WITH A :: B
    DO i=1,N
    A(i) = i
    END DO
    !HPF$ INDEPENDENT
    FORALL (i=1:N) B(i) = A(i)*2
    WRITE (6,*) 'A = ', A
    WRITE (6,*) 'B = ', B
    STOP
    END
```

Output:

```
0: A = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0: B = 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
```

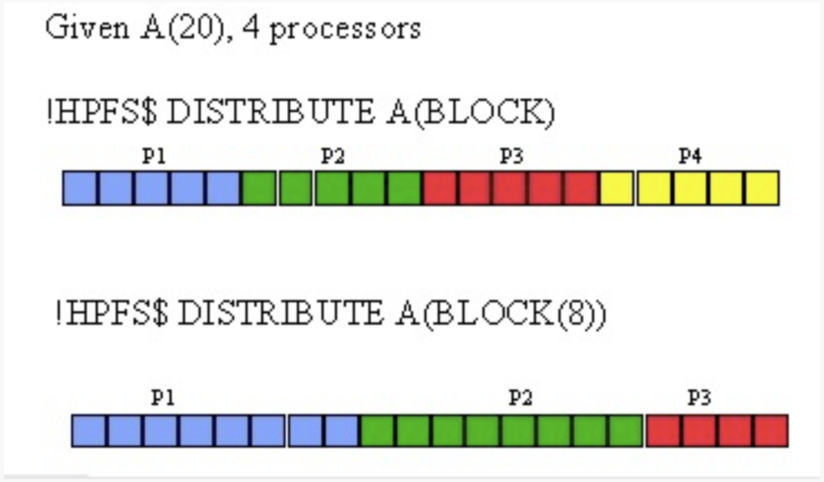
HPF Compiler Directives

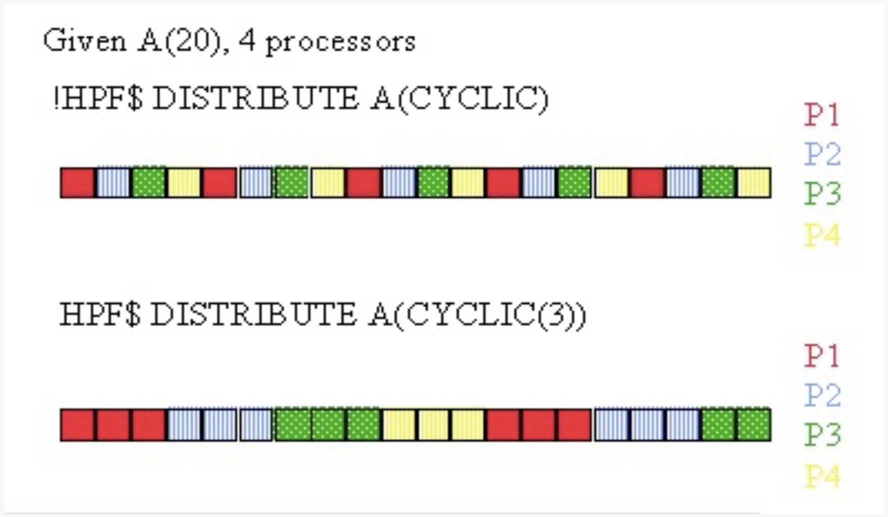
```
trigger-string hpf-directive
```

- `trigger-string` - comment followed by HPF$
- `hpf-directive` - an HPF directive and its arguments
  - **DISTRIBUTE, ALIGN, etc.**

HPF - Distribute

- **`!HPF$ DISTRIBUTE object (details)`**
  - distribution details - comma separated list, for each array dimension
    - BLOCK, BLOCK(N), CYCLIC, CYCLIC(N)
  - object must be a simple name (e.g., array name)
  - object can be *aligned to*, but not aligned


Given A(20), 4 processors
!HPFS$ DISTRIBUTE A(BLOCK)
P1
P2
P3
P4
!HPFS$ DISTRIBUTE A(BLOCK(8))
P1
P2
P3


Given A(20), 4 processors
!HPF$ DISTRIBUTE A(CYCLIC)
P1
P2
P3
P4
HPF$ DISTRIBUTE A(CYCLIC(3))
P1
P2
P3
P4

HPF - ALIGN

- **`!HPF$ ALIGN alignee(subscript-list) WITH object(subscript-list)`**
- **`alignee`** - undistributed, simple object
- **`subscript-list`**
  - All dimensions
  - Dummy argument (int constant, variable or expr.)
  - :
  - *

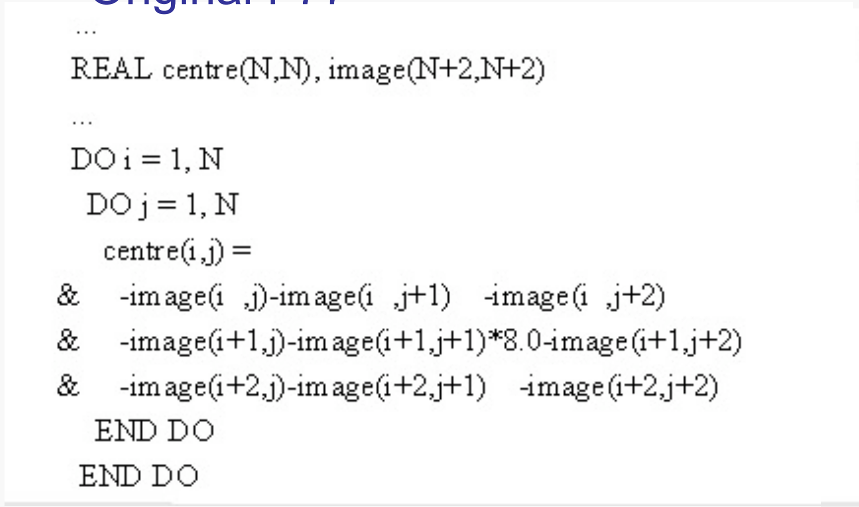
HPF - ALIGN

Equivalent directives, with !HPF$ DISTRIBUTE A(BLOCK,BLOCK)

```
!HPF$ ALIGN B(:,:) WITH A(:,:)
!HPF$ ALIGN (i,j) WITH A(i,j) :: B
!HPF$ ALIGN (:,:) WITH A(:,:) :: B
!HPF$ ALIGN WITH A :: B
```

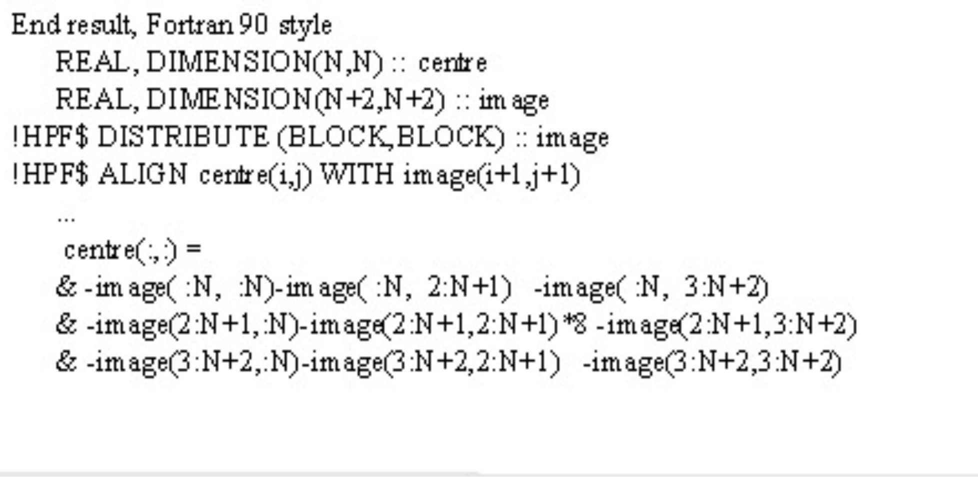
Example

Original F77



```
 ...
 REAL centre(N,N), image(N+2,N+2)
 ...
 DO i = 1, N
  DO j = 1, N
   centre(i,j) =
&    -image(i  ,j)-image(i  ,j+1)   -image(i  ,j+2)
&    -image(i+1,j)-image(i+1,j+1)*8.0-image(i+1,j+2)
&    -image(i+2,j)-image(i+2,j+1)   -image(i+2,j+2)
   END DO
  END DO
```

HPF



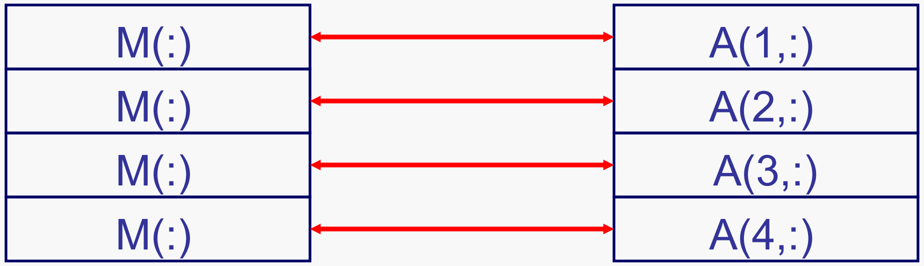
```
End result, Fortran 90 style
    REAL, DIMENSION(N,N) :: centre
    REAL, DIMENSION(N+2,N+2) :: image
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: image
!HPF$ ALIGN centre(i,j) WITH image(i+1,j+1)
    ...
     centre(:,:) =
    & -image( :N,  :N)-image( :N,  2:N+1)  -image( :N,  3:N+2)
    & -image(2:N+1,:N)-image(2:N+1,2:N+1)*8 -image(2:N+1,3:N+2)
    & -image(3:N+2,:N)-image(3:N+2,2:N+1)  -image(3:N+2,3:N+2)
```

HPF - Alignment for Replication

- Replicate heavily read arrays, such as lookup tables, to reduce communication
  - Use when memory is cheaper than communication
  - If replicated data is updated, compiler updates ALL copies

- If array M is used with every element of A:

```
INTEGER M(4)
INTEGER A(4,5)
!HPF$ ALIGN M(*) WITH A(i,*)
```



HPF Example - Matrix Multiply

```
PROGRAM ABmult
IMPLICIT NONE
INTEGER, PARAMETER :: N = 100
INTEGER, DIMENSION (N,N) :: A, B, C
INTEGER :: i, j
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: C
!HPF$ ALIGN A(i,*) WITH C(i,*)
! replicate copies of row A(i,*)
! onto processors which compute C(i,j)
!HPF$ ALIGN B(*,j) WITH C(*,j)
! replicate copies of column B(*,j))
! onto processors which compute C(i,j)
A = 1
B = 2
C = 0
DO i = 1, N
DO j = 1, N
! All the work is local due to ALIGNs
C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
END DO
END DO
WRITE(*,*) C
```

HPF - FORALL

- A generalization of Fortran 90 array assignment (not a loop)
- Does assignment of multiple elements in an array, but order not enforced
- Uses
  - assignments based on array index
  - irregular data motion
  - gives identical results, serial or parallel
- Restrictions
  - assignments only
  - execution order undefined
  - not iterative

```
FORALL (I=1:N) B(I) = A(I,I)
FORALL (I = 1:N, J = 1:N:2, J .LT. I) A(I,J) = A(I,J) / A(I,I)
```

Table of Contents

Chapel

- The Cascade High-Productivity Language (Chapel)
  - Developed by Cray as part of DARPA HPCS program
  - Draws from HPF and ZPL
  - Designed for “general” parallelism

    Supports arbitrary nesting of task and data parallelism
  - Constructs for explicit data and work placement
  - OOP and generics support for code reuse

Adapted From:http://chapel.cs.washington.edu/ChapelForAHPCRC.pdf

The Chapel Model

- Execution Model
  - Explicit data parallelism with **`forall`**
  - Explicit task parallelism **`forall`**, **`cobegin`**, **`begin`**
  - Implicit communication
  - Synchronization
    - Implicit barrier after parallel constructs
    - Explicit constructs also included in language
  - Memory Consistency model still under development

Chapel - Data Parallelism

- `forall` loop

  loop where iterations performed concurrently

```
    forall i in 1..N do
      a(i) = b(i);
```

alternative syntax:

```
    [i in 1..N] a(i) = b(i);
```

Chapel - Task Parallelism

- **`forall`** expression
  allows concurrent evaluation expressions

```
 [i in S] f(i);
```

- **`cobegin`**
  indicate statement that may run in parallel

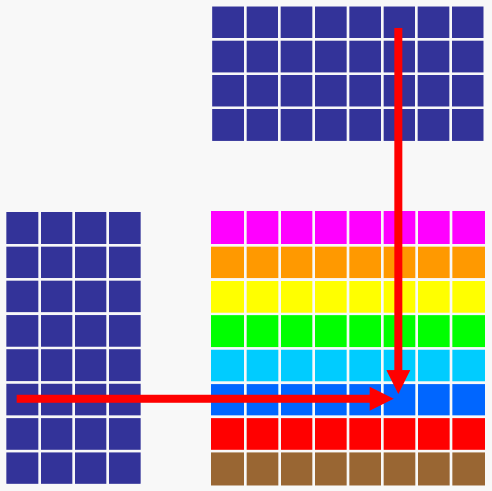
```
      cobegin {
       ComputeTaskA(…);
       ComputeTaskB(…);
   }
```

- **`begin`**
  spawn a computation to execute a statement

```
   begin ComputeTaskA(…); //doesn't rejoin
    ComputeTaskB(…);      //doesn't wait for ComputeTaskA
```

Chapel - Matrix Multiply

```
var A: [1..M, 1..L] float;
var B: [1..L, 1..N] float;
var C: [1..M, 1..N] float;

forall (i,j) in [1..M, 1..N] do
   for k in [1..L]
      C(i,j) += A(i,k) * B(k,j);
```



Chapel - Synchronization

- `single` variables
  - Chapel equivalent of **futures**
  - **Use** of variable stalls until variable **assignment**

```
var x : single int;
begin x = foo();  //sub computation spawned
var y = bar;
return x*y;       //stalled until foo() completes.
```

- `sync` variables
  - generalization of single, allowing multiple assignments
  - ***full / empty*** semantics, read ‘empties’ previous assignment
- `atomic` statement blocks
  - transactional memory semantics
  - no changes in block visible until completion

Chapel - Productivity

- New programming language
- Component reuse
  - Object oriented programming support
  - Type generic functions
- Tunability
  - Reduce latency via explicit work and data distribution
- Expressivity
  - Nested parallelism supports composition
- Defect management
  - ‘Anonymous’ threads for hiding complexity of concurrency
    no user level thread_id, virtualized

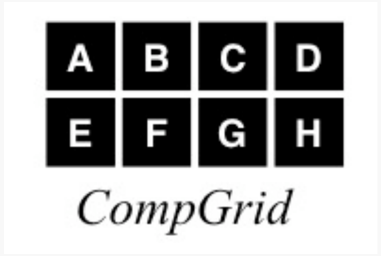
Chapel - Performance

- Latency Management
  - Reducing
    - Data placement - distributed domains
    - Work placement - `on` construct
  - Hiding
    - `single` variables
    - Runtime will employ multithreading, if available

Chapel - Latency Reduction

- Locales
  - Abstraction of processor or node
  - Basic component where memory accesses are assumed uniform
  - User interface defined in language
    - integer constant **numLocales**
    - type **`locale`** with (in)equality operator
    - array **`Locales`**`[1..`**`numLocales`**`]` of type **`locale`**

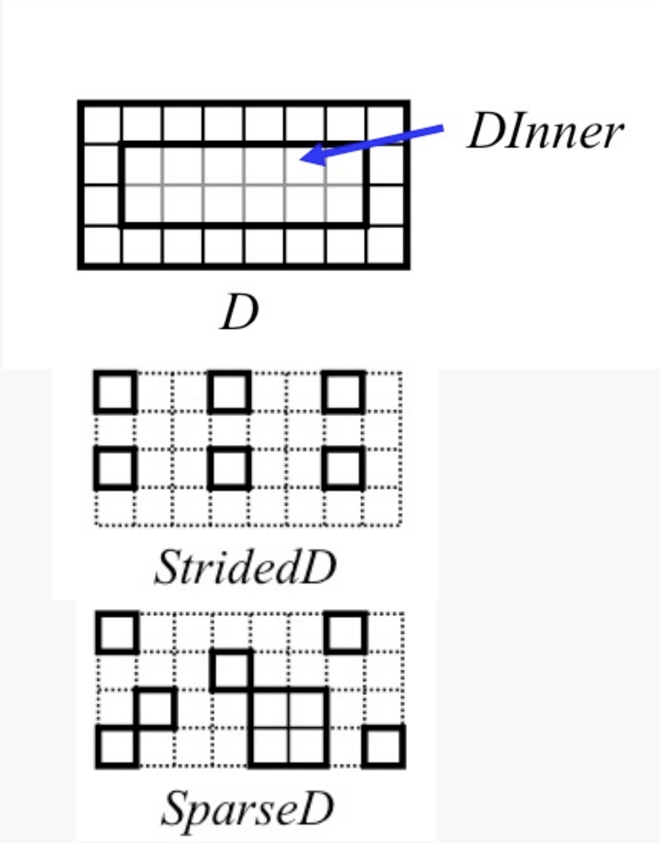
```
var CompGrid:[1..Rows, 1..Cols] local = ...;
```



Chapel - Latency Reduction

- Domain
  - set of indices specifying size and shape of aggregate types (i.e., arrays, graphs, etc)

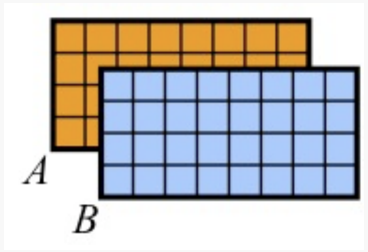
```
var m: integer = 4;
var n: integer = 8;
var D: domain(2) = [1..m, 1..n];
var DInner: domain(D) = [2..m-1, 2..n-1]


var StridedD: domain(D) = D by (2,3);


var indexList: seq(index(D)) = ...;
var SparseD: sparse domain(D) = indexList;
```



Chapel - Domains

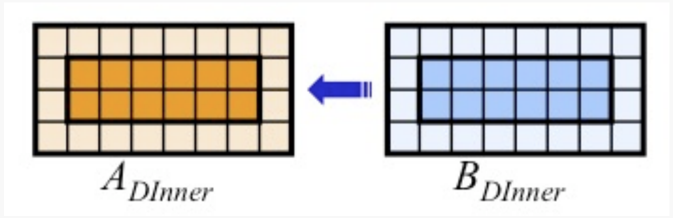
- Declaring arrays

```
var A, B: [D] float
```



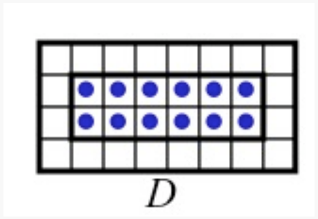
- Sub-array references

```
A(Dinner) = B(Dinner);
```



- Parallel iteration

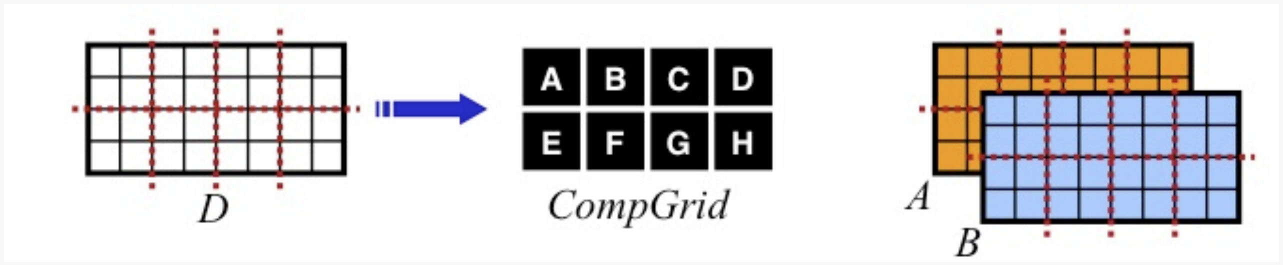
```
forall (i,j) in Dinner { A(i,j} = ...}
```



Chapel - Latency Reduction

- Distributed domains
  - Domains can be *explicitly* distributed across locales

```
var D: domain(2) distributed(block(2) to CompGrid) = ...;
```

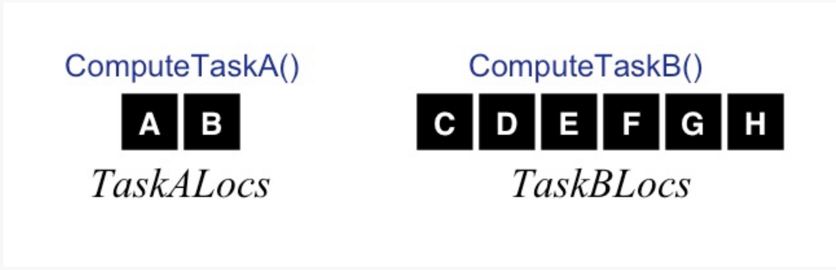


  - Pre-defined
    - `block, cyclic, block-cyclic, cut`
  - User-defined distribution support in development

Chapel - Latency Reduction

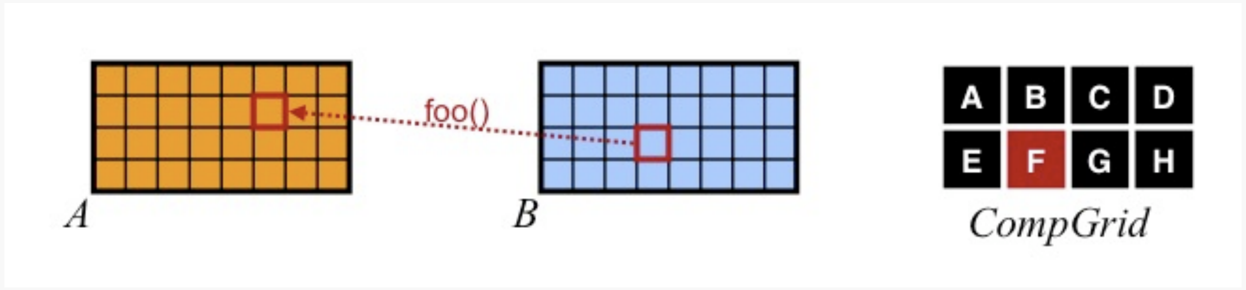
- Work Distribution with **on**

```
cobegin {
  on TaskALocs do ComputeTaskA(...);
  on TaskBLocs do ComputeTaskB(...);
}
```



alternate data-driven usage:

```
forall (i,j) in D {
  on B(j/2, i*2) do A(i,j) = foo(B(j/2, i*2));
}
```



Chapel - Portability

- Language based solution, requires compiler
- Runtime system part of Chapel model. Responsible for mapping implicit multithreaded, high level code appropriately onto target architecture
- **locales** machine information available to programmer
- Parallel model not effected by underlying machine
- I/O API discussed in standard, scalability and implementation not discussed

Table of Contents

The Fortress Model

- Developed by Sun for DARPA HPCS program
- Draws from Java and functional languages
- Emphasis on growing language via strong library development support
- Places parallelism burden primarily on library developers
- Use of extended Unicode character set allow syntax to mimic mathematical formulas

```
trait EquivalenceRelation⟦T extends EquivalenceRelation⟦T, ∼⟧, opr ∼⟧
      extends { Reflexive⟦T, ∼⟧, Symmetric⟦T, ∼⟧, Transitive⟦T, ∼⟧ }
end
```

Adapted From: http://irbseminars.intel-research.net/GuySteele.pdf

The Fortress Model

Execution Model

- User sees single-threaded execution by default
  - Loops are assumed parallel, unless otherwise specified
- Data parallelism
  - Implicit with `for` construct
  - Explicit ordering via custom Generators
- Explicit task parallelism
  - Tuple and `do all` constructs
  - Explicit with `spawn`

The Fortress Model

Execution Model

- Implicit communication
- Synchronization
  - Implicit barrier after parallel constructs
  - Implicit synchronization of reduction variables in `for` loops
  - Explicit `atomic` construct (transactional memory)
- Memory Consistency
  - Sequential consistency under constraints
    - all shared variable updates in `atomic` sections
    - no implicit reference aliasing

Fortress - Data Parallelism

- `for` loops - default is parallel execution

```
for i←1:m, j←1:n do
  a[i,j] := b[i] c[j]
end
```

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

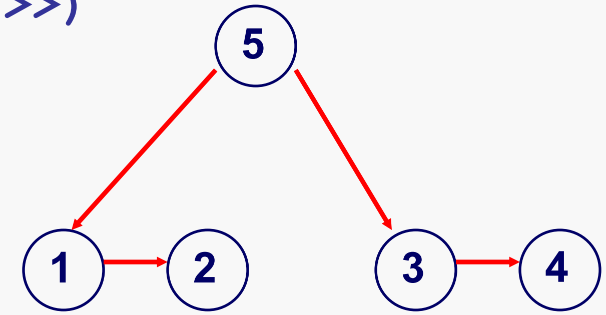
`1:N` and `seq(1:N)` are *generators*

`seq(1:N)` *is generator for sequential execution*

Fortress - Data Parallelism

- Generators
  - Controls parallelism in loops
  - Examples
    - Aggregates - `<1,2,3,4>`
    - Ranges - `1:10` and `1:99:2`
    - Index sets - `a.indices` and `a.indices.rowMajor`
    - `seq(g)` - sequential version of generator `g`
  - Can compose generators to order iterations

```
seq(<5,<seq(<1,2>), seq(<3,4>)>>)
```


$$\begin{tikzcd} & 5 \arrow[dl] \arrow[drr] & & \\ 1 \arrow[r] & 2 & 3 \arrow[r] & 4 \end{tikzcd}$$

Fortress - Explicit Task Parallelism

- Tuple expressions
  - comma separated exp. list executed concurrently

```
(foo(), bar())
```

- **`do-also`** blocks
  - all clauses executed concurrently

```
do
   foo()
also do
   bar()
end
```

Fortress - Explicit Task Parallelism

- Spawn expressions (futures)

```
…
v = spawn do
   …
end
…
v.val()    //return value, block if not completed
v.ready()  //return true iff v completed
v.wait()   //block if not completed,
```

-
```
           //no return value
v.stop()   //attempt to terminate thread
```

Fortress - Synchronization

- `atomic` blocks - transactional memory
  - other threads see block completed or not yet started
  - nested **`atomic`** and parallelism constructs allowed
  - **`tryatomic`** can detect conflicts or aborts

