DF21500 Multicore Computing

# Concepts of Parallel Programming Languages for Multicore Computing

**Christoph Kessler, IDA**



## The Multicore Challenge

- Technical + physical causes enforce that we will use parallel systems with more and more CPUs (but no higher clock frequency) in the foreseeable future
  - Number of cores expected to double every ~ 2 years
    - In 2007 8..32 hardware threads per chip
    - By 2011 ca. 100 hardware threads per chip
    - By 2017 ca. 1000 hardware threads per chip
    - ...



## The Multicore Challenge

- **Need to expose massive explicit parallelism in programs**
  - all application areas, not only traditional HPC
    - desktop applications, graphics, games, embedded, DSP
  - Automatic parallelization?
    - at compile time: Often not feasible, needs rewriting ... (ok for SIMDization, ILP extraction, kernel recognition)
    - at run time (e.g. speculative multithreading): not scalable
- **Bad news 1:**
  Many programmers (also less skilled ones) need to use parallel programming languages in the future



## The Multicore Challenge (cont.)

- Different kinds of multicore based parallel systems:
  - "Simplest" (?): symmetric multiprocessor (SMP)
    - current homogeneous multicores: (quadcore Xeon etc.) shared memory, tightly coupled ((partly) common L2 cache)
    - CC-NUMA systems built from these
    - usually cache-based – data locality matters (exception: hardware multithreading within a core)
  - The future (?!): asymmetric (heterogeneous) chip multiprocessor (CMP)
    - few "fat cores" for sequential parts, plus accelerators, arrays of simple processors, GPU-on-chip, … for parallel / accelerable program parts
  - For HPC: multicomputer / cluster / distributed memory system
- **Bad news 2:**
  There will be no single uniform parallel programming model as we were used to in the old sequential times
  → Several competing general-purpose languages will co-exist, and DSLs (domain-specific languages)



## Parallel Language Concepts

- **Parallelism / par. control flow**
  - Processes, threads, tasks
  - Fork-join style parallelism, SPMD style parallelism
  - Nested parallelism
  - Parallel loops, Sections
  - Parallel loop scheduling
  - Implicit parallelism
- **Synchronization & Consistency**
  - Futures
  - Supersteps and Barriers
  - Array assignments
  - Fence / Flush
  - Semaphores & Monitors
  - Atomic operations
  - Transactions
- **Address space**
  - (Partitioned) Global Address Space
  - Sharing
  - Pointer models
  - Tuple space
- **Data locality & mapping control**
  - Co-Arrays
  - Virtual topologies
  - Alignment, distribution, mapping
  - Data distributions
  - Data redistribution
- **Communication**
  - Collective communication
  - One-sided communication
    (see earlier lecture on MPI)



## Some parallel programming languages

**(partly) covered here:**

- Fork (see FDA125)
- Cilk (see later lecture)
- MPI (see earlier course)
- OpenMP
- HPF
- UPC / Titanium
- NestStep
- ZPL
- X10, Chapel
- Java (5+) Concurrency Package
- Cell SDK (see earlier lecture)
- CUDA / OpenCL (see later lecture)

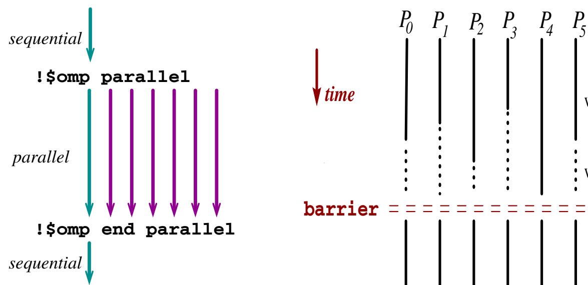
## Relationship between parallel and sequential programming languages

- Big issue: Legacy code in Fortran, C, (C++)
  - Practically successful parallel languages must be interoperable with, and, even better, syntactically similar to one of these
- Compliance with sequential version is useful
  - e.g. C elision of a Cilk program is a valid C program doing the same computation
  - OpenMP
- Incremental parallelization supported by directive-based languages
  - e.g. OpenMP, HPF