```
$sum : N := 0$
$accumArray[\![N$ extends Additive, nat $x]\!](a : N[x]) : () =$
  for $i \leftarrow a.indices$ do
    atomic $sum \mathrel{+}= a[i]$
  end
```

```
do
    $x : \mathbb{Z} := 0$
    $y : \mathbb{Z} := 0$
    $z : \mathbb{Z} := 0$
    atomic do
        $x \mathrel{+}= 1$
        $y \mathrel{+}= 1$
    also atomic do
        $z := x + y$
    end
    $z$
end
```

Fortress - Productivity

- Defect management
  - Reduction
    - explicit parallelism and tuning primarily confined to libraries
  - Detection
    - integrated testing infrastructure
- Machine model
  - *Regions* give abstract machine topology

Fortress - Productivity

Expressivity

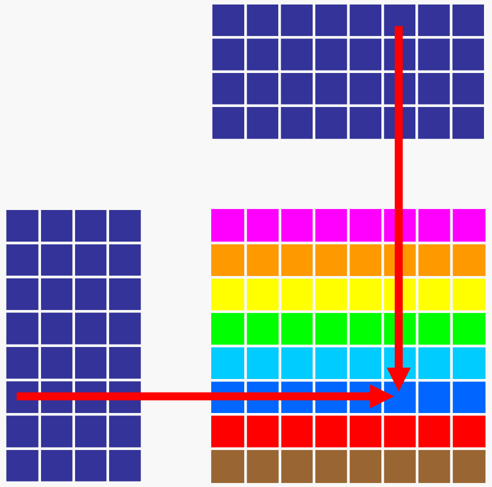
- High abstraction level
  - Source code closely matches formulas via extended Unicode charset
  - Types with checked physical units
  - Extensive operator overloading
- Composition and Reuse
  - Type-based generics
  - Arbitrary nested parallelism
  - Inheritance by traits
- Expandability
  - ‘Growable’ language philosophy aims to minimize core language constructs and maximize library implementations

Fortress - Productivity

- Implementation refinement
  - Custom generators, distributions, and thread placement
- Defect management
  - Reduction
    - explicit parallelism and tuning primarily confined to libraries
  - Detection
    - integrated testing infrastructure
- Machine model
  - *Regions* give abstract machine topology

Fortress - Matrix Multiply

```
matmult(A: Matrix[/Float/],
        B: Matrix[/Float/])
       : Matrix[/Float/]
  A B
end

C = matmult(A,B)
```

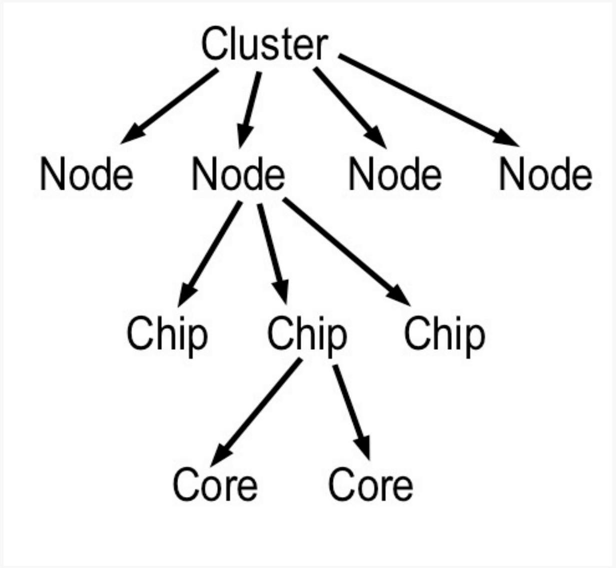


Fortress - Performance

- Regions for describing system topology
- Work placement with **`at`**
- Data placement with Distributions
- **`spawn`** expression to hide latency

Fortress - Regions

- Tree structure of CPUs and memory resources
  - Allocation heaps
  - Parallelism
  - Memory coherence
- Every thread, object, and array element has associated region



```
obj.region()   //region where object obj is located
r.isLocalTo(s) //is region r in region tree rooted at s
```

Fortress - Latency Reduction

- Explicit work placement with `at`

inside `do also`

```
do
    $v := a_i$
also at $a.region(j)$ do
    $w := a_j$
end
```

with `spawn`

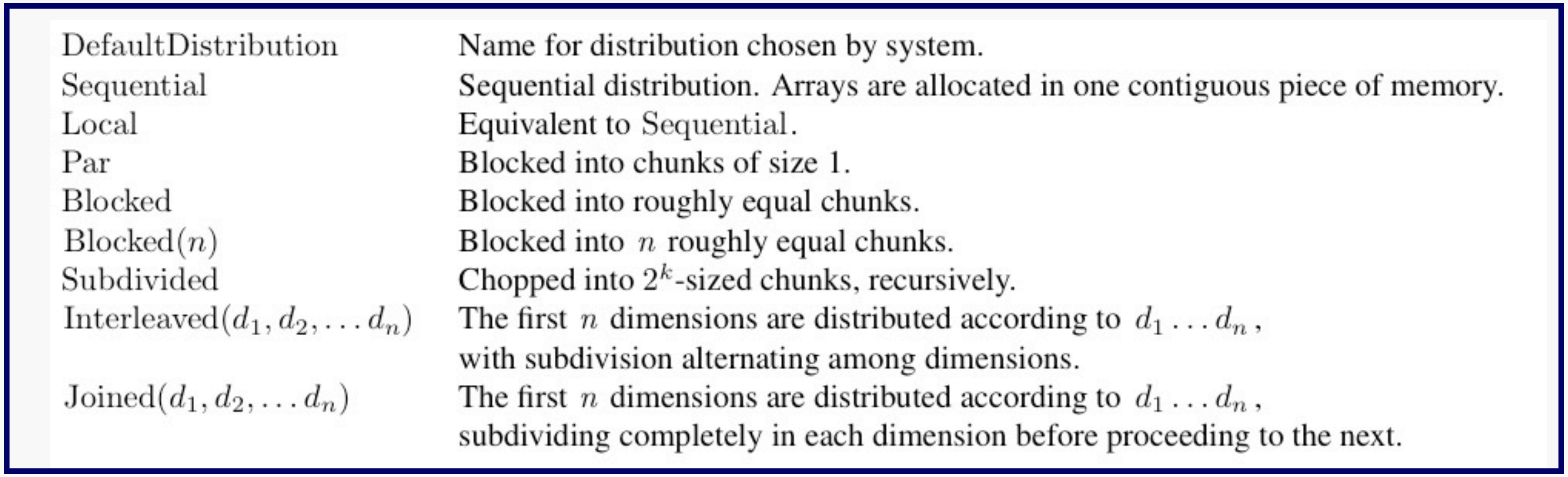
```
$v =$ spawn at $a.region(i)$ do
                $a_i$
         end
$w =$ spawn at $v.region()$ do
                $v.val() \cdot 17$
         end
```

regular block stmt

```
do
    $v := a_i$
    at $a.region(j)$ do
        $w := a_j$
    end
    $x = v + w$
end
```

Fortress - Latency Reduction

- Explicit data placement with Distributions



DefaultDistribution	Name for distribution chosen by system.
Sequential	Sequential distribution. Arrays are allocated in one contiguous piece of memory.
Local	Equivalent to Sequential.
Par	Blocked into chunks of size 1.
Blocked	Blocked into roughly equal chunks.
Blocked(n)	Blocked into n roughly equal chunks.
Subdivided	Chopped into 2^k-sized chunks, recursively.
Interleaved($d_1, d_2, \ldots d_n$)	The first n dimensions are distributed according to $d_1 \ldots d_n$, with subdivision alternating among dimensions.
Joined($d_1, d_2, \ldots d_n$)	The first n dimensions are distributed according to $d_1 \ldots d_n$, subdividing completely in each dimension before proceeding to the next.

```
a = Blocked.array(n,n,1); //Pencils along z axis
```

- User can define custom distribution by inheriting **Distribution** trait
  - Standard distributions implemented in this manner

Fortress - Portability

- Language based solution, requires compiler
- Runtime system part of Fortress implementation
  Responsible for mapping multithreaded onto target architecture
- **Regions** make machine information available to programmer
- Parallel model not affected by underlying machine

Table of Contents

The STAPL Model

- Standard Adaptive Parallel Library
- Developed by Lawrence Rauchwerger, Nancy Amato and several grad students and postdocs at Texas A&M
- Library similar and compatible with to STL
- Strong library development support
- Places parallelism burden primarily on library developers
- Influenced in earlier versions: Intel TBB

Adapted From: http://irbseminars.intel-research.net/GuySteele.pdf

STAPL: Standard Template Adaptive Parallel Library

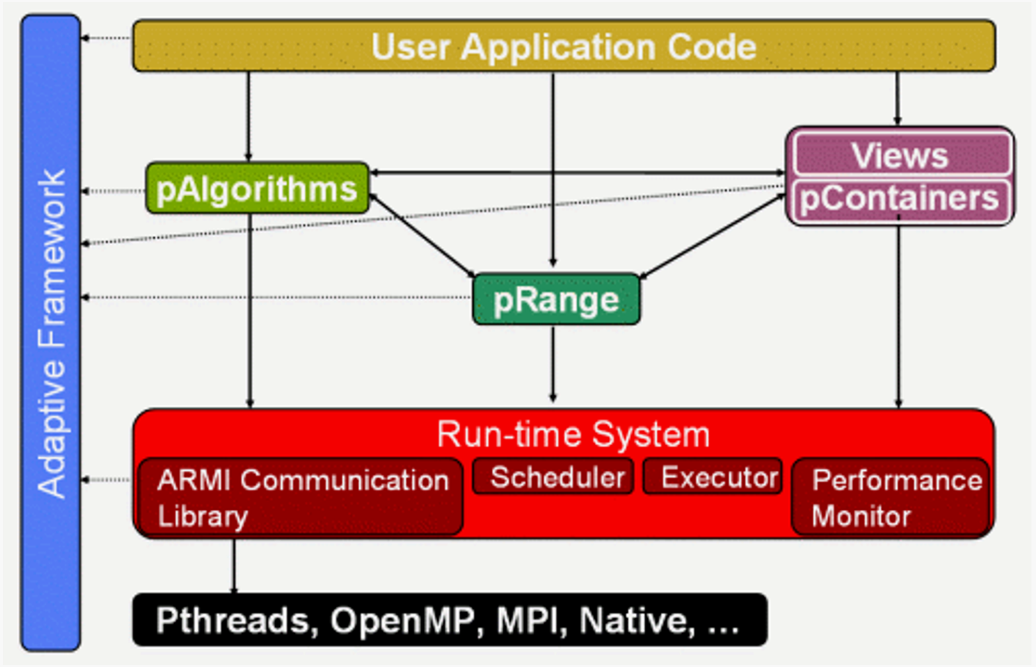
A library of parallel components that adopts the generic programming philosophy of the C++ Standard Template Library (STL)

Application Development Components

- pAlgorithms, pContainers, Views, pRange
- Provide Shared Object View to eliminate explicit communication in application

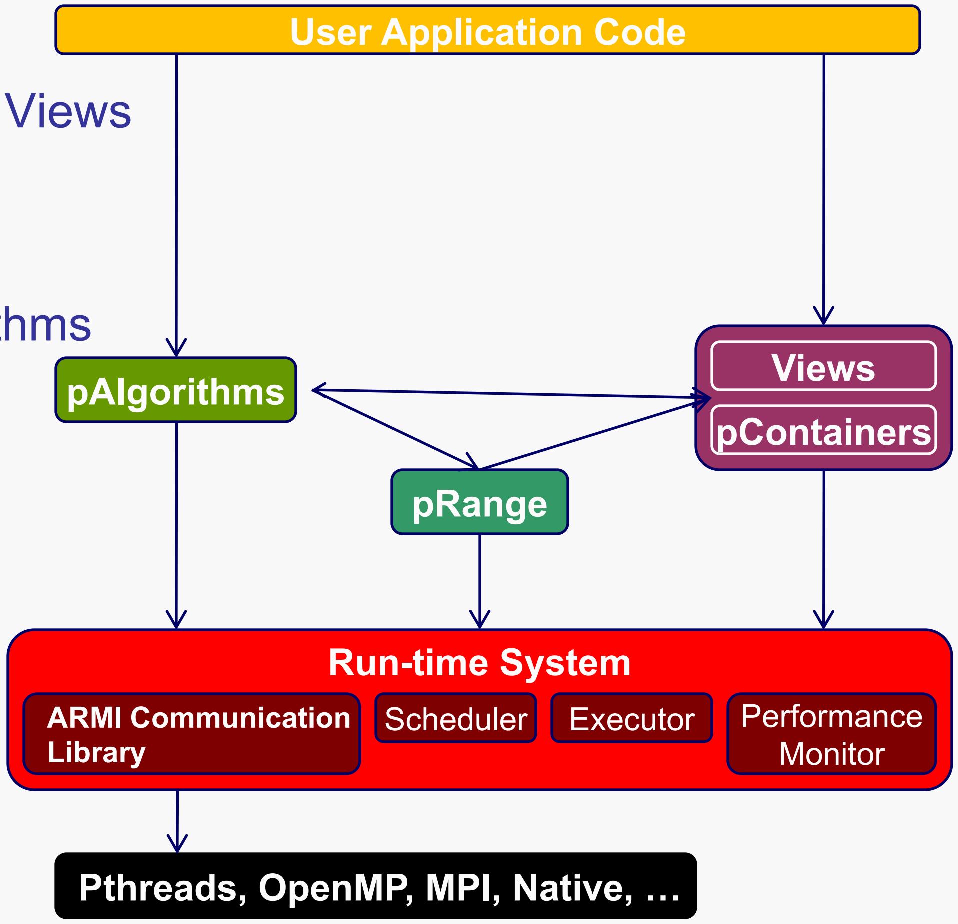
Portability and Optimization

- Runtime System(RTS) and Adaptive Remote Method Invocation (ARMI) Communication Library
- Framework for Algorithm Selection and Tuning (FAST)



Three STAPL Developer Levels

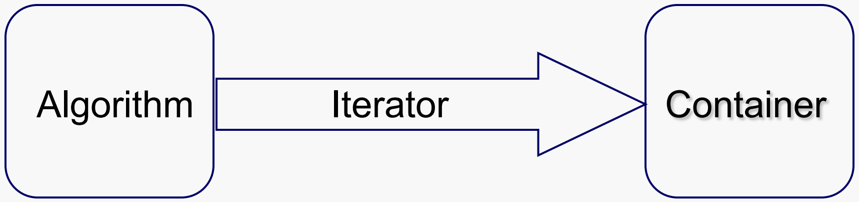
- Application Developer
  - Writes application
  - Uses pContainers, pAlgorithms, and Views
- Library Developer
  - Writes new pContainers and pAlgorithms
  - Uses pRange and RTS
- Run-time System Developer
  - Ports system to new architectures
  - Writes task scheduling modules
  - Uses native threading and communication libraries



Standard Template Library (STL)

Generic programming components using C++ templates.

- Containers - collection of other objects.
  - vector, list, deque, set, multiset, map, multi_map, hash_map.
  - Templated by data type. `vector<int> v(50);`
- Algorithms - manipulate the data stored in containers.
  - manipulate the data stored in containers.
  - count(), reverse(), sort(), accumulate(), for_each(), reverse().
- Iterators - Decouple algorithms from containers.
  - Provide generic element access to data in containers.
  - can define custom traversal of container (e.g., every other element)
  - `count(vector.begin(), vector.end(), 18);`



Execution Model

- Two models: User and Library Developer
- Single threaded – User
- Multithreaded – Developer
- Shared memory – User
- PGAS – Developer
- Data & task parallelism
- Implicit communications: User
- Explicit communications: Developer

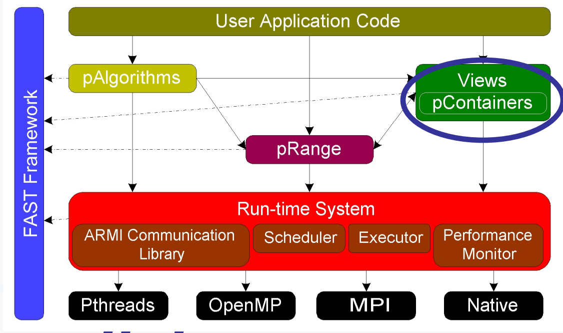
Execution Model

- Memory Consistency:
  - Sequential for user
  - Relaxed for developer (Object level)
  - Will be selectable
- Atomic methods for containers
- Synchronizations: Implicit & Explicit

STAPL Components

- Components for Program Development
  - pContainers, Views, pRange, pAlgorithms
- Run-time System
  - Adaptive Remote Method Invocation (ARMI)
  - Multithreaded RTS
  - Framework for Algorithm Selection and Tuning (FAST)

pContainers

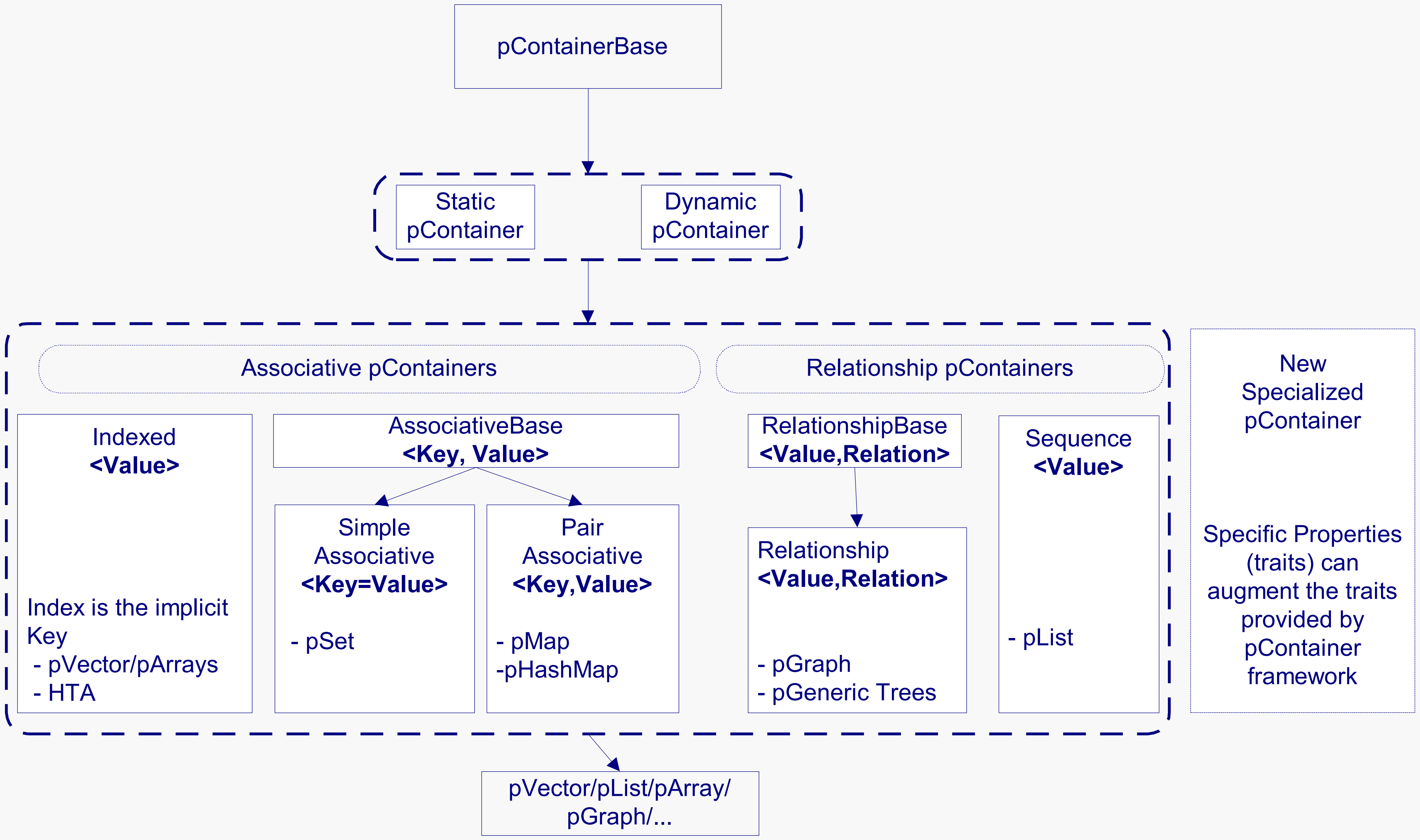


Generic, distributed data structures with parallel methods.

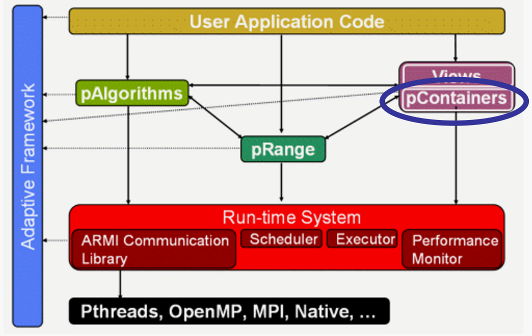
- Ease of Use
  - Shared object view
  - Generic access mechanism through Views
  - Handles data distribution and remote data access internally
  - Interface equivalent with sequential counterpart
- Efficiency
  - OO design to optimize specific containers
  - Template parameters allow further customization
- Extendability
  - New pContaintеrs extend Base classes
- Composability
  - pContainers of pContainers

Currently Implemented
pArray, pVector, pGraph, pMap, pHashMap, pSet, pList

pContainer Taxonomy


pContainerBase
Static pContainer
Dynamic pContainer
Associative pContainers
Relationship pContainers
Indexed <Value>
Index is the implicit Key
- pVector/pArrays
- HTA
AssociativeBase <Key, Value>
Simple Associative <Key=Value>
- pSet
Pair Associative <Key,Value>
- pMap
-pHashMap
RelationshipBase <Value,Relation>
Relationship <Value,Relation>
- pGraph
- pGeneric Trees
Sequence <Value>
- pList
New Specialized pContainer
Specific Properties (traits) can augment the traits provided by pContainer framework
pVector/pList/pArray/ pGraph/...

pContainers : Parallel Containers



- Container - Data structure with an interface to maintain and access a collection of generic elements
  - STL (vector, list, map, set, hash), MTL[1] (matrix), BGL[2] (graph), etc.

- pContainer - distributed storage and concurrent methods
  - **Shared Object View**
  - Compatible with sequential counterpart (e.g., STL)
  - Thread Safe
  - Support for user customization (e.g., data distributions)
  - Currently Implemented: pArray, pVector, pList, pGraph, pMatrix, pAssociative

1] Matrix Template Library 2] Boost Graph Library

pContainer Framework

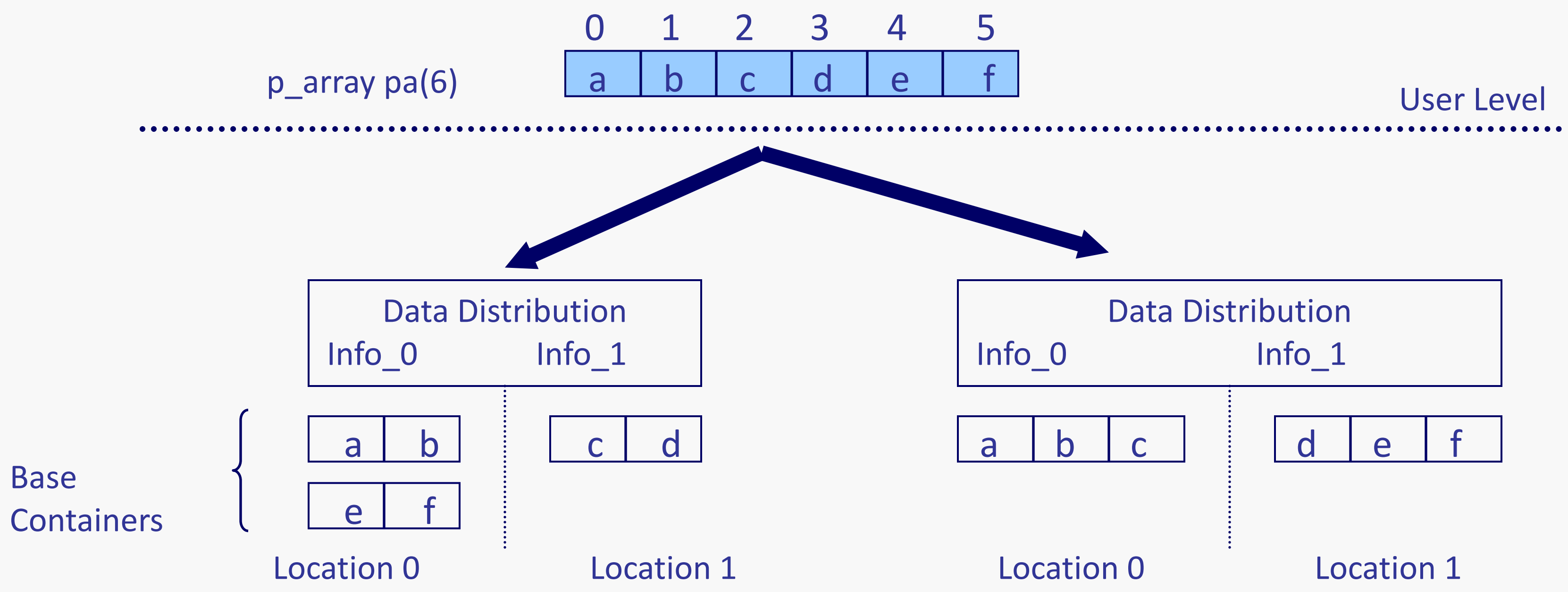
Concepts and methodology for developing parallel containers

- pContainers - a collection of base containers and information for parallelism management
- Improved user productivity
  - Base classes providing fundamental functionality
    - Inheritance
    - Specialization
  - Composition of existing pContainers
- Scalable performance
  - Distributed, non replicated data storage
  - Parallel (semi-random) access to data
  - Low overhead relative to the base container counterpart

pContainer Framework Concepts

- Base Container : data storage
  - sequential containers (e.g., STL, MTL, BGL)
  - parallel containers (e.g., Intel TBB)

- Data Distribution Information
  - Shared object view
  - Global Identifier, Domain, Partition, Location, Partition Mapper



pContainer Interfaces

- Constructors
  - Default constructors
  - May specify a desired data distribution
- Concurrent Methods
  - Sync, async, split phase
- Views

```
stapl_main(){
  partition_block_cyclic partition(10); //argument is block size
  p_matrix<int> data(100, 100, partition);
  1D_view<int> view(data);
  p_generate(view, rand());
  res=p_accumulate(view);
}
```

Method Aggregation Support

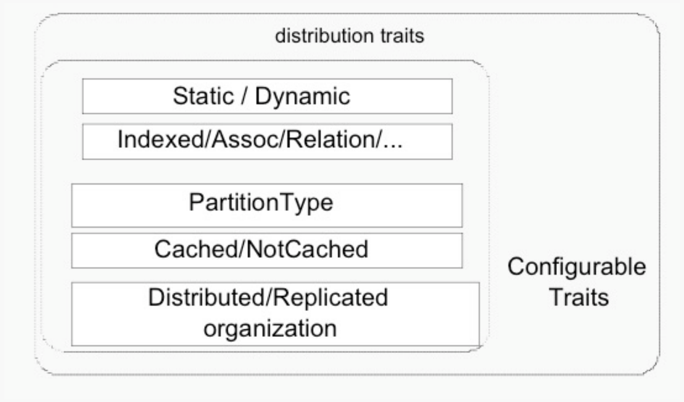
- pContainers support three types of methods: synchronous, asynchronous and split phase.
- Asynchronous and split phase methods benefit from
  - Aggregation : better communication computation overlap
  - Combining : same method is repeatedly invoked
- Aggregation may negatively affect performance when data is needed immediately
  - Need for adaptivity

pContainer Customization

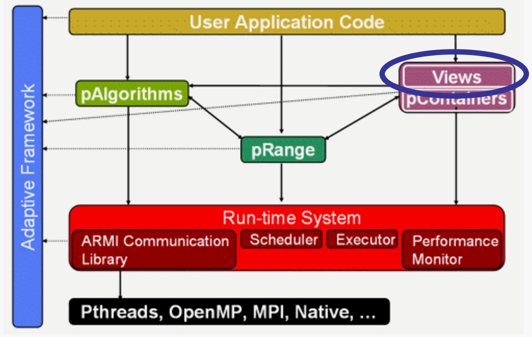
Optional user customization through pContainer **Traits**.

- Enable/Disable Performance Monitoring.
- Select Partition Strategies.
- Enable/Disable Thread Safety.
- Select Consistency Models

```
class p_array_traits {
 Indexed, Assoc/Key=Index,
 Static,IndexedView<Static,..., Random>,
 DistributionManagerTraits,
 -u-Monitoring,
 -u-Relaxed
}
```



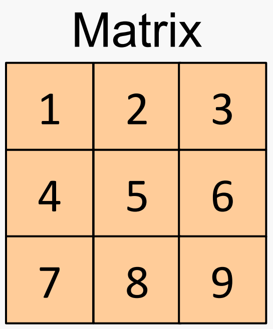
Views



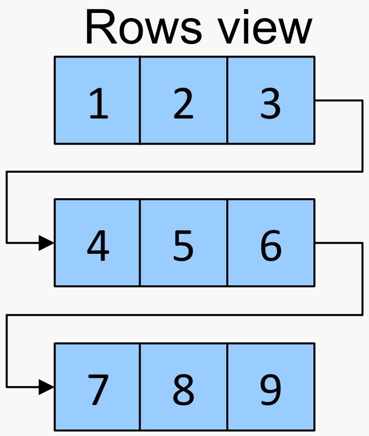
- A View defines an abstract data type that provides methods for access and traversal of the elements of a pContainer that is independent of how the elements are stored in the pContainer.
  - Example: Matrix View of the elements in a pVector

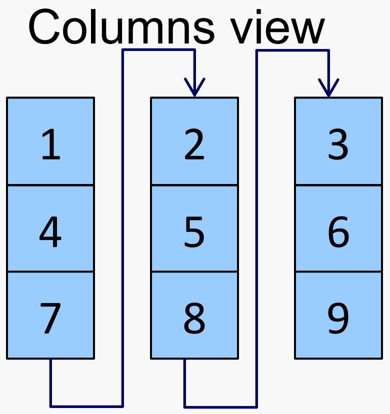
- A View has four primary components
  - The pContainer that stores the data being ordered
  - A Domain that defines which elements of the pContainer are visible
  - A Mapping Function that translates between View and pContainer indices
  - The Set of Operations that define the data access operations supported

View Example



- Print elements of Matrix
  - row-wise or column-wise?
  - Implement several print methods…
  - Use one generic print method with different views





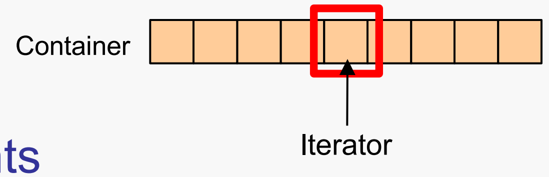
```
print(View v)
  for i=1 to v.size() do
    print(v[i])
```

Output
1,2,3,4,5,6,7,8,9

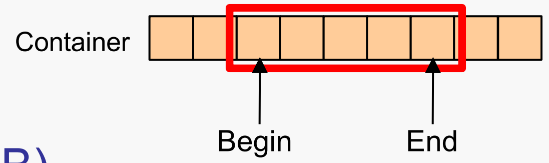
Output
1,4,7,2,5,8,3,6,9

Iterators, Ranges, and Views

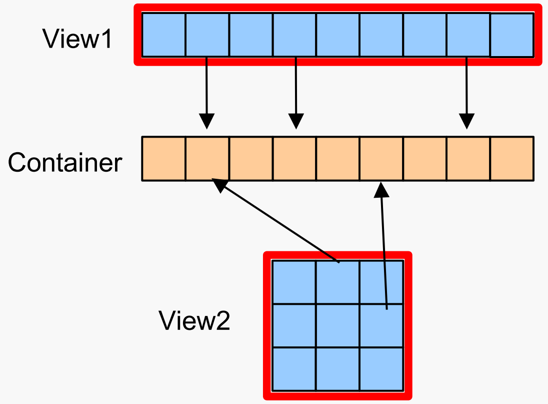
- **Iterators :** Abstraction of pointers
  - Provide element access and traversal.
  - Algorithms decoupled from containers (e.g., STL)
  - Naturally sequential, limited use in parallel environments



- **Ranges:** Combine two iterators (start and end)
  - All the advantages of iterators
  - Split range and process subranges in parallel (e.g., TBB)
  - Some algorithms (e.g., blocked matmul) not easy/intuitive to implement

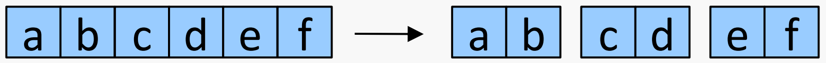


- **Views:** define an abstract data type
  - Allow decoupling of container interface and storage
  - Provide container behavior (interface)
  - Allow transformation of one data structure into another
  - Work well in parallel environments

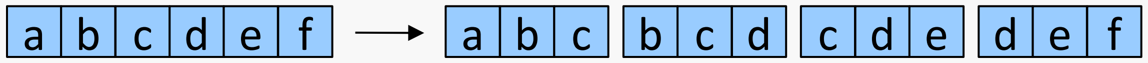


Using Views in STAPL

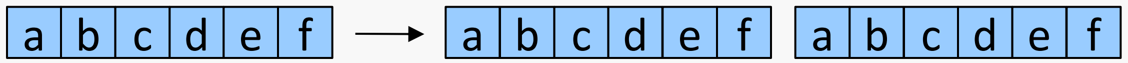
- Views can be partitioned into subviews
  - Partition expresses the finest grain of data used in an algorithm
  - Each subview may be processed in parallel
- Partitions identified for parallel versions of STL algorithms
  - Proper partition (accumulate, find, etc.)



  - Overlap partition (adjacent difference)



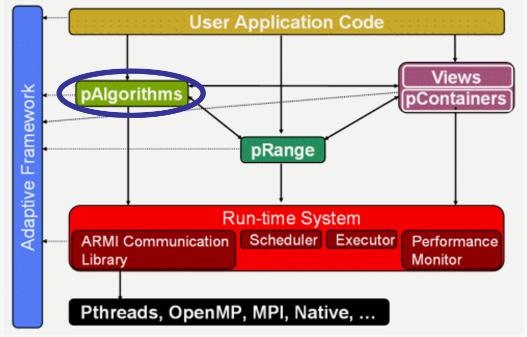
  - Full overlap partition (substring matching)



pContainers and Views in the STAPL Programming Model

- Provide Shared object view of data to application
  - Data partition and distribution can be specified.
  - Implementation of distribution encapsulated in pContainer.
- Allows storage independent data access
  - Views provide data access operations
  - View domains can be partitioned independently of data storage in pContainer.
- User focused on application instead of data access and distribution details.

pAlgorithms



- Build and execute pRanges to perform computation
- Easy to develop
  - Work functions look like sequential code
  - Work functions can call STAPL pAlgorithms
  - pRange factories simplify task graph construction
- STAPL pAlgorithms accelerate application development
  - Basic building blocks for applications
  - Parallel equivalents of STL algorithms
  - Parallel algorithms for pContainers
    - Graph algorithms for pGraphs
    - Numeric algorithms/operations for pMatrices
- Demonstrate how to code in the STAPL programming model.

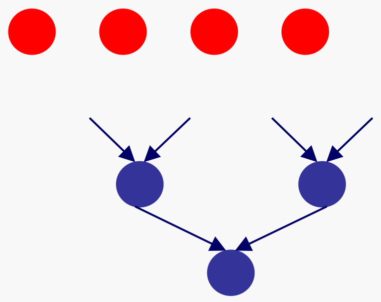
Parallel Count

- Count how many elements matches a given predicate

```
Template <typename Func>
size_t
stapl::count_if(View view, Func f)
  {
  return
  map_reduce(
    f,
    stapl::plus<size_t>,
    view,
  );
}
```

map operation

reduce operation



```
class Func {
public:
  template <typename T>
  bool operator() (T x) {
    return ...;
  }
};
```

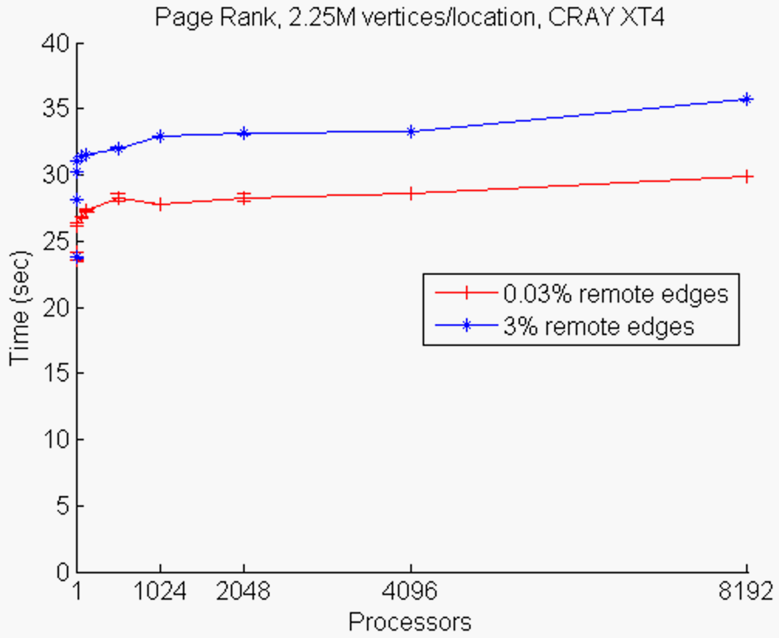
Parallel Sample Sort

- pAlgorithm written using sequence of task graphs.

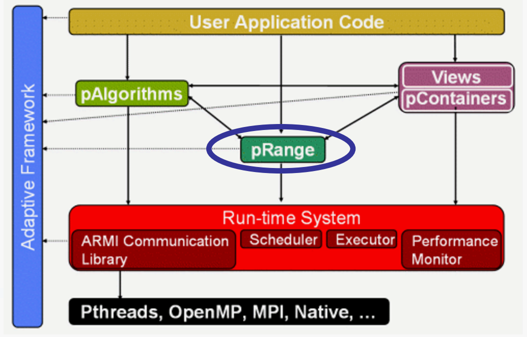
```
p_sort(View view, Op comparator)
  // handle recursive call
  if (view.size() <= get_num_locations())
    reduce(view, merge_sort_work_function(comparator));

  sample_view = map(view, select_samples_work_function());

  // sort the samples
  p_sort(sample_view, comparator);

  // paritition the data using the samples
  partitioned_view = map(view, full_overlap_view(sample_view),
                         bucket_partition_work_function(comparator));

  // sort each partition
  map(partitioned_view, sort_work_function(comparator));
```

Paragraph Page Rank Algorithm

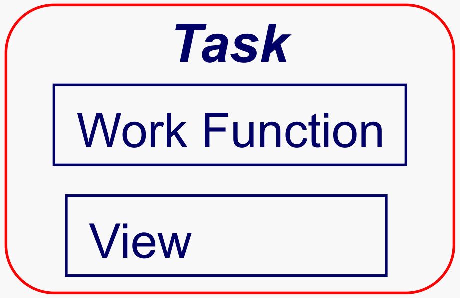
```
page_rank(GView graph, SrcView source_view, TgtView target_view) {
  map(init(damping), source_view)//initial rank
  while(!done()) {
    //Initialize/Reset target_view
    map(init(1-damping), target_view);

    //update neighboring page ranks based on each page's rank.
    map(page_rank_task(), graph.vertices(), source_view, target_view);

    //copy updated ranks
    p_copy(target_view, source_view);
  }
}
```

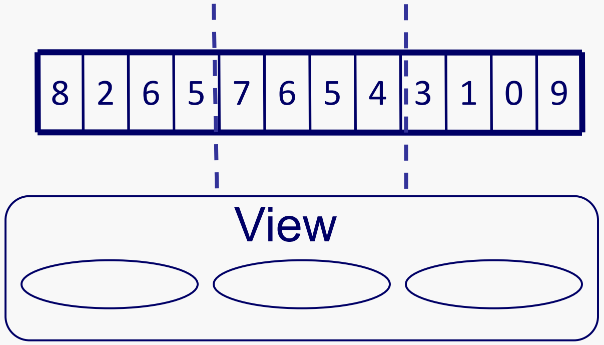


Paragraph: Task Graphs in STAPL

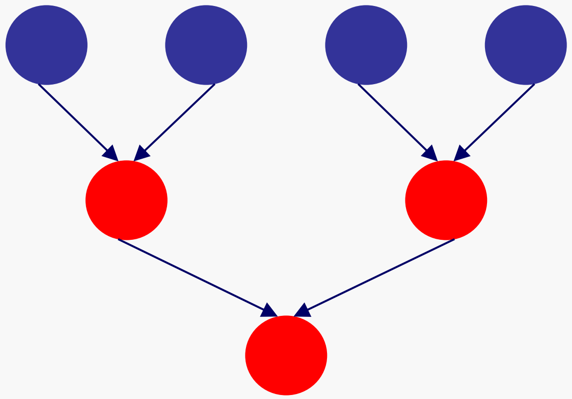


- Task
  - Work function
  - Data to process



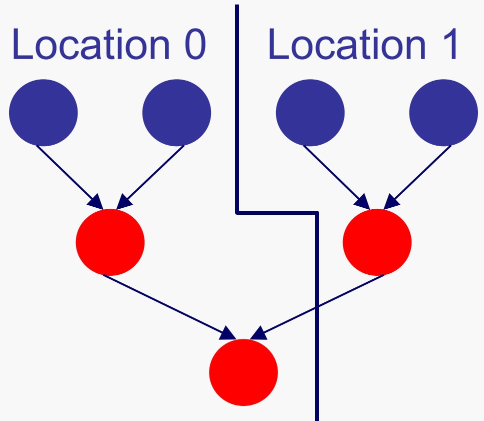


- Task dependencies
  - Expressed in Task Dependence Graph (TDG)
  - TDG queried to find tasks ready for execution



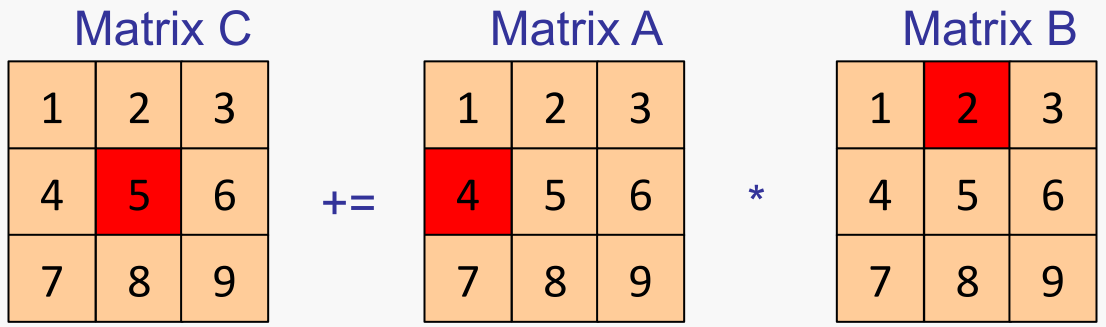
Constructing Task Graphs

- Task Factory
  - Given work functions to use and views.
  - Constructs the pRange tasks and specifies dependencies.
  - Encodes a particular computation pattern.
    (e.g. doall, reduce, etc.)
- Task Factory generates tasks incrementally
  - pRange requests an initial set of tasks from the factory.
  - pRange asks factory for more tasks as execution progresses.
  - pRange execution is complete when all tasks have been generated and executed.
  - Example: Incremental generation of map-reduce on 2 locations



Task Placement

- Factories generate tasks in a distributed manner
  - Domain of views to be processed is split across locations
  - A location may generate a task whose data is not local
  - Solution: Query domain for locality information and forward task
    - Tasks processing multiple views collect votes and then elect location.
    - Location election algorithm can be specified for each task graph instantiation.
    - Domains return a definite location or the location where election is repeated.
  - Example: Task computing C[2][2] += A[2][1] * B[1][2]
    - Each block is on a separate location. No location has majority.
    - Location election specified to give priority to location with writes.
    - Allows specialization of task to use BLAS.



Task will be placed on location 5 for processing.

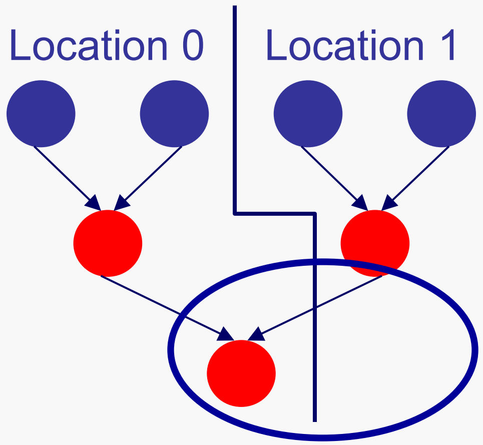
Processing Task Graphs

- pRanges are processed by the Executor.
  - Component of the Runtime System.
  - Allows scheduling the execution order of ready tasks. (E.g., execute tasks on critical path of task graph first)

```
execute(pRange prange)
  while(!prange.finished())
    runtime_scheduler.insert(prange.get_ready_tasks())
    foreach task in runtime_scheduler
      task() // execute the task
      prange.processed(task.id) // report execution to pRange
    endfor
  endwhile
end
```

Result Forwarding

- Result of a task may be needed by a non-local task.
- Tasks created notify predecessors of their location.
- Predecessor task sends result to all successor locations when it executes.
- Functionality is encapsulated in data_flow_view class.
  - Task graph factories specify only that a task consumes a result
  - Task registration and result forwarding hidden from pRange.



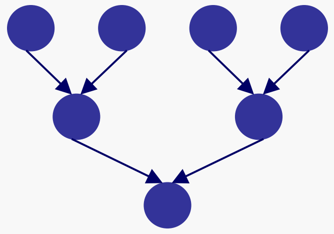
Data Parallelism withParagraph

Task Factories for common parallel patterns are provided.

- Map
  - Apply work function to each element of input view.
- Reduce
  - Apply binary reduction operator to input view.
- Map-Reduce
  - Reduction performed on results of map operation.
- Prefix Scan
  - Produces task graph for prefix sum algorithm.

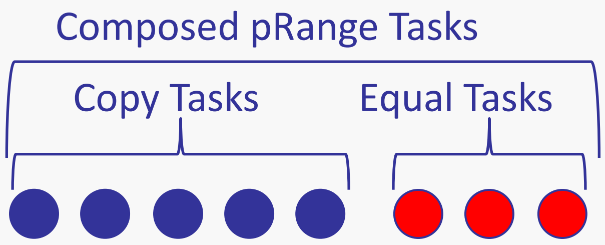
```
pContainer c
make_reduce(c.view(), plus())
```

New patterns can be added as needed.



Task Parallelism with Paragraph

- Task parallelism achieved through pRange composition.
- Static composition
  - Parallel – execution of pRanges' tasks are interleaved arbitrarily
  - Sequential – tasks of first pRange executed before tasks of second



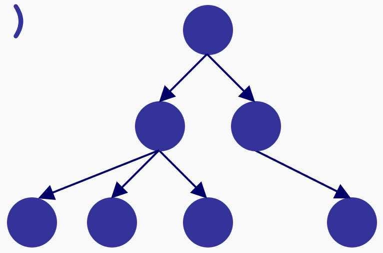
```
View source, destination
View friends, neighbors
parallel(map(copy(), source.view(), destination.view()),
         map(equal(), friends.view(), neighbors.view()))
```

- Dynamic composition
  - Allows conditional and repeated execution of pRanges
  - Repeat-until – process the pRange until the condition given is false
  - If-then-else – process one of two pRanges based on a condition
  - Switch – process one pRange from a set based on an input value

Dynamic Task Graphs

- Factories can't express all parallel computation efficiently.
  – Graph Traversal: complete specification requires doing traversal.
  – Work List Algorithms: new elements for processing added by tasks.
- pRange allows addition of tasks during execution.
  – Only one task has to be specified initially to begin computation.
  – New tasks can depend on tasks already present in the pRange.

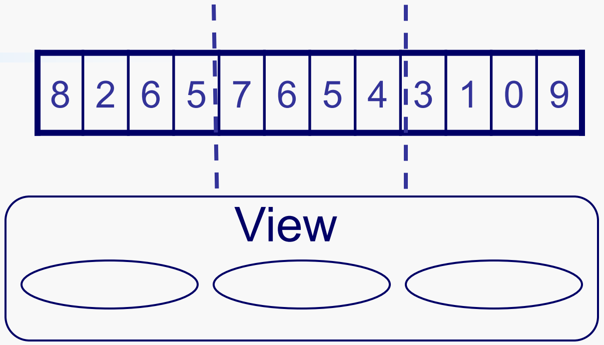
```
stapl_main()  {
  pGraph g;
  //initialize g;
  make_dynamic(g.view(), traverse_from_vertex, vertex(0))
}