# Parallel Control Flow

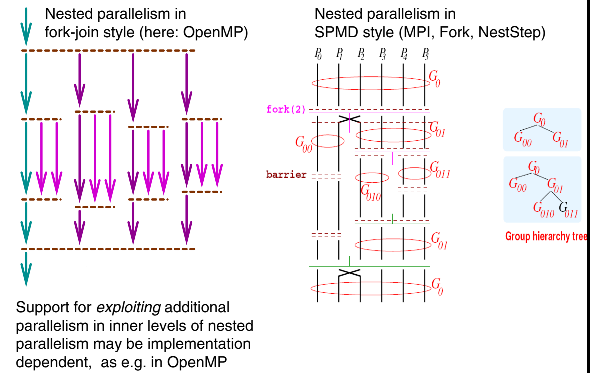
Christoph Kessler, IDA,
Linköpings universitet, 2009.

## Fork-Join-Style Parallelism vs. SPMD-Style Parallelism



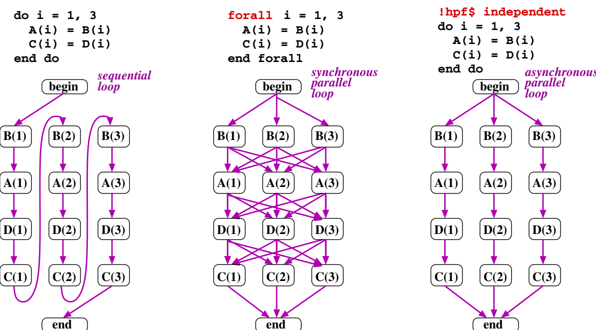



## Nested Parallelism



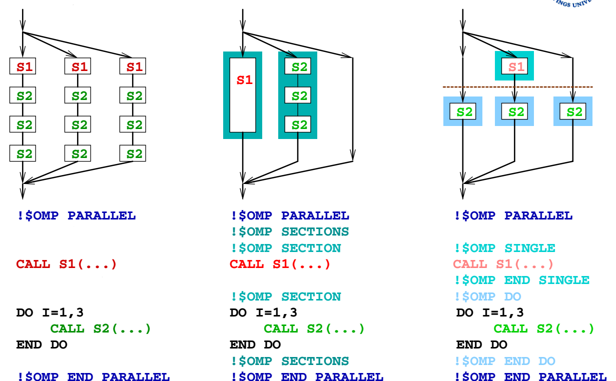



## Parallel Loop Constructs






## Parallel Sections



## Parallel Loop Scheduling (1)

- Static scheduling
  - Chunk scheduling



```
!$omp do schedule ( STATIC, 2 )
      do i = 1, ..., 11
         ....
      end do
```



## Parallel Loop Scheduling (2)

- Dynamic Loop Scheduling
  - Chunk Scheduling



```
!$omp do schedule ( DYNAMIC, 1 )
      do i = 1, ..., 8
         ....
      end do
```



## Parallel Loop Scheduling (3)

- Guided Self-Scheduling
  - Chunk scheduling



```
!$omp do schedule ( GUIDED, 1 )
      do i = 1, ..., 12
         ....
      end do
```