traverse_from_vertex(Vertex v) {
  //process data stored in vertex
  foreach (edge in v.outgoing_edges())
    prange.add_task(edge.destination, traverse_from_vertex)
}
```



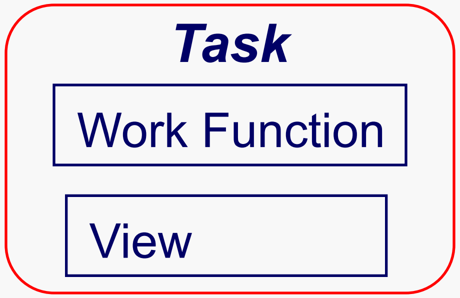
Paragraph in the STAPL Programming Model

- Allows concise specification of parallel algorithms
  - Factories for common computation patterns provided.
  - New factories leverage entire task graph implementation.
  - Dynamic tasks available when capturing pattern in a factory is expensive/impossible.
- Avoids performance penalty of abstract data access
  - Task placement maximizes locality of data to process.
  - Result forwarding is efficient placement of new data with the tasks that need it.
- Developer focused on algorithm operations instead of task placement.

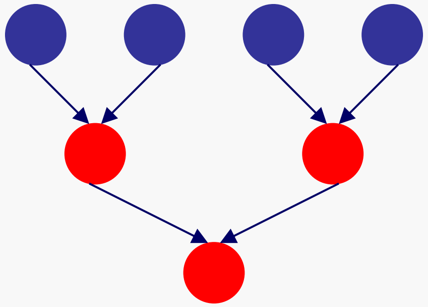
pRange -- Task Graphs in STAPL



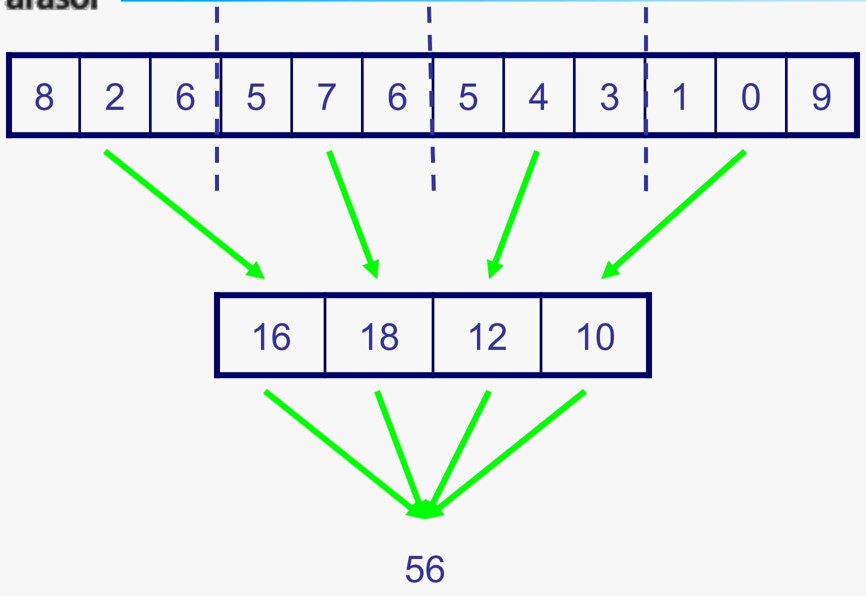
- Data to be processed by pAlgorithm
  - View of input data
  - View of partial result storage
- Work Function
  - Sequential operation
  - Method to combine partial results
- Task
  - Work function
  - Data to process



- Task dependencies
  - Expressed in Task Dependence Graph (TDG)
  - TDG queried to find tasks ready for execution



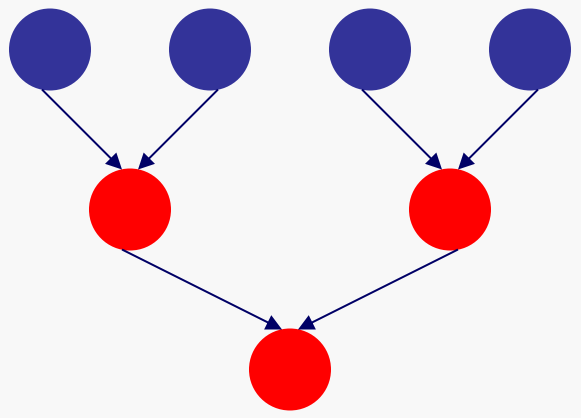
Task graph of pAlgorithm



A task is a work function and the set of data to process.

● = Find sum of elements

● = Combine partial results

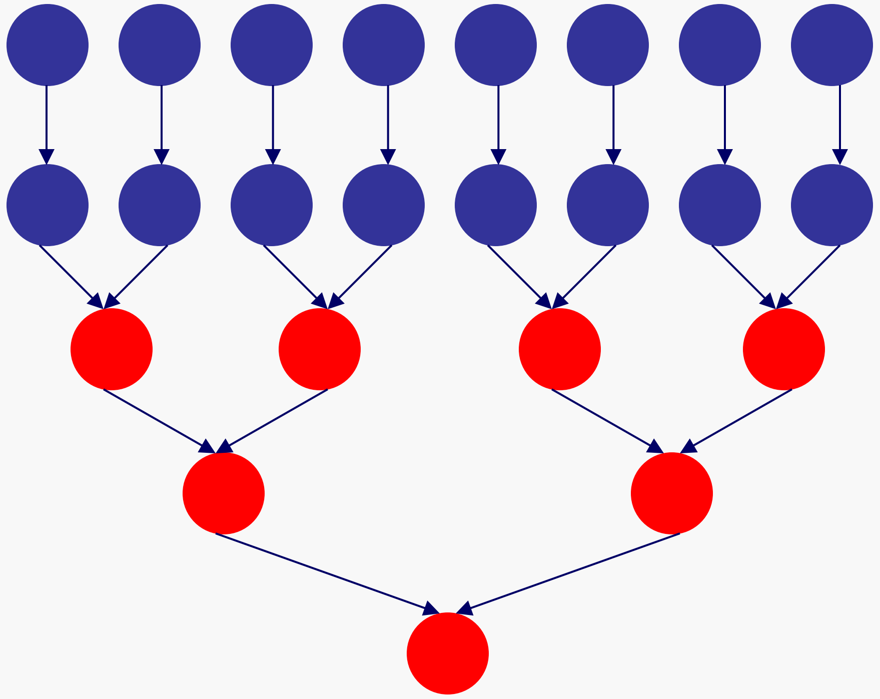


Tasks aren't independent.

Dependencies specify execution order of tasks.

Composing Task Graphs

- Increases amount of concurrent work available
- Forms a MIMD computation
- Dependencies between tasks specified during composition



● = Add 7 to each element

● = Find sum of elements

🔴 = Combine partial results

Dependencies only needed if tasks process the same data

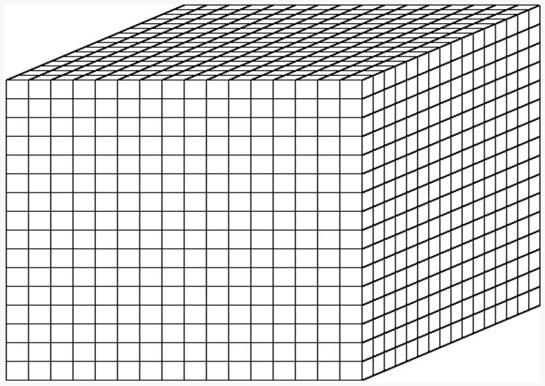
Simple Dependence Specification

- Goal: Developer concisely expresses dependencies
  - Enumeration of dependencies is unmanageable

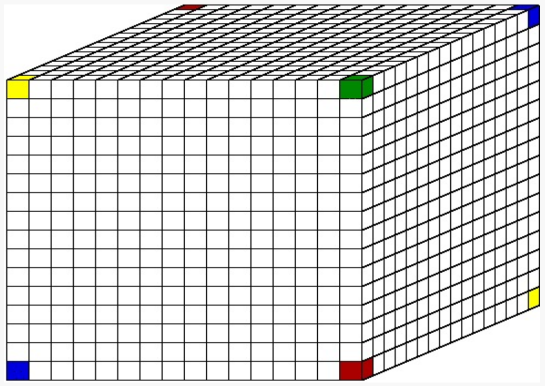
- Common patterns will be supported in pRange
  - Sequential – sources depend on sinks
  - Independent – no new dependencies needed in composed graph
  - Pipelined – dependencies follow a regular pattern

Discrete Ordinates Particle Transport Computation

- Important application for DOE
  - E.g., Sweep3D and UMT2K
- Large, on-going DOE project at TAMU to develop application in STAPL (TAXI)

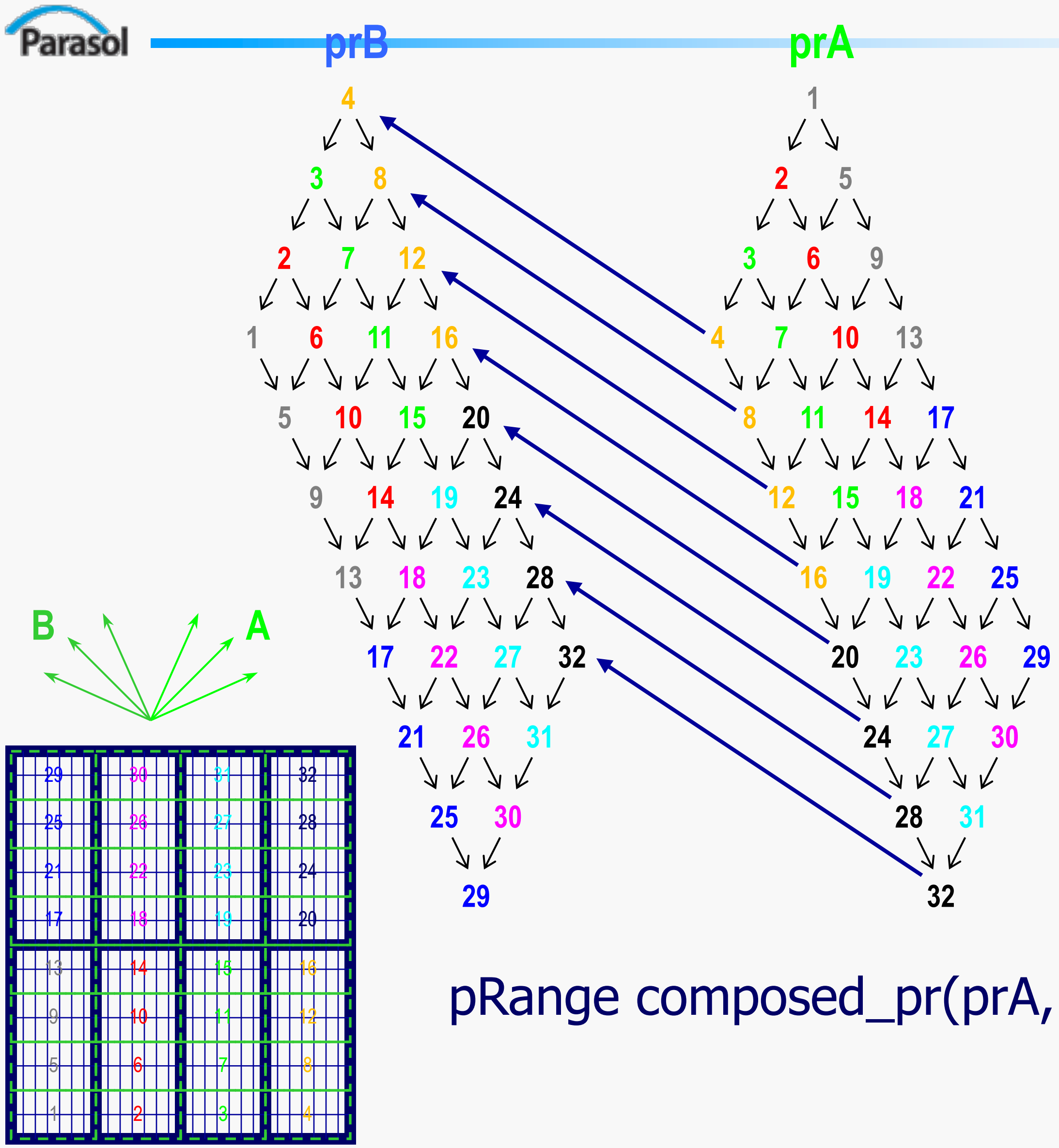


One sweep



Eight simultaneous sweeps

Pipeline Pattern Example



- pRanges are sweeps in particle transport application
- Reflective materials on problem boundary create dependencies
- Pipeline pattern will allow easy composition

pRange composed_pr(prA, prB, Pipeline(4,32,4));

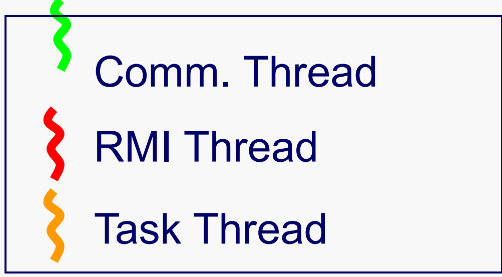
Paragraph Summary

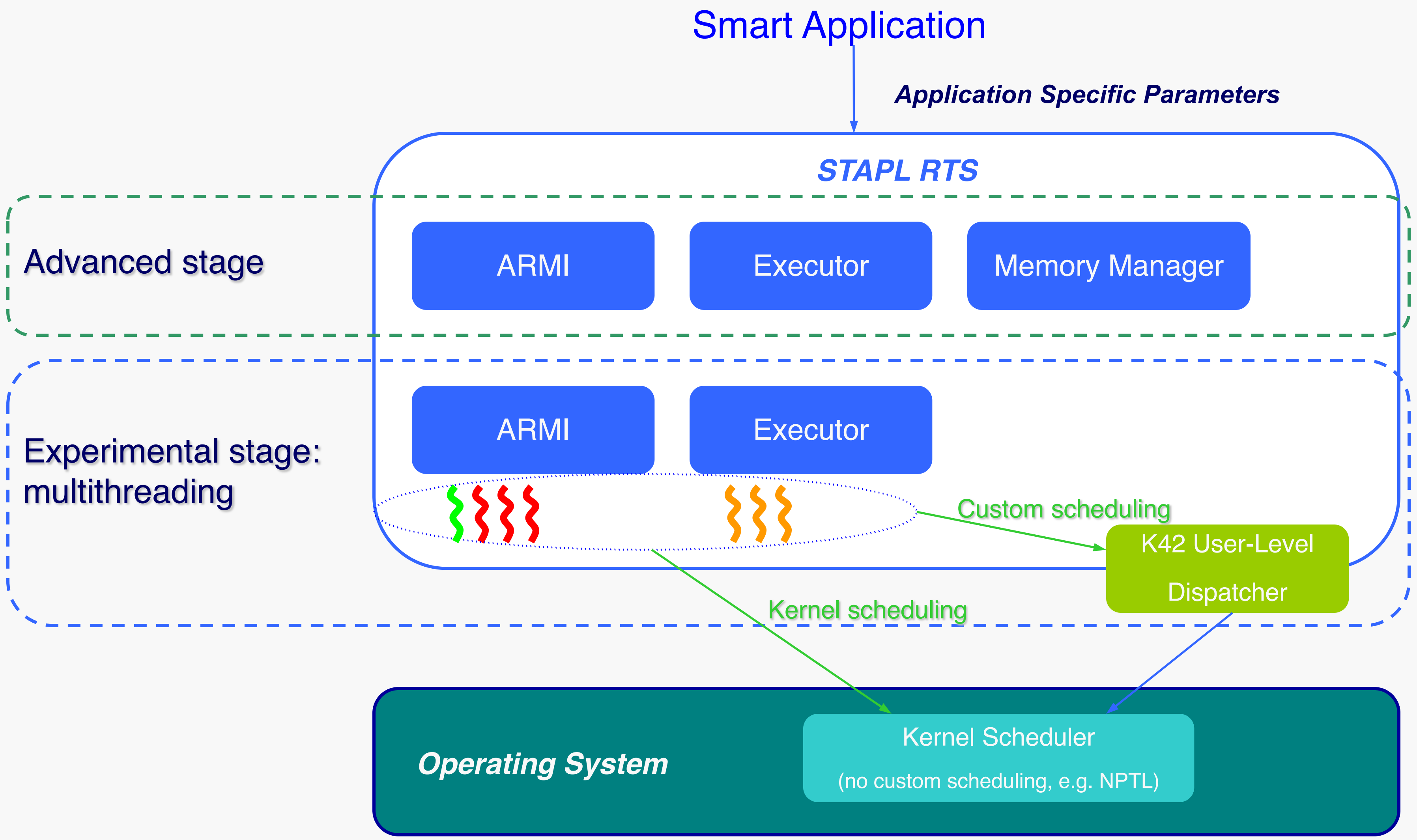
- Binds the work of an algorithm to the data

- **Simplifies** programming task graphs
  - Methods to create tasks
  - Common dependence pattern specifications
  - Compact specification of task dependencies
  - Manages task refinement
  - Simple specification of task graph composition

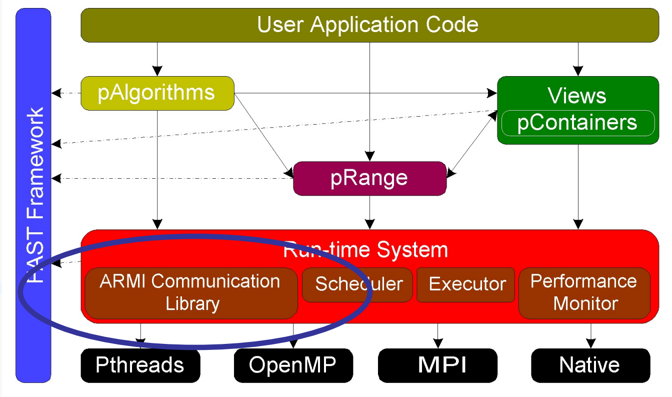
- Supports multiple programming models
  - Data-parallelism
  - Task-parallelism

RTS – Current state





ARMI – Current State



ARMI: Adaptive Remote Method Invocation

– Abstraction of shared-memory and message passing communication layer (MPI, pThreads, OpenMP, mixed, Converse).

– Programmer expresses fine-grain parallelism that ARMI adaptively coarsens to balance latency versus overhead.

– Support for sync, async, point-to-point and group communication.

– Automated (de)serialization of C++ classes.

ARMI can be as easy/natural as shared memory and as efficient as message passing.

ARMI Communication Primitives

Point to Point Communication

armi_async - non-blocking: doesn't wait for request arrival or completion.

armi_sync - blocking and non-blocking versions.

Collective Operations

armi_broadcast, armi_reduce, etc.

can adaptively set groups for communication.

Synchronization

armi_fence, armi_barrier - fence implements distributed termination algorithm to ensure that all requests sent, received, and serviced.

armi_wait - blocks until at least at least one request is received and serviced.

armi_flush - empties local send buffer, pushing outstanding to remote destinations.

RTS – Multithreading (ongoing work)

In ARMI

- Specialized communication thread dedicated the emission and reception of messages
  - Reduces latency, in particular on SYNC requests
- Specialized threads for the processing of RMIs
  - Uncovers additional parallelism (RMIs from different sources can be executed concurrently)
  - Provides a suitable framework for future work on relaxing the consistency model and on the speculative execution of RMIs

In the Executor

- Specialized threads for the execution of tasks
  - Concurrently execute ready tasks from the DDG (when all dependencies are satisfied)

RTS Consistency Models

- Processor Consistency (default)
  - Accesses from a processor on another's memory are sequential
  - Requires in-order processing of RMIs
    - Limited parallelism
- Object Consistency
  - Accesses to different objects can happen out of order
  - Uncovers fine-grained parallelism
    - Accesses to different objects are concurrent
    - Potential gain in scalability
  - Can be made default for specific computational phases
- Mixed Consistency
  - Use Object Consistency on select objects
    - Selection of objects fit for this model can be:
      - Elective – the application can specify that an object's state does not depend on others' states.
      - Detected – if it is possible to assert the absence of such dependencies
  - Use Processor Consistency on the rest

RTS Executor

Customized task scheduling

- Executor maintains a ready queue (all tasks for which dependencies are satisfied in the DDG)
- Order tasks from the ready queue based on a scheduling policy (e.g. round robin, static block or interleaved block scheduling, dynamic scheduling …)
- The RTS decides the policy, but the user can also specify it himself
- Policies can differ for every pRange

Customized load balancing

- Implement load balancing strategies (e.g. work stealing)
- Allow the user to choose the strategy
- K42 : generate a customized work migration manager

RTS Synchronization

- Efficient implementation of synchronization primitives is crucial
  - One of the main performance bottlenecks in parallel computing
  - Common scalability limitation

Fence
  - Efficient implementation using a novel Distributed Termination Detection algorithm

Global Distributed Locks
  - Symmetrical implementation to avoid contention
  - Support for logically recursive locks (required by the compositional SmartApps framework)

Group-based synchronization
  - Allows efficient usage of ad-hoc computation groups
  - Semantic equivalent of the global primitives
  - Scalability requirement for large-scale systems

Productivity

- Implicit parallelism
- Implicit synchronizations/communications
- Composable (closed under composition)
- Reusable (library)
- Tunable by experts (library not language)
- Compiles with any C++ compiler (GCC)
- Optionally exposes machine info.
- Shared Memory view for user
- High level of abstraction – Generic Programming

Performance

- Latency reduction: Locales , data distribution
- Latency Hiding: RMI, multithreading, Asynch Communications
- Optionally exposes machine info.
- Manually tunable for experts
- Adaptivity to input and machine (machine learning)

Portability

- Library – no need for special compiler
- RTS needs to be ported – not much else
- High level of abstraction

References

Cilk

- http://supertech.csail.mit.edu/cilk/
- http://supertech.csail.mit.edu/cilk/manual-5.4.3.pdf
- Dag-Consistent Distributed Shared Memory, Blumofe, Frigo, Joerg, Leiserson, and Randall, In 10th International Parallel Processing Symposium (IPPS '96), April 15-19, 1996, Honolulu, Hawaii, pp. 132-141.

TBB

- http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm
- TBB Reference Manual – provided with package
- http://idfemea.intel.com/2006/prague/download/SShah_IDF_Keynote_2006-10-03_v1.pdf

STAPL

- http://parasol.tamu.edu/groups/rwergergroup/research/stapl/

References

- HPF
  - HPFF Homepage - http://hpff.rice.edu/
  - High Performance Fortran: history, overview and current developments. H Richardson, Tech. Rep. TMC-261, Thinking Machines Corporation, April 1996.
  - http://www.nbcs.rutgers.edu/hpc/hpf{1,2}/
- Chapel
  - http://chapel.cs.washington.edu/
  - Chapel Draft Language Specification. http://chapel.cs.washington.edu/spec-0.702.pdf
  - An Introduction to Chapel: Cray's High-Productivity Language. http://chapel.cs.washington.edu/ChapelForAHPCRC.pdf

References

- Fortress
  - http://research.sun.com/projects/plrg
  - Fortress Language Specification. http://research.sun.com/projects/plrg/fortress.pdf
  - Parallel Programming and Parallel Abstractions in Fortress. Guy Steele. http://irbseminars.intel-research.net/GuySteele.pdf
- Stapl
  - http://parasol.tamu.edu/groups/rwergergroup/research/stapl
  - A Framework for Adaptive Algorithm Selection in STAPL, Thomas, Tanase, Tkachyshyn, Perdue, Amato, Rauchwerger, In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, pp. 277-288, Chicago, Illinois, Jun 2005.

Table of Contents

UPC

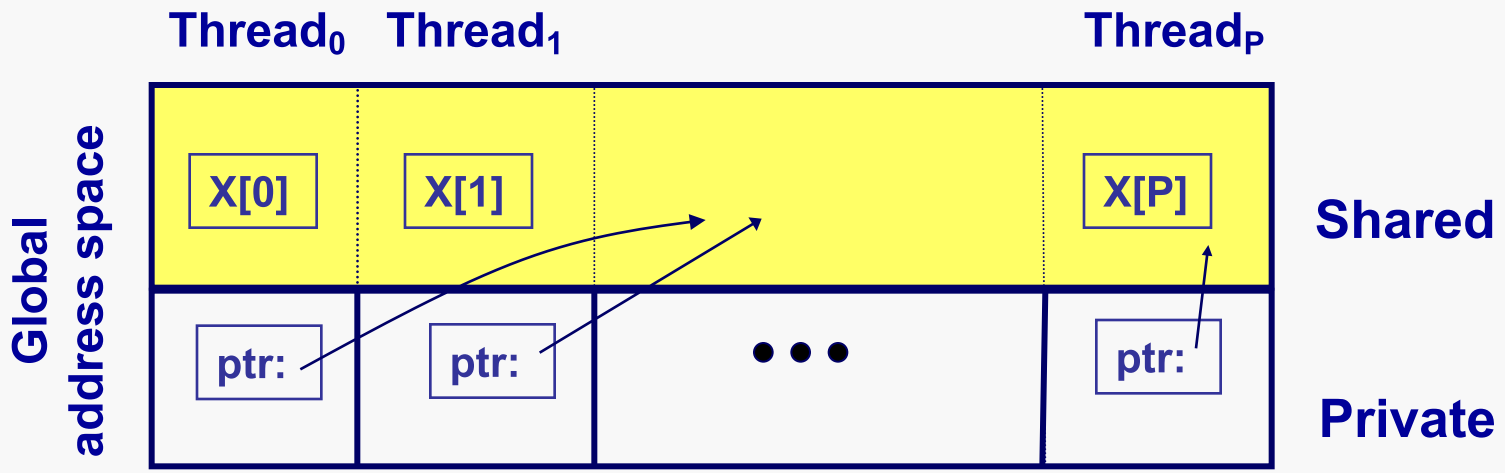
- Unified Parallel C
- An explicit parallel extension of ISO C
- A partitioned shared memory parallel programming language
- Similar to the C language philosophy
  - Programmers are clever

Adapted from http://www.upc.mtu.edu/SC05-tutorial

Execution Model

- UPC is SPMD
  - Number of threads specified at compile-time or run-time;
  - Available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (`0..THREADS-1`)
- There are two compilation modes
  - Static Threads mode:
    - THREADS is specified at compile time by the user
    - THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

UPC is PGAS



- The languages share the global address space abstraction
  - Programmer sees a single address space
  - Memory is logically partitioned by processors
  - There are only two types of references: local and remote
  - One-sided communication

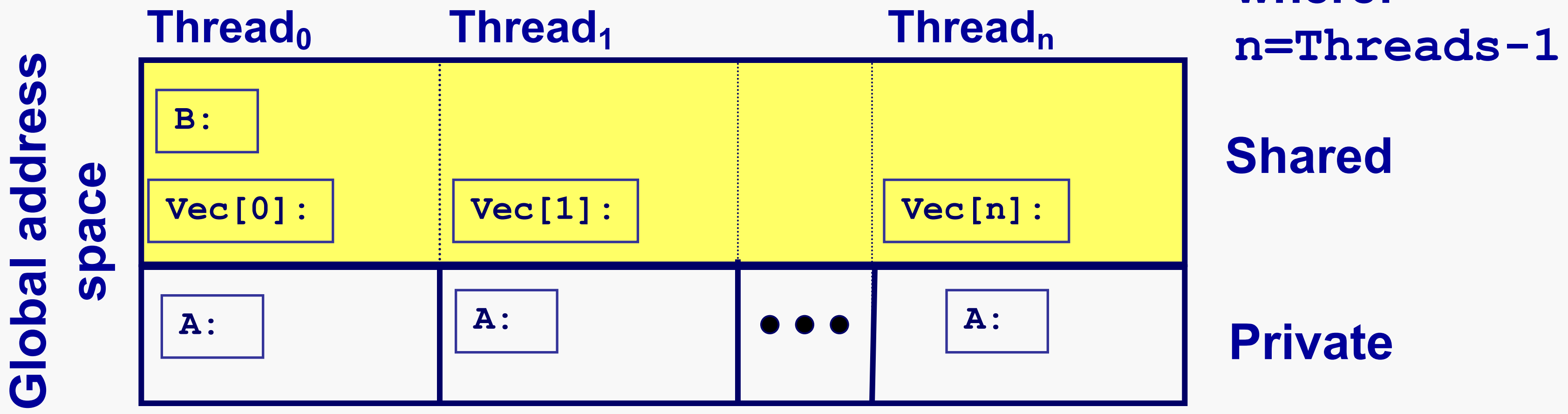
Hello World

- Any legal C program is also a legal UPC program
- UPC with P threads will run P copies of the program.
- Multiple threads view

```
#include <upc.h>  /* UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC
  world\n", \
          MYTHREAD, THREADS);
}
```

Private vs. Shared Variables

- Private scalars (`int A`)
- Shared scalars (`shared int B`)
- Shared arrays (`shared int Vec[TREADS]`)
- Shared Scalars are always in threads 0 space
- A variable local to a thread is said to be **affine** to that thread



Data Distribution in UPC

- Default is cyclic distribution
  - `shared int V1[N]`
  - Element `i` affine to thread `i%THREADS`
- Blocked distribution can be specified
  - `shared [K] int V2[N]`
  - Element `i` affine to thread `(N/K)%THREADS`
- Indefinite ()
  - `shared [0] int V4[4]`
  - all elements in one thread
- Multi dimensional are linearized according to C layout and then previous rules applied

Work Distribution in UPC

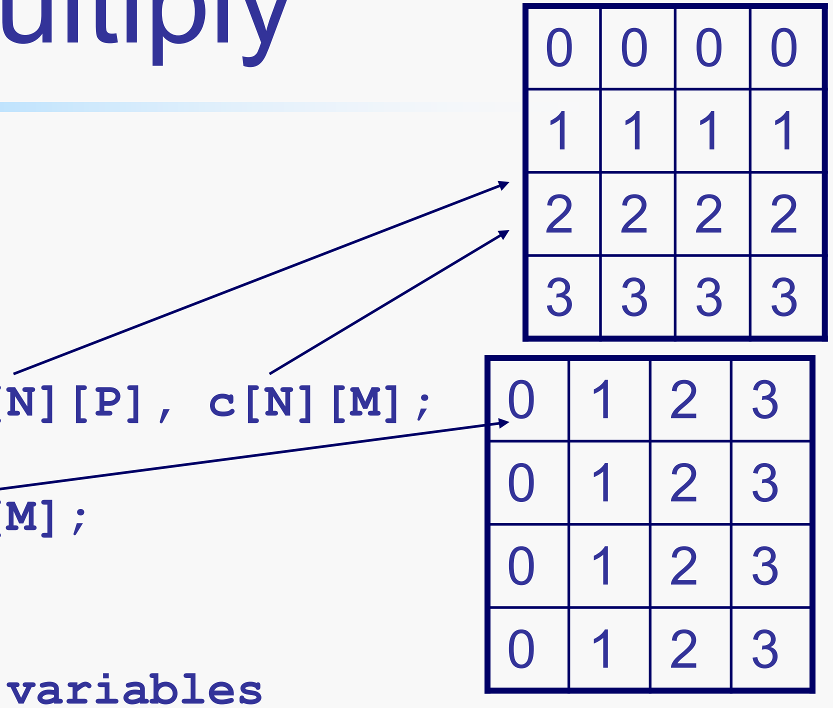
- UPC adds a special type of loop

```
    upc_forall(init; test; loop; affinity)
        statement;
```

- Affinity does not impact correctness but only performance
- Affinity decides which iterations to run on each thread. It may have one of two types:
  - Integer: `affinity%THREADS` is `MYTHREAD`
  - E.g., `upc_forall(i=0; i<N; i++; i)`
  - Pointer: `upc_threadof(affinity)` is `MYTHREAD`
  - E.g., `upc_forall(i=0; i<N; i++; &vec[i])`

UPC Matrix Multiply

```
#define N 4
#define P 4
#define M 4
// Row-wise blocking:
shared [N*P/THREADS] int a[N][P], c[N][M];
// Column-wise blocking:
shared[M/THREADS] int b[P][M];

void main (void) {
  int i, j , l; // private variables

  upc_forall(i = 0 ; i<N ; i++; &c[i][0])
    for (j=0 ; j<M ;j++) {
      c[i][j] = 0;
      for (l= 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b[l][j];
    }
}
```



Replicating **b** among processors would improve performance

Synchronization and Locking

- Synchronization
  - Barrier: block until all other threads arrive
    - `upc_barrier`
  - Split-phase barriers
    - `upc_notify` this thread is ready for barrier
    - `upc_wait` wait for others to be ready
- Locks: `upc_lock_t`
  - Use to enclose critical regions
    - `void upc_lock(upc_lock_t *l)`
    - `void upc_unlock(upc_lock_t *l)`
  - Lock must be allocated before use

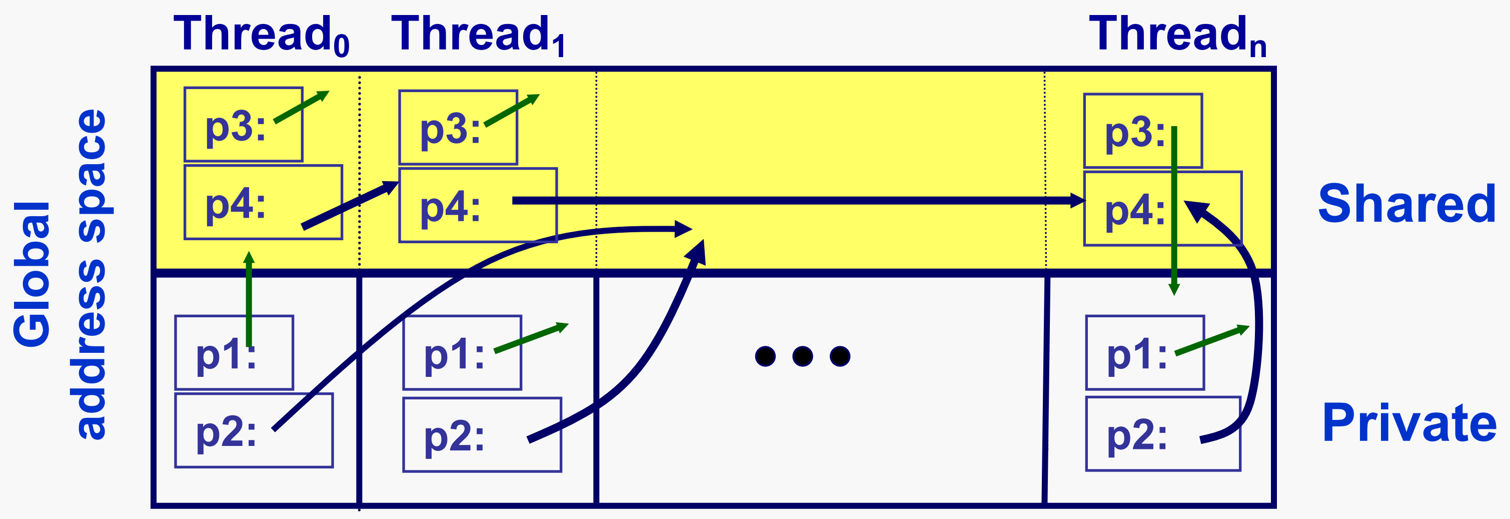
Collectives

- Must be called by all the threads with same parameters
- Two types of collectives
  - Data movement: scatter, gather, broadcast,…
  - Computation: reduce, prefix, …
- When completed the threads are synchronized
- E.g.,

```
res=bupc_allv_reduce(int, in, 0, UPC_ADD);
```

Type Input Thread Operation

UPC Pointers



```
int *p1;             /* private pointer to local memory */
shared int *p2; /* private pointer to shared space */
int *shared p3; /* shared pointer to local memory */
shared int  *shared p4; /* shared pointer to
                           shared space */
```

- Pointers-to-shared are more costly to dereference
- The use of shared pointers to local memory are discouraged

Memory Consistency

- UPC has two types of accesses:
  - Strict: Will always appear in order
  - Relaxed: May appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:

    ```
    #include <upc_relaxed.h>
    ```

  - All accesses in the program unit relaxed by default
  - Use strict on variables that are used as synchronization (`strict shared int flag;`)

    ```
    data = …                    while (!flag) { };
    flag = 1;                   … = data;   // use the data
    ```

Additional Features

- Latency management: two levels of proximity exposed to the user
- Portability: UPC compilers are available for many different architectures
- Productivity: UPC is a low-level language, the main objective is performance

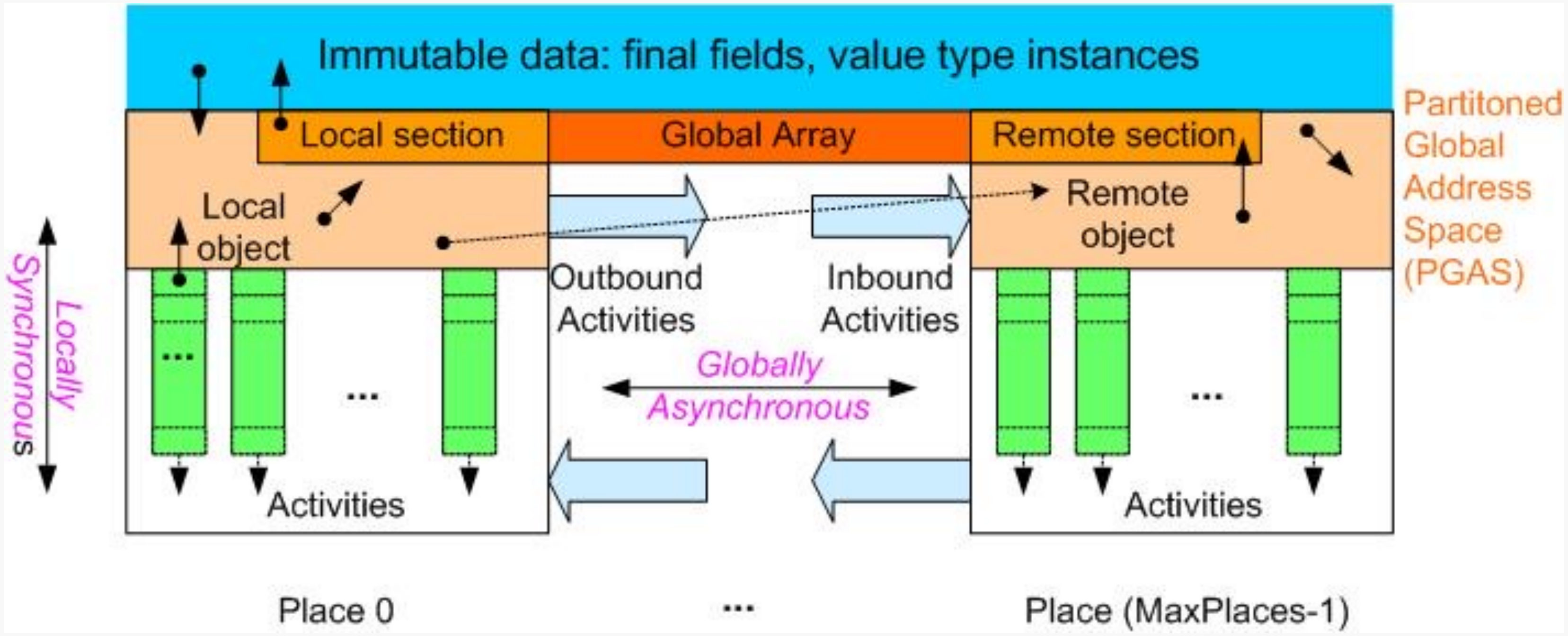
Table of Contents

X10

- Developed by IBM as part of DARPA HPCS
- Draws from Java syntax and arrays in ZPL
- Partitioned Global Address Space (PGAS)
- Clocks - generalized barrier synchronization
- Constructs for explicit data and work placement

Adapted from presentations at: http://x10.sourceforge.net/tutorial/presentations

The X10 Model



Place - collection of resident activities & objects
(e.g., SMP node of cluster).

Activities - lightweight thread of execution.

Locality Rule
Access to data must be performed by a local activity.
Remote data accessed by creating remote activities

Ordering Constraints (Memory Model)
Locally Synchronous:
Guaranteed coherence for local heap.
Strict, near sequential consistency.

Globally Asynchronous:
No ordering of inter-place activities.
Explicit synchronization for coherence.

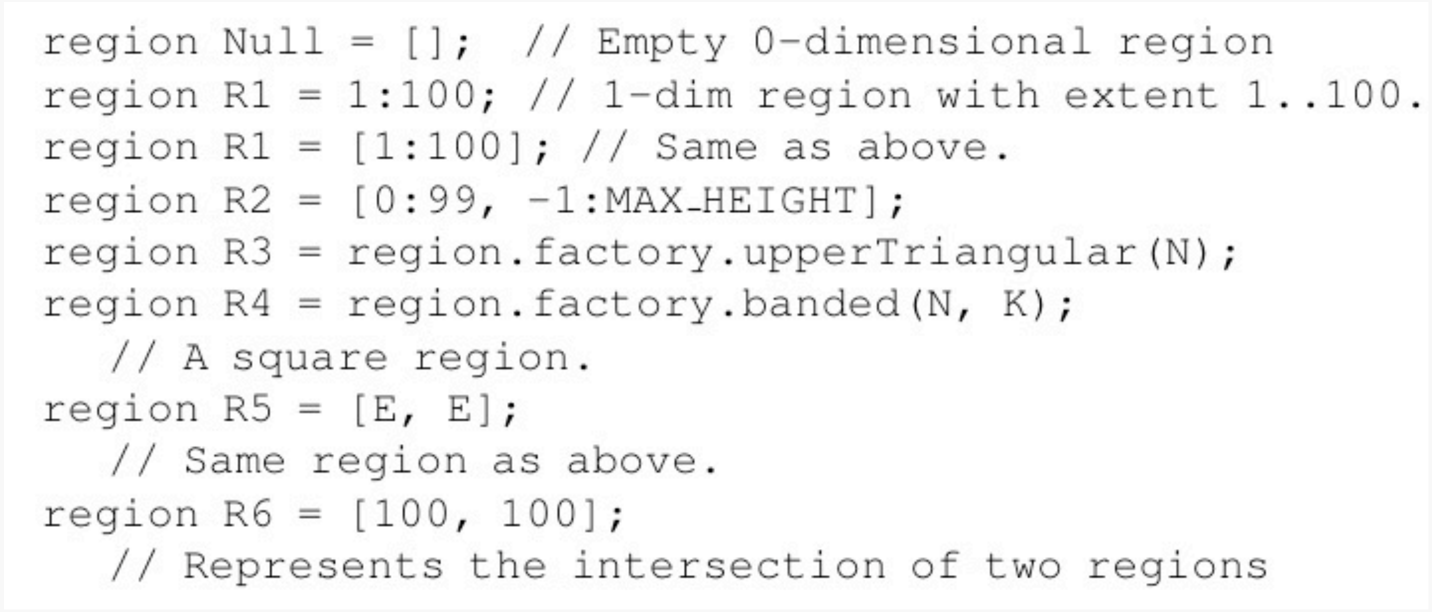
The X10 Model

Execution Model

- Explicit data parallelism, `foreach`
- Explicit task parallelism `future, async`
- Explicit, asynchronous, one-sided communication with `future`
- Explicit synchronization
  - `clock`, `finish`, `future`, `atomic` section (within a place)
- Multi-level memory model under development
  - Within a place - more strict, not quite sequential consistency
  - Across places - relaxed, explicit synchronization required

X10 - Regions

- Defines a set of *points* (indices)
  - Analogous to Chapel domains
  - User defined regions in development



```
region Null = [];  // Empty 0-dimensional region
region R1 = 1:100; // 1-dim region with extent 1..100.
region R1 = [1:100]; // Same as above.
region R2 = [0:99, -1:MAX_HEIGHT];
region R3 = region.factory.upperTriangular(N);
region R4 = region.factory.banded(N, K);
   // A square region.
region R5 = [E, E];
   // Same region as above.
region R6 = [100, 100];
   // Represents the intersection of two regions
```

X10 - Distributions

- Maps every point in a region to a place
  - Analogous to Chapel distributed domains
  - User distributions regions in development

```
dist D1 = dist.factory.constant(R, here);//maps region R to local place
dist D2 = dist.factory.block(R);         //blocked distribution
dist D3 = dist.factory.cyclic(R);        //cyclic distribution
dist D4 = dist.factory.unique();         //identity map on [0:MAX_PLACES-1]

double[D] vals;
vals.distribution[i] //returns place where ith element is located.
```

X10 - Data Parallelism

`[finish] foreach(i : Region) S`

Create a new activity at place P for each point in Region and execute statement S. Finish forces termination synchronization.

```
public class HelloWorld2 {
  public static void main(String[] args) {
    foreach (point [p] : [1:2])
      System.out.println("Hello from activity " + p + "!");
  }
}
```

X10 - Data Parallelism

`[finish] ateach(i : Distribution) S`

Create a new activity at each point in Region at the place where it is mapped in the Distribution. Finish forces termination synchronization.

```
public class HelloWorld2 {
  public static void main(String[] args) {
      ateach (place p: dist.factory.unique(place.MAX_PLACES))
      System.out.println("Hello from place " + p + "!");
  }
}
```

X10 - Task Parallelism

`[finish] async(P) S`

Create a new activity at place P, that executes statement S.

```
//global array
double a[100] =  …;
int k = …;

async (3) {
    // executed place 3
    a[99] = k;
}

//continue without waiting
```

```
//global array
double a[100] =  …;
int k = …;

finish async (3) {
    // executed place 3
    a[99] = k;
}

//wait for remote completion
```

X10 - Task Parallelism

`future(P) S`

Similar to **`async`**, *returns result from remote computation.*

```
// global array
final double a[100] =  …;
final int idx = …;

future<double> fd =
  future (3)
  {
    // executed at place 3
      a[idx];
  };

int val = fd.force(); //wait for fd
completion
```

X10 - Synchronization

- Atomic block
  - conceptually executed in a single step while other activities are suspended
  - must be nonblocking, no task spawning (e.g., no communication with another place)

```
// push data onto concurrent
// list-stack
Node node = new Node(data);
atomic {
     node.next = head;
     head = node;
}
```

X10 - Synchronization

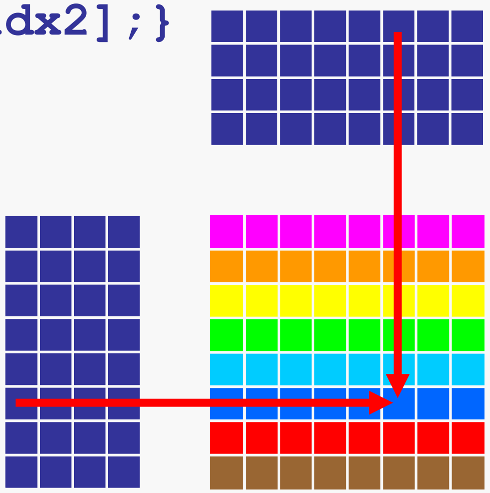
- Clocks
  - Generalization of barrier
    - Defines program phases for a group of activities
    - Activities cannot move to next phase until all have acquiesced with a call to **`next`**
  - Activities can register with multiple clocks
  - Guaranteed to be deadlock free
  - **`next, suspend, resume, drop`**

X10 - Synchronization

```
final clock c = clock.factory.clock();
foreach (point[i]: [1:N]) clocked (c) {
    while ( true ) {
      //phase 1
      next;
      //phase 2
      next;
      if ( cond )
         break;
    } // while
 } // foreach
 c.drop();
```

X10 - Matrix Multiply

```
double[.] A = new double[D1]; //defined on Region R1
double[.] B = new double[D2]; //defined on Region R2
double[.] C = new double[D3]; //defined on Region R3
...
finish ateach(point ij : D3) {
   for(point k : R1[1]) {
     point idx1 = new point(ij[0],k);
     point idx2 = new point(k, ij[1]);
     future<double> a(A[idx1].location) {A[idx1];}
     future<double> b(B[idx2].location) {B[idx2];}
     C[i] += a.force() * b.force();
   }
}
```



X10 - Productivity

- New programming language based on Java
- Abstraction
  - Relatively low for communication and synchronization
  - Transparency was a design goal
- Component reuse
  - Java style OOP and interfaces
  - Generic types and type inference under development

X10 - Productivity

- Tunability
  - Implementation refinement via Distributions and work placement
- Defect management
  - Reduction with garbage collection
  - Detection and removal with integration with Eclipse toolkit
- Interoperability
  - C library linkage supported, working on Java

X10 - Performance

- Latency Management
  - Reducing
    - Data placement - distributions.
    - Work placement - `ateach, future, async`
  - Hiding
    - Asynchronous communication with `future`
    - Processor virtualization with activities
- Load Balancing
  - Runtime can schedule activities within a place

X10 - Portability

- Language based solution, requires compiler
- Runtime system not discussed. Must handle threading and communication - assumed to be part of model implementation
- **places** machine information available to programmer
- Parallel model not effected by underlying machine
- I/O not addressed in standard yet

References

- UPC
  - http://upc.gwu.edu/
  - http://www.upc.mtu.edu/SC05-tutorial
- X10
  - http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html
  - http://x10.sourceforge.net/tutorial/presentations

Table of Contents

Linda

- History
  - Developed from 1992 by N. Carriero and D. Gelernter
  - A Commercial version is provided by Scientific Computing Associates, Inc.
  - Variations: TSpace (IBM), JavaSpaces (SUN)
- Programming Style
  - Processes creation is implicit
  - Parallel processes operate on objects stored in and retrieved from a shared, virtual, associative memory (**Tuple Space**)
  - Producer-Consumer approach

Adapted from http://www.lindaspaces.com/teachingmaterial/LindaTutorial_Jan2006.pdf

Linda

- Productivity
  - Linda extends traditional languages (C, Java,…)
  - The abstraction provided is intuitive for some class of problems
  - Object stored in the Tuple Space has a global scope: the user have to take care of associates the right keys
- Portability
  - Tuple Space has to be implemented
  - Code analysis is architecture dependent
  - If objects in the shared space contains references to values a shared memory has to be provided

Linda

- Performance
  - Depends on Tuple Space implementation
    - Architecture is hidden to the user
  - Code analysis can provide optimizations
- Defect analysis
  - Commercial implementation provides debuggers and profilers

Tuple Space

- A Tuple is a sequence of typed fields:
  - (“Linda”, 2, 32.5, 62)
  - (1,2, “A string”, a:20) // array with size
  - (“Spawn”, i, f(i))
- A Tuple Space is a repository of tuples
- Provide:
  - Process creation
  - Synchronization
  - Data communication
  - Platform independence

Linda Operations (read)

- Extraction
  - `in("tuple", field1, field2);`
    - Take and remove a tuple from the tuple space
    - Block if the tuple is not found
  - `rd("tuple", field1, field2);`
    - Take a tuple from the space but don't remove it
    - Block if the tuple is not found
  - `inp`, `rdp`: as in and rd but non-blocking

Linda Operations (write)

- Generation
  - `out("tuple", i, f(i));`
    - Add a tuple to the tuple space
    - Arguments are evaluated before addition
  - `eval("tuple", i, f(i));`
    - A new process compute f(i) and insert the tuple as the function returns
    - Used for process creation

Tuple matching

Parasol

- Tuples are retrieved by matching
  - `out(“Hello”, 100)`
  - `in(“Hello”, 100) // match the tuple`
  - `in(“Hello”, ?i) // i=100`
- Tuples matching is non-deterministic
  - `out(“Hello”, 100)`
  - `out(“Hello”, 99)`
  - `in(“Hello”, ?i) // i=99 or i=100`
- Tuple and template must have the same number of fields and the same types

Atomicity

- The six Linda operations are atomic
  - A simple counter

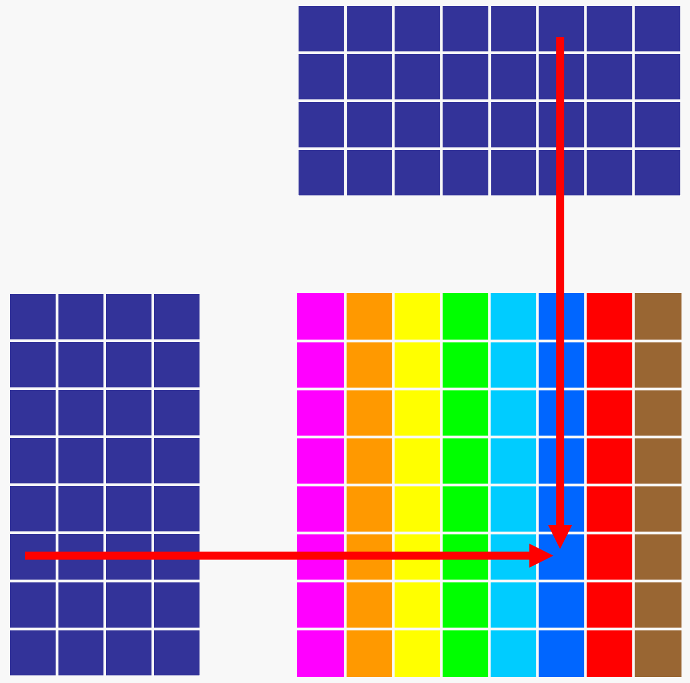
```
in("counter", ?count);
out("counter", count+1);
```

  - The first operation remove the tuple gaining exclusive access to the counter
  - The second operation release the counter

Hello world

```
linda_main(int i) {
   out("count", 0);
   for(int i=1; i<=NUM_PROCS; i++)
       eval("worker",hello_world(i));
   in("count", NUM_PROCS);
   printf("All processes done.\n");
}

void hello_world (int i) {
   int j;
   in("count", ?j); out("count", j+1);
   printf("Hello world from process %d,", i);
   printf(" count %d\n", j);
}
```

Matrix Multiply



```
for(int i=0; i<M; ++i) {
   for(int k=0; k<L; ++k) {
      for(int j=0; j<N; ++j) {
         C[i][j] =
               A[i][k]*B[k][j];
      }
   }
}
```

A parallel specification:

C**$_{ij}$* **is the dot-product of row *i* of *A* and column *j* of *B

Matrix Multiply in Linda

```
Void // Compute C=A*transpose(B)
matrix_multiply(double A[m][n],B[l][n],C[m][l]) {
   for (int i=0; i < m; i++) // Spawn internal products
       for (int j=0; i < l; j++) {
              ID = i*n + j;
              eval("dot", ID, \
                     dot_product(&A[i], &B[j], ID));
       }
   for (int i=0; i < n; i++) // Collect results
       for (int j=0; j < n; j++) {
              ID = i*n + j;
              in("dot", ID, ?C[i][j]);
       }
}
```

Matrix Multiply in Linda (2)

```
double dot_product(double A[n],\
                   double B[n], int ID) {
   // ID is not used in the
   // sequential version of dot_product
   double sum=0;
   for (int i=0; i<m; i++)
                 sum += A[i]*B[i];
   return sum;
}
```

Parallel dot-product