## Parallel Loop Scheduling (4)

- **Affinity-based Scheduling**
  - Dynamic scheduling, but use locality of access together with load balancing as scheduling criterion
  - ”cache affinity”
- Example: UPC forall loop
  - ```
    shared float x[100], y[100], z[100];
    ...
    upc_forall ( i=0; i<100; i++; &x[i] )
       x[i] = y[i] + z[i];
    ```
    
  - Iteration i with assignment x[i] = y[i] + z[i] will be performed by the thread storing x[i], typically (i % THREADS)



DF21500 Multicore Computing

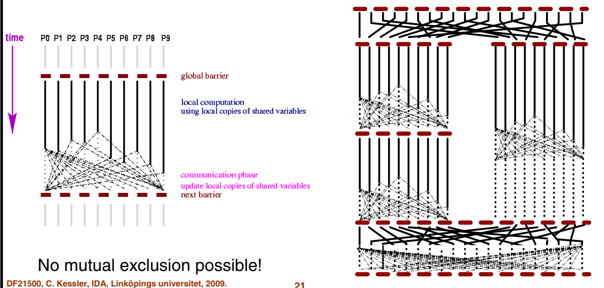
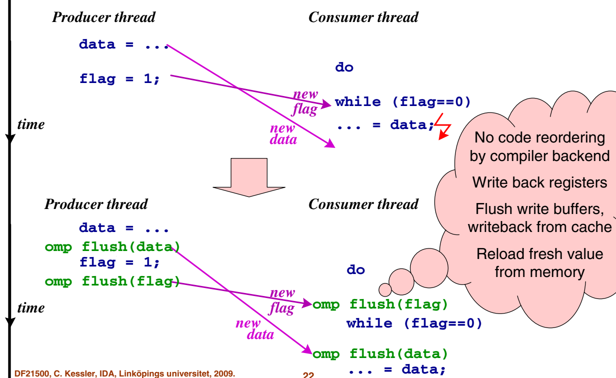
# Synchronization and Consistency

Christoph Kessler, IDA, Linköpings universitet, 2009.

## Natural Synchronization

inherent in the model, here: PRAM model / Fork language, also with data-parallel languages e.g. HPF



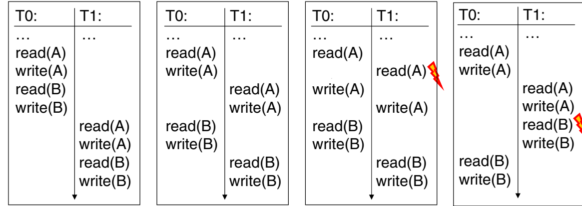
```
S:  a[$$] = a[$$] + a[$$+1];

        //  $$ in {0..p-1} is processor rank
```
```

## Futures

- A ***future call*** by a thread T1 starts a new thread T2 to calculate one or more values and allocates a ***future cell*** for each of them.
- T1 is passed a read-reference to each future cell and continues immediately.
- T2 is passed a write-reference to each future cell
- Such references can be passed on to other threads
- As (T2) computes results, it writes them to their future cells.
- When any thread touches a future cell via a read-reference, the read stalls until the value has been written.
- A future cell is written only once but can be read many times.
- Used e.g. in Tera-C [Callahan/Smith'90], ML+futures [Blelloch/Reid-Miller'97], StackThreads/MP [Taura et al.'99], Java (5+) Concurrency Package [SUN'04]



## Futures in Java (5+) Concurrency

```
class FibTask implements Callable<Integer>
{
  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask( int n )
  {
     arg = n;
  }
  public Integer call()
  {
     if (arg > 2) {
        Future<Integer> left  = exec.submit( new FibTask( arg – 1 ));
        Future<Integer> right = exec.submit( new FibTask( arg – 2 ));
        return  left.get() + right.get();
     } else
        return 1;
  }
}
```

A Callable<T> object represents a task that return a value of type T. Method call() (without parameter) returns the result.

Creates a thread pool to which tasks can be submitted, waited for, and killed.

Java thread-pools abstract from platform-specific thread management

spawn two tasks to compute fib(arg-1) and fib(arg-2), with handles left and right, respectively

get(): blocks until future value (arg) has been written or task is completed

See book, chapter 16 [Herlihy/ Shavit'08]



## Supersteps

- BSP model: Program executed in series of supersteps
- Nestable supersteps
  - PUB library [Bonorden et al.'99], NestStep [K.'99]




No mutual exclusion possible!