```
double dot_product(double *A, double *B, int ID) {
   double p;
   for (int i=0 ; i < m ; i++)
       eval("p-dot", ID, p_prod(A,B,i*(n/m),(n/m)));
   sum = 0;
   for (int i=0 ; i < m ; i++) {
       in("p-dot", ID, ?p);
       sum += p ;
   }
   return sum ;
}
double p_prod(double *A,double *B,int start, int len) {
   double sum = 0;
   for (int i=start; i < len+start; i++)
       sum += A[i]*B[i];
   return sum;
}
```

Nested Parallelism

- Matrix multiply uses nested parallelism
- Tuples of dot_product have the same types as tuples in matrix_multiply but they have a different string identifier
  - (“dot”, int, double*)
  - (“p-dot”, int, double*)
- Correctness is guaranteed by ID and commutativity of addition

Runtime

- Tuple rehashing
  - Runtime observe patterns of usage, remaps tuple to locations
    - Domain decomposition
    - Result tuples
    - Owner compute
- Long fields handling
  - Usually long fields are not used for mathcing
  - Bulk transfer
- Knowing implementation and architecture details and helps in optimizing user code

Table of Contents

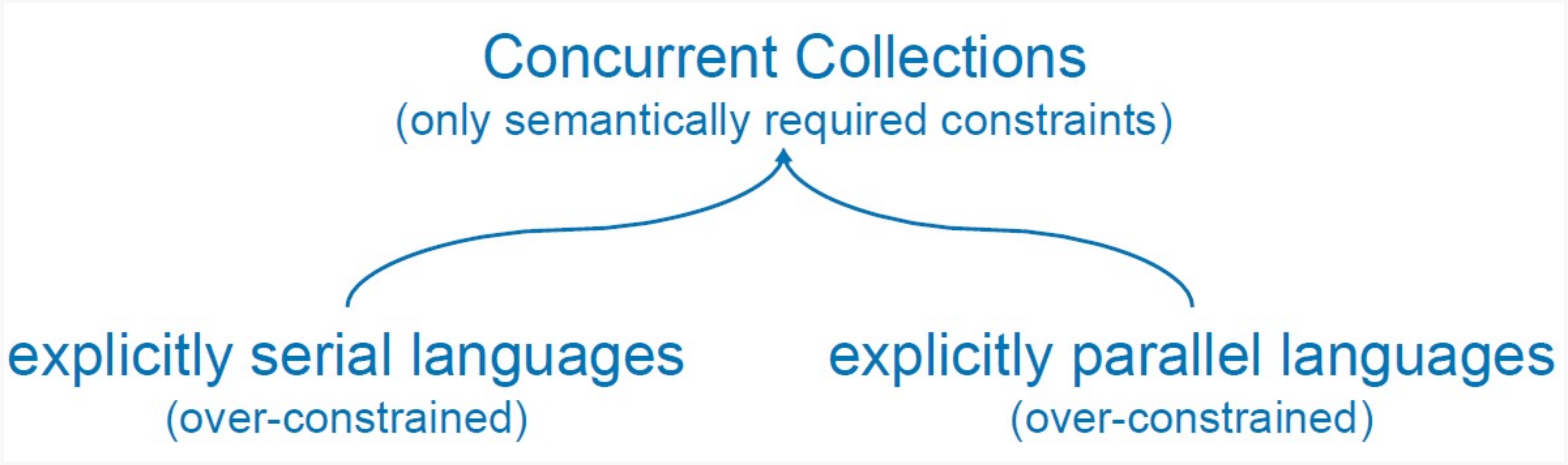
Intel Concurrent Collections

- Parallel language
- Separating the concerns of
  - Domain-expert - semantics of the application
  - Tuning-expert - performance.

- Specifies mechanisms for composing of building blocks
- Building blocks are specified in C++

CnC – the Big Idea

- Separation between
  - Program developer
  - Runtime/compilation support


Concurrent Collections
(only semantically required constraints)
explicitly serial languages
(over-constrained)
explicitly parallel languages
(over-constrained)

Picture from The Concurrent Collections (CnC) Parallel Programming Model – Foundations and Implementation Challenges - Kathleen Knobe, Vivek Sarkar

CnC – the General View

- Similar to streaming languages
  - Implementation of building blocks (**steps**) is independent of the application graph
  - Steps are stateless and mutually independent
- Different from streaming languages
  - Arbitrary (user defined) execution (no FIFO)
  - Order Defines control (tag matching)

Intel CnC Model

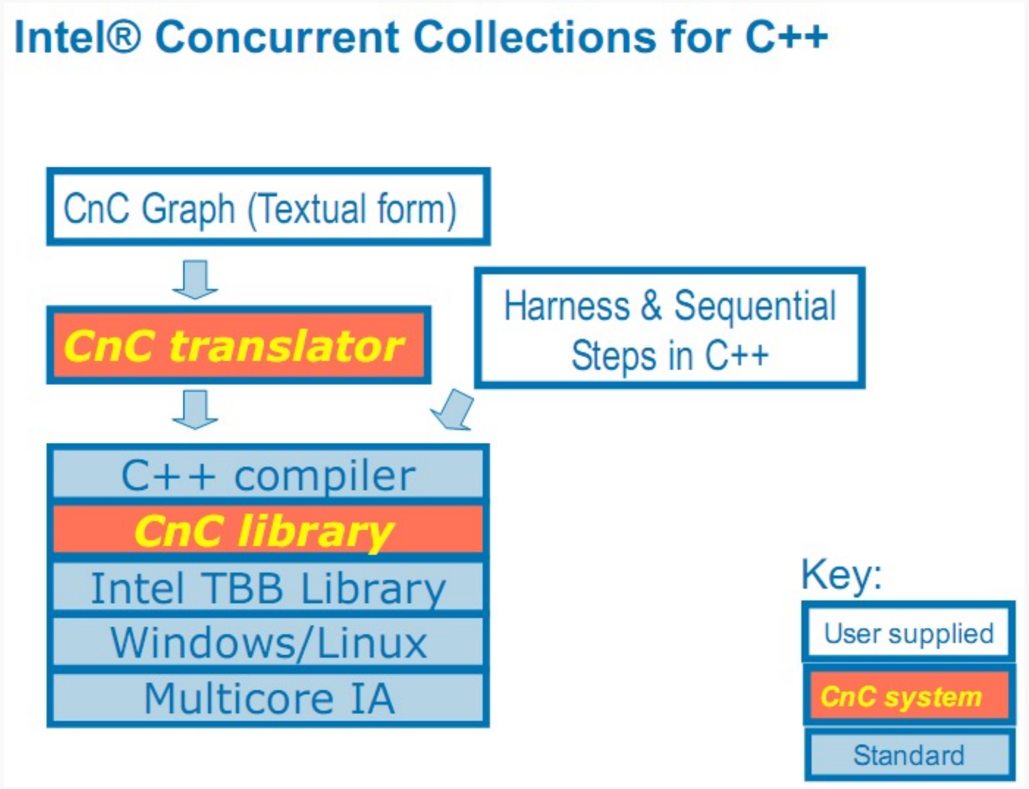
- Execution Model
  - Implicitly parallel for domain expert
  - Race free – computation based on values
  - Performing computations based on matching *tags* and (*item, tag*) pairs
  - Unordered execution on tags sets

Intel CnC Model

- Productivity
  - Higher level operators and data structures
  - Expressing the computation in any serial programming language
    - Intel Concurrent Collections for C++
    - Rice Concurrent Collections for Java
    - Rice Concurrent Collections for .Net
  - Can be used with Visual Studio with Visual Studio Debugger
  - Provide *Tracing Utility*
- Performance
  - Leaving tuning to the tuning expert (person or program)

CnC Model

- Portability
  - Independent of the underlying architecture
  - Targets shared memory systems



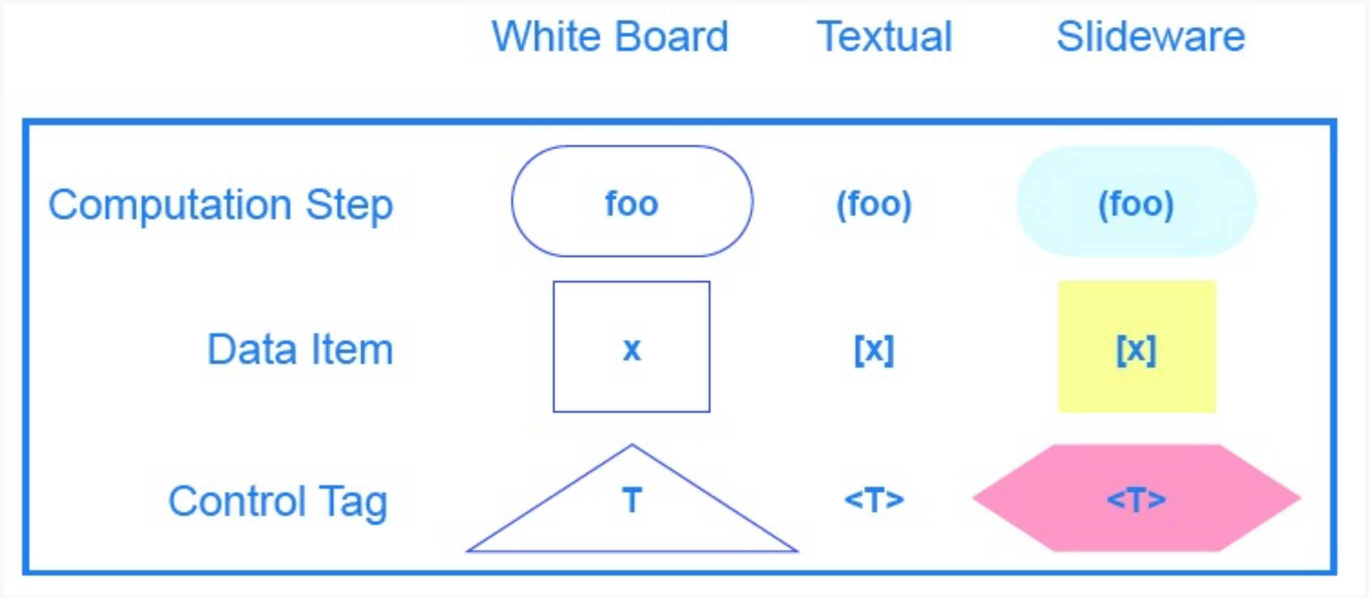
Picture from The Concurrent Collections (CnC) Parallel Programming Model – Foundations and Implementation Challenges - Kathleen Knobe, Vivek Sarkar

CnC - Constructs

- A CnC program consists of
  - **Steps**
    - computation that can run in parallel computation
    - C++ code
  - **Items** – inputs and outputs to/from each step
  - **Tags** – determining instances of *steps* or *items*

- Determines which steps can run in parallel
  - Relieves ordering

CnC - Notation



	White Board	Textual	Slideware
Computation Step	foo	(foo)	(foo)
Data Item	x	[x]	[x]
Control Tag	T	<T>	<T>

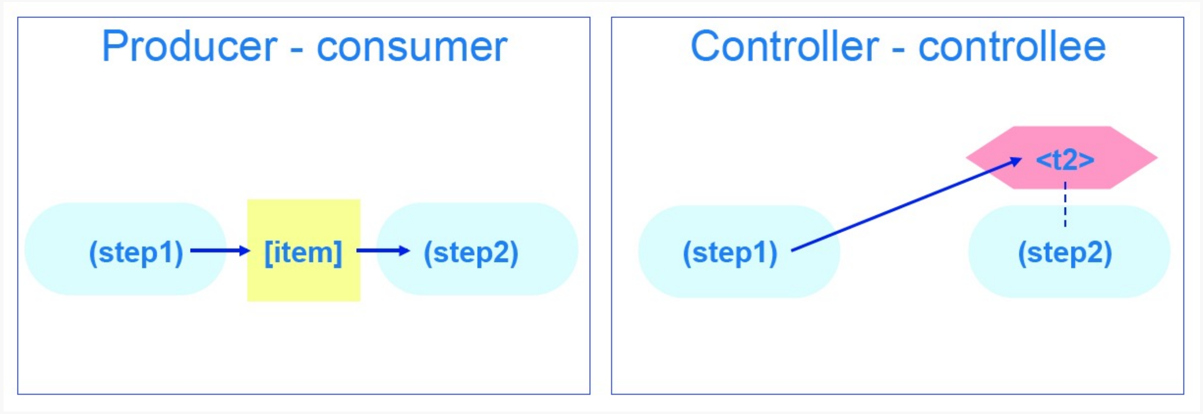
Picture from The Concurrent Collections (CnC) Parallel Programming Model – Foundations and Implementation Challenges - Kathleen Knobe, Vivek Sarkar

CnC – Relationship

- **Consumer** – items are input to a step
- **Producer** – a step creates tags or items
- **Prescription** – determines the number of times a step will run

Two types of ordering

- Producer/consumer – data-ordering
- Controller/controllee – control-ordering



CnC – Fibonacci Example

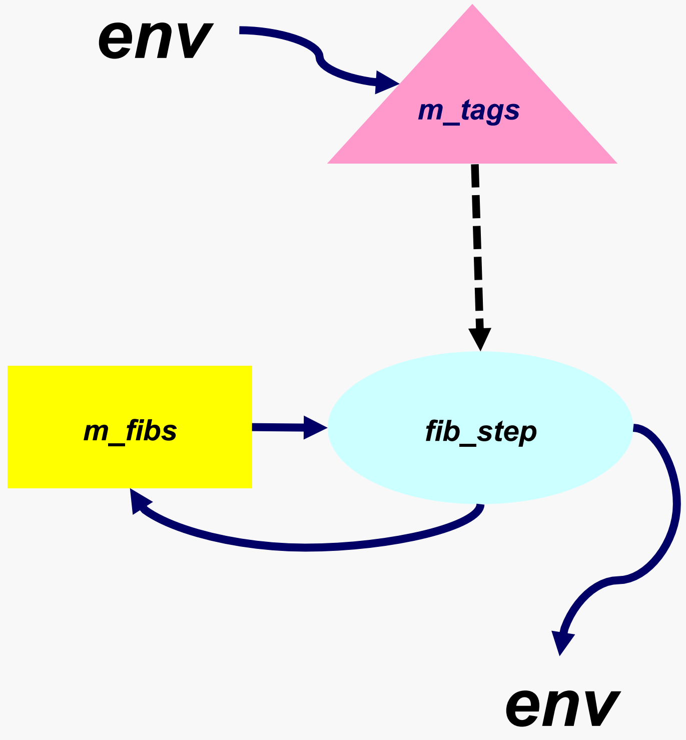
- First step – building the graph
  - tags, steps, and inputs relations

```
//the program receives a number of tags from the environment
<long m_tags>
env -> <m_tags>

//the compute step will receive inputs and tags of type long, respectively,
//and will produce a long elements.

//Remember that fib_step should be implemented in C++ by the user
[long m_fibs <long>]
<m_tags> ::  (fib_step)
[m_fibs] -> (fib_step) -> [m_fibs];

//the program will send the results to the environment (main function)
[m_fibs] -> env;
```

CnC – Fibonacci Example

- Works almost sequentially – due to dependencies
- Tags are created in main function [0, 1, …, n] for fib(n)

```
int fib_step::execute( const long & tag)
     const{
    switch( tag ) {
       //the first two numbers
       case 0 : m_fibs.put( tag, 0 ); break;
       case 1 : m_fibs.put( tag, 1 ); break;
       //for the rest
       default :
           //get from m_fibs two previous no.
           long f_1; m_fibs.get( tag - 1, f_1 );
           long f_2; m_fibs.get( tag - 2, f_2 );
            m_fibs.put( tag, f_1 + f_2 );
    }
}
```



CnC – Another Example: Face Recognition - Parallel

- Applying a set of filters on an blocks of an image
  - Each step detects nose, eyes, … and removes blocks from the next step

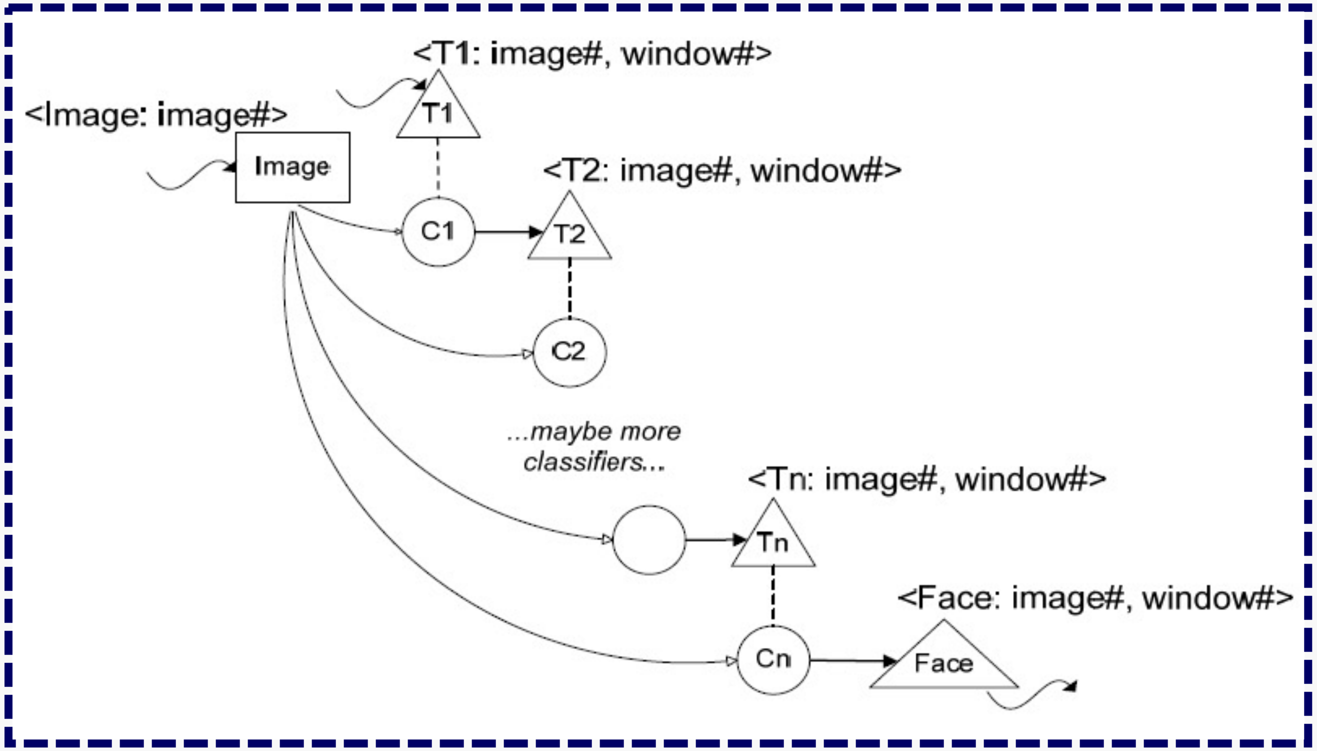


Image from CnC Tutorial

CnC - Summary

- Expressing program in terms of higher level and domain specific operators and data types
- Single Assignment rule (analogous to streaming)
- Based on data and control dependences
- Race free
- Platform independent

MapReduce

- Used by Google for production software
- Used on 1000s processors machines
- Automatic parallelization and distribution
- Fault-tolerance
- It is a library built in C++

Adapted From: http://labs.google.com/papers/mapreduce.html

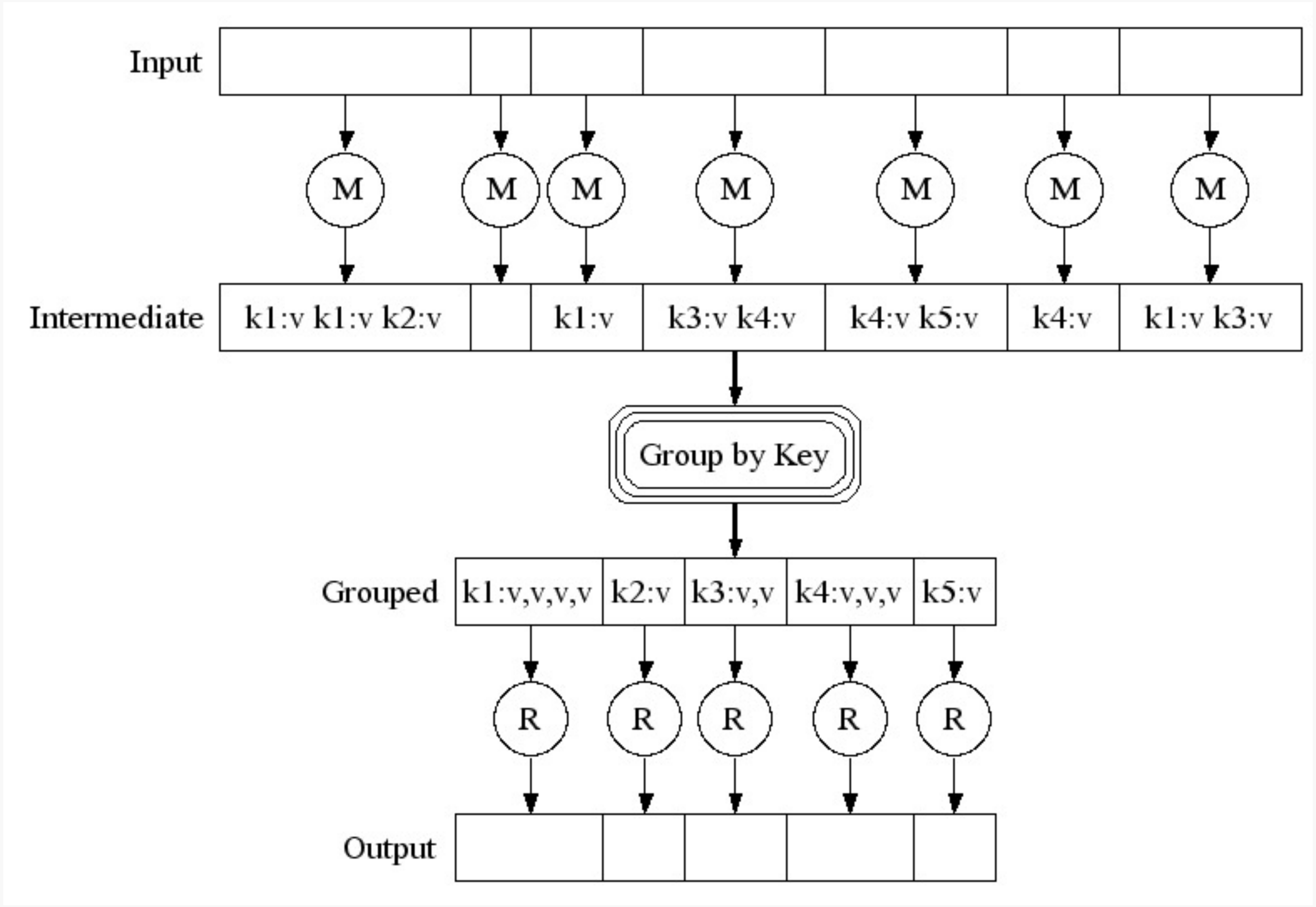
MapReduce Model

- Input & Output are sets of key/value pairs
- Programmer specifies two functions:
  - `map(in_key, in_value) -> list(out_key, intermediate_value)`
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - `reduce(out_key, list(intermediate_value)) -> list(out_value)`
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)

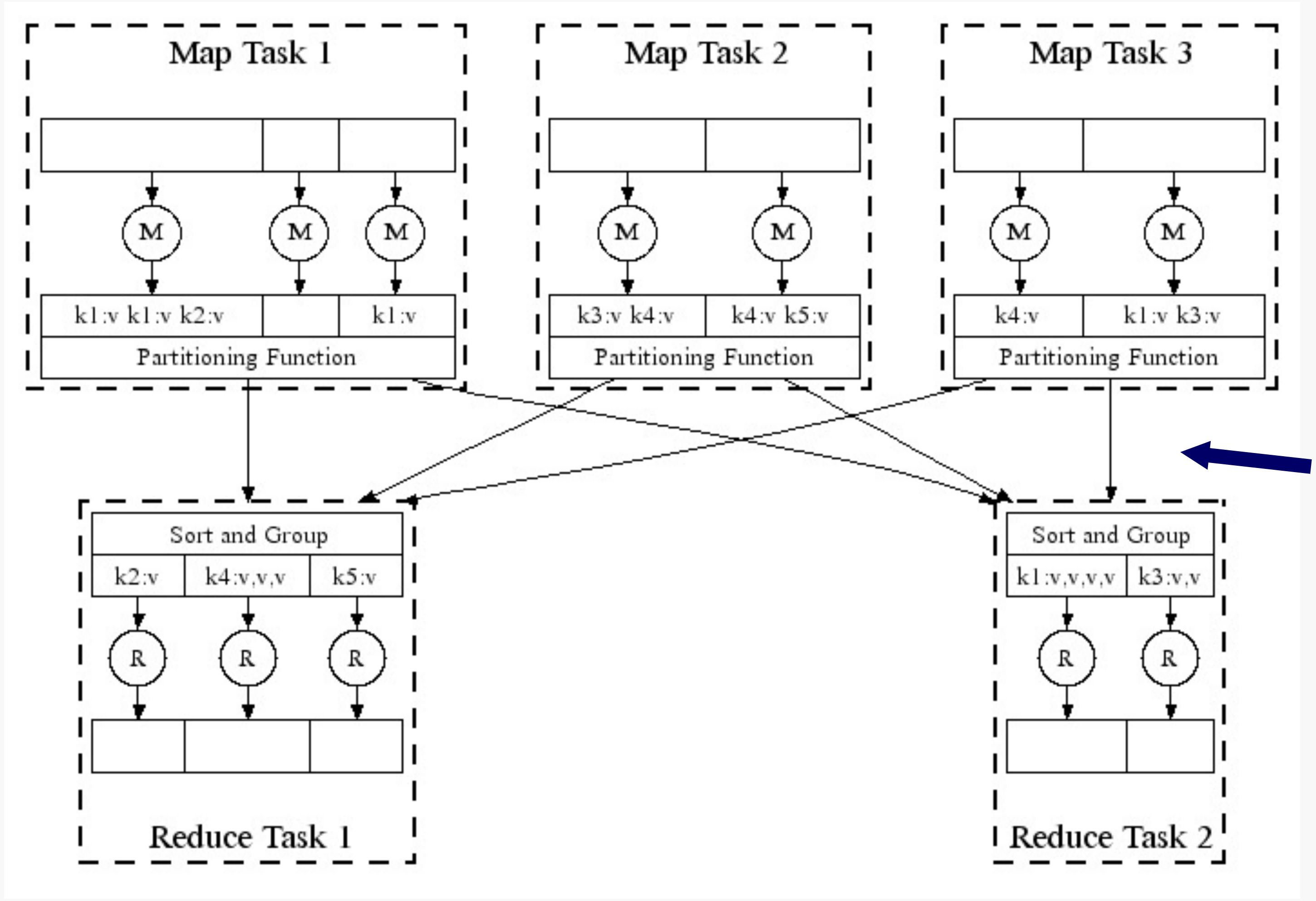
Example: Word Count

```
map(String input_key, String input_value):
   // input_key: document name
   // input_value: document contents
   for each word w in input_value:
       EmitIntermediate(w, "1");

reduce(String output_key, Iterator
   intermediate_values):
   // output_key: a word
   // output_values: a list of counts
   int result = 0; for each v in
   intermediate_values: result += ParseInt(v);
   Emit(AsString(result));
```

Sequential Execution Model



Parallel Execution Model



Shuffling & Sorting

Parallel Execution Model

- Fine granularity tasks: many more map tasks than machines
- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing
- Often use 200,000 map/5000 reduce tasks w/ 2000 machines

Performance

- Typical cluster:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - distributed file system manages data (GFS)
  - Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Performance: Locality

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect: Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

Performance: Replication

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

Performance

- Sorting guarantees within each reduce partition
- Compression of intermediate data
- Combiner: useful for saving network bandwidth

Fault Tolerance

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Master failure not handled yet
- Robust: lost 1600 of 1800 machines once, but finished fine

Productivity

- User specifies only two functions
- May be complex to specify a general algorithm
- Highly productive for specific kind of problems

References

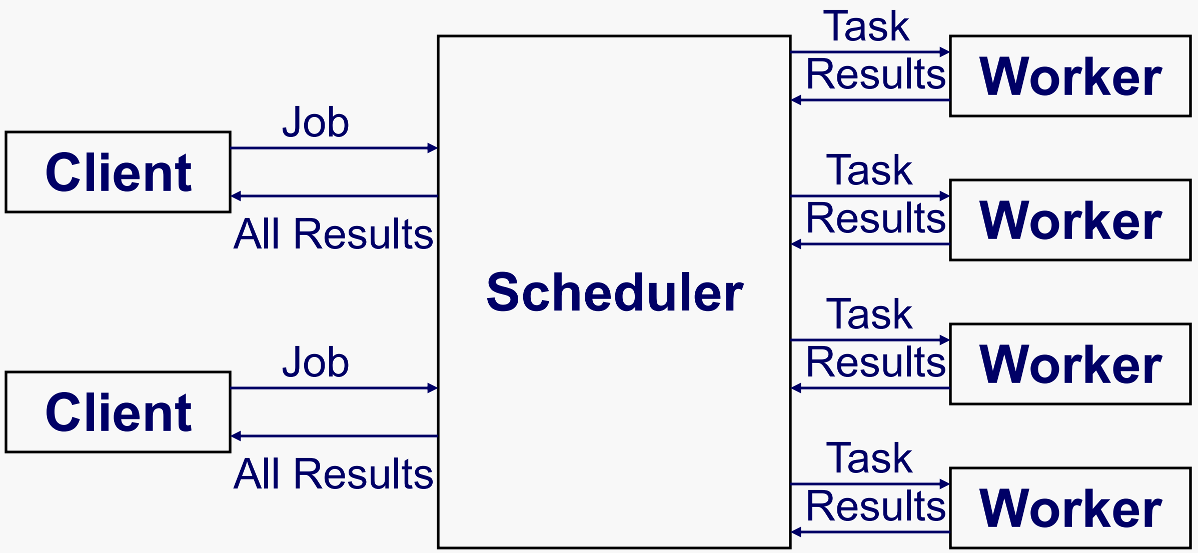
- Linda
  - http://www.lindaspaces.com/about/index.html
  - http://www.almaden.ibm.com/cs/TSpaces/
  - http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/
- MapReduce
  - http://labs.google.com/papers/mapreduce.html
  - http://www.cs.virginia.edu/~pact2006/program/mapreduce-pact06-keynote.pdf
- CnC
  - http://software.intel.com/en-us/sites/whatif/runtime_api/index.html

Table of Contents

MATLAB DCE

- Executing independent jobs in a cluster environment
- A job is a set of tasks
- A task specifies input data and operations to be performed
- A scheduler takes a job and executes its tasks

Execution Model



Job Creation and Execution

Parasol

- Create a Scheduler: `sched = findResource('scheduler', 'type', 'local')`
- Create a Job: `j = createJob(sched);`
- Create Tasks
  - `createTask(j, @sum, 1, {[1 1]});`
  - `createTask(j, @sum, 1, {[2 2]});`
- Submit job: `submit(j);`
- Get results
  - `waitForState(j);`
  - `results = getAllOutputArguments(j)`

    ```
    results =
    [2]
    [4]
    ```
- Destroy job: `destroy(j);`

Number of output arguments

Portability

- Different ways to pass data to workers
  - Passing paths for data and functions when using a shared file system
  - Compressing and passing data and functions to workers initializing an environment at worker place
- The first way is less portable even though more efficient

Productivity

- MATLAB DCE is a queuing system
- Schedule independent jobs
- It may be difficult to code an arbitrary parallel algorithm
- Good for speeding up huge computation with very high level independent tasks

References

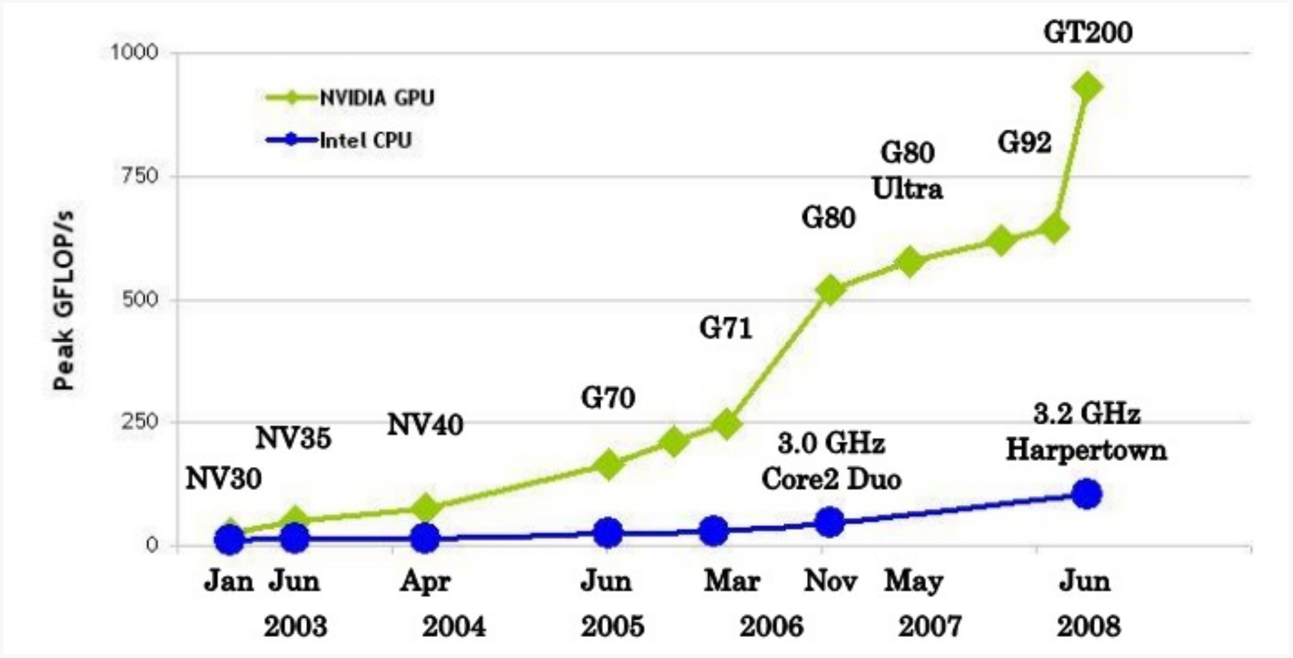
- Linda
  - http://www.lindaspaces.com/about/index.html
  - http://www.almaden.ibm.com/cs/TSpaces/
  - http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/
- MapReduce
  - http://labs.google.com/papers/mapreduce.html
  - http://www.cs.virginia.edu/~pact2006/program/mapreduce-pact06-keynote.pdf
- MATLAB DCE
  - http://www.mathworks.com/products/distriben/
  - http://www.mathworks.com/products/distribtb/

Streaming and GPUs

- Most of the streaming languages focus on GPUs
- GPUs – Graphics Processing Unit
  - Suitable for intensive data-parallel apps
  - Cheap < 1000$ for TeraFLOPS computation power
  - Previously used only for projecting images on 2D planes
  - Now used for General Purposes → GPGPUs

But why GPUs?

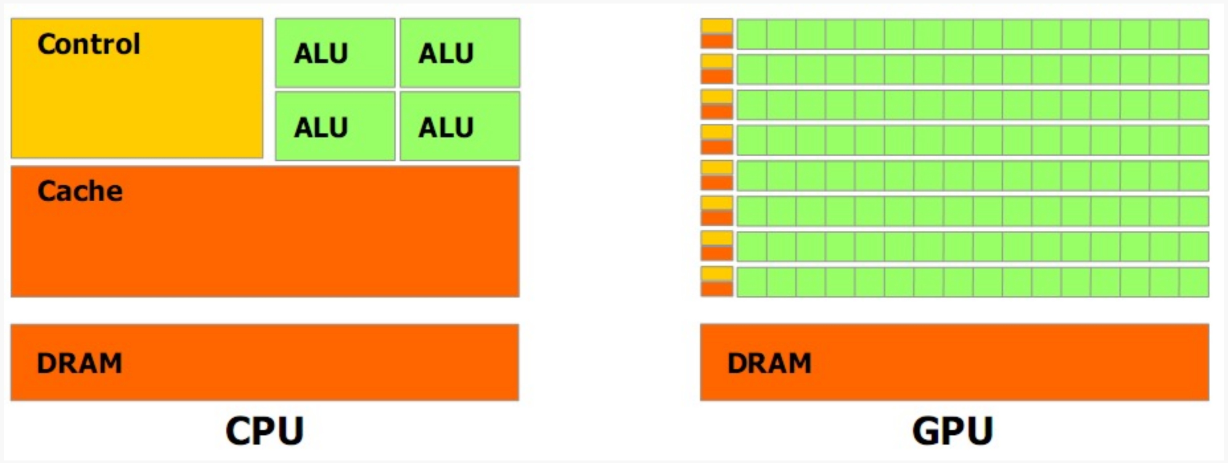
- In 2010 – NVIDIA Tesla S1070 delivers up to
  - 4.14 TFLOPS Single precision
  - 408 GB/sec memory bandwidth



Graph from - CUDA Programming Guide 3.0

CPU vs. GPU

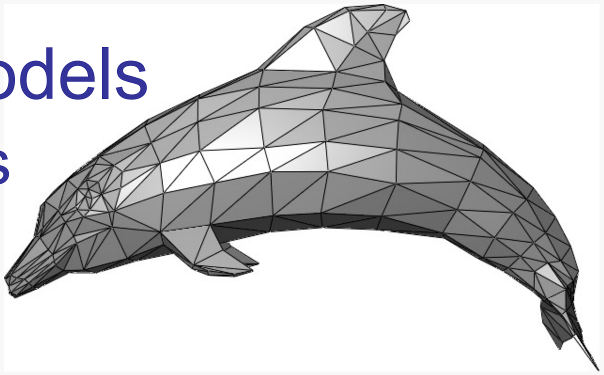
- **CPU**
  - Does more than computing
    - Branch prediction
    - Caching
    - Address translation (TLB, protection)
    - …
- **GPU**
  - compute-intensive
  - Focuses on
    - data-parallel computations
    - With high **computation/memory access** ratio
  - highly parallel computation
  - more transistors for computation



Picture from - CUDA Programming Guide 3.0

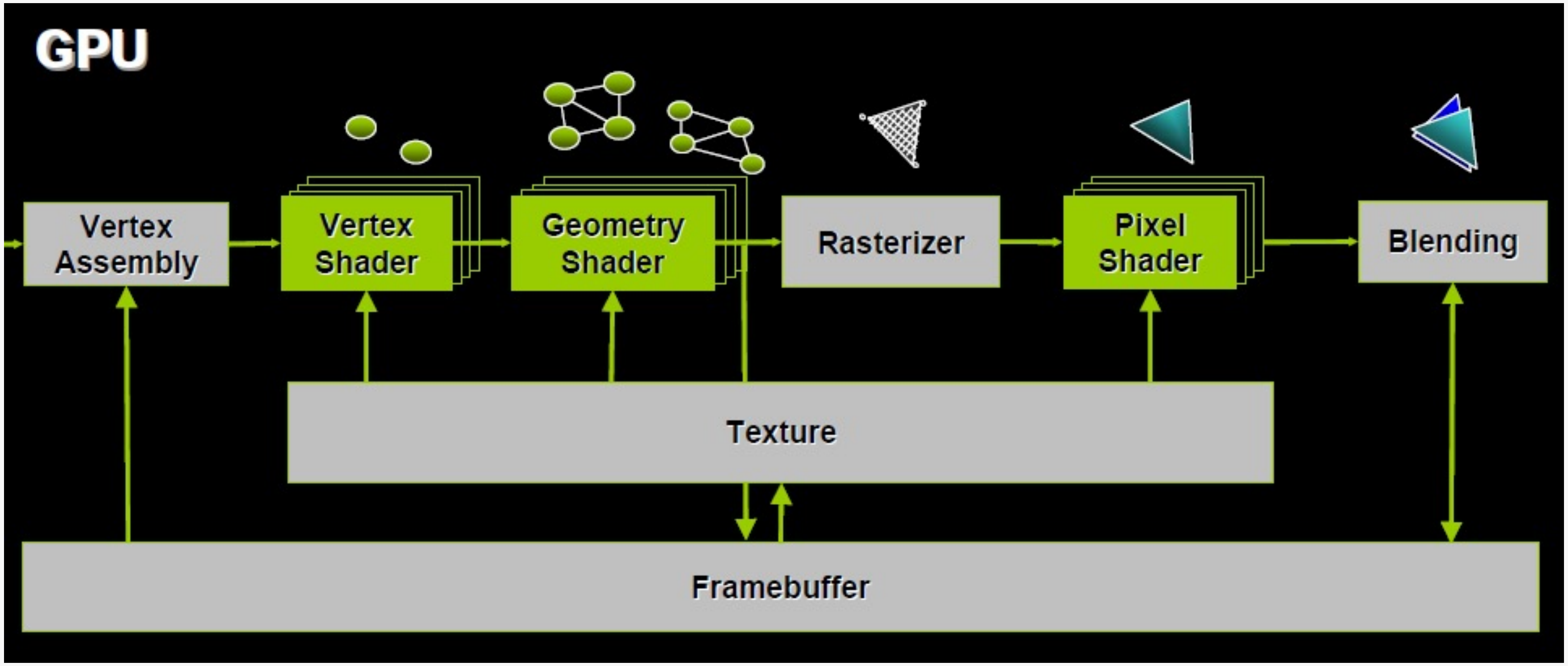
GPU Architecture

- Based on the concept of 3D models
  - 3D models are made of triangles
  - Vertices and legs



- GPUs has 3 steps in a pipeline to convert 3D to 2D
  - **Vertex Shaders** - projecting vertices to pixels
  - **Geometry Shaders** - generating legs for triangles and producing *pixels* for the legs (Rasterizing)
  - **Fragment (Pixel) Shaders** - determining colors of each pixel

GPU Architecture



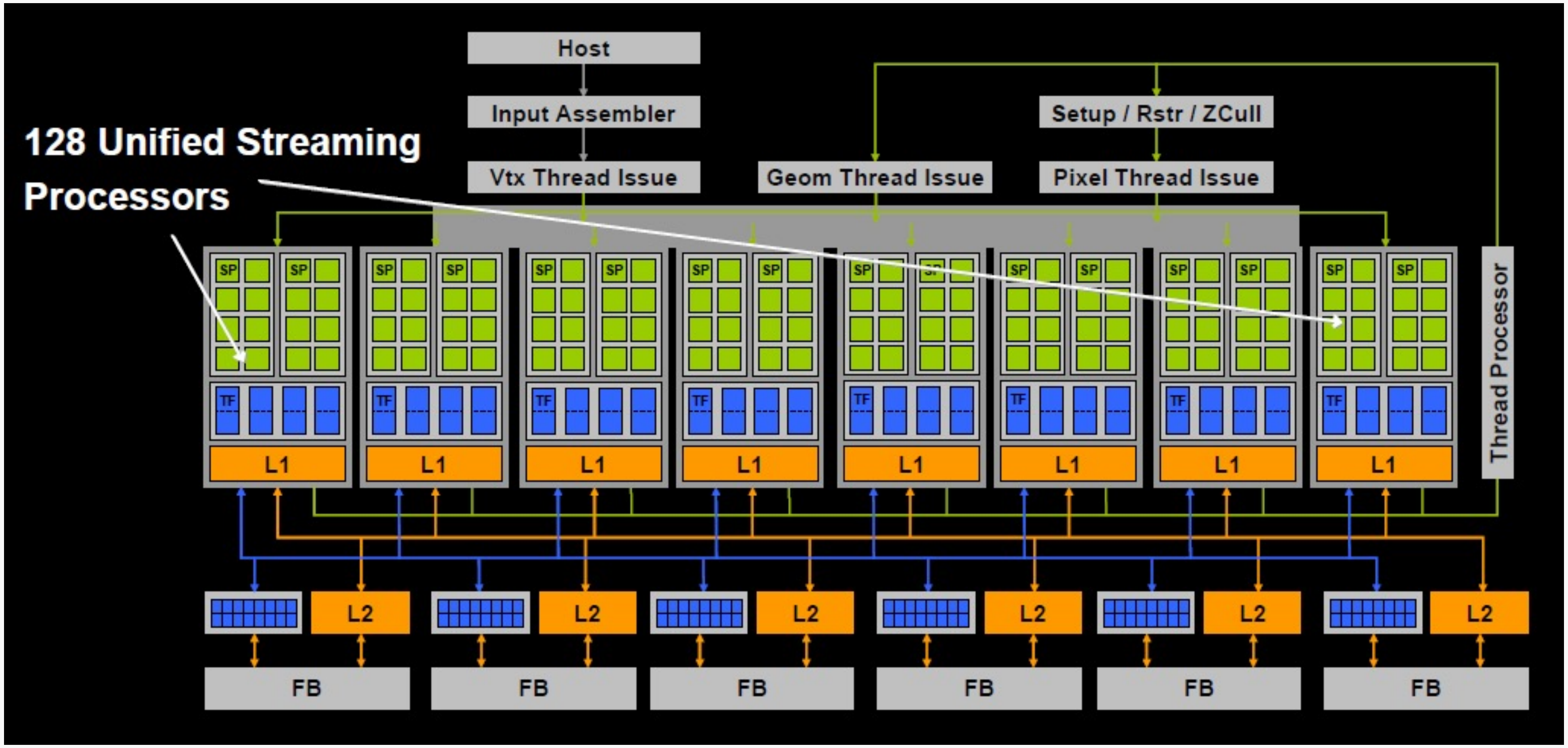
Picture from - An Introduction to Modern GPU Architecture by Ashu Rege

Modern GPU Architecture

- A farm of unified processors (shaders)
- Replacing the pipeline architecture
- Each shader
  - Receives input
  - Performs an action (making pixels, rasterizing, ...)
  - Produces output

Modern GPU Architecture

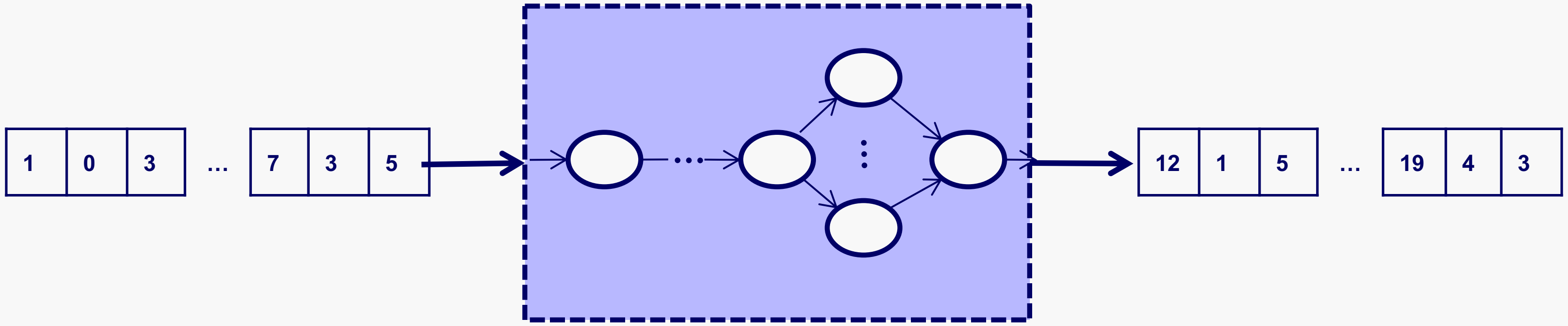
- GeForce 8800 GTX (G80)



Picture from - An Introduction to Modern GPU Architecture by Ashu Rege

Streaming - Definition

- What is streaming?
  - Continuous input data
  - Processed by a set of *filters* (also called *kernels*, or *streams*)
  - Continuous output data



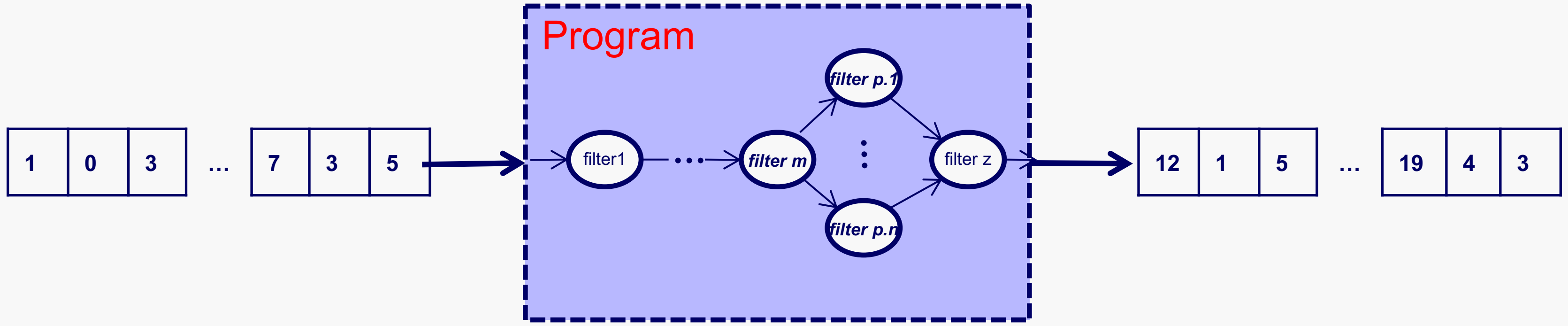
Streaming: Application Domain

- Audio & video streaming
- Digital signal processing
- Networking and encryption, etc.

- Examples:
  - Consumer electronics (Audio, Video, Bluetooth, etc.)
  - Embedded systems (Encryption and networking)
  - Videocards (Video)

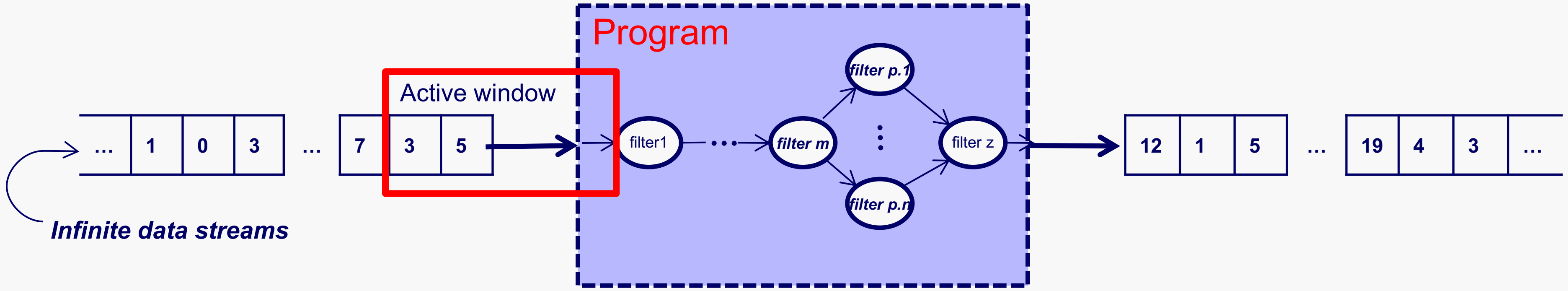
Streaming: A Closer Look

- Program representation: data-flow graph
  - Collection of filters
  - Connected to each other using streaming *channels*
  - Called ***filter graph*** or ***stream graph***



Streaming Characteristics

- Data-intensive
- Continuous (infinite) data inputs – ***data streams***
- Typically
  - Data with the same type
  - With low re-usability
- Consists of ***active windows*** – covering current data and the processed **data**



Streaming Characteristics

- Parallelism
  - Data parallelism
  - Task parallelism
  - Pipeline parallelism
- Communication
  - Producer-consumer locality
  - Predictable memory access pattern
  - No read-write hazards
- A lot like vector machines

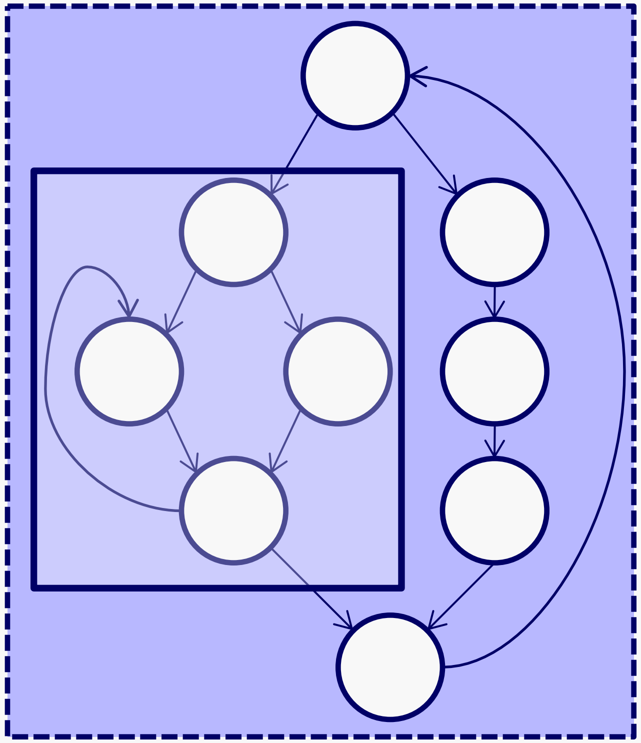
Streaming Based Languages

- **Lucid** (1974)
  - The first data-flow and streaming language
- **StreamIt** from MIT
- **BrookGPU** for GPUs from Stanford

- And many other languages and standards: StreamC, OpenCL, FastFlow, etc.

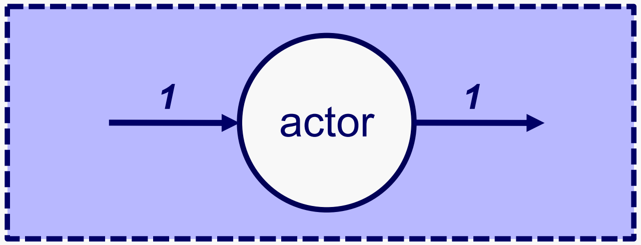
StreamIt

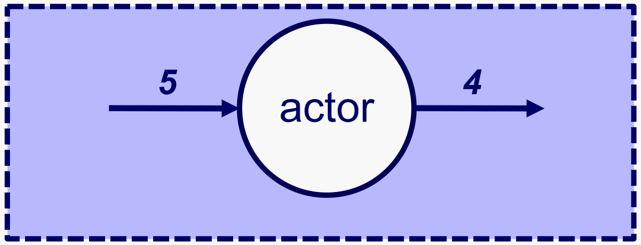
- Is a **programming language** and a **compilation infrastructure** for streaming
- Addresses both **performance improvement** and **ease of programmability**
- Properties
  - Composable filters (white box)
  - Teleport messaging for events – later in the slides
  - The ability to operate directly on compressed data streams
  - Based on Cyclo-static Dataflows (next slide)

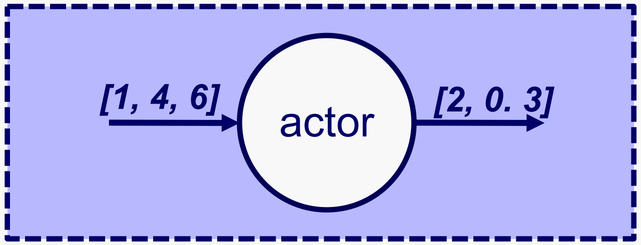


Cyclo-Static Dataflow

- **Basic dataflow**
  - both consumption and production rate are always 1
- **Synchronous dataflow**
  - consumption and production rate are known
- **Cyclo-static dataflow**
  - consumption and production rate change cyclically on known numbers
- All of the above allow static scheduling of a dataflow graph

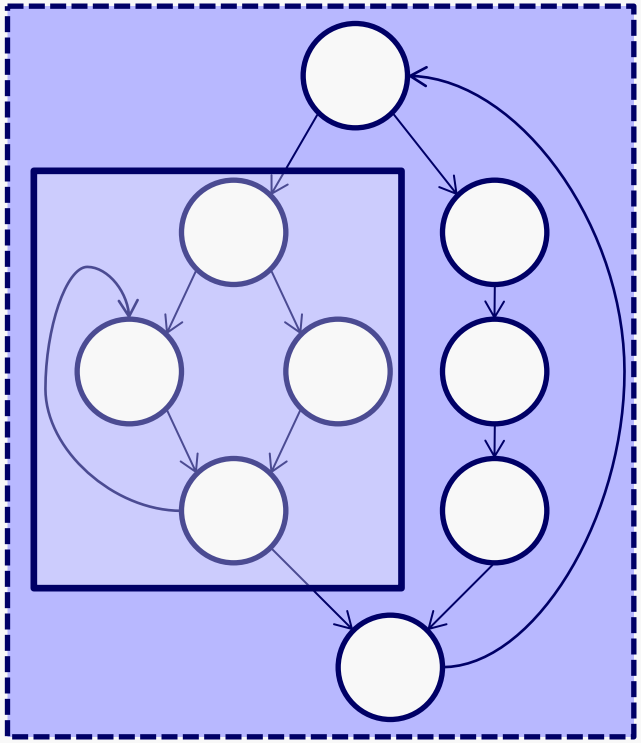






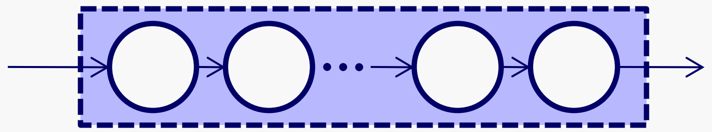
More about StreamIt

- High-level architecture independent language
- Backends for uni/multi processors
- Static scheduling (CSDF)
- Based on structured graphs
- Providing different levels of parallelism
  - **Data** – duplicating coarse grained
    data parallel units (fusing DOALL steps)
  - **Task** – split-join
  - **Pipeline** – using software pipelining

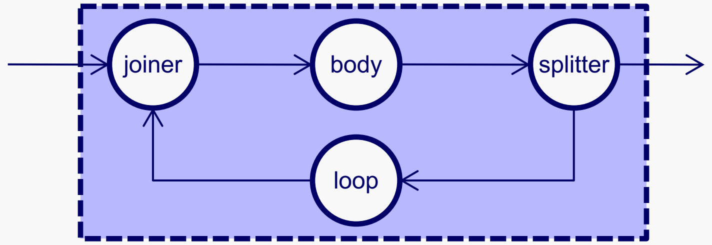


Programming in StreamIt

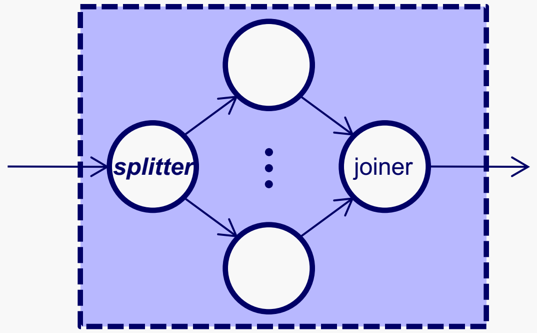
- Top-down approach
- Decomposing the application to autonomous unit of computation – *filters*
- Filters use FIFO communication channel
- Uses three constructs to connect filter:



pipeline



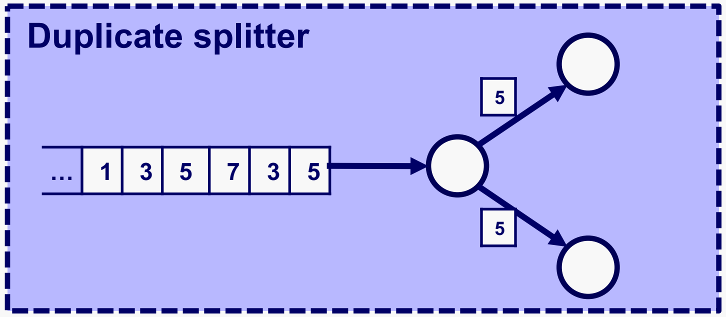
feedback-loop

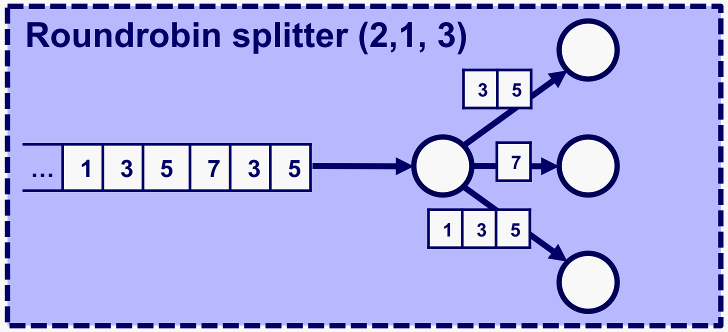


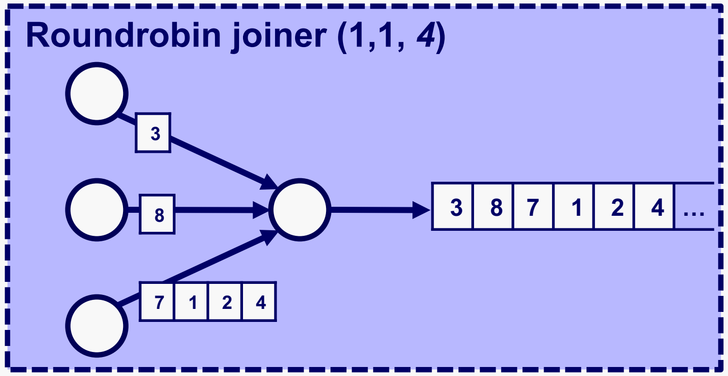
split-join

StreamIt Splitters and Joiners

- Splitters
  - Duplicate – copies each input item to each output channel
  - Roundrobin
    - Annotated with weights (w_1, w_2, …, w_n)
    - n = number of output channels
    - Puts the first w_1 elements on the first channel, w_2 on the second, ..., and w_n on the last channel
- Joiners
  - Only roundrobin joiner
    - Annotated with weights (w_1, w_2, …, w_n)
    - n = number of input channels
    - Receives the first w_1 elements from the first channel, w_2 from the second one, ..., and w_n from the last channel







Teleport Messaging

- The idea
  - Sending messages outside of the normal stream of data
- Example – radio application
  - Request for change listening frequency (infrequent event)
  - Detected by filter late in the application
  - Require a change by a filter early in the application
  - **Solution** – teleport messaging
  - Affects data computation
  - Does not violates static scheduling

Some StreamIt Characteristics

- Execution Model
  - Explicitly parallel
  - Dynamic rates – only as [min..max] ranges (still statically schedulable)
  - Single-input stream and single-output stream filters
  - Teleport messaging – out-of-band control messages
  - Peeking – reading data without popping it from the stream
  - Nested parallelism support – nested filters
  - Filters have private and independent address space

StreamIt

- Productivity
  - A new language – based on streaming semantics
  - Compiler produces streaming graphs showing the application
  - High level of abstraction – using streaming semantics
  - User level tunability
    - Compiler can produce C or C++ codes from the program
    - User can modify those files before final compilation
  - Composability supported by nesting filters
- Performance
  - Load balancing
    - Compiler finds the best combination of data, task, pipeline parallelism
    - Using fission and fusion
  - Using teleport messaging (asynchronous control messages) instead of feedback loop
    - 49% improvement for a software radio benchmark

StreamIt: Portability

- Compiler can compile to C or C++ sources
  - Can be compiled on different platforms along with StreamIt runtime library
- No reliance on system feature – StreamIt runtime handles everything
- Targeting parallelism on multicores

Hello World Example

- Printing numbers

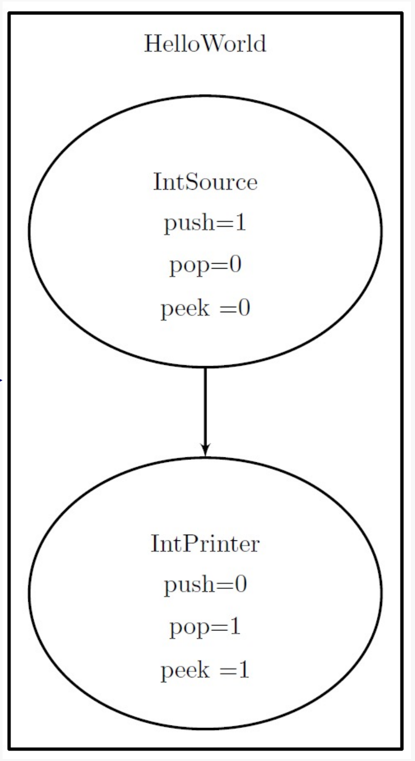
- Components:
  - **IntSource** creates elements
  - Printer prints the elements
  - **HelloWorld** connects these two filters using a pipeline