## Memory Fence / Flush






## Atomic Operations

- Atomic operations on a single memory word
  - SBPRAM / Fork mpadd() etc.
  - OpenMP `atomic` directive for simple updates (x++, x--)
  - test&set, fetch&add, cmp&swap, atomicswap ...
- Can be used to implement locks and semaphores
- Can be used to implement non-blocking data structures →



## Parallel Design Pattern:
### Using Spinlocks on Cache-based Multiprocessors

- Recall busy waiting at spinlocks:

```
// … lock initially 0 (unlocked)
while ( ! test_and_set( &lock ))
    ;
// … the critical section …
lock = 0;
```

- Test_and_set in a tight loop → high bus traffic on multiprocessor
  - Cache coherence mechanism must broadcast all writing accesses (incl. t&s) to lock immediately to all writing processors, to maintain a consistent view of lock's value
  - → contention
  - → degrades performance

**Solution 1:**

- Combine with ordinary read:

```
while ( ! test_and_set( &lock ))
   while ( lock )
       ;
// … the critical section …
```

- Most accesses to lock are now reads
  → less contention,
  as long as lock is not released.

**Solution 2:**

- ```
  while ( ! test_and_set( &lock ))
      do_nothing_for ( short_time );
  // … the critical section …
  ```
  ```

## Atomic Transactions, Serializability

Serialization (in some arbitrary order, dep. on scheduler) **implies** race-free-ness, but not vice versa.

Serialization is overly restrictive.

**Example**: Shared variables A, B; transactions T0, T1 e.g. { a=A=..A..; B=..B..a..; }




Serialized as T0→T1, e.g. using a mutex lock

Concurrent, but serializable as T0→T1 [SGG7 Sect. 6.9.4.1]

Incorrect – not serializable, not atomic

Incorrect – not serializable, not atomic



## Atomic Transactions

- For atomic computations on multiple shared memory words
- Abstracts from locking and mutual exclusion
  - coarse-grained locking does not scale
  - declarative rather than hardcoded atomicity
  - enables lock-free concurrent data structures
- Transaction either commits or fails
- Variant 1: atomic { ... } marks transaction
- Variant 2: special transactional instructions
  e.g. LT, LTX, ST; COMMIT; ABORT
- Speculate on atomicity of non-atomic execution
  - Software transactional memory
  - Hardware TM, implemented e.g. as extension of cache coherence protocols [Herlihy,Moss'93]



## Atomic Transactions Example: Lock-based vs. Transactional Map based Data Structure

```
class LockBasedMap
   implements Map
{
   Object mutex;
   Map m;

   LockBasedMap (Map m) {
      this.m = m;
      mutex = new Object();
   }

   public Object get() {
      synchronized (mutex) {
         return m.get();
      }
   }

   // other Map methods. . .
}
```

```
class AtomicMap
   implements Map
{
   Map m;

   AtomicMap (Map m) {
      this.m = m;
   }

   public Object get() {
      atomic {
         return m.get();
      }
   }

   // other Map methods. . .
}
```

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.



## Example: Thread-safe composite operation

- Move a value from one concurrent hash map to another
- Threads see each key occur in exactly one hash map at a time

```
void move (Object key) {
   synchronized (mutex ) {
      map2.put ( key, map1.remove(key));
   }
}
```

Requires (coarse-grain) locking (does not scale)

or rewrite hashmap for fine-grained locking (error-prone)

```
void move (Object key) {
   atomic (mutex ) {
      map2.put (key, map1.remove( key));
   }
}
```

Any 2 threads can work in parallel as long as different hash table buckets are accessed.

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.



## Software Transactional Memory

User code:

```
int foo (int arg)
{
   ...
   atomic
   {
      b = a + 5;
   }
   ...
}
```

Compiled code:

```
int foo (int arg)
{
   jmpbuf env;
   …
   do {
      if (setjmp(&env) == 0) {
         stmStart();
         temp = stmRead(&a);
         temp1 = temp + 5;
         stmWrite(&b, temp1);
         stmCommit();
         break;
      }
   } while (1);
   …
```

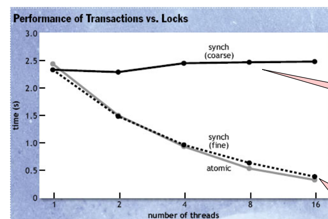
checkpoint current execution context for case of roll-back

Instrumented with calls to STM library functions. In case of abort, control returns to checkpointed context by a **longjmp**()

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.



## Transactions vs. Locks




Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007. © ACM

Times for a sequence of Java HashMap insert, delete, update operations, run on a 16-processor shared memory multiprocessor

Coarse-grained locking à la Java "synchronized" does not scale up

Fine-grained locking scales here equally well as atomic transactions, but is more low-level, susceptible to bugs leading to deadlocks or races

## Transactional Memory

- Old idea: Transaction concept comes from database systems
- Much hype in the last 5 years in computer architecture
  - Pros: Comfortable, no deadlocks, no races, possibly more scalable
  - But no breakthrough for STM yet – does not scale, high overheads
  - Interaction of transactional with non-transactional code (I/O etc.)?
- For known, frequently used data structures and operations:
  Fine-grained locking or non-blocking implementations look more promising
  → lecture on non-blocking synchronization
- Good introduction:

  A. Adl-Tabatabai, C. Kozyrakis, B. Saha:
  Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.
- More references:
  See course homepage – list of papers for presentation



DF21500 Multicore Computing

# Address space

Christoph Kessler, IDA,
Linköpings universitet, 2009.

## Partitioned Global Address Space

- Local memory modules and main (shared) memory are embedded into a global address space
  - Enables pointers across memory module boundaries
  - May still require non-uniform memory access
  - **Variant 1**: Mapping by MMU / OS.
    - Orthogonal to Virtual Memory
    - Both local and global addresses may be visible to the user
  - **Variant 2**: Mapping by language, compiler and/or run-time system
    - Often done data-structure wise (e.g., array distributions)
    - Remote accesses explicit or intercepted by run-time system
- Example for v1: Cell BE effective addresses for SPE local stores
- Example for v2: UPC distributed arrays



## Tuple space

- Linda [Carriero, Gelernter 1988]
- Tuple space
  - Associative memory storing data records
  - Physically distributed, logically shared
  - Atomic access to single entries: **put**, **get**, **read**, ...
  - Query entries by pattern matching, e.g.:
    **get** ( ”task”, &task_id, args, &argc, &producer_id, 2 );
    = atomic find&remove
- Can be used to coordinate processes
  - E.g., task pool for dynamic scheduling
  - E.g., producer-consumer interaction



DF21500 Multicore Computing

# Data Locality Control

Christoph Kessler, IDA,
Linköpings universitet, 2009.

## Locality Control in OpenMP?

- OpenMP has no constructs to control data locality
  - Designed for plain shared memory
  - But virtually all shared memory systems profit from access locality in some form
- Implementations usually apply the *first-touch placement policy*
  - Memory for a variable or object will be allocated on the node containing the thread accessing it for the first time.
  - Performs often better than random placement.
  - May be controlled by an option or by directives on some platforms

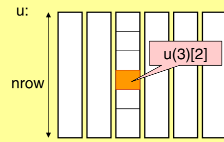
## Co-Arrays

- Co-Array Fortran [Numrich / Raid '98]
- **Co-Arrays**
  - Distributed shared arrays with a **co-array dimension** spanning the processors in a SPMD environment
  - arr(j)[k] – addresses processor k's copy of arr(j)
  - x(:) = y(:)[q]



## Co-Array Fortran Example

```
subroutine laplace ( nrow, ncol, u )
  integer, intent(in)   :: nrow, ncol
  real, intent(inout)   :: u(nrow) [*]
  real                  :: new_u(nrow)
  integer               :: i, me, left, right

  new_u(1) = u(nrow) + u(2)
  new_u(nrow) = u(1) + u(nrow-1)
  new_u(2:nrow-1) = u(1:nrow-2) + u(3:nrow)
  me = this_image(u)   ! Returns the co-subscript within u
                       ! that refers to the current image
  left = me-1;
  if (me == i) left = ncol
  right = me + i;
  if (me == ncol) right = 1
  call sync_all( (/left,right/) )  ! Wait if left and right have not already reached here
  new_u(1:nrow) = new_u(1:nrow) + u(1:nrow) [left] + u(1:nrow) [right]
  call sync_all( (/left,right/) )
  u(1:nrow) = new_u(1:nrow) - 4.0 * u(1:nrow)
end subroutine laplace
```




Source: Numrich, Reid: *Co-Array Fortran for parallel programming.* Technical report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, 1998.



## Virtual topologies (MPI)

Example: arrange 12 processors in 3x4 grid:

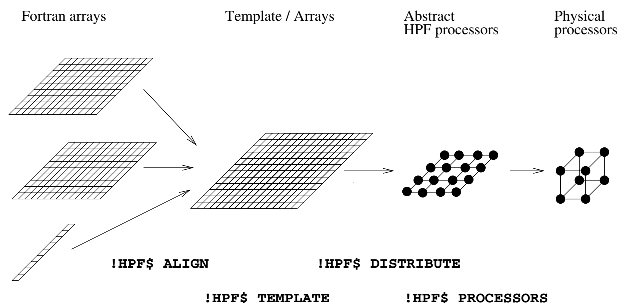
```
int dims[2], coo[2], period[2], src, dest;
period[0]=period[1]=0;  // 0=grid, !0=torus
reorder=0; // 0=use ranks in communicator,
           // !0=MPI uses hardware topology
dims[0] = 3; // extents of a virtual
dims[1] = 4; // 3X4 processor grid

// create virtual 2D grid topology:
MPI_Cart_create( comm, 2, dims, period,
                 reorder, &comm2 );

// get my coordinates in 2D grid:
MPI_Cart_coords( comm2, myrank, 2, coo );

// get rank of my grid neighbor in dim. 0
MPI_Cart_shift( comm2, 0, +1, // to south,
                &src, &dest); // from south
...
```

| (0,0) | (0,1) | (0,2) | (0,3) |
|---|---|---|---|
| (1,0) | (1,1) | (1,2) | (1,3) |
| (2,0) | (2,1) | (2,2) | (2,3) |

```
...
coo[0]=i; coo[1]=j;

// convert cartesian coordinates
// (i,j) to rank r:
MPI_Cart_rank(comm, coo, &r);

// and vice versa:
MPI_Cart_coords(comm,r,2,coo);
```



## HPF Mapping Control: Alignment, Distribution, Virtual Processor Topology






## Declaring Data Distribution in HPF (1)

```
!HPF$ PROCESSORS P(4)
REAL, DIMENSION (23) :: A
```

P(1) P(2) P(3) P(4)

```
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A
```

A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

```
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A
```

A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

```
!HPF$ DISTRIBUTE (BLOCK(7)) ONTO P :: A
```

A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

```
!HPF$ DISTRIBUTE (CYCLIC(3)) ONTO P :: A
```

A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23



## Declaring Data Distribution in HPF (2)

```
!HPF$ PROCESSORS P(4)
REAL, DIMENSION (8,8) :: A
```

P(1) P(2) P(3) P(4)

```
!HPF$ DISTRIBUTE (BLOCK, *) ONTO P :: A
```

A

```
!HPF$ DISTRIBUTE (CYCLIC,*) ONTO P :: A
```

A

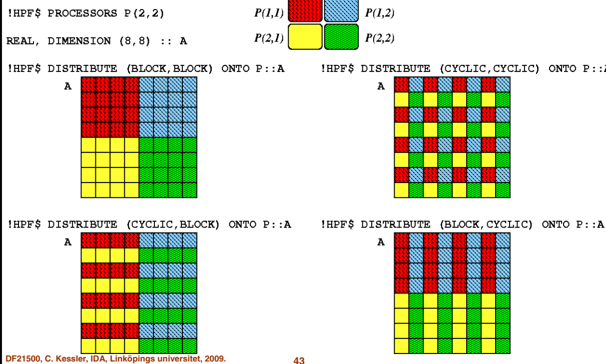
```
!HPF$ DISTRIBUTE (*, BLOCK) ONTO P :: A
```

A

```
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO P :: A
```

A

## Declaring Data Distribution in HPF (3)



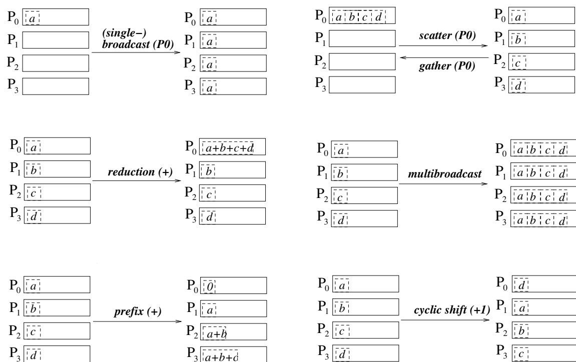





# Communication

Christoph Kessler, IDA,
Linköpings universitet, 2009.

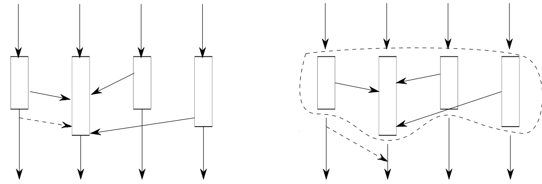
## Collective Communication






## Encapsulation of communication context

- Example: MPI Communicator
- Needed for parallel components







# Wrap-up: Parallel Programming Models and Languages for Multicore

**Current State and Trends**

Christoph Kessler, IDA,
Linköpings universitet, 2009.

## Which Parallel Programming Model?

**Several competing parallel programming models (and languages)**

- **Memory and Communication**
  - Shared memory (OpenMP, pthreads, Cilk)
  - Partitioned Global Address Space (UPC, NestStep, X10)
  - Local Address Space, Message passing (MPI, BSPlib, Cell SDK)
- **Parallel Control**: MIMD vs. Data-parallel (HPF, F95+)
- **Memory Model**
  - Strict, sequential, relaxed consistency models
- **Implicit (built-in) Synchronization**
  - Unstructured, hard-coded (locks/semaphores, message passing)
  - Directed acyclic task graph / Futures / Dataflow (Cilk, OpenMP-3.0)
  - Atomic transactions
  - Event-driven
  - Supersteps (BSP, NestStep)
  - Instruction-level synchronicity (PRAM, SIMD)

## What will the next generation general-purpose (parallel) programming language look like?

- **Uniform for Multicore, SMP Servers and Clusters**
  but variety of languages for different problem domains or special platforms
- **Hybrid and nested parallelism** at multiple levels of abstraction/efficiency
  - Task parallelism (tasks, threads, ...)
  - Data parallelism
  - Pipelining
  - Need a compositional coordination mechanism to manage complexity
    e.g. parallel components; algorithmic skeletons (→ separate lecture)
- **(Partitioned) Global address space**
- **Control of data locality**
  - Data layout, Affinity scheduling, Prefetching, Cache control, Buffering
- **Parallel libraries and data structures**
  e.g. skeletons; distributed arrays; lock-free shared data structures
- **Current design approaches:**
  - [Fortress (Sun)], Chapel (Cray), X10 (IBM) for HPC domain
  - OpenMP 3.0 (2008) – still no good locality control,
    restructuring needed for good performance



## Summary

- Trend towards Many-Core for the foreseeable future
  - Unavoidable technical reasons
  - Both HPC, desktop, DSP and embedded domain
- Programmers will need to expose explicit, massive parallelism
- Need new programming models, languages, compilers,
  tools, libraries and support for components
  - usable by both experienced and novice programmers
  - support for semiautomatic transition from sequential code?
  - Multi-model environments
    - Simple high-level general-purpose model and language
      for convenient programming and portability
    - Advanced model layer(s) for special domains and performance-tuning
  - Component framework for multi-language integration