```
void->int filter IntSource {
    int x;
    init {
        x = 0;
    }
    push(x++);
}

int->void filter IntPrinter {
        println(pop());
}

void->void pipeline HelloWorld {
    add IntSource();
    add IntPrinter();
}
```

Visual Representation

- StreamIt compiler produces several graphs for an application
- One is the application graph

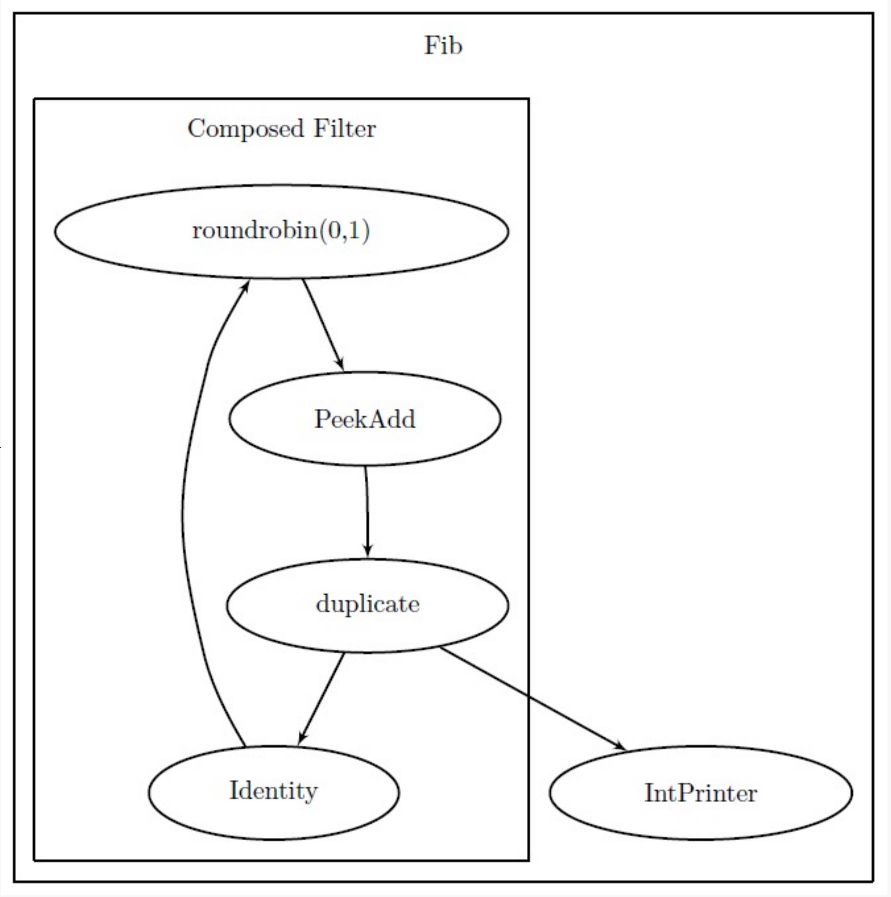
```
void->int filter IntSource {
   int x;
   init {
      x = 0;
   }
   push(x++);
}

int->void filter IntPrinter {
      println(pop());
}

void->void pipeline
HelloWorld {
   add IntSource();
   add IntPrinter();
}
```



A More Concrete Example: Fibonacci Code

- Note that roundrobin(0,1) means that the actor will only use the value from identity (the arrow coming to the loop is not shown)

```
void->void pipeline Fib {
   add feedbackloop {
     join roundrobin(0, 1);
     body PeekAdd();
     loop Identity<int>();
     split duplicate;
     enqueue 0; //entering the first two
     enqueue 1; //numbers in the queue
   };
   add IntPrinter();
}

int->int filter PeekAdd {
  push(pop() + peek(0)) ;
}

int->void filter IntPrinter {
  println(pop());
}
```



StreamIt - Summary

- Based on stream graph (filter dependence graph)
- No stream data type – data type should match channel type
- Partial reconfiguration – letting the filters replace themselves by an updated version
- Expressive & productive for streaming apps.
- Performance is good
- Needs and has good compiler

BrookGPU – Streaming Language

- Designed for streaming processors such as Merrimac (Streaming Processor from Stanford) and GPUs

- Similar in terminology to other streaming languages
  - Applies ***kernels*** on ***data streams***

- Designed based on four goals
  - Portability
  - Performance
  - Data parallelism
  - Computational intensity

BrookGPU - Model

- Execution Model
  - Can work on multiple graph
  - Explicitly parallel
  - Kernels accept
    - Multi-dimensional streams
    - And multiple inputs and outputs
  - **No** nested call – flat dataflow
  - **No** recursion – as to allow inter-procedural analysis and inlining
  - **Restricted** pointer operation - allowing only pass by reference to functions
  - **No** memory allocation in the kernels

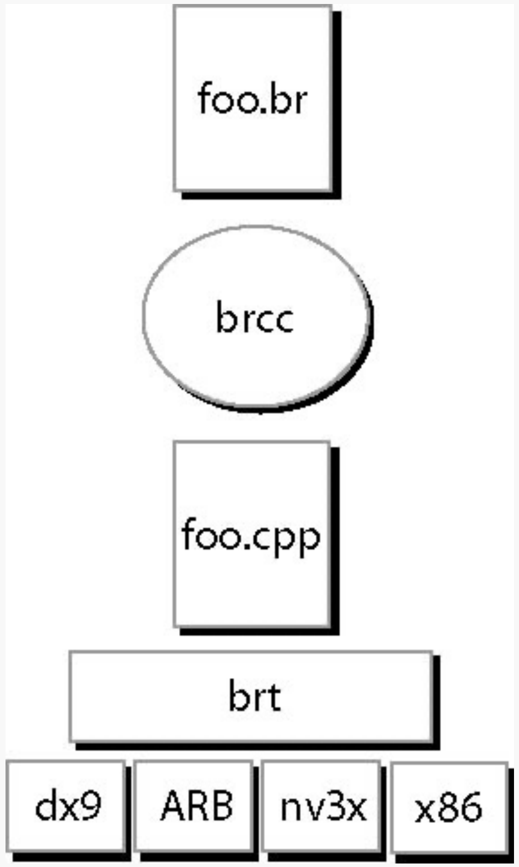
BrookGPU Model

- Productivity
  - Extension of standard ANSI C with streaming constructs
  - High level of abstraction – suitable for streaming application
  - Composability and nested calls are not supported
  - Multiple/multidimensional streams
- Performance
  - Load balancing – Brook compiler does nothing for load balancing
  - Scalable parallelism

BrookGPU Model

- Portability
  - Maps to various streaming architectures – Merrimac (Stanford streaming supercomputer), GPUs, etc.
  - Free of any explicit graphics construct
    - Works for NVIDIA and ATI hardware, and CPU

Brook Outline



.br

Brook source files

brcc

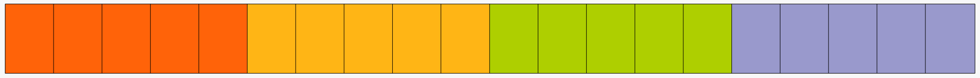
source to source compiler

brt

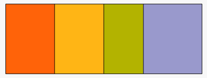
Brook run-time library

BrookGPU - Syntax

- Stream declarations
  - **float** <> a – a stream of floats
  - **float** <> b [2][3] – a stream of [2, 3] float arrays
  - **float** <10> c – a stream of size 10 (size in <> used for aligning purposes)
  - **float** <100, 200, 300> d - a stream of size 100x200x300
- Matching stream sizes before function calls
  - Example: sum of each 5 elements in the result
  - **float** <20> a



  - **float** <5> result



  - sum (a, r) – function is shown in the next slide

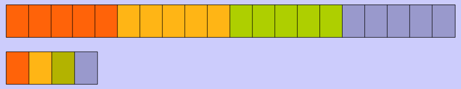
Reduction in BrookGPU

- Two mechanisms
  - **Reduction operators** – providing mechanism for performing simple reductions on **native types** with **predefined operators** (+, *, &, |)
  - **Reduction functions** – allowing reductions on **complex data types** and with **user-defined operators**

BrookGPU – Reduction Variable

- Applying reduction using reduction variables
  - Perceived as a sequential operation
  - Performed in any possible order
- Example showing both aligning and reduction variables

```
void kernel sum (float a<>, reduce float
    result) {
      result = result + a;
}
float <20> a;
float <5> result;
reduce_sum(a, r);
```



```
for (int i = 0; i < 20; i++) {
        result [i / 5] += a[i];
}
```

BrookGPU – Reduction Function

- Applying reduction using both *reduction functions* and *reduction variables*
  - This example shows it for native types and basic operators
  - This reduction can be extended to complex data types and operators

```
void reduce sum (complex a<>, reduce
     complex result) {
       result = result + a;
     }

complex <100> a;
reduce_sum(a);
```

```
for (int i = 0; i < 100; i++) {
         result += a[i];
}
```

BrookGPU - Summary

- Flat stream graph
- Stateless kernels
- Task parallelism by using kernels
- Data parallelism by using streams
- Static/Dynamic rate of consumption and production determined by compiler/at runtime
- Controls are non-kernel part of the code in C
  - Main function
  - Dependency between kernels

Parallel Skeletons

- Skeletons term - by Murray Cole for the first time
  - Many parallel apps have common interaction patterns (pipeline, farm, …)
  - Skeletons capture those common algorithmic forms
  - Skeletons - building blocks for creating applications
- Benefits
  - Higher level programming interface
  - Allowing formal analysis
  - Easier transformation of a program
  - Allowing portable and efficient generic implementations

Parallel Skeletons - Origin

- Inspired by functional programming paradigms
- Programs are based on two type of skeletons
  - **Algorithmic skeletons** – a higher-order function
  - **Architectural skeletons** – implementing the paradigm on a target
- Benefits
  - Hiding parallelism and communication details in architectural skeletons
  - Letting a programmer to focus on the algorithmic part

Homomorphic Skeletons

- Also called **catamorphic skeletons**
- Providing a more formal framework for
  - Program **construction**
  - Program **transformation**

- Defined for lists, arrays, trees and graphs
- Applicable to other datatypes only by defining new operators

- Act similar to abstract data type
  - Defining a set of parallel operators
  - Hiding implementation details

Skeletons – Formal Definition

- Notations
  - f a = f(a)
  - f ∘ g a = f (g a)
  - **id** ∘ f = f ∘ **id** = f
  - [a1, ..., am] **++** [b1, ..., bn] = [a1, ..., am, b1, ..., bn]
    - concatenation

  - **map** f [a1, ..., am] = [f a1, ..., f am]
  - **red** (⊛) [a1, ..., am] = a1 ⊛ a2 ⊛ … ⊛ am
  - **scan** (⊛) [a1, ..., am] = [a1, a1 ⊛ a2, ..., a1⊛ … ⊛ am]
  - **suf** (⊛) [a1, ..., am] = [a1 ⊛ … ⊛ am, ..., a1 ⊛ a2, a1]

Homomorphic Skeletons - Definition

- If a function is homomorphism on a list
  - It is convertible to map and reduce
  - Means - O(log n) if tree-based reduction is used

- A function $h \in [A] \to X$ is a **list** homomorphism w.r.t. binary operator iff for any list l1 and l2:

$$h\ (l1 +\!+\ l2) = (h\ l1) \circledast (h\ l2)$$

- A function h on lists is a homomorphism w.r.t. an associative operator $\circledast$ iff

$$h = red(\circledast) \circ map\ \varphi$$

Where $\varphi\ a = h\ [a]$

Homomorphic Representation

- **Useful**
  - Converting the program to a *map* and a *reduce phase*
  - Efficient implementation for map and reduce in many parallel environments
- **Problem**
  - Is useful *only* if the application is convertible to homomorphic skeletons
  - Even if it is convertible, conversion is hard, and consists of many steps (example in the next slide)
- **Solution**
  - Automatic extraction of homomorphic functions
  - One possible solution – **cons-snoc Method**

Homomorphism Example - Maximum Segment Sum (MSS)

- Algorithm – finding contiguous list segment with largest sum and returning the sum
- Intuitive formal definition of *mss* ($\uparrow$ is operator 'max')

$$mss = red(\uparrow) \circ map\ (red\ (+)) \circ segs$$

- Converted to homomorphic representation

$$mss = (fst \circ fst) \circ red\ (\Uparrow \curlyvee (\uparrow \curlywedge +)) \circ map(\Delta \circ \Delta)$$

- In which
  - $\Delta\ a = (a, a)$
  - $fst\ (a, b) = a$
  - $(a, b) \Uparrow (c, d) = (a \uparrow c, b \uparrow d)$
  - $((r1, s1), (t1, u1)) \Uparrow \curlyvee (\uparrow \curlywedge +)\ ((r2, s2), (t2, u2)) =$

    $= ((r1 \uparrow r2 \uparrow (t1 + s2), s1 \uparrow (u1 + s2)), (t2 \uparrow (t1 + u2), u1 + u2))$
- Problem – it is both hard to write and hard to understand

Automatic Extraction of Homomorphism for Lists

- A function $h \in [A] \rightarrow X$ is a homomorphism *iff* it is both **conslist** and **snoclist** homomorphism

- Notations
  - $\prec$ (cons) – attaches an element to the *front* of a list

    $$a \prec l = [a] ++ l$$
  - $\succ$ (snoc) – attaches an element to the *end* of a list

    $$l \succ a = l ++ [a]$$

- Function $h \in [A] \rightarrow X$ is a **conslist homomorphism** w.r.t. $\Phi \in [A] \rightarrow X$ and $\odot \in A \times X \rightarrow X$ if:

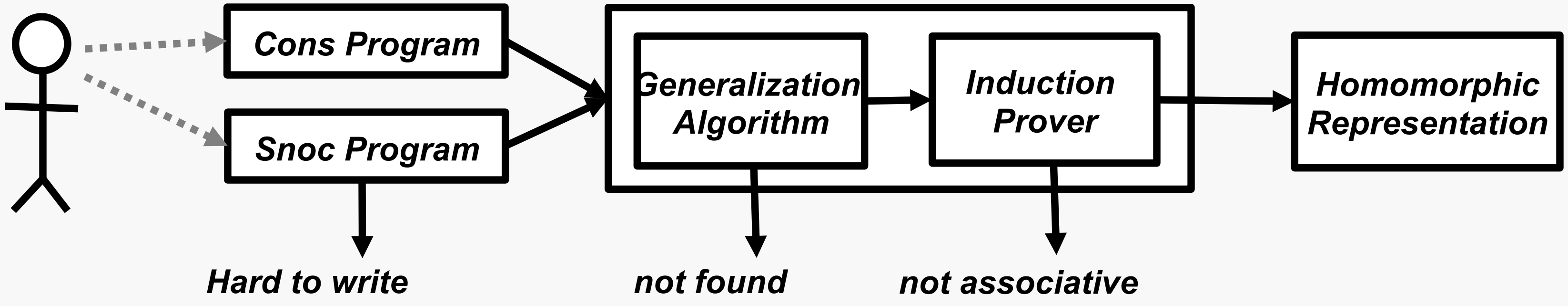
  $h\ [a] = \Phi\ a$

  $h\ (a \prec l) = a \odot (h\ l)$

Automatic Extraction of Homomorphism for Lists

- User should provide
  - Cons representation of the program
  - Snoc representation of the program

- CS (cons and snoc) method could provide **homomorphic representation**



Parallel Skeletons - Summary

- **Useful**
  - For small algorithms and small applications
  - Can be supported by formal proofs
  - For MPI implementers to find better implementations

- Not practical for real-life parallel programs
  - Gets complicated too soon

- Conclusion
  - higher level of application abstraction is needed
  - Like what happened to sequential programs – UML, LOP, AOP, etc.

Conclusions

- High level PPM – high productivity
- Low level PPM – high performance ?
- Safety in higher abstraction
- Needed: Parallel RTS, Debuggers
- Desperately Needed: Compilers
- Domain is in flux .... We shall